2018

Seed: A Distributed, Peer-to-Peer Blockchain Solution For Networked Games

COMP8045 - Major Project

Carson Roscoe - A00925976

September 25th, 2018

Table Of Contents

Introduction

Student Background

The Seed research project is focused on creating a solution to various problems in blockchain technology. This exciting new technology, which I have developed a great passion for, offers the ability to create trustless, decentralized applications that cannot be terminated by any individual entity.

The majority of modern video games have a networking aspect to them. Once a game is networked between users, the requirement of security becomes more of a priority. I plan to explore how blockchain technology can be used to power these video games, allowing for provably fair gameplay. In the past, decentralized peer-to-peer video games have been less secure than their centralized counterparts, but the new practice of integrating blockchain technology may bring a newfound level of security to these peer-to-peer applications. My interest is in observing how blockchain can be repurposed to power these video games in ways that are adequately secure, while still meeting the low-latency, high-throughput requirements many games desire.

As the software developer behind this project, my qualifications include an educational background in Video Game Development, Internetworking, and Data Communication. I have experience in the field of blockchain development, having deployed a successful Ethereum Smart Contract and surrounding application this past year alongside Jaegar Sarauer.

Project Description

The Seed project has two primary parts. The first portion involves creating a blockchain protocol which solves the essential problems outlined below and supports the networking requirements of a video game. The second portion of the project implements the designed protocol, creates a launcher for apps that would run on the network, as well as then creating a decentralized application within the launcher to showcase the platform.

Essential Problems

Blockchains Can’t Support Low Latency

Blockchain protocols traditionally have miners or consensus nodes produce blocks at set intervals. As an example, Bitcoin has a block propagated every ten minutes, Ethereum has a block propagated every fifteen seconds, and Steem has a block propagated every three seconds. Video games often have latency requirements in the milliseconds, where propagation time should be the only expected delay, not a arbitrary block timer.

Blockchains Can’t Support Mass Transactions

Traditional blockchain protocols cannot scale to support mass transactions with regard to storage. If a videogame transaction is stored on a blockchain and that video game has one gigabyte of data transferred in transactions per day, its blockchain’s size would scale at one gigabyte per day. This linear relationship between data sent and data stored is not sustainable. Much of this excess storage becomes irrelevant after an extended period of time.

Blockchains Aren’t Fully Distributed

Blockchains rely on a tier system of users. Regular users send transactions on the network, while a special class of users, usually referred to as miners, create and propagate blocks. These miners keep the network decentralized, however users are forced to communicate with miners rather than each other directly, preventing the system from being truly distributed. The miners are the primary bottleneck in the propagation time of transactions; their role is the cause of throttling transaction propagation to arbitrary block timers.

Blockchains Have Fees

Miners are expected to receive payment for their work as they burn electricity while securing the network. For currency transactions, fees are not necessarily a problem, because they are usually very small relative to the sum of currency transferred. With every transaction requiring a paid fee, certain applications become much more expensive than others. A cryptocurrency has a limited number of actions, such as sending currency from one user to another. In the realm of video games, because there is a greater number of actions, with each one requiring a fee, video games become very expensive to operate, making them undesirable for players.

Application Agnostic Blockchains Requires Overhead

Existing application agnostic blockchains, such as Ethereum, require a large amount of overhead. Running an Ethereum node is expensive, as it uses hundreds of gigabytes of data storage. Online video games are primarily web based and should be able to run in web browsers, which are limited to ten megabytes of offline storage. These tools are large, requiring their own programming language and virtual machines to constantly run.

Goals and Objectives

The Seed project is a lightweight framework for creating applications with a networking component that is powered by the Seed blockchain protocol.

Seed Protocol

The Seed protocol attempts to solve all of the listed essential problems, namely reducing storage size, removing arbitrary block timers, removing fees, becoming fully distributed, and requiring little overhead while running. This protocol will be developed with minimal dependencies and will offer a low-level API for decentralized applications to use.

Application Launcher

The Seed Launcher is an application launcher for decentralized applications that wish to exist in Seed. This launcher provides applications with a high-level API wrapper around the low-level API implementation of the Protocol. Applications are loaded dynamically by the launcher, allowing newly developed applications to be easily added.

Cryptocurrency Wallet

The Seed Wallet is the first application built for the Seed Launcher. This application acts as a cryptocurrency wallet for the Seed cryptocurrency. The Seed cryptocurrency is the first module developed, showcasing how a developer may use the Seed system. This wallet contains all functionality available in Ethereum’s ERC-20 tokens.

Development Documentation

The development of the project is documented on the website Steemit.com/@carsonroscoe, with the fully open source codebase available at Github.com/cajarty/seed. Various blogs and articles were written during development, explaining the rationale behind decisions, project designs, and development updates.

Body

Background

The Seed project focuses on creating a framework for secure peer-to-peer distributed application development. This is achieved through the use of blockchain technology, a decentralized open record storing protocol which achieves its high security through a methodical use of cryptography.

Though blockchain technology has many benefits, it also has its drawbacks. The Seed protocol attempts to learn from blockchain technology, inheriting its benefits, but taking a new approach on combating the many flaws. Notable problems the Seed protocol wishes to solve include: reducing the message validation time for lower-latency applications, reducing the data storage size, and creating a protocol which effectively scales with usage.

Blockchain Technology

Blockchain technology was invented by Satoshi Nakamoto during the development of the Bitcoin protocol in the year 2008, which outlines the blueprint for the Bitcoin cryptocurrency. Blockchain acts as the ledger for the Bitcoin protocol.

Blockchains are a very secure data structure. Their security comes from a combination of well-used cryptography for securing transactions and a validation mechanism known as proof-of-work for securing blocks. Alternatives to proof-of-work have been developed, however from a security standpoint, proof-of-work remains the dominant block validation mechanism.

Proof-of-work, despite being a strong validation mechanism, is also the root cause for many of the flaws Blockchains experience. Proof-of-work requires incentivising block creation, splitting the interests of users and validators. It can require a tremendous amount of energy usage, as showcased through Bitcoin’s validation process, which burns 1% of the world's electricity.

Literature Review - The Flaws of Proof-of-Work

Introduction

Blockchain technology has been a rapidly growing interest over the last few years in technology focused communities, as well as in investment focused communities. From blockchain technology grew Bitcoin, the first blockchain based digital cash to solve the double spending problem. Bitcoin paved the way for a new genre of digital currencies known as cryptocurrencies, most of which rely on the proof-of-work mechanism developed by Satoshi Nakamoto during the development of Bitcoin. Proof-of-work is a mechanism used for determining the order in which transactions took place. In the system a type of user referred to as a miner utilizes their GPU time to compete in a race over hashing and validating a block of transactions. The first miner to verify all the recent transactions were legitimate, and to show the appropriate work as proof to other miners, earns a payment in Bitcoin for their efforts [1][2]. This system secures the Bitcoin network and allows for alternating independent individuals to contribute to the decentralization of the blockchain, without any centralized servers or governing bodies. Proof-of-work has been successful in providing security to a decentralized network, however it isn't without consequence.

It results in a misalignment of interests between general users, who want a fast, low-fee network, and miners, who are motivated by returns on their investment. Proof-of-work is also purposely inefficient in it's processing usage, as it requires multiple miners to each spend processing time verifying the same transactions, with each miner expending energy trying to find valid work. My research question, "How would one replace competitive proof-of-work with cooperative proof-of-play for decentralized blockchain game servers", is aimed at addressing the issues with proof-of-work and layout an alternative tailored for a decentralized video game server.

Misalignment of Interests

The first consequence of proof-of-work to address is the misalignment of interests between general users and miners. A user in a cryptocurrency network has a variety of interests, including the security of their assets, security of transactions, cost of transacting, ease of

transacting and transaction confirmation times. The interest of miners, however, is one of profit. Miners are in competition with one other to be the first miner to mine a block. Miners pick and choose which transactions make it into their blocks, allowing them the freedom to prioritize transactions based on which would pay them the greatest in fees. Miners prefer less competition to gain a higher portion of the total hashing power, despite this desire conflicting with the desire of a secured decentralized system [3]. Miners may also resist future changes that benefit users, if those same changes do not benefit the miners. This phenomenon has occurred multiple times on the Bitcoin network, with the most noteworthy example in recent history occurring on August 1st of 2017. A fork occurred, which is a split in a blockchains network, where one part of the Bitcoin ecosystem chose to move forward with changes to Bitcoin which would solely benefit users while hurting miners, and the rest stayed back, refusing to upgrade.

This upgrade caused a change in the mining algorithm which would break certain existing mining hardware, however heavily benefit users in terms of transaction speeds, transaction fees, confirmation times and would also heavily benefit the decentralization of the network. These speed benefits would not be immediately seen, however would be enabled by the protocol upgrade through second layer solutions. The current miners, having sunk costs into such hardware and desiring to keep the system centralized to just those who have such hardware, refused to upgrade. On the other side of the debate, the clear majority of users accepted the upgrade.

This conflict resulted in a split of the Bitcoin network, creating a new coin known as Bitcoin Cash, which is effectively the legacy Bitcoin version from the miners who refused the upgrade. Human nature has shown to be the cause of virtually every cryptocurrency split, with proof-of-work being one of the frequent mechanisms causing the lack of unity in these cryptocurrency ecosystems. Creating an alternative where different users do not have separate roles in the system is crucial in aligning these interests to better avoid these issues in the future. As with any community or group, if the interests of all members are in harmony, the group will function smoothly. It's when people become at odds with each other, desiring different outcomes, that humans begin tripping each other up. Although this misalignment of interests may not be a issue from a technical standpoint, it has been one of Bitcoins biggest flaws from a socioeconomic standpoint.

Inefficiencies in Proof-of-Work

The next consequences to address are those that stem from the proof-of-work mechanism's inefficiencies. Proof-of-work forces miners to do duplicate work in a race with each other, each attempting to be the first miner to solve their block. This means multiple miners are validating the same transactions, and then discarding their work once they see a new completed block to mine. Once all the transactions have been validated, but a finished block is still not received from other miners, the mining system has miners keep trying to find a hash for their block that starts with a set amount of zero bits [1][2].

The energy waste of proof-of-work is also a terrible side effect that is required to maintain its security. This artificial work for proof-of-work both throttles the system, as well as wastes processing power doing work whose whole purpose is to simply determine which miner gets the right to determine the current block. The proof-of-work mechanism boils down to simply being a solution to a synchronization problem. The equivalent problem, if written in a threaded software program, would be one where multiple threads want to write to the same buffer while validating what the other threads wrote. The proof-of-work solution is to keep randomly guessing a number, and have the first thread who guesses the correct number being allowed to write to the buffer. The system is secure and reliable at decentralizing block creation, however inefficient in how it goes about these goals.

If we analyze the problem through the lens of a synchronization problem, as mentioned before, we may find different alternatives that may be more efficient. These alternative solutions to the synchronization problem could be letting the threat with the most to lose write to the buffer, which is very similar to proof-of-stake. We can theorize other solutions following this model which do not require the wasteful work, such as letting the last trusted thread nominate the next thread who gets to write. This does away with the energy wastefulness of proof-of-work specifically, however it doesn't address the bottleneck this extra step puts on the system. What this mechanism causes is a delay between transaction propagation and block propagation. In layman's terms, it means that there is a artificial delay between when a user says "I paid Bob $10", and from when Bob will see a block that has a transaction notifying them that they received the $10. This delay exists in all competitive mechanisms, not just proof of

work, however proof-of-work is the only mechanism that does it in a energy-wasting manner rather than strictly a time wasting manner. This extra step between transaction propagation and block propagation needs to be addressed, as it's been an accepted requirement for proof-of-work alternatives that remains a bottleneck in all the systems.

Alternative Mechanisms to Proof-of-Work

Alternatives to Proof-of-Work currently exist, which various other papers have discussed in the past. The most common alternative mechanism seen in cryptocurrencies is the proof-of-stake mechanism, which allows users to view their cryptocurrency holdings as their stake in the network, and get voting rights based on that stake [3][4][5]. Another alternative mechanism is a hybrid approach of proof-of-work and proof-of-stake known as proof-of-activity. These existing solutions succeed in being alternative mechanisms for a blockchain based cryptocurrency, however they do not work perfectly, in their raw form, for blockchain products that do not have the same technical requirements as a currency. Proof-of-stake relies on having a form of cryptocurrency to act as your holdings, whereas not all blockchain products necessarily possess a dedicated token or cryptocurrency in their implementations. Products may have different requirements that currencies do not have, such as requiring near-instant transactions, or near-instant confirmations.

One example of such systems would be a video game server, which uses blockchain technology to run a decentralized online video game while maintaining a consensus on the games state across every player. Such a system would not have a dedicated currency for users to own a stake in without relying on that stake being a video game currency, which would be ineffective due to the lack of wealth equality in video games when comparing inexperienced players to veterans. A system would be required where the top ranked players could not simply pool their power together to break the system, which is what would most likely occur in a proof-of-stake implementation. Proof-of-activity is an effective alternative in theory, however it relies on two rounds of communication to establish a block, whereas proof-of-work, proof-of-stake and proof-of-burn do this in one round of communication [6]. This results in proof-of-activity having to deal with double the latency, which is not a constraint that most online games can accept.

Another, interesting aspect of alternative mechanisms that is not explored often is changing the socioeconomic approach to a validation mechanism. Bitcoin and other cryptocurrencies use what I will refer to as competitive proof-of-work or competitive proof-of-stake. The Steem blockchain uses what has now been referred to as subjective proof-of-work, where the subjectivity of a users opinion determines how they upvote content, vote witnesses and flag content. Changing the approach to a validation mechanism is a valid way to alter the mechanism without necessarily losing the advantages it gives, depending on what the change is of course. This change of approach has worked on the Steem blockchain, however has not been without some criticism. Regardless, it has proved that a change in socioeconomic viewpoint on implementing existing mechanisms can majorly change how a system functions while maintaining the security given by the mechanism.

Therefore, competing mechanisms to date have had two avenues of exploration, changing the underlying mechanism (proof-of-work, proof-of-stake, etc) and changing the socioeconomic approach (competitive, subjective, etc). Various mechanisms have their own improvements, faults and criticism, however on the research side this is lacking. Essentially, there are a lot of case studies of single cryptocurrencies attempting a change, without much publicly available research comparing these differences in controlled environments.

Conclusion on the Flaws of Proof-of-Work

The Proof-of-Work mechanism requires a large amount of energy consumption to secure the network, which may be replaceable with a more efficient protocol energy-usage wise. Proof-of-stake is the main competitor, which uses substantially less energy on average, however still suffers from various other issues with proof-of-work. Proof-of-stake and other cryptocurrencies require a two-step propagation technique where first a transaction is propagated to the network (e.g. Bob sends $10 to Bill) and second where a block containing the transactions is propagated to the network after validation (e.g. Bill see’s in the new block that he received $10 from Bob). This validation step occurs due to the model most mechanisms follow where a certain type of user (Miners, Consensus Nodes, Witnesses, etc.) handle block creation and validation. This step adds an extra delay, as well as can often misalign the interests of users, since the general users of a network may desire something different than the people creating and validating blocks. For example, in Bitcoin, users want a quick network with low fees

while maintaining the security, while miners are incentivized to earn as much money as possible for their hardware investment. These misalignment of interests can lead to a split in the community or often forks in the network between competing camps.

None of these methods, however, is perfect in any way, they all have their strengths and weaknesses. One strength that has not been proven in any mechanism, however, would be a mechanism that minimizes ping, maximizes scalability and can handle high demand networking requirements. For example, there is no protocol that has shown to reliably handle a near-real time MMO in a scalable manner.

From a mechanism viewpoint, I believe it has not been explored enough treating transactions as modifications of blockchains by causal effect relationships. Following a pattern like this, it may be possible to take a game world, implement peoples changes, playing on their behalf, and using this gameplay itself as a form of validation that the worlds are in agreement.
From a socio economic viewpoint, I believe it has not been explored enough in blockchain technology to view transaction confirmation as a cooperative activity of users validating one-other, rather than a competitive activity of miners trying to beat one other.

The lack of alternatives that fit this networking niche use of blockchains, as well as the beliefs I've outlined above, both lead to the question, "How would one replace competitive proof-of-work with cooperative proof-of-gameplay for decentralized blockchain game servers". The previous papers I've referenced on proof-of-stake, and proof-of-activity all attempted to fix various parts of proof-of-work, each under their own constraints. Other papers I've read, specifically on Smart Contracts [7][8] but also on transaction channels [9][10], have shown how one can execute useful work on the blockchain in a timely manner, with transaction channels doing so utilization a second layer solution. My research question is effectively intending on creating an alternative mechanism to proof-of-work which treats all users as equals, does not inherently require a underlying currency (though it may), and allows for gameplay to execute across the blockchain in near real-time conditions.

Problem

Problem Statement

It is unknown how a cooperative mechanism affects a cryptocurrencies efficiency, nor how a blockchain protocol can meet real-time server requirements. In this context, a cooperative validation mechanism refers to one where every user cooperatively validates each other, rather than designated users competing for the right to be the validator.

Sub-Problems

1. To determine how historic block storage size can be reduced in order to make blockchains effectively scale with usage.
2. To determine how directed acyclic graphs can be used to create a low-latency, secure environment for decentralized applications.
3. To determine how directed acyclic graphs and blockchains can be combined to create a hybrid approach.
4. To determine whether blockchain technology can meet a low-latency, high-throughput video game's networking requirements.

Research Question

Following the problem statements driving this research, the research question is "How would one replace Competitive Proof-of-Work with Cooperative Proof-of-Play for decentralized blockchain game servers".

Chosen Solution

Cooperative Proof-of-Play

The proposed validation mechanism of Seed is referred to as cooperative proof-of-play. The mechanism is cooperative, as transactions validate each other in a distributed manner, rather than rely on competitive miners to validate the transactions. Proof-of-Play is similar to proof-of-work, however where proof-of-work has miners expend computing power to find a valid

nonce, proof-of-play has users simulate the execution of previous transactions to validate their honesty. Users simulate each others code execution, cryptographically signing the results of the execution. Users validate each other through play, rather than work.

The decision for a cooperative approach was explained in the conclusion of the literature review on the flaws of proof-of-work. In short, the competitive nature of miners in proof-of-work leads to a split between the interests of miners and users. These differing interests leads to misaligned goals among users and other socioeconomic problems in the traditional cryptocurrency model. A cooperative approach which keeps all users equal in the protocol would align the interests of the user base, potentially fixing these issues.

The decision for developing a proof-of-play mechanism, rather than use an existing proof-of-work alternative, stems from the end goal of the Seed project. Seed is attempting to create a framework for distributed applications, where the sole action of each transaction is executing code for applications. In order to accomplish this in a distributed manner, users must process each others tasks and validate each other at some point in the process. This differing requirement changes what must be accomplished during validation, making other alternatives such as proof-of-stake unfit in Seed's particular use case.

Transaction Squashing

Transaction squashing is the act of merging sequential ledger updates into one single update. This act of squashing condenses the size of transaction data, however it also harms Seed's ability to validate historic data. The chosen implementation would remove all error correction abilities, however retain enough validation information to handle error detection. Transaction squashing could only be safely done on transactions which are already deemed valid by the validation mechanism. Transaction squashing simply acts as a way to reduce the storage size of validated historic data.

Directed Acyclic Graph & Blockchain Hybrid

There are various emerging design decisions to be made when structuring a softwares blockchain, as well as whether to use a blockchain at all. The Seed project will take a hybrid approach between a blockchain solution and the more recently emerged directed acyclic graph (DAG) solution. The intention is to initially have transactions be connected through a DAG,

validating each other as they are added. Once transactions become valid, they may invoke the squashing mechanism, squashing multiple validated transactions together into blocks on the blockchain. Each block then represents multiple condensed updates of the ledger. This approach attempts to minimize the required validation time of transactions while also drastically reducing the blockchain storage size.

Details of Design and Development

Cryptography Design

The cryptographic elements of the Seed blockchain are required to handle user validation, user authentication, user consent confirmation and the correctness of agreed upon states of the blockchain. Private and public addresses will be required for each user, where their private keys are used for identifying users. Cryptographic signatures will be sued to validate and authenticate users transactions on the network. Elliptic Curve Cryptography (ECC) will be used as our cryptographic approach of choice due to its efficient calculation time and its high security.

Private Key

The private key generated is a 256 bit key. This key will need to be very secure, as it will act as the entry point into any user, and be required to sign all transactions.

In order to raise the security in generating private keys, our user interface will request the users give entropy. This entropy could be mouse movement changes, ten miscellaneous words or even user entered text.

Public Key

Each private key will generate one public key on our network, which is also 256 bits. More specifically, on the elliptic cryptography curve, a public key is an X coordinate that offers two possible Y coordinates (one positive, the other negative). Therefore, we can store the public key in 257 bits, with 256 bits for the X coordinate, then one bit for the sign to denote the positive or negative Y chosen.

The public key will be publicly available to all, however will generally be passed around in a base58 check encoded form known as a public address. This public key will be the unique identifier each users account is stored under, acting as the unique ID of each account.

Public Address

The Public Address is a base58 check encoded version of a users public key, which goes through the encoding with the network identifier appended to the end. What this effectively does is allow us to take a base64 encoded hexadecimal string and turn it into base58, a more human-readable format for the address. The reason it is more readable is base58 excludes certain characters that may cause confusion. Examples of omitted characters are a capital letter i, which can easily be mistaken for a lower-case L.

By appending the network identifier to the encoding, the same private key/public key combination can have a unique address for each network, allowing the same user to exist for both a private and public network with separate addresses in order to avoid sending the funds through the wrong network.

Transaction Signing

Transaction signing is where a private key and arbitrary data goes through a mathematical process in cryptography where that private key creates a cryptographic signature of the data. When this occurs, any user can validate the signature with the public key relating to the private key which signed the data, allowing for the authorization of commands with provable consent, without users ever making their private key public.

Elliptic Curve Digital Signature Algorithm

The type of cryptography we decided to go with was Elliptic Curve Cryptography (ECC). This cryptography is generally very easy to recreate, however is very difficulty to break with a small amount of bits, making it a prime candidate for cryptocurrency cryptography.

The Elliptic Curve Digital Signature Algorithm (ECDSA) is an algorithm used with the ECC to allow for transaction signing. It it used by Bitcoin and Ethereum with great success, and after doing a further analysis, our conclusion was that this was still the most reliably route we could go during the design phase.

Secp256k1 Curve Parameters

Secp256k1 refers to the parameters of the ECDSA curve and is defined in the Standards for Efficient Cryptography (SEC). Most elliptic curves are created with a random form, however the Secp256k1 curve is created in such a way that its shape is fully deterministic and easy to compute. It is estimated to be roughly 30% faster to compute compared to other commonly used ECDSA curves.

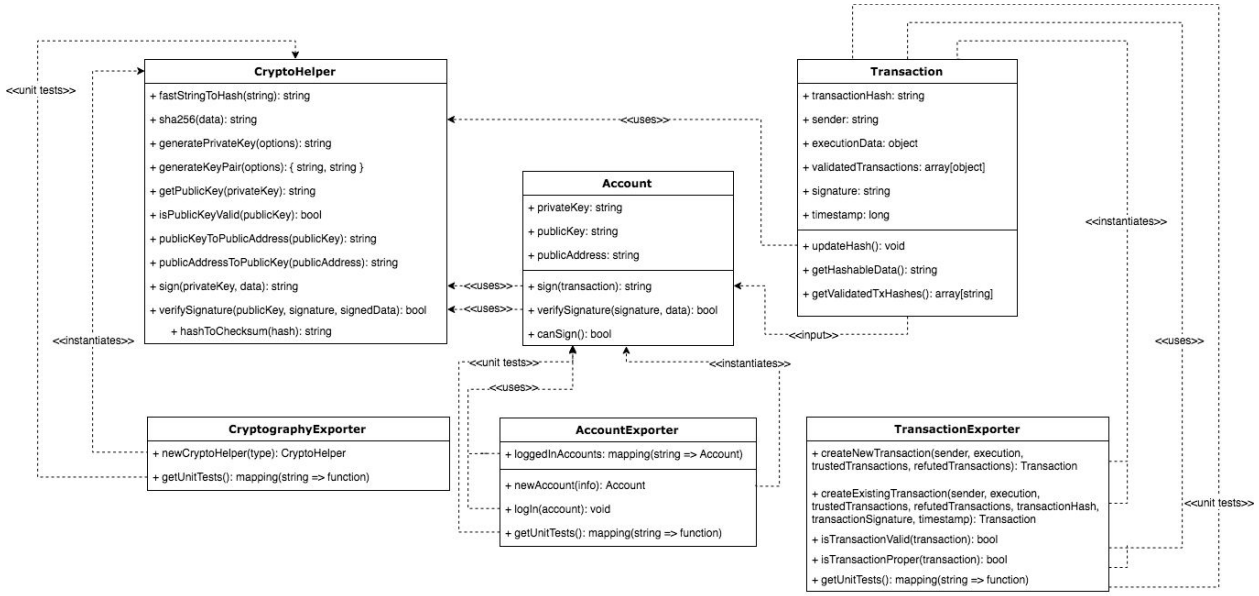
Following this standard also proves that nothing extra was added to the curve by developers, while also allows more confidence to be had in the security of the curve in general.

Nested SHA256 Hashing

SHA-256 appears to be the safest hashing algorithm that we can use while minimizing computation time and the amount of bits per hash. SHA-1 previously has been deemed not secure due to a vulnerability known as “The Birthday Attack”. This attack is believed to not be possible in SHA-2, however some experts still fear we may discover this attack in the future. However, specialists agree that, should such an attack be likely to occur, or should SHA-256 be proven to be less secure than originally anticipated, nesting SHA-256 hashes would raise security sufficiently over a simple single-pass hash.

Therefore, in all cases where hashing must be done, such as for hashing transactions, nested SHA-256 hashes will be executed.

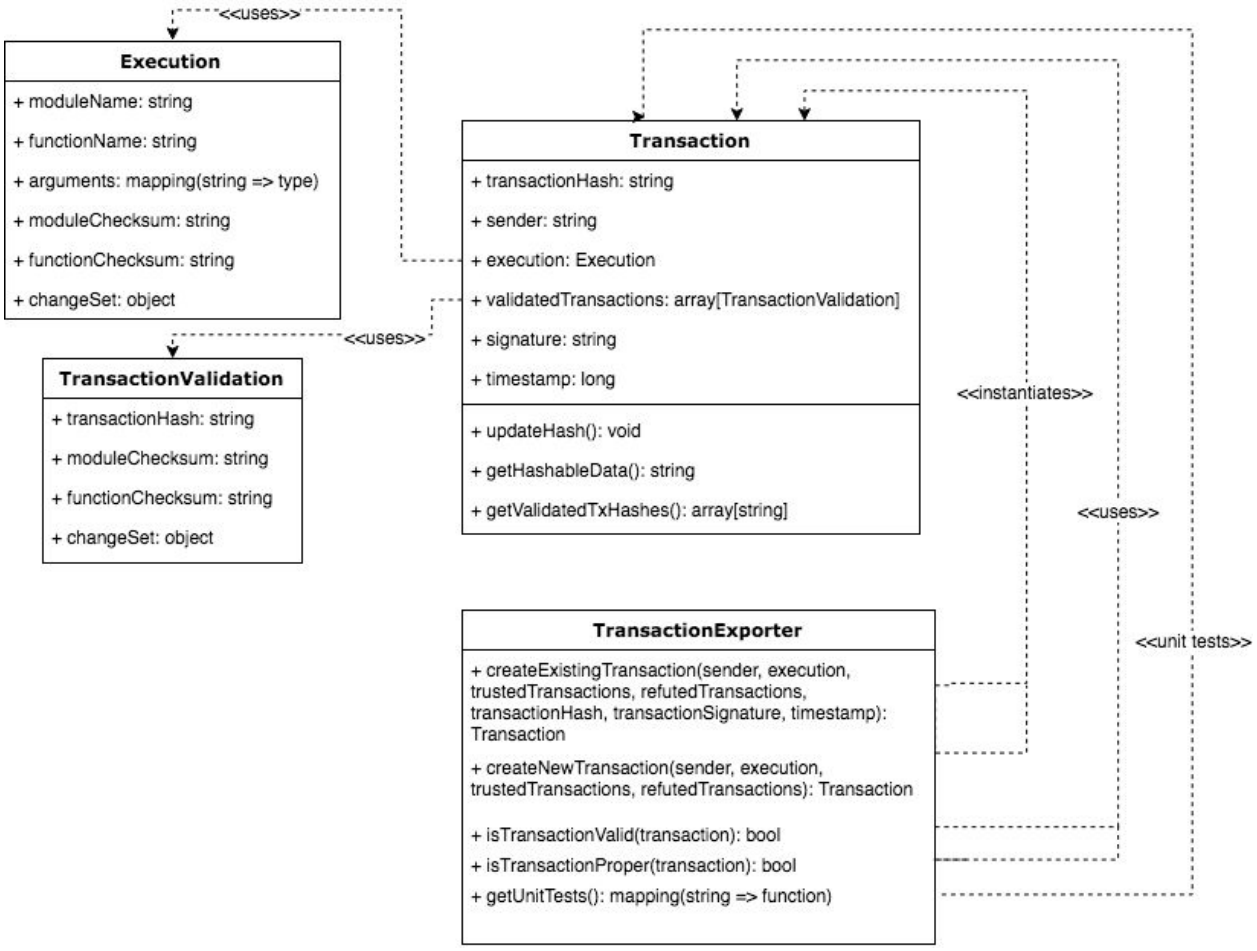
Class Diagrams



Transaction Design

A transaction is a cryptographically signed message outlining how the sender intends on changing the ledger through the execution of a Module's function. Because it is cryptographically signed, it allows us to prove the intent of the sender, and confirm their consent to any changes this message causes. In Seed, a transaction represents code execution, and therefore must describe exactly which code to execute. These messages and their executable code must be fully deterministic.

Transaction Class Design



Validation Rules

Rule	Description
#1	Validate that the hash of all the transaction data creates the same hash as the claimed transaction hash
#2	The transaction sender must be a valid public key format and be derived from a valid private key
#3	The transaction's cryptographic signature must be verifiable by the transaction sender, proving consent

#4	The transactions which are validated by the incoming transaction must be verified by other users
#5	This new transaction and its validate transactions cannot create a cycle in the DAG when added
#6	Prove that we agree on the module code being executed, so we're using the same versions
#7	Prove that we agree on the function code being executed, so we're using the same versions
#8	Prove that, when we simulate the execution, we get the same ChangeSet, and therefore agree on how it changed the ledger
#9	The validation work done must match the transactions they attempt to validate
#10	Prove that, when we simulate the execution of their validated transactions, their execution was also right (Prove their "work" was right).
#11	Prove that a user does not validate them-self
#12	If a transaction refutes another transaction, it cannot be added to the entanglement as long as the refuted transaction exists.

Block Design

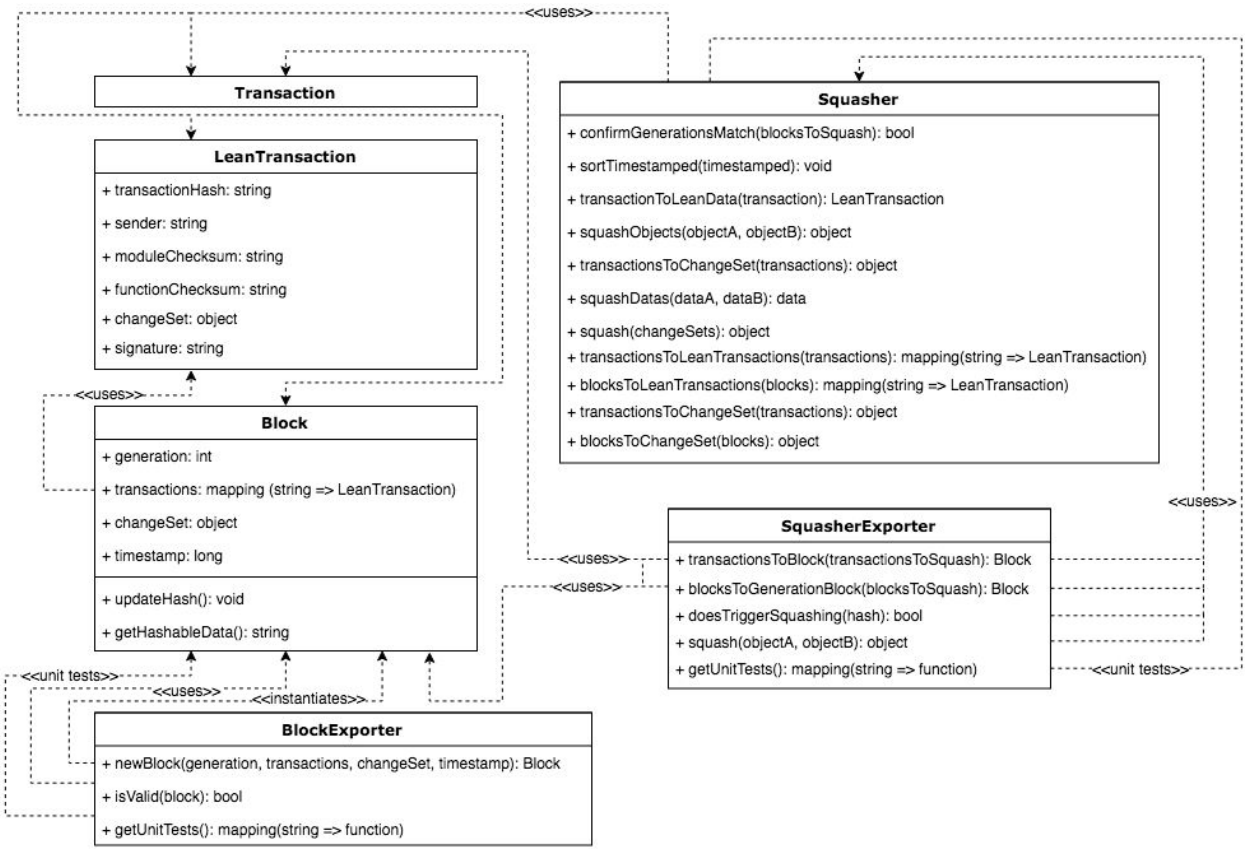
A Block in Seed is not exactly the same as a traditional blockchain Block. In traditional blockchain systems, blocks are fully verifiable, as they are a key component of the validation system. In Seed, transactions are already valid once they enter a block, removing the requirement for blocks to handle validation. In traditional blockchain systems, a miner creates the block, finding a nonce by executing a computationally heavy problem solving activity known as mining. In Seed, blocks are fully deterministic in their creation, with miners being excluded from the picture.

Squashed Data

Due to transactions being validated by the entanglement, all blocks are inherently trusted. In order to keep the blockchain lean, the squashing process removes a large amount of the information for each transaction. The intention is to keep enough information that users can validate a blocks legitimacy, without needing to know all the information that came with each transaction.

When a block is created, all transactions have key pieces of information extracted. Enough information that a user can execute all of the block's transactions and determine how the transactions would have modified the ledger. These combined changes to the ledger can also be found inside the block itself, allowing the legitimacy of a block to be validated. If the execution of all transactions matched the block's squashed changes, the claimed changes are honest.

Block Class Design



Validation Rules

Rule	Description
#1	Validate that the hash of all the transaction data creates the same hash as the claimed transaction hash
#2	The transaction sender must be a valid public key format and be derived from a valid private key
#3	The transaction's cryptographic signature must be verifiable by the transaction sender, proving consent

Transaction & Block Squashing Design

Transaction squashing is the simple concept of merging sequential changes into one single change. In Github, a technology created to allow for open source code contributions, a directed acyclic graph (DAG) is used to manage user code changes, where each node in the DAG is a commit of changes. The git approach has already proven the ability to squash multiple sequential commits into one merged commit, while still being done from a DAG. The squashing algorithm used differs from how Seed will handle squashing, however the general concept has been proven

In Seed’s implementation, a validated transaction which meets certain criteria will trigger a squashing mechanism. This will take a group of validated transactions, extract them from the DAG, and squash them into a condensed block.

Squashing Data

When a transaction triggers the squashing mechanism, it and all validated connected nodes are pulled from the Entanglement. Afterwards, an empty Block is created. Each transaction pulled from the Entanglement has various properties extracted, including the transaction sender, executed module’s source code checksum, the executed function’s code’s checksum, the arguments passed into the function upon executed, and the cryptographically signed signature which the sender signed upon the transactions creation.

Afterwards, all of the changes each transaction applied to the ledger are squashed.

Once the block is finished being created, it is considered a "first generation block" and added to the blockchain. The selected transactions are removed from the entanglement, with the transaction which triggered the mechanism being replaced by the newly created block's hash.

Relative vs. Absolute Changes

Squashing relative changes is less complicated than absolute changes. That is, squashing "Bob sent Jill $10" and "Jill sent Bob $5" into "Bob sent Jill $5" is easy. Squashing "Bob set the sign

to say Hello" and "Jill set the sign to say World" is hard, since whether the end result is "Hello" or "World" comes down to jitter and timestamps.

In order to accomplish this for consistency for absolute changes, as all blocks must be fully deterministic in their creation, with transactions having their timestamp declared upon their creation. This timestamp is stripped from them during the squashing phase, however it is used for ordering the transactions inside the block. As long as we know the order is known, setting absolute changes works properly.

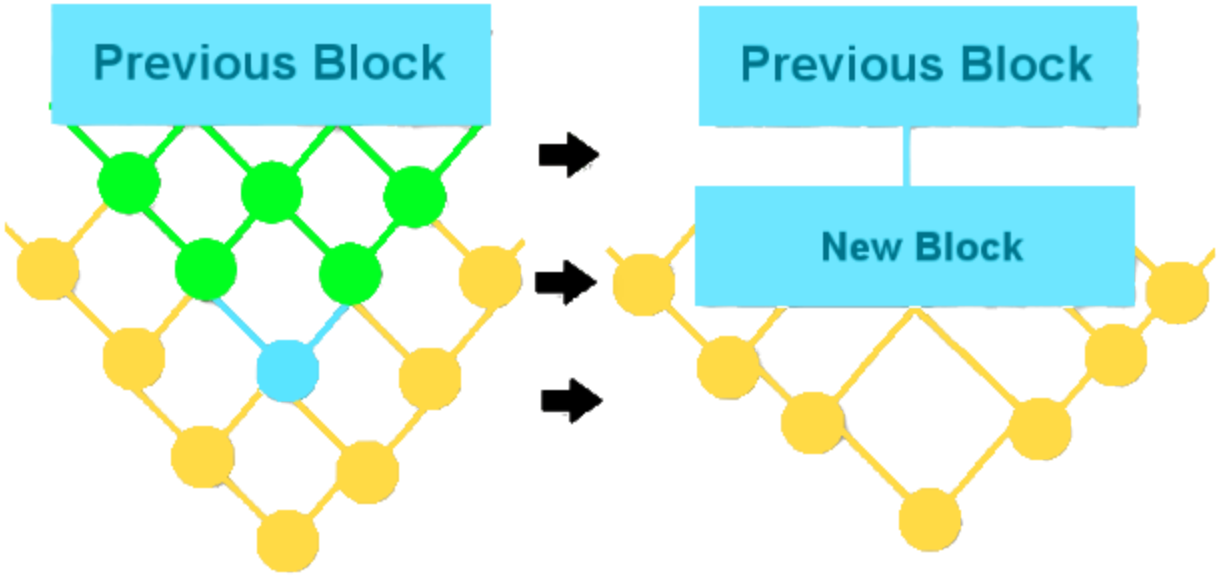
This timestamp proposes a minor flaw in the protocol. At an application level, users cannot lie about their timestamps by a large margin, as stating they are from the past too far may make them ineligible for validation, and making the timestamp from the future would fail validation checks done by other users. However, users could lie about their timestamp within a short range of time, which allows for plenty of opportunity to reroll their transaction hashes. Through rerolling their hashes, they could purposely create transactions which invoke the squashing mechanism, making squashing occur more often than expected.

This, however, is not a major flaw. There is no incentive to do such a thing and no benefit to gain from such an act. There is also no true harm overuse of this mechanism could cause. The possibility for the proposed abuse does exist, however.

Triggering Squashing

Transactions or blocks can both trigger the squashing mechanism. If a transaction triggers the mechanism, it and all upwardly connected transactions in the DAG will be squashed into a block. When a block triggers the mechanism, it and all other blocks of the same generation will become squashed as well.

Below is a diagram to explain this relationship.



In the diagram, the blue node represents a newly validated transaction that is triggering the squashing mechanism. The green nodes represents the other transactions that are being squashed. The green nodes were validated by the blue node. If the blue node is valid, all the green nodes must also have been already considered valid.

The green nodes are all upward nodes directly or indirectly validated by the blue node. This relationship continues upwards until either the first node of the entire DAG is found, or until a block hash is found. The blue node and all green nodes represent the transactions which will become squashed, as depicted in the diagram.

The orange nodes represents transactions which would not be squashed in the mechanism. These may be valid and may validate transactions which are being squashed, however they are not themselves squashed. They will remain in the DAG until a transaction which validates them also triggers the squashing mechanism.

Entanglement Design

The entanglement is the directed acyclic graph (DAG) used by the Seed project as an alternative to a traditional blockchain for transaction storage. This structure allows transactions

to be stored out of order, joining the graph asynchronously, providing an alternative validation mechanism for determining how much trust a transaction is given.

DAG

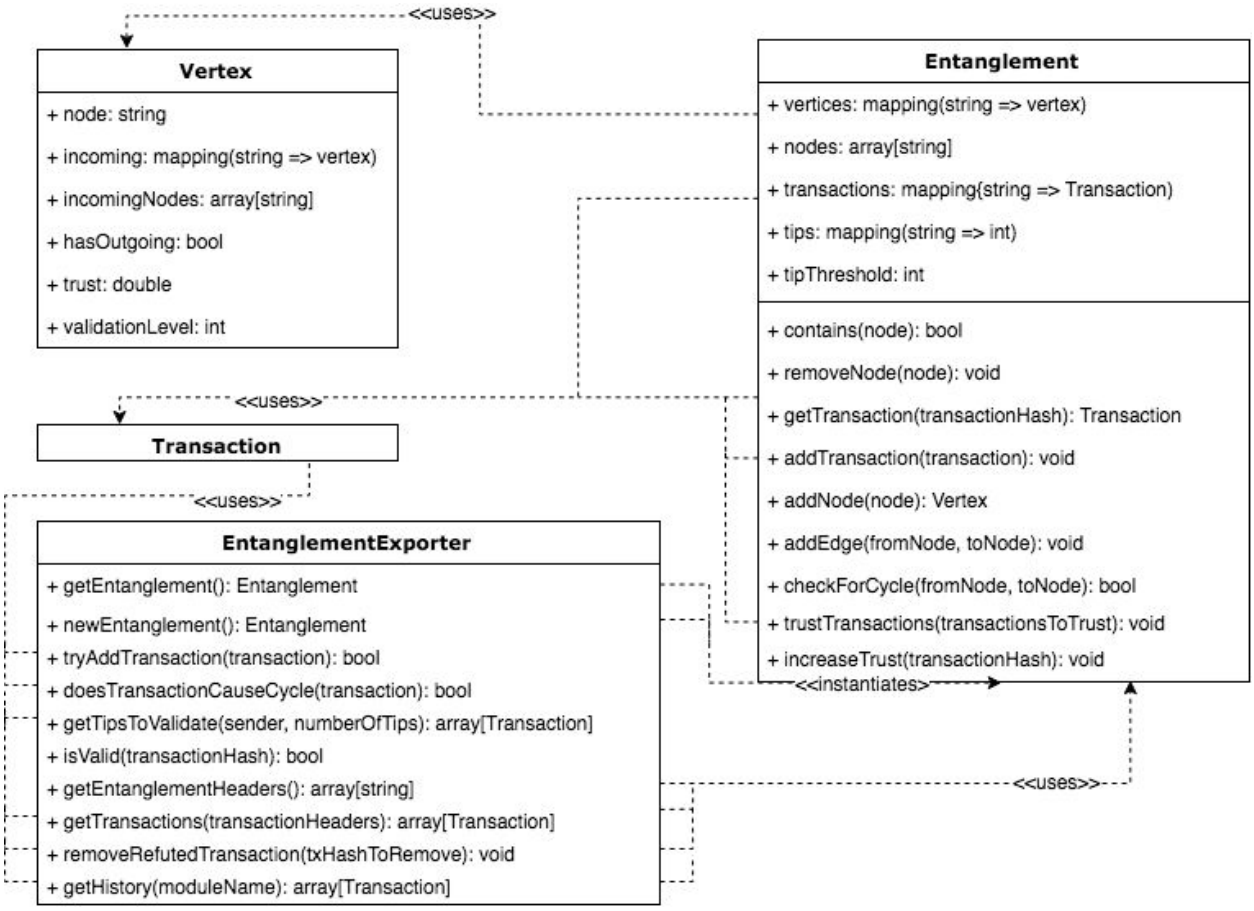
Directed Acyclic Graph solutions have become a very prominent contender for a blockchain alternative technology. Relying on transactions to validate other transactions removes the requirement of miners, making the validation process a truly distributed process rather than a decentralized process run by several miners. The DAG approach is much more ecologically friendly, as the mining process in proof-of-work cryptocurrencies is a very computationally expensive process. Although the DAG approach does require a form of proof-of-work present in validating previous transactions, it isn't the same as the expensive blockchain equivalent. Where blockchain transactions are executed synchronously, DAG transactions arrive out of order and are executed asynchronously. There is no true order, however DAG's can be flattened into an imperfect ordering if required.

Validation

Transactions in the DAG are validated by other transactions. Therefore, there are a number of factors that play a role in how much confidence a transaction is given. For example, the number of parents who validate the transaction plays a role, as well as the trustworthiness of the valid parents. The children transactions that were validated by the inspected transaction are also analyzed. If any of those children are not valid, it can be taken into consideration with regards to trusting transactions which claimed it were valid.

This validation process allows for the fuzzification of trust. That is, the ability to inspect the trust we have in a transaction on a quantitative comparable scale. This allows for applications to respond to transactions before they are fully validated. For example, a user interface may require certain transactions are 15% trusted before display their effect on the applications visuals, while waiting for more security critical functionality to reach full confirmation before reacting. This allows for interfaces to respond how they choose, depending on their activity and use cases. This fuzzification of trust gives an interesting dynamic view of trust, rather than the binary "confirmed" or "unconfirmed" equivalent found in blockchain technology.

Entanglement Class Design



Blockchain Design

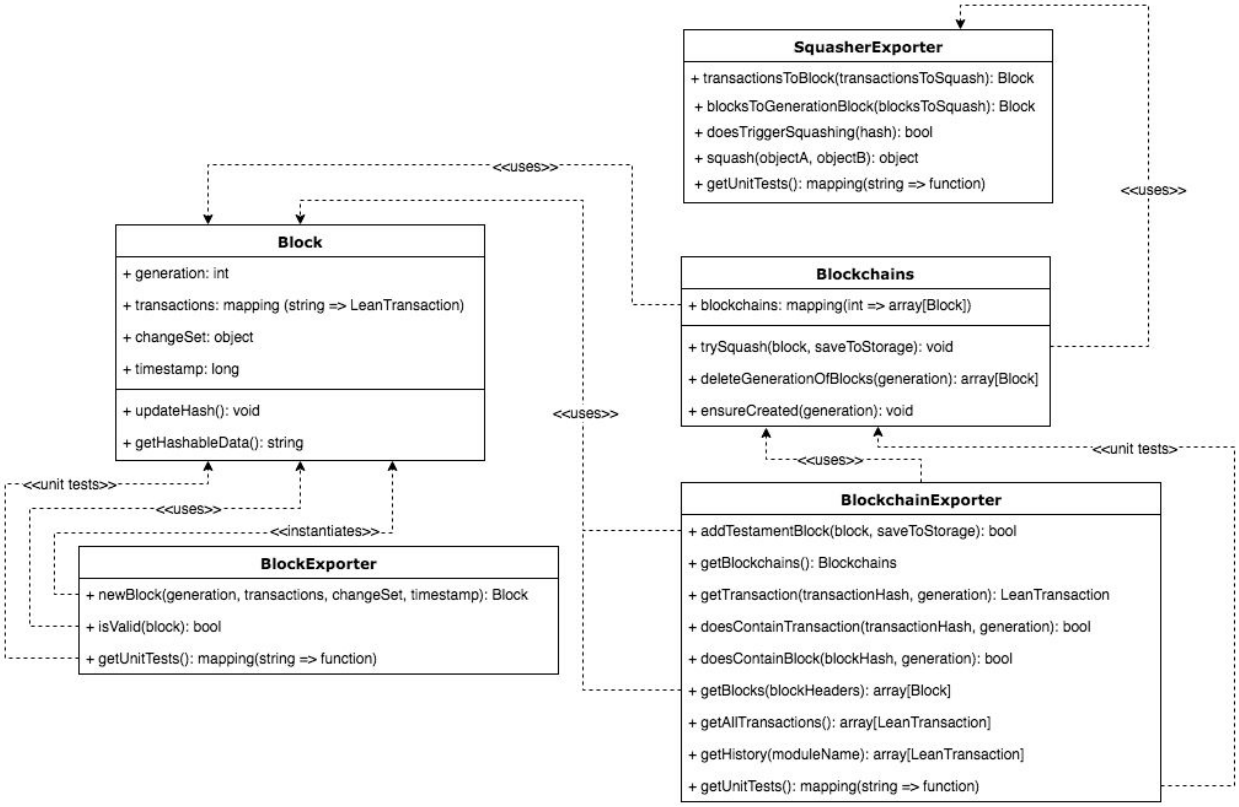
In traditional blockchains, a blockchain is simply a cryptographically secured variation of the linked list data structure. Each block is a node on this list, who knows only of the node before it. In Seed, blocks do not know of the blocks which come before it, and are stored in multiple lists rather than one.

In Seed, once transactions are squashed into a block, they are added to the blockchain as a "first generation testament block". If the newly created block's hash matching the requirements for triggering the squashing mechanism, all blocks of the same generation are squashed together and re-added into the blockchain, creating a more condensed "second generation" block. This cycle can repeat indefinitely, keeping the size of the blockchain to a minimum.

Block Squashing

Similar to transaction squashing, blocks can also be squashed into more condensed blocks. A group of second generation blocks can therefore be squashed into a third generation block. The same squashing algorithm is used as when squashing transaction data.

Blockchain Class Design



Module Design

Decentralized Applications

Modules are the portion of the Seed project which allows for the creation of decentralized applications. Decentralized applications are applications which are not hosted on centralized servers, and are instead distributed between users. The largest decentralized application system in the world is Ethereum, an ecosystem built on top of the traditional blockchain proof-of-work protocol.

Seed Modules are similar to Ethereum Smart Contracts, however are programmed in Javascript, with differing constraints, and no concept of gas to fuel the smart contracts.

Capabilities

Local Data

Modules contain local data pertaining to the module itself in the form of key-value storage. Storage can be either a number, a string or an object.

User Data

Modules also contain user data for every user which uses the module. This user data structure is defined at module creation time, with a copy of the data schema being added on behalf of each user upon the first time that the user is referenced within the module. Therefore, the first time a user uses a module, or the first time another user references a given user within a function, will result in the new user being given a blank set of data for the module.
The user data is simply a mapping inside the local data called "userData". For this reason, "userData" is a protected term that cannot be used within functions.

Functions

Each module has one or more functions associated with it. These functions give logic to the module and are all public, invokable by external users and other functions. Functions require a container parameter in order to read module data, however they can also require a second parameter, requesting a ChangeContext object, if they intend on modifying a modules state.

Module Creation

Below is a simple example of creating a simple module, whose state is one variable any user can increment.

```
let module = require("./seedSrc/module.js").createModule({
    module : "ExampleModule",
```

```
    initialData : {
        stateVariable : 0
    }
    functions : {
        incrementCounter : function(container, changeContext) {
            changeContext.add(1, { key : "stateVariable" });
            return changeContext;
        },
        getCounters : function(container) {
            return container.getModuleData("stateVariable");
        }
    }
});
```

This module has one local variable, "stateVariable", which is incremented any time any user invokes the "incrementCounter" function. "getCounters" is a getter function which returns the variable.

Seed Cryptocurrency Module Design

The Seed module is based heavily on the ERC-20 standard for creating Ethereum tokens. This common implementation of a cryptocurrency was ported over to the Seed ecosystem. This standard was copied due to its proven logic and being easily testable.

This interface includes methods for transferring SEED, giving allowances, transfering on other users behalfs, getting the symbol/total supply/decimals,etc. The full ERC-20 compliant standard.

Initial State Data

```
const initialSeedState = {
    totalSupply : 0, // Total supply of SEED in circulation
    symbol : "SEED", // Symbol of SEED cryptocurrency for UI's
```

```
    decimals : 4 // Amount of decimals used when displaying the SEED cryptocurrency.
Maximum divisible amount
}
```

The above code describes the initial state data of the Seed Module. Upon creation, there will be zero Seed in circulation. The display symbol for the currency is “SEED”, and the display will be rounded to the fourth decimal.

Initial User State Data

```
const initialUserState = {
    balance : 0, // A users SEED balance
    allowance : {} // How much SEED a given user allows other users to spend on their behalf
}
```

The above code describes the initial state data for each individual user upon their first time interacting with the Seed Module. Each user is assigned their Seed balance, which starts at zero as they do not have any Seed. Each user is given an allowance, which is a mapping regarding how much Seed each given user allows other users to spend on their behalf.

Seed.constructor

```
const constructor = function(container, changeContext) {
    let sender = container.sender;
    let initialSeed = container.args.initialSeed;

    changeContext.add(initialSeed, { user : sender, key : "balance" });
    changeContext.add(initialSeed, { key : "totalSupply" });

    return changeContext;
}
```

Constructor's are a special case function Modules may support. A Constructor is unique in that it can only be called once in a Modules lifetime. Once a Module has had its constructor invoked, it can never be invoked again.

For testing purposes, the sample Seed constructor takes a argument regarding how much Seed to award the creator. This information is read through the container, passed in as the first argument into the constructor. The container contains a read-only state of the ledger, transaction information and arguments, among other things such as access to a pseudo-randomness object. The state-modifying changes are applied to the ChangeContext object passed in as the second parameter. A ChangeContext offers read-write capabilities regarding how the ledger shall be changed once the execution is validated.

Seed.transfer

```
const transfer = function(container, changeContext) {
    // Gather readonly data
    let to = container.args.to;
    let value = container.args.value;
    let sender = container.sender;
    let fromBalance = container.getSenderData().balance;

    // Confirm adequate balance for the transaction
    if (fromBalance >= value && value > 0) {
        changeContext.subtract(value, {user : sender, key : "balance"});
        changeContext.add(value, { user : to, key : "balance"} );
    }

    return changeContext;
}
```

The transfer function is an example of a simple state-modifying function in Seed. Similar to the constructor, the transfer function takes two arguments, one which offers read-only information regarding the state, and another which allows for declaring how the state of the ledger should be modified.

The transfer function begins by reading information from the container regarding who is sending currency, who is receiving it, how much is being sent, and checking the value of the available funds from the sender. Through the container, the transfer function requests the user data of the sender, allowing access to read the state of their balance.

If the transaction sender has adequate funds, the change context is notified to subtract the funds from the sender, and add them to the receivers' balance.

Seed.getBalanceOf()

```
const getBalanceOf = function(container) {
    return container.getUserData(container.args.owner).balance;
}
```

The getBalanceOf function is an example of a getter function, which is a unique type of function which does not modify the ledger state. Instead, getters allow for reading the ledger and returning data to the caller. Getters have one parameter, requesting the read-only container.

Getters can be run locally without requiring the creation nor propagation of transactions.

Bundling The Seed Module

```
module.exports = {
    getModule : function() {
        return moduleExporter.createModule({
            module : "Seed",
            initialData : initialSeedState,
            initialUserData : initialUserState,
```

```
        functions : {
            constructor : constructor,
            transfer : transfer,
            transferFrom : transferFrom,
            approve : approve,
            burn : burn,
            getBalanceOf : getBalanceOf,
            getAllowance : getAllowance,
            getTotalSupply : getTotalSupply,
            getSymbol : getSymbol,
            getDecimals : getDecimals,
            mine : mine
        }
    });
  }
}
```

The Seed Module has its functions and state data bundled up as shown above. A name is given to the module, as well as the initial state data and what will be the initial state data for every user. A mapping of functions is added, pairing the name of a function to the variables the respective functions are stored in.

The function named “constructor” is a special case function, which can only be invoked once in a modules lifetime.

Networking Design

A blockchain protocol outlines a secure method to validate data being transferred, however it does not specify how data is transferred. Data, in the form of transactions, could technically be transferred via email, portable USB or fax if required. However, for the sake of usability, a more advanced form of transferring transactions is required. The majority of blockchains propagate their transactions through established peer-to-peer networks connected via software interfaces. The Seed project includes a Seed launcher, which does just that.

Chosen Peer-To-Peer Topology

One of the primary goals of the Seed network is to be a distributed blockchain system, giving no user more power than other users. This holds true at the Seed protocol level, however the same cannot necessarily be true for creating the networking level. A distributed network would be a full mesh topology, where every client is connected to every other client. Mesh topologies scale incredibly poorly with size, since each additional client increases every other clients required connections.

The decentralized approach blockchains traditionally take is to have a unique role known as a miner or a consensus node. These are special nodes which are part of the blockchain protocol itself. However, in Seed, no such role exists.

In order to minimize the amount of connections between clients, the simplest solution is to allow nodes to optionally act as a hub. A hub would be a special node which listens for other clients to connect. If a topology only has one hub, it is generally not secure as it has a single point of failure. However, if any client could choose to be a hub, there would be multiple points of upkeep, keeping the network online as long as one hub stays active.

The chosen design is to use these hub nodes and have them relay transactions between each other. These nodes will be referred to as Relay Nodes.

Client

Clients connect to listening relay nodes using web sockets. The Client instance will take the IP of their selected Relay Node as the parameter into the constructor.

Clients must be able to request blockchain and entanglement header information, as well as request specific blocks and transactions. They must also be able to propagate transactions through Relay Nodes.

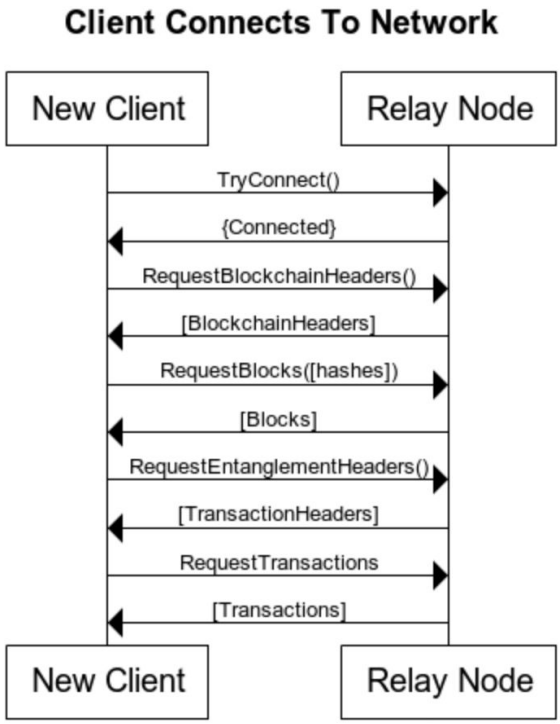
Relay Node

Relay Nodes listen for clients to connect, and relay messages between clients. The Relay Node instance will take an array of IP addresses for other relay nodes they wish to communicate with as the parameter in the constructor.

Relay Nodes must be able to listen for connecting clients, process all client requests, and request the initial state from other Relay Nodes.

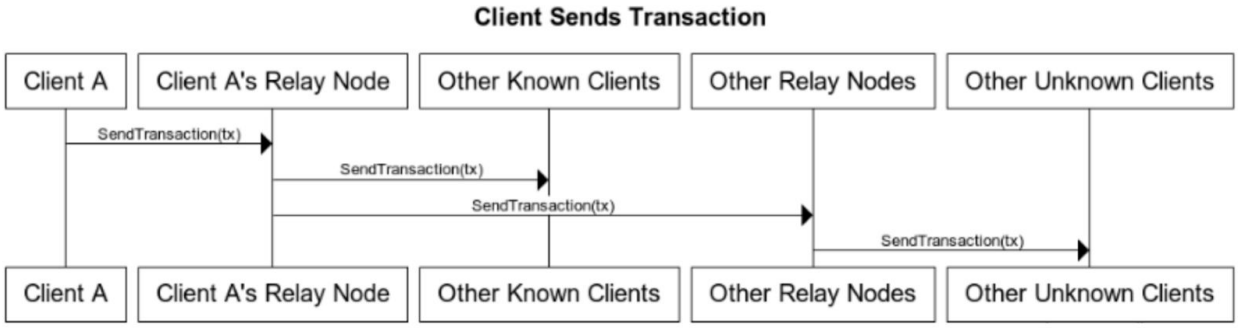
System Sequence Diagrams

Client Connects To Network



Clients must be able to connect to the network via any connected relay node. Once connected, a Client can request blockchain and entanglement header data to determine what data it needs to request. If its a clients first time running, they would request all block and transaction data. However, if the client has offline data already stored, they would request only block and transaction hashes that belong to transactions they do not already own.

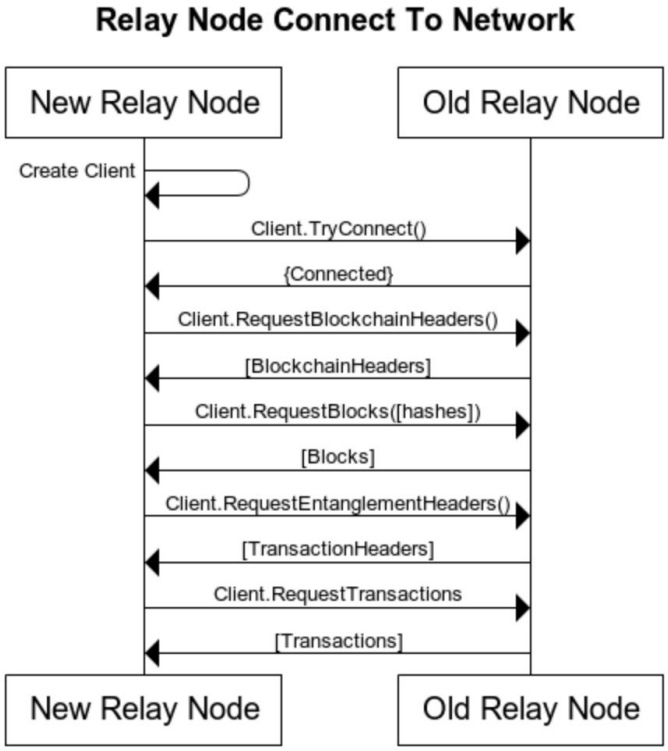
Client Sends Transactions



Clients must be able to propagate transactions to the network. A client would send their transactions directly to any relay nodes they choose. Relay nodes will then propagate the transactions to other users.

Relay nodes will also relay transactions between each other, routing transactions efficiently around the network.

Relay Nodes Connects To Network



Relay nodes must be able to connect to the network in the same way as clients. The simplest solution is to have relay nodes create clients which communicate with other relay nodes. A connecting relay node can then request through their client all the information needed to be caught up to date.

Storage Design

Whether a cryptocurrency is based upon blockchain or directed acyclic graphs, there is one aspect that is always certain; Blocks and transactions are meant to be stored. An app is not meant to stay open constantly, as users should be able to close an app without worrying about losing their data. Losing data would mean requesting more data over the network whenever a user tries to reconnect. The goal is to minimize network requirements as much as possible in order to achieve the goal of creating a high-throughput, low-latency blockchain solution.

Exposed Functionality

The Storage class will expose three notable functions to be invoked. The first is loading the initial ledger state from storage, next is saving a block to storage, and finally saving a transaction to storage.

Loading Initial State From Storage

When loading the initial state, the order of execution may matter for applying transactions and blocks. For this reason, blocks and transactions must be read in as close to proper order as possible.

First all blocks are read in by the database injector, then sorted by timestamp. They are then checked for validity and added to the blockchain, applying its changes to the ledger.

Once all blocks are processed, all transactions in the saved Entanglement are read, then sorted by their timestamps. These transactions are added to the live Entanglement one-by-one, validating that each transaction meets all validation checks.

Save Blocks To Storage

When saving a block, all transactions in storage which belong to that block must be removed by the database injector first. After removing all of the transactions belonging to the block, the database injector will write the block to storage.

Save Transactions To Storage

Saving a transaction must be done when creating a transaction to propagate, or receiving one from the network. These transactions are saved to the stored entanglement by the database injector.

Browser vs. Desktop Constraints

Due to the Seed low level API being environment agnostic, it cannot be assumed whether a user is running Seed on a web browser, a server, or an Electron app. As such, the storage implementation may or not have access to differing storage options, such as the file system, Local Storage or cookies.

Modular Storage With IDatabaseInjector

The differing environments added the constraint of modular storage. The act of "storing a block" or "storing a transaction" must not be concerned with regards to which platform is being used. However, the implementation behind that action must be able to choose which database provider best suits the current environment.

This requirement led to the creation of the "IDatabaseInjector' interface. Although JavaScript is a dynamic language and does not have true interfacing built in, objects can still be created with the same function names, and then be used interchangeably as if they were objects sharing an interface. This "interface" is more of a pattern to follow, which contains functions the Storage object will expect for usage.

The expected functions for all IDatabaseInjector implementations to have are the following:

```
readBlockAsync(generation, storageName, callback)
readBlockSync(generation, storageName)
```

```
readBlockchainSync(generation)
readBlockchainsSync()
readTransactionAsync(storageName, callback)
readTransactionSync(storageName)
readEntanglementSync()
writeBlockAsync(storageName, storageObject, generation, callback)
writeTransactionAsync(storageName, storageObject, callback)
removeTransactionAsync(transactionName, callback)
removeBlockAsync(generation, blockName, callback)
```

File System Storage Injector Implementation

The File System Injector implements data storage through the file system's read and write operations for managing multiple files in folders. Due to relying on the operating systems IO, this implementation of the IDatabaseInjector interface cannot run on all environments, such as certain browser web apps. This implementation was built primarily for the Seed Launcher and other Electron DApps, but should work on all systems which have file system read and write access.

Blockchain Storage Schema

Blockchains are organized by the generation of blocks within the chains. Within each chain is an array of blocks named by their block hash. Within each block is the data it represents.

This relationship can be easily represented with files and folders. In the base folder, /data/blockchains/, will be strictly subfolders named by which generation of blocks reside within it. If all of the blocks are first generation blocks, there would simply be one folder, /data/blockchains/1/.

Within each subfolder would be a bunch of files. Each file is in the .json file format, representing blob storage of an object. These files contain a single block, where the file is named after the blocks hash.

Entanglement Storage Schema

For the Entanglement storage, transactions are simply stored in a /data/entanglement folder. Each transaction is a .json file containing the transaction in JSON format, with the file name being the hash of the transaction.

Local Storage Injector Implementation

The Local Storage Injector implements data storage through a passed in LocalStorage object for reading and writing. Due to relying on the LocalStorage object, this implementation requires the environment to have DOM access, and is intended to be used for web apps.

Blockchain Storage Schema

As Local Storage follows a key/value pattern for data storage, the subfolders approach taken in the File System Injector is not viable. However, blocks must still be organized by generations, in order to be organized by blockchains.

Blocks are stored following a naming convention for their keys. The key is the block's generation followed by the block's hash, being separated by an underscore. For example, "1_BlockHash1" would be the key for storing a block who's hash is "BlockHash1" as a first generation block.

Entanglement Storage Schema

Transactions for the Entanglement are stored with their key being their transaction hash. Due to the underscore not being a character found in base56 encoded characters, it can easily be determined that if a underscore is found in the name, it's a block. Otherwise, if its a valid hash it has to be a transaction.

High Level API Design

DApps which choose to be hosted inside the launcher will have the luxury of accessing a High Level API for Seed. This High Level API is referred to as the SeedHLAPI. This API wraps all logic needed regarding how Seed works under the hood, allowing developers to create transactions, read from modules or subscribe for updates in a few simple API calls.

Electron Constraints

Electron brings its own set of constraints we must adhere to when building the HLAPI. The primary constraints we must respect is how Electron and the underlying Chromium system its built on communicate across processes.

Each process has its own set of memory, with each window being a separate process. The base process is known as "Main", which creates each individual window. Each window has its own process, which is known as a Renderer. Renderer's and Main communicate with one-other through Electron's IPC channels. These IPC channels cannot have JavaScript references sent through it, so all data or objects that are sent over IPC must be serializable.

Considering this constraint prevents the transfer of objects by reference, the processes cannot communicate with the same Seed API instances directly. Instead, Main must be the process that holds the Seed LLAPI instance used by the Seed HLAPI, while the Renderers must request that Main do the processing. These requests must be made through interprocess communication (IPC), as it is the most efficient way for the processes to share information with one-other.

DApp Requirements

DApps must be able to make HLAPI requests, wait for a response from a separate process, and continue their work once the execution of the HLAPI has completed. Therefore, the DApps require that the HLAPI is asynchronous, despite the LLAPI being synchronous.

DApps must be able to:

API	**Description**
Switch Accounts	DApps may request an accounts witch, as users may have more than one account they wish to use.
Get Account	DApps may want to know which account is currently logged in.

Create Transaction	DApps must be able to create transactions when a user chooses to execute a function in their module.
Get Transactions	DApps may want to fetch a transaction from storage by the transactions hash in order to read it.
Call "Getters"	DApps may want to call any modules getters, even other modules.
Read	DApps may want to read raw data from the ledger regarding a module.
Subscribe	DApps must be able to subscribe to function callbacks and data changes in order to live update appropriately.
Unsubscribe	DApps must be able to unsubscribe to avoid memory leaks and unwanted callback invocations.
Create Modules	DApps must be able to create modules, as well as add them to the Seed system.
Get Modules	DApps may want to fetch a module to read their data directly.
Reconnect Client To New Relay Node	DApps may want to disconnect the networked Client from any ongoing Relay Node connections, and establishes a new Relay Node connection.
Re-Request Entanglement & Blockchain State	DApps will want to request from the connected Relay Node the required information to rebuild the entanglement and blockchain locally.

HLAPI Class Design

Seed High Level API (SeedHLAPI)
+ switchAccount(privateKey): Account
+ createAccount(accountEntropy): Account
+ getAccount():Account
+ getTransaction(transactionHash): Transaction
+ createTransaction(moduleName, functionName, args): Transaction
+ propagateTransaction(Transaction): void
+ createAndPropagateTransaction(moduleName, functionName, args): Transaction
+ addTransaction(Transaction): array[LeanTransaction]
+ getter(moduleName, getterFunctionName, args): object
+ read(moduleName, dataKey, optionalUser): object
+ subscribeToFunctionCallback(moduleName, functionName): string
+ subscribeToDataChange(moduleName, dataKey, optionalUser): string
+ unsubscribe(moduleName, funcNameOrDataKey, receipt, optionalUser): void
+ addModule(newModule): void
+ createModule(moduleName, initialStateData, initialUserStateData): Module
+ getModule(moduleName): Module
+ reconnectClientToNewRelayNodeIP(relayNodeIPAddress): bool
+ reloadEntanglementAndBlockchainsState(): bool
+ getHistory(moduleName): array[LeanTransaction]
+ publicAddressToPublicKey(publicAddress): string

Testing Details and Results

The testing required for the Seed project is split into three segments. The first segment consists of the automated unit test. These tests are used to prove the functionality of the subsystems and confirm the project is working as expected under the hood. The second segment of tests revolves around scenario testing the Seed module itself, confirming modules run as expected through the Seed system. The third segment of testing consists of manual testing, confirming the product use cases function as expected.

Screengrabs for all tests are found at the back of the report under Appendix C.

Unit Tests and Results

Summary

The following automated unit tests showcase the functionality of all notable subsystems within the Seed project. Such tasks include tasks such as confirming the cryptography subsystem creates the proper result when requesting data to be hashed, or confirming the entanglement can correctly identify when a transaction would cause a cycle when added.

The tested subsystems include, but are not limited to, testing cryptography, user accounts, pseudorandomness, block & transaction creation, entanglement & blockchain functionality, offline storage, among others subsystems.

See Appendix C1 for screengrabs regarding the unit tests.

#	Description	Procedure	Result
Cryptography Subsystem			
1	Correctly hash with SHA256.	Access the cryptography subsystem and hash the string “Test #1”. Compare result with 3rd party proven SHA256 hasher.	Passed

2	Correctly hashes small data with SHA256.	Repeat test #1, however pass in the string “1”	Passed
3	Correctly hashes large data with SHA256.	Repeat test #1, however pass in the Seed literature review as input.	Passed
4	Throws a error message when attempting to hash undefined data.	Repeat test #1, however pass in undefined, and try to catch the correct error message.	Passed
5	Throws a error message when attempting to hash empty data.	Repeat test #4, however pass in an empty string.	Passed
6	Correctly generates a private key	Access the crypto subsystem and generate a private key.	Passed
7	Correctly generates a private key with user defined entropy.	Repeat test #6, however pass in as entropy the string“Test #7”.	Passed
8	Correctly generates a pair of valid private/public keys.	Repeat step #6, however access the generateKeyPair functionality.	Passed
9	Correctly generates a pair of valid private/public keys with user defined entropy.	Repeat step #8 however pass in entropy for private key creation.	Passed
10	Correctly fetches the public key that belongs to a proposed private key.	Access the cryptography subsystem and request a public key, passing in a private key derived from the entropy “ABC”. Compared the result with the expected result.	Passed
11	Throws a error message when attempting to fetch the public key for a undefined private key.	Repeat test #10, however pass in undefined as the private key. Catch the error and compare it for the expected error message.	Passed
12	Correctly takes a public key and correctly converts it to a public address.	Access the cryptography subsystem and request a conversation from public key to	Passed

		public address. Compare the result with the expected result.	
13	When converting to a public address, throws a error message when a empty parameter is passed in instead of a valid public key.	Repeat test #12, however pass in undefined as the public key. Catch the error and compare it for the expected error message.	Passed
14	Correctly signs data on behalf of a private key.	Access the cryptography subsystem and request a signature on behalf of the proposed private key. Compare the result with the expected result.	Passed
15	When signing data, throws a error message when a undefined parameter is passed in instead of a valid private key.	Repeat test #14, however pass in undefined as the private key. Catch the error and compare it for the expected error message.	Passed
16	Correctly verifies the validity of a signature.	Repeat test #14, however afterwards, request from the cryptography subsystem for the verification of the signature.	Passed
17	Catches invalid signatures when failing to validate them.	Access the cryptography subsystem and request the verification of a signature, however pass in an invalid signature. Check for a return value of false.	Passed
18	When validating signatures, fails to validate signatures who belong to a different public key.	Run test #16, however pass in another private key's signature. Expect the validator to fail.	Passed
19	Correctly generates the proper checksum when given a valid hash.	Access the cryptography subsystem and request the hash to checksum feature. Compare the result with the expected value.	Passed
20	Throws a error message when a undefined parameter is passed in	Access the cryptographic subsystem and request the hash of undefined. Catch the	Passed

	instead of a valid hash.	thrown error and compare it to the expected error message.	
Account Subsystem			
21	Creating an account out of a private key generates proper data (e.g. public key and public address).	Access the account subsystem and request the creation of an account, passing in a private key created from the entropy “ABC”. Compare the results with the expected data.	Passed
22	Throws a error message when a undefined parameter is passed in instead of a valid private key.	Access the account subsystem and request the creation of an account, passing in undefined as the private key. Catch the thrown error and compare it to the expected error message.	Passed
23	Creating an account out of a public key generates a proper data (e.g. public address).	Access the account subsystem and request the creation of an account, passing in a public key created from the private key who’s entropy was “ABC”. Compare the results with the expected data.	Passed
24	Throws a error message when a undefined parameter is passed in instead of a valid public key.	Access the account subsystem and request the creation of an account, passing in undefined as the public key. Catch the thrown error and compare it to the expected error message.	Passed
25	Creating an account out of raw entropy generates a proper data (e.g. private key, public key and public address).	Access the account subsystem and request the creation of an account, passing in the entropy “ABC”. Compare the results with the expected data.	Passed
26	Correctly identifies when a account has the capability to create signatures.	Create an account through a given private key, and request from the account subsystem whether the account can sign	Passed

		messages. Expect the result to be true.	
27	Correctly identifies when a account does not have the capability to create signatures	Create an account through a given public key, and request from the account subsystem whether the account can sign messages. Expect the result to be false.	Passed
28	Accounts with signing capability sign signatures correctly.	Repeat test #26, however after confirming the account can sign, sign a message. Compare the results to an expected result.	Passed
29	Accounts without signing capability cannot sign signatures.	Repeat test #27, however after confirming the account cannot sign messages, try and sign a message. Catch the thrown error message and compare it to the expected error.	Passed
30	Differing accounts signing the same message will produce differing signatures.	Create two accounts from differing private keys, and have them sign the same message. Expect the signatures to not be the same.	Passed
31	Accounts signing separate messages will produce differing signatures.	Create an account through a given private key, and sign two separate messages. Expect the two signatures to not be the same.	Passed
32	Accounts with signing capabilities can verify their signatures.	Repeat test #28, however afterwards, request from the account itself it validate the signature. Expect it to return true.	Passed
33	Accounts without signing capabilities can verify their signatures.	Repeat test #29, however after signing the message, create another account from the first accounts public key. With the new account, request from it that it validate the original signature. Expect it to return true.	Passed
34	Accounts cannot verify signatures which	Create an account from the account	Passed

	are invalid.	subsystem, and verify an invalid signature. Catch for an error message and compare it to the expected error message.	
35	Accounts cannot verify signatures they did not sign.	Repeat test #33, however create the second account out of a different public address. With the new account, request from it that it validate the original signature. Expect it to return false.	Passed
Random Subsystem			
36	Generates the proper Seed out of passed in hashes.	Access the random subsystem and request a randomness seed value, passing in hashes. Compare the result with the expected result.	Passed
37	Throws an error message upon passing in undefined input into seeding generation.	Repeat test #36, passing in undefined. Catch for errors and compare to the expected error message.	Passed
38	Throws an error message upon passing in a empty array as input into seeding generation.	Repeat test #37, however passing in a empty array.	Passed
39	Generates expected pseudo random values based on passed in seed.	Access the random subsystem and set the seed value to a set value. Request random ints and floats, comparing for expected values.	Passed
40	Randomness falls under a valid distribution.	Access the random subsystem, set the seed value and request a random int from 1 to 10 one hundred thousand times. Compare the distribution, checking that there is no statistical significance between favouring numbers over each other.	Passed

Block Subsystem			
41	Block creation creates blocks with valid and accurate data, as well have as a correctly generated hash.	Access the block subsystem and request to make a block, passing in the block data. Compare the block's data and hash with expected values.	Passed
42	Validates that the block validation system is correct in positive cases.	Access the block subsystem and request the creation of a valid block. Next, request block validation for the block. Expect the result to be positive.	Passed
43	Validates that the bl-ock validation system is correct in failing blocks which don't meet block validation rule #1.	Access the block subsystem and request the creation of a valid block in accordance to rule #1.	Passed
44	Validates that the block validation system is correct in failing blocks which don't meet block validation rule #2.	Run test #3, however the check is in accordance of rule #2.	Passed
45	An exception is thrown when an invalid block is checked for validation.	Access the block subsystem and request the validation of a block, however pass in undefined. Check for a thrown error and compare to expected error.	Passed
Transaction Subsystem			
46	Transaction creation creates transactions with valid and accurate data, as well have as a correctly generated hash.	Access the transaction subsystem and request to make a transaction, passing in the transaction data. Compare the transaction's data and hash with expected values.	Passed
47	Validates that the transaction validation system is correct in positive cases.	Access the transaction subsystem and request the creation of a valid transaction. Next, request transaction validation for the transaction. Expect the result to be positive.	Passed

48	Validates that the transaction validation system is correct in failing transactions which don't meet transaction validation rule #1.	Access the transaction subsystem and request the creation of a valid transaction in accordance to rule #1.	Passed
49	Validates that the transaction validation system is correct in failing transactions which don't meet transaction validation rule #2.	Run test #48 however the check is in accordance of rule #2.	Passed
50	Validates that the transaction validation system is correct in failing transactions which don't meet transaction validation rule #3.	Run test #48 however the check is in accordance of rule #3.	Passed
51	Validates that the transaction validation system is correct in failing transactions which don't meet transaction validation rule #4.	Run test #48 however the check is in accordance of rule #4.	Passed
52	Validates that the transaction validation system is correct in failing transactions which don't meet transaction validation rule #5.	Run test #48 however the check is in accordance of rule #5.	Passed
53	Validates that the transaction validation system is correct in failing transactions which don't meet transaction validation rule #6.	Run test #48 however the check is in accordance of rule #6.	Passed
54	Validates that the transaction validation system is correct in failing transactions which don't meet transaction validation rule #7.	Run test #48 however the check is in accordance of rule #7.	Passed
55	Validates that the transaction validation system is correct in failing transactions	Run test #48 however the check is in accordance of rule #8.	Passed

	which don't meet transaction validation rule #8.		
56	Validates that the transaction validation system is correct in failing transactions which don't meet transaction validation rule #9.	Run test #48 however the check is in accordance of rule #9.	Passed
57	Validates that the transaction validation system is correct in failing transactions which don't meet transaction validation rule #10.	Run test #48 however the check is in accordance of rule #10.	Passed
58	Validates that the transaction validation system is correct in failing transactions which don't meet transaction validation rule #11.	Run test #48 however the check is in accordance of rule #11.	Passed
59	An exception is thrown when an invalid transaction is checked for validation.	Access the transaction subsystem and request the validation of a transaction, however pass in undefined. Check for a thrown error and compare to expected error.	Passed
Squashing Subsystem			
60	Confirms squasher would trigger on proper hashes for valid cases.	Access the squashing subsystem and invoke the trigger check, passing into it a positive case hash. Expect the result to return true.	Passed
61	Confirms squasher would not trigger on a invalid hash.	Access the squashing subsystem and invoke the trigger check, passing into it a negative case hash. Expect the result to return false.	Passed
62	Confirms squashing two objects works properly while following the "relative	Access the squashing subsystem and invoke the squashing mechanism, passing	Passed

	data” squashing rules.	in two objects where all the variables are numbers. The squashed result should be one object with the relative values added, as if two vectors were added.	
63	Confirms squashing two objects works properly while following the “absolute data” squashing rules.	Run test #62, however have the variables in the objects be strings, and overwrite the strings assuming the later parameters were the recent changes.	Passed
64	Confirms order matters with “absolute data” rules, with rearranging order changing the squashed result.	Run test #63, however run a second instance where the parameters were in a differing order, and then confirm that the squashed objects do not match.	Passed
65	Confirm squashing transactions into a block produced a valid block.	Access the squashing subsystem and squash multiple transactions into a block. Confirm the block subsystem validates the newly created block.	Passed
66	Confirm squashing blocks into a block produced a valid block.	Access the squashing subsystem and squash multiple blocks into a block. Confirm the block subsystem validates the newly created block.	Passed
Entanglement			
67	Confirms transactions can be added to the entanglement.	Access the entanglement subsystem and add a transaction to the entanglement. Confirm the transaction was added.	Passed
68	Confirms adding transactions fails if the transaction is invalid.	Run test #67, however passing in a invalid transaction. Catch the thrown error, confirming that the caught error matches the expected error.	Passed
69	Confirms adding transactions fails if the transaction would cause a cycle in the	Run test #68, however passing in a transaction which is valid, however causes	Passed

	directed acyclic graph.	a cycle if added to the DAG.	
70	Confirms adding transactions validates older ones.	Access the entanglement subsystem and add multiple transactions to the entanglement, until the first is validated.	Passed
Blockchain Subsystem			
71	Confirms blocks can be added to the blockchains.	Access the blockchains subsystem and add a block to the blockchain. Confirm the block was added.	Passed
72	Confirms adding blocks fails if the block is invalid.	Run test #71, however passing in a invalid block. Catch the thrown error, confirming that the caught error matches the expected error.	Passed
73	Confirm blocks can invoke the block squashing mechanism if they have the right hash.	Access the blockchains subsystem and add a block who's hash would trigger squashing. Afterwards, confirm the added block does not exist in the blockchain, and that its transactions belong to a new block.	Passed
Ledger			
74	Confirm that the ledger can be read from.	Access the ledger subsystem and request a read, passing invalid parameters. Compare the results with the expected results.	Passed
75	Confirm the ledger can have changes applied to it which change the state of the ledger.	Access the ledger subsystem and request to apply a "ChangeContext" object to it, modifying the ledgers state. Run test #71, reading for our newly saved information, confirming it was applied.	Passed
76	Confirm the ledger can create a deep copy of module data.	Access the ledger subsystem and request a deep copy of module data. Confirm this	Passed

		data is proper. Afterwards, modify the data, and then read from the eldger. Confirm the ledger did not have its data change when the deep copy was changed.	
77	Confirm numerous transactions can have their changes applied and get the correct result.	Access the ledger subsystem and apply multiple transactions to the ledger. Confirm the state of the ledger matches expected results.	Passed
78	Confirm a block can have its changes applied to the ledger and get the correct result.	Access the ledger subsystem and apply the block to the ledger. Confirm the state of the ledger matches expected results.	Passed
Virtual Machine Subsystem			
79	Confirm the virtual machine can have modules added to it and be stored in the ledger.	Access the virtual machine subsystem and add a valid module to it. Access the ledger subsystem and validate that it has the modules data properly loaded.	Passed
80	Confirm the virtual machine can read a modules data from the ledger.	Access the virtual machine subsystem and read module data. Confirm that the returned results matches expected results.	Passed
81	Confirm “getter” functions can be invoked to fetch Module data.	Access the virtual machine subsystem and invoke a “getter” call to read a Module’s data. Compare the results with the expected results.	Passed
82	Confirm “setters” can be simulated	Access the virtual machine subsystem and invoke a “setter” call to modify a Modules state. Compare the results with the expected results.	Passed
83	Confirm “setters” can be invoked and the ledger updates accordingly.	Access the virtual machine subsystem and invoke a “setter” call. Compare the results with the expected results, and confirm that	Passed

		the ledger change appropriately as well.	
84	Confirm transactions can be added to the virtual machine, executing them and storing their changes to the ledger.	Access the virtual machine subsystem and add a transaction too it. Confirm that the transaction was executed and it was added to the entanglement.	Passed
FileStorage			
85	Confirm that transactions can be written to storage asynchronously.	Create a FileStorageInjector object and invoke the its save transaction asynchronously passing in a valid transaction. Confirm that the transaction was saved.	Passed
86	Confirm that transactions can be written to storage synchronously.	Create a FileStorageInjector object and invoke the its save transaction synchronously passing in a valid transaction. Confirm that the transaction was saved.	Passed
87	Confirm that blocks can be written to storage asynchronously.	Create a FileStorageInjector object and invoke the its save block asynchronously passing in a valid transaction. Confirm that the block was saved.	Passed
88	Confirm that blocks can be written to storage synchronously.	Create a FileStorageInjector object and invoke the its save block synchronously passing in a valid transaction. Confirm that the block was saved.	Passed
89	Confirm that FileStorage can read transactions synchronously.	Create a FileStorageInjector object and through it invoke reading transactions synchronously. Load a transaction and compare its loaded data against expected results	Passed
90	Confirm that FileStorage can read	Create a FileStorageInjector object and	Passed

	transactions asynchronously.	through it invoke reading transactions asynchronously. Load a transaction and compare its loaded data against expected results	
91	Confirm that FileStorage can read blocks synchronously.	Create a FileStorageInjector object and through it invoke reading blocks synchronously. Load a block and compare its loaded data against expected results	Passed
92	Confirm that FileStorage can read blocks asynchronously.	Create a FileStorageInjector object and through it invoke reading transactions asynchronously. Load a transaction and compare its loaded data against expected results	Passed
93	Confirm that transactions can be removed from storage.	Create a FileStorageInjector object and remove a transaction from storage. Confirm the transaction no longer exists.	Passed
94	Confirm that blocks can be removed from storage.	Create a FileStorageInjector object and remove a block from storage. Confirm the block no longer exists.	Passed
95	Confirm that FileStorage can read all transactions in the entanglement synchronously.	Create a FileStorageInjector object and read all transactions in storage, building an entanglement. Compare this with the expected state of the entanglement.	Passed
96	Confirm that FileStorage can read all blocks for a generation from the blockchain synchronously.	Create a FileStorageInjector object and read all blocks in storage for a given generation, building a blockchain. Compare this with the expected state of the blockchain.	Passed
97	Confirm that FileStorage can read all blocks for all generations of blockchains	Create a FileStorageInjector object and read all blocks in storage, building the	Passed

	synchronously.	blockchains. Compare this with the expected state of the blockchains.	
LocalStorage			
98	Confirm that transactions can be written to storage asynchronously.	Create a LocalStorageInjector object and invoke the its save transaction asynchronously passing in a valid transaction. Confirm that the transaction was saved.	Passed
99	Confirm that transactions can be written to storage synchronously.	Create a LocalStorageInjector object and invoke the its save transaction synchronously passing in a valid transaction. Confirm that the transaction was saved.	Passed
100	Confirm that blocks can be written to storage asynchronously.	Create a LocalStorageInjector object and invoke the its save block asynchronously passing in a valid transaction. Confirm that the block was saved.	Passed
101	Confirm that blocks can be written to storage synchronously.	Create a LocalStorageInjector object and invoke the its save block synchronously passing in a valid transaction. Confirm that the block was saved.	Passed
102	Confirm that FileStorage can read transactions synchronously.	Create a LocalStorageInjector object and through it invoke reading transactions synchronously. Load a transaction and compare its loaded data against expected results	Passed
103	Confirm that FileStorage can read transactions asynchronously.	Create a LocalStorageInjector object and through it invoke reading transactions asynchronously. Load a transaction and compare its loaded data against expected	Passed

		results	
104	Confirm that FileStorage can read blocks synchronously.	Create a LocalStorageInjector object and through it invoke reading blocks synchronously. Load a block and compare its loaded data against expected results	Passed
105	Confirm that FileStorage can read blocks asynchronously.	Create a LocalStorageInjector object and through it invoke reading transactions asynchronously. Load a transaction and compare its loaded data against expected results	Passed
106	Confirm that transactions can be removed from storage.	Create a LocalStorageInjector object and remove a transaction from storage. Confirm the transaction no longer exists.	Passed
107	Confirm that blocks can be removed from storage.	Create a LocalStorageInjector object and remove a block from storage. Confirm the block no longer exists.	Passed
108	Confirm that FileStorage can read all transactions in the entanglement synchronously.	Create a LocalStorageInjector object and read all transactions in storage, building an entanglement. Compare this with the expected state of the entanglement.	Passed
109	Confirm that FileStorage can read all blocks for a generation from the blockchain synchronously.	Create a LocalStorageInjector object and read all blocks in storage for a given generation, building a blockchain. Compare this with the expected state of the blockchain.	Passed
110	Confirm that FileStorage can read all blocks for all generations of blockchains synchronously.	Create a LocalStorageInjector object and read all blocks in storage, building the blockchains. Compare this with the expected state of the blockchains.	Passed
Storage Subsystem			

111	Confirm that Storage can save a transaction to the file system using FileStorageInjector.	Access the storage subsystem and create a Storage object with a FIleStorageInjector, then save a transaction to file. Confirm the transaction was saved.	Passed
112	Confirm that Storage can save a block to the file system using FileStorageInjector.	Access the storage subsystem and create a Storage object with a FIleStorageInjector, then save a block to file. Confirm the block was saved.	Passed
113	Confirm that Storage, using FileStorageInjector, can load all the initial ledger state, reading all blockchains/entanglement and applying all blocks/transactions to the virtual machine.	Access the storage subsystem and create a Storage object with a FIleStorageInjector, then invoke the load from initial state function on the storage subsystem, then confirm the ledger, blockchain and entanglement states match the expected states.	Passed
114	Confirm that Storage can save a transaction to local storage using LocalStorageInjector.	Access the storage subsystem and create a Storage object with a LocalStorageInjector, then save a transaction to local storage. Confirm the transaction was saved.	Passed
115	Confirm that Storage can save a block to local storage using LocalStorageInjector.	Access the storage subsystem and create a Storage object with a LocalStorageInjector, then save a block to local storage. Confirm the block was saved.	Passed
116	Confirm that Storage, using LocalStorageInjector, can load all the initial ledger state, reading all blockchains/entanglement and applying all blocks/transactions to the virtual machine.	Access the storage subsystem and create a Storage object with a LocalStorageInjector, then invoke the load from initial state function on the storage subsystem, then confirm the ledger, blockchain and entanglement states match	Passed

		the expected states.	
Messaging Subsystem			
117	Confirm the ability to subscribe for messages relating to module function callbacks being executed in the ledger machine.	Access the messaging subsystem and subscribe for a function callback. Afterwards, invoke a transaction which matches the subscribed message. Confirm that the callback was invoked.	Passed
118	Confirm the ability to subscribe for messages relating to module data changes callbacks being executed in the ledger machine.	Access the messaging subsystem and subscribe for a module data change callback. Afterwards, invoke a transaction which changes the piece of data the above is listening on. Confirm that the callback was invoked.	Passed
119	Confirm the ability to unsubscribe from messaging.	Run tests #117 and #118, and then unsubscribe the two callbacks. Confirm that the callbacks were unsubscribed.	Passed
120	Confirm, once unsubscribed, previously invokable callbacks stop being invoked.	Run test #119, however after unsubscribing, invoke a transaction which would trigger both callbacks. Confirm neither are invoked.	Passed

Scenario Tests and Results

Summary

Scenario testing is a form of testing Seed modules. It allows developers to test their modules by dictating scenarios between users sending transactions, and allows developers to compare the actual results with the expected results at any point in the scenarios.

See Appendix C2 for screengrabs regarding running the scenario tests outlined below.

Inline “Game” Module Scenario Test

Setup	**Scenario**
Create a simple Module inline. This module creates a “Wall” object at position (-3,0) in memory, starting all new users at position (0,0). The functions available are moving to the left (as long as it would not collide with the wall), getting a users x/y values, and checking if a point is a wall.	A user will call the “moveLeft” function three times, starting at (0,0), moving to (-1,0), then (-2,0), and then once again, attempt to move to (-3,0). However, as there is a wall at position (-3,0), the transaction will fail, and the user will remain in position (-2,0).

Seed Cryptocurrency Module Scenario Tests

Setup	**Scenario**
Load the Seed cryptocurrency module and confirm it can be properly instantiated.	A user will invoke the constructor of the Seed cryptocurrency, The initial Seed data will be loaded, with the user who invoked the constructor being rewarded 1000 SEED. The state of the ledger will then be evaluated and confirmed to be proper.
Seed has a “transfer” function. After Seed is loaded, users will transfer the currency around via that function.	After repeating scenario #1, User1 will transfer 500 SEED to User2, 250 SEED to User3, and 250 SEED to User4. Use1 will attempt to send another 50 SEED, however fail, as they are out of currency. User1 will send 50 SEED to User3, and then User3 will send 150 SEED to User5. User3 will attempt to send 200 SEED, fail as they do not have sufficient funds, then succeed at sending 150 afterwards. The state of the ledger will then be evaluated and confirmed to be proper.
Seed has an allowance system, allowing	After repeating scenario #1, User1 will approve User2 of transferring 200 SEED.

users to “approve” each other for allowance and then” transferFrom” on behalf of one-other. After Seed is loaded, uses will transfer the currency around via those functions.	User2 will send 100 of User1’s SEED to User3. User2 will then send another 100 of the SEED to themself. User2 will attempt to send another 100, however it will fail, as they ran out of their approved allowance. The state of the ledger will then be evaluated and confirmed to be proper.
Seed has a “burn” function, which destroys and amount of currency, permanently removing it from circulation.	After repeating scenario #1, User1 will burn 150 SEED. They will then transfer 100 SEED to User2, who will burn 25 of it. The state of the ledger will then be evaluated and confirmed to be proper.
A scenario which utilizes all of the above functionality will take place, interwinding them all.	After repeating scenario #1, User1 will approve User2 for an allowance of 250. User2 will transfer 100 to themself and 100 to User3. User3 will then send 50 SEED to User1. User2 will then burn 25 SEED. The state of the ledger will then be evaluated and confirmed to be proper.

Manual Tests and Results

Summary

Manual testing is used for testing the use cases that users can expect to encounter when running a Seed Client or Relay Node. These are primarily networked tests showcasing the communication between clients, proving the functionality through real-world scenarios.

See Appendix C3 for screengrabs regarding the following manual tests that are run.

Test #1) Seed Relay Nodes Run

Background	**Scenario**
A Seed Launcher can be run as a relay node by executing the command “Electron . --relay --storage”.	In the Macintosh environment, a relay node will be launched via the npm run relay:osx script.

Launching scripts exist, such as “npm run relay:osx” on Macintosh, “npm run relay:linux” on Linux, and “npm run relay:win” on Windows.	A screengrab is taken showing the launched window beside the terminal which launched it.

Test #2) A Seed Client Can Run & Connect To A Relay Node

Background	**Scenario**
A Seed Launcher can be run as a client, connecting to a designated relay nodes. To launch a client, execute the command “Electron . --client --ip:127.0.0.1”, where the IP passed in is the IP of the relay node to connect to. Launching commands exist, such as “npm run client:osx” for Macintosh, “npm run client:linux” on Linux and “npm run client:win” on Windows.	In the Macintosh environment, a relay node will be launched via the npm run client:osx script. A screengrab is taken showing the launched window beside the terminal which launched it. In the terminal window will be logs showcasing it successfully connected to the relay node.

Test #3) Multiple Clients Can Connect To A Relay Node

Background	**Scenario**
A relay node can have multiple clients connect to it. The theoretical limit of connections is roughly 10,000 connections.	After completing Test #2, two more clients will be launched, for a total of three connecting clients. Two of the uses will launch the Seed application, and two of users will launch the Cube Runner application. A screengrab is taken, showcasing the scenario outlined above.

Test #4) Users On Various Modules Can Validate Each Other

Background	**Scenario**

All transactions in Seed can validate one-other, regardless of which Module and application they belong to.	The first screengrab taken showcases the relay node having invoked the Seed constructor, which was validated by clients #1 and #3 moving their character in the Cube Runner game. The relay node's logged in user has 1000 SEED, while the green and red cube have moved from the top-left starting position, proving the network successfully validated each others actions. The second screengrab shows more transactions being processed after the first. The relay node sent 100 of their Seed to client #3, and the players of Cube Runner have moved their squares further along the grid. The third screengrab showcases further use of the application between the four users in the network. Client #2 burnt 25 SEED, removing it from circulation. The players using Cube Runner have traveled great distance.

Test #5) Clients Can Join The Network Mid Session

Background	**Scenario**
Clients must be able to join the network mid-session and quickly get up to date.	The recorded screengrab demonstrates a fourth client connecting to the network. They have not yet interacted with the system, however their character can be seen at the top-left of the Cube Runner map in the starting position.

Test #6) Clients In Seed Can Use All DApps

Background	**Scenario**

All users of Seed must be able to use all decentralized applications and Modules in Seed.	The first screengrab showcases all five connected clients running the Cube Runner application, moving their squares around the playing field. The second screengrab showcases the same five connected clients running the Seed cryptocurrency application, having transferred funds, using each function available in the Module.

Test #7) Client History Can Be Properly Parsed

Background	**Scenario**
Despite transactions occurring for multiple Modules, and being stored in block Blocks and in the Entanglement, the history of transactions must be parseable.	The recorded screengrab demonstrates an example of the history being parsed in a Seed wallet. The first transaction in history is the constructor, afterwards its transfers between users and a burn function invocation.

Test #8) Networked Users Have Identical Views Of History

Background	**Scenario**
Users connected on the network must have identical views of history, proving they are synchronized.	The recorded screengrab shows the displayed history between three users in a network, demonstrating how their view of history is identical. This history was recorded from the end of test #6.

Implication of Implementation

Seed Protocol

Successes

Seed was, for the most part, successful in the creation of a working validation mechanism which achieves the original goals.

The protocol does offer the potential for a low-latency communication to occur between users, with the bottle neck arising from each users ping between each other and relay nodes, rather than the protocol itself being the bottle neck. The validation time goes down with increased usage, meaning the protocol scales efficiently with usage. The protocol also successfully reduced the overall size of data stored tremendously.

Limitations

The limitations of the Seed protocol result from concerns of security. There are two glaring flaws that cannot be ignored.

Gaming Transaction Hashes

The less harmful concern is the ability for users to reroll their hashes. Users can lie within milliseconds with regards to when their transaction was created. This gives the users plenty of attempts at gaming their hash value, as if they want to reroll for another one, they can simply increase their timestamp by increments of one millisecond until they get a hash they desire.

This has limited uses for abuse, however users could theoretically use this to trigger the transaction squashing mechanism on command. Overuse of this mechanism would not have any glaring negative effects, however it could cause nonoptimal transaction squashing, slightly increasing the overall data size for storing transactions and blocks.

Another potential harm would be if a Module chooses to use transaction hashes as the seed into the pseudorandom number generator. Hashes are the only reliable, seemingly random yet deterministic value modules could access when determining their random seed. If a Module

deploys to the Seed ecosystem relying on their transaction hash, users may be able to predict the random value and reroll hashes until they get the results they desire.

Poison Chains

There is a limitation to the directed acyclic graph (DAG) alternatives to blockchains, which has yet to be solved without implementing proof-of-work or proof-of-stake validation mechanisms. Although precautions are implemented to reduce the threat of poison chains, such as limiting how old they can be before rejoining the main chain, they still pose a very real security threat.

A poison chain is when the DAG begins to split, with malicious users creating transactions which they do not propagate to the main network. They keep these transactions and the knowledge of their existence a secret at first, however continue to use and validate the transactions. They then create a malicious transaction which lies with regards to how it affected the public ledger. This malicious group then continues to validate transactions, pretending the fake transaction is authentic. Once it has enough validation information backing it to inherently be considered honest, all of the transactions within the poison chain is propagated to the main network. The malicious transaction is added to the main network, however enough validation information quickly follows it, that nodes inherently trust it, applying its changes to the ledger.

If this is done quick enough, and no user attempts to create a transaction during the timespan between the network learning of the malicious transaction and the network receiving the validation information, a poison chain could theoretically affect the network.

The Seed protocol does take precautions to prevent poison chains from merging back in with the primary entanglement. When a transaction is created, it not only offers validation information regarding which other transactions it validates, however it also contains information regarding which transactions they are refuting. If transaction A refutes transaction B, transaction A and B cannot coexist in the DAG. This forces clients to reevaluate transaction B, regardless of its validation status, to confirm that it is, in fact, valid. If it fails to validate transaction B, then we know transaction A was honest, and remove B, along with all transactions which lied regarding validating B.

This creates an obstacle for poison chains, making it a unreliable bug to exploit. However, it does require an honest node catches the invalid transaction, and propagates its own transaction to refute it. If no honest node checks the invalid transaction, then the poison chain will successfully merge into the chain.

This threat makes the Seed protocol, at the time of writing, unfit for use as a genuine currency. However, the solution to poison chains is more effective for video games than it is for a currency. In video games, users are constantly propagating transactions, keeping the validation process fluid. This fluidity tremendously raises the likelihood that some user will discover the poison chain and refute it. Traditional currencies require much less transactions, and therefore have a reduced effect on catching the poison chain. Considering video games do not have the same critical security requirements as currencies, this limitation should not be much of a concern for games.

Seed Launcher & Application

On an application level, the Seed project was a complete success. Every part that was set out to be created has been successfully created, working as intended. Modules can be created in Javascript, with a launchable HTML & Javascript DApp that can be dynamically loaded into the launcher. Clients can connect to Relay nodes or act as a Relay Node for other clients if they choose, successfully requesting data and propagating transactions across the network.

The storage implementation works properly, allowing for storage to occur in various environments, including ones with file system access and even those with nothing but local storage access. All unit tests, scenario tests and manual tests pass.

Innovation

Despite the limitations, various forms of innovation sprung

Secure Peer-to-Peer Protocol

A secure peer-to-peer protocol was developed and implemented. The protocol may not be at the security level needed for software with high security requirements, however that use case is a better fit for a different blockchain solution, such as a traditional blockchain.

Despite not being as secure as a traditional blockchain, this solution is more secure than traditional peer-to-peer projects which do not rely on blockchain technology. For example, if a peer-to-peer networked game, such as Halo CE, were developed using the Seed project, it would require multiple users within a game session to cheat together in order to beat the security. Instead, any one user could cheat at Halo CE, as their peer-to-peer protocol did not allow for users to properly validate each other. In Halo CE, among many other peer-to-peer networked games, any single user had the ability to cheat.

The Seed protocol is more secure than most peer-to-peer games, however is less secure than most blockchain solutions.

Distributed Protocol

One of the primary goals of the Seed project was to create a cooperative, distributed protocol, rather than a competitive decentralized once. That is, no single user is to be more important than others in the validation system, and users are incentivized to validate each other, rather than race each other.

Seed was successful in this endeavor. With traditional blockchains, miners act as another class of user with their own separate incentives from the regular user base. Proof-of-stake alternatives to proof-of-work still have this same fault, splitting the user base into two tiers of users. Even other directed acyclic graph approaches, such as the Tangle, had to rely on a centralized node known as The Coordinator in order to upkeep the network during inactive periods.

The Seed protocol is truly a distributed solution, rather than a decentralized or semi-centralized one.

Low-Latency Blockchain

The system is a very low latency compared to other blockchain protocols. The latency is limited by usage, with higher usage resulting in lower latency. Users with low-ping requirements can talk to each other directly, or even send nearly empty messages to validate each other if needed. The fuzzification of trust allows applications to operate before a transaction is fully validated, while allowing applications to then detect and react to flawed actions if the transaction does not become adequately validated.

Validation Time Scaling

Traditional blockchains scale poorly with users with regards to latency. The traditional Bitcoin implementation can only handle eight transactions per second in the protocol before transaction validation starts being delayed. With Seed, with more transactions comes more validations, which can speed up the validation process tremendously. The protocol is not the bottleneck, instead the bottleneck is the network throughput available. Seed is effectively limited by usage and internet speeds.

Storage Scaling

Traditional blockchains scale poorly with size as well. The size of Bitcoin and Ethereum are huge. Blockchain and traditional DAG's data scales at O(n), where n is the number of transactions being stored. While the Seed protocol stores header-data at O(n), the bulk of the transaction data is stored at O(log(n)) due to transaction squashing. This still technically results in a O(n) scaling, however it does keep the storage size down tremendously.

Simple Decentralized Application Development

The Seed project succeeded in making decentralized application development easy. Modules for the Seed ecosystem are developed in raw vanilla Javascript, and can even be read in as a simple JSON object from a file. These Modules are lightweight and very easy to work with.

Complexity

The Seed project had a high overall complexity. A lot of technical knowledge was required, including understanding various types of cryptography to choose the best fit, understanding how Blockchain technology works at an implementation level, understanding the different validation mechanisms blockchains can implement, alternatives to blockchains entirely, creating complex systems, and even requiring the development of a useful API to communicate with the Seed system.

Cryptography

The difference between a blockchain and a linked list comes down to its methodical use of cryptography, and its validation mechanism. As Seed was designed from scratch, there was a large amount of cryptography knowledge that was needed before being able to understand how blockchains worked at an intricate level.

On the simple side was learning more about hashing and understanding how cryptographic signatures work. On the more complex side was analyzing various public key encryption schemes, choosing elliptic curve cryptography and justifying the choice of which specific curve to use. This was not difficult on a programming level, as it simply required using the correct libraries. However, it required a lot of knowledge with regards to planning and making the correct design decision.

Blockchain, Directed Acyclic Graphs & Validation Mechanisms

Blockchain adds another level of complexity on top of cryptography. Instead of simply using cryptography and public key encryption, a complex data structure is instead built using cryptography as one of many tools within it. Blockchains and understanding the problems they attempt to overcome was a complex task in itself.

On top of simply understanding basic blockchain implementations, learning the traditional proof-of-work validation mechanism, where it succeeds, where it struggles, and how alternatives fit into the problem was a monumental task.

Once it was clear how blockchain works with varying validation mechanisms, then learning the directed acyclic graph alternative and its various validation mechanisms continued to increase the complexity.

Complex System

On top of the Seed protocol itself being a very complex task, at an application level, the system itself was also quite complex. It required multiple levels, everything from a launcher which dynamically loads third party custom Modules, to implementing the Seed protocol in code, creating a dynamic storage system, implementing data squashing, creating a peer-to-peer networked system, UI development, and much more. At the time of writing, twenty-six separate Javascript files were needed to implement the API, which does not include the application code or third party modules code. In total, thirty-seven files of code were written, which came out to a lowball estimated line count of 11,500 lines of code, including comments and whitespace.

Reusable API Development

The project was split into two major portions. The Seed protocol implementation, and the Seed launcher. The Seed protocol implementation attempts to be lean, using as little dependencies as possible. The intention was to use NodeJS for cryptography and networking Node Package Manager modules, and keep all custom written code as vanilla Javascript as possible. This was done in order to allow Seed to be implemented in as many Javascript supporting environments as possible.

In the Seed protocol implementation exists the Low Level API. This API exposes the underlying subsystems, allowing developers to communicate directly with Seed in a synchronous manner.

The launcher, which dynamically loads third party decentralized applications, is more lenient with regards to dependencies. The launcher is built using NodeJS and Electron, allowing the development of cross-platform desktop applications. This allowed the Seed launcher and wallet to run on all of Windows, Mac and Linux. For applications which wish to be run inside the launcher, a High Level API was developed. This wrapped the logic regarding communicating with the Low Level API, removing the developers need to understand how Seed works under

the hood. The High Level API is also an asynchronous library which hides the ability to manipulate the entanglement and blockchain from developers, hosting it on a separate process which can only be communicated with through IPC.

Seed was built with a strong foundation and well organized API. This added another level of complexity to the project, having to plan out how decentralized applications would wish to use the Seed API, while assuming developers did not understand how the implementation exactly worked.

Research in New Technologies

Cryptography

The Seed project required a large amount of research into understanding various use cases of cryptography. Through this research, various interesting use cases arose. Cryptography can be used to validate software versions. This can be done by taking the string value of the code to be executed, getting a hash of the string, and taking the first few bytes of that hash. These few bytes represent the checksum, which is small in storage size, and can easily be compared across users. If two users hash their code and compare checksums, versions can effectively be compared.

This is how Seed compares Module code when validating it for execution [11]. A transaction takes the Javascript code for their module, and sends the checksum of the entire module, as well as the invoked function of code executed. This allows receivers to validate that users are running the same, untampered code.

Another interesting use case that is thoroughly used in blockchain technology is the use of cryptographic signature for consent. Cryptographic signatures are the foundation for how blockchains are used. Users receive a private key and public key, sharing the public key. Users can sign messages, such as transactions, with private keys. This digital signature can be validated the public key, allowing anyone who has both the signature and public key to validate the signature. Once a signature has been validated, it can then be inferred that the associated private key must have signed it, and therefore acts as consent on the private key’s owner’s

behalf. The result of the validation is the original message before being signed, allowing the validator to also prove the authenticity of the message, and confirm no data was tampered with.

Blockchain Technology

Blockchain technology is the basis of the Seed project. It is a revolutionary use of cryptography in computer science, which comes with its own benefits and consequences. This fascinating technology allows for open yet secure communication between users, which exists independent of any single authority.

Various changes have been attempted in in hopes of maintaining blockchain's benefits while reducing some or all of its drawbacks. To date, no alternative succeeds in every facet, however numerous advancements have been made.

Some of the problems which blockchain solved, and some of the problems each alternative attempts to solve, are fascinating. For example, a large problem which Bitcoin, the first blockchain cryptocurrency, solved was the threat of a double spend attack. A double spend attack is when a user spends their currency twice, effectively creating currency via exploitation. Bitcoin's Proof-of-Work secures itself from this threat via the act of mining. Mining is essentially adding extra data to a block upon block creation, where that extra data must change the hash of the block to start with a certain amount of leading zeros. In order for a user to have a block which produces a valid hash, they must have brute-force attempted to find the random data which resulted in the correct hash being produced. This brute-force attempt requires large amounts of computing power. In order to modify history, a user would have to recreate every nonse of every block after the historic modification, which would be virtually impossible in Bitcoins case. The act of mining Bitcoin consumes 1% of the world's electricity, making Bitcoin a very secure blockchain.

The alternatives to Bitcoin attempt to fix some issues. Some alternatives attempt to reduce the required electricity expenditure. Directed Acyclic Graph (DAG) alternatives are a common route, some of which avoiding Proof-of-Work, creating alternative validation mechanisms in their place. However, DAG's bring their own set of consequences. A unsolved problem in DAG technology is preventing poison chains in a distributed manner. Poison chains are when a group of users

base their transactions off an existing DAG, however do not reconnect their chain to the main network. In their own transaction mesh, they lie and validate each other as if no user lied. After a long enough time, they reconnect to the original DAG. If the original portion of the DAG trusts the deeply embedded, previously validated transactions inherently, then the poison chain will have successfully lied. However, not trusting deeply validated transactions would mean having every user validate every single transaction, which would prevent a DAG from scaling effectively with a large enough user base.

Researching the various solutions, which each come with their own set of problems, is a fascinating exercise.

NodeJS & Electron

From a software perspective, NodeJS and Electron were fascinating tools to learn for the development of the Seed project. Electron allows for the creation of cross platform desktop applications in HTML, JavaScript and NodeJS. NodeJS brings both the utility of a package installer and package development into the JavaScript ecosystem, as well as brings a level of security to JavaScript which is unprecedented.

As Electron and NodeJS split windows into differing processes, the Seed launcher was constrained to use IPC channels for communication. This led to hands on learning of interprocess communication and confidently giving dynamically loaded third party applications control of their own windows in the launcher, knowing they can only communicate with the Seed ecosystem on the main process through IPC. This was an unexpected security benefit that greatly increased confidence in Electron and NodeJS as solutions.

Future Enhancements

Lattice Based Cryptography

In an attempt to make the Seed project quantum resistant and futureproof its survival in the world of quantum computers, the Seed project will be using a modification on Elliptic Curve Cryptology which uses a lattice in the creation of private and public keys.

Traditional cryptography is based on NP-Hard problems, which are theorized to be inadequately secure when quantum computers become more powerful. The quantum computing algorithm known as Shor's Algorithm[12] successfully solves the factoring problem in polynomial time. It is not proven without a shadow of a doubt that this will mean traditional blockchain cryptocurrencies will be broken in the future, however it does cast enough doubt in traditional cryptography that many cryptocurrencies have been seeking alternatives.

Lattice based cryptography is based upon using mathematical lattices when building problems, with certain lattice problems being hard enough that they can be used in place of NP-Hard problems for building cryptography. It is currency theorized that quantum computers would not be able to solve these problems in polynomial time, making it both resistant to traditional computing and quantum computing [13].

The Seed project will attempt to base its future cryptography off lattice hard problems in an attempt to remain secure as computers evolve with time.

C API Implementation

The Seed project began its implementation of the Seed protocol in JavaScript. This was done in an attempt to showcase that it can be used under web browser constraints. However, in order for low-level desktop applications to utilize Seed, a traditional library, such as a Windows DLL, would be developed. The targeted language would be C, as C libraries are low-level, efficient, can be developed with cross-platform support, and can be used by other languages such as C++ and C#. The Seed protocol is simply a protocol for transacting on the Seed network, however it does not necessarily require users create and propagate these transactions over JavaScript. Theoretically, any language or platform can implement the protocol, as long as they can handle simulating an environment to execute a Modules code.

Modules use very simple JavaScript, which cannot access external code. Therefore, there is no requirement to recreate NodeJS or any JavaScript API in the new environment.

Seed Launcher & Wallet UI Overhaul

The Seed launcher and the Seed wallet both have very simple user interfaces. These interfaces were built in order to be functional, however they are poorly designed in comparison to other applications.

In the future, Seed will be redesigned for user convenience and user experience. Visuals were not a concern for the project as a research project, however the product must be visually appealing before release.

Timeline & Milestones

Timeline	**Milestone**	**Estimated Hours**	**Actual Hours**
Jan 4th to Jan 18th	Cryptographic Component ● Design ● Write Unit Tests ● Write Implementation (Private key generation, hashing between forms of keys, transaction signing)	58	60
Feb 1st to April 10th	Design System ● Research & Plan Blockchain & DAG Hybrid ● Research & Plan Transaction Squashing ● Research & Plan Provable JavaScript Execution ● Research & Plan Lattice Cryptography	80	80
April 20th to June 10th	Seed System ● Design Modules	40	70

	● Design Module Interpreter ● Implement Module Subsystem, Module Interpreter ● Design & Implement Seed Module ● Implement Scenario Tests For Seed Module ● Create bare-bone “Launcher” and “Seed Wallet” shell windows.		
June 14th to July 5th	Entanglement System ● Design “Entanglement” DAG subsystem ● Design Transactions ● Design Transaction Validation ● Implement Entanglement, Transactions & Validation	40	60
July 6th to July 16th	Seed Launcher & HLAPI ● Design Seed Launcher ● Design Dynamic App Loading ● Design High Level API (HLAPI) For Loaded Apps ● Design Seed Wallet ● Implement Seed Launcher With Dynamic Loading, HLAPI And Seed Wallet	30	45
July 20th to August 5th	Transaction Squashing System ● Design Transaction Squashing System ● Design Blocks ● Design Blockchain ● Implement Blocks, Transaction Squashing, Block Squashing and Blockchain	30	40
August 6th to August	Offline Storage ● Design Modular Storage For WebApps and	20	30

14th	Desktop Apps ● Implement LocalStorage Implementation And FileSystem Implementation ● Implement Saving & Loading To Existing Systems		
August 16th to August 28th	Unit Test Project ● Design Unit Tests For Cryptography, Account, Transaction, Blocks, Entanglement, Blockchain, Storage, And Other Subsystems ● Implement Unit Tests ● Fix Any/All Newly Found Bugs In Program	40	40
September 1st to September 12th	Networking System ● Learn NodeJS Networking ● Design Clients ● Design RelayNodes ● Design Network Topology & SSD Scenarios ● Implement Clients & Relay Node ● Update Launching Scripts To Launch Under Client Or Relay Node Modes	30	48
September 13th to September 30th	Draft Report & Test ● Draft Final Report ● Finalize All Manual Tests ● Finalize Any Lacking Unit Tests & Scenario Tests ● Finalize All Documentation	40	80
October 1st to October 15th	Finalize Report & Demo Build ● Finalize Report ● Finalize Any Changes For Demo Build ● Finalize All Testing & Update Report With	40	56

	New Tests		

Conclusion

The Seed project was, overall, a success and a wonderful learning opportunity. The project was fairly complex, and required a lot of research. This greatly increased my ability to analyze and process information, as I read many peer reviewed research papers and white papers. The project has given me ample opportunity to practice designing projects before programming, as well as document the thought process behind my design decisions. Through creating a custom testing environment, I have learned a lot about testing and how beneficial it is to implement tests early on in the development cycle.

Lessons Learned

Through designing a blockchain protocol and implementing it from scratch, I learned a lot about effective design. Each and every subsystem of the Seed project was designed before being implemented, with the thought process behind every decision being fully documented. This taught me a great deal about the importance of designing first, and thinking through problems from start to finish before implementing them. A large reason behind my successfully documenting my thought process and not getting lazy about design was that I wrote blogs and articles about the project as it was being developed.

On the blockchain based website Steemit.com, I wrote blogs regarding my early-thoughts, and articles about my designs & implementations. During my time blogging on Steemit, I acquired 438 followers to date, and submitted my implementation articles to a non-profit group which rewards open source development in the blockchain ecosystem. In order for my articles to be approved, moderators from the community had to evaluate each article and their respective Github pull requests. This gave me the needed accountability to be consistent in my designs, coding conventions, and code comments, while drastically increasing my ability to write effectively. This development process also forced me to prove my project worked as expected during development, implementing testing early on.

With moderators approving each submitted pull request, I had to be certain that the project worked as expected, even before a graphical user interface was implemented. Early on, in the

first month of development, the cryptography and account subsystems were unit tested to ensure that the base project functioned properly. Once Modules were implemented, I built a scenario test, which tested a sample module by creating users and having them interact with the system. This allowed me to prove everything worked as expected, even before the full unit testing suite existed. Once the majority of subsystems were completed, I learned to write a full unit testing suite with zero dependencies in order to intricately test all of the subsystems. This process of planning for tests and testing early on in the development cycle helped maintain the high quality of the project.

Finally, I have grown all around as a software developer. This project is very large, consisting of over 11,500 lines of code and comments. Despite being the largest project I have ever worked on, all of the lessons I have learned kept the project manageable. Minimal issues arose during the development of the project, with no hiccups taking more than a day to resolve.

Closing Remarks

From an application standpoint, the Seed project was a success. The Seed launcher successfully loads third party applications, offering a high-level API to these loaded apps for communicating with the Seed system and it's low-level API. A Seed Wallet app was developed to showcase how an application might use the API. The high-level and low-level API's are run in separate processes, communicating through interprocess communication at the operating system level, keeping the Seed system secure from any potential malicious apps. The low-level API successfully implements the Seed protocol in fairly vanilla JavaScript with the only dependency being NodeJS and a single cryptography library. The launchers are networked, and act as either a client or relay node, which influences how each respective node initially connects to the network. Users can successfully communicate with each other through this network, propagating transactions that successfully run on each users' machine. Every aspect of the project was completed and works as intended.

From a blockchain protocol standpoint, the concept works, however there is a security flaw that I was unable to properly resolve. The Seed protocol's alternative to proof-of-work does not successfully solve the poison chain problem of directed acyclic graphs. In short, a malicious group of users could create a malicious transaction, only share it within the group at the start,

create enough validation work that the transaction would be inherently trusted, and then rejoin the network. This could be done in seconds. When the network receives the transaction, and then immediately receives enough other transactions that it becomes instantly validated, it would then be accepted into history as if it were a valid transaction.

Seed implements a mechanism which reduces the chance of a poison chain successfully remerging with the entanglement, however it does not resolve the issue completely.

There are solutions to the poison chain problem that individual modules could choose to implement. For example, they could choose to allow their user base to inherently trust what their relay nodes claim is correct, and then have their application's primary relay node validate all transactions as they come in, refusing to relay transactions until they become valid. Theoretically, this solution would work, however it would only fix the problem at an application level, not at the protocol level. As far as the protocol is concerned, the poison chain problem is a very real threat which I was unable to solve.

The protocol did succeed at being secure enough for most peer-to-peer video games, as they are generationally terribly insecure. The Seed protocol could successfully be used by a game which does not require a critical level of security. However, the protocol is not secure enough for security critical software, such as currencies.

Returning to the beginning of this project, the problem statement of this research was the following; "How would one replace Competitive Proof-of-Work with Cooperative Proof-of-Play for decentralized blockchain game servers". In this light, I believe this research project was a success. A distributed, cooperative alternative to proof-of-work was created, which can successfully power video game networking requirements using blockchain technology.

Appendix

Appendix A: Approved Proposal

2017

Seed: A Decentralized Server Ecosystem Using Blockchain Technology

Table of Contents

Personal Profile

Computer science has been a hobby of mine, specifically with games development, since I was twelve years old. I began creating video games in Game Maker originally, and eventually upgraded to Lua, Java, and C++. I became a large fan of cryptocurrencies in high school, spending thousands of dollars to create a mining rig to support the network and earn Litecoins and Dogecoins. I began creating video games in payment of cryptocurrencies for users on the social media platform Reddit. When I graduated high school, I set aside these hobbies temporarily while I moved away from Vancouver Island to attend BCIT.

Education

Previous, I completed BCIT's Computer System Technology (CST) diploma, graduating from the Data Communications and Internetworking option with distinction. During this time, I completed many successful projects, including ones which resulted in winning an award for Best Personal Project at the BCIT 2016 Open House.

Currently, I am enrolled in BCIT's Bachelor of Technology in Computer Systems, finishing my degree in the Games Development option. I have recently been awarded the Allen and Linda Stefanson Memorial Award due to my academic performance, and will be graduating with distinction.

Work Experience

After completing first year of CST, I landed myself a job as a C# .NET developer at JRP LTD. During this summer, I created Android and iOS applications in Xamarin, as well as learned constraint programming in C++ to write optimizers for our company's existing software.

I proceeded to be rehired and work two more summers at JRP LTD, taking on larger tasks each year. I continued to work primarily in C# and C++ at JRP, however I was moved away from mobile app development and moved primarily onto Windows application development during the second summer.

In the third summer, I was personally tasked with creating the proof of concept for our software that was then submitted as our application in a bidding war for a multimillion dollar contract. A key feature in this project was creating a decentralized peer-to-peer caching system across multiple computers on the network.

Area of Specialization

My specializations range primarily from data communications to games development. I chose the Data Communications diploma option for CST, as well as the Games Development option for BTECH, both specifically to aide me in following my dreams of creating networked games. Over the last few years, my interests have also began including blockchain technologies more and more.

I previous was a cryptocurrency miner for Litecoin and Dogecoin, created my first Smart Contract for the NEO cryptocurrency ecosystem during this past summer, and began prototyping my very own blockchain in Python. My recent desire has been to merge all my specializations, pairing the security and structure of decentralized blockchain technology with networked video games.

Description

This project aims to explore using blockchain technology to create a provably-fair decentralized video game server. A blockchain protocol will be designed and implemented for this purpose. The protocol will be implemented in web technologies to prove its effectiveness in it's desired use case. The protocol and implementation will replace the traditional competitive proof-of-work mechanism found in cryptocurrencies with a new mechanism that will be referred to as proof-of-gameplay. This project will also be designed around being modular, where it can be reused for multiple games.

Background

The goal of this project is to create a decentralized system that allows for an online multiplayer game to exist without a centralized server, and accomplishing this by utilizing blockchain technology. Decentralized game servers have been a challenge in the past, with one of the biggest flaws being the lack of authority to dictate the truth when a user decides to cheat in peer-to-peer play. If a mechanism was implemented that allowed for complete trust of the information being passed into the game, decentralized servers would have the potential to be much more common place. Blockchains are a rapidly growing technology that allows for decentralized trustless communication to occur between multiple parties without a centralized authority. This project plans on using blockchains to solve the issues with trust that occur in decentralized servers. This technology also allows for the game to be provably fair, with all logic and communication being open for all clients to judge and validate each other. Traditionally, blockchains were used for trustless systems that desired security over all else, such as with the most famous example, Bitcoin [1-2]. Bitcoin was the first electronic cash built on blockchain technology, with many alternatives following all known as cryptocurrencies.

These cryptocurrencies were created with different constraints than that of a video game, as they were intended to be used in place of a currency, rather than a live video game server. One primary difference in constraints is that of speed. In Bitcoin, transactions can be sent out by anyone at any time. A type of user, known as miners, listens for these transactions and adds them to a data structure known as a block. A new block is created and propagated on the network every ten minutes, meaning that when the network is running smoothly, there is roughly a ten-minute turn around time for sending and receiving transactions. At this point, however, the transactions are unconfirmed, and users are supposed to wait another six blocks until we can guarantee with complete certainty that the block is an accurate representation of the transaction history. Bitcoin does not care about speed as much as a game server does. A bank wire transfer can take an entire week, so since Bitcoin is attempting at emulating a currency, speed is not a high priority. Online video games can have varying network requirements, some of which needing to send messages every few seconds, and others needing to send multiple per second. Therefore, this project must modify the underlying validation and block creation mechanism to speed up the system.

The largest bottleneck in Bitcoin is that only one megabyte of transactions is stored in a block, which is created every ten minutes, meaning only one hundred kilobytes worth of data can be transacted a minute on the entire network. Although the size of a block and creation interval are arbitrarily defined, this bottleneck exists due to the proof-of-work mechanism used for validating the computing usage exhausted by miners during block creation [1-2]. Alternatives to proof-of-work exist, such as proof-of-stake, proof-of-activity, ghost and proof-of-burn. Each of these alternatives solved a separate issue with proof-of-work, however each come with their own flaws as well [3-6].

This project will require a new mechanism to be developed for it, which will be referred to as proof-of-gameplay. This mechanism will allow for the replacement of competitive miners in the network, instead replacing them with a validation approach similar to that of the Tangle used in the IOTA cryptocurrency. Each transaction will include validation proof of previous transactions, allowing for transactions to be trusted without waiting for miners to mine blocks.

This project will also require a reconstruction of the blockchain data structure. Blockchains are traditionally a chain of blocks, effectively a linked list of block objects, who each know of the block before them. The size of these blockchains will increase with each new block, raising the issue of memory storage [1-2]. In a cryptocurrency, we need to store all previous information as we care about the history and accuracy of all transactions. However, in a live server, not all transactions need to be stored in long term storage. For example, if we joined a game and a goblin had been killed beside where we were standing a minute prior, we would care about that, as we most likely need to render the corps. We do not, however, need to know every goblin who has ever been killed, and by who. This difference in requirements leads to a difference in design decisions, namely what information is stored on the blockchain. The Seed System redesigns how blockchains function, mixing concepts such as IOTA's tangle, transaction squashing and circular blockchains to maximize storage efficiency while minimizing transaction confirmation wait times.

Innovation

The seed system is a blockchain protocol to allow use of a decentralized blockchain server for multiple games. Blockchain technology is a very new field, with many variations, both tested and non-tested, surfacing every month. Cryptocurrencies such as Ethereum or NEO have risen which allowed for code to be executed on the blockchain [7-8]. Cryptocurrencies such as IOTA have restructured blockchains and the ledger, allowing for an alternative to block mining to be required for transaction confirmations. Unique ideas have surfaced in isolation, however no system has taken from these concepts and combined them in such a way that allowed for a decentralized live game server.

The first source of innovation is found in the seed systems design. The system has two separate chains that branch out of the genesis block, one for temporary storage, while the other is for permanent storage. The temporary storage uses a circular buffer styled blockchain of fixed length, with dynamically sized blocks, to store temporary data. The permanent storage side uses an entanglement system like that of IOTA, to receive transactions out of order, and then stores these entangled transactions into blocks to squash them then add to the permanent blockchain [9]. This design allows for a separation of importance between transaction types, effectively giving the system an ordered quick temporary storage blockchain, or an unordered delayed permanent storage blockchain.

Another source of innovation in the system is the concept of transaction squashing. When the entangled transactions are put into blocks, to reduce space, these transactions become squashed. A squashed transaction is a simplified version of history, where the difference between the start and end states is recorded, not the steps to achieve the end state. For example, if there were three transactions, and in each one User A sent 10 units of currency to User B, the squashed recorded history would be a single update which stated User A sent 30 units of currency to User B. This concept will simplify recorded history, drastically reduce the storage size, as well as keep the integrity of the system in tact.

The modular reuse of the system between multiple games is another source of innovation. Instead of having hardcoded game logic in the blockchain, the system will interpret commands based on the

defined module a transaction claims it belongs too. This adds a level of complexity to the users of the system, as they must define the specifics of their own game logic, however it adds a level of modularity to the system that is unprecedent.

Different modules, as well, will be able to nominate trusted nodes in which they trust. This would mean that the blockchains run fully decentralized, however trust of a specific module can be given to certain nodes, which effectively get a large voting right in the network. This would allow for modules themselves to choose to forgo pure decentralization in the name of network speed, without harming the legitimacy of the rest of the network. An example could be that Game A nominates ten users, which are their own servers, to act as trusted nodes. Users playing their game would follow the regular decentralized rules with each other, however if they receive a transaction being propagated from a trusted node, they would instantly trust that transaction as they trust it's source inherently. This would reduce propagation time on the network for modules that choose to implement it.

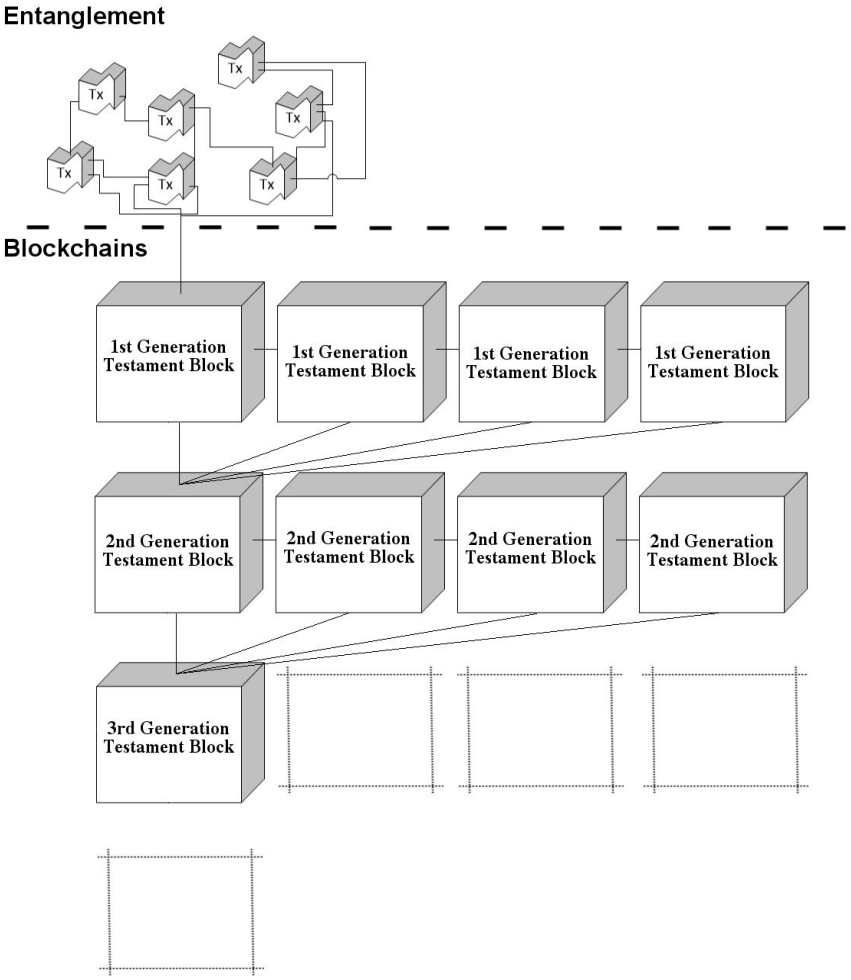
The final innovation that is to arise is the cryptocurrency inside the blockchain known as Seed. Seed is the currency of the network, which will aim to have real world value and use like most cryptocurrencies. Seed is not a in-game currency that modules share, instead it is a different module altogether that acts as the default module all addresses use upon creation. Essentially, Seed uses the system just like any other module, however it is used exactly like a traditional cryptocurrency, with limited extra functionality. Commands do not exist across modules; therefore, seed cannot be used to buy in-game items of any modules, nor can it be used to trade value from one module to another. It is simply its own cryptocurrency, which can be traded between any users on the network. The seed system, modules and coins are outlined in more detail in the following Seed section.

Seed

Seed is both the name of the entire ecosystem of the seed system, as well as the name of the cryptocurrency of the seed system. For the purposes of this proposal, the currency token will be referred to as Seed token, while the ecosystem is referred to as the Seed system.

Seed System Design

The seed system is a combination of subsystems that click together to create a decorated blockchain that can handle both rapid succession temporary transactions, as well as unordered permanent transactions



The basis for the Seed design is to condense excess data once it becomes valid, in order to minimize storage size. When transactions are first added to the system, they join the Entanglement. Once enough transactions reach a point of validation, they may trigger a squashing mechanism which condenses the transaction data into a first generation block. Once enough first generation blocks are in the blockchain, they may trigger the squashing mechanism, squashing them into second generation blocks. This pattern will repeat indefinitely, constantly squashing data, minimizing storage size.

Entanglement

The entanglement is the grouping of tangled permanent transactions. When a transaction is received, it contains validation work on previous transactions in the tangle. These transaction validations may either be for transactions currently in the entanglement, or ones that are being held in a first generation testament block, which holds the recently validated transactions. Once a transaction is added to the entanglement, it is unconfirmed, however it being added helped confirm other previous transactions, pushing the system along forward. After more transactions are received, incoming transactions will validate the original transaction, allowing more faith to be put in the validity of the received transaction. Once the entanglement reaches a certain size, the system squashes the entanglement transactions that are confirmed enough to be trusted into a first generation testament block

Testament Block

A testament block holds the data resulted from squashing multiple transactions. A testament block represents a grouping of changes in history that these transactions caused. Testament blocks can go through multiple generations, where a first generation testament block is created from squashing raw transactions, and a second generation testament block is created from squashing multiple first generation testament blocks together. This squashing mechanism can continue indefinitely, continuously keeping the blockchain lean in storage size.

The squashing mechanism listed above is the act of merging together multiple transactions or blocks into a new testament block. The testament block acts as one large squash of transactions, allowing it to take up very little disk space. For example, if the following scenario of transactions was received from newly validated transactions:

Sender	Receiver	Amount
User C	User B	100
User B	User A	50
User B	User A	50
User B	User A	50
User A	User B	25

The testament block would store a squashed version of the transaction info, which effectively is stored as the following.

User	Amount Changed
User C	-100
User B	+25
User A	+125

This compression of data is essential for the scalability of the network. When a user appears on the network, they can simply request the testament blocks and easily recreate history.

When recreating history, despite lacking a timestamp, we can still order the testament blocks based on their generations. Therefore, if the highest generation on the network is a fifth generation testament block, users would request from the network all testament blocks starting with the fifth generation blocks.

After applying each block to the ledger one-by-one, it will result in the same version of confirmed history

that the rest of the network see's. They can then request for the entanglement to view the unconfirmed or freshly confirmed transactions that are occuring live.

Modules

A module is defined server logic that dictates the rules for a given modules implementation. Transactions for a given module must be validated according to that modules rules. Modules are, essentially, an abstraction layer of server logic, allowing for multiple different modules to run on the same system, validating transactions from other modules. Instructions for transaction validation are given in a modules implementation by in comma delimited commands. Modules should be able to do complicated checks such as confirm a user has the correct units of a given type before they can purchase another, or confirm that a user's position data is within a certain range of another user's position data before attacking.

Modules will allow for key value pair data storage of undefined structure. For example, a module may define that each user has a structure of data which represents an inventory, as well as a structure of data which represents their level progress. Modules can define the structure of their data, as well as set the rules for what can and cannot be modified.

When a transaction is to be created, it must validate previous transactions to contribute work to the network. These transaction verifications are module agnostic. A transaction for Module A can validate transactions from Module B as their contribution to the network, and vice versa. Designing the system this way allows for each module to scale with the next, as well as allow for the upkeep of a modules network while user is actively using such module. For example, a user can transact for Module A on one day, and another user can transact for Module A on another day, and those transactions would still go through despite no other active users using the network during those times, simply because Module B's community validated the transactions for Module A.

Seed Currency Module

The Seed Currency Module is the initial, default module deployed to the system, which is hardcoded into the genesis block. This module defines a single variable, Seed, which is to be used as the digital currency of the network. This module will define rules for transactions with Seed, such as requiring users to own at least the amount they intend to transfer, as well as require all Seed transactions be on the permanent transaction chain rather than the temporary transaction chain.

This modules currency is the currency that powers the network. It is connected to the proof-of-gameplay mechanism in the currency. Proof-of-gameplay takes elements from proof-of-work, proof-of-stake and proof-of-activity to create an alternative mechanism which aligns the desires of all users of the network, rewards users for actively using the network and rewards users for validating fellow users' transactions.

Users pay on the network for every transaction they submit, while also being paid in seed on the network by validating other user's transactions when they submit a transaction. Effectively, if a user validates three other transactions every time they transact, they will pay zero fees to the network as their support of the network is paid off by cooperatively validating a fellow user's transactions. If users choose, they can refuse to validate and pay the network in Seed instead. If users want to earn Seed, they may choose to validate more than three transactions per outgoing transaction, creating a surplus of

validation. This validation adds extra security to the network, and reduces validation propagation time as more sources around the globe are validating any transactions.

Based on a set interval, Seed will be periodically released to users on the network. The breakdown of earned seed will be based on each user's activity and their validated transaction work. The amount of transactions a user submitted since the last payout, added to the amount of transactions they validated since the last payout, is added together to represent their stake in the pool. When that pool of money is released, it is split to all the users based on their stake.

For example, suppose a pool of 1000 Seed were released every hour, and the following table represents the amount of transactions and validations each user did during that hour. The final column represents how much Seed would be paid out to each user.

User	Transactions Sent	Transactions Validated	Stake in Pool	% Of Pool	Seed Payout
A	10	30	40	5.37%	53.69
B	10	60	70	9.40%	93.96
C	5	30	35	4.70%	46.98
D	100	300	400	53.69%	536.91
E	20	0	20	2.68%	26.85
F	25	75	100	13.42%	134.23
G	20	60	80	10.74%	107.38
Total:	**190**	**555**	**745**	**100%**	**1000.00**

This system showcases how the seed payout is a good representation of your network activity and contributions. User D is a hyperactive user who only validates three transactions at a time, however they used the network much more than any other user. They used more and validated more, however they brought the largest amount of validation with them, and were paid out accordingly. User C represents a user who barely used the system, however validated twice as much per transactions than required. We can see that, despite barely using the system, this user received nearly the same amount as User A who used it twice as much, despite validating the same amount of transactions. This importance on transactions sent is there to reward users for using the network, as well as to disincentives pure miners, users who validate much more than they transact in order to gain coins, from taking over the network. User E is an example of a user who did not confirm transactions and instead chose to pay the network instead. In a more detailed example, their Seeds sent would have been added to the 1000 Seed pool, showcasing how existing users are rewarded more when other users chose to pay instead of work.

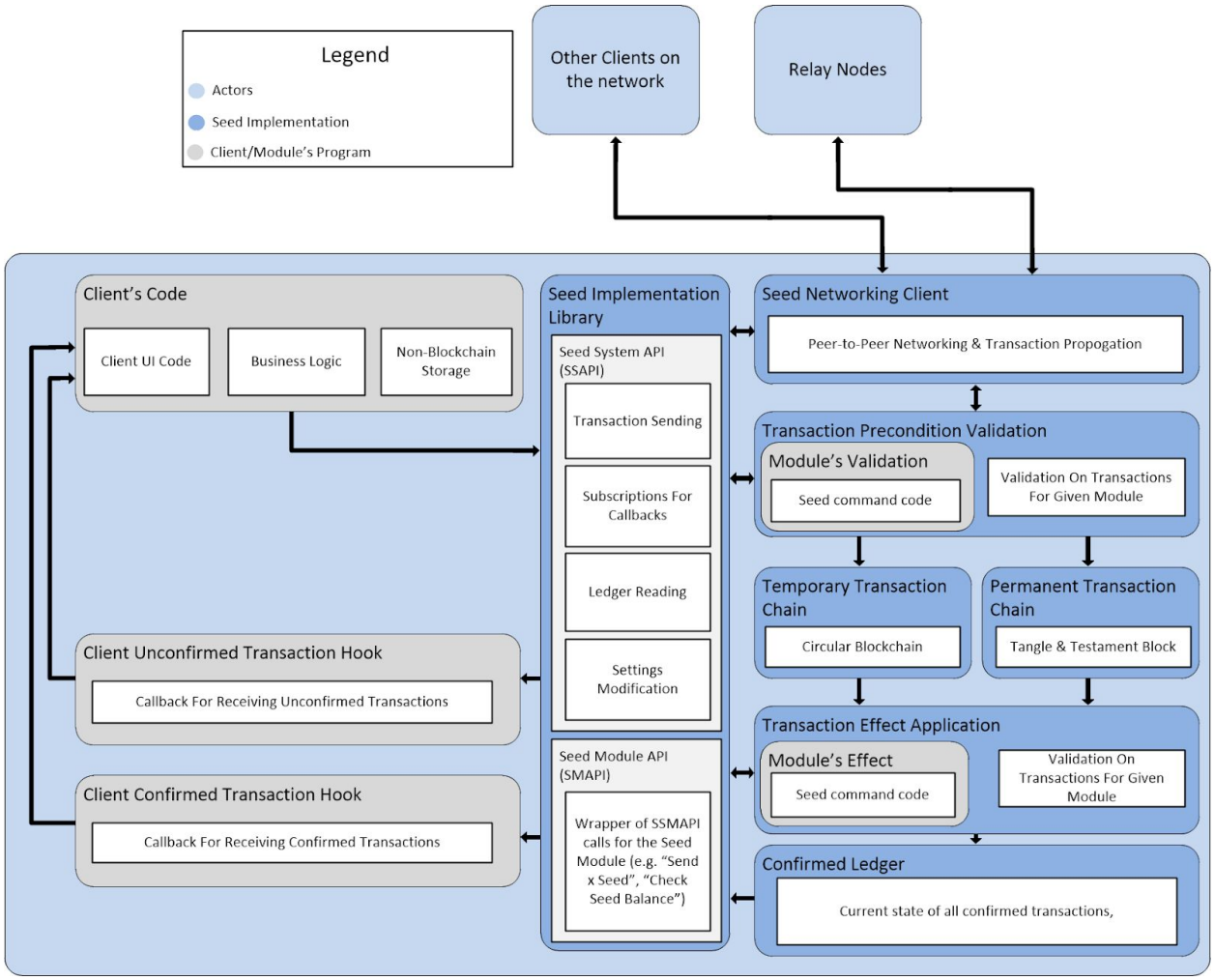
In proof-of-work systems, the coin distribution is isolated to the top miners, while in a proof-of-stake system, the coin distribution is isolated to the richest users. This proof-of-gameplay system keeps the coins constantly distributed to average users, preventing the wealth from being isolated.

System Architecture Overview

Seed is a protocol, with the Seed Implementation Library (SIL) being the implementation of the Seed protocol created for this project. This library is to be used by other programs, handling their networking and storage needs as a wrapper around the seed protocol's implementation. SIL will contain API calls for communicating with the Seed system known as the SSAPI. It will also contain a set of convenience API calls for the Seed module known as the SMAPI.

This project will include the creation of one application, The Seed Wallet, which will be a client implementation built for the Seed module that uses SIL and it's SMAPI. This application will allow for the transfer of the Seed cryptocurrency, conveniently from user to user.

The following architecture diagram showcases the relationships between SIL, a client's program, and other actors in the network.



The Client's code contains the UI that users interact with. The client's codebase can request from SIL to be notified of incoming transaction for a given module type. The client's codebase can also send transactions through SIL. When a transaction is received, or when an unconfirmed transaction becomes confirmed, the client's code will be notified via callback's. SIL offers API calls for reading both the Confirmed Ledger of trusted information, as well as the unconfirmed transaction data.

Extra diagrams showcasing the communication of subsystems in the architecture can be found at the bottom of this proposal in the System Architecture Figures section.

Scope

The majority of the requirements are found in the table below.

Functional Requirements

The Seed System & Seed Implementation Library (SIL)

#	Description
1	Two hardcoded chains will fork out of the Genesis block
2	One fork leads to the Garburator block
3	The Garburator Block can take in transactions, hash them, store the hash then discard it
4	The Circular Blockchain takes in transactions, each of which have validation of another block on the chain
5	The Circular Blockchain removes bad transactions from blocks as they get validated by incoming transactions
6	The Circular Blockchain passes validated transactions to the Garburator when it must overwrite a block
7	The other fork leads to a Testament Block
8	The Testament Block holds a squashed list of transactions merged together
9	The Gate Block takes in transactions from the Lower Entanglement
10	The Gate Block can squash transactions and feed them to the Testament Block whenever it gets a new wave of transactions
11	The Entanglement can take in new transactions, add them to the entanglement, and check their validations to confirm other transactions
12	The Entanglement can pass its deeper entangled lower half into the Gate Block when it reaches a maximum capacity
13	The Seed Systems can parse transactions and determine whether they get added to the Circular Blockchain or the Entanglement based on whether they are a temporary or permanent transaction
14	A ledger can recreate the world by requesting the Testament Block and applying the squashed changes
15	API calls that wrap communication of subsystems for different use cases

Cryptographic Requirements

#	Description
1	Users can generate a private key securely
2	Private Keys can be one-way hashed via SHA256 to create the public key
3	Public Keys can be double SHA256 hashed to create a public hash
4	Public Hash's can be Base58Check encoded to create a Public Address
5	Transaction hashes can be signed by a user's private key
6	Signed transactions can have their signage validated by verifying it was signed by a user via their public key

Module Requirements

#	Description
1	Modules can specify key/value pair structures that exist for storage on a per-user basis
2	Modules can specify global key/value pair structures for the entire module
3	Modules can specify verification logic for transaction verification
4	Client's module implementation can specify the execution of transactions
5	Modules can nominate specific public keys to act as trusted nodes for transactions of that module type
6	Modules can read from other modules' data locations; however, they cannot write to them

Network Propagation / Validation Requirements

#	Description
1	Relay nodes act as decentralized user registry's. They keep track of which IP's are online
2	Users can connect to a relay node to be told of users near them (People who are worthy of propagating to)
3	When a user connects, the relay node they connected to tells all the other relay nodes about this user
4	Users can send Transactions to users near them and to relay nodes
5	Relay nodes relay all transactions to other relay nodes and to everyone they know about on the network

6	When a user disconnects from a relay node, it unsubscribes them and tells all other nodes/users near them to not send to that user anymore

Non-Functional Requirements

Flexibility

The module system should be powerful enough that any client running any module can receive transactions from a different module that this client is not developed for, and still be able to validate the transaction to support the system. That means the power of the commands that can be used to build validation functions for a module must be strong enough that it can all be handled without running the actual client for that module.

The module system should be modular enough that it can power a variety of projects, such as a cryptocurrency or a near live multiplayer online game server.

Scalability

The system should scale with more users and modules. The more users, the more confirmations, the quicker the network speed. The system should be able to handle enough users to power an online game without slowing down noticeably.

Technical Challenges

Scope

Scope is an issue in this project. The project is using recent technology, and aiming be strong enough to support an actual online networked game.

Performance

Gaining performance from the system will be a challenge, as confirmation times, as well as propagation time, both are concerns that have not been adequately addressed. We may find that those two factors are too slow for a system as speed reliant as it.

Scalability

The system intends on scaling with an increase in traffic. This approach to blockchain to achieve this is not well tested, and may prove to not work as planned.

Flexibility

The flexibility of the system is a large requirement. It will require a solid design with a lot of planning to do properly in an efficient manner.

Technical Knowledge

The project requires a high degree of technical knowledge in an innovative technology, advancing in a field with minimal academic research.

Methodology

This research project will be conducted following a modified version of agile scrum.

This project is being developed alongside another research project created by Jaegar Sarauer. His research project is to create a 2D JavaScript MMO video game, in which this project will power the

networking of his project. His project will have its server logic written into another module on this system.

Due to the positions of our projects, his project is effectively this projects' first client. Our agile scrum meetings will coexist as we work side by side on two separate projects. This projects' deliverables will be handed off to him as the system is more and more usable.

In the scenario that this project runs behind schedule or is deemed unfit to sustain the server for Jaeger's system, a Node.js server will be created in it's place. This effort is to mitigate concerns of dependency between the two projects, allowing Jaegar's to thrive under the scenario that this project fails or is delayed.

This project will also be following test driven design. As such, part of the bi-weekly deliverables will include all the tests for each component being delivered.

This project will be following sprint durations of two weeks.

Technologies

The protocol being developed does not require any technologies. It, like any other blockchain protocol, can be implemented in any language and should be able to fully comply with all other clients.

The implementation being developed will initially target web browsers. The purpose of this is to prove that it can be developed for a web browser MMO video game, as that is the desired use case attempted to be solved with this project.

For this implementation, JavaScript will be used along with Node.js for the required socket connection. A cryptography library capable of doing SHA256 hashing and Base58Check encoding will also be required. Jasmine, a JavaScript library for Unit Testing, will be used to accomplish all testing.

Detailed test plan

This project will be developed following test-driven design. Unit testing will be used to test the expected behaviours of methods as components are developed, as well as test the behaviour and expected outputs of larger systems which are built on the tested components. Every behaviour in the blockchain should be deterministic, making it the perfect candidate for such testing.

Unit Testing

The JavaScript library Jasmine will be used for the creation and execution of unit tests, as it is supported in all major browsers. Components will be developed following test-driven design, requiring that nearly every component should have tests written for it. Ideally, following this design should reduce the number of unforeseen bugs, despite it requiring a little extra boilerplate work. Unit tests will be created every sprint. All unit tests will be run as recent changes come in to confirm that old components still function as required.

Test Case Examples

Transaction

#	Test Description
1	Transactions can be validated for any module
2	Transactions can contain a dynamic amount of other transaction validations in them

3	Transactions can be signed with their sender's private key
4	Transactions can be validated with their sender's public key
5	Equivalent transactions can be squashed together without losing information

Circular Blockchain

#	Test Description
1	Transactions can be added to the blockchain
2	Blocks can remove transactions that are deemed invalid by future transactions
3	Block can overwrite old blocks like a circular buffer
4	The blockchain can be validated by running through each transaction in the circular buffer and validating that they are based on valid info, can be run, and any dependent transactions can have their hashes retrieved from the garburator.

Entanglement

#	Test Description
1	Transactions can be added to the entanglement
2	Transactions validating previous transactions only can validate transactions who's validated transactions were deemed valid
3	Once reaching a certain capacity, it can take all transactions that have enough confirmations to be considered valid, and hand them to the gate

Gate Block

#	Test Description
1	Can receive a bunch of unordered transactions from the lower entanglement
2	Can squash transactions without losing information
3	Can feed squashed transactions over to the Testament block

Testament Block

#	Test Description
1	Can receive squashed groupings of transactions
2	Can squash the squashed groupings of transactions to simplify the data further

Behaviour Checks

#	Test Description	Expected Result
1	Transactions that come in with validation work that does not match the majority of other transactions agreed upon validation work	Rejected
2	Transactions come in with validation work that does match the majority of other transactions agreed upon validation work	Accepted
3	Transactions come in, but then the majority of other transactions validating it claim it is not valid	Rejected
4	Transactions come in, but it validated 'work' for transactions that have already been either been fed to the garburator or squashed	Do not confirm the transaction, however if other users confirm it, trust it
5	Transactions come in which is signed by an address in a module's trusted node list	Accepted, instantly trust the confirmation work shown
6	Transactions come in which validate work for transactions that do not belong to the same module	Accepted
7	A user connects to the network for the first time, and requests to be brought up to date.	User is given the garburator block, circular blockchain data, entanglement and testament block logic, can recreate the world accurately

Details about estimated milestones

Timeline	Milestone	Time Estimation	Total
Jan 4th to Jan 18th	**Sprint 1** Cryptographic Component • Design • Write Tests • Write Implementation (Private key generation, hashing between forms of keys, transaction signing) Seed System • Design	**Design**: 8 hours **Tests**: 5 hours **Implementation**: 40 hours	58 Hours

Jan 19th to Feb 1st	**Sprint 2**
	Seed System
	• Write Tests
	• Write Subsystem Implementations
	  ○ Temporary Transaction Chain
	  ○ Permanent Transaction Chain
	
	Module System
	• Design
		Design:
		4 hours
		Tests:
		8 hours
		Implementation:
		40 hours
			62 Hours

Feb 2nd to Feb 15th	**Sprint 3** Module System • Write Tests • Write Implementation • Write Documentation Seed Module • Design	**Design**: 4 hours **Tests**: 4 hours **Implementation**: 40 hours	48 Hours
Feb 16th to Feb 29th	**Sprint 4** Seed Module • Write Tests • Write Implementation Relay Node / Networking Component • Design	**Design**: 6 hours **Tests**: 6 hours **Implementation**: 40 hours	52 Hours
Mar 1st to Mar 14th	**Sprint 5** Relay Node / Networking Component • Write Tests • Relay Node Implementation (Nodes that users can connect to as an entrance point to the network) • Network Implementation (JS Clients connection code for communicating with relay nodes & each other) Integration • Integration Design	**Design**: 10 hours **Tests**: 10 hours **Implementation**: 40 hours	60 Hours

Mar 14th to Mar 28th	**Sprint 6** Integration • Write Tests for integration, confirm all previous tests still pass • Integrate Stress Test • Stress Test Network	**Tests**: 30 hours **Integration**: 20 hours	50 Hours
Mar 29th to Apr 11th	**Contingency Sprint** • Fix, Repair and Retest any parts that failed after integration or are behind schedule • Polish any components which are not finished • If time permits after finalizing the project, begin Release stage	**Contingency**: 50 hours	50 Hours
Apr 12th to Apr 25th	**Release** • Final Testing • Final Documentation • Final Report	**Testing**: 10 hours **Documentation**: 10 hours **Report**: 30 hours	50 Hours
Estimated Total Time			430 Hours

Detail all deliverables

Seed: Protocol / Whitepaper

A flexible decentralized server protocol that can be implemented in any language, on any platform. This documentation will showcase the protocol in detail, explaining exactly what must be done for another client to implement the same protocol.

Seed Implementation Library: A JavaScript Library Implementation

A JavaScript implementation of the Seed protocol. This library can be included by any JavaScript client which supports Node.js, allowing them to connect to the network, send transactions and utilize modules.

Seed: The Cryptocurrency Module & Client

The first module to be released onto the Seed ecosystem is the Seed cryptocurrency. Code for this module will be delivered, as it is the default module which gives this ecosystem it's first use case. This module will come with a simple client that utilizes the Seed Implementation Library in order to handle transaction with other clients over the Seed network.

Final Report

The final report on the progress, development and results of the project.

Development in Expertise

While I have developed basic smart contracts for the NEO cryptocurrencies' ecosystem, and attempted at developing a Python cryptocurrency implementation in the past, this project will further increase my understanding of blockchain technology due to it's sheer depth and complexity. Blockchain technologies

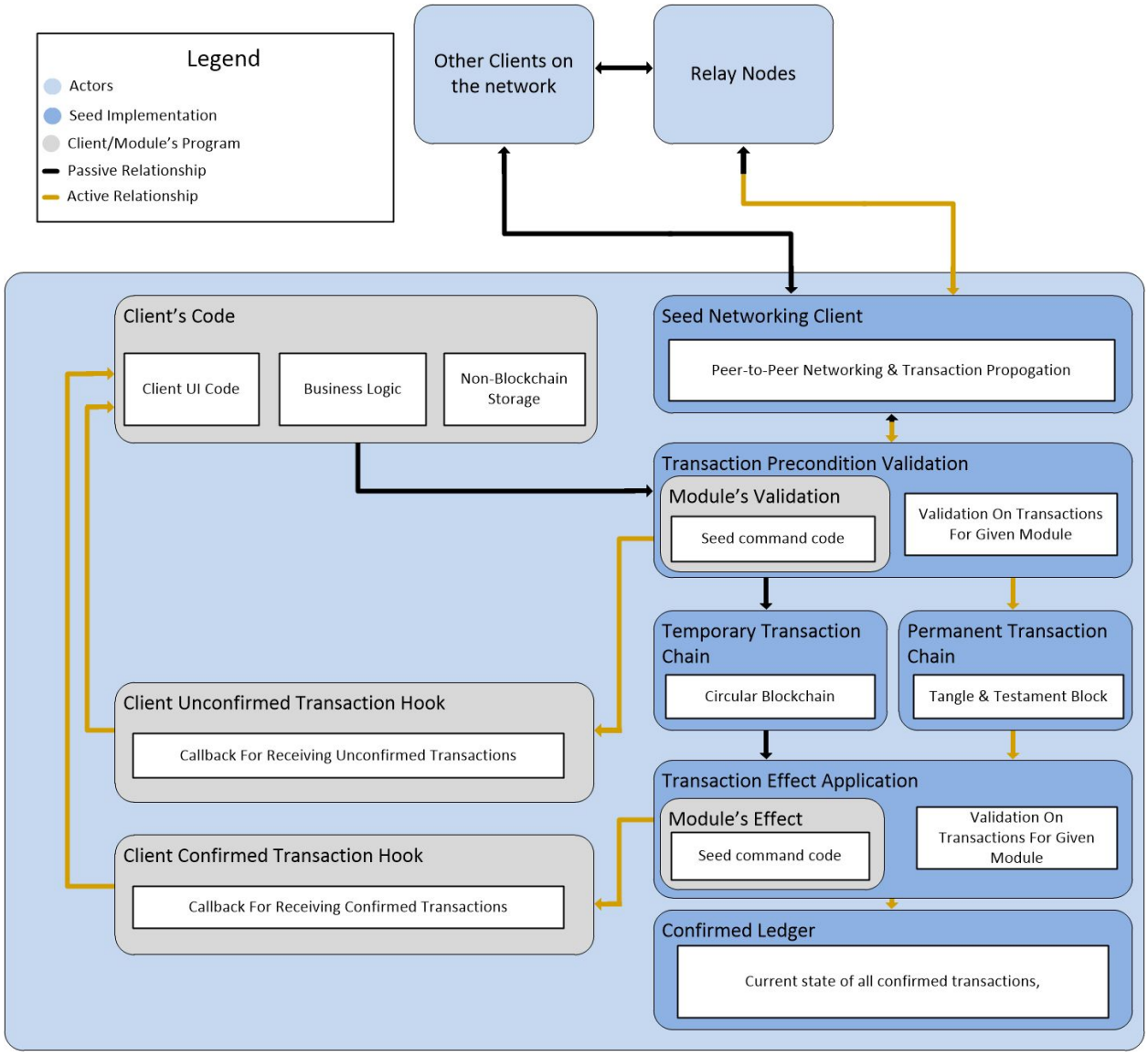
have been a passion of mine for years, but I've never had an opportunity to implement my ideas to the extent of this project. The project scope is large; however, I firmly believe I have the skillset, capabilities and learning capacity required to succeed in creating this system.

System Architecture Figures

The following figures were created to showcase the relationships and communication of different subsystems.

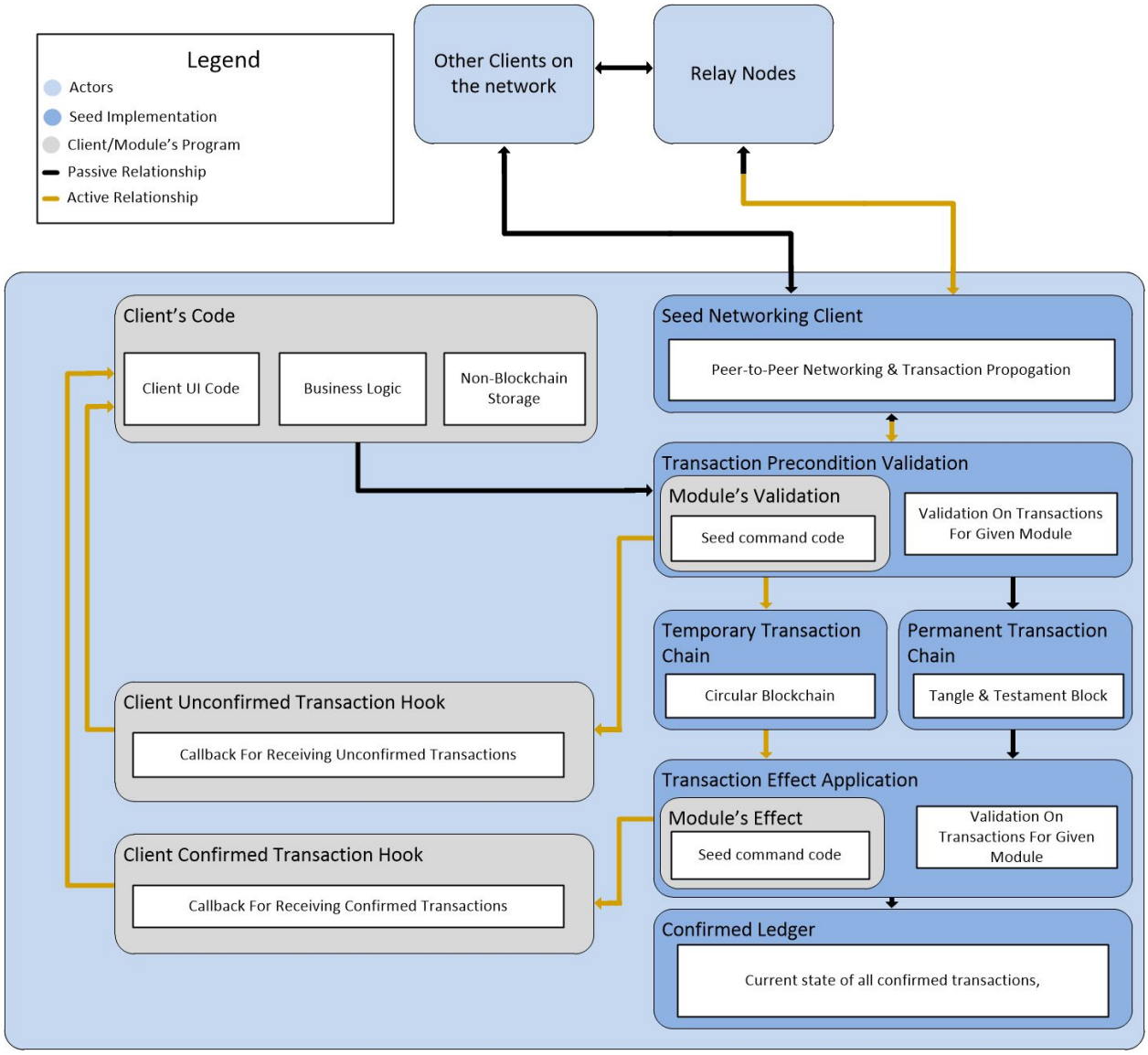
Receive Permanent Transaction

The chain of communication begins from the Relay Nodes (or possibly directly from another client), storing the data through the permanent transaction chain, into the ledger, while notifying the clients callback's regarding the process where appropriate.



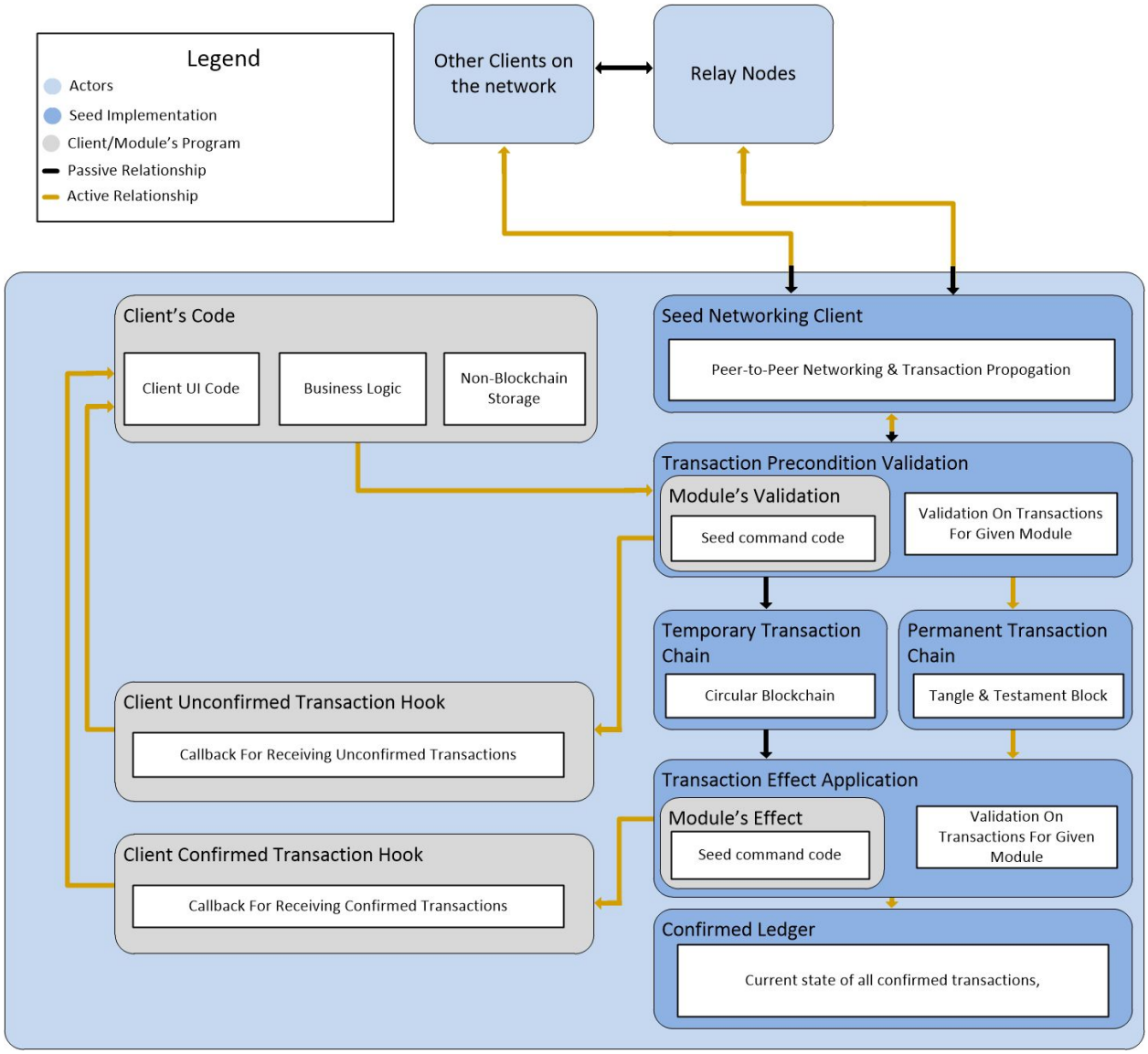
Receive Temporary Transaction

The chain of communication begins from the Relay Nodes (or possibly directly from another client), storing the data temporarily in the temporary transaction chain, while notifying the clients code about the process where appropriate.



Send Permanent Transaction

The communication begins from the Client's codebase as it requests SIL submit a transaction to the network. SIL first validates the transaction, to confirm the request is proper, validates a few unconfirmed transactions to use as work for the network, then sends the transaction to other clients and relay nodes. It then handles the transaction as if it just received it, adding it to the permanent transaction chain, the ledger, and calling all client callback's along the way.



References

[1] Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system.

[2] Decker, C., & Wattenhofer, R. (2013, September). Information propagation in the bitcoin network. In Peer-to-Peer Computing (P2P), 2013 IEEE Thirteenth International Conference on (pp. 1-10). IEEE.

[3] Bentov, I., Gabizon, A., & Mizrahi, A. (2016, February). Cryptocurrencies without proof of work. In International Conference on Financial Cryptography and Data Security (pp. 142-157). Springer Berlin Heidelberg.

[4] King, S., & Nadal, S. (2012). Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. self-published paper, August, 19.

[5] Vasin, P. (2014). Blackcoin's proof-of-stake protocol v2.

[6] Bentov, I., Lee, C., Mizrahi, A., & Rosenfeld, M. (2014). Proof of Activity: Extending Bitcoin's Proof of Work via Proof of Stake [Extended Abstract] y. ACM SIGMETRICS Performance Evaluation Review, 42(3), 34-37.

[7] Buterin, V. (2014). A next-generation smart contract and decentralized application platform. white paper.

[8] Kosba, A., Miller, A., Shi, E., Wen, Z., & Papamanthou, C. (2016, May). Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In Security and Privacy (SP), 2016 IEEE Symposium on (pp. 839-858). IEEE.

[9] Popov, S. (2016). The tangle. *cit. on*, 131.

Appendix B: Project Supervisor Approval

From: A. Abdulla
Sent: Tuesday, October 16, 2018
To: Elsie Au
Subject: Carson Roscoe - Practicum

Hi Elsie,

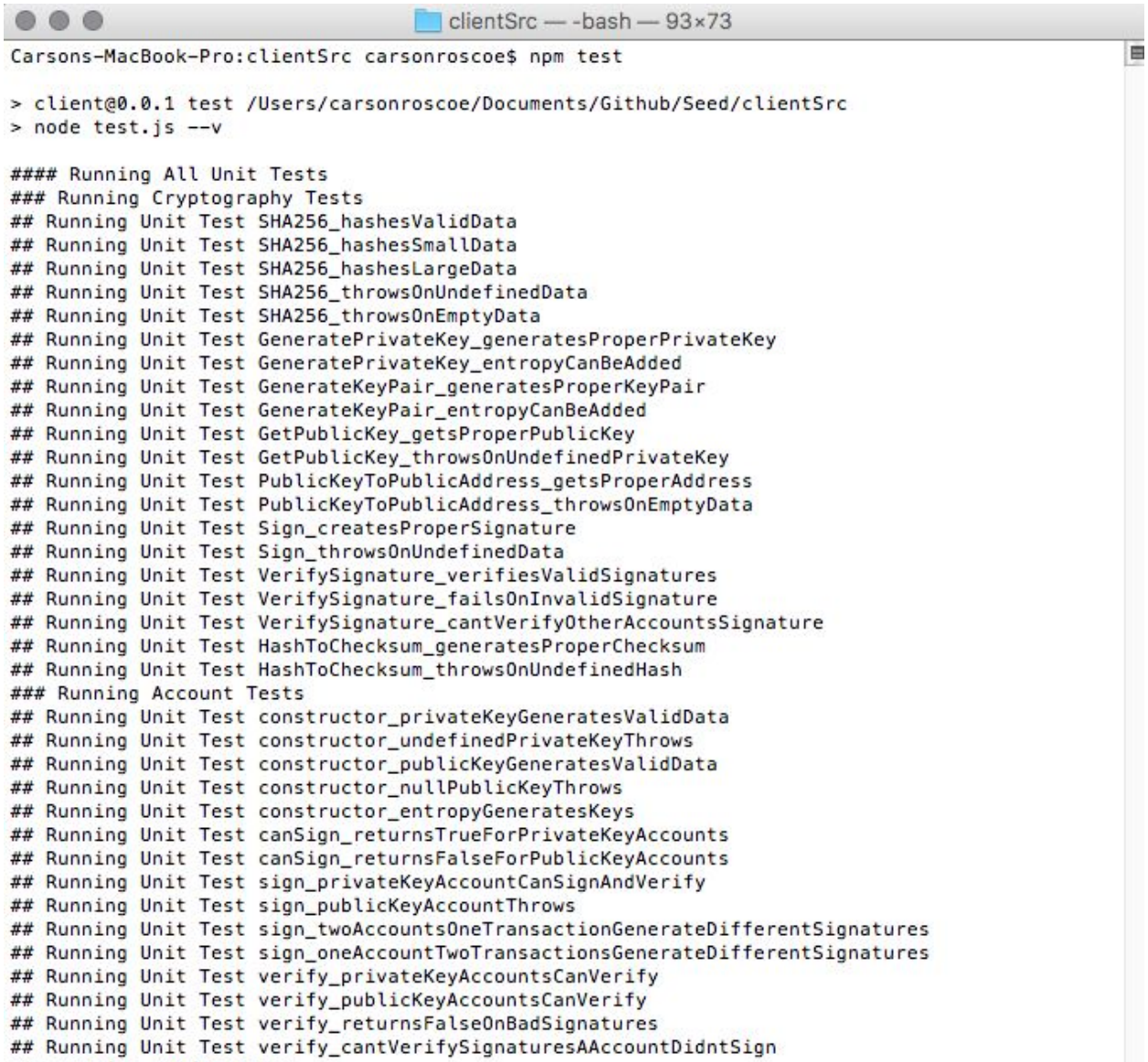
Carson has successfully demonstrated his practicum project to me. I am quite satisfied that that he has met all of the requirements of his practicum proposal.

Aman

Appendix C: Test Screengrabs

Appendix C1: Unit Test Screengrabs

The following three screengrabs are taken from executing one hundred and twenty unit tests with verbose turned on, allowing some tests to display information during execution which may be of interest.



```
### Running Random Tests
## Running Unit Test seedFromHashes_generatesProperSeedFromHashes
## Running Unit Test seedFromHashes_throwsForUndefinedInput
## Running Unit Test seedFromHashes_throwsForUEmptyInput
## Running Unit Test random_generatesRandomValueBasedOnSeed
## Running Unit Test random_randomnessFallsUnderValidDistributions
Over 10 seeds, invoke total of 100,000 random calls, and mapping distribution into 10 buckets
Highest Distributed Value:  10121
Lowest Distributed Value:  9764
Distribution of random values { '<0.1': 10063,
  '<0.2': 10004,
  '<0.3': 9764,
  '<0.4': 10110,
  '<0.5': 10015,
  '<0.6': 10121,
  '<0.7': 9775,
  '<0.8': 9985,
  '<0.9': 10096,
  '<=1.0': 10067 }
### Running Block Tests
## Running Unit Test blockCreation_createsAValidBlockWithValidHash
## Running Unit Test blockValidation_createAndValidateABlock
## Running Unit Test blockValidation_failsBlocksBreakingRule1
## Running Unit Test blockValidation_failsBlocksBreakingRule2
## Running Unit Test blockValidation_throwsMalFormedBlock
### Running Transaction Tests
## Running Unit Test transactionCreation_createsAValidTransactionWithValidHash
## Running Unit Test transactionValidation_createAndValidateATransaction
## Running Unit Test transactionValidation_failsTransactionsBreakingValidationRule1
## Running Unit Test transactionValidation_failsTransactionsBreakingValidationRule2
```

```
## Running Unit Test transactionValidation_failsTransactionsBreakingValidationRule3
## Running Unit Test transactionValidation_failsTransactionsBreakingValidationRule4
## Running Unit Test transactionValidation_failsTransactionsBreakingValidationRule5
## Running Unit Test transactionValidation_failsTransactionsBreakingValidationRule6
## Running Unit Test transactionValidation_failsTransactionsBreakingValidationRule7
## Running Unit Test transactionValidation_failsTransactionsBreakingValidationRule8
## Running Unit Test transactionValidation_failsTransactionsBreakingValidationRule9
## Running Unit Test transactionValidation_failsTransactionsBreakingValidationRule10
## Running Unit Test transactionValidation_failsTransactionsBreakingValidationRule11
## Running Unit Test transactionValidation_throwsMalFormedTransaction
### Running Squasher Tests
## Running Unit Test squashingTrigger_wouldTriggerSquasherForProperHashes
## Running Unit Test squashingTrigger_wouldNotTriggerSquasherForInvalidHashes
## Running Unit Test squash_squashesRelativeData
Squashing::  { a: 5, b: -5, c: 1, e: { a: 5 } } { a: -3, b: -2, d: 3, e: { a: 2 } }
Resulted Data { a: 2, b: -7, c: 1, e: { a: 7 }, d: 3 }
## Running Unit Test squash_squashesAbsoluteData
Squashing::  { a: 'Hello', b: { d: 'a' }, c: 1 } { a: 'World', b: { d: 'b' }, d: 3 }
Resulted Data { a: 'World', b: { d: 'b' }, c: 1, d: 3 }
## Running Unit Test squash_orderMattersForAbsoluteData
Squashing::  { a: 'Hello' } { a: 'World' }
Resulted Data { a: 'World' }
Squashing::  { a: 'World' } { a: 'Hello' }
Resulted Data { a: 'Hello' }
## Running Unit Test squash_transactionsIntoBlock
## Running Unit Test squash_blocksIntoBlock
### Running Entanglement Tests
## Running Unit Test entanglement_addsValidTransactionsToEntanglement
## Running Unit Test entanglement_doesNotAddInvalidTransactions
## Running Unit Test entanglement_addingTransactionsValidatesOthers
## Running Unit Test entanglement_doesNotAddTransactionsIfCausesCycle
```

```
### Running Blockchain Tests
## Running Unit Test blockchain_addsValidBlockToBlockchain
## Running Unit Test blockchain_doesNotAddInvalidBlockToBlockchain
## Running Unit Test blockchain_blocksCanInvokeSquashingMechanism
### Running Ledger Tests
## Running Unit Test ledger_readFromLedger
## Running Unit Test ledger_appliedChangesModifyState
## Running Unit Test ledger_canCreateDeepCopiesOfModuleData
## Running Unit Test ledger_multipleTransactionsChangingInSequenceGivesCorrectResult
## Running Unit Test ledger_appliedChangesFromBlockGiveSameResult
### Running SVM Tests
## Running Unit Test svm_modulesCanBeAdded
## Running Unit Test svm_canReadModuleDataFromLedger
## Running Unit Test svm_canInvokeModuleGetterFunctions
## Running Unit Test svm_canSimulateModuleSetterFunctions
## Running Unit Test svm_canInvokeModuleSetterFunctionsWhichAndStateChanged
## Running Unit Test svm_addingTransactionsExecutesAndStoresInledger
### Running FileStorage Tests
## Running Unit Test fileStorage_storesTransactionsAsynchronously
## Running Unit Test fileStorage_storesTransactionsSynchronously
## Running Unit Test fileStorage_storesBlocksAsynchronously
## Running Unit Test fileStorage_storesBlocksSynchronously
## Running Unit Test fileStorage_readsTransactionsSynchronously
## Running Unit Test fileStorage_readsTransactionsAsynchronously
## Running Unit Test fileStorage_readsBlocksSynchronously
## Running Unit Test fileStorage_readsBlocksAsynchronously
## Running Unit Test fileStorage_removesTransactionFromStorage
## Running Unit Test fileStorage_removesBlocksFromStorage
## Running Unit Test fileStorage_readsFullEntanglementFromStorage
## Running Unit Test fileStorage_readsABlockchainFromStorage
## Running Unit Test fileStorage_readsAllBlockchainsFromStorage
```

```
### Running LocalStorage Tests
## Running Unit Test localStorage_storesTransactionsAsynchronously
## Running Unit Test localStorage_storesTransactionsSynchronously
## Running Unit Test localStorage_storesBlocksAsynchronously
## Running Unit Test localStorage_storesBlocksSynchronously
## Running Unit Test localStorage_readsTransactionsSynchronously
## Running Unit Test localStorage_readsTransactionsAsynchronously
## Running Unit Test localStorage_readsBlocksSynchronously
## Running Unit Test localStorage_readsBlocksAsynchronously
## Running Unit Test localStorage_removesTransactionFromStorage
## Running Unit Test localStorage_removesBlocksFromStorage
## Running Unit Test localStorage_readsFullEntanglementFromStorage
## Running Unit Test localStorage_readsABlockchainFromStorage
## Running Unit Test localStorage_readsAllBlockchainsFromStorage
### Running Storage Tests
## Running Unit Test storage_canSaveTransactionToFileSystem
## Running Unit Test storage_canSaveBlockToFileSystem
## Running Unit Test storage_canLoadInitialStateFromFileSystem
## Running Unit Test storage_canSaveTransactionToLocalStorage
## Running Unit Test storage_canSaveBlockToLocalStorage
## Running Unit Test storage_canLoadInitialStateFromLocalStorage
### Running Messaging Tests
## Running Unit Test messaging_subscribingForModuleFunctions
## Running Unit Test messaging_subscribingForModuleDataChanges
## Running Unit Test messaging_unsubscribingFromMessaging
## Running Unit Test messaging_unsubscribingCleansUpCallbacksWithoutLeaks
#### Tests Complete
## Passed All 120 Unit Tests
```

Appendix C2: Scenario Test Screengrabs

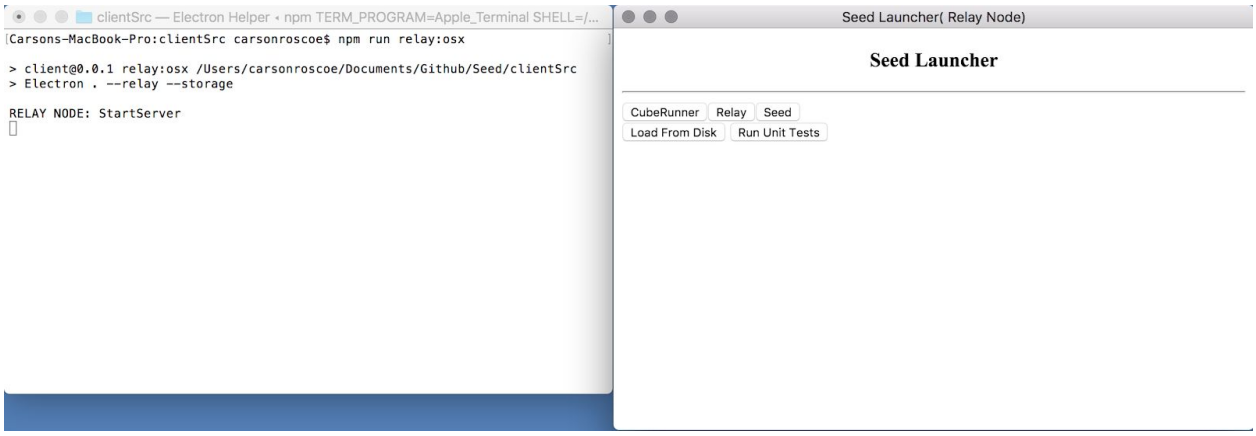
The following screenshots were taken from executing seven scenario tests, which attempt to prove the overall functionality of the system, rather than testing a particular subsystem. These tests involve the entire Seed system, modifying the ledger, creating transactions, even requiring them to validated through the entanglement.

```
### Running Scenario Test InlineModule_SuccessfullyCreated
## Begin Testing Module Inline
Creating a inline module which has a wall users cannot walk into. Users can walk left until they hit a wall.
Starting a user at (0,0) with a wall at (-3,0)
Moving the user left to (-1,0)
Moving the user left to (-2,0)
Trying to move user left to (-3,0), however a wall is in the way
Confirming users could not walk into the wall, and were stuck at (-2,0)
## Inline Test Complete
## Passed All 10 Asserts
### Running Scenario Test SeedModule_SuccessfullyCreated
## Begin Testing Module Seed
Loading Seed Module
Invoking Seed Constructor
Asserting Seed's data was loaded (Symbol was "SEED", goes to fourth decimal place, defaulted to 1000 SEED
Asserting the initial 1000 SEED supply went to the creator
## Seed Test Complete
## Passed All 4 Asserts
### Running Scenario Test SeedModule_TransfersBetweenUsers
## Begin Testing Module Seed
Loading Seed Module
Invoking Seed Constructor
Asserting the initial supply was 1000 SEED, and it all went to User1
User1 sends 500 Seed to User2
User1 sends 250 Seed to User3
User1 sends 250 Seed to User4
User1 tries to send 50 Seed to User4, however fails as they ran out
User2 sends 50 to User3
User3 sends 150 to User5
User3 tries to send 200 Seed to User5, however fails as they have insufficient funds
User3 sends 150 to User5
## Seed Test Complete
## Passed All 18 Asserts
### Running Scenario Test SeedModule_ApprovingAllowancesAndTransferingThem
## Begin Testing Module Seed
Loading Seed Module
Invoking Seed Constructor
Asserting the initial supply was 1000 SEED, and it all went to User1
User1 approves User2 to send 200 Seed
User2 sends 100 Seed from User1 to User3
User2 sends 100 Seed from User1 to User2
User2 tries to send 100 Seed from User1 to User2, however fails as their allowance has run out
## Seed Test Complete
## Passed All 12 Asserts
### Running Scenario Test SeedModule_BurningCurrency
## Begin Testing Module Seed
Loading Seed Module
Invoking Seed Constructor
Asserting the initial supply was 1000 SEED, and it all went to User1
User1 burns 150 Seed
User1 transfers 100 Seed to User2
User2 burns 25 Seed
## Seed Test Complete
## Passed All 9 Asserts
### Running Scenario Test SeedModule_ComplexScenario_TransfersAllowancesBurningAndSubscriptions
## Begin Testing Module Seed
Loading Seed Module
Invoking Seed Constructor
Asserting Seed's data was loaded (Symbol was "SEED", goes to fourth decimal place, defaulted to 1000 SEED
Asserting the initial 1000 SEED supply went to the creator
User1 approves User2 for an allowance of 250
User2 sends User2 100 SEED on User1's behalf
User2 sends User3 100 SEED on User1's behalf
User3 sends User1 50 SEED
User2 burns 25 SEED
## Seed Test Complete
## Passed All 10 Asserts
```

Appendix C3: Manual Testing Screengrabs

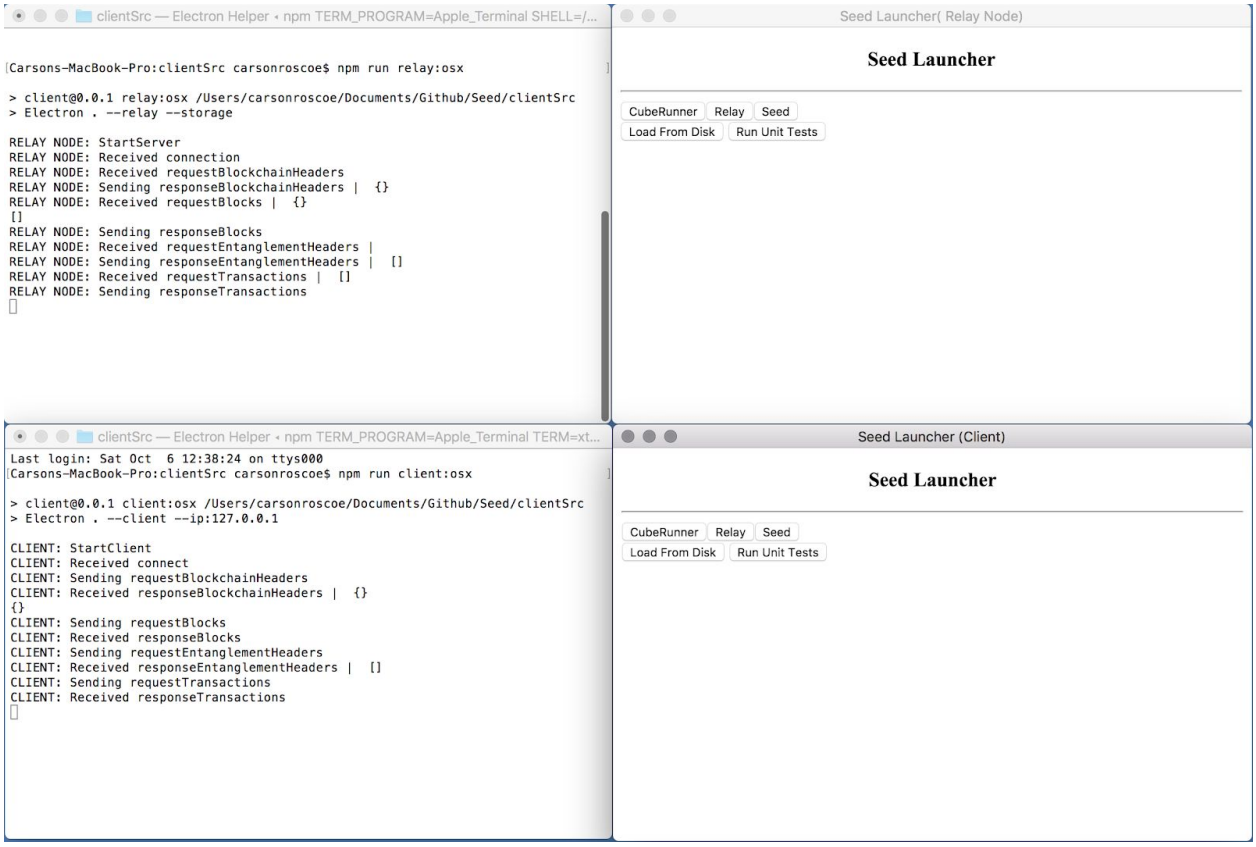
1) Seed Relay Nodes Run

The below screenshot demonstrates launching the application in a Macintosh environment.



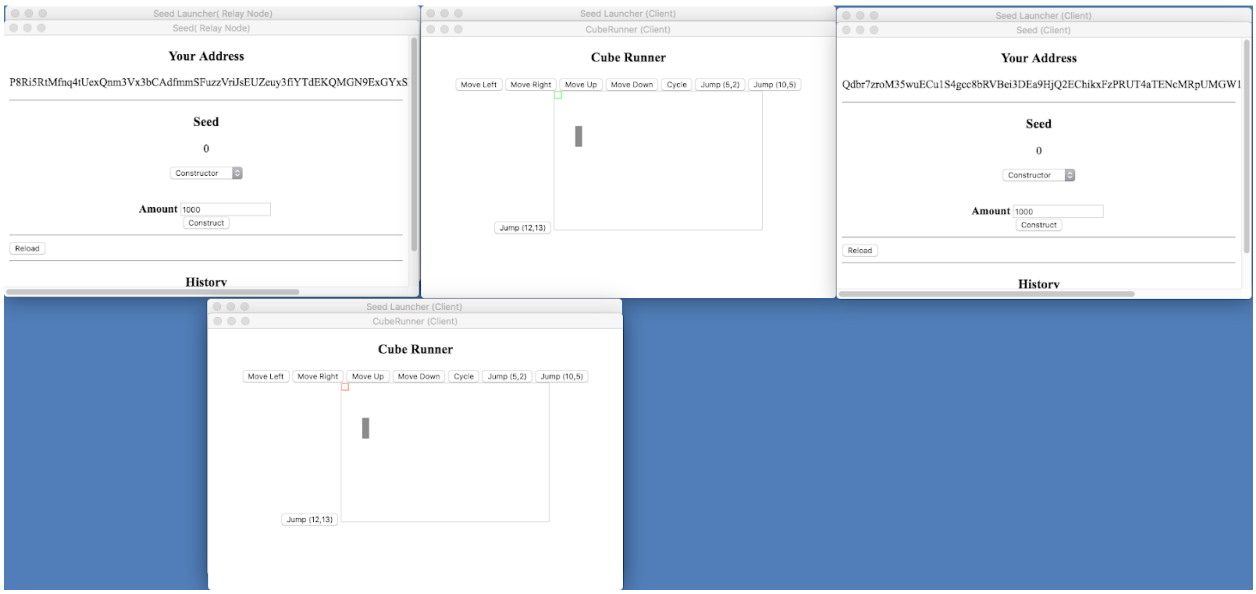
2) A Seed Client Can Run & Connect To Relay Node

The below screenshot demonstrates a client launching and connecting to the Relay Node opened in test #1.



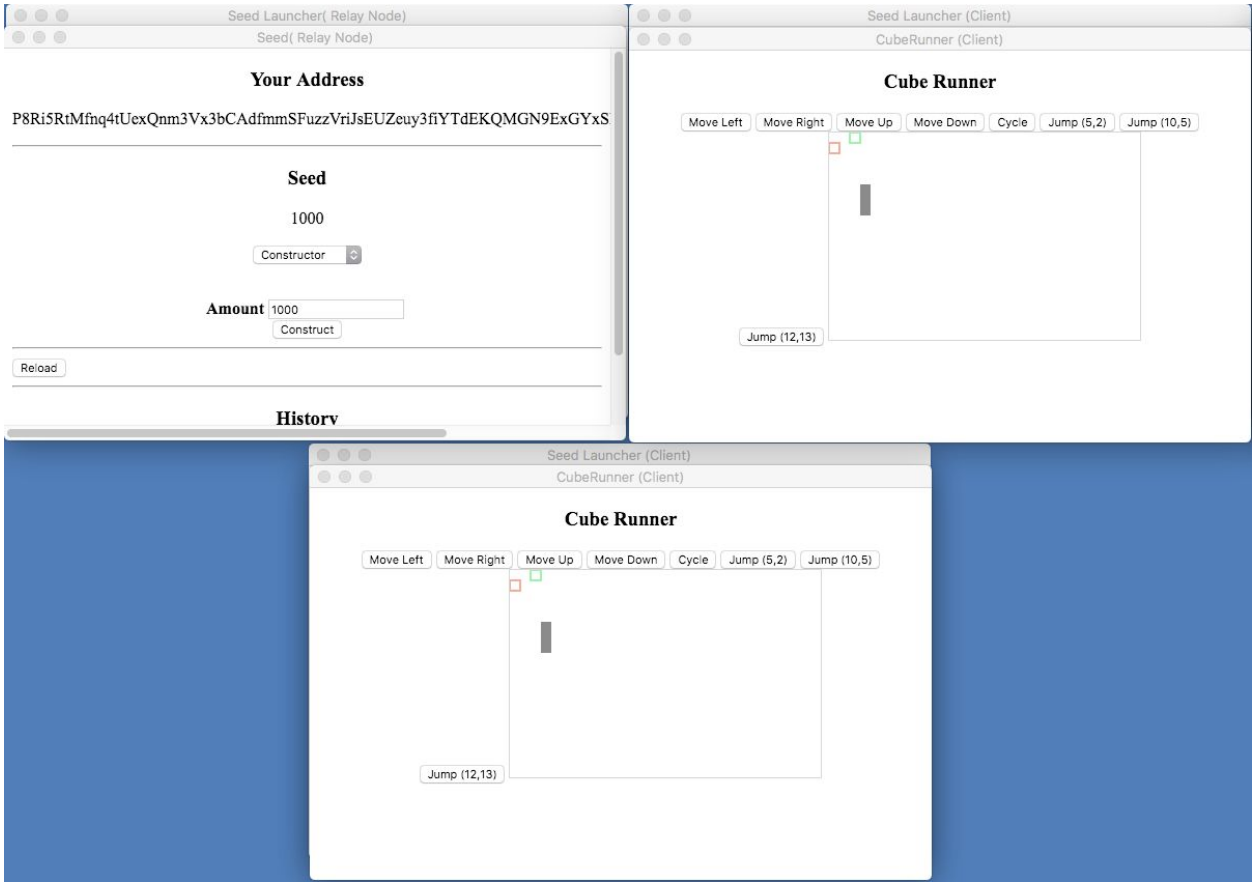
3) Multiple Clients Can Connect To A Relay Node

The below screenshot demonstrates three clients (middle, right and bottom) connecting to a relay node (top left). The relay node and client #2 launch the Seed application, while clients #1 and #3 launch the Cube Runner application.

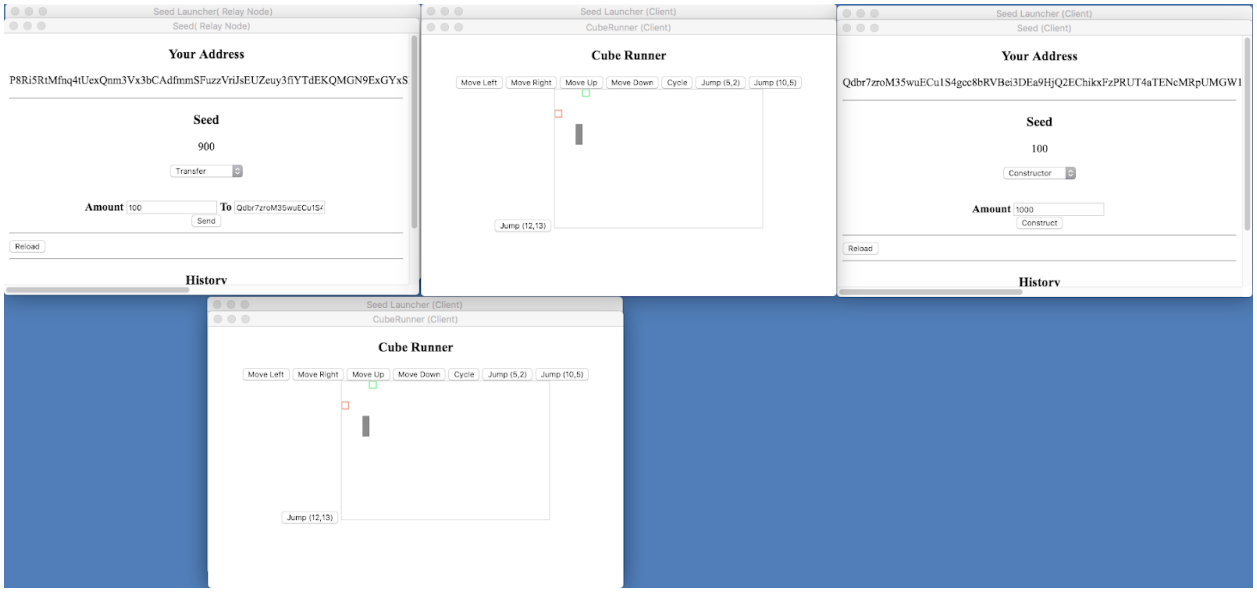


4) Users On Various Modules Can Validate Each Other

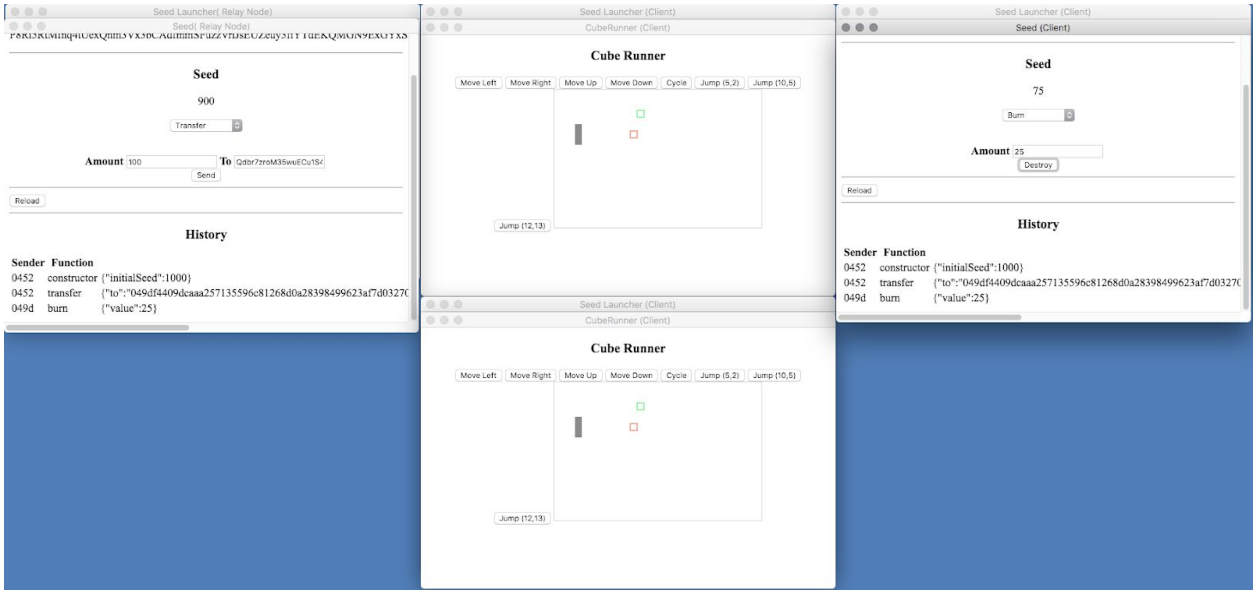
The screenshot below showcases the relay node having invoked the Seed constructor, which was validated by clients #1 and #3 moving their character in the Cube Runner game. The relay node's user has 1000 SEED, while the green and red cube have moved from the top-left starting position, proving the network successfully validated each others actions.



The following screenshot shows more transactions being processed. The relay node sent 100 of their Seed to client #3, and the players of Cube Runner have moved their squares further along the grid.

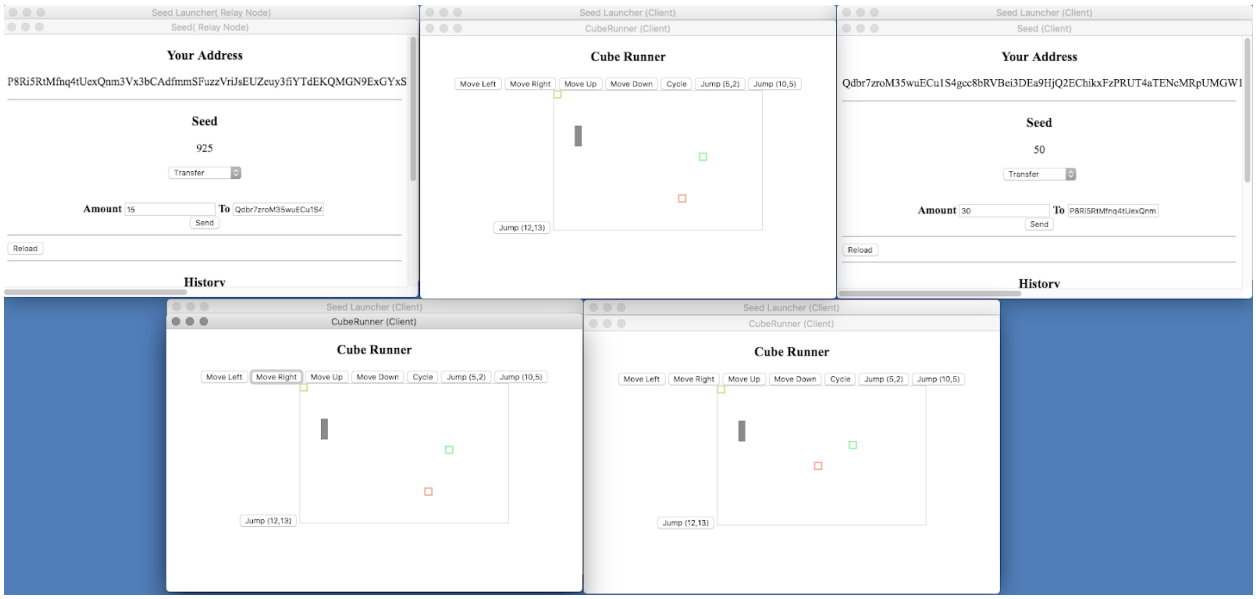


The following screenshot showcases further use of the application between the four users in the network. Client #2 burnt 25 SEED, removing it from circulation. The players using Cube Runner have traveled great distance.



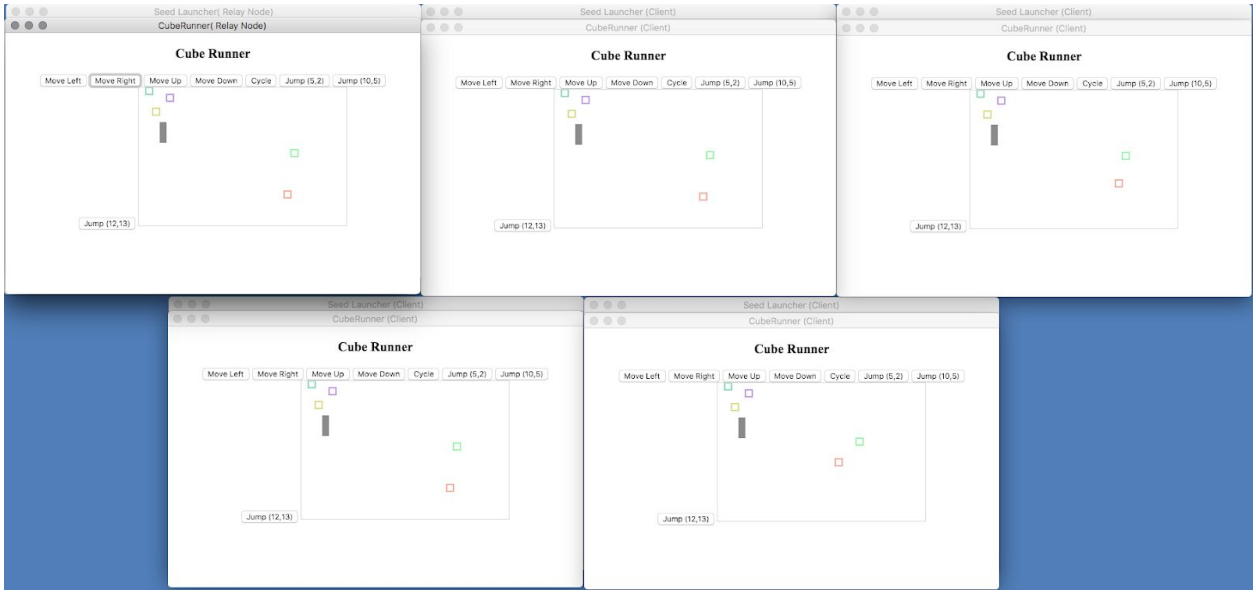
5) Clients Can Join Mid Session

The screenshot below demonstrates a fourth client connecting to the network. They have not yet interacted with the system, however their character can be seen at the top-left of the Cube Runner map in the starting position.

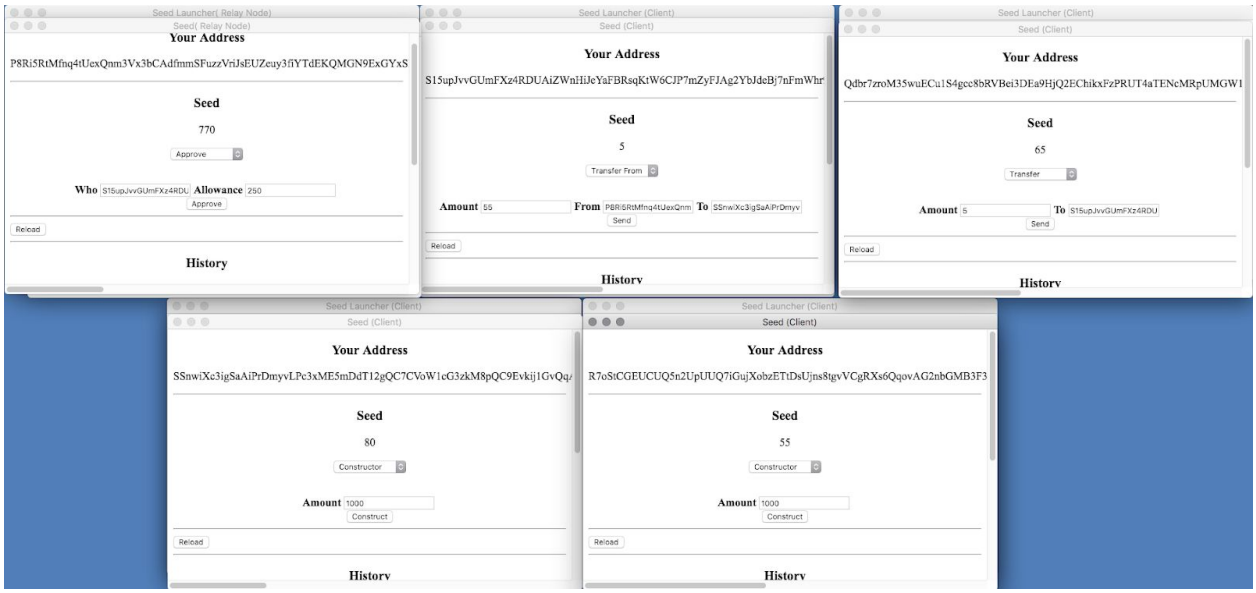


6) Clients In Seed Can Use All DApps

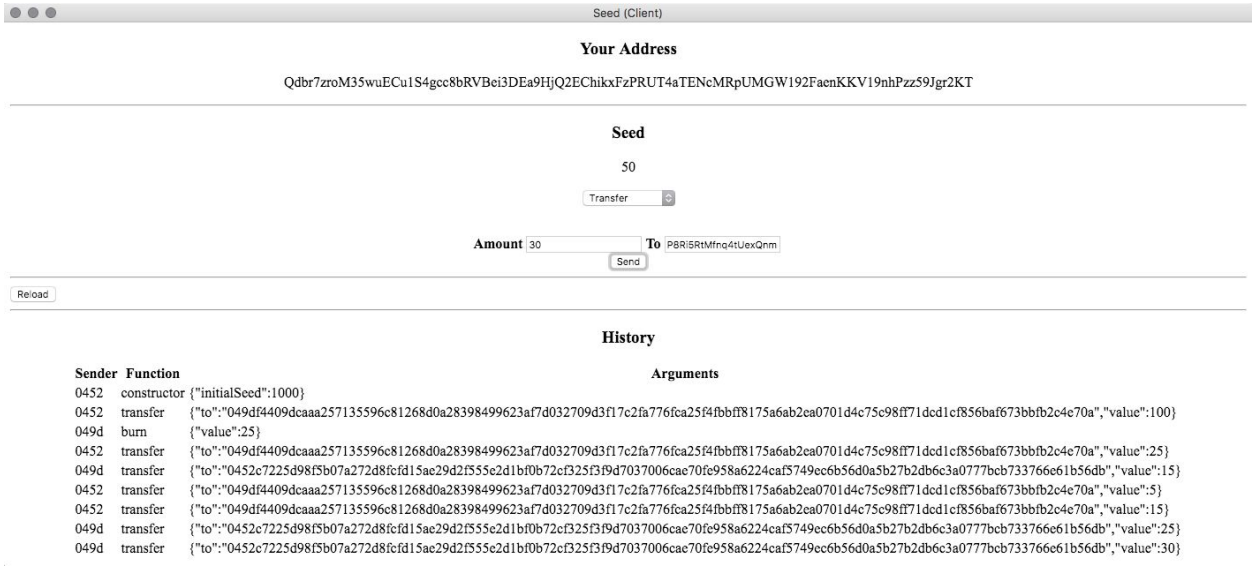
The below screenshot showcases all five clients running Cube Runner, moving their squares around.



The below screenshot showcases the same five connected clients running the Seed cryptocurrency application, having transferred funds and using each function available in the Module.

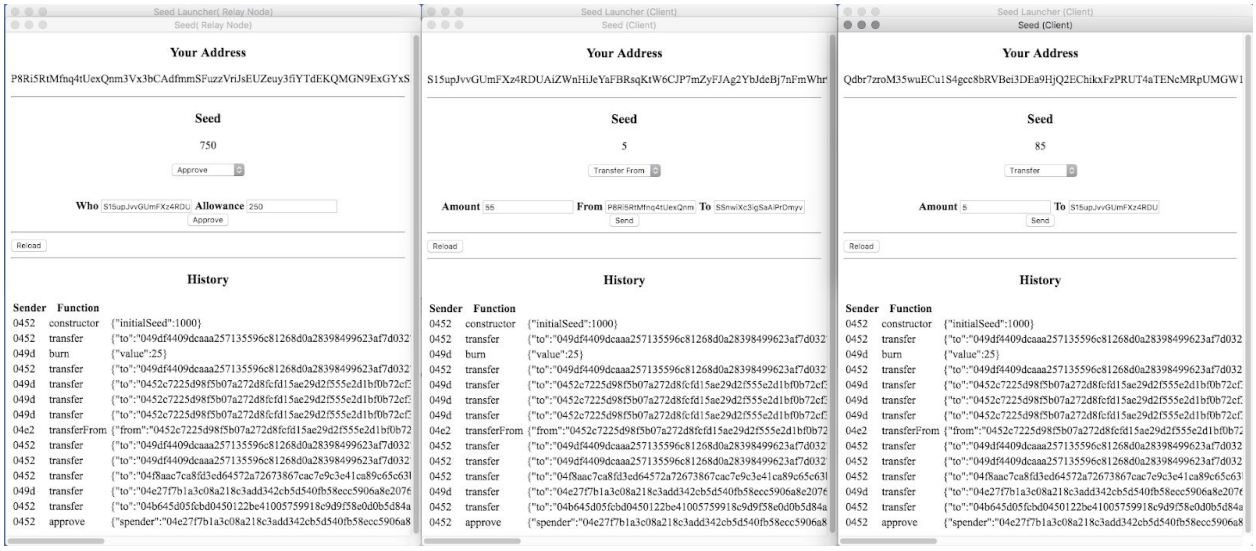


The below screenshot demonstrates an example of the history being parsed in a Seed wallet. The first transaction in history is the constructor, afterwards its transfers between users and a burn function invocation.



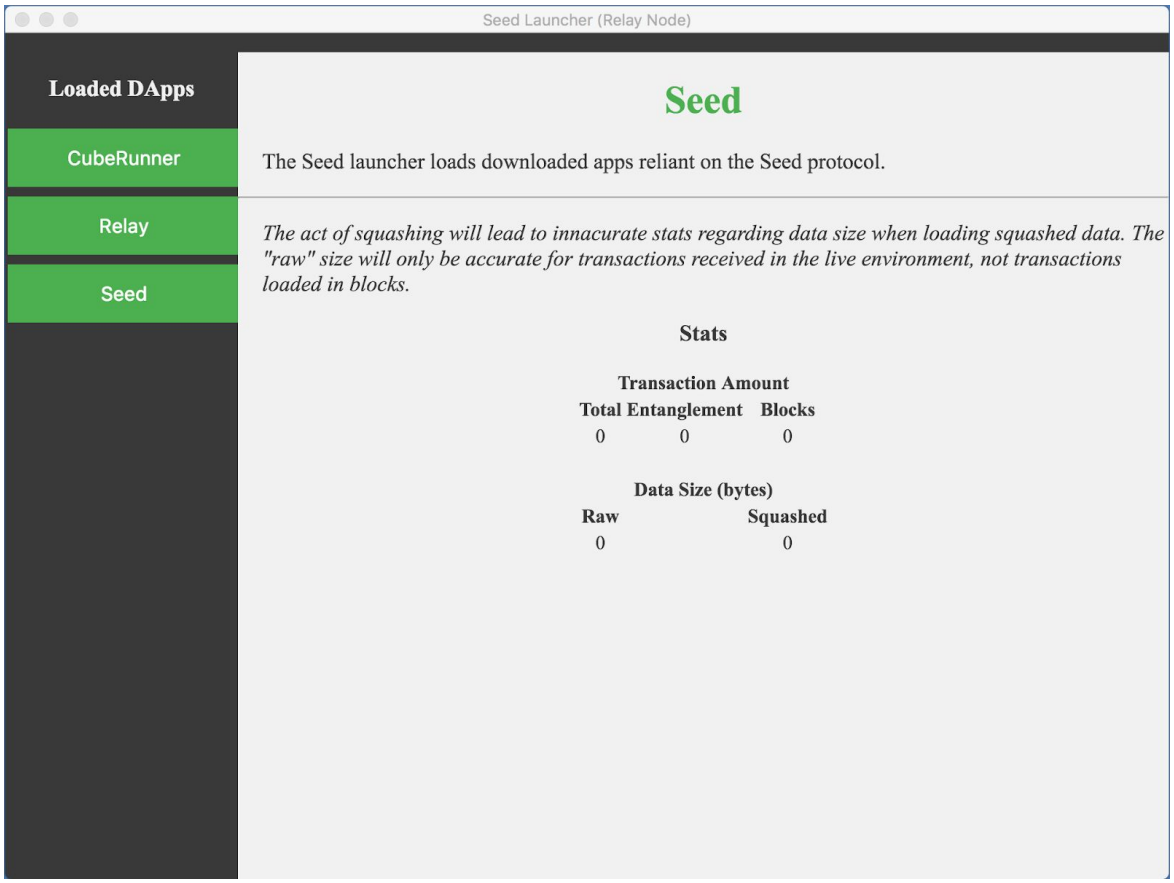
8) Networked Users Have Identical Views Of History

The below screenshot shows the displayed history between three users in a network, demonstrating how their view of history is identical.

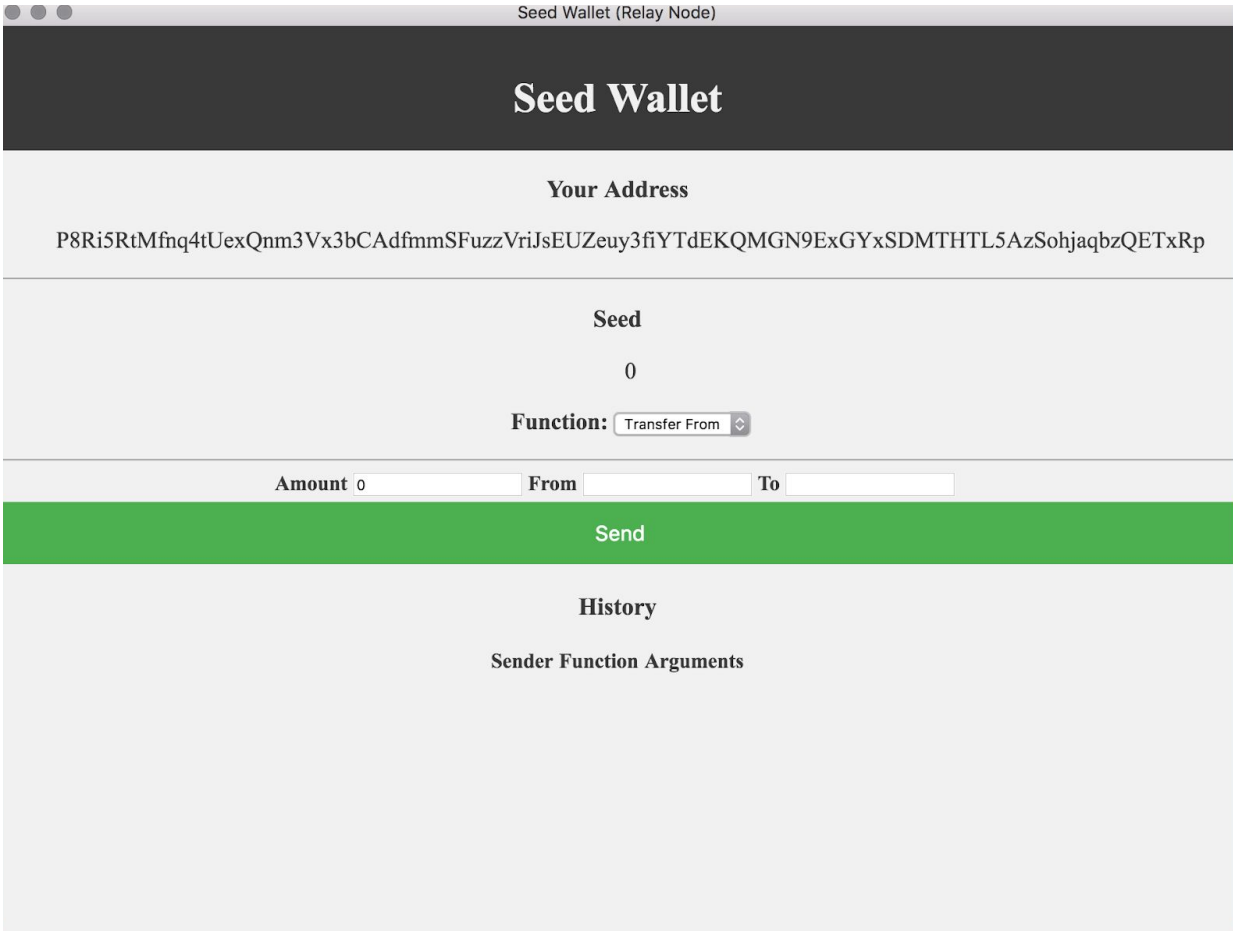


Appendix D: Final Product Screengrabs

Appendix D1: Seed Launcher

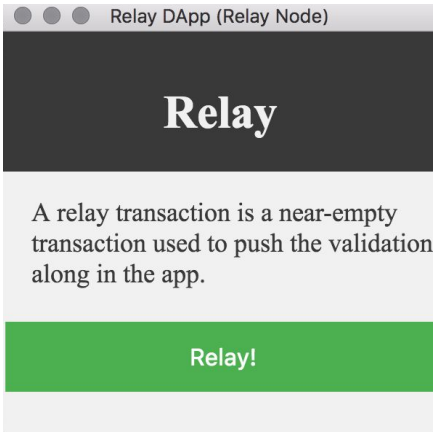


Appendix D2: Seed Wallet



Appendix D3: Cube Runner App

Appendix D4: Relay App



References

[1] Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system.

[2] Decker, C., & Wattenhofer, R. (2013, September). Information propagation in the bitcoin network. In Peer-to-Peer Computing (P2P), 2013 IEEE Thirteenth International Conference on (pp. 1-10). IEEE.

[3] Bentov, I., Gabizon, A., & Mizrahi, A. (2016, February). Cryptocurrencies without proof of work. In International Conference on Financial Cryptography and Data Security (pp. 142-157). Springer Berlin Heidelberg.

[4] King, S., & Nadal, S. (2012). Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. self-published paper, August, 19.

[5] Vasin, P. (2014). Blackcoin’s proof-of-stake protocol v2.

[6] Bentov, I., Lee, C., Mizrahi, A., & Rosenfeld, M. (2014). Proof of Activity: Extending Bitcoin's Proof of Work via Proof of Stake [Extended Abstract] y. ACM SIGMETRICS Performance Evaluation Review, 42(3), 34-37.

[7] Buterin, V. (2014). A next-generation smart contract and decentralized application platform. white paper.

[8] Kosba, A., Miller, A., Shi, E., Wen, Z., & Papamanthou, C. (2016, May). Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In Security and Privacy (SP), 2016 IEEE Symposium on (pp. 839-858). IEEE.

[9] Decker, C., & Wattenhofer, R. (2015, August). A fast and scalable payment network with bitcoin duplex micropayment channels. In Symposium on Self-Stabilizing Systems (pp. 3-18). Springer International Publishing.

[10] Kraft, D. (2016). Game Channels for Trustless Off-Chain Interactions in Decentralized Virtual Worlds. Ledger, 1, 84-98.

[11] Steemit. (2018, February 6). Seed - Dev. Discussion - Provable Execution With Function Hashing In Javascript [Blog post]. Retrieved from https://steemit.com/blockchain/@carsonroscoe/seed-dev-discussion-provable-execution-with-function-hashing-in-javascript

[12] Shor’s Algorithm. (n.d.). In *Wikipedia*. Retrieved September 25, 2018, from https://en.wikipedia.org/wiki/Shor%27s_algorithm

[13] Steemit. (2018, April 8). Seed - Dev. Discussion - Lattice Based Cryptography [Blog post]. Retrieved from https://steemit.com/crypto/@carsonroscoe/seed-dev-discussion-lattice-based-cryptography-part-1

Change Log

Proposal V2 (September 14th, 2018)

A secondary version of the proposal was resubmitted to Aman. These changes were minor, having no change in the end product, the expected behaviour, nor the scope of the project. The proposal included minor changes to the internal workings of the Seed blockchain/entanglement design. In short, these changes involved simplifying the system, removing the need for a circular blockchain or temporary transactions.

Justification

There are two reasons these changes were made. The first was because I realized the circular blockchain concept could not work while still being a distributed system rather than a decentralized one. Bitcoin and other blockchains are decentralized due to having "miners" act as special nodes, while IOTA and other DAG cryptocurrencies are distributed, with no user having a different role than one-other. I was unable to come up with a "blockchain design" that worked in a distributed fashion, unless blocks were already validated by the tangle/entanglement (in which case, the validation process has already completed, so miners are no longer required for block validation. This approach is what led to the alternative, and the block squashing design).

The other reason was to simplify the design. The older approach had more parts to it, however these extra parts did not add anything to the system. For example, a "Gate block" holding the lower half of the entanglement felt no different than having a squashing mechanism be triggered deterministically which squashed validated transactions. The more I worked on the old design, the more I kept finding flaws that were caused due to the complexity of the system, rather than due to the problem at hand.