Cross-chain Trading Network Using Smart Contracts

by
Quang Tran

A Thesis submitted to the Department of Computer Science,

College of Natural Sciences and Mathematics

in partial fulfillment of the requirements for the degree of

Master of Science
in Computer Science

Chair of Committee: Weidong Larry Shi

Committee Member: Lei Xu

Committee Member: Panruo Wu

University of Houston
May 2020

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor, Dr. Weidong Larry Shi, for his patience, guidance, kindness, support, and instructions during my work on this thesis, especially for his motivation so that I can complete my Master's Degree.

I am very grateful to Dr. Lei Xu, for serving on my thesis committee and for being my reviewer to correct my mistake in many projects.

I wish to thank Dr. Panruo Wu for serving on my committee and for reviewing, providing advice on my thesis.

Last but not least, I also want to thank other students in Dr. Shi's group, Keshav, Kelvin, and especially Glenn, for suggestions and comments.

ABSTRACT

The blockchain, a secured and distributed database design, has been introduced and launched for ten years. Since then, a significant number of cryptocurrencies and financial trading markets using digital assets have also been introduced over time. Despite the fluctuation in the exchange rate, blockchains and its cryptocurrencies have no remarkable change overall. They still follow the old-traditional trading mechanism, which exchanges between cryptocurrencies and fiat currencies. Although few projects are emerging to enlarge a blockchain's trading usage, i.e., cross-chain liquidity, it still limits itself on swapping one type of cryptocoin with another type. Beyond supporting swapping coins, the blockchain could be applied and extended to become a trade-payment network that connects multiple solitary blockchain-based platforms. In this thesis, I propose a heuristic cross-chain trading system that leverages the blockchain technology to build a fair and border-less trading network across multiple decentralized blockchains by taking advantage of smart contracts and ERC-20 protocol. Last sentence, I provide a concrete example and design of a cross-chain trading network between three popular systems, which are Bitcoin, Lightning Network Daemon, and Ethereum.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 Introduction

The concept of blockchain was first introduced as the backbone of Bitcoin, a cryptocurrency without a centralized component [1]. Intuitively, a blockchain is a distributed database that is securely managed by multiple parties. All storage devices for the database are not managed, maintained by central authorities, or servers. Instead, a growing list of records, called blocks, of all transactions or digital events that have been executed and shared among participating parties, links to each other to maintain the trust in a tamper-proof distributed database. Since the creation of Bitcoin, cryptocurrencies have gained the popularity that starts a flourishing period of blockchain technology, and financial trading markets using digital coins.

One of the most significant extensions of the original blockchain concept is a smart contract [2, 3, 4], which enables more complex functions than simple transactions to be enforced with a blockchain. Specifically, a smart contract is a piece of computer programming code that is replicated across clients in a blockchain network. As the name suggests, a smart contract expresses the agreement between involved parties, and the blockchain facilitates the verification and enforcement of the agreement without relying on any trusted third party. Several popular blockchain platforms support smart contracts such as Ethereum, EOS, Hyperledger Fabric, and Stellar, which can be utilized for solving a range of business challenges [5, 6, 7].

During the past ten years, the exchange rates among different cryptocurrencies have fluctuated significantly. However, blockchain-based platforms, in general, have no significant development from end-users' perspective except improvements such as privacy enhancement [8]. Blockchains and their supporting digital assets still follow an old and traditional trading mechanism that is cryptocurrencies and fiat currencies exchange. Recently, several projects have emerged to enlarge the blockchain's usage in trading, such as swapping one type of cryptocoin to another. However, like the current financial trading system, trading is not limited only to exchanging one currency for another. Blockchain can be applied as a backbone of various trades using digital assets.

In this thesis, I propose a novel cross-chain trading system. This network can leverage the

use of blockchain technology to aggregate existing blockchain-based platforms to create a scalable trading network by taking advantage of smart contracts and ERC-20, which is a de facto standard of token implementation. To provide heuristic schemes as a demonstration, proposing protocols are implemented on three popular decentralized blockchain ledgers which are Bitcoin, Lightning Network Daemon, and Ethereum.

The contributions in this paper include:

- Leveraging the use of blockchain as a backbone model of the cross-chain digital asset trade;
- Proposing protocols that can aggregate multiple solitary blockchain platforms;
- Proposing a trading network across multiple decentralized blockchains; and
- Suggesting future work and additional research on improving performance.

The rest of the thesis is organized as follows. In section 2, I briefly cover the history of blockchain technology as well as some basic designs of Bitcoin, Lightning Network, and Ethereum. Then, I provide some related works and motivation for me to come up with this thesis. In section 4, I propose protocols and a detail scheme to build a cross-chain trading network that allows clients from other side-chains, Bitcoin and Lightning Network, to be able to join a main-chain Ethereum. Next, I also discuss threat models and analyze the securities of the proposing protocol. In the end, I show how I set up an experiment and also provide capturing screenshots of the experiment's results before giving a conclusion.

2 Background on The Blockchain

2.1 Blockchain - A Brief History

In this section, I briefly cover the history of blockchain technology. Centralization is a common and pervasive form of regulating and controlling by a single authority or governance. People sometimes trust or lack trust of central authorities, i.e., banks, organizations, and governments, to regulate or to maintain order, trust, and fairness during operation. There have been many examples that show the vulnerability of these authorities in operation, e.g., the financial crisis of 2007-2008, the crisis in Venezuela, and leaked customers' bank information. Thus, centralization is not always a suitable mechanism. Also, a centralized model allows for the monopolization of power. It tends to serve the interests of a group of people rather than the community as a whole. These two reasons motivate for giving birth to blockchain technology.

Many of the technologies, which people currently take for granted, undergo small and silent transformations. For instance, computers and the internet have changed human life. They have been around for a few decades. We are now in the midst of another quiet innovation that is called a blockchain. Even today, many people believe Bitcoin and the blockchain are the same. However, they are not the same as being described later. As early as 1991, the fundamental idea behind blockchain technology was first described when researchers, Stuart Haber and W. Scott Stornetta, introduced a computationally practical solution to implement a system where document timestamps could not be tampered with [9]. A chain of blocks, applying cryptography to secure, was implemented to store the time-stamped documents. One year later, Merkle Trees were incorporated into the design, which made it more efficient by allowing several documents to be collected into one block.

The blockchain is a growing list of records, called blocks, that are linked using cryptography. Each block contains a cryptographic hash of the previous block, a timestamp, and data (see Figure 2.1). By design, a blockchain can be applied as a distributed database that keeps record transactions between several parties efficiently, securely since it is resistant to alteration of data

[10, 11, 12, 13].

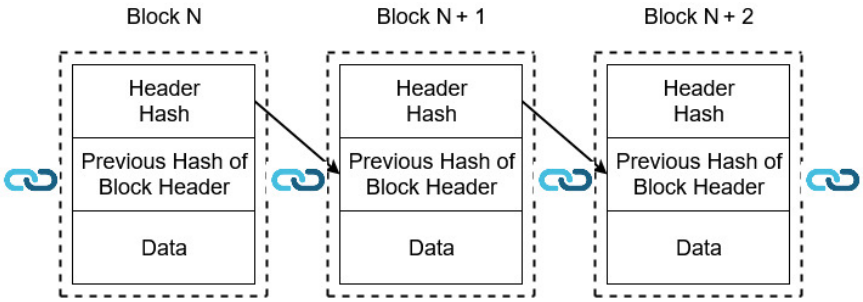

Figure 2.1: Simple Block Structure

During the past ten years, the blockchain innovation can be summarized as:

- The introduction of Bitcoin has started an era of blockchain and cryptocurrencies,
- A thriving release of many other digital assets along with the new, distinct, and efficient design of blockchain platforms. Major financial institutions and hi-tech corporations are also following a new trend of the decentralized record-keeping model,
- The implementation of smart contract protocol is considered as the second generation of blockchain technology. Smart contracts provide the performance of trustworthy transactions without requiring the involvement of third parties,
- Current generation blockchains are secured by Proof-of-Work (PoW), which requires a group of miners contributing their effort in solving mathematical puzzles. This method limits the scalability of blockchain's performance and usage. Hence, a new consensus algorithm, Proof-of-Stake (PoS), has been proposed to solve the issue,
- Not just limited to exchange and finance, blockchain technology can now be applied to many different fields of services such as supply chain and IoT.

2.2 Bitcoin

After the financial crisis in 2008, the community's faith in financial and banking systems went down. An unknown founder, using the name Satoshi Nakamoto, utilized the cryptography and blockchain technology to create Bitcoin. In early 2009, Bitcoin's ledger was introduced and launched

as the first model of a decentralized blockchain and digital currency. It has become a global decentralized financial system and empowers people to have full control over their finances. Without a central bank and a single administrator, a transaction among clients can be executed and sent using a peer-to-peer Bitcoin network. Even though Bitcoin lacks the ability to be extended or replaced with a traditional currency system, it still has the capability of becoming a digital store-of-value with the protection of anonymity, decentralization, and motivation of the users' power over the control of global finance. The Bitcoin's purpose has never changed since the very beginning of its release.

In Bitcoin, a block size is initially hard-coded in size of 1MB to prevent spam transactions. However, Bitcoin's scalability becomes an issue when it is getting more attractive and well-known. There is an expected block generation time that is approximately 10 minutes. The expected block time is set as a constant value so that miners cannot impact the security of the network by adding more computational power. This implementation also helps to minimize the impact of communication delay of peer-to-peer in a network. When a block is successfully mined, transactions are added into a block by miners. However, the number of transactions that can be added is limited by the block size. In addition to a long block generation time, block size also contributes to limit the Bitcoin's scalability. By design, Bitcoin cannot fit a high volume of transactions to apply to a financial system widely. Thus, Segregated Witness (SegWit) was introduced to limit the size of each transaction. Just as there are multiple versions of the Internet Protocol, the Bitcoin address also has multiple formats.

- Legacy address format (P2PKH): P2PKH stands for Pay-to-Pubkey Hash. This format is an original version of Bitcoin's address and still works to this day. If a client uses a legacy address format, it can sign a message and can be easily verified by using traditional cryptography (ECDSA).
- P2SH address format: this is one type of SegWit formats. Its structure is similar to the P2PKH format. But, the P2SH format enables more elaborate functionality than a legacy format. A P2SH script function most commonly uses for multi-sig addresses, which require

multiple digital signatures to authorize the transaction. This format is widely supported and can be used to transfer funds to both a legacy address and a Bech32 address format. However, the P2SH address format cannot sign and verify a message publicly as a P2PKH format.

- Bech32 address format: Bech32 is a native SegWit address format. It looks distinctly different from the P2-style formats. Despite being supported by a majority of software and hardware wallets, a minority of exchanges adopts the Bech32 address format. Similar to a P2SH format, the Bech32 format cannot sign and verify a message by the standard cryptography.

In Bitcoin, a blockchain's record-keeping model adopts a method, called UTXO (Unspent Transaction Output). Each transaction spends an output from previous transactions and then generates new outputs that can be spent by future transactions (see Figure 2.2). Each transaction is represented by an address, i.e., P2SH address format. When new outputs are created, new addresses are also generated. All of the unspent transactions are kept in each node. A user's node takes responsibility for keeping track of the unspent transactions associated with the addresses owned by a user. Intuitively, the balance of a node is a total sum of the amounts of unspent transactions. Furthermore, since new outputs generate new distinct addresses, it is difficult to detect the transaction's sender and the total balance of a specific account even though transactions are publicly announced.

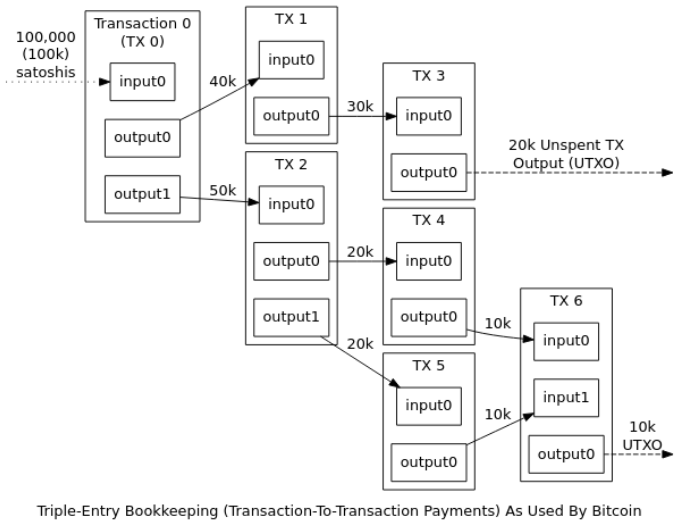

Figure 2.2: Unspent Transaction Output - UTXO

Image from [1]

2.3 Lightning Network Daemon

Ten years after Bitcoin was first released, scalability still has fallen behind current demanding needs. The high latency of peer-to-peer communication and low throughput of consensus protocol are two fundamental barriers to their wider adoption. Although the capability of processing around 7 transactions per second (TPS) was enough at the very beginning of the Bitcoin launch, the existing system ultimately falls behind current demanding needs. It is unable to compete or to alternate the existing payment systems that can be able to process a large number of transactions (Visa achieved 47,000 TPS) [14]. As a distributed database, the Bitcoin blockchain holds great promise that protects privacy and security. However, it lacks, as a payment system, the capability to cover the world's commerce. Bitcoin's community has come up with several proposals that can improve Bitcoin's scalability over the year. Currently, there is one proposal in the process of launching and testing that might work to solve the problem[15, 16]. This platform is called the Lightning Network Daemon[2]

[1] https://bitcoin.org/
[2] https://lightning.network/

Improving the time to finalize a transaction and the number of transactions, which can be handled per unit of time, is perhaps the most crucial need of distributed public blockchains. The Lightning Network is a system of instant, high-volume micropayments in a peer-to-peer fashion. Its payment protocol is a second layer that operates on top of blockchain-based cryptocurrency, Bitcoin. Bitcoin's ledger provides an advanced scripting language that allows users to program instructions on protecting, releasing, and transferring funds. Taking advantage of Bitcoin's built-in scripting, the Lightning Network solves Bitcoin's current problems by the implementation of a multi-signature scripting smart contract (HTLC) [17, 18], which is possible to create a secure mesh network of participants in trading with high volume and high speed. Not only does it provide better scalability, but it also enables users to perform off-chain payments and with low or negligible fees.

Like a traditional process of consensus, transactions among parties are required to broadcast and to be verified (a.k.a. a mining process) by other nodes in a network before they are finalized and added into blocks. In the case of recurring transactions among two parties, it must be gone through the same process, which is cumbersome and decreases the performance of the blockchain protocol. "If a tree falls in the forest and no one is around to hear it, does it make sound?"[14]. Similarly, if only two participants care about recurring transactions, other nodes in the network do not necessarily need to know about these transactions. Instead, only essential information is preferred to be added to the blockchain. Thus, Lightning Network proposes an off-chain payment channel, which essentially helps to cut-off a large number of recurring transactions. Two parties create a multi-signature channel, requiring both participants to sign off, in which funds are placed. By having many of these payment channels, it forms up a mesh network of connections among participants. Furthermore, Lighting Network also implements a method that helps to find a path across networks, similar to routing packets on the internet, to transfer funds to a recipient in the case of trading between two participants without the creation of a new channel. This method is called a multi-hop payment (Figure 2.3).

A payment channel is represented as an entry on the Bitcoin's ledger. Thus, it requires a transaction of a channel creation to be finalized on-chain, Bitcoin. Once a payment channel is

created, transactions among nodes in the network can be done off-chain by using a scripting Hash Time-Locked Contract (HTLC). By design, the channel is a multi-signature address between only two stakeholders. In releasing funds of the channel, signatures from two stakeholders must be provided. Updating new owner of transferring funds can be processed off-chain, within the Lightning Network, instantly and compatibly instead of requiring to reach consensus in the Bitcoin's chain. With each off-chain transaction, a secret is generated by each party with each new version of the balance sheets. A hash function is applied to this secret to generate a unique digital fingerprint that is then given to the other party to update and to prove the new ownership (see Figure 2.4). This implementation permits the financial relationships between two or multiple participants in the mesh network can be trustlessly deferred to broadcast recurring transactions on-chain to a later date. The payment channel can remain open indefinitely unless at least one of the two stakeholders requests to close the channel. Once the channel is closed, a final version of balance sheets and proof of ownership must then be broadcast and finalized in the Bitcoin's chain.

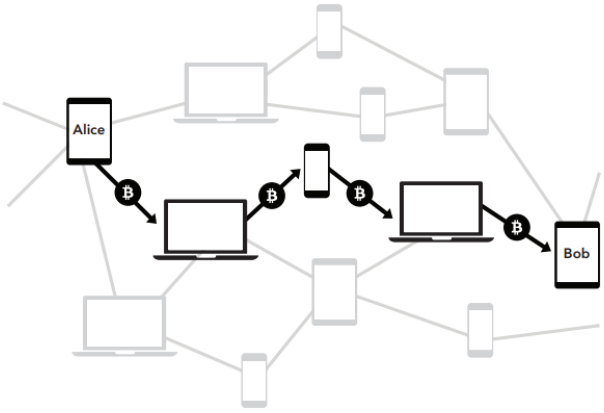

Figure 2.3: Multi-hop payment between two participants

Image from [3]

[3]https://lightning.network/

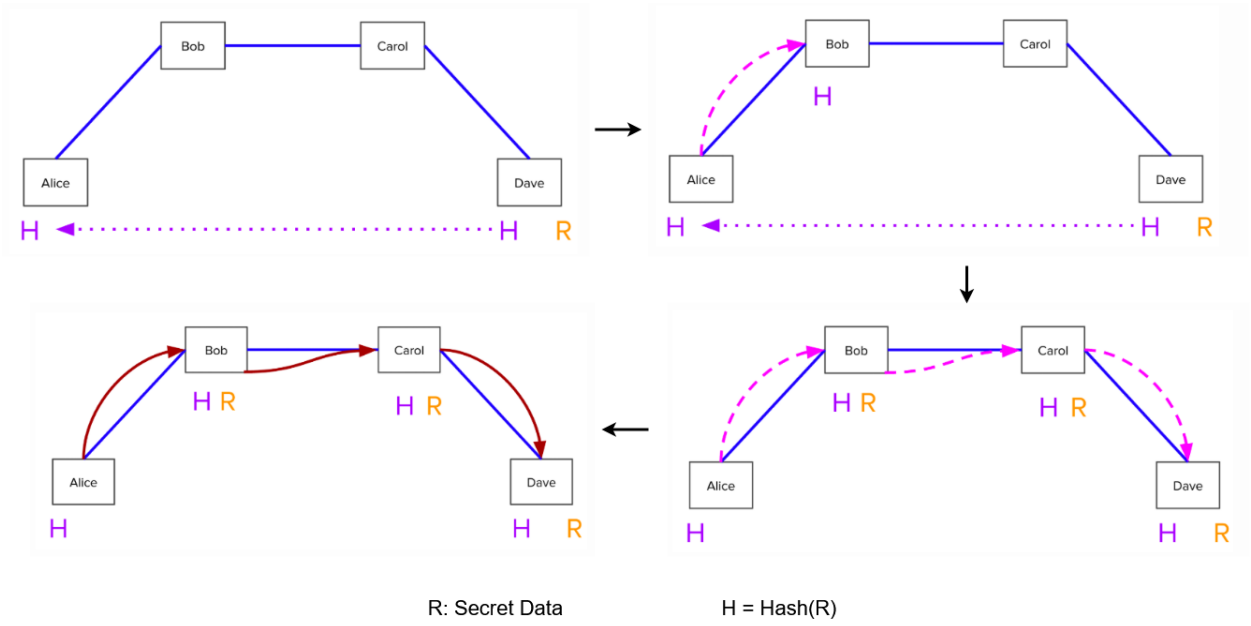

Figure 2.4: HTLC in case of multi-hop payment

Image from [4]

In general, the Lightning Network provides a potential solution to solve the scalable problem of Bitcoin, in which it offers:

- A mesh network of off-chain payment channels in the peer-to-peer fashion;
- Fast and instant off-chain transactions among nodes;
- Micro-payments which allows a lower limit of transferring coins per transaction; and
- Low cost

2.4 Ethereum

Ethereum, on the other hand, is not only a payment system. When Ethereum is mentioned, people think about smart contracts. This decentralized platform was launched in 2015 and is widely known as the second generation of blockchain. Ethereum Founder, Vitalik Buterin, believes that blockchain technology has more utility than just being a simple payment-service provider.

[4] https://lightning.network/

Developers can create many real-world applications on top of it. Hence, a smart contract, an extended protocol, was introduced to digitally facilitate, verify, or enforce negotiation or performance of multiple involved parties. Smart contracts are computer programming running on top of a blockchain-based platform, and containing a set of pre-defined regulations among participants of a contract who agree to deal with each other. Smart contracts provide the performance of trustworthy transactions without the involvement of third parties. Smart contracts are stored on a blockchain and are replicated across clients in a blockchain network. Thus, the security and immutability of the smart contract still fit the ideal of blockchain technology.

To create a protocol for smart contracts that offer beneficial and efficient interactions between participants in the network, Ethereum builds a Turing-complete machine, called the Ethereum Virtual Machine or EVM, as the heart of its blockchain platform. Developers can create their applications, which run on EVM using any friendly programming languages, to create their arbitrary rules of ownership, transaction formats, and state transition functions. In terms of a smart contract, Ethereum can sometimes be considered as a "world computer."

There are two types of record-keeping models that are widely used in decentralized blockchain today. One of them, the UTXO model, as described in previous sections, is adopted by Bitcoin. Ethereum is currently using an Account/Balance model. The model keeps track of the total balance of each account as a global state. When a spending transaction is executed, the total balance of an account is checked to guarantee that it is greater than or equal to a spending amount. For the benefit of developing complex smart contracts, especially the contracts that require state information and involve multiple parties, the Account/Balance model brings simplicity and efficiency. Each transaction needs only to validate that the sender has enough balance to pay for it.

In Ethereum, a user is called a client. A client that runs mining on Ethereum blockchain is called a miner/a node. A client can send Ether, which is the official currency of Ethereum, to a smart contract or other clients by issuing a transaction. To protect its platform and transactions among clients, the Ethereum security model is based on a standard cryptography elliptic curve ecp256k1 (ECDSA) to sign and to validate transactions. A private key, required to be 256-bit long,

is associated with one node client. A corresponding public key is derived using the group operation of elliptic curve cryptography. In Ethereum, a node is associated with an address that is a 160-bit value and is defined as the rightmost 160-bits of the Keccak hash of the corresponding ECDSA public key.

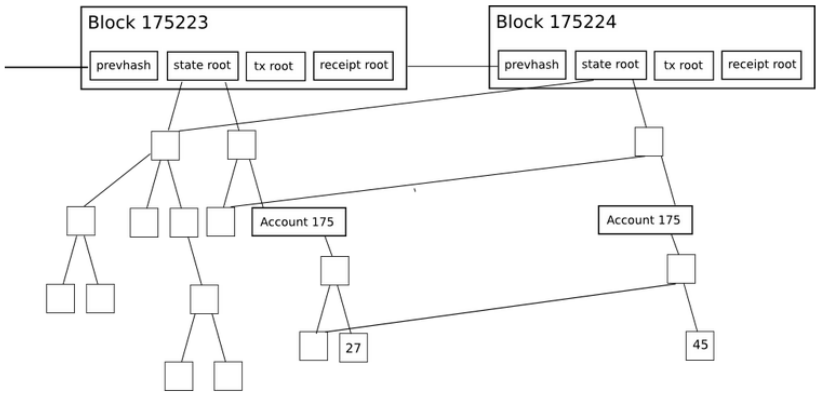

Figure 2.5: Merkle Proof in Ethereum

Image from [5]

Furthermore, Merkle Tree (Figure 2.5) is a fundamental feature that makes blockchains achieve its purpose as decentralized databases. A Merkle tree is a way of hashing a large number of data. Data are split into buckets, then taking the hash of each bucket and repeating until the total number of hashes remaining becomes only one, a root hash. Unlike Bitcoin, every block header in Ethereum contains three Merkle trees for three separating kinds of objects - transactions, receipts, and state. By design, the total balance of an arbitrary account and the sender/receiver of a transaction in the Ethereum blockchain can be easily verified.

[5] https://blog.ethereum.org

3 Related Works of Cross-chain Trading

3.1 Related Works

It has been a decade since the first time that blockchain technology was introduced and launched by Bitcoin. A massive number of cryptocurrencies, around 3,000 cryptocoins, according to CoinMarketCap[6], have been introduced over time, and yet begun a flourishing period of blockchain technology and financial trading markets (e.g., Coinbase, Binance, Bitfinex) using cryptocoins along with a high volume of trade per day. However, this trading system has some limitations since it is sometimes required to trade directly or swap with real fiat currency (e.g., USD and EURO). For example, a payer, who has Bitcoin coins, needs to pay for a payee that accepts payment of Ether only. In this case, a payer has to sell Bitcoin, then buy a corresponding amount of Ether along with paying a transaction or trading fee through an exchange market. Furthermore, the current trading system lacks providing flexible use of cryptocoins for daily activities.

With the need for flexibility in transactions, the idea of cross-chain swapping coins gains more attention in recent years. An atomic swap is one of a solution that enables the exchange from one type of cryptocoins to another one without the existence of a trusted third party or a centralized exchange market. Liquality [19], an atomic swap platform between Bitcoin, DAI, and Ether, has been launched in late 2018. This project provides a web application that allows users to swap assets across supported blockchains in a peer-to-peer fashion. Its atomic swap is based on the use of Hash Time-Locked Contracts (HTLCs) [14, 17, 18], which is a time-bound conditional escrows contracts to minimize risk and lay the foundation for peer-to-peer value transfer adopted by Lightning Network Daemon

Another platform, Kyber Network[7] [20], is an on-chain liquidity protocol that provides swapping cryptocoins service from a wide range of reserves. This platform is implemented as a set of smart contracts on decentralized blockchain platforms that support a smart contract protocol. Kyber

[6]Available at: https://coinmarketcap.com/all/views/all/
[7]https://kyber.network/

Network requires a variety of Reserves that refers to anyone/foundations who wish to provide liquidity, to support swapping of multiple cryptocoins. Its infrastructure is designed and relied on the practical use of smart contracts and decentralized applications (DApp) [21, 22]. They design smart contracts, called Kyber Core Contracts, that take responsibility as HTLCs. In case of swapping coins, a requester transfers his/her coins into Kyber Core Contracts with a request of swapping to another supported coins. Then, it triggers an event to query all offering exchange rates from a list of reserved-contracts to find the best deal. Once it is found, the core contract does swapping coins by sending requesting coins to a requester while transferring input swapping coins to a chosen reserved-contract.

3.2 Motivation

An idea of cross-chain swapping has gained attention from the cryptocurrency community in recent years. However, cryptocurrency and exchange networks have not ultimately aggregated to leverage the use of blockchain technology. Just as the current financial trading systems in the world today, trading is not limited to exchanging one currency for another. It is the purchase and sale of goods in the form of cash payments through supporting fiat currencies. This incentive, as well as taking fundamental ideas from other projects [23, 24], and practical design ideas of Libra blockchain [25], I propose a heuristic scheme that leverages the use of blockchain and its related cryptocurrencies to promote a scalable trading environment across multiple blockchain-based platforms. Table 3.1 compares the cross-chain trading system proposed in this work with related works.

Scheme	Supporting coins	Scalability	Service
Liquality	Bitcoin, Ether, DAI	Limited supporting platforms	atomic swap
Kyber Network	Multiple coins (>20)	Compatibility with smart contracts	atomic swap
This paper	Multiple coins	Multiple platforms w/o smart contracts compatibility	Trade, Payment, Cashout

Table 3.1: The Cross-chain trading scheme in comparison with other networks.

4 Cross-chain Trading Network Scheme

In this section, I present the design of the cross-chain trading network scheme. The proposing network is an additional layer protocol running on top of supporting blockchain platforms. It does not affect the consensus or create a fork-chain on supporting platforms. The cross-chain trading scheme involves at least two public blockchain-based cryptocurrencies. One of them works as the main chain, which supports smart contract to process the trading. In the following, I use three popular public blockchain cryptocurrency systems, Bitcoin, Lightning Network, and Ethereum, to demonstrate the concept and design. The method can be easily extended to more blockchain-based cryptocurrencies and supports more complex trading requests and scenarios.

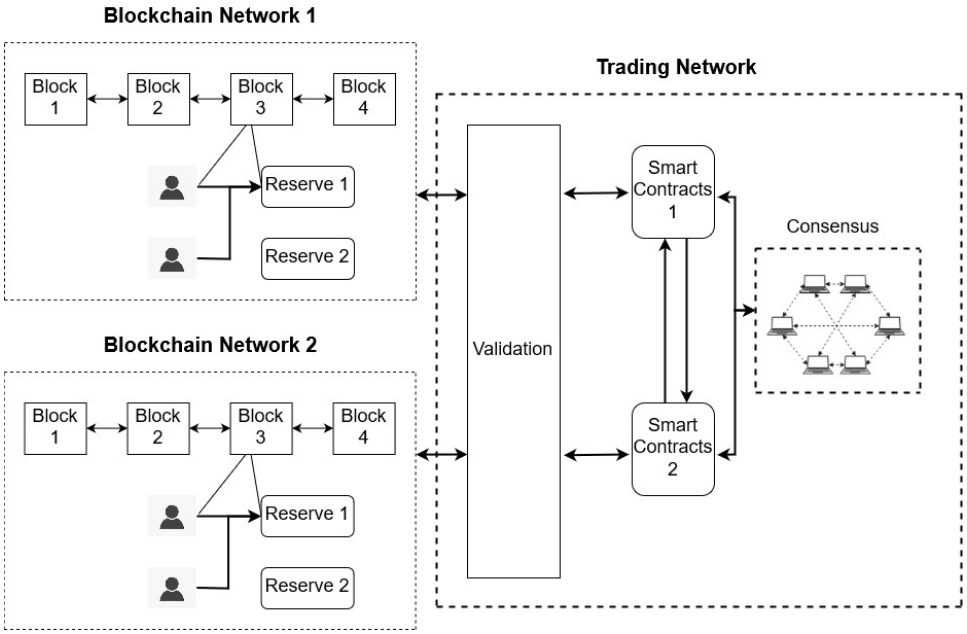

Figure 4.6: High-level architecture of the trading network.

In this thesis, I use Ethereum as the main chain as it supports EVM for smart contract execution, and I call it *Virtual Main-chain* ($\mathcal{VMC}$). However, a $\mathcal{VMC}$ can be applied to any public blockchain with EVM support. Bitcoin work as the other blockchain system that participates in the cross-chain trading network. This blockchain called *Physical Side-chain* ($\mathcal{PSC}$). A client that wants to use the trading service needs to have a node on each blockchain. In the case of another blockchain

platform, i.e., Lightning Network, it acts as another $\mathcal{PSC}$ with respect to the $\mathcal{VMC}$.

A smart contract cannot automatically verify transactions on the physical side-chains ($\mathcal{PSC}$s). Instead, I propose impartial witness roles, *Founders* and *Validators*, to verify side-chain transactions. Any client that is approved by the Founders can be a *Validator*. Also, the *Validators* are required to have nodes on both sides, $\mathcal{PSC}$ and $\mathcal{VMC}$. In this thesis, smart contracts are designed as a record of notary service. A smart contract records a list of *Validators* so that they can be verified publicly and easily. By design, communication protocols, as being described later, should be created to guarantee all processes, requirements must be met so that a cross-chain trading network can be run smoothly.

A cross-chain trading scheme can be broken down with four functional components:

- Wallet Initialization: this process requires cross-chain activities. A wallet, in this case, is a smart contract. A couple of processes and requirements must be met before receiving a number of tokens.
- Sending/Receiving Tokens: this feature can be executed as an in-chain process. Sending and Receiving Tokens are executed by smart contracts within $\mathcal{VMC}$. Client-to-Client transfer of tokens can be applied within a group or across groups.
- Adding Tokens: this feature is a cross-chain procedure. Clients are required to transfer their funds in requesting to add a number of Tokens.
- Cashing out: this is also a cross-chain procedure. The client submits a request to convert their Tokens back to supporting side-chain coins, i.e., Token $\rightarrow$ BTC.

4.1 Pre-defined Smart Contracts

In the cross-chain trading scheme, smart contracts hold fundamental roles, e.g., supporting trading features among clients, keeping track of a record of cross-chain verification, and enforcement to guarantee a fair and trustworthy trade among anonymous clients. To achieve the expected purpose, I propose four smart contracts that act as the backbone of the cross-chain trading network.

Token Contract: A *Token Contract* (Algorithm 1) is a Genesis smart contract, a contract

defined by the trading system and included in the block of $\mathcal{VMC}$. *Token Contract* keeps a record of founders, groups, user's balance, and transactions along with some utilizing functionalities, e.g., mint/burn Tokens, and getBalance. At the very beginning of launching the cross-chain trading network, the Founder deploys the *Token Contract* to obtain a contract's address ($\mathcal{T}$). A group that belongs to the cross-chain trading network must link its contract to $\mathcal{T}$.

Algorithm 1 Token Contract - A Genesis Contract

- Founder deploys a Token contract to obtain $\mathcal{T}$
- Token Contract contains:
+ Name: a name of Token coins
+ Symbol: a symbol of Token coins
+ Total Supply: a total number of Token being released
+ Used: a total number of Token being mint
+ map[address][boolean]Founders: a record list of founders
+ map[address][boolean]Groups: a record list of groups have joined a network
+ map[address][uint256]BalanceOf: a record list of balance of each wallet
+ Public map[uint256]TxInfo: a record list of in-chain transactions
+ $mint(\Delta, \mathcal{W})$: increase user's wallet address ($\mathcal{W}$) a number of Δ Tokens
+ $burn(\Delta, \mathcal{W})$: decrease user's wallet address ($\mathcal{W}$) a number of Δ Tokens
+ $getBalance(\mathcal{W})$: return a total balance of wallet's address ($\mathcal{W}$)
+ $remove(\mathcal{W})$: set the balance of a wallet ($\mathcal{W}$) back to 0 and remove it from the list
+ $sendUpdate(\mathcal{W}_1, \mathcal{W}_2, \mathcal{G}_2, \Delta)$: update the balance of Sender, Receiver and also save a transaction info
+ $cashoutUpdate(\mathcal{W}_1, \mathcal{W}_2, \mathcal{G}_2, \Delta)$: update the balance of Payer, Payee and also save a transaction info

Record Contract: A *Record Contract* (Algorithm 2) is also a Genesis smart contract. Similar to the *Token Contract*, the *Record Contract* is also deployed to obtains an address $\mathcal{R}$ at the beginning. *Record Contract* takes responsibility as a record of notary service. It saves a list of validators, used transaction id, and pre-claim information. Along with recording vital information, *Record Contract* also has a function that can verify endorsing signatures. To initialize, add, or cash-out Tokens, clients must submit required signatures of endorsement that are given by *Validators* for verification. A group that successfully joins a network is also required to link its contract to $\mathcal{R}$.

Algorithm 2 Record Contract - A Genesis Contract

- Founder deploys a Record contract to obtain $\mathcal{R}$
- Record Contract contains:
+ Token Contract: Token contract's address $\mathcal{T}$
+ map[address][boolean]Validators: a record list of validators
+ map[string][boolean]TxID: a record list of used TxIDs
+ map[address][Info]PreClaim: a record list info of pre-claim
+ $addvalidator(\mathcal{V})$: add validator's address ($\mathcal{V}$) into a record list
+ $addTxID(\mathcal{T}_{id})$: add transaction id into a list to prevent a replay attack
+ $checkTxID(\mathcal{T}_{id})$: check whether transaction id has been used before
+ $addPreClaim(\mathcal{S}_i, \mathcal{P}_k, \Delta)$: add sign information ($\mathcal{S}_i$), pubkey ($\mathcal{P}_k$), and amount ($\Delta$) as pre-claim
+ $getPreClaim(\mathcal{C})$: return pre-claim info by a client ($\mathcal{C}$)
+ $verify(\mathcal{H}_m, \mathcal{S}_{v1}, \mathcal{S}_{v2}, \mathcal{S}_{v3})$: retrieve a signer of message hash ($\mathcal{H}_m$) with each signature ($\mathcal{S}_{v1}, \mathcal{S}_{v2}, \mathcal{S}_{v3}$) and check with a list of validators.

Wallet Contract: A *Wallet Contract* (Algorithm 3) is a pre-defined contract that can be deployed by a client.

Algorithm 3 Wallet Contract - A Pre-defined Contract

- Client deploys a Wallet contract to obtain $\mathcal{W}$
- Wallet Contract contains:
+ Owner: 256-bit Ethereum address of account's owner $\mathcal{C}$
+ GroupTrade: an address of *GroupTrade* ($\mathcal{G}$)
+ InitBalance: an initialized amount in the wallet
+ $setInitBalance(\Delta)$: set balance of a wallet ($\mathcal{W}$) before joining a *GroupTrade* ($\mathcal{G}$)
+ $deposit(\mathcal{T}_{id}, \mathcal{S}_{v1}, \mathcal{S}_{v2}, \mathcal{S}_{v3}, \Delta)$: request to add more Tokens into a wallet
+ $setGroup()$: called by *GroupTrade* contract ($\mathcal{G}$)
+ $getGroup()$: return a *GroupTrade* contract ($\mathcal{G}$) that holds this wallet
+ $removeGroup()$: called by *GroupTrade* contract ($\mathcal{G}$)

Clients deploy the contract source code to create an instance of *Wallet Contract* ($\mathcal{W}$). Unlike regular wallets that hold coins, wallet contracts do not hold any Tokens in my scheme. Since Ethereum does not support external assets other than Ether (ETH) but supporting ERC-20 [26, 27], I take advantage of the ERC-20 protocol to create my *Token Contract* to issue Token coins. The total balance of each wallet ($\mathcal{W}$) is saved by the *Token Contract* instead. When a *Wallet Contract* is deployed, a client is required to specify a number of Tokens to be initialized. However, this initial

balance has not been activated yet. To activate it, clients are required to collect signatures from *Validators* and successfully deploy/join a *GroupTrade* contract ($\mathcal{G}$). By design, one node client's address ($\mathcal{C}$) on the $\mathcal{VMC}$ may create multiple *Wallet Contracts*. But, each *Wallet Contract* should only join one *GroupTrade*.

GroupTrade Contract: A *GroupTrade* contract (Algorithm 4) is a pre-defined contract that can be deployed to the blockchain. Unlike a payment channel in the Lightning Network, *GroupTrade* contains a group of clients who agree to join by a specific trade deal.

Algorithm 4 GroupTrade Contract

- A group's host deploys a GroupTrade contract to obtain $\mathcal{G}$ and adds its wallet $\mathcal{W}$ into this group
- GroupTrade Contract contains:
+ Token Contract: Token contract's address $\mathcal{T}$
+ Record Contract: Record contract's address $\mathcal{R}$
+ map[address][address]Wallet: a mapping list of wallet ($\mathcal{W}$) and account ($\mathcal{C}$)
+ map[address][Request]CashoutReq: a mapping list of cash-out requests by an account $\mathcal{C}$
+ $getBalance(\mathcal{C})$: return the balance of an account ($\mathcal{C}$) within this *GroupTrade*
+ $getWallet(\mathcal{C})$: return a wallet address ($\mathcal{W}$) of an account ($\mathcal{C}$) within *GroupTrade*
+ $joinGroup(\mathcal{W}, \mathcal{T}_{id}, \mathcal{S}_{v1}, \mathcal{S}_{v2}, \mathcal{S}_{v3})$: request to join *GroupTrade* with endorsement signatures
+ $leaveGroup()$: request to leave this *GroupTrade*
+ $cashoutRequest(\mathcal{B}, \Delta)$: request a cash-out with information - receiving address ($\mathcal{B}$), amount (Δ)
+ $getRequest(\mathcal{C})$: return a cash-out request information by a client ($\mathcal{C}$)
+ $claimToPayReq(\mathcal{C}_2)$: Payer ($\mathcal{C}_1$) claims to pay the cash-out request of Payee ($\mathcal{C}_2$)
+ $send(\mathcal{C}, \mathcal{G}, \Delta)$: send an amount ($\Delta$) Tokens to an account ($\mathcal{C}$) of *GroupTrade* ($\mathcal{G}$)
+ $pay(\mathcal{C}, \mathcal{G}, \mathcal{T}_{id}, \mathcal{S}_{v1}, \mathcal{S}_{v2}, \mathcal{S}_{v3}, \Delta)$: pay the cash-out request of an account ($\mathcal{C}$) within *GroupTrade* ($\mathcal{G}$) an amount (Δ)

A host of a group deploys the contract source code to create an instance of *GroupTrade* contract ($\mathcal{G}$). When *GroupTrade* is deployed, it also automatically triggers an event to request adding ($\mathcal{G}$) into a *Token Contract* ($\mathcal{T}$) along with minting Token coins. A group's host may start a *GroupTrade* with *0* Token. Regardless of initializing amount, members and the host are required to collect required signatures and to pass a verification process before successfully joining the trading network. Similar to a mesh network of nodes in the Lightning Network, my proposing scheme also allows trading among participants in different *GroupTrades*. Moreover, the scheme provides a base code

of *GroupTrade* that contains minimal and fundamental functions to operate. In reality, clients are allowed to modify the code to suit their intended usage, i.e., adding more functions or linking to another contract that defines an exchange rate in swapping Tokens.

4.2 Wallet Initialization

The wallet initialization is a cross-chain process. A wallet is created by deploying a smart contract and is set an initial balance by clients on $\mathcal{VMC}$. In return, clients obtain a 256-bits address of a wallet ($\mathcal{W}$). However, this wallet ($\mathcal{W}$) has not entirely been activated. To activate it, clients are required to broadcast a request message to collect enough signatures of endorsement from the *Validators*. Upon gathering enough endorsements, clients must then deploy or join a group payment by using a pre-defined contract source code. A *GroupTrade* contract has a function that internally sends a request to mint Tokens for $\mathcal{W}$ in *Token* contract.

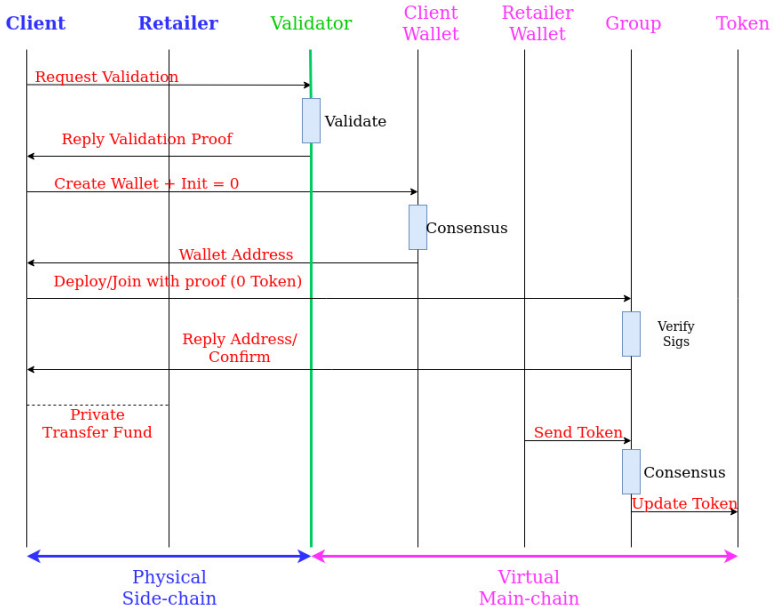

Figure 4.7: Initialize Wallet with $\sigma = 0$

For wallet initialization, Clients have two options:

- $\sigma = 0$ Token: clients join a trading network with *0* Token. Later, they can make a trade deal with other third parties to receive Tokens (Figure 4.7).

- $\sigma = k$ Tokens: clients are required to transfer funds into Reserves[8] ($\mathcal{RS}$) on $\mathcal{PSC}$ (Figure 4.8).

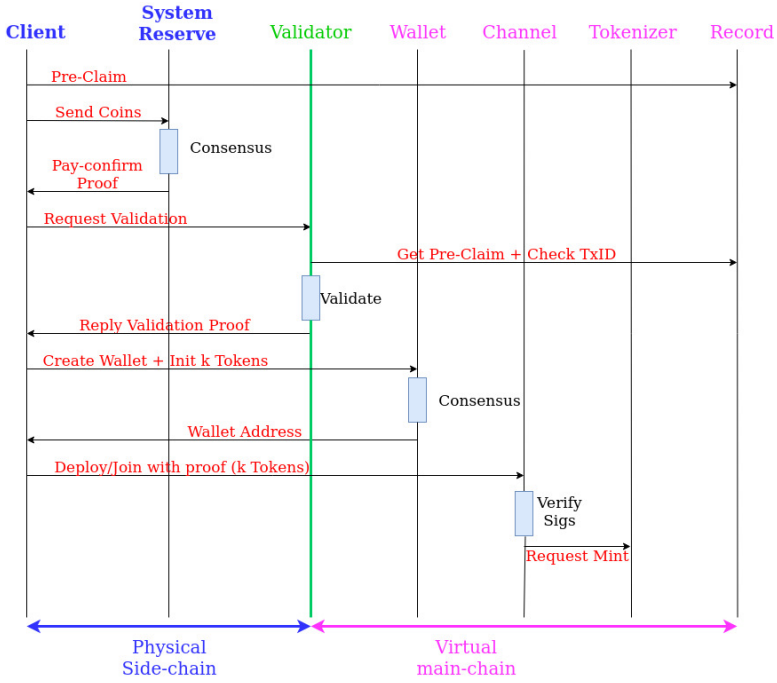

Figure 4.8: Initialize Wallet with $\sigma = k$

To participate in a cross-chain trading network on the $\mathcal{VMC}$, a Bitcoin client ($\mathcal{BC}$) or a Lightning Network client ($\mathcal{LC}$) is required to go through a process of verification regardless of an initializing amount. The process and time to join the network, when $\sigma = 0$, is faster than the other since clients do not need to transfer funds to Reserves ($\mathcal{RS}$), and *Validators* do not have to verify a transferring fund transaction on $\mathcal{PSC}$. In the next following parts, the thesis elaborates concrete procedures of how clients on the $\mathcal{PSC}$ (e.g., Bitcoin or Lightning Network) can initialize their wallets on the $\mathcal{VMC}$.

Balance Init $\sigma = 0$: Both $\mathcal{LC}$ and $\mathcal{BC}$ have the same procedures in requesting to join a cross-chain trading network (see Algorithm 5). Regardless of requesting *0 Token*, a client is required to send a request to *Validators* to collect a number of required signatures. Since transferring funds to Reserves (RS) does not take place, transaction id ($\mathcal{T}_{id}$) is not generated. Instead, in step 1, this client concatenates an arbitrary string (λ) with a node's address ($\mathcal{C}$) on the $\mathcal{VMC}$, then passes it to the *SHA256* function to generate $\mathcal{T}_{id}$. By this design, one client is allowed to create as many wallets

[8]Reserves: System Addresses that receive funds from Clients on $\mathcal{PSC}$

as they need as long as the initialized amount $\sigma = 0$. In the case of *validate_0*, a verification process is quick and straightforward. *Validators* only check whether the requesting amount is $\sigma = 0$, then they can reply to the client with a signature of endorsement (γ). Once collecting enough signatures, this client can start to create or join a *GroupTrade* ($\mathcal{G}$) along with submitting signatures for another verification process by *Token Contract* and *Record Contract*.

Balance Init $\sigma = k$: In the case of requesting an amount of Token ($\sigma = k$) in initializing a wallet, $\mathcal{LC}$ and $\mathcal{BC}$ have different procedures to prepare data before claiming. Thus, *Validators* also have different mechanisms to verify claiming data. In the next following part, I provide detail procedures for these two types of the client when they request to join, to initialize $\sigma = k$ Tokens and also how *Validators* verify data upon receiving a request.

Algorithm 5 Wallet Initialization - Apply for $\mathcal{BC}$ and $\mathcal{LC}$

procedure BALANCE INIT $\sigma = 0$
- Step 1: Client prepares $\mathcal{T}_{id}$ = SHA256(m) with $m = \lambda + \mathcal{C}$, λ is arbitrary string
- Step 2: Client broadcasts a request (*validate_0*, $\mathcal{T}_{id}$, $\sigma = 0$) to collect signatures
- Step 3: Upon receiving a request (*validate_0*), Validators verify a request
  if $\sigma = 0$ **then**
    Reply proof $\gamma = Sign(m_{rep})$ with $m_{rep} = (\text{"Init"}, \mathcal{T}_{id}, 0, \mathcal{C})$
  end if
- Step 4: Upon receiving a reply, Client saves into its list $\mathcal{LS}$
  if $len(\mathcal{LS}) = \alpha$ with α = (3,..,n) **then**
    Move to Step 5
  end if
- Step 5: Client creates *Wallet Contract* ($\mathcal{W}$), and sets *init_balance = 0*
- Step 6: Client executes a contract to *create/join GroupTrade* ($\mathcal{G}$) with proof in $\mathcal{L}$
- Step 7: A contract ($\mathcal{G}$) sends a request with signatures ($\gamma_1, \gamma_2, ...$) to ($\mathcal{T}$)
  if $(\gamma_1, \gamma_2, ...) = true$ **then**
    Save $\mathcal{T}_{id}$ in *Record Contract* ($\mathcal{R}$)
    Internally request to add ($\mathcal{G}$) into *Token Contract* ($\mathcal{T}$)
    Mint *0* Token for $\mathcal{C}$
  end if

end procedure

Bytes	Name	Data Type	Description
4	version	int32_t	Transaction version number (note, this is signed); currently version 1 or 2. Programs creating transactions using newer consensus rules may use higher version numbers. Version 2 means that BIP 68 applies.
Varies	tx_in count	compactSize uint	Number of inputs in this transaction.
Varies	tx_in	txIn	Transaction inputs. See description of txIn below.
Varies	tx_out count	compactSize uint	Number of outputs in this transaction.
Varies	tx_out	txOut	Transaction outputs. See description of txOut below.
4	lock_time	uint32_t	A time (Unix epoch time) or block number. See the locktime parsing rules.

Figure 4.9: Raw Transaction format

Image from [9]

```
> bitcoin-cli -regtest createrawtransaction '"
  [
   {
    "txid": "'$UTXO_TXID'",
    "vout": '$UTXO_VOUT'
   }
  ]
 "' '"
  {
   "'$NEW_ADDRESS'": 49.9999
  }'"
01000000017b1eabe0209b1fe794124575ef807057c77ada2138ae4fa8d6c4de\
0398a14f3f0000000000ffffffff01f0ca052a010000001976a914cbc20a7664\
f2f69e5355aa427045bc15e7c6c77288ac00000000

> RAW_TX=01000000017b1eabe0209b1fe794124575ef807057c77ada2138ae4[...]
```

(a) Create Raw Tx

```
> bitcoin-cli -regtest signrawtransaction $RAW_TX
{
  "hex" : "01000000017b1eabe0209b1fe794124575ef807057c77ada213\
       8ae4fa8d6c4de0398a14f3f0000000049483045022100894 9f0\
       cb400094ad2b5eb399d59d01c14d73d8fe6e96df1a7150deb38\
       8ab8935022079656090d7f6bac4c9a94e0aad311a4268e082a7\
       25f8aeae0573fb12ff866a5f01ffffffff01f0ca052a0100000\
       01976a914cbc20a7664f2f69e5355aa427045bc15e7c6c77288\
       ac00000000",
  "complete" : true
}

> SIGNED_RAW_TX=01000000017b1eabe0209b1fe794124575ef807057c77ada[...]
```

(b) Hash of Sign Raw Tx

Figure 4.10: Create Raw Transaction and Sign Raw Transaction

Image from [9]

Balance Init $\sigma = k$ with $\mathcal{BC}$ (Algorithm 6): The $\mathcal{BC}$ first self-creates a Raw Transaction ($\mathcal{RT}$). Then, this client uses its private key ($\mathcal{B}_{prk}$) to generate a hash of Sign Raw Transaction (Φ) as a digital fingerprint to prove an account's ownership (see Figure 4.10b) before transferring funds

to *Reserves* ($\mathcal{RS}$), .

Algorithm 6 Wallet Initialization - Apply for $\mathcal{BC}$

procedure BALANCE INIT $\sigma = k$
- Step 1: Client self-creates a Raw Transaction ($\mathcal{RT}$)
- Step 2: Client generates a Sign Raw Transaction (Φ) with $\Phi = (\mathcal{RT}, \mathcal{B}_{prk})$
- Step 3: Client submits $\mathcal{H}_S = Sign(\Phi, \mathcal{C}_{prk})$ to *Record Contract* ($\mathcal{R}$) as a pre-claim
- Step 4: Client transfers funds to *Reserve* ($\mathcal{RS}$) and waits until $\mathcal{T}_{id}$ is mined
- Step 5: Client broadcasts a request (*validate_k*, $\mathcal{T}_{id}$, Φ, $\sigma = k$) to collect signatures
- Step 6: Upon receiving a request (*validate_k*), Validators verify a request
  if $\mathcal{R}(\mathcal{T}_{id}) = false$ **then**
    Get transaction info using $\mathcal{T}_{id}$
    if $(\sigma = k) = true$ and $\mathcal{RS} = true$ and $\Phi = true$ **then**
      Get pre-claim info from $\mathcal{R}$
      if $\mathcal{C} = ecrecover(\mathcal{H}_S, \Phi)$ **then**
        Reply proof $\gamma = Sign(m_{rep})$ with $m_{rep} = (\text{"Init"}, \mathcal{T}_{id}, k, \mathcal{C})$
      end if
    end if
  end if
- Step 7: Upon receiving a reply, Client saves into its list $\mathcal{LS}$
  if $len(\mathcal{LS}) = \alpha$ with $\alpha = (3,..,n)$ **then**
    Move to Step 8
  end if
- Step 8: Client creates *Wallet Contract* ($\mathcal{W}$), and sets *init_balance* $= k$
- Step 9: Client executes a contract to *create/join GroupTrade* ($\mathcal{G}$) with proof in $\mathcal{L}$
- Step 10: A contract ($\mathcal{G}$) sends a request with signatures $(\gamma_1, \gamma_2, ...)$ to ($\mathcal{T}$)
  if $(\gamma_1, \gamma_2, ...) = true$ **then**
    Save $\mathcal{T}_{id}$ in *Record Contract* ($\mathcal{R}$)
    Internally request to add ($\mathcal{G}$) into *Token Contract* ($\mathcal{T}$)
    Mint k Tokens for $\mathcal{C}$
  end if

end procedure

In Bitcoin, a transaction, which generated among peers, is broadcast in a serialized byte format, called raw format. This raw format is a pre-form to create a transaction id (TxID). A raw transaction contains inputs and outputs. Transaction's inputs are addresses of unspent transactions. One of the outputs is addresses of receivers while another output is a new generating address for

remaining change after spending (see Figure 4.9, Figure 4.11a, Figure 4.10a).

```
01000000 ..... Version

01 ..... Number of inputs
|
| 7b1eabe0209b1fe794124575ef807057
| c77ada2138ae4fa8d6c4de0398a14f3f ..... Outpoint TXID
| 00000000 ..... Outpoint index number
|
| 49 ..... Bytes in sig. script: 73
| | 48 ..... Push 72 bytes as data
| | | 30450221008949f0cb400094ad2b5eb3
| | | 99d59d01c14d73d8fe6e96df1a7150de
| | | b388ab8935022079656090d7f6bac4c9
| | | a94e0aad311a4268e082a725f8aeae05
| | | 73fb12ff866a5f01 ..... Secp256k1 signature
|
| ffffffff ..... Sequence number: UINT32_MAX

01 ..... Number of outputs
| f0ca052a01000000 ..... Satoshis (49.99990000 BTC)
|
| 19 ..... Bytes in pubkey script: 25
| | 76 ..... OP_DUP
| | a9 ..... OP_HASH160
| | 14 ..... Push 20 bytes as data
| | | cbc20a7664f2f69e5355aa427045bc15
| | | e7c6c772 ..... PubKey hash
| | 88 ..... OP_EQUALVERIFY
| | ac ..... OP_CHECKSIG

00000000 ..... locktime: 0 (a block height)
```

(a) Bytes of Raw String Explain

```
> bitcoin-cli -regtest decoderawtransaction $RAW_TX
{
  "txid" : "c80b343d2ce2b5d829c2de9854c7c8d423c0e33bda264c4013\
       8d834aab4c0638",
  "hash" : "c80b343d2ce2b5d829c2de9854c7c8d423c0e33bda264c40138d834aab4c063
  "size" : 85,
  "vsize" : 85,
  "version" : 1,
  "locktime" : 0,
  "vin" : [
    {
      "txid" : "3f4fa19803dec4d6a84fae3821da7ac7577080ef75\
           451294e71f9b20e0ab1e7b",
      "vout" : 0,
      "scriptSig" : {
        "asm" : "",
        "hex" : ""
      },
      "sequence" : 4294967295
    }
  ],
  "vout" : [
    {
      "value" : 49.99990000,
      "n" : 0,
      "scriptPubKey" : {
        "asm" : "OP_DUP OP_HASH160 cbc20a7664f2f69e5355a\
             a427045bc15e7c6c772 OP_EQUALVERIFY OP_CHECKSIG",
        "hex" : "76a914cbc20a7664f2f69e5355aa427045bc15e\
             7c6c77288ac",
        "reqSigs" : 1,
        "type" : "pubkeyhash",
        "addresses" : [
          "mz6KvC4aoUeo6wSxtiVQTo7FDwPnkp6URG"
        ]
```

(b) Decode Raw Tx

Figure 4.11: Raw String and Raw Transaction Explanation

Image from [9]

In Bitcoin, the ledger provides a method to decode a raw transaction. Thus, detail inputs and outputs of a transaction can be verified (Figure 4.11b). Furthermore, a Sign Raw Transaction is also included in a transaction when it is successfully mined and added into a block. Importantly, only a node's owner can generate a Raw Transaction and a Sign Raw Transaction. Taking advantage of these features, the scheme requires the $\mathcal{BC}$ to prepare pre-claim information so that this client,

[9] https://bitcoin.org/

at a later time, can claim that the transferring funds are successfully and indeed sent by him/her. To make it more secure, the Sign Raw Transaction (Φ) is then encrypted once again by using a node client's private key ($\mathcal{C}_{prk}$) on $\mathcal{VMC}$ to generate a signature ($\mathcal{H}_S$) before submitting it to *Record Contract* ($\mathcal{R}$) as a pre-claim. Once the funds are successfully transferred to *Reserves* ($\mathcal{RS}$) and a transaction ($\mathcal{T}_{id}$) is completely added into a block in the $\mathcal{PSC}$, the $\mathcal{BC}$ can start to request (*validate_k*, $\mathcal{T}_{id}$, Φ, $\sigma = k$) to collect endorsements from *Validators*. Waiting for a transaction completely being added into a block guarantees that this transaction is irreversible.

Unlike verifying a request *validate_0*, *Validators* must verify a transferring fund transaction ($\mathcal{T}_{id}$) on the $\mathcal{PSC}$, Bitcoin. To prevent a replay attack, *Validators* access *Record Contract* to check whether $\mathcal{T}_{id}$ has been claimed before querying to retrieve detail transaction information. Once the verification is passed, *Validators* reply to a requestor ($\mathcal{BC}$) a signature (γ) as a proof of endorsement. A message, using to generate γ, consists of a request's purpose, a TxID, an amount, and an address of Requestor so that this endorsement is bound only to the request. In receiving Tokens, the $\mathcal{BC}$ is required to create or to join a *GroupTrade* ($\mathcal{G}$), including to provide the signatures of endorsement. *Token Contract* and *Record Contract* take responsibility to verify that all approval signatures are generated by authorized *Validators* before activating a number of requesting Tokens. In the end, the $\mathcal{BC}$ receives an address of transaction when the request is accepted or receives an error message otherwise.

Balance Init $\sigma = k$ with $\mathcal{LC}$: The Lightning client ($\mathcal{LC}$) has two options to initialize its wallet on $\mathcal{VMC}$. Each of them has a different procedure to verify. Two options are:

- Option 1: the $\mathcal{LC}$ transfers funds to Reserves ($\mathcal{RS}$).
- Option 2: the $\mathcal{LC}$ opens a payment channel with the Reserves ($\mathcal{RS}$).

In Option 1, the $\mathcal{LC}$ also goes through quite similar processes as the $\mathcal{BC}$ in initializing wallet with a number of Tokens (see Algorithm 7). But, the $\mathcal{LC}$ has a little difference in preparing data to pre-claim. Unlike the $\mathcal{BC}$ that is supported RPC calls to generate Raw Transaction (RT) and Sign Raw Transaction (Φ), the $\mathcal{LC}$ has to manually specify and create an array of addresses of unspent transactions ($\mathcal{UT}$) before applying a SHA256 function on an array to generate a hash ($\mathcal{H}_I$). This

hash is encrypted once again by using a private key ($\mathcal{C}_{prk}$) on $\mathcal{VMC}$ to generate a signature $\mathcal{H}_S$ before submitting to *Record Contract* ($\mathcal{R}$). By using two layers of encryption, secret data can be protected, and only a transaction's creator ($\mathcal{LC}$) can reveal the secret data to claim later on.

Algorithm 7 Wallet Initialization - Apply for LC Option 1

procedure BALANCE INIT $\sigma = k$
- Step 1: Client specifies and creates an array of addresses of unspent transactions going to be spent ($\mathcal{UT}$)
- Step 2: Client generates a hash $\mathcal{H}_I$ = SHA256($\mathcal{UT}$)
- Step 3: Client submits $\mathcal{H}_S = Sign(\mathcal{H}_I, \mathcal{C}_{prk})$ to *Record Contract* ($\mathcal{R}$) as a pre-claim
- Step 4: Client transfers funds to *Reserve* ($\mathcal{RS}$) and waits until $\mathcal{T}_{id}$ is mined
- Step 5: Client broadcasts a request (*validate_k*, $\mathcal{T}_{id}$, $\mathcal{UT}$, $\sigma = k$) to collect signatures
- Step 6: Upon receiving a request (*validate_k*), Validators verify a request
  if $\mathcal{R}(\mathcal{T}_{id}) = false$ **then**
    Get transaction info using $\mathcal{T}_{id}$
    if $(\sigma = k) = true$ and $\mathcal{RS} = true$ and $\mathcal{UT} = true$ **then**
      Get pre-claim info from $\mathcal{R}$
      Generate a hash $\mathcal{H}_V$ = SHA256($\mathcal{UT}$)
      if $\mathcal{C} = ecrecover(\mathcal{H}_S, \mathcal{H}_V)$ **then**
        Reply proof $\gamma = Sign(m_{rep})$ with $m_{rep} = (\text{"Init"}, \mathcal{T}_{id}, k, \mathcal{C})$
      end if
    end if
  end if
- Step 7: Upon receiving a reply, Client saves into its list $\mathcal{LS}$
  if $len(\mathcal{LS}) = \alpha$ with α = (3,..,n) **then**
    Move to Step 8
  end if
- Step 8: Client creates *Wallet Contract* ($\mathcal{W}$), and sets *init_balance* = k
- Step 9: Client executes a contract to *create/join GroupTrade* ($\mathcal{G}$) with proof in $\mathcal{L}$
- Step 10: A contract ($\mathcal{G}$) sends a request with signatures $(\gamma_1, \gamma_2, ...)$ to ($\mathcal{T}$)
  if $(\gamma_1, \gamma_2, ...) = true$ **then**
    Save $\mathcal{T}_{id}$ in *Record Contract* ($\mathcal{R}$)
    Internally request to add ($\mathcal{G}$) into *Token Contract* ($\mathcal{T}$)
    Mint k Tokens for $\mathcal{C}$
  end if
end procedure

Option 2 (Algorithm 8) is a particular method that only applies to the $\mathcal{LC}$ since Lightning

Network supports a payment channel among two stakeholders. By using a supporting RPC API, *OpenChannel*[10], the $\mathcal{LC}$ can open a payment channel, which funds are placed into, with one of the system *Reserves* ($\mathcal{RS}$) in the Lightning Network.

Algorithm 8 Wallet Initialization - Apply for LC Option 2

procedure BALANCE INIT $\sigma = k$
- Step 1: Client opens a channel payment $\mathcal{P}(\vartheta, \eta)$ to ($\mathcal{RS}$), with ϑ is local_amt, and η is push_amt
- Step 2: Client queries a new opening channel to retrieve a channel id (Ψ)
- Step 3: Client submits $\mathcal{H}_S = Sign(\Psi, \mathcal{C}_{prk})$ to *Record Contract* ($\mathcal{R}$) as a pre-claim
- Step 4: Client broadcasts a request (*validate_k*, Ψ, $\sigma = k$) to collect signatures
- Step 6: Upon receiving a request (*validate_k*), Validators verify a request
  if $\mathcal{R}(\Psi) = false$ **then**
    Send an HTTPS request to a webserver to query the channel info
    if $\sigma = k = \eta$ and $\mathcal{RS} = true$ **then**
      Get pre-claim info from $\mathcal{R}$
      if $\mathcal{C} = ecrecover(\mathcal{H}_S, \Psi)$ **then**
        Reply proof $\gamma = Sign(m_{rep})$ with $m_{rep} =$ ("Init", $\Psi, k, \mathcal{C}$)
      end if
    end if
  end if
- Step 7: Upon receiving a reply, Client saves into its list $\mathcal{LS}$
  if $len(\mathcal{LS}) = \alpha$ with $\alpha =$ (3,..,n) **then**
    Move to Step 8
  end if
- Step 8: Client creates *Wallet Contract* ($\mathcal{W}$), and sets *init_balance* $= k$
- Step 9: Client executes a contract to *create/join GroupTrade* ($\mathcal{G}$) with proof in $\mathcal{L}$
- Step 10: A contract ($\mathcal{G}$) sends a request with signatures $(\gamma_1, \gamma_2, ...)$ to ($\mathcal{T}$)
  if $(\gamma_1, \gamma_2, ...) = true$ **then**
    Save channel id (Ψ) in *Record Contract* ($\mathcal{R}$)
    Internally request to add ($\mathcal{G}$) into *Token Contract* ($\mathcal{T}$)
    Mint k Tokens for $\mathcal{C}$
  end if
end procedure

Without specifying a *push_amt* parameter, the RPC call only attempts to open a singly funded channel to a remote peer, which means that depositing funds are signed and under controlled by

[10] https://api.lightning.community/#openchannel

the $\mathcal{LC}$ instead of a remote peer, $\mathcal{RS}$ [28]. However, the Lightning Network also supports a feature that a channel's creator can transfer coins to a remote peer when a channel is created. In this case, the $\mathcal{LC}$ must specify two fundamental parameters, *local_amt* (ϑ) and *push_amt* (η). The *local_amt* specifies a total amount of funds that is placed into a channel, while *push_amt* is a number of coins that are transferred to a remote peer. For example, Alice opens a payment channel $\mathcal{P}(100, 40)$, $\vartheta = 100$ and $\eta = 40$, with Bob. In this example, a payment channel between Alice and Bob has a total of 100 coins with Alice owns 60 coins and Bob owns 40 coins in the channel $\mathcal{P}$.

Once a payment channel is created, a transaction $\mathcal{T}_{id}$ is then broadcast to the Bitcoin's ledger. Unlike Option 1, the proposing scheme does not use $\mathcal{T}_{id}$ as pre-claim information. Querying $\mathcal{T}_{id}$ only retrieves a total number of funds (ϑ) that has been transferred into a channel $\mathcal{P}$. *Validators* need the information of the *push_amt* (η) instead. Thus, to prepare pre-claim information, the $\mathcal{LC}$ has to query a new opening channel to get a channel id (Ψ) that is a unique string and is known by only two stakeholders. The channel id (Ψ) is then signed by using a private key ($\mathcal{C}_{prk}$) on $\mathcal{VMC}$ to generate a signature ($\mathcal{H}_S$) that is then submitted to *Record Contract* ($\mathcal{R}$) before broadcasting a request (*validate_k*, Ψ, $\sigma = k$) to *Validators* for endorsement.

The channel id (Ψ) and its information are secret and available to query only by a channel's creator ($\mathcal{LC}$) or a remote peer ($\mathcal{RS}$). Hence, the scheme also implements a web server that handles a request to query the channel's information. To verify, *Validators* sends an HTTPs request to the webserver to query the channel Ψ. Once *Validators* verify that required funds are successfully sent to *Reserve*, they reply to the $\mathcal{LC}$ a signature (γ) as a proof of endorsement. And the rest process to initialize a number of Tokens for the *Wallet Contract* ($\mathcal{W}$) of the $\mathcal{LC}$ is similar to Option 1.

4.3 Sending and Adding Tokens

4.3.1 Sending Tokens

The process is simple, fast, and straightforward by using smart contracts. There is no cross-chain validation when sending Tokens among clients in the $\mathcal{VMC}$. Instead, transactions are generated and verified through smart contracts, then added into blocks by a mining process that is an Ethereum's

consensus in this case. Clients can spend on what they have possessed, which is saved by *Token Contract* ($\mathcal{T}$). Furthermore, the trading network may consist of many *GroupTrade* (e.g., $\mathcal{G}_1$, $\mathcal{G}_2$, $\mathcal{G}_3$). By creating a scalable and unlimited trading environment, my trading network allows members of this group can trade with members of other groups.

4.3.2 Adding Tokens

Algorithm 9 Add Token - Apply for *BC*

procedure ADD $\sigma = k$ TOKENS
- Step 1: Client self-creates a Raw Transaction ($\mathcal{RT}$)
- Step 2: Client generates a Sign Raw Transaction (Φ) with $\Phi = (\mathcal{RT}, \mathcal{B}_{prk})$
- Step 3: Client submits $\mathcal{H}_S = Sign(\Phi, \mathcal{C}_{prk})$ to *Record Contract* ($\mathcal{R}$) as a pre-claim
- Step 4: Client transfers funds to *Reserve* ($\mathcal{RS}$) and waits until $\mathcal{T}_{id}$ is mined
- Step 5: Client broadcasts a request (*add_k*, $\mathcal{T}_{id}$, Φ, $\sigma = k$) to collect signatures
- Step 6: Upon receiving a request (*add_k*), Validators verify a request
  if $\mathcal{R}(\mathcal{T}_{id}) = false$ **then**
    Get transaction info using $\mathcal{T}_{id}$
    if $(\sigma = k) = true$ and $\mathcal{RS} = true$ and $\Phi =$ true **then**
      Get pre-claim info from $\mathcal{R}$
      if $\mathcal{C} = ecrecover(\mathcal{H}_S, \Phi)$ **then**
        Reply proof $\gamma = Sign(m_{rep})$ with $m_{rep} =$ ("Add", $\mathcal{T}_{id}, k, \mathcal{C}$)
      end if
    end if
  end if
- Step 7: Upon receiving a reply, Client saves into its list $\mathcal{LS}$
  if $len(\mathcal{LS}) = \alpha$ with $\alpha =$ (3,..,n) **then**
    Move to Step 8
  end if
- Step 8: Client calls *addToken()* of *GroupTrade* ($\mathcal{G}$) with proof in $\mathcal{L}$
- Step 9: A contract ($\mathcal{G}$) sends a request with signatures $(\gamma_1, \gamma_2, ...)$ to ($\mathcal{T}$)
  if $(\gamma_1, \gamma_2, ...) = true$ **then**
    Save $\mathcal{T}_{id}$ in *Record Contract* ($\mathcal{R}$)
    Internally request to mint k Tokens for $\mathcal{C}$ in *Token Contract* ($\mathcal{T}$)
  end if
end procedure

This function helps clients, both $\mathcal{BC}$ or $\mathcal{LC}$, to replenish a number of Tokens in their existing accounts. The process (Algorithm 9, 10) is almost similar to initializing a wallet with $\sigma = k$ that is presented in previous parts. To request, both types of clients ($\mathcal{BC}$ or $\mathcal{LC}$) must have their *Wallet Contract* ($\mathcal{W}$) still being active and joining a *GroupTrade* ($\mathcal{G}$).

Algorithm 10 Add Token - Apply for *LC*

procedure ADD $\sigma = k$ TOKENS
- Step 1: Client specifies a list of unspent transactions going to be spent ($\mathcal{UT}$)
- Step 2: Client generates a hash $\mathcal{H}_I$ = SHA256($\mathcal{UT}$)
- Step 3: Client submits $\mathcal{H}_S = Sign(\mathcal{H}_I, \mathcal{C}_{prk})$ to *Record Contract* ($\mathcal{R}$) as a pre-claim
- Step 4: Client transfers funds to *Reserve* ($\mathcal{RS}$) and waits until $\mathcal{T}_{id}$ is mined
- Step 5: Client broadcasts a request (*add_k*, $\mathcal{T}_{id}$, $\mathcal{UT}$, $\sigma = k$) to collect signatures
- Step 6: Upon receiving a request (*add_k*), Validators verify a request
  if $\mathcal{R}(\mathcal{T}_{id}) = false$ **then**
    Get transaction info using $\mathcal{T}_{id}$
    if $(\sigma = k) = true$ and $\mathcal{RS} = true$ and $\mathcal{UT} = true$ **then**
      Get pre-claim info from $\mathcal{R}$
      Generate a hash $\mathcal{H}_V$ = SHA256($\mathcal{UT}$)
      if $\mathcal{C} = ecrecover(\mathcal{H}_S, \mathcal{H}_V)$ **then**
        Reply proof $\gamma = Sign(m_{rep})$ with $m_{rep} = (\text{"Add"}, \mathcal{T}_{id}, k, \mathcal{C})$
      end if
    end if
  end if
- Step 7: Upon receiving a reply, Client saves into its list $\mathcal{LS}$
  if $len(\mathcal{LS}) = \alpha$ with α = (3,..,n) **then**
    Move to Step 8
  end if
- Step 8: Client calls *addToken()* of *GroupTrade* ($\mathcal{G}$) with proof in $\mathcal{L}$
- Step 9: A contract ($\mathcal{G}$) sends a request with signatures $(\gamma_1, \gamma_2, ...)$ to ($\mathcal{T}$)
  if $(\gamma_1, \gamma_2, ...) = true$ **then**
    Save $\mathcal{T}_{id}$ in *Record Contract* ($\mathcal{R}$)
    Internally request to mint k Tokens for $\mathcal{C}$ in *Token Contract* ($\mathcal{T}$)
  end if
end procedure

Before requesting to add Tokens, clients are required to transfer funds into *Reserves* ($\mathcal{RS}$) and wait for $\mathcal{T}_{id}$ to be mined. Unlike initializing a wallet with $\sigma = k$ that offers the $\mathcal{LC}$ a method to open

a payment channel in transferring funds, the adding Token feature does not support this option for all $\mathcal{LC}s$. Opening a new payment channel, when coins need to be transferred to *Reserves*, generates uncontrolled dummy channels on the $\mathcal{PSC}$, Lightning Network. In addition, the Lightning Network also provides an RPC call, "*sendpayment*"[11], for existing payment channels in the network. This method is an HTLC off-chain payment that does not require a transaction to be finalized on-chain immediately. Thus, it could be faster to transfer funds to a remote peer. However, my scheme also does not support this option since this method requires an invoice to be generated, which is described later. Generating an invoice, in this case, might cause an unwanted delay. This problem is caused by the limitation of smart contracts since a smart contract cannot automatically generate a secure hash of invoice on the $\mathcal{PSC}$.

4.4 Cashing Out Tokens

The *Cash-out* is a function that helps clients convert their Tokens back to coins on the corresponding $\mathcal{PSC}$. This feature is a cross-chain procedure, and *Validators* have to be involved in verifying a payment among a requestor and a payer. Clients are allowed to convert all of their Tokens, stored by *Token Contract* ($\mathcal{T}$), back to supporting side-chain coins. In reality, this process may accompany a small amount of fee. For the sake of simplicity, my scheme does not take it into account. The procedure of cashing-out Tokens consists of three phases:

- Phase 1: Payee ($\mathcal{C}_1$) submits a cash-out request to *GroupTrade*, i.e., $\mathcal{G}_1$
- Phase 2: Payer ($\mathcal{C}_2$), in any group, claims to pay a request. After claiming, $\mathcal{G}_1$ locks a cash-out request and only allows Payer ($\mathcal{C}_2$), *Validators* to access and to get the information.
- Phase 3: Payer ($\mathcal{C}_2$) accesses $\mathcal{G}_1$ and gets the cash-out request information, then transfers funds to Payee ($\mathcal{C}_1$) on $\mathcal{PSC}$. Finally, Payer ($\mathcal{C}_2$) updates its payment using *pay()* in *GroupTrade* contract (i.e., $\mathcal{G}_2$) to request adding Tokens.

To cash-out Tokens, requestors (Payee) submit a cashout request and add it into the *GroupTrade* contract ($\mathcal{G}$). For a method of payment, they may choose their preferred account, Bitcoin and/or

[11] https://api.lightning.community/#sendpayment

Lightning account, in receiving coins. Thus, the essential information of a cash-out request should be specified.

Cash-out request's information includes:

- An address of Requestor's account ($\mathcal{C}_1$) on the $\mathcal{VMC}$;
- A Bitcoin's address ($\mathcal{B}_1$) in a Bitcoin's ledger (Option 2);
- A Lightning's address ($\mathcal{L}_1$) in a Lightning Network (Option 1);
- A hash of payment request ($\mathcal{H}_{pr}$) in a Lightning Network (Option 1);
- A secret message ($\mathcal{M}_s$) for payment confirm (Option 1);
- A requesting cash-out amount (Δ);
- An address of Payer's account ($\mathcal{C}_2$) on the $\mathcal{VMC}$;
- A boolean value *isLocked*.

In Phase 1: A requestor, who submit a cash-out request, must specify an amount (Δ) and the address on $\mathcal{PSC}$ to receive coins. Smart contracts verify a cash-out amount with the total balance that a requestor owns and is saved in the *Token Contract*. Once a request is accepted and saved by a *GroupTrade* contract, it cannot be accessed by any clients, except *Validators*, since Payer has not claimed to pay yet. There are three cases to provide the necessary information in receiving coin:

- Payee only accepts payment via Bitcoin's chain (Option 1). Thus, a Bitcoin address ($\mathcal{B}_1$) must be provided, whereas other information ($\mathcal{L}_1$, $\mathcal{H}_{pr}$, and $\mathcal{M}_s$) can be omitted.
- In the case of accepting payment via the Lightning Network's chain only (Option 2), Payee must provide $\mathcal{L}_1$, $\mathcal{H}_{pr}$, and $\mathcal{M}_s$, whereas Bitcoin's address ($\mathcal{B}_1$) can be left in blank.
- In the last case, accepting payment by both methods, the essential information, $\mathcal{B}_1$, $\mathcal{L}_1$, $\mathcal{H}_{pr}$, and $\mathcal{M}_s$, should be provided in the cash-out request.

In Phase 2: Payer ($\mathcal{C}_2$) calls *claimToPayReq()* in a contract ($\mathcal{G}_1$), a *GroupTrade* contract that a requestor submits his/her cash-out request, with an address of the requestor, Payee ($\mathcal{C}_1$). Using the providing address $\mathcal{C}_1$, the contract $\mathcal{G}_1$ queries its mapping to find a request and to update it with an address of Payer ($\mathcal{C}_2$). Once it is claimed, the contract $\mathcal{G}_1$ locks and binds the Payer with this cash-out request, ($\{$*cashout*, $\mathcal{C}_1$, $\mathcal{B}_1$, $\mathcal{L}_1$, $\mathcal{H}_{pr}$, $\mathcal{M}_s$, Δ, *address(0)*, false$\}$ $\rightarrow$ $\{$*cashout*, $\mathcal{C}_1$, $\mathcal{B}_1$,

$\mathcal{L}_1$, $\mathcal{H}_{pr}$, $\mathcal{M}_s$, Δ, $\mathcal{C}_2$, true}). By now, Payer ($\mathcal{C}_2$) and *Validators* can access to get the request information.

In Phase 3: Payer ($\mathcal{C}_2$) accesses the *GroupTrade* contract ($\mathcal{G}_1$) once again to get the necessary information to make a payment. This process does not generate a transaction. Thus, it is fast to retrieve the information. After this step, cross-chain activities are taken place. This procedure requires the involvement of *Validators* to verify a transferring fund transaction on $\mathcal{PSC}$ that can be either Lightning Network or Bitcoin's chain depending on a payment method that Payee accepts.

In the next following sections, the thesis presents how a client makes a payment for a cash-out request on the $\mathcal{PSC}$ and also how to request to update Tokens on the $\mathcal{VMC}$ by providing a detailed procedure for both types of payment options, via Lightning Network and Bitcoin.

4.4.1 Paying and Updating Tokens Using Option 1

In this option 1, Payer ($\mathcal{LC}_2$) can make a payment for a cash-out request from a requestor, Payee ($\mathcal{LC}_1$). This method requires a fundamental condition, which is that Payer must open a payment channel with Payee, or there should be a route from Payer to Payee so that funds can be transferred along the route in the Lighting Network. In this thesis, the scheme assumes that a payment channel between Payer and Payee has been created. Hence, Payer can send coins to Payee using a supporting RPC call, *sendpayment*, over a channel among them. However, this procedure still has a critical problem when *Validators* verify a payment. After a payment channel is created among two stakeholders, transactions among them can be done in off-chain fashion using the scripting HTLC contract. Some information about an off-chain transaction is made public, whereas scripting signatures and other important information are kept secret among two stakeholders. This secret information is only revealed to verify publicly when a payment channel is closed. Hence, if Payer wants to prove to *Validators* that payment has successfully paid, Payer has to disclose transaction's scripting signatures and other confidential information. Attackers can make use of these to drain out all coins in this channel. Therefore, it raises a problem, how to prove that Payer successfully sends a payment to Payee without disclosing secret information.

In the Lightning Network, Payee is required to create an invoice to inform a network and also to prepare fundamental procedures for an upcoming payment. An invoice, in a design of Lightning Network, consists of some necessary information [28]. But, the thesis points out only three fundamental elements.

- A *pay_req* ($\mathcal{H}_{pr}$): a hash of an invoice that Payer can use for making a payment.
- A *value*: a total amount of coins that Payer has to pay for a request.
- An *r_preimage*: a confidential scripting proof of releasing funds among two parties in the channel. Before payment, this can be shown only on the Payee's side through an RPC call, *lookupinvoice*[12]

When Payer successfully pays for a request, Payer receives the *r_preimage* from Payee through scripting HTLC contract. This *r_preimage* proves that Payer completely releases the funds, and Payee successfully receives and updates new ownership of transferring funds from Payer. However, this factor is unable to verify publicly by *Validators*. The scheme solves this problem by adding one essential element, a *pay_confirm* (χ). The Lightning Network provides two RPC calls, one for signing a message and the other one for verifying a signature, *signmessage*[13] and *verifymessage*[14] respectively. Taking advantage of these supporting RPC calls, the *pay_confirm* (χ) is created by applying the *signmessage* RPC call on a *pay_req* ($\mathcal{H}_{pr}$) and a secret message ($\mathcal{M}_s$), $\chi = signmessage(\mathcal{H}_{pr}, \mathcal{M}_s)$. Similar to the *r_preimage*, this *pay_confirm* (χ) is also added into an invoice and is kept secretly. Once Payee successfully receives a payment from Payer, both *r_preimage* and *pay_confirm* (χ) are sent to Payer. In the scheme, Payer uses *r_preimage* to prove with the Lightning Network, whereas the *pay_confirm* (χ) is used to prove with *Validators* that an invoice has been completely paid. *Validators* or any node in the Lightning Network can easily verify the *pay_confirm* (χ) as long as the essential information is provided. By providing a *pay_confirm* (χ), a *pay_req* ($\mathcal{H}_{pr}$), and a secret message ($\mathcal{M}_s$), the *verifymessage* function can decrypt to acquire a signer's public address.

[12]https://api.lightning.community/#lookupinvoice
[13]https://api.lightning.community/#signmessage
[14]https://api.lightning.community/#verifymessage

In the following part, the thesis presents all steps of Phase 3 that is how the $\mathcal{LC}_2$ pays a cash-out request in the Lighting Network and updates the total balance of *Wallet Contract* ($\mathcal{W}_2$) in the $\mathcal{VMC}$ (see Algorithm 11). The scheme assumes Payer and Payee have already joined into two different *GroupTrade* contracts ($\mathcal{G}_1$, $\mathcal{G}_2$).

Algorithm 11 Pay Cash Out and Update Tokens - Option 1

procedure PAY (*cash out*, $\mathcal{L}_1$, $\mathcal{H}_{pr}$, $\mathcal{M}_s$, Δ) AND UPDATE $\sigma = \Delta$ TOKENS
- Step 1: Client accesses a *GroupTrade* ($\mathcal{G}_1$) to retrieve essential information ($\mathcal{L}_1$, $\mathcal{H}_{pr}$, $\mathcal{M}_s$)
- Step 2: Client makes a payment using a providing *pay_req* hash ($\mathcal{H}_{pr}$)
- Step 3: Client receives a *pay_confirm* (χ) after successfully paid
- Step 5: Client broadcasts a request (*pay_k*, $\mathcal{H}_{pr}$, χ, Δ, $\mathcal{C}_1$, $\mathcal{G}_1$) to collect signatures
- Step 6: Upon receiving a request (*pay_k*), Validators verify a request
  if $\mathcal{R}(\mathcal{H}_{pr}) = false$ **then**
    Get cash-out request's information using $\mathcal{C}_1$, $\mathcal{G}_1$
    Decode a *pay_req* hash ($\mathcal{H}_{pr}$) to compare with the cash-out request's information
    if $(\sigma = \Delta) = true$ and $\mathcal{L}_1 = true$ **then**
      if $\mathcal{L}_1 = verifymessage(\mathcal{H}_{pr}, \chi, \mathcal{M}_s)$ **then**
        Reply proof $\gamma = Sign(m_{rep})$ with $m_{rep} = (\text{"Pay"}, \mathcal{H}_{pr}, \Delta, \mathcal{C}_2)$
      end if
    end if
  end if
- Step 7: Upon receiving a reply, Client saves into its list $\mathcal{LS}$
  if $len(\mathcal{LS}) = \alpha$ with α = (3,..,n) **then**
    Move to Step 8
  end if
- Step 8: Client calls *pay()* of *GroupTrade* ($\mathcal{G}_2$) with proof in $\mathcal{L}$
- Step 9: A contract ($\mathcal{G}_2$) sends a request with signatures $(\gamma_1, \gamma_2, ...)$ and Payee's info ($\mathcal{C}_1$) to ($\mathcal{T}$)
  if $(\gamma_1, \gamma_2, ...) = true$ **then**
    Save $\mathcal{H}_{pr}$ in *Record Contract* ($\mathcal{R}$)
    Internally request to mint Δ Tokens for $\mathcal{C}_2$ in *Token Contract* ($\mathcal{T}$)
    Internally request to burn Δ Tokens for $\mathcal{C}_1$ in *Token Contract* ($\mathcal{T}$)
  end if
end procedure

To pay a cash-out request, Payer ($\mathcal{C}_2$) accesses a *GroupTrade* ($\mathcal{G}_1$) to get a Payee's address in the Lightning Network ($\mathcal{L}_1$), a hash of *pay_req* ($\mathcal{H}_{pr}$), and a secret message ($\mathcal{M}_s$) for making a

payment. Notably, the cash-out request's accessibility is only for Payer ($\mathcal{C}_2$) and *Validators* since the contract $\mathcal{G}_1$ locks and binds this request to Payer after claiming to pay. This design helps to prevent a replay attack in the case of transferring funds using multi-hop payment in the Lightning Network. In this case, the *pay_confirm* (χ) is transferred along the route. If malicious nodes on the route knew the *pay_req*, and could also access a smart contract to retrieve request information, they would claim that they had already paid a cash-out request so that they could also demand to add Tokens.

By using a providing *pay_req* hash ($\mathcal{H}_{pr}$), Payer's node ($\mathcal{LC}_2$) in the Lightning Network pays an invoice using the supporting RPC call, *sendpayment*[15]. In return, Payer obtains the *pay_confirm* (χ) when a Payee's node ($\mathcal{LC}_1$) successfully receives full payment of this invoice. Upon receiving the *pay_confirm* (χ), Payer's node ($\mathcal{C}_2$) in the $\mathcal{VMC}$ broadcasts a request (*pay_k*, $\mathcal{H}_{pr}$, χ, Δ, $\mathcal{C}_1$, $\mathcal{G}_1$) to collect signatures from *Validators*. And a process of attestation starts taking place. To prevent a replay attack, *Validators* accesses a *Record Contract* ($\mathcal{R}$) to query $\mathcal{H}_{pr}$ in the case it was claimed before. If a hash $\mathcal{H}_{pr}$ is a new one, *Validators* then decode it to compare with information that is retrievd from a cash-out request. When an amount (Δ) and a Payee's address ($\mathcal{L}_1$) are matched, they move on checking whether the Payer successfully pay the request by using the supporting RPC call, *verifymessage*, on $\mathcal{H}_{pr}$, χ, and $\mathcal{M}_s$ to retrieve a signer's address to compare with Payee's address ($\mathcal{L}_1$). Once all verification steps are passed, Payer's node ($\mathcal{C}_2$) can obtain signatures of endorsement $H(Pr)$ from *Validators* so that Payer can apply for updating balance (see Algorithm 11).

4.4.2 Paying and Updating Tokens Using Option 2

Option 2 is a method to pay a cash-out request using Bitcoin's chain. Payer applies for updating Token in his/her *Wallet Contract* ($\mathcal{W}_2$) in the $\mathcal{VMC}$. This option also assumes both nodes, Payer and Payee, have already joined into two different *GroupTrade* contracts ($\mathcal{G}_1$, $\mathcal{G}_2$). Similar to Option 1, Payer's node ($\mathcal{C}_2$) in the $\mathcal{VMC}$ must claim to pay a cash-out request before accessing a *GroupTrade* ($\mathcal{G}_1$) to query the request's information. Instead of transferring funds to *Reserves* ($\mathcal{RS}$), Payer

[15] https://api.lightning.community/#sendpayment

sends coins to Payee's address ($\mathcal{B}_1$) in the Bitcoin's chain. The cross-chain processes to claim and to update the wallet's balance (Algorithm 12) are almost similar to the procedure of wallet initialization, and the request to add Tokens (see Algorithm 6, Algorithm 9).

Algorithm 12 Pay Cash-Out and Update Tokens - Option 2

procedure PAY (*cash out*, $\mathcal{B}_1$, Δ) AND UPDATE $\sigma = \Delta$ TOKENS
- Step 1: Client self-creates a Raw Transaction ($\mathcal{RT}$)
- Step 2: Client generates a Sign Raw Transaction (Φ) with $\Phi = (\mathcal{RT}, \mathcal{B}_{prk})$
- Step 3: Client submits $\mathcal{H}_S = Sign(\Phi, \mathcal{C}_{prk})$ to *Record Contract* ($\mathcal{R}$) as a pre-claim
- Step 4: Client transfers funds to $\mathcal{B}_1$ and waits until $\mathcal{T}_{id}$ is mined
- Step 5: Client broadcasts a request (*pay_k*, $\mathcal{T}_{id}$, Φ, Δ, $\mathcal{C}_1$, $\mathcal{G}_1$) to collect signatures
- Step 6: Upon receiving a request (*pay_k*), Validators verify a request
  if $\mathcal{R}(\mathcal{T}_{id}) = false$ **then**
    Get transaction info using $\mathcal{T}_{id}$
    Get cash out request information in $\mathcal{G}_1$
    if $(\sigma = \Delta) = true$ and $\mathcal{B}_1 = true$ and $\Phi =$ true **then**
      Get pre-claim info from $\mathcal{R}$
      if $\mathcal{C}_2 = ecrecover(\mathcal{H}_S, \Phi)$ **then**
        Reply proof $\gamma = Sign(m_{rep})$ with $m_{rep} =$ ("Pay", $\mathcal{T}_{id}, \Delta, \mathcal{C}_2$)
      end if
    end if
  end if
- Step 7: Upon receiving a reply, Client saves into its list $\mathcal{LS}$
  if $len(\mathcal{LS}) = \alpha$ with $\alpha =$ (3,..,n) **then**
    Move to Step 8
  end if
- Step 8: Client calls *pay()* of *GroupTrade* ($\mathcal{G}_2$) with proof in $\mathcal{L}$
- Step 9: A contract ($\mathcal{G}_2$) sends a request with signatures ($\gamma_1, \gamma_2, ...$) and Payee's info ($\mathcal{C}_1$) to ($\mathcal{T}$)
  if $(\gamma_1, \gamma_2, ...) = true$ **then**
    Save $\mathcal{T}_{id}$ in *Record Contract* ($\mathcal{R}$)
    Internally request to mint Δ Tokens for $\mathcal{C}_2$ in *Token Contract* ($\mathcal{T}$)
    Internally request to burn Δ Tokens for $\mathcal{C}_1$ in *Token Contract* ($\mathcal{T}$)
  end if
end procedure

5 More Discussion of the Proposed Scheme

In this section, I provide an additional discussion of the thesis's contribution. Concurrently, I also describe the threat model and analyze the securities of the proposed protocol.

5.1 Trade vs. Swap

Unlike Kyber Network and Liquality, the contribution in this thesis is to provide a protocol and a blockchain-based platform that can link and connect other ones to promote a universal cross-chain trading network. Trade of assets and swaps can have a close meaning. However, trade still has a broader scope of transactions than simple swaps, where the swap is a particular case. After coins are converted to tokens using this proposed network, any kind of trade deals can be supported on top using smart contracts, including swap. For example, Client A transfers 1 Bitcoins to the address of *Reserve* ($\mathcal{RS}$). In return, Client A receives an equivalent of Token that is 10^8 Bit-Tokens. In the meantime, Client B also transfers 1 Ether to a designated system-reserved wallet address in requesting 10^{18} Ether-Tokens. The value of these tokens is not equal so that it can be traded equally. Instead, in case of a swap, Client A and Client B can make their smart contract deal in exchanging Token coins and link to their *GroupTrade* contract. Not only limited to swap, but the trading network also promotes group-trades and payment-networks by using the offering backbone smart contracts. Clients can transfer their tokens, same or different types of tokens, within a group or from group-to-group as long as they agree on their trade deals.

5.2 Security Properties

Impartial witness role: The cross-chain trading network is a series of cross-chain activities that require smart contracts as record-keeping services and *Validators* as on-chain operators to take responsibility as a notary of the cross-chain activities. In the proposed scheme, *Validators* do not intervene or affect any trades among clients. They take responsibility as border gatekeepers to verify side-chain transactions before giving an endorsement to allow clients to cross a border

between chains, to join/exit a trading network, or to attest activities that relate to cross-chain procedures.

Replay attack: In receiving a corresponding amount of Tokens in the trading network, clients are required to transfer their funds to designated system-reserved addresses. The proposed model considers a presence of replay attacks in which a transaction could be claimed twice by one client or by multiple clients since system-reserved addresses are public and transaction id can be queried publicly and easily. The replay attacks can be prevented by applying pre-claim steps (Steps 1-3) before sending a transaction in the proposed protocols. The secret information of a transaction can be determined beforehand by only an owner of a transaction. It then is encrypted using a cryptography method to generate a signature before submitting to *Record Contract* as a pre-claim proof. By revealing this encrypted data, a client can prove that he/she is a transaction's sender. These steps work effectively on both UTXO and Account/Balance based models. In the Account/Balance based model, a client specifies an address, storing its funds on a *PSC*, that expects to be spent. By applying ECDSA cryptography, a transaction's issuer can easily prove a transaction's ownership through its signature. Likewise, the UTXO model, as discussed in previous sections, is hard to detect a transaction's sender by the fact that this model keeps generating a new address to store updating funds in every transaction. Thus, it seems to be impossible to keep track of funding addresses of arbitrary nodes in a network. As a result, pre-claim steps would help to identify the creator of a transfer-fund transaction. When multiple clients concurrently claim on one transaction, pre-claim steps can resolve the conflict.

Furthermore, the scheme proposes two layers of verification to protect a cross-chain trading system from replay attacks. *Validators* execute the first layer of verification, and a smart contract does the other one. A client sends a request, including essential information, i.e., a transaction id, to collect signatures of endorsement. Before giving an endorsement to a request, *Validators* must verify whether providing transaction id has been claimed previously by checking it with a record list in the *Record Contract* ($\mathcal{R}$). Once a client gathers enough required signatures, its request to *GroupTrade* contract ($\mathcal{G}$) must include transaction id along with submitting signatures of endorsement for the

second layer of verification. When all of these are verified successfully, transaction id and signatures are saved in the *Record Contract* ($\mathcal{R}$). As a result, a replay attack is not possible when applying two layers of verification.

Man-in-the-middle attack: Along with *Replay attacks*, the scheme also considers the presence of *Man-in-the-middle attacks*. For security reasons, sensitive data are required to sign by a user using ECDSA cryptography. For example, pre-determined transaction information, in pre-claim steps (Step 1-3), is encrypted using a user's private key to generate a signature. This information is kept as a secret until it is claimed and revealed by a user. In addition, the validation process, which is done by *Validators*, is also accompanied by signatures. By this design, *Man-in-the-middle attacks* can be prevented.

Validation credibility: Smart contracts cannot automatically verify side-chain transactions. This attestation should rely on impartial witnesses. Thus, the credibility of validating processes is considered as a top priority. The signatures of endorsement are strictly required to be provided only by authorized *Validators*. By providing a Genesis *Record Contract*, the authorized *Validators* can be verified. Upon receiving endorsements, clients must check the validity of these signatures. Even in the worst case that clients skip checking these signatures, the validation credibility is additionally forcibly complied by the second layer of verification that is the smart contract.

6 Experiment Setup and Result

6.1 Network and Configuration

In this section, I provide how I set up an experiment to demonstrate the protocols of the cross-chain trading scheme. The network is created by establishing a private test-net among eight nodes (Figure 6.12):

- Three *Validators* verify cross-chain procedures running on three separate servers using Amazon Cloud Server (AWS). Each server has a configuration: Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz, 8GB RAM, and running Ubuntu 18.04,
- The *Reserve* ($\mathcal{RS}$) server is also created by running another AWS server. This instance has a configuration: Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz, 4GB RAM, and running Ubuntu 18.04,
- Two node clients ($\mathcal{C}_1$, $\mathcal{C}_2$) connect to the network using a local computer that has configuration: Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz, 16GB RAM, and running Ubuntu 18.04,
- Another two clients ($\mathcal{C}_3$, $\mathcal{C}_4$) also connect to the network using another desktop computer that has configuration: Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz, 16GB RAM, and running Ubuntu 18.04.

Distributed blockchain-based applications include:

- Ethereum: the smart contracts are built on top of a private test-net of Ethereum's blockchain. The experiment uses the Go Ethereum application (Geth v1.8.16) that acts as the main chain to demonstrate. This code-base has been modified to add more protocols so that the communication among clients within a network as well as cross-chain procedures can be effectively implemented,
- Bitcoin: The experimental network uses Bitcoin Core RPC client (v0.18.0.0) to set up a private side-chain distributed ledger. All RPC methods, supported by Bitcoin Core, are utilized within this project without modification. Instead, the additional protocols that support

cross-chain procedures are implemented to Lightning Network Daemon,

- Lightning Network: This is a second side-chain distributed ledger. A private test-net of clients is built by using Lightning Network written in Golang (lnd v0.5.1). Along with using the supporting RPCs, additional command-line-interfaces, e.g., *lncli deploy*, *lncli join*, and *lncli deposit* (see Appendix), and protocols are implemented to support the cross-chain scheme.

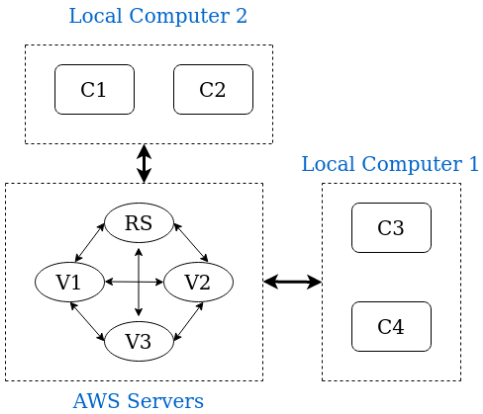

Figure 6.12: The high-level architecture of a cross-chain network using in experiment

Along with cross-chain activities, there are some processes, as provided in previous sections, that can be executed within the $matchcalVMC$, e.g., sending Tokens, querying balance, and submitting a cash-out request. To support these in-chain processes, I also create a local web GUI (Figure 6.13) instead of using RPCs and CLIs. The local web GUI includes:

- Send Tokens: A function that helps transfer Tokens among clients within a group or across groups as long as both sender and receiver have already joined the trading network,
- Cashout Request: A function that helps submit a cash-out request and add to a *GroupTrade* contract,
- Claim To Pay: this function helps Payer claim to pay for a cash-out request. It accesses a *GroupTrade* contract, which saves a cash-out request, to update Payer's address and to lock request's accessibility,
- Wallet Balance: Clients can use this function to query the balance of an arbitrary account within the network,

- Wallet Address: this function returns a *Wallet Contract* ($\mathcal{W}$) corresponding to only one Ethereum account's address ($\mathcal{C}$) within a *GroupTrade* ($\mathcal{G}$),
- Channel Address: this function returns a *GroupTrade* address ($\mathcal{G}$) that a *Wallet Contract* ($\mathcal{W}$) has joined,
- Query Cash-out Info: this function helps Payer to query cash-out request information. Notably, after a cash-out request is claimed to pay, it can be accessed by only Payer and *Validators*.

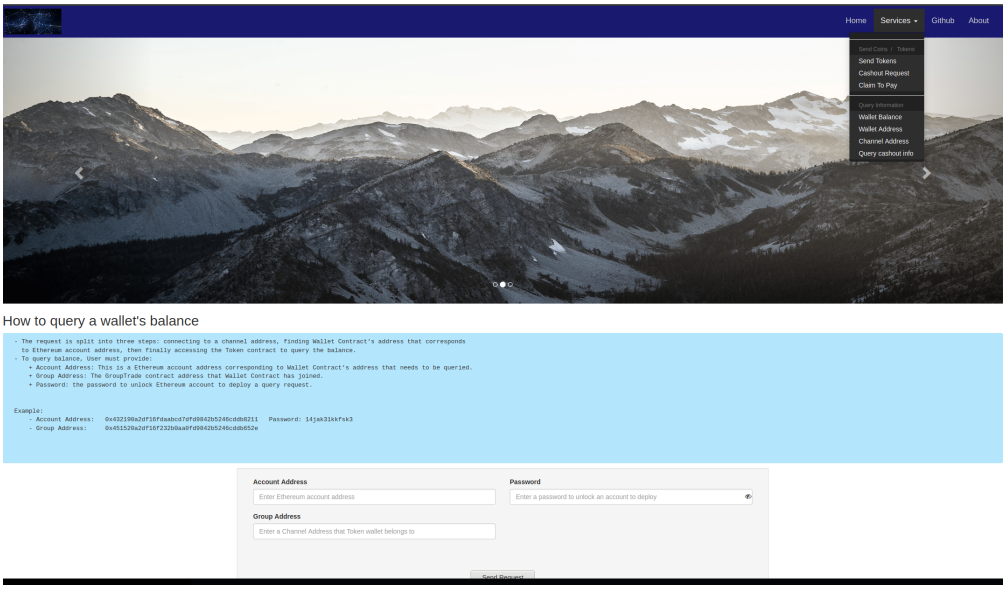

Figure 6.13: The local Web Application supports in-chain activities

6.2 Experiment Results

To demonstrate proposed protocols of the cross-chain trading network, the experiment establishes two *GroupTrades* ($\mathcal{G}_1$, $\mathcal{G}_2$) that have two node clients in each group. Two clients, $\mathcal{C}_1$ and $\mathcal{C}_3$, join the *GroupTrade* ($\mathcal{G}_1$). The other two clients, $\mathcal{C}_2$ and $\mathcal{C}_4$, join *GroupTrade* ($\mathcal{G}_2$). In the following part, the thesis provides the results of creating wallets of each client, sending tokens among clients, and cashing-out by running experimental scenarios.

- *Creating Wallets:* In this experiment, each client deploys or joins a *GroupTrade* by using different proposed methods. The scenarios as following:

- First, the client $\mathcal{C}_1$ starts the *GroupTrade* ($\mathcal{G}_1$) by initializing its wallet ($\mathcal{W}_1$) $\sigma = 0$ Token (see Figure 6.15),
- Second, the client $\mathcal{C}_3$ joins the *GroupTrade* ($\mathcal{G}_1$) in accompany with requesting to initialize its wallet ($\mathcal{W}_3$) $\sigma = 50$ BTC $= 50 * 10^8$ Tokens (Algorithm 6),
- On the other hand, the *GroupTrade* ($\mathcal{G}_2$) is created by $\mathcal{C}_2$ using Algorithm 6 to initialize the wallet ($\mathcal{W}_2$) $\sigma = 100 * 10^8$ Tokens (100 BTC),
- Finally, the client $\mathcal{C}_4$ initializes the wallet ($\mathcal{W}_4$) $\sigma = 80$ BTC $= 80 * 10^8$ Tokens, then joins the *GroupTrade* ($\mathcal{G}_2$) using Algorithm 7.

Figure 6.14: The client $\mathcal{C}_1$ requests to join a network and starts *GroupTrade* $\mathcal{G}_1$ with *0* Token

The client $\mathcal{C}_1$ starts the *GroupTrade* ($\mathcal{G}_1$) using CLI "lncli deploy." This RPC call handles a procedure of requesting signatures for attestation, creating a wallet, deploying a Group Trade. This command sends a request *validate_0* to collect signatures of endorsement from *Validators*. Once gathering enough signatures, a *Wallet Contract* ($\mathcal{W}_1$) is created. Then, the *GroupTrade* ($\mathcal{G}_1$) is deployed and adds $\mathcal{W}_1$ into $\mathcal{G}_1$ as well as requesting to mint *0* Token in the *Token Contract*. If there is no error, the client $\mathcal{C}_1$ receives two addresses in return. The first address is a Wallet Contract address, and the second one is an address of GroupTrade $\mathcal{G}_1$ (see Figure 6.14).

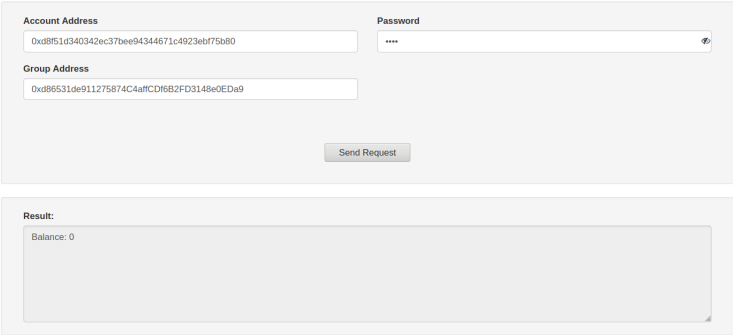

Figure 6.15: The balance of $\mathcal{C}_1$ after successfully deploys the *GroupTrade* ($\mathcal{G}_1$)

After the *GroupTrade* $\mathcal{G}_1$ is formed up, the client $\mathcal{C}_3$ starts joining this group. Unlike the client $\mathcal{C}_1$ that requests to join with *0* Token, the client $\mathcal{C}_3$ has to deposit coins into *Reserve* ($\mathcal{RS}$) and to submit pre-claim information before requesting to join *GroupTrade* $\mathcal{G}_1$ (Figure 6.16). The command "lncli deposit" takes responsibility for preparing pre-claim information, transferring funds into $\mathcal{RS}$, and submitting the pre-claim into the *Record Contract*. Similar to "lncli deploy," the command "lncli join" takes responsibility in requesting to join the *GroupTrade* ($\mathcal{G}_1$) as well as requesting to initialize $\mathcal{W}_2$ several Tokens, $\sigma = 50 * 10^8$ (Figure 6.17).

```
quangtran8588@QuangTran:~$ lncli deposit --localhost=127.0.0.1:5063 --testnet=-r
egtest --rpcuser=client3 --rpcpass=test --rpcport=7003 --receiver=2N5kVnetbuAvt3
qngiLyPdZaGdSuNbXMp4J --amt=50 --contract=0x4CF79c9268691c80eC959791847Ae37A1401
2E47 --eth_addr=0x95b3e0ea2d93f9fd7507ea85bb074ca208726f9c
Start depositing bitcoins ......
Claim deposit request is deployed successfully
{
        "Txid": "a8efa04da16d7f350d4d48bc4170075ef2f899a79bc06710eb8aaaa7180d147
1",
        "Sign_Raw_Hex": "02000000000101a5bf27aa7c25e5f7b6a786843f16ca40642508c1e
6fe161ac419bd6e25e8f02c0100000017160014b823092c4772c80106348a06f1f6bc2bc048e6b9f
fffffff0200f2052a0100000017a914892ac2d3b9ecf06463a70990acc750af5a0365698738ee282
60800000017a9143227b91a3e9d386b1526e47110aef1b14b13efcb870247304402207841580bda
f1a4f9fbc9b029734b2110a20464082cce916ccc920043df94fd3022064bbccdc2e6f1f788ec18d8
a74240eda5651bfc53b24ea183f39f6c58d8edaa7012102394546d3f58ce4734953cf9db3386168b
60b9e40765a6f9af93bd4de9173fb0000000000"
}
quangtran8588@QuangTran:~$ lncli join --localhost=127.0.0.1:5063 --eth_addr=0x95
b3e0ea2d93f9fd7507ea85bb074ca208726f9c --opt_code=1 --tx=a8efa04da16d7f350d4d48b
c4170075ef2f899a79bc06710eb8aaaa7180d1471 --signtx=02000000000101a5bf27aa7c25e5f
7b6a786843f16ca40642508c1e6fe161ac419bd6e25e8f02c0100000017160014b823092c4772c80
106348a06f1f6bc2bc048e6b9fffffffff0200f2052a0100000017a914892ac2d3b9ecf06463a7099
0acc750af5a0365698738ee28260800000017a9143227b91a3e9d386b1526e47110aef1b14b13efc
b870247304402207841580bdaf1a4f9fbc9b029734b2110a20464082cce916cccc920043df94fd30
22064bbccdc2e6f1f788ec18d8a74240eda5651bfc53b24ea183f39f6c58d8edaa7012102394546d
3f58ce4734953cf9db3386168b60b9e40765a6f9af93bd4de9173fb0000000000 --amt=50000000
00 --contract_addr=0xd86531de911275874C4affCDf6B2FD3148e0EDa9
Start joining a contract......
Contract is deployed successfully.
Wallet address: 0xbdB16CE8fE83f94B4433E85527d3c5d3D29683d9
Node joins the contract channel successfully.
quangtran8588@QuangTran:~$
```

Figure 6.16: The client $\mathcal{C}_3$ joins *GroupTrade* $\mathcal{G}_1$ with requesting $\sigma = 50 * 10^8$ Tokens

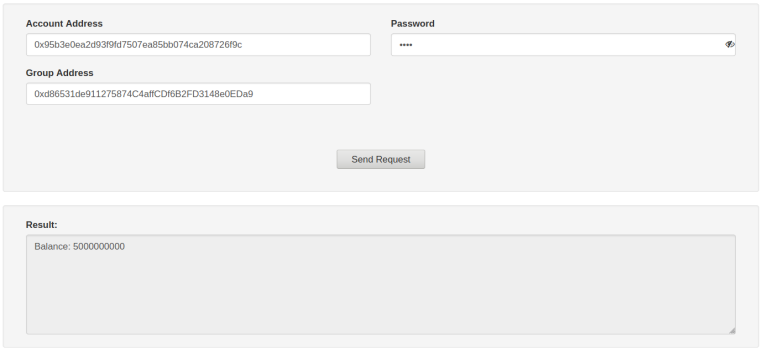

Figure 6.17: The balance of $\mathcal{C}_3$ after successfully joins the *GroupTrade* ($\mathcal{G}_1$)

The second group, *GroupTrade* ($\mathcal{G}_2$), is created by the client $\mathcal{C}_2$ using Algorithm 6. As the same as client $\mathcal{C}_3$, the client $\mathcal{C}_2$ is also required to transfer funds into *Reserve* ($\mathcal{RS}$) before deploying a *GroupTrade* $\mathcal{G}_2$ since this client also requests to create a *Wallet Contract* with some Tokens (Figures 6.18, 6.19).

```
jasontran@jasontran-HP-ENVY-Notebook:~$ lncli deposit --localhost=127.0.0.1:5062
 --testnet=-regtest --rpcuser=client2 --rpcpass=test --rpcport=7002 --receiver=2
N5kVnetbuAvt3qngiLyPdZaGdSuNbXMp4J --amt=100 --contract=0x4CF79c9268691c80eC9597
91847Ae37A14012E47 --eth_addr=0x74e57164ca24416929f48b9722e1b28f257aae2c
Start depositing bitcoins ......
Claim deposit request is deployed successfully
{
        "Txid": "ea07748ac47f424d810add2bb2d0f5dc76e5651f76f9744466ee5685a2ba6e2
b",
        "Sign_Raw_Hex": "02000000000101b5d2dce68a5535c4400988d4ba7585abf19d8627d
ccbf742037b96685fd579f001000000171600144feb705e9ed163f6979fc57c31c7b5375e0084abf
fffffff0200e40b540200000017a914892ac2d3b9ecf06463a70990acc750af5a03656987d03623f
c0600000017a914187f7dbe238827ed79646fcc0022703ff046a4918702473044022 02ec1696aad0
343c9c2f45e16f9606a02f73286e3472f7c2eaac8c295b99f53c302207b6036ffe3b6a79d7825d15
bf7cb1d16cf5d1b4c4b695a1860f1935c99e651f401210313aecf2cd0d492473de698934b8fa0db0
633ec118f879e1de3b17c76c5502abf00000000"
}
jasontran@jasontran-HP-ENVY-Notebook:~$ lncli deploy --localhost=127.0.0.1:5062
--token_addr=0x67817A83645726d81e92afcb69A30Cb2FD6D4Ee6 --verifier_addr=0x83a399
5D1c493174630697d135a195c270932458 --collateral_addr=0x4CF79c9268691c80eC9597918
47Ae37A14012E47 --eth_addr=0x74e57164ca24416929f48b9722e1b28f257aae2c --opt_code
=1 --tx=ea07748ac47f424d810add2bb2d0f5dc76e5651f76f9744466ee5685a2ba6e2b --signt
x=02000000000101b5d2dce68a5535c4400988d4ba7585abf19d8627dccbf742037b96685fd579f0
01000000171600144feb705e9ed163f6979fc57c31c7b5375e0084abfffffffff0200e40b54020000
0017a914892ac2d3b9ecf06463a70990acc750af5a03656987d03623fc0600000017a914187f7dbe
238827ed79646fcc0022703ff046a4918702473044022 02ec1696aad0343c9c2f45e16f9606a02f7
3286e3472f7c2eaac8c295b99f53c302207b6036ffe3b6a79d7825d15bf7cb1d16cf5d1b4c4b695a
1860f1935c99e651f401210313aecf2cd0d492473de698934b8fa0db0633ec118f879e1de3b17c76
c5502abf00000000 --amt=10000000000
Start deploying contract......
Contract is deployed successfully
Contract address: 0xeb1802d42FdA2f8090c02A85C5398213CFA7503e
Contract is deployed successfully
Contract address: 0x5048B91FEb86dEc8AAe46ff21666733f35d3c079
```

Figure 6.18: The client $\mathcal{C}_2$ starts *GroupTrade* $\mathcal{G}_2$ with initializing $\sigma = 100 * 10^8$ Tokens

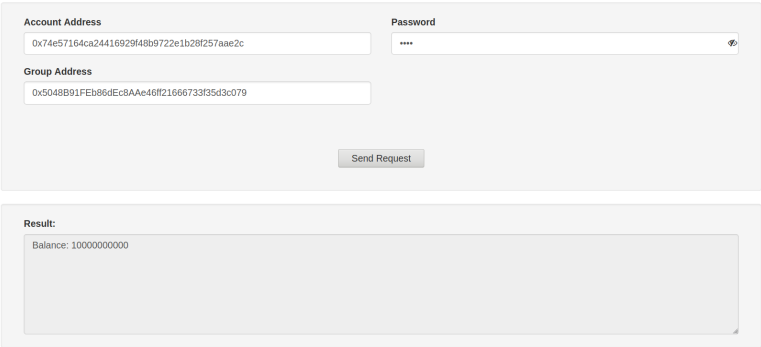

Figure 6.19: The balance of $\mathcal{C}_2$ after successfully create the *GroupTrade* ($\mathcal{G}_2$)

Finally, the client $\mathcal{C}_4$ joins the *GroupTrade* $\mathcal{G}_2$ (Figure 6.21). In this experiment, the client $\mathcal{C}_4$ is a Lightning Network Daemon (LND) node. Instead of using "lncli deposit" to transfer funds to $\mathcal{RS}$, this experiment uses another command that is "lncli sendcoins" (Figure 6.20). This RPC call, which is supported by LND, helps to transfer funds and to finalize a generating transaction on-chain. The project has modified the command so that it can be applied to the cross-chain scheme.

```
jasontran@jasontran-HP-ENVY-Notebook:~/bitcoin_testnet/lnd$ lncli --rpcserver=lo
calhost:10001 --macaroonpath=./data/chain/bitcoin/regtest/admin.macaroon sendcoi
ns --addr=2N5kVnetbuAvt3qngiLyPdZaGdSuNbXMp4J --amt=8000000000 --virtual=1 --loc
alhost=127.0.0.1:5064 --eth_addr=0xa3f42c656cdcff6b362e418b6430fbb452aabdb1 --co
ntract=0x4CF79c9268691c80eC959791847Ae37A14012E47
Claim deposit request is deployed successfully
{
        "txid": "31add9cda449df3a4e73bf6c3844ccb79dae22bde5fc8271c86211f0df610da
6",
        "TxsIn": [
                "ef48865d7aefda87186e1b0014072321913e210f1a948a9e000bd7ce1ee9895
9"
        ]
}
jasontran@jasontran-HP-ENVY-Notebook:~/bitcoin_testnet/lnd$ lncli join --localho
st=127.0.0.1:5064 --eth_addr=0xa3f42c656cdcff6b362e418b6430fbb452aabdb1 --opt_co
de=3 --tx=31add9cda449df3a4e73bf6c3844ccb79dae22bde5fc8271c86211f0df610da6 --txs
in=ef48865d7aefda87186e1b0014072321913e210f1a948a9e000bd7ce1ee98959 --amt=800000
0000 --contract_addr=0x5048B91FEb86dEc8AAe46ff21666733f35d3c079
Start joining a contract......
Contract is deployed successfully.
Wallet address: 0xDBF46Fb664BcBaf93d2c103016581CeC8bAd5450
Node joins the contract channel successfully.
jasontran@jasontran-HP-ENVY-Notebook:~/bitcoin_testnet/lnd$
```

Figure 6.20: The client $\mathcal{C}_4$ joins *GroupTrade* $\mathcal{G}_2$ with initializing $\sigma = 80 * 10^8$ Tokens

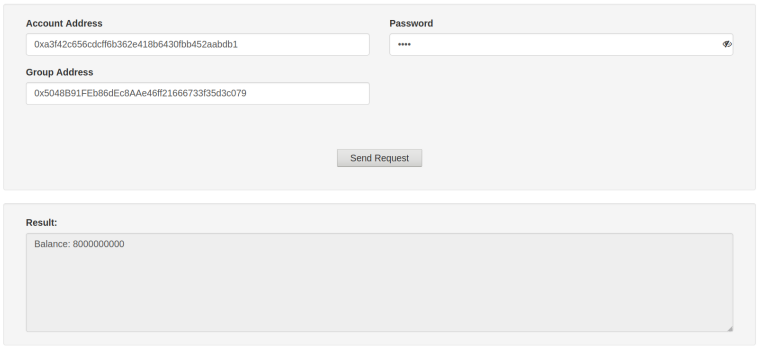

Figure 6.21: The balance of $\mathcal{C}_4$ after successfully join the *GroupTrade* ($\mathcal{G}_2$)

- *Sending Tokens:* Sending and Receiving Tokens are executed by smart contracts within a private Ethereum test-net. Client-to-Client transfer of tokens can be applied within a group or across groups. Instead of using RPC commands as above, clients can utilize a local web application to execute in-chain processes when they have successfully joined the trading network. In demonstrating these features, the experiment executes some basic scenarios as follows:

- The client $\mathcal{C}_3$ transfers to the host of *GroupTrade* ($\mathcal{G}_1$) $2 * 10^7$ Tokens (Figure 6.22). Client-to-Client transferring is applied within a group since both clients, $\mathcal{C}_1$ and $\mathcal{C}_3$, have joined the same *GroupTrade* ($\mathcal{G}_1$). The balance of both clients is updated after this transaction is accepted (Figures 6.23, 6.24).

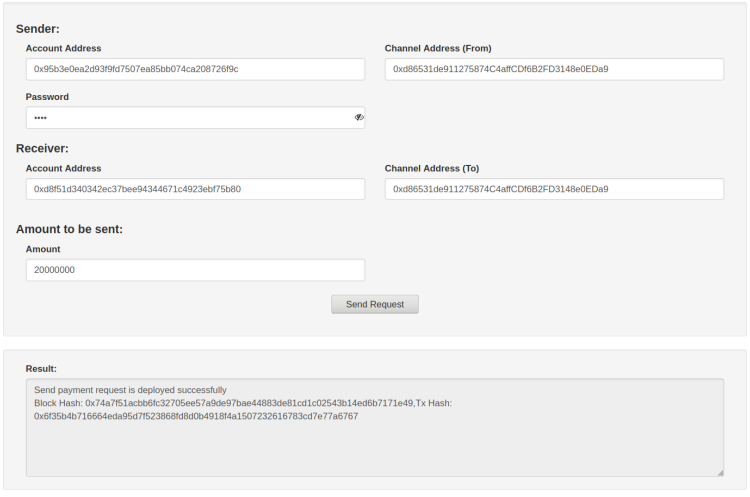

Figure 6.22: The client $\mathcal{C}_3$ sends to $\mathcal{C}_1$ several Tokens $k = 2 * 10^7$ using a local Web GUI

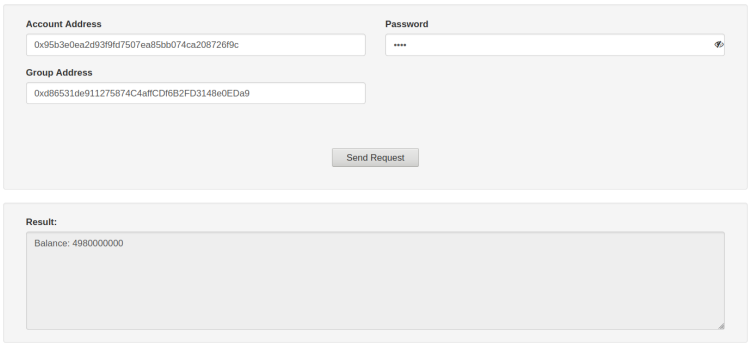

Figure 6.23: The balance of $\mathcal{C}_3$ after transferring Tokens to $\mathcal{C}_1$

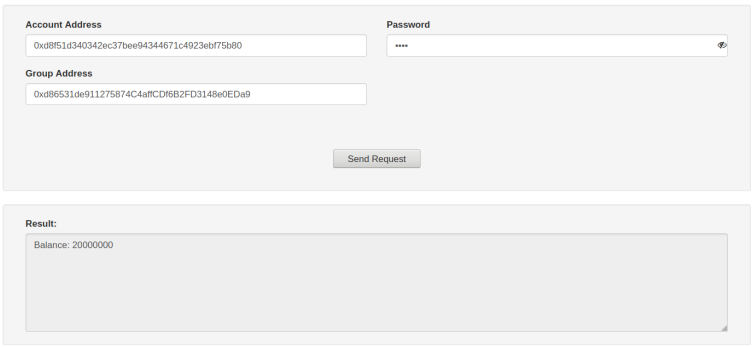

Figure 6.24: The balance of $\mathcal{C}_1$ after receiving Tokens $k = 2 * 10^7$ from $\mathcal{C}_3$

- The client $\mathcal{C}_4$ transfers to the host of *GroupTrade* ($\mathcal{G}_1$) $8 * 10^7$ Tokens (Figures 6.25, 6.26, 6.27). Client-to-Client transferring is applied across groups since $\mathcal{C}_1$ has joined the *GroupTrade* ($\mathcal{G}_1$) while $\mathcal{C}_4$ is a member of *GroupTrade* ($\mathcal{G}_2$)

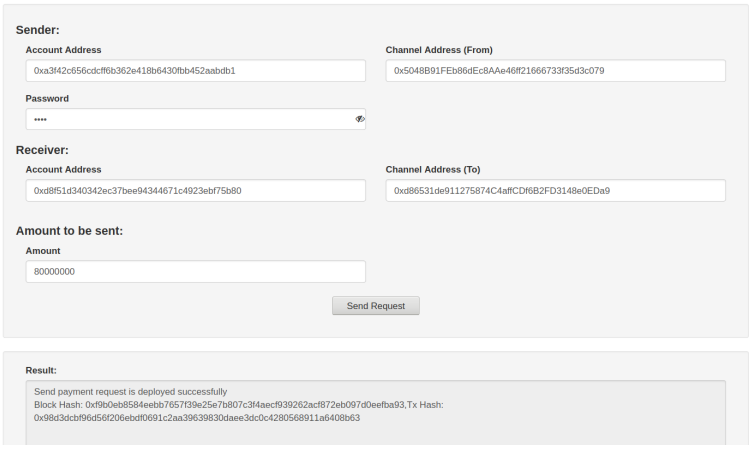

Figure 6.25: The client $\mathcal{C}_4$ sends to $\mathcal{C}_1$ some Tokens $k = 8 * 10^7$ using a local Web GUI

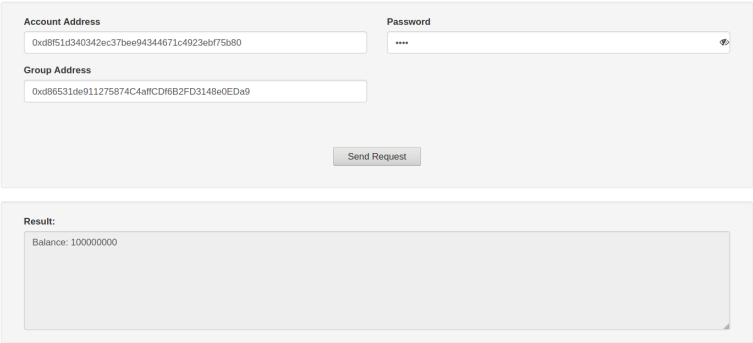

Figure 6.26: The balance of $\mathcal{C}_1$ after receiving Tokens from $\mathcal{C}_4$

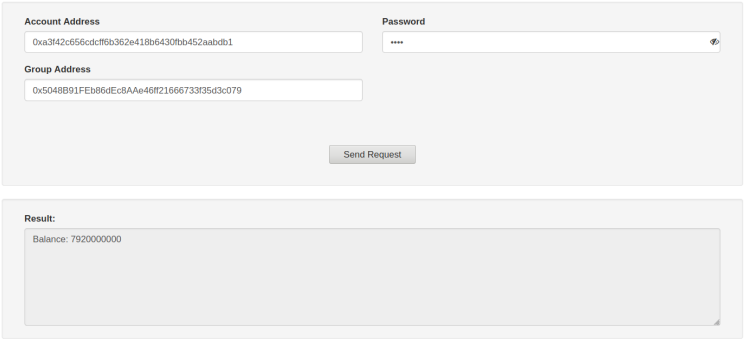

Figure 6.27: The balance of $\mathcal{C}_4$ after transferring Tokens $k = 8 * 10^7$ to $\mathcal{C}_1$

- *Cash-out:* this feature is a cross-chain procedure. As discussed in the previous sections, this procedure can be broken down into three phases. To demonstrate these phases, the experiment creates a scenario as follows:

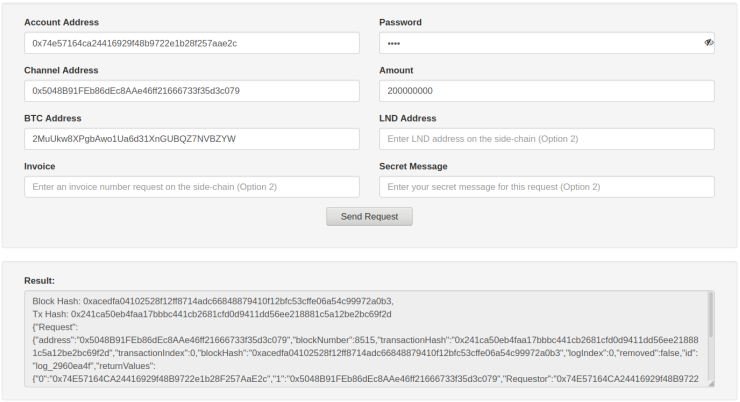

Figure 6.28: The client $\mathcal{C}_2$ requests to cash-out $2 * 10^8$ Tokens = 2 BTC to *GroupTrade* ($\mathcal{S}_2$)

- The client $\mathcal{C}_2$ submits a cash-out request into a *GroupTrade* ($\mathcal{G}_2$). This process is an in-chain process. Thus, a local web GUI can be utilized (see Figure 6.28). After a request is successfully submitted into the *GroupTrade* ($\mathcal{G}_2$), this request can be accessed only by *Validators* before the Payer claims it. Even an owner of this request, the client $\mathcal{C}_2$, cannot access to query the request information (Figures 6.29, 6.30).

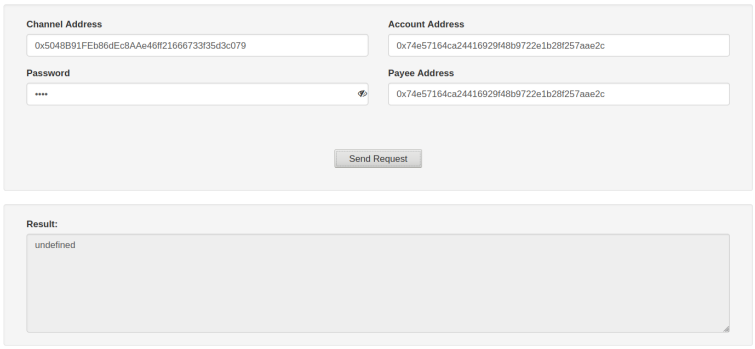

Figure 6.29: The client $\mathcal{C}_2$ fails to query a cash-out request

- The client $\mathcal{C}_1$ claims to pay the cash-out request by accessing the *GroupTrade* ($\mathcal{G}_2$) to update and to lock the request (Figure 6.31). When this client claims successfully, the *GroupTrade* ($\mathcal{G}_2$) binds the request to an address of the client $\mathcal{C}_1$. From now, this *GroupTrade* contract allows the client $\mathcal{C}_1$ to access the request while it blocks the accessibility from other clients except for *Validators* in querying information.

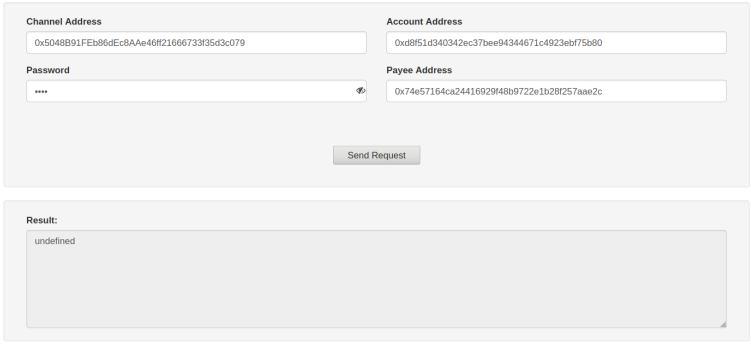

Figure 6.30: The client $\mathcal{C}_1$ fails to query a cash-out request before claiming to pay

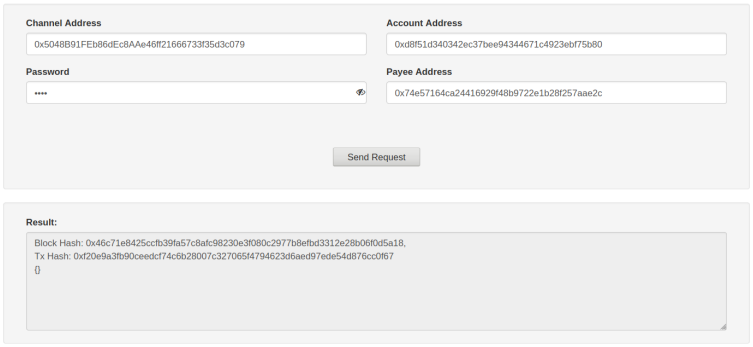

Figure 6.31: The client $\mathcal{C}_1$ claims to pay for a cash-out request

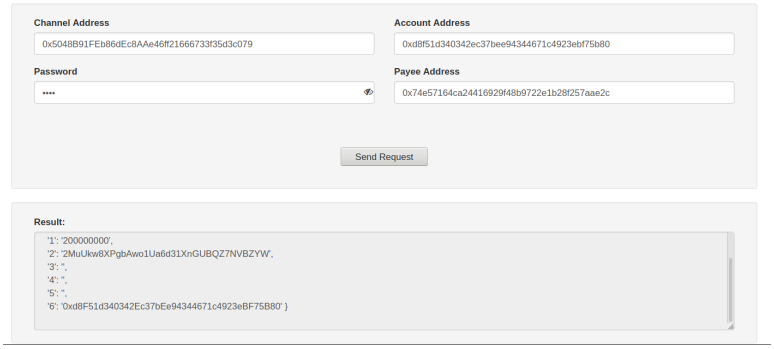

Figure 6.32: The client $\mathcal{C}_1$ succeeds to query a cash-out request after claiming to pay

- After claiming to pay for a cash-out request, the client $\mathcal{C}_1$ accesses the *GroupTrade* ($\mathcal{G}_2$) to query Payee's address for making a payment (Figure 6.32). Similar to transferring funds to *Reserve*, the command "lncli deposit" can be utilized. In this case, a receiver is the one that is queried from the previous step. After executing a previous command, the client $\mathcal{C}_1$ can start sending a request to update the wallet's balance by using a command "lncli updatepayreq" (Figure 6.33). This command takes responsibility for gathering signatures of attestation before requesting to update the balance in a smart contract. Once the payment of a cash-out request is verified successfully by both *Validators* and *Record Contract*, the balance of both clients, $\mathcal{C}_1$ and $\mathcal{C}_2$, is updated accordingly in the *Token Contract* $\mathcal{T}$ (Figures 6.34, 6.35).

```
quangtran8588@QuangTran:~$ lncli deposit --localhost=127.0.0.1:5061 --testnet=-r
egtest --rpcuser=client1 --rpcpass=test --rpcport=7001 --receiver=2MuUkw8XPgbAwo
1Ua6d31XnGUBQZ7NVBZYW --amt=2 --contract=0x4CF79c9268691c80eC959791847Ae37A14012
E47 --eth_addr=0xd8f51d340342ec37bee94344671c4923ebf75b80
Start depositing bitcoins ......
Claim deposit request is deployed successfully
{
        "Txid": "a57fc59c4c384979f92c4127ed4e880f44bd87cdafe6f03506b8d0741e6cb94
1",
        "Sign_Raw_Hex": "02000000000101a391f999f1fcbb4d2e55e7d7d2af991d09ffd6709
e66ab2040ed8326b98c0f960000000017160014652 4fa4742e1c5f0f8f317928a547db09fffa57af
ffffffff0200c2eb0b0000000017a914187f7dbe238827ed79646fcc0022703ff046a4918768774f9
80b00000017a914f6f67b8e5886a34f882c8371e108d2dcc5a55c7e870247304402202 3914b6e066
2880ee5faea0c4d3173ecfc96bd39b2a8fb1ae350f75064079d7402205ec49a069f51133cc54bdfe
e081704ec34cf50a451f1603675250218550e4698012103b9da579f74563cde31e9a255ff0800011
232683769a8ba61efc2a71bc3218a9000000000"
}
quangtran8588@QuangTran:~$ lncli updatepayreq --localhost=127.0.0.1:5061 --sende
r_eth=0xd8f51d340342ec37bee94344671c4923ebf75b80 --receiver_eth=0x74e57164ca2441
6929f48b9722e1b28f257aae2c --receiver_btc=2MuUkw8XPgbAwo1Ua6d31XnGUBQZ7NVBZYW --
payreq_channel=0x5048B91FEb86dEc8AAe46ff21666733f35d3c079 --payer_channel=0xd865
31de911275874C4affCDf6B2FD3148e0EDa9 --payreq_channel_type=1 --payer_channel_typ
e=1 --opt_code=1 --tx=a57fc59c4c384979f92c4127ed4e880f44bd87cdafe6f03506b8d0741e
6cb941 --signtx=02000000000101a391f999f1fcbb4d2e55e7d7d2af991d09ffd6709e66ab2040
ed8326b98c0f960000000017160014652 4fa4742e1c5f0f8f317928a547db09fffa57affffffff02
00c2eb0b0000000017a914187f7dbe238827ed79646fcc0022703ff046a4918768774f980b000000
17a914f6f67b8e5886a34f882c8371e108d2dcc5a55c7e870247304402202 3914b6e0662880ee5fa
ea0c4d3173ecfc96bd39b2a8fb1ae350f75064079d7402205ec49a069f51133cc54bdfee081704ec
34cf50a451f1603675250218550e4698012103b9da579f74563cde31e9a255ff0800011232683769
a8ba61efc2a71bc3218a9000000000 --amt=200000000
Start updating payment on virtual chain .......
Updating pay request is deployed successfully
quangtran8588@QuangTran:~$
```

Figure 6.33: The client $\mathcal{C}_1$ pays the cash-out and requests to update the balance

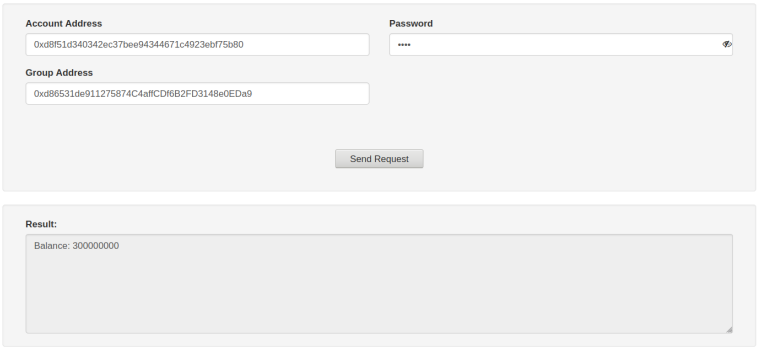

Figure 6.34: The balance of $\mathcal{C}_1$ is updated after making a payment

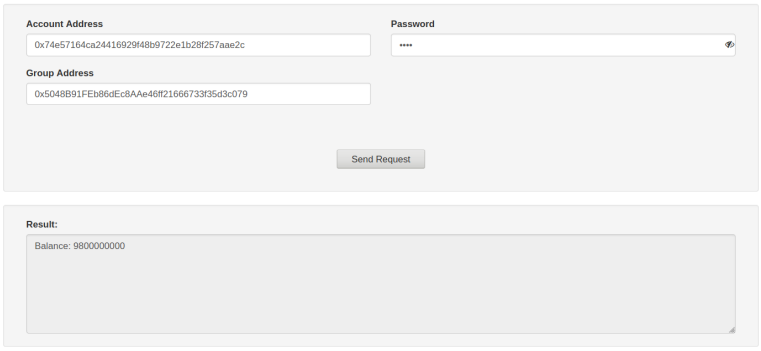

Figure 6.35: The balance of $\mathcal{C}_2$ is updated after receiving a payment

7 Conclusion and Future Works

The described system leverages the use of the blockchain technology to aggregate multiple solitary blockchain platforms and its crypto-currencies to establish a scalable, fair, and decentralized trading network of digital assets. To demonstrate the protocols, the thesis chooses Ethereum, Bitcoin, and Lightning Network as a deployment scenario to show its feasibility. The scheme can be widely applied to other blockchain ledgers and cryptocurrencies.

In addition to decentralized cryptocurrencies, the key innovation of a blockchain is the consensus mechanisms. At the heart of blockchain technology, consensus algorithms, e.g., PoW [29] and pBFT [30], guarantee all participants to be honest in mining and help to maintain one longest chain without relying on any centralized server. Despite Proof-of-Work (PoW) having a reliable security mechanism to protect a blockchain and to prevent threats such as Sybil attacks, it still has a severe scalability problem for decentralized cryptocurrencies, especially in supporting scalable trade of assets. A low transaction throughput [31, 32], Ethereum supports up to 20 transactions per second, is unable to support the widespread use of a trading network. The problem can be solved by applying a faster and more scalable consensus algorithm, i.e., Proof-of-Stake with Sharding [33, 34]. The next step of this work is to integrate the protocol with sharding based EVM chain, for instance, protocols such as Ethereum 2.0-Casper [33] and Harmony [35].

Bibliography

[1] S. Nakamoto. (2008) Bitcoin: A peer-to-peer electronic cash system. Visited 11-15-2019. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[2] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," in *2016 IEEE Symposium on Security and Privacy (SP)*, San Jose, CA, USA, 2016, pp. 839–858.

[3] V. Buterin. A next-generation smart contract and decentralized application platform. Visited 11-16-2019. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper

[4] K. Christidis and M. Devetsikiotis, "Blockchains and smart contracts for the internet of things," *IEEE Access*, vol. 4, pp. 2292–2303, 2016.

[5] T. Sameeh, "An overview of the most reliable cryptocurrency smart contract platforms," Nov 2018, visited 11-16-2019. [Online]. Available: https://www.cointelligence.com/content/smart-contract-platforms-guide/

[6] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, 2014, visited 11-15-2019. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf

[7] Hyperledger. Smart contracts and chaincode. Visited 11-25-2019. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-1.4/smartcontract/smartcontract.html

[8] I. Miers, C. Garman, M. Green, and A. D. Rubin, "Zerocoin: Anonymous distributed e-cash from bitcoin," in *2013 IEEE Symposium on Security and Privacy*, Berkeley, CA, USA, 2013, pp. 397–411.

[9] S. Haber and W. S. Stornetta, "How to time-stamp a digital document," in *Advances in Cryptology-CRYPT0' 90*. Springer, Berlin, Heidelberg, 1991, p. 437–455.

[10] B. Marr, "A very brief history of blockchain technology everyone should read," *Forbes*, visited 02-26-2020. [Online]. Available: https://www.forbes.com/sites/bernardmarr/2018/02/16/a-very-brief-history-of-blockchain-technology-everyone-should-read/#785650737bc4

[11] Binance. (2020, Jan) History of blockchain. Visited 02-26-2020. [Online]. Available: https://www.binance.vision/blockchain/history-of-blockchain

[12] V. Gupta, "A brief history of blockchain," *Harvard Business Review*, Aug 2019, visited 02-26-2020. [Online]. Available: https://hbr.org/2017/02/a-brief-history-of-blockchain

[13] ConsenSys. (2018, Nov) A brief history of blockchain: Blockchain basics book from consensys academy. Visited 02-26-2020. [Online]. Available: https://consensys.net/academy/blockchain-basics-book/brief-history-of-blockchain/

[14] J. Poon and T. Dryja. (2016, 01) The bitcoin lightning network:scalable off-chain instant payments. Visited 12-10-2019. [Online]. Available: https://lightning.network/lightning-network-paper.pdf

[15] I. A. Seres, L. Gulyás, D. A. Nagy, and P. Burcsi. (2019, Apr) Topological analysis of bitcoin's lightning network. Visited 12-10-2019. [Online]. Available: https://arxiv.org/abs/1901.04972

[16] S. Martinazzi and A. Flori. (2020) The evolving topology of the lightning network: Centralization, efficiency, robustness, synchronization, and anonymity. Visited 02-10-2020. [Online]. Available: https://journals.plos.org/plosone/article/file?id=10.1371/journal.pone.0225966&type=printable

[17] Interledger. Hashed-timelock agreements (htlas). Visited 02-10-2020. [Online]. Available: https://interledger.org/rfcs/0022-hashed-timelock-agreements/

[18] M. Conoscenti, A. Vetrò, J. D. Martin, and F. Spini, "The cloth simulator for htlc payment networks with introductory lightning network performance results," *Information*, vol. 9, no. 9, p. 223, Mar 2018.

[19] A. Min. (2019, Jul) Atomic swap between bitcoin, dai, and ether: Liquality is live on mainnet. Visited 11-20-2019. [Online]. Available: https://liquality.io/blog/liquality-atomic-swaps-on-mainnet/

[20] KyberNetwork. (2019) Kyber: An on-chain liquidity protocol. Visited 12-10-2019. [Online]. Available: https://files.kyber.network/Kyber_Protocol_22_April_v0.1.pdf

[21] K. Wu, "An empirical study of blockchain-based decentralized applications," 02 2019, visited 11-20-2019. [Online]. Available: https://arxiv.org/pdf/1902.04969.pdf

[22] K. Wu, Y. Ma, G. Huang, and X. Liu. (2019) A first look at blockchain-based decentralized applications. Visited 12-15-2019. [Online]. Available: https://arxiv.org/pdf/1909.00939.pdf

[23] R. Khalil, A. Zamyatin, G. Felley, P. Moreno-Sanchez, and A. Gervais, "Commit-chains: Secure, scalable off-chain payments," 2018, visited 12-05-2019. [Online]. Available: https://eprint.iacr.org/2018/642

[24] YongJie, Zhang, Jingjing, Wu, Weigang, Luo, Cao, and Jiannong, "Boros: Secure cross-channel transfers via channel hub," Nov 2019, visited 12-01-2019. [Online]. Available: https://arxiv.org/abs/1911.12929

[25] Libra. Libra white paper: Blockchain, association, reserve. Visited 07-10-2019. [Online]. Available: https://libra.org/en-US/white-paper/

[26] Ethereum. Ethereum wiki: Standardized contract apis. Visited 11-05-2019. [Online]. Available: https://github.com/ethereum/wiki/wiki/Standardized_Contract_APIs

[27] Ethereum, "Eip 20: Erc-20 token standard," 11 2015, visited 11-05-2019. [Online]. Available: https://eips.ethereum.org/EIPS/eip-20

[28] LightningNetwork, "LND gRPC API Reference," visited 01-25-2020. [Online]. Available: https://api.lightning.community/

[29] C. Dwork and M. Naor, "Pricing via processing or combatting junk mail," in *Advances in Cryptology — CRYPTO' 92*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 139–147.

[30] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 4, pp. 398–461, 2002.

[31] 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 *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 3–16.

[32] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, E. Gün Sirer, D. Song, and R. Wattenhofer, "On scaling decentralized blockchains," in *Financial Cryptography and Data Security*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 106–125.

[33] V. Buterin and V. Griffith, "Casper the friendly finality gadget," *CoRR*, vol. abs/1710.09437, 2017, visited 11-30-2019. [Online]. Available: http://arxiv.org/abs/1710.09437

[34] O. Moindrot and C. Bournhonesque, "Proof of stake made simple with casper," 2017, visited 12-3-2019. [Online]. Available: https://www.scs.stanford.edu/17au-cs244b/labs/projects/moindrot_bournhonesque.pdf

[35] Harmony, "Harmony: Technical white paper," visited 12-01-2019. [Online]. Available: https://harmony.one/whitepaper.pdf

A Appendix

In this appendix, the paper provides more information of additional and modified RPC commands that are implemented in the Lightning Network Daemon (LND) and utilized in cross-chain procedures.

```
var depositCommand = cli.Command{
  Name:     "deposit",
  Usage:    "Deposit collateral (only Bitcoin client)",
  Description: "Command that serves btc/lnd clients to deposit collateral" +
    "This collateral can be used to request a corresponding Token coins" +
    "on virtual trading system" +
    "This command requires client must start bitcoin server" +
    "This command supports to create bitcoin raw transaction",
  Flags: []cli.Flag{
    cli.StringFlag{
      Name:  "testnet",
      Usage: "A bitcoin server testnet i.e. regtest/testnet/mainnet (Require)",
    },
    cli.StringFlag{
      Name:  "rpcuser",
      Usage: "RPC User to connect to bitcoin server (Require)",
    },
    cli.StringFlag{
      Name: "rpcpass",
      Usage: "RPC password to connect to bitcoin server (Require)",
    },
    cli.StringFlag{
      Name:  "rpcport",
      Usage: "RPC port to connect to bitcoin server (Require)",
    },
    cli.StringFlag{
```

```
      Name: "receiver",
      Usage: "A bitcoin receiver address that bitcoins will be sent (Require)",
    },
    cli.StringFlag{
      Name: "amt",
      Usage: "An amount of bitcoin (Require)",
    },
    cli.StringFlag{
      Name: "localhost",
      Usage: "A localhost to connect to client on virtual (Require)",
    },
    cli.StringFlag{
      Name: "eth_addr",
      Usage: "A ethereum client address on virtual trading network (Require)",
    },
    cli.StringFlag{
      Name: "contract",
      Usage: "A contract address on virtual trading system to claim (Require)",
    },
  },
  Action:   actionDecorator(deposit),
}
```

```
var deployContractCommand = cli.Command{
  Name:     "deploy",
  Usage:    "Generate a contract",
  Description: "Command that serves btc/lnd clients to deploy a contract on virtual trading system" +
      "Clients may either have a bitcoin wallet or lightning network payment channel to request" +
      "Client transfers funds to Reserve address, then requests to create a Group Trade on virtual system",
  Flags: []cli.Flag{
    cli.StringFlag{
      Name:  "localhost",
```

```
10        Usage: "the localhost IP + port that client wants to connect and to send a request (Require)",
11      },
12      cli.StringFlag{
13        Name: "token_addr",
14        Usage: "A system-token contract address (Require) ",
15      },
16      cli.StringFlag{
17        Name:  "verifier_addr",
18        Usage: "A contract that uses to verify endorsed signatures (Require)",
19      },
20      cli.StringFlag{
21        Name: "collateral_addr",
22        Usage: "A system Collateral contract address (Require)",
23      },
24      cli.StringFlag{
25        Name: "eth_addr",
26        Usage: "An address of Ethereum account on a trading system (Require)",
27      },
28      cli.StringFlag{
29        Name: "tx",
30        Usage: "A chan_id / hash transaction that shows client has already created a channel payment" +
31          " or deposit an amount of coins into System wallet (Require)",
32      },
33      cli.StringFlag{
34        Name: "txsin",
35        Usage: "All hash transactions inputs that client has been used to spend" +
36          "when client open a channel to connect to a designated system wallet" +
37          "TxIDs must be in restricted sequence order as the same as client has pre-claimed" +
38          "and TxIDs are separated by a comma" +
39          "If opt_code = 3, txsin is required",
40      },
41      cli.StringFlag{
42        Name: "signtx",
```

```
      Usage: "A hash signature of raw transaction that client has used to pre-claim or " +
          " a signature of chan_id (Optional)" +
        "If opt_code = 1 or opt_code = 4, signtx is required",
    },
    cli.StringFlag{
      Name: "amt",
      Usage: "An amount of token coins to request on virtual system (Require)." +
        "The amount should be in Satoshi format",
    },
    cli.IntFlag{
      Name: "opt_code",
      Usage: "the option code to join virtual trading network (Require)" +
        "1. Bitcoin debit option\n" +
        "2. Bitcoin credit option\n" +
        "3. Lightning debit option via send coins\n" +
        "4. Lightning debit option via open channel\n" +
        "5. Lightning credit option\n",
    },
  },
  Action:   actionDecorator(deployContract),
}
```

```
var joinContractCommand = cli.Command{
  Name:     "join",
  Usage:    "Join a contract",
  Description: "Command that serves btc/lnd clients to join a GroupTrade contract in the virtual trading system" +
    "Clients may either have a bitcoin wallet or lightning network payment channel to request" +
    "Client transfers funds to Reserve, then requests to join a channel on virtual system",
  Flags: []cli.Flag{
    cli.StringFlag{
      Name:  "localhost",
      Usage: "the localhost IP + port that client wants to connect and to send a request (Require)",
```

```
11      },
12      cli.StringFlag{
13        Name: "signtx",
14        Usage: "A chan_id / hash signature of raw transaction that client has used to pre-claim or a signature of chan_id
15          "If opt_code = 1, signtx is required",
16      },
17      cli.StringFlag{
18        Name: "eth_addr",
19        Usage: "An address of Ethereum account on a virtual system (Require)",
20      },
21      cli.StringFlag{
22        Name: "contract_addr",
23        Usage: "A GroupTrade contract address on a virtual system (Require)",
24      },
25      cli.StringFlag{
26        Name: "tx",
27        Usage: "A hash transaction that shows client has already created a channel payment" +
28          " or deposit an amount of coins into System wallet (Require)",
29      },
30      cli.StringFlag{
31        Name: "txsin",
32        Usage: "All hash transactions inputs that client has been used to spend" +
33          "when client open a channel to connect to a designated system wallet" +
34          "TxIDs must be in restricted sequence order as the same as client has pre-claimed" +
35          "and TxIDs are separated by a comma" +
36          "If opt_code = 3, txsin is required",
37      },
38      cli.StringFlag{
39        Name: "amt",
40        Usage: "An amount of token coins to request on virtual system (Require)" +
41          "The amount should be in Satoshi format",
42      },
43      cli.IntFlag{
```

```
      Name: "opt_code",
      Usage: "the option code to join virtual trading network (Require)\n" +
        "1. Bitcoin debit option\n" +
        "2. Bitcoin credit option\n" +
        "3. Lightning debit option via send coins\n" +
        "4. Lightning debit option via open channel\n" +
        "5. Lightning credit option\n",
    },
  },
  Action:   actionDecorator(joinContract),
}
```

```
var sendCoinsCommand = cli.Command{
  Name:      "sendcoins",
  Category:  "On-chain",
  Usage:     "Send bitcoin on-chain to an address.",
  ArgsUsage: "addr amt",
  Description: `
  Send amt coins in satoshis to the BASE58 encoded bitcoin address addr.
  Fees used when sending the transaction can be specified via the --conf_target, or
  --sat_per_byte optional flags.
  Positional arguments and flags can be used interchangeably but not at the same time!`,
  Flags: []cli.Flag{
    cli.StringFlag{
      Name:  "addr",
      Usage: "the BASE58 encoded bitcoin address to send coins to on-chain",
    },
    cli.BoolFlag{
      Name: "sweepall",
      Usage: "if set, then the amount field will be ignored, " +
        "and all the wallet will attempt to sweep all " +
        "outputs within the wallet to the target " +
```

```
21          "address",
22      },
23      cli.Int64Flag{
24        Name:  "amt",
25        Usage: "the number of bitcoin denominated in satoshis to send",
26      },
27      cli.Int64Flag{
28        Name: "conf_target",
29        Usage: "(optional) the number of blocks that the " +
30          "transaction *should* confirm in, will be " +
31          "used for fee estimation",
32      },
33      cli.Int64Flag{
34        Name: "sat_per_byte",
35        Usage: "(optional) a manual fee expressed in " +
36          "sat/byte that should be used when crafting " +
37          "the transaction",
38      },
39
40      //  adding arguments for virtual trading network
41      cli.BoolFlag{
42        Name:   "virtual",
43        Usage:  "the boolean value option to join virtual trading network" +
44          "1: activate. It requires to enter all requirement fields" +
45          "0: not-activate",
46      },
47      cli.StringFlag{
48        Name: "localhost",
49        Usage: "the localhost + port to connect to TCP Server" +
50          "Require when virtual is set active",
51      },
52      cli.StringFlag{
53        Name: "eth_addr",
```

```
      Usage: "The address of Ethereum account on the virtual trading network" +
        "Require when virtual is set active",
    },
    cli.StringFlag{
      Name: "contract",
      Usage: "the Collateral contract address on the virtual trading network to submit pre-claim info" +
        "Require when virtual is set active",
    },
  },
  Action: actionDecorator(sendCoins),
}
```

```
var updatePayReqCommand = cli.Command{
  Name:     "updatepayreq",
  Usage:    "Updating cashout request payment",
  Description: "Command that serves btc/lnd clients to update cashout request payment on the virtual trading system" +
    "Clients sendcoin/payinvoice on lightning network can send a request to add Tokens or to decrease credit balance",
  Flags: []cli.Flag{
    cli.StringFlag{
      Name:  "localhost",
      Usage: "the localhost IP + port that client wants to connect and to send request (Require)",
    },
    cli.StringFlag{
      Name: "receiver_btc",
      Usage: "A bitcoin account addr of receiver (Optional)" +
        "If opt_code = 1, receiver_btc is required",
    },
    cli.StringFlag{
      Name: "sender_eth",
      Usage: "An address of Ethereum account of sender on a virtual system (Require)",
    },
    cli.StringFlag{
```

```
21        Name: "receiver_eth",
22        Usage: "An address of Ethereum account of receiver on a virtual system (Require)",
23      },
24      cli.StringFlag{
25        Name: "receiver_lnd",
26        Usage: "An address of lightning network account of receiver (Optional)" +
27          "If opt_code = 3, receiver_lnd is required",
28      },
29      cli.StringFlag{
30        Name: "pay_req",
31        Usage: "A payment request (Optional)" +
32          "If opt_code = 3, pay_req is required",
33      },
34      cli.StringFlag{
35        Name: "secret_msg",
36        Usage: "A secret message between sender and receiver for creating a payment confirm signature (Optional)" +
37          "If opt_code = 3, secret_msg is required",
38      },
39      cli.StringFlag{
40        Name: "pay_confirm",
41        Usage: "A payment confirm from receiver when a payment is successfully paid (Optional)" +
42          "If opt_code = 3, pay_confirm is required",
43      },
44      cli.StringFlag{
45        Name: "tx",
46        Usage: "A hash transaction that shows client has already made a payment (Optional)" +
47          "If opt_code = 1 or opt_code = 2, tx is required",
48      },
49      cli.StringFlag{
50        Name: "txsin",
51        Usage: "All hash transactions inputs that client has been used to make a payment (Optional)" +
52          "TxIDs must be in restricted sequence order as the same as client has pre-claimed" +
53          "and TxIDs are separated by a comma" +
```

```
54          "If opt_code = 2, txsin is required",
55      },
56      cli.StringFlag{
57        Name: "signtx",
58        Usage: "A hash signature of raw transaction that client has used to pre-claim (Optional)" +
59          "If opt_code = 1, signtx is required",
60      },
61      cli.StringFlag{
62        Name: "amt",
63        Usage: "An amount of tokens or credit balance that client requests to receive on the virtual system (Require)",
64      },
65      cli.IntFlag{
66        Name: "opt_code",
67        Usage: "the type that shows how client has made a payment (Require)" +
68          "1. Send Coins via Bitcoin CLI\n" +
69          "2. Send Coins via LND CLI\n" +
70          "3. Send Coins via LND channel payment\n",
71      },
72      cli.IntFlag{
73        Name: "payreq_channel_type",
74        Usage: "the type of GroupTrade that a cash-out pay request was issued (Require)" +
75          "1. DChannel\n" +
76          "2. CChannel\n",
77      },
78      cli.IntFlag{
79        Name: "payer_channel_type",
80        Usage: "the type of GroupTrade that a payer currently joined (Require)" +
81          "1. DChannel\n" +
82          "2. CChannel\n",
83      },
84      cli.StringFlag{
85        Name: "payreq_channel",
86        Usage: "A GroupTrade contract that a cash-out pay request was issued (Require)",
```

```
87      },
88      cli.StringFlag{
89        Name: "payer_channel",
90        Usage: "A GroupTrade contract of the Payer (Require)",
91      },
92    },
93    Action:   actionDecorator(updatePayReq),
94  }
```