Lockchain: Blockchain Locking System for Revocable Access

Final Report

Marcus Samuel Arellano

Supervised by: *Olivier, Marin*

Abstract

Revocable access is a problem that many locking systems do not effectively solve. This is a difficult problem because of the tradeoff between security and performance in distributed systems. This paper proposes a locking system to solve this problem based on blockchain smart contracts. This system's scale is more fitting for this specific problem than existing alternatives. These smart contracts will implement a user hierarchy which controls user access permissions and records all actions in the system. This paper also implements a user interface for interacting with these smart contracts, and a physical smart lock to give a tangible example of the system in action.

Contents

1. Introduction

1.1 Context

While locks have been around for millenia, we still face many issues with them every day. One of these major issues is controlling who has access to the key. Traditional physical keys can be easily lost or stolen. If no master key is kept, reproduction can be difficult and if a malicious entity gets hold of the key, you will want to replace the lock altogether. If you want to give someone else the key to open a lock one time, you can not be completely sure that they did not make a duplicate while it was in their possession.

These issues also exist for many types of virtual keys. Virtual keys such as passwords can easily be forgotten, and if you want to let someone else use the key, they can't simply return the knowledge or digital copy of your password.

Whether it is a physical or digital key, you would need to fully reset the lock to ensure they don't use it again in the future. We also have another major issue: it is impossible to know the identity of the person who used the key from the key alone. This paper proposes a system focusing on solving the problem of easily revocable access. This is especially important in situations where many different entities must have access at different times, whether in the real world or digitally. This system will also be able to track more meta information about access patterns to provide transparency to all users involved.

1.2 Objective

The specific problem I am focusing on occurs when a system requires a set of entities to access a critical section where the set of entities is highly prone to change. These systems often require easy revocation of any entity's access. A real world example that fulfills both of these conditions is AirBnb. AirBnb hosts have many different guests at different times, and the guests themselves may also need to give other entities access to the location during their stay. However, at the end of their stay the host needs to ensure that none of the guests still have access to the premise while giving access to the new guests who will arrive next. This can be hard to fully ensure and is done on a trust system with penalties, but the host often has no way to be confident about whether the previous tenants will return the keys and will not copy them for later use. This type of problem can be generalized to many other situations dealing with both physical and digital keys.

2. Related Works

In order to gain a better understanding of how to approach solving the issue of access control, my research method was twofold. First, I analyzed other ways similar problems are solved. Second, I compiled a survey of existing blockchains including their capabilities and limitations. I focused on security, reliability and simplicity on the user side, so analyses are made with these considerations in mind.

2.1 Distributed Mutual Exclusion

Giving different entities access to a critical resource or section for a limited time is often referred to as mutual exclusion. Mutual exclusion in a distributed system has already been extensively researched. This problem has two main challenges to overcome. First, ensuring that permission is revocable when a process is finished or disconnects and another process needs access. Second, ensuring all process can agree on a single process to gain access and ensuring that this can be done in a timely manner.

Paxos [9] is one of the industry standards for solving this problem. In short, it gives different processes different roles: acceptors and proposers. Proposers elect values denoting which process should gain access to the critical section next, while acceptors reply whether or not they accept such a value. Once the proposer receives over a majority of positive responses from the acceptors, it will give that process access. Using majority instead of consensus also solves a lot of interesting problems that arise with leasing. Some of these problems are partitioning of the processes and allowing reconnected processes to learn previous values. Paxos' wide applicability to real world systems has led to its widespread use.

Some of the systems that use the Paxos algorithm are Microsoft's Centrifuge [2], Google's Chubby [3], and Hadoop's Zookeeper [15]. These are all different forms of distributed mutual exclusion solutions focused on different problem domains, ranging from distributed computers on the same cluster to managing server states for different websites.

Another method for addressing distributed mutual exclusion is a token based approach as is seen in the Suzuki Kasami Token Broadcast Algorithm [13]. This algorithm uses access tokens to ensure mutual exclusion. There is a single unique virtual token that allows access to the critical section. When a process wants to access the critical section it broadcasts a request message to all other processes. When the process that has the token sees this message it will give the requesting process the token when it is able to. If it is in the critical section it will transfer the token as soon as it exits the critical section.

2.1.1 Problems with Distributed Mutual Exclusion

While it would be possible to retrofit the discussed systems to suit my problem, the scope of most of them are far too large for my project domain. Many of these are setup to handle the critical section being accessed thousands of times per minute or for managing access among thousands of entities. While this may be a consideration farther in the future when designing for digital systems, for physical locks I do not anticipate someone locking and unlocking the door thousands of times per minute. Furthermore, a bigger issue is that of dealing with malicious entities. In many of these systems, their algorithms only function because all of the processes have no incentive for going against the system. For example, the Suzuki-Kanami algorithm would fall apart if the process passing around the token had incentive to keep the token instead of passing it to the next process that needs access. One malicious actor could have a lot to gain by breaking the system and entering your house or database. To address many of these problems I will use a blockchain, so I will go into a survey of different blockchain technologies along with their benefits and restrictions.

2.2 Blockchain Survey

At its foundation, a blockchain is a decentralized distributed database. In a traditional centralized database, all information is stored in one location and users come to the central authority to access it. A common example is the bank, where all transactions must go through one central authority. In a decentralized distributed database, there is no one central authority. The database is copied and owned by every entity involved. This allows for far greater transparency and immutability of data. Once a change occurs, everyone can see it happen and it cannot be undone.

The blockchain itself accomplishes this by being a chain of blocks. Each block is a record of transactions or changes that occured in that timestep. Each entity involved gets a copy of this block and verifies it. Once it is verified, it is put on top of the previous blocks. This creates the chain, and by accessing one block, you can follow the chain backwards to see all transactions that ever happened. Every entity in the network maintains a copy of this record and any new entity can download the record themself at anytime. In addition to simple transactions of tokens between accounts, more complex transactions can occur with the use of smart contracts. Smart contracts are collections of rules and conditions that must be fulfilled for a specific outcome to occur. Smart contracts let developers include conditionals, time delays, and many other functionalities in the transactions to make them more robust and useful in the real world.

Table 1: Comparison of Different Blockchains

Platform	Type	Size	Block Interval	Language
Bitcoin	Public	96GB	10min	Bitcoin scripts & signatures
Counterparty	Public*	96GB*	10min*	EVM Bytecode
Ethereum	Public	17-60BG**	15sec	EVM Bytecode
Stellar	Public	?	3sec	Transaction Chains & MultiSigs
Lisk	Private	?	Custom	JS & Node.js

*Counterparty uses the Bitcoin blockchain and therefore these metrics are the same for both
**The size of the Ethereum blockchain differs depending on which client and pruning method you use to access it

This table compares some of the more prominent blockchains. Entries with '?' are such because concrete answers were not found, whether due to lack of easily found statistics(Stellar) or because its private (Lisk).

2.2.1 Bitcoin

Bitcoin [10] is one of the original blockchains and is used primarily for transferring digital currency between accounts. It uses "proof of work" cryptographic puzzles to create new blocks and verify transactions. It can implement smart contracts, but only in a very limited fashion. While it does have a scripting language, it is not Turing complete, meaning the smart contracts in Bitcoin are very limited. Furthermore, Bitcoin does not deal directly with balances per account. Instead, they have a sum of Unspent

Transaction Outputs, or UTXO's. These UTXO's can make creating smart contracts very complex. Since the programmer cannot simply increment or decrement a count, they must cleverly combine and take apart these UTXO's. However, despite all the issues with smart contracts, Bitcoin is still the original blockchain. It has extensive documentation and tutorials in addition to a large group of users and developers. These are very valuable aspects for any new developers.

2.2.2 Ethereum

Ethereum [4] is a blockchain that implements a similar consensus algorithm to Bitcoin. However, while Bitcoin's scripting language is not very expressive, Ethereum was developed with expressive smart contracts in mind. It has a Turing complete scripting language which allows for very simple development of very complex smart contracts. It also has a much smaller block creation interval than bitcoin, which allows for quicker verification between parties (in the optimal case, there are currently some issues with scaling and bloat on the live Ethereum network). It also has high level languages such as Solidity, which can compile into the bytecode language used for smart contracts. Because of widespread adoption, it also has very extensive documentation and a large user and developer base. There are some issues with including randomness in smart contracts due to the language's deterministic nature, however, this issue is of little concern to the scope of my project as I will not be utilizing random elements.

2.2.3 Counterparty

Counterparty [5] is a platform that does not have its own blockchain, it embeds data into the Bitcoin blockchain. This data is inconsequential to Bitcoin nodes but is used to execute operations for Counterparty. The most interesting part of this process is that instead of rewarding nodes with currency on block completion like Ethereum of Bitcoin do, the transaction fees are destroyed. This advantages miners by indirectly inflating the value of the currency they already own. This process is called “proof of burn”. However, while is is an interesting project, it lacks the expressiveness of Ethereum or the documentation and user base of Bitcoin and is more of a proof of concept at its current point of development.

2.2.4 Stellar

Stellar [11] is a platform that has its own public blockchain. Its differentiating factor is the lack of “proof of work” for validating blocks, instead it relies on a Byzantine agreement system. This removes the massive electricity use of the traditional “proof of work” system and is in theory almost as secure. However, there is no built in smart contract language. Stellar does allow multisignature accounts: accounts controlled by multiple entities which require either majority or consensus to make decisions. Multisignature accounts combined with transaction chaining allow for rudimentary smart contracts, albeit in a very roundabout way.

2.2.5 Lisk

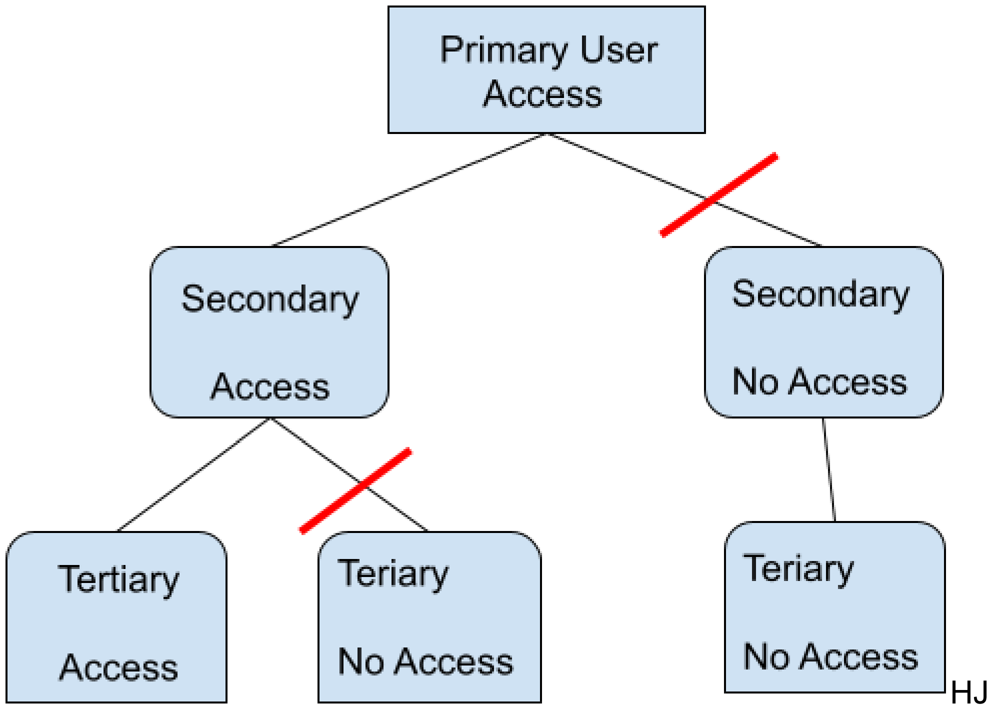
Finally we have Lisk [10]. Lisk, like Ethereum and Bitcoin, is both a currency and a blockchain. It uses “proof of stake” for verification and supports smart contracts written in Javascript or Node.js. Unlike Ethereum, determinism is not ensured by the language, it is left up to the developer. Lisk is far from a mature platform. While it may be an interesting alternative in the future, for the scope of my project I would rather use a widely adopted and highly documented system if possible.

2.3 Summary

While distributed mutual exclusion is a well researched and mature problem, I decided to instead build my system on blockchain due to issues with malicious actors and a more fitting scale for my problem. Among the different blockchains available I chose to utilize an Ethereum private network for two major reasons. The first is the extensive documentation and user base. Blockchain development is still a rather new field and I would like to give myself as many opportunities to find assistance as possible. Secondly, while Bitcoin also is a mature and well documented system, I will need to do extensive work with smart contracts to properly manage the access permissions I’m working with. Even if writing similar smart contracts would be possible in Bitcoin, the process would be very awkward and cumbersome. Ethereum has more expressive options for making smart contracts and this was the ultimate distinguishing factor.

3. Solution

My approach to easily revocable access is a system of hierarchical access. Each user that currently has access may give access to any other user. The second user is then added as the first user’s child. This can be done again by the second user to give a third user access. A higher level user may revoke access from any of its descendant users at any time. When access is revoked from a user, all of its descendants also lose access. This process is illustrated in the diagram below.



3.1 Blockchain Development

There are many issues with using the live Ethereum blockchain for this application. The live blockchain can take a large amount of time to validate transactions, especially when the network is under a lot of stress by a large amount of transaction throughput. For a simple change of state users can wait up to half an hour. This delay could massively impact the security of the system as this gives a large window between when a primary user revokes access from a secondary user and when that secondary user actually loses access to the lock. Furthermore, it would impact the user experience by forcing them to wait minutes just to open a lock that they already have access to. To address these issues, I use a private ethereum blockchain instead of the official Ethereum blockchain. This also allows more control for small organizations or entities who wish to use this system in the future. I use the program Geth to setup a simple private blockchain that runs locally on a computer. Geth [6] is a command line interface for running and interacting with ethereum nodes. This simplifies the process of creating your own blockchain, launching your smart contracts, and allowing users to join the network and interact. For testing purposes I use the program Ganache for quickly creating dummy accounts. Ganache [14] is a system for expediting the testing process in ethereum development. It allows users to create fake test accounts and allocate fake testing values among the accounts. It is quicker than setting these values up from scratch each time the system needs to be changed or relaunched. In live scenarios, I create a set of accounts on the blockchain. This approach was very simple, but could easily scale for larger organizations using my system. All that would be required is a cluster of computers running this blockchain in the background to keep the system online.

3.2 Smart Contract Development

The smart contracts the project uses contain and implement the rules of the locking system. Each smart contract has a list of accounts that are allowed to access the lock. The major functionality of the system is deciding whether or not to open the lock for a user. The user first sends a request for lock access. The smart contract then checks an internal list to see if that user’s address is contained within the list. If it is, it will open the lock, otherwise the lock remains closed.

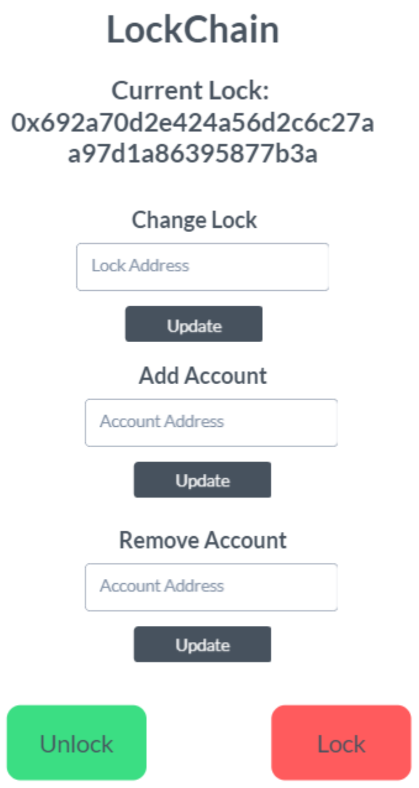
The next major functionality is controlling which users are on the access list. This is done with respect to the hierarchical user structure mentioned earlier. There is an ‘add user’ functionality that checks if a user is already in the access list. If the user is in the access list, then they are allowed to add another user to it. That user will then be stored as one of their immediate descendants in the access tree. There is also a functionality to allow a user to delete another user from the access list. To do this the user calling the delete function must already be in the access list and must be an ancestor of the user they are trying to delete. The user and all their descendants will then be deleted. If either of these conditions are not met, the function will fail.

These primary functionalities are the basic actions for opening the lock and changing who is allowed to access the lock. In addition to those functionalities, there are also methods for logging metadata about the smart contract. This metadata is public and shows which users attempted to access the lock, in addition to which users changed permission values of which other users. This functionality increases the transparency and security of the locking system. Increased transparency allows for greater security for all users as anyone with access to the lock will be able to view all actions that occurred with the lock.

Each lock has its own smart contract deployed. Even though the rules for each lock are the same, deploying multiple smart contracts makes accessing different locks easier and compartmentalizes as much data as possible for security reasons. A lock can be interacted with by finding its smart contract address and then invoking any of the functions. This can be done through a command line interface with a program such as geth, but in order to craft a better user experience the project also includes a simple user interface.

3.3 User Interface Development

The mobile interface is a web application made with the JavaScript library web3.js [7]. This library was designed for communicating between a blockchain and a website. It allows me to either display information from the smart contract on the website or to allow the user to interact with the smart contract directly through a user interface. The goal of creating the user interface was to make an easy to use interface that was simpler for many users than the command line.



To use the interface, the user must first choose which lock they will be interacting with. This is done by inputting the smart contract address of the lock. The chosen lock can be changed at anytime from the main screen, but a lock must be selected before any other methods can be called. The lock and unlock methods are simple buttons that can be pressed to attempt to access the lock. The add user and remove user methods are text boxes where the user can input address of the account they want to either give access to or revoke access from.

3.4 Physical Lock Development

This locking system could be applied to different digital and physical access problems. For a demonstration of these capabilities in real time, a physical lock was made that could be interacted with via the mobile interface and the blockchain. This lock is built on top of a basic arduino servo. This servo is connected to a processing sketch which opens a network client that connects to the blockchain. This client listens for either a 1 to open the lock, or a 0 to close it. It relays this information to the arduino sketch and the servo acts accordingly. From the side of the blockchain, it sends a 1 to the processing client whenever a user sends an unlock request to the blockchain that gets confirmed. It will send a 0 to the client whenever a user sends a lock request to the blockchain that gets confirmed.

4. Discussion

4.1 Challenges

Throughout the process of this project, the system design underwent many drastic changes. The initial proposal was based on revocable cryptographic access tokens. It was not far into development when I realized that this approach would be infeasible as I could not be sure a user would give up the token when required by the system. This token based approach was one of my main reasons for choosing to build on top of a blockchain system. As I got further into development I realized this was not necessary.

Another challenge was deciding whether or not to build my blockchain from scratch. The original system utilized a custom blockchain, but for the sake of ease of implementation and saving development time, I then decided to utilize the Ethereum blockchain. The Ethereum blockchain has extensive documentation, a large user base, and has been tried and tested as functional and secure. A small amount of customizability was given up to save development time.

Overall, most of the challenges came from unfamiliarity with the domain. While I did extensive research beforehand on most of the topics, it was not until actually beginning development with these technologies that I understood their capabilities and constraints.

4.2 Critical Analysis

When I began development I chose blockchain technology as I could not think of a different way to solve this problem. My original solution was token based, but I was not able to find a reliable and secure way to revoke access tokens from different users. Because of this, I switched to a permission based system which kept track of access by referencing a list of users on the smart contract, not by verifying an access token. Another big design change was the decision to use a private blockchain instead of the main Ethereum blockchain. This was done to improve runtime of operations and made the system run similar to a small server farm or cluster of computers.

These two changes gave up two of the big reasons for using blockchain technologies: elegant token management and the security of being part of a large distributed database. Implementing a permission based access system on a cluster of servers is completely possible with traditional distributed computing systems. In some aspects, using a traditional distributed system would also have advantages for the final system. Runtime of the system could be further improved by removing many of the blockchain protocols and only implementing the features necessary for the system. The system would also be more scalable in a more customized distributed system. Blockchains often have the problem of decreased performance as the number of nodes increases. A more customizable distributed system could be designed to tackle this issue.

However, I still do find merit in my choice of creating this system with blockchain. The first reason is for simplicity. Many traditional distributed computing systems are very complex due to the many exceptions that can occur in a large distributed system. Building on top of a major blockchain reduced development complexity and allowed me to complete my main objectives for this project in the time allotted. The second reason is for security. With complex distributed systems there are many failure points for possible attacks or exploits. The blockchain protocol has many built-in safeguards that remove the burden from the developer for addressing these exploits. Finally, the immutability of the system is important to security of the system. It also allows for complete transparency of the system and makes displaying system access data much simpler.

I still stand by my initial choice to implement this system using blockchain technology. Many of the advantages to a customized distributed system do not apply to my specific problem. The performance improvement moving from the main Ethereum blockchain to a private blockchain was in terms of minutes. Moving from my current system to a distributed system would only speed up the processes for the end user by fractions of a second. Scalability may be an issue in the future, but I do not expect my system to accommodate more than a few hundred users at a time in practice. In future development it may prove useful to create a custom distributed system, but as a proof of concept blockchains solve the current problems.

5. Conclusion

5.1 Final Assessment

Revocable access is a problem with wide applications in many fields that will only continue to grow as we move towards more of a sharing economy. This paper proposes a proof of concept locking system which addresses this problem. This system is composed of three parts: blockchain smart contracts which manage user access and implement the user hierarchy rules, a simple user interface for interacting with these smart contracts, and a smart lock to provide a tangible example of the system in action. This system successfully ensures quick and secure revocable access for all users involved. This system is also very quick to set up for a small set of users and can be easily generalized to many applications.

5.2 Further Development

This project provides a simple proof of concept of the system I proposed, but many improvements could be made with further development. There are three main areas for improvements: design, vulnerabilities, and functionality.

5.2.1 Design

The user interface have a few areas for improvement. One major improvement would be removing all text boxes for blockchain addresses. These addresses are 40 to 60 random characters long so it is easy for a user to make a small mistake or just find tedious to deal with. For the lock smart contract address I would like to give each lock a QR code that contains its smart contract address. The user could then scan the QR code instead of inputting the smart contract address character by character. To replace the ‘remove user’ text box I want to make a tree visualization of all the users descendants so they can click on the tree to select a user rather than input an address. This would also remove the possibility of a user trying to remove a user they are not allowed to as only users they can remove will be shown in the tree. The design of the physical lock could also be improved, however, it was created mainly for demonstration purposes. Any system utilizing the locking system would hopefully have more robust technology for their locks.

5.2.2 Vulnerabilities

There are some vulnerabilities to the current system that must be addressed moving forward. One is the possibility of attacking the system by overflowing it with requests. If a malicious user sends thousands or millions of trivial transactions, the system may not be able to handle it. This is a result of moving from the main blockchain to a private blockchain. The main blockchain deters this type of attack by incurring a gas cost on all transactions. This gas has a monetary value that must be spent by the user making this large scale attack economically infeasible. My system uses trivial currency given out to all users. This gas has no monetary value. This attack would also be even easier on my system as my private blockchain does not have the scale of the main blockchain, so overwhelming it would take even less resources. This could be addressed by controlling how the gas is dispersed through the system or by placing hard limits on the number of actions a user can make. Finding an optimal way to address this issue would be worth researching further.

Another issue is that a higher level user can revoke access from one of its descendants at any time. This assumes the higher level user will act completely fairly, which is a bad assumption in the real world. This can be addressed by adding more robust smart contracts to control the duration and conditions of access between users.

5.2.3 Functionality

I would like to add more functionalities to the system regarding data tracking and customizability. For data tracking I would like more in depth displays of the access patterns, user activity, and the possibility to easily export this data instead of just viewing it through the user interface. I would also like to create more customizability options for users, such as limiting who is allowed to add users, how many users they can add, or banning specific users from ever being given access to the network. I would also like to make a more streamlined deployment system. Currently, everything requires setup from various programs to get up and running, however, this could be packaged into a standalone application that users could launch with a single action.

6. References

[1] Bartoletti, M., & Pompianu, L. (2017). An Empirical Analysis of Smart Contracts: Platforms, Applications, and Design Patterns. *Financial Cryptography and Data Security Lecture Notes in Computer Science,*494-509. doi:10.1007/978-3-319-70278-0_31

[2] Adya, A., Dunagan, J., & Wolman, A. (2010). "Centrifuge: Integrated lease management and partitioning for cloud services. *NSDI'10: Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation*.

[3] Burrows, M. (n.d.). The Chubby lock service for loosely-coupled distributed systems. Retrieved from https://static.googleusercontent.com/media/research.google.com/en//archive/chubby-osdi06.pdf

[4] Buterin, V. (n.d.). A Next-Generation Smart Contract and Decentralized Application Platform. Retrieved from https://github.com/ethereum/wiki/wiki/White-Paper

[5] CounterParty About. (n.d.). Retrieved from https://counterparty.io/docs/assets/

[6] Ethereum, "ethereum/go-ethereum," GitHub. [Online]. Available: https://github.com/ethereum/go-ethereum/wiki/geth. [Accessed: 13-May-2018].

[7] Ethereum, "ethereum/web3.js," GitHub. [Online]. Available: https://github.com/ethereum/web3.js/. [Accessed: 13-May-2018].

[8] Kolbeck, B., Högqvist, M., Stender, J., & Hupfeld, F. (2011). Flease - Lease Coordination Without a Lock Server. *2011 IEEE International Parallel & Distributed Processing Symposium*. doi:10.1109/ipdps.2011.94

[9] Lamport, L. (2001). Paxos made simple. *SIGACT News,32*(4), 18-25.

[10] Lisk Whitepaper. (n.d.). Retrieved from https://github.com/slasheks/lisk-whitepaper

[11] Mazieres, D. (n.d.). The Stellar Consensus Protocol: A Federated Model for Internet-level Consensus. Retrieved from https://www.stellar.org/papers/stellar-consensus-protocol.pdf

[12] Nakamoto, S. (n.d.). Bitcoin: A Peer-to-Peer Electronic Cash System. Retrieved from https://bitcoin.org/bitcoin.pdf

[13] Ogata, K., & Futatsugi, K. (2002). Formal Analysis of Suzuki&Kasami Distributed Mutual Exclusion Algorithm. *Formal Methods for Open Object-Based Distributed Systems V,*181-195. doi:10.1007/978-0-387-35496-5_13

[14] Trufflesuite, “trufflesuite/ganache,” GitHub. [Online]. Available: https://github.com/trufflesuite/ganache. [Accessed: 13-May-2018].

[15] Welcome to Apache ZooKeeper™. (n.d.). Retrieved February 26, 2018, from http://hadoop.apache.org/zookeeper/