MITIGATING SOFTWARE ENGINEERING COSTS IN DISTRIBUTED LEDGER TECHNOLOGIES

by

Jonathan Taylor Heinecke

A dissertation submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

November 2018

DEDICATION

I dedicate this document to my wife Frances and my daughter Elena. Without the support of my loving family I would have never made it through such an endeavour.

VITA

Jonathan Taylor Heinecke was born in Kalispell, MT in 1987. Raised by Tom and Terry Heinecke, Taylor grew up with strong influences in both the arts and engineering. Taylor attended Flathead High School and excelled in writing and literature. He graduated with honors in 2006.

Taylor attended Montana State University and received a Bachelor of Arts in University Studies with a minor in Writing. After graduating, Taylor ran a successful catering business, hosting private events and selling food at the Bozeman Farmers Market. Eventually, Taylor took a job at Schedulicity, a local tech startup, and closed the catering company.

In 2014, Taylor enrolled in the Computer Science Department as a Masters degree seeking student at Montana State University. For the first two years of his time in graduate school, Taylor was president of the RoboSub club. RoboSub is an autonomous underwater robot designed and built as an interdisciplinary engineering project. Taylor's role as president of the RoboSub club was to onboard and manage undergraduate engineers with the intent to work on the submarine until their senior capstone project. Following RoboSub, Taylor with the help of his advisor Dr. Mike Wittie and Dr. David Millman pioneered blockchain research in the Networks and Algorithms lab of the department. Taylor plans to graduate with a Masters of Science in Computer Science in the Fall semester of 2018.

TABLE OF CONTENTS

TABLE OF CONTENTS – CONTINUED

LIST OF TABLES

Table Page

LIST OF FIGURES

LIST OF FIGURES – CONTINUED

LIST OF FIGURES – CONTINUED

Figure Page

ABSTRACT

Distributed ledger technologies (DLTs) are currently dominating the field of distributed systems research and development. The Ethereum blockchain is emerging as a popular DLT platform for developing software and applications. Several challenges in Ethereum software development are the complex nature of working with DLTs, the lack of tools for developing on this DLT, and poor documentation of concepts for DLT developers. In this thesis, we provide building blocks that reduce the complexity of DLT operations and lower the barrier to entry into DLT development. We do this by providing a Node.js library, Ethereum-Easy, that simplifies operations on Ethereum. We implement this library into a sample application called Rock, Paper, Scissors (RPS) and built a continuous delivery, continuous integration pipeline for deploying Ethereum code (Jenk-Thereum). This thesis aims to make development on DLTs easier, quicker, and less expensive.

1

INTRODUCTION

The goal of distributed systems is to scale data processing by moving computation to distributed worker nodes. The field has seen decades of research producing novel means for distributed data processing.. Leslie Lamport pioneered many concepts in distributed computing during the late 1970's and 1980's. These ideas regarding distributed timestamps [11, 12] and Byzantine fault tolerance [14] gave way to new consensus algorithms. From unique ideas in leader election [7], to novel concepts in quorums [15], to the algorithms used in distributed computing today [13, 18], distributed systems continued to make distributed computing more efficient. Publication of several peer-to-peer file sharing papers in the early 2000's shaped research of the time with new distributed datastores based on distributed hash tables [4, 23]. Bitcoin [17] emerged in 2008 as a distributed ledger technology (DLT) with a novel process for consensus between voluntary node operators. First introduced in 2015, Ethereum [2] was the first DLT to support programmable *smart contracts*, or computer code executed across a decentralized network referred to as a *world computer*. Bitcoin removed central stewardship of transactions and Ethereum removed centralized execution of computer code. With the creation of Ethereum we are seeing a deluge of products such as decentralized prediction markets [19] and DLTs like IOTA, EOS, and others [8, 10, 20]. With each period of research in distributed systems comes technological advancements that harness more computing power from distributed systems.

DLTs provide novel means of decentralized computing. However, there are several barriers for companies to integrate DLT advancements into their business logic.

A primary issue for many companies is the lack of qualified blockchain developers in the space [1]. One big hurdle for DLT development is the complex nature of DLT software engineering. First, DLT programming requires a broad background in computer science know-how. Development on a DLT demands skills in software engineering, cryptography, distributed systems, and web development. Next, there is currently a lack of tooling in the DLT community. For example, the only library for working with the Ethereum blockchain is Web3.js. Contrast this to another database, like MongoDB, which has hundreds of libraries for reducing the challenges of using the product. Finally, there is a severe lack of documentation of the concepts and tools in the space. Web3.js is currently the most used library for interacting with the Ethereum blockchain, with version 1.0 released earlier this year. However, there is a warning on the Web3.js documentation stating, "This documentation is work in progress and Web3.js 1.0 is not yet released!" Further issues around consistency, completeness, and thoroughness of the documentation abound with this library. These issues in DLT complexity lead to an increased cost of implementing DLT solutions for companies. It is our belief that mitigating these costs is necessary in order for DLTs to reach their full business potential.

In this thesis, we provide building blocks that reduce the complexity of DLT operations and lower the barrier to entry into DLT development. First, we propose Ethereum-Easy, a Node.js library built from Web3.js that abstracts away the complexities of Web3.js to reduce the learning curve of blockchain software development. Ethereum-Easy simplifies the complex object building required in Web3.js and reduces hundreds of lines of Web3.js code down to five functions that encompass all the needed functionality for creating, deploying, using, and monitoring smart contracts. Ethereum-Easy also comes with an instructional example to aid in education and documentation further reducing the barriers to entry in blockchain

software development. Our instructional example is a rock, paper, scissors game (RPS). With Ethereum-Easy we also build a software development and testing pipeline to give programmers a familiar framework for writing Ethereum code called Jenk-Thereum. We believe Ethereum-Easy and its accompanying tools for teaching and software development will help reduce the cost of adoption in the Ethereum coding ecosystem and provide much needed industry standard coding tools for software developers.

4

BACKGROUND

The field of distributed systems has seen many iterations over the years. Early research in the field described problems in finding synchronizing actions among distributed actors in a system, paving the way for consensus protocols for achieving agreement in networks with different failure conditions. Consensus protocols gave way to powerful solutions to peer-to-peer communication and file sharing. With the emergence of Bitcoin, new concepts of distributed ledgers began to surface, solidifying the field of DLT.

Defining Distributed Systems

Distributed systems is a field of computer science research defined by a system of connected computers who coordinate actions based on message passing over asynchronous networks. In the early days of distributed systems research, thought leaders in the space identified challenges to coordinating nodes in an network. Leslie Lamport outlined many of the problems associated to distributing processing. In "Time, Clocks, and the Ordering of Events in a Distributed System", Lamport describes challenges with the ordering events in a distributed system and also provides an algorithm for synchronizing a system of local clocks to create such an ordering [11]. Lamport expands on this research by defining the Byzantine Generals Problem, a metaphoric description of the challenges asynchronous distributed processing faces in computer networks [14]. Around the same time, solutions emerged to facilitate the execution of computer programs through the process of leader election. Maekawa devised a method for creating mutual exclusion in a computer network with only $c\sqrt{N}$ number of messages, where N is the number of nodes and c is a constant [15]. Garcia-Molina invented an algorithm for dealing with node failure and leader election in a

distributed system [7]. Fischer et al defines a seminal problem for distributed systems with their "impossibility result" [5]. This paper demonstrates that every distributed systems protocol has the possibility for non-determinism. This key characteristic defines the main challenge with distributed systems and will be the focus of future consensus algorithms designed to mitigate failure states in distributed processing.

Consensus Protocols

From these initial works in distributed systems came some of the most cited papers in computer science history. These fundamental algorithms for coordination work in asynchronous systems are now applied to databases, cloud computing, machine learning, and nearly every aspect of modern software engineering.

Paxos

In one of his most referenced papers to date, commonly referred to as "Paxos", Lamport describes a protocol for consensus in a distributed system that guarantees consistency among the nodes of the system [13]. While a failure state is still possible, as described by the Impossibility Result, Paxos makes this state difficult to achieve while maintaining the consistency guarantee. A primary assumption of this particular algorithm is that collusion between nodes for nefarious purposes and corruption of messages between nodes is not considered. With that in mind, Paxos describes three kinds of nodes in the system:

1. Proposers
2. Acceptors
3. Learners

In the first round of Paxos, a Proposer sends out a "Prepare" message to the Acceptors with a unique ID. When the Acceptors hear this message, they respond with a "Promise" message to reject any other Proposers that send a message with a lower Prepare ID. If a Proposer get a majority of Promise messages back, they send out an "Accept-Request" message to the Acceptors with the value they want the system to agree on. Once the Acceptors receive this Accept-Request message, they will record the value sent by the Proposer and send an "Accept" message. However, if at this point another Proposer sends out a Prepare message with a higher Prepare ID value, the Acceptors will respond with the value they have accepted, informing the new Proposer that they are not accepting other values. The new Proposer then falls back to propagating this returned value instead of their own. These steps, described at a high level, illistrate the Paxos consensus algorithm.

However, Paxos does come with downsides. Despite this high level overview, this algorithm has many issues with implementations. It is widely regarded that Paxos is a remarkably hard consensus protocol to properly build. To demonstrate, Raft, a successor algorithm for consensus, qualitatively evaluated this protocol by teaching both algorithms to computer science students and measuring the ease to which students understood Raft compared to Paxos [18]. Their results show Paxos is a challenging concept to grasp. Additionally, there are few implementations of Paxos used in production environments compared to other consensus mechanisms.

Raft

Over twenty years after Lamport introduced the Paxos algorithm, Raft emerged as an alternative consensus mechanism [18]. Raft's primary upsell was that it greatly simplifies the consensus process, making it easier to understand at a high level and implement for production use cases.

The Raft algorithm makes similar assumptions in that Byzantine fault tolerance is not considered. In Raft, a node can have one of three states:

1. Leader
2. Candidate
3. Follower

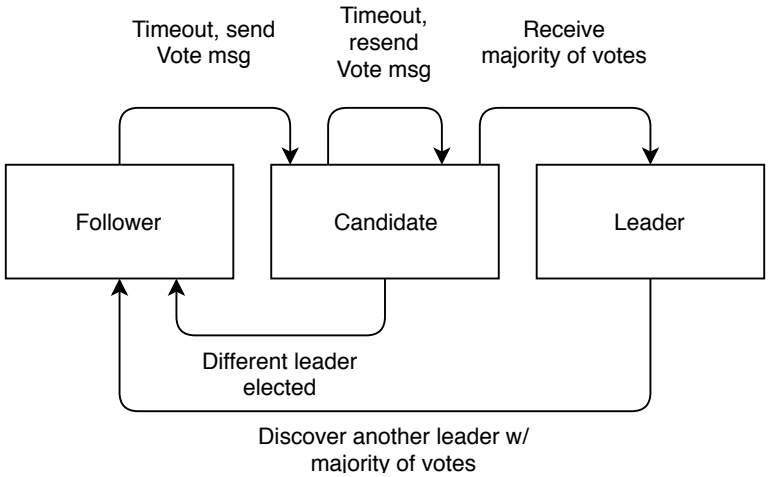

Figure 2.1: This state diagram represents the states and transitions for Raft nodes. A Raft node can be in one of three states: Follower, Candidate, and Leader. Through a series of timeouts and voting, a node progresses through the possible states, with one eventually becoming Leader.

In Figure 2.1 we see a Raft state diagram. All nodes in a Raft cluster start out in the Follower state and sets a random timeout value that it begins to count down from. Once the countdown time has completed, a node will send out a "Vote" message to the other follower nodes in the cluster saying it wants to be leader. When this occurs, a node enters the Candidate state. Ideally one node in the cluster will timeout before all the rest. If that is the case, each follower that receives a Vote message will vote for the candidate that send the message. At this point the node who timed out first

becomes leader and starts sending out a heartbeat message confirming this is the case. If two nodes timeout at the same time and both become Candidates, the node who receives the majority of the votes will become leader and start sending out heartbeat messages confirming this, at which point the candidate with the minority of the votes will realize their subservience and accept the other Candidate as Leader. If multiple nodes timeout and no one is able to get sufficient votes for leader election, they will all remain in a Candidate state until another node times out. This node becomes the new Leader, in which case all previous Candidates return to a Follower sate.

Peer-to-peer File Sharing

After a relative quiet in distributed systems research in the 1990's, a new body of research emerged progressing the field with a novel concept: The Distributed Hash Table (DHT). Acting much like a normal hash table with key-value pairs to identify data with a known reference key, DHTs distributed the responsibility of storing data and maintaining correct mappings of key-value pairs to distributed node operators in the network. Products like Napster and Gnutella proceeded DHTs, perhaps providing the catalyst to this renewed interest in distributed systems research.

DHTs explored the concept of voluntary nodes hosting database infrastructure for the benefit of all the users of the ecosystem. These distributed databases are nearly impossible to censor due to the volume of node operators. Content, including restricted content, can flow easily between users of DHTs. This censorship resistance proved useful for sharing static files. However, DHTs do not address any concept of ordering between files or pieces of content. There is no implementation of Raft or Paxos over a DHT. This lack of consensus in DHTs made keeping records of things like monetary transactions impossible.

DLTs

It wasn't until 2008, when a paper written under the pseudonym Satoshi Nakamoto appeared in cypherpunk message boards did a new novel idea appear in the space. Bitcoin emerged with new concepts for leader election, distributed record keeping, and a unique approach to digital ownership. Since its inception, Bitcoin has spawned a revolution of novel thought around Distributed Ledger Technology (DLT), the backbone of this DLT network. Ten years later in the DLT ecosystem, we see thousands of new blockchains with several gaining massive market adoption and developer focus. In this section, we focus on the primary DLTs shaping the research and development of the space.

Bitcoin

Bitcoin was the first proposed DLT which laid the foundation for a new form of consensus protocol called *Proof of Work* (PoW). PoW is the process by which the Bitcoin network, and many other DLTs, elect leaders for updating the network. Leader election is important in distributed networks. Electing leaders is an efficient way to coordinate the work of many distributed nodes. Upon electing a leader in a distributed system, all other nodes become followers and take orders from the leader node. In the Bitcoin network, node operators propose sets of new transactions to the network in the form of blocks. A block is a record of transactions that have occurred in the Bitcoin network over a period of time. Solving a guess and check puzzle called PoW allows a *Miner*, or node, in the network to become leader for the current period. As a leader, a node adds their block to the head of the blockchain, knowing that the other miners will support their proposal.

In PoW, miners calculate a hash value, by way of a hash function like SHA-

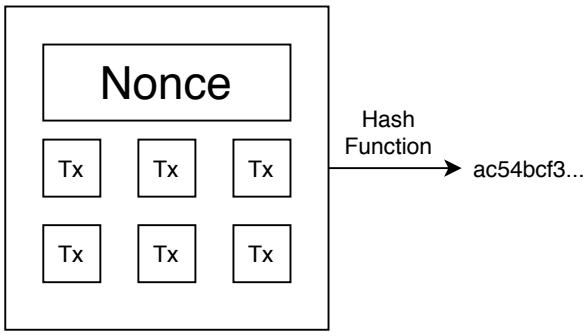

Figure 2.2: This is a representation of a Bitcoin block. A block includes all the transactions in any order over a set period of time and a *nonce* value, which is a variable controlled by the miner. Changing the nonce value generates new hash outputs of the aggregated block data. Miners change the nonce value to create new hash outputs of their blocks.

256 [16], of all of the data in their proposed block. Figure 2.2 demonstrates information within a Bitcoin block. Hashing the aggregate data within a block generates the hash output. A valid block has an output hash value below a set value threshold. When a miner finds a valid block, the block is then broadcast to the other mining nodes in the network. When other node operators receive this block they will verify the transactions in the block, accept it, gather another set of transactions from the network, and begin the algorithm again by iterating their nonce to solve PoW once more. If a proposed block is invalid, either by including transactions from accounts with insufficient funds or spending the same funds twice, the other miners will reject the proposed block. A node operator receives payment in Bitcoins when they find the correct nonce value for their sub-ledger.

Adjusting the difficulty of PoW mining is crucial for the Bitcoin network to keep block generation at a consistent pace. As miners enter or leave the network the aggregate mining power of the network fluctuates. In order to keep block generation consistent, mining difficulty in the Bitcoin blockchain is periodically adjusted based on the aggregate power of the network so that new blocks get created approximately every

10 minutes. Bitcoin increases the difficulty of mining by lowering value threshold for finding a correct nonce and decreases it by setting a larger value threshold for finding a correct nonce. If two or more miners propose a block at the same time, the block with the majority of total miners accepting it will become the next block. In this way, a blockchain validates the longest chain at all times. A valid block increases its chances of confirmation with each subsequent block added to the blockchain. In the Bitcoin network, a transaction is considered "confirmed" when it is six blocks deep, about an hour after it appeared on the blockchain. With this system, PoW provides a probabilistic solution to Byzantine fault tolerance in the Bitcoin blockchain network, embodying an novel consensus protocol in the field of distributed systems [9].

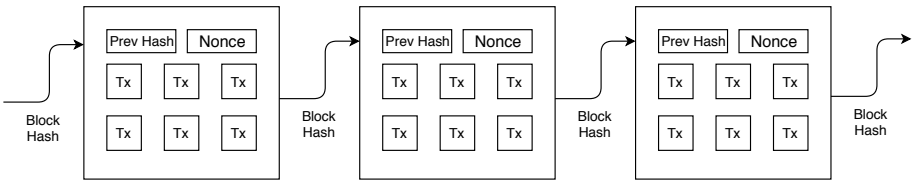

Figure 2.3: This is a representation of a blockchain. This blockchain contains blocks of information connected with pointers from one block to the next. Each block contains a ledger of transactions from the network in a given timeframe, the aggregate hash value of the previous block, and the nonce value. By including the previous block's hash value in the current block, an immutable property emerges in the blockchain.

The blockchain itself is a data structure similar to a linked list. In Figure 2.3 we can see a visual representation of a blockchain. Changing a single bit of data in past blocks creates a ripple effect, making it apparent if nodes are maintaining an inaccurate record of past blockchain blocks. This form of record keeping makes it easy for miners to ensure that fraudulent actors are not tampering with blockchain data by proposing altered histories of the blockchain.

Because the records that maintain the history of transactions in the Bitcoin network are immutable and invalid blocks get rejected by miners, a novel form of

digital ownership emerges. By auditing the entire history of the blockchain, one can determine the ownership of every Bitcoin ever generated. Any user claiming they own assets not supported by an audit of the blockchain are easy to identify as fraudulent. This form of unique asset ownership inherent in a blockchain network is a novel way to ensure one's control of a digital asset without the use of a central entity.

Over the last 10 years the Bitcoin protocol for consensus has proven to be a viable decentralized way of recording and maintaining a ledger of transactions, demonstrating probabilistic Byzantine fault tolerance in a network of unreliable node operators [6].

Ethereum

Ethereum is a DLT that hosts executable code called smart contracts as well as monetary transactions similar to Bitcoin [2]. On one hand, it does have many similarities to Bitcoin. Users can send Ether, the cryptocurrency native to Ethereum, from address to address in the network. The Ethereum blockchain proves ownership of Ether in the same way Bitcoin does, by providing an auditable chain of transactions starting at the genesis of the network. The consensus mechanism is currently PoW. Ethereum provides the same decentralized transactional ledger with probabilistic Byzantine fault tolerance consensus as Bitcoin. However, they ways that it differs from Bitcoin is where Ethereum makes its mark.

In addition to storing Ether, Ethereum addresses can host smart contracts. Smart contracts are executable computer code stored and processed on the Ethereum blockchain network. Users can write smart contracts and pay to have Ethereum miners propose the contracts to the network. After the miner confirms the contract is on the blockchain, the owners of the contract have access to the deployed code. Smart contracts have two kinds of operations:

1. Stateful

2. Stateless

Stateful operations can include logic run by the contract, updating stateful variables in the contract, or invoking the constructor of the contract. Every time there is a change to the state of the contract, the Ethereum network requires a stateful call to that contract. A stateful call to a contract will need to update the internal state of the contract or run some internal logic in the contract, so the user appends a new contract, in the same address, to the head of the blockchain. Because stateful changes to contracts require the user to append the change to the head of the blockchain, users pay miners in the form of Ether to propose this action. Once a miner appends the updated state to the blockchain and a new block includes this transaction, the user can see the new state of the smart contract by examining the most recent block in the chain or looking directly at the address associated to the contract.

Stateless operations on smart contracts read static information stored at the contract address. This is mostly limited to reading variables that are set to specific values on the contract. Because these variables are already initialized in the smart contract, they do not require a state update to the contract to read. When a stateless call to a smart contract occurs, the read operation does not require Ether to perform and there is no smart contract appended to the head of the blockchain.

Deploying smart contracts to the blockchain requires miners to be able to compile the smart contract code and execute smart contract operations. To accomplish this, the designers of Ethereum created the Ethereum Virtual Machine (EVM). This is a specifically designed virtual machine (VM) for the Ethereum blockchain environment. There are several reasons for designing a custom VM:

1. Low overhead

2. Licensing requirements of other options

3. Ethereum gas

First, other VMs options come with unnecessary features that add overhead when executing very specific code in a distributed network like Ethereum. For example, the Java Virtual Machine (JVM) provides support for reading JAR files, IPV6, several JDKs, and many more tools that simply are not used in smart contracts. Building a custom VM allows Ethereum miners a more lightweight option for executing smart contract code. Additional issues surround customizing the JVM. Licensing put in place on the JVM disallows customization at the level that retooling the software for Ethereum would require.

Another issue specific to Ethereum is the concern of spamming or denial-of-service (DoS) attacks against the network. Without measures in place, miners could continuously run useless smart contracts, requiring all nodes in network to execute unnecessary operations, limiting availability for honest users of Ethereum. To avoid spamming the network, Ethereum implements a system called Gas. Executing smart contract code costs Ether in the form of Gas to perform operations. Each operation in the EVM costs a set amount to execute. When a user wants to deploy or execute a smart contract, they specify the exchange rate of Ether to Gas they are willing to pay and a maximum amount of Ether they are willing to spend on the operation. Miners will decide to include operations on a smart contract in the next block if the amount of Ether in the exchange rate is sufficiently high. The miner will then run the operation on their EVM. If the EVM execution is successful, they will include the information as a transaction in the next Ethereum block and keep the Ether spent on Gas as a fee. If the smart contract operation fails in the EVM, either by a runtime

error in the contract or the operation reaching the Ether limit set by the user, the system refunds the Ether spent on the operation back to the original user. This system of Gas inherent in running Ethereum code not only makes DoS attacks on the network costly, it pays miners for their efforts running the network. Furthermore, it adds unique complexity to compiling and running smart contracts, creating the need for a specialty VM.

In its entirety, Ethereum creates an ecosystem for miners, entrepreneurs, software engineers, traders, and application users to interact in meaningful ways. Miners host tools on their hardware to facilitate transactions and run smart contracts in exchange for payment in Ether. Entrepreneurs create new applications over the Ethereum infrastructure to empower users and developers in this new platform. Software engineers will build these apps and discover new solutions to interesting distributed systems challenges. Traders will provide liquidity to Ether so users of the system, be it companies hosting apps or average consumers paying for services, can purchase Ether and interact with the network. And average consumers will use Ethereum, knowingly or unknowingly, as their favorite tools and apps adopt blockchain concepts into their own business logic.

Ripple

Ripple is a DLT that functions as a decentralized clearing house, currency exchange, and remittance network for banks and large financial institutions. Ripple claims to, "secure, instantly and nearly free global financial transactions of any size with no chargebacks." [21] Ripple's native cryptocurrency, XRP, at the time of writing this paper is third in overall marketcap, just following Bitcoin and Ethereum [3]. Ripple as a protocol presents a unique consensus concept in the DLT space, different from the PoW underpinnings in both Bitcoin and Ethereum.

There are some similarities worth noting between Ripple and the previous blockchains mentioned. Miners in the Ripple protocol share a distributed ledger in which they agree on the included transactions. Any interested user can run a Ripple node and participate in consensus. Similar to Bitcoin, the Ripple network specifically acts as a currency exchange without smart contract functionality.

Unlike the Bitcoin and Ethereum blockchains, Ripple fundamentally looks at consensus differently [22]. Instead of electing a leader to propose transactions, all Ripple nodes dynamically work to get a supermajority of nodes to agree on a transaction set and order to create a block. The Last Closed Ledger (LCL) is the most recent block in Ripple. A Round is the process of creating and LCL, or achieving consensus in the network. Each Round consists of iterations of message passing between each miner. When a Round starts, each miner in the network will have a candidate set of transactions they will propose to the network. They send these candidate transactions out to the other miners in the network, and likewise other nodes send their candidate sets. When a miner receives a new set of candidate transactions, this is an iteration. On each iteration, the miner will compare its candidate set to the candidate sets it receives. When a miner receives a matching transaction in a candidate set, the matching transaction in their own set gets one vote. After a set amount of time, all nodes in the network will discard transactions in their own candidate set that do not have 60% or greater votes. Now the process continues like before. Transactions get votes based on similarity between candidate sets. After another set period of time elapses, the miners cull transactions from their candidate sets with less then 70% of the votes. This iteration continues until the miners reach 80% certainty on the transactions in their candidate set. When there is 80% certainty on the ledger there is mathematical certainty in the network that a supermajority of nodes have the same ledgers, as described in the Ripple White

Paper. This ledger becomes the LCL and the algorithm starts again.

IOTA

IOTA is a DLT paving the way for micropayments in internet-of-things (IOT) applications [20]. Similar to Bitcoin and Ripple, IOTA acts as a payment gateway, tailoring itself to IOT use cases. IOTA boats feeless transactions, no miners, no consensus rounds, and no blockchain like the previously mentioned DLTs. This is all possible through a novel datastructure called "The Tangle" and a unique probabilistic consensus protocol.

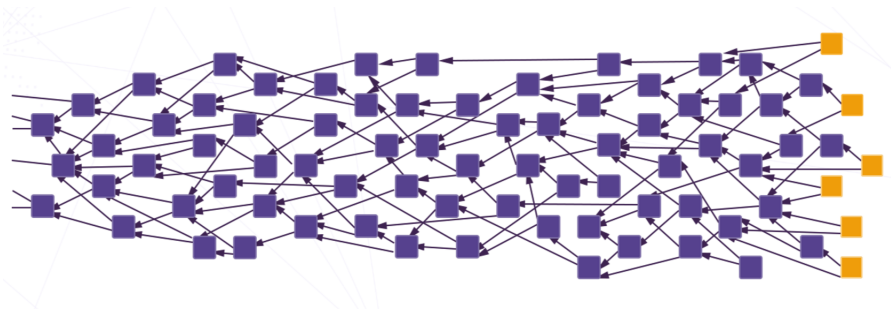

Figure 2.4: The Tangle is a directed acyclic graph (DAG) where users append their transaction the DAG by verifying two past transactions. Once the user validates two transactions and completes a very simple PoW algorithm to prevent spamming, their transaction appears on the Tangle as a "Tip", or yet-to-be validated IOTA transaction. In this diagram, Tips appear in orange and validated transactions appear in purple.

The Tangle is an entirely unique datastructure to the DLT ecosystem. In Figure 2.4, we see an example of the IOTA Tangle. Instead of a blockchain, with sequential ledgers recording transactions over a linear time span, the IOTA Tangle is a directed acyclic graph (DAG) where each node in the DAG is a single transaction in the network. Any user in IOTA can directly append to the Tangle without transmitting their request through a miner. In order to make an IOTA transaction, a user must confirm the validity of two previous transactions. As there are no consensus rounds

in IOTA, Tips appear on the Tangle when posted by the user. The IOTA DAG grows as users create a new transaction, validate Tips on the network, and append their transaction to the Tangle.

Because a transaction appears on the Tangle before validation, erroneous transactions can appear as Tips in the network. This means that the Tangle is never a valid set of transactions, but a set of transactions with a subset of valid transactions. Validating happens when a user appends a new transaction to the network and confirms two previous Tips. However, the problem remains of what Tips to validate. The solution is to do a pseudorandom walk of the DAG from a selected starting point until the user finds a Tip transaction. The IOTA foundation, the campany creating IOTA, provides software primitives in their ecosystem for facilitating this pseudorandom walk. With this functionality, the user can adjust this pseudorandom walk with an α variable to tune the entropy with which the walk follows the heaviest weighted transaction on its path. Every transaction in the Tangle has a weight value. To determine this value, one adds the inherent weight of the transaction, currently 1 for all transactions, and the sum of the weights of all the transactions that approved this transaction. The weight of a transaction gives a metric for confidence of that transaction in the DAG. An $\alpha = 0$ disregards weight values when performing the walk and an $\alpha = \infty$ makes the walk strictly follow the heaviest weighted path. In this way, a user can decide to follow the most validated transactions on the network or to take a random path through the network.

Just because a transaction on the Tangle has a validator, it does not mean it is a confirmed transaction. Consensus in the IOTA network is probabilistic, just like Bitcoin, and thus a certain amount of time needs to pass for a transaction to have the opportunity to have a high enough probability for confirmation. Derived from the weight of the transaction, a certain probability threshold exists for the network

to consider a transaction confirmed or not. Currently, a centrally run mechanism called the "Coordinator" inspects the network every two minutes and determines if transaction have the required threshold to meet confirmation. The Coordinator's job is twofold: it spams the network with 0 value transactions to weight valid transactions enough to provide meaningful measurements for confirmation and it does the measurements to indicate confirmation of past transactions. As described in the white paper, this mechanism is unnecessary if IOTA has enough transactions per second by users. IOTA engineers one day hope to eliminate the Coordinator form the network and rely on the volume of transactions to support self regulation of a minimum viable confirmation weight threshold.

20

ETHEREUM-EASY

The Ethereum-Easy library makes operations on the Ethereum blockchain easier by abstracting the challenges of the Web3.js library to 5 simple function calls. Users can transact, deploy contracts, interact with blockchain elements, and listen for general events on the Ethereum network. Ethereum-Easy is currently built and accessible on both GitHub (https://github.com/le-sanglier/Ethereum-Easy) and Node Package Manager (https://www.npmjs.com/package/ethereum-easy). The library contains detailed documentation and instructional videos to easily get users up to speed on writing Ethereum smart contracts.

Description

The Ethereum-Easy library provides 5 functions for interacting with the blockchain.

1. sendTx
2. deployContract
3. contractCall
4. contractSend
5. listen

The *sendTx* function allows a user to send Ether from one account to another over the blockchain. The *deployContract* function allows a user to deploy a smart contract to the Ethereum blockchain. The *contractCall* function returns any non-state changing information stored on a smart contract on the Ethereum blockchain, similar to that of a getter method. The *contractSend* function calls any smart

contract methods in the user's deployed contract that change the state of the smart contract. The *listen* function allows a user to define custom blockchain content and scan incoming blocks in the Ethereum blockchain for matches to this content. For example, a user can search through new blocks for particular addresses, values, or any information in the objects posted to the blockchain. When the listen function finds a match, it returns the transaction within the current block to the user.

Functions

Here is the list of functions included in the Ethereum-Easy library and detailed documentation on how to call them. Required for these functions to successfully complete is a config.json file that lives in the users application folder. In section 3.3.1 we further describe config.json.

sendTx

The sendTx function allows a user to send Ether over the blockchain. Use the function call as follows:

```
sendTx(toAddress, transactionAmount)

    toAddress : String -> This is the address that this user is
    sending Ether.

    transactionAmount : int -> This is the amount the user would
    like to send the toAddress.

    Returns : Object -> Containing information about the
    transaction that occurred.
```

Calling this function with these parameters will send Ether from the account mnemonic in the config.json file to the account listed in the toAddress.

deployContract

The deployContract function allows a user to deploy a smart contract to the Ethereum blockchain. Use the function call as follows:

```
deployContract(contractPath, transactionAmount, arguments)

    contractPath : String -> This is the absolute file path to
    the smart contract the user would like to deploy.

    transactionAmount : int -> This is the Ether amount that the
    user would like to send to the contract. This can be 0.

    arguments : Array(String) -> These are the arguments
    that the user wants to pass to their smart contract based
    on the method the user is calling. This is an optional
    parameter.

    Returns : Object -> Containing information about the
    transaction that occurred.
```

Calling this function with these parameters will deploy a smart contract from the absolute path location on a user's computer listed in contractPath to the Ethereum blockchain with the arguments provided in arguments from the account listed in the config file with the value of transactionAmount.

contractCall

The contractCall function returns any non-state changing information stored on a smart contract on the Ethereum blockchain, similar to that of a getter method. Use the function call as follows:

```
contractCall(toAddress, contractPath, funct)
```

```
toAddress : String -> This is the smart contract location
(address) on the blockchain.

contractPath : String -> This is the absolute file path
to the smart contract the user would like to deploy.

funct : String -> This is the method name on the smart
contract that the user wants to invoke.

Returns : String Containing the response from the method
called.
```

Calling this function with these parameters will return any stateless smart contract information on the blockchain stored at the address toAddress accessed with the method function.

contractSend

The contractSend function calls any smart contract methods in the user's deployed contract that change the state of the smart contract. For example, if the user wants to update an internal value in the contract, they will use this function. Use the function call as follows:

```
contractSend(toAddress, contractPath, transactionAmount, funct,
arguments)

    toAddress : String -> This is the smart contract location
    (address) on the blockchain.

    contractPath : String -> This is the absolute file path to
    the smart contract the user would like to deploy.

    transactionAmount : int -> This is the Ether amount that the
    user would like to send to the contract. This can be 0.

    funct : String -> This is the method name on the smart
    contract that the user wants to invoke.

    arguments : Array(String) -> These are the arguments that
    the user wants to pass to their smart contract based on
    the method they are calling. This is an optional parameter.

    Returns : Object -> Containing information about the state
    change to the contract that occurred.
```

Calling this function with these parameters will trigger a method on the smart contract, funct, at address toAddress on the Ethereum blockchain from the account associated to the user's mnemonic in the config.json file with the arguments needed for the method call.

listen

The listen function allows a user to define custom blockchain content and scan incoming blocks in the Ethereum blockchain for matches to this content. For example, a user can search through new blocks for particular addresses, values, or any information that may be stored in the objects posted to the blockchain. When the listen function finds a match, it returns the transaction within the current block to the user. Use the function call as follows:

```
listen(matchVar)
```

```
matchVar : String | int -> This is the content that a user
would like to search in new blocks posted to the blockchain.
```

```
Returns : Object Containing the transaction from the current
block that matches the matchVar.
```

Calling this function with these parameters allows a user to search for content defined in matchVar on any new block posted to the Ethereum blockchain.

Architecture

Figure 3.1 demonstrates the architecture of the Ethereum-Easy library. The functions sendTx, deployContract, contractCall, and contractSend use the Appender to write from a user to the Ethereum blockchain. The Listen function uses the Lisetner module of Ethereum-Easy to read information back to the user. With this architecture, we create an interaction loop with the Ethereum-Easy library functions.

With the Appender module, the user can write data to the blockchain. The Appender gathers information from the NODE_ENV variable to decide what version

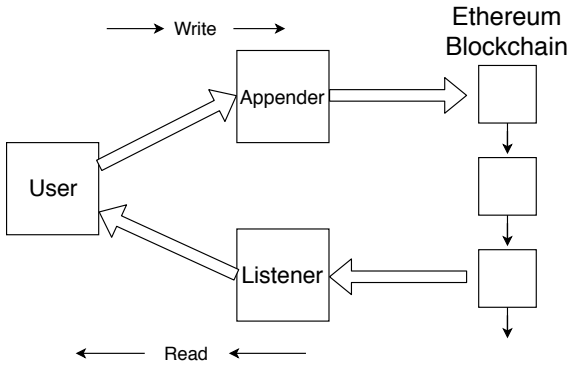

Figure 3.1: This diagram illustrates the architecture of the Ethereum-Easy Node package. Information written to the blockchain with the Appender. Information read from the blockchain uses the Listener. This creates an interaction loop from which the user can build software applications utilizing blockchain technology.

of the Ethereum blockchain the user is operating on. After determining the intended blockchain, the Appender reads more variables off of the config.json file for the function operation. After aggregating this information, the Appender it can perform the desired operation on the correct blockchain for the user. All of the functions have data returned to the user indicating the operation was a success or failure. A user can find the return values for each of the functions in the library documentation on NPM.

The Listener module facilitates reading data from the Ethereum blockchain. When a user calls the *listen* function with a "matchVar", or subscription variable, the Listener module begins reading blocks off of the blockchain and looking for the intended data. The Listener module periodically queries the blockchain endpoint listed in the config.json file looking for new blocks on the blockchain. When a new block emerges, the Lisetener combs through the block searching for any piece of information matching the matchVar function argument. Upon discovering a match, the Listener returns the transaction in the block that matches the matchVar. This will contain all of the information about the transaction in the block for the user.

With both of these modules, Ethereum-Easy allows a user to create an interaction loop on the Ethereum blockchain. This loop is valuable in providing read and write functionality to the Ethereum blockchain. The interaction loop provides the backbone for any application looking to harness Ethereum blockchain technology. We explore the possibilities of the Ethereum-Easy interaction loop with two applications. The first is a rock, paper, scissors game (RPS) that uses the append and listen functionalities of our library. Second is an Ethereum smart contract CI/CD pipeline (Jenk-Thereum). In both applications, the interaction loop is crucial for blockchain interaction.

config.json and Environment Variables

Before any of these functions will work, a developer needs to setup a configuration file to store needed information required in Ethereum operations. Ethereum-Easy pulls information from a *config.json* file that located in the source folder of the project for two reasons. The first is that there is a lot of redundant information required to run these functions. Putting this information in a config.json file removes the need to provide this information as arguments to the functions every time a user calls them. Second, some of this information is very sensitive. For security reasons, it is prudent to define these variables outside of the application code.

Creating an Ethereum-Easy config.json file is easy. The structure of the file is as such:

Each of these values is very important to have set correctly. The *mnemonic* value is the secret mnemonic for an Ethereum private key. This is a very sensitive piece of information used in nearly every function call in Ethereum-Easy, so locating it in the config.json file is necessary. Generating a mnemonic for working with the Ethereum network can be challenging if a user is new to working with this network.

```
{
  "mnemonic": "<mnemonic>",
  "ropsten": "<Ropsten endpoint>",
  "rinkeby": "<Rinkeby endpoint>",
  "kovan": "<Kovan endpoint>",
  "ethereum": "<Ethereum main net endpoint>",
  "network_prerference": "<preferred testnet>",
  "ganacheEndpoint": "http://localhost:8545",
  "gasPriceOracle": "https://ethgasstation.info/json/ethgasAPI.json"
}
```

Thankfully, MetaMask is a service that makes generating and managing mnemonics very easy. More on this service in section 3.3.2.

The *ropsten*, *rinkeby*, *kovan*, and *ethereum* values are endpoints to each of the Ethereum networks. Ethereum has several test networks to make building and testing smart contracts easy and free. Because the live Ethereum network costs real money in the form of Ether to work with, each of these three test networks (Ropsten, Rinkeby, and Kovan) are available. Developers can get free fake Ether for using these testnets from sources called *Faucets*. Faucets can be easily found online. The *endpoints* or access points to these networks can be another challenging task to implement. Again, there is a service called Infura which provides these endpoints for free. More information on Infura in the Section 3.3.3 of this paper. Finally, a user can set which network they would like to work with in the *network_prerference* value of the config.json file. This value can be ropsten, rinkeby, or kovan.

Ganache is a local test blockchain that is fast and efficient for testing and

development. The *ganacheEndpoint* is similar to the endpoints for the Ethereum live net and test networks, except that it lives specifically on a local machine. A user can install Ganache by running "npm install -g ethereumjs-testrpc" in their terminal. Once installed, Ganache will automatically operate through port 8545, unless changed by the user. If for some reason a user needs to change the default port setting for Ganache, they can update that port number here in the config.json file.

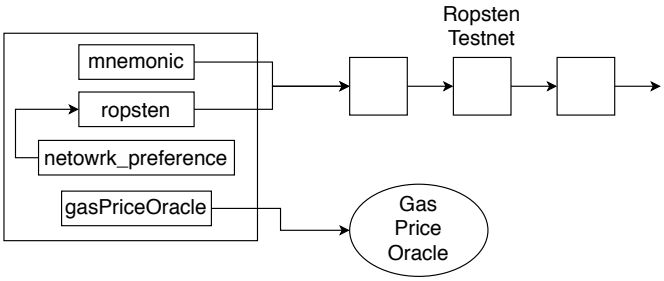

Figure 3.2: This diagram depicts an example of the config.json file when used in an Ethereum-Easy function call. In this example, the *network_preference* variable tells the Ethereum-Easy that we want this function to use the Ropsten testnet. The *ropsten* variable provides our Ropsten testnet endopoint. Our *mnemonic* variable provides the private key information for our wallet required for any blockchain operation. The endpoint and mnemonic values together give us access to the blockchain of choice. Finally, to calculate the gas needed for this operation, the config.json file informs Ethereum-Easy of the gas oracle that we would like to use.

Next, when working with the Ethereum blockchain, a user needs to query the network to know how much to pay for transactions on the network. The *gasPriceOracle* is a link to an outside oracle service to supply information about the current rate of exchange between Ether and Gas such that Ethereum-Easy can accurately calculate the cost of a transaction. Ethgasstation.info is a commonly used oracle service for this functionality, but the user can use any oracle they choose and update it with this value in the config.json file.

Finally, there is one environment variable that the user needs to set. *NODE_ENV* determines what network a user is going to be working with in the Ethereum-Easy

function calls. NODE_ENV has three possible options: *dev_local*, *dev_live*, and *prod*. The option dev_local makes all of the Ethereum-Easy functions operate on the Ganache local test blockchain. The option dev_live makes all of the Ethereum-Easy functions operate on the Ethereum test networks, specifically the one defined in config.json under network_preference. The option prod makes all of the Ethereum-Easy functions operat on the live Ethereum blockchain.

Figure 3.2 demonstrates how the config.json file gets used in during a function call. All of the function calls in the Ethereum-Easy library use this dataflow to operate on the blockchain.

MetaMask

MetaMask is a Chrome browser extension specifically to help with Ethereum account management. Metamask provides users mnemonics for interacting with the Ethereum blockchain, manages users private keys, and shows transactions from accounts connected to the Ethereum main network and all Ethereum test networks. This extension is free of charge and a necessary element in any Ethereum developers toolbelt. Using Metamask, users of Ethereum-Easy can easily generate the mnemonic value in their config.json file and manage their transactions on the Ethereum main and test networks.

Infura

Infura is a service that provides API endpoints, a requirement when using Web3.js. An endpoint is an API to an Ethereum miner. Web3.js calls endpoints *providers* and are a necessary element for building Web3.js objects that operate on the blockchain. The provider in Web3.js defines the connection protocol and the access point the user is going to use to connect to the network. From deploying and invoking calls on smart contracts to sending Ether between users in the network, a

provider is an essential access point to the Ethereum network. Anyone can setup their own miner for the Ethereum network and use the endpoints provided from this infrastructure. However, Infura provides a service where a user can use their endpoints, making programming on the blockchain much easier.

Installation

Installing the Ethereum-Easy libary into a project is very simple. The Node Package Manager (NPM) is a package manager and repository for libraries for the Node.js language. To install Ethereum-Easy, simply navigate to NPM and search for the name of the package. The page for Ethereum-Easy contains instructions for installing this into a project. From the command line, navigate to the source folder of a project and run the command "npm install ethereum-easy". Now the project will support using Ethereum-Easy functions.

Import into Project

Once a user has their config.json file ready with the necessary values, they can begin using Ethereum-Easy in their codebase. Any module which requires Ethereum-Easy needs a line of code importing the package into the module. Simply add "const easyEth = require('ethereum-easy');" to the top of a module to import the package.

Dependencies

The Ethereum-Easy library depends on several other Node packages for building Ethereum objects, managing private keys, running local blockchains, and compiling Ethereum code. As follows are all the dependencies for Ethereum-Easy and their descriptions.

Web3

Web3.jsis the a library for interacting with Ethereum miners using HTTP or IPC connections. It is a quintessential library for Ethereum developers and smart contract engineers. It currnetly has close to 90,000 downloads per week from NPM and is one of the most downloaded blockchain packages in the entire ecosystem. It is a massive library with lots of functionality. This functionality comes at the price of understandability, which is why we created Ethereum-Easy.

Solc

As mentioned, Ethereum has a special virtual machine called the EVM to execute smart contract code. With the EVM is a new programming language called Solidity. Solc is a Javascript binding for the Solidity compiler, allowing Ethereum-Easy to compile smart contracts. When deploying smart contracts to the Ethereum blockchain or calling functions on previously deployed smart contracts, compiling the smart contract on your local machine is necessary.

Truffle-Hdwallet-Provider

A twelve word mnemonic is one way to reference private keys in the Ethereum blockchain. A mnemonic is particularly helpful as it can reference many different addresses, all controlled from the one mnemonic. One mnemonic can managing many addresses, which is a helpful tool in the smart contract engineering process. Truffle-Hdwallet-Provider provides an interface for easily referencing different accounts under the same mnemonic. Ethereum-Easy uses this package for manipulating the mnemonic listed in the config.json file.

RPS

To demonstrate the functions of Ethereum-Easy, we built a Rock, Paper, Scissors (RPS) game that is entirely run on the Ethereum blockchain. All game logic and message passing between players occurs on the blockchain. The game uses the functions in the Ethereum-Easy library, features a commitment scheme on the blockchain to keep players from cheating, and uses multiple smart contracts to facilitate game play.

Description

The game follows standard Rock, Paper, Scissors rules. In Rock, Paper, Scissors each player *throws*, or creates a figure with their hand, symbolizing Rock, Paper, or Scissors. In the game, Rock beats Scissors, Scissors beats Paper, and Paper beats Rock. The winner of the game is the player who throws the winning figure. If there is a tie, the game repeats.

In our version of RPS, the game starts when each player runs the RPS application, entering in their throw as an argument in the command line. The application will go through a series of game states, each facilitating some interaction on a smart contract for the game logic and communicating their current game state over the blockchain. Two smart contracts, one for game logic and the other for communication, accomplish this interaction. The game also features a commitment scheme so no player can determine what their opponent threw before each of them have committed their answer to the blockchain. As mentioned, all operations on the blockchain for RPS use function calls provided by Ethereum-Easy.

Architecture

Figure 4.1 depicts the architecture for our RPS game. With this layout, each node is able to read and write information to the Ethereum blockchain. Reading and writing to the blockchain is necessary for state dissemination and game message passing. Each state of the RPS game states requires reading or writing on one of the smart contracts.

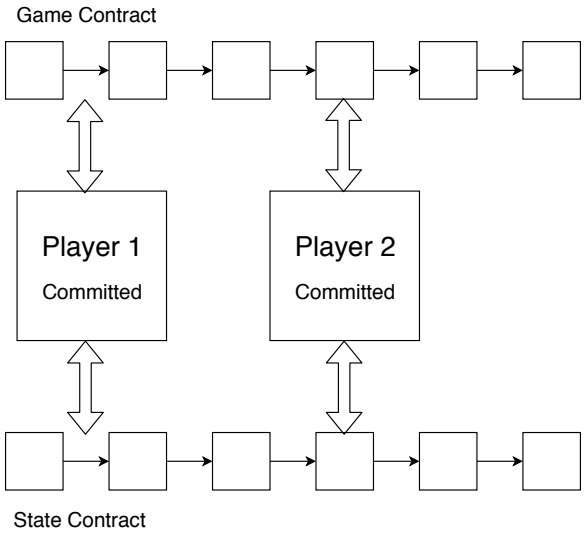

Figure 4.1: This diagram illustrates the architecture of the RPS blockchain application. Player 1 and Player 2 are both implementations of the Ethereum-Easy interaction loop. Each player can read and write to two smart contracts. The first smart contract is the Game Contract and the second smart contract is the State Contract.

Smart Contracts

There are two smart contracts that make the RPS game possible. The first smart contract, called the Game Contract, handles player throws. This is also the contract that facilitates our commitment scheme, hiding the throws until a final reveal phase. Player 1 at the beginning of the game will deploy this contract so that it is a fresh copy of the contract. This contract has a constructor function to initialize the owner

of the contract as the player who first deploys it. It has two setter functions to allow each player to provide their throws to the smart contract. Finally, it has a reset function for the reuse of this smart contract in the case of a tie.

The second contract, called the State Contract, is for message passing between the two players. This contract is already deployed to the Ethereum network. Each player is aware of the address so they can access the contract. Passing message is how each player communicates what game state they are in. This contract has several setter functions the players invoke as they progress through their game states. First, each player has a setter function to tell the opponent what game state they are on. There is also a setter function for communicating the address of the deployed game contract once it is live on the Ethereum network. Next, there are two setter functions for communicating the reveal key for the encoded committed answers of each player. Finally, this contract has a reset game function to clear out set variables for subsequent games.

Game States

As the game progresses, each node will enter a series of states and communicate those states via a smart contract to their opponent. This is how each player knows what actions to take and the game moves forward. The first game state is *Ready*. As each player comes online, they post their Ready message to the smart contract.

Once Player 1 is in the Ready state and confirms that Player 2 is in the Ready state via the smart contract, Player 1 will move into the *Deploying* state. In this state, Player 1 will deploy the game logic smart contract for recording Player throws. When Player 1 is in the Deploying state, Player 2 will wait for confirmation that of the deployed contract. When the contract is live on the blockchain, Player 1 will post their throw to the contract as an encoded message, following the commitment scheme

of the game.

Once Player 1 submits their throw to the contract, they will enter the *Submitted*, state. When Player 2 has confirmation that Player 1 is in the Submitted state, they will also submit their throw to the contract. When Player 2 commits their throw, they update their state to Submitted on the communication contract as well.

Now Player 1 sees that Player 2 is in the Submitted state and moves on to verifying Player 2 has actually submitted an answer. Player 1 examines the smart contract to see if there is an encoded answer in Players 2's answer variable. Player 1 then records this answer for later deciphering. When Player 1 sees this, they move into the *Committed* state. And likewise, when Player 2 sees Player 1 is in the Committed state, they verify Player 1 has an encoded answer on the blockchain. If they do, Player 2 records this value and enters the Committed state as well.

Finally, when Player 1 has confirmation that Player 2 is in the Committed state, they will post their caesar cipher key to the communication smart contract. When this key is live on the smart contract, Player 1 enters the *Reveal* state. This prompts Player 2 to read Players 1's cipher key and post their own key to the communication smart contract. When Player 2 completes this, they enter the Reveal state. At this point, Player 1 can read Player 2's cipher key. Now both players can decode their opponent's throw and decide who won the game.

Commitment Scheme

The RPS game uses a basic commitment scheme to make it so neither player knows what the other has thrown until both players are sure their opponent has thrown their choice of rock, paper, or scissors. To do this, each player encodes their throw in a caesar cipher. A caesar cipher is a substitution cipher. A substitution cipher shifts each letter of the alphabet certain number of places down the alphabet.

When each player sets their initial throw in the smart contract, the other player will not be able to read the answer in plain text. Once both players commit their encoded answers to the blockchain, they pass the cipher key to one another to reveal the answer. This ensures both players have zero knowledge of their opponents answer until each throw is on the blockchain.

JENK-THEREUM

Another implementation of the Ethereum-Easy library is a one step build and deploy tool and monitoring tool for writing smart contracts. To make the Ethereum-Easy library work better for this use case we created Jenk-Thereum, a wrapper library around Ethereum-Easy for working with build pipelines and out-of-the-box monitoring capabilities. With the Appender functionality of the library, we can deploy code to the local test blockchain and Ethereum testnet, run software tests on the smart contract, and deploy to the live Ethereum network. With the Listener function in the library we are able to create a monitoring tool for recognizing calls to a smart contract on the Ethereum blockchain. In the build pipeline we use Jenkins, a third party software testing and deployment tool used by many software engineers today. These tools provide industry standard devops functionality to writing and monitoring Ethereum smart contracts.

Jenkins

For building the deployment pipeline, we integrate our Ethereum-Easy wrapper, Jenk-Thereum, with the Jenkins software development tool. Jenkins is the leading open source one step automation tool, providing support for building, testing, and deploying software in an enterprise environment. We choose to use Jenkins for two reasons. First, it provides a well tested and supported tool for the automation portion of the one step build and deploy tool. Second, it is a well know and highly used tool in the software engineer's tool belt. Using Jenkins makes this implementation of Ethereum-Easy user friendly to any developer looking to get into smart contract development.

Jenkins uses "pipelies" to create an ordering of the steps required in the building,

testing, and deploying process. Normally, a Jenkins pipeline will pull code from an source code manager (SCM), run any scripts needed to initialize the code, invoke unit tests locally on the code to asses quality, and then deploy the code to a server. Implementations for different use cases will have different work flows, but the process of pulling code from an SCM, running tests, and then deploying the code to a server is a very common application of Jenkins.

System Architecture

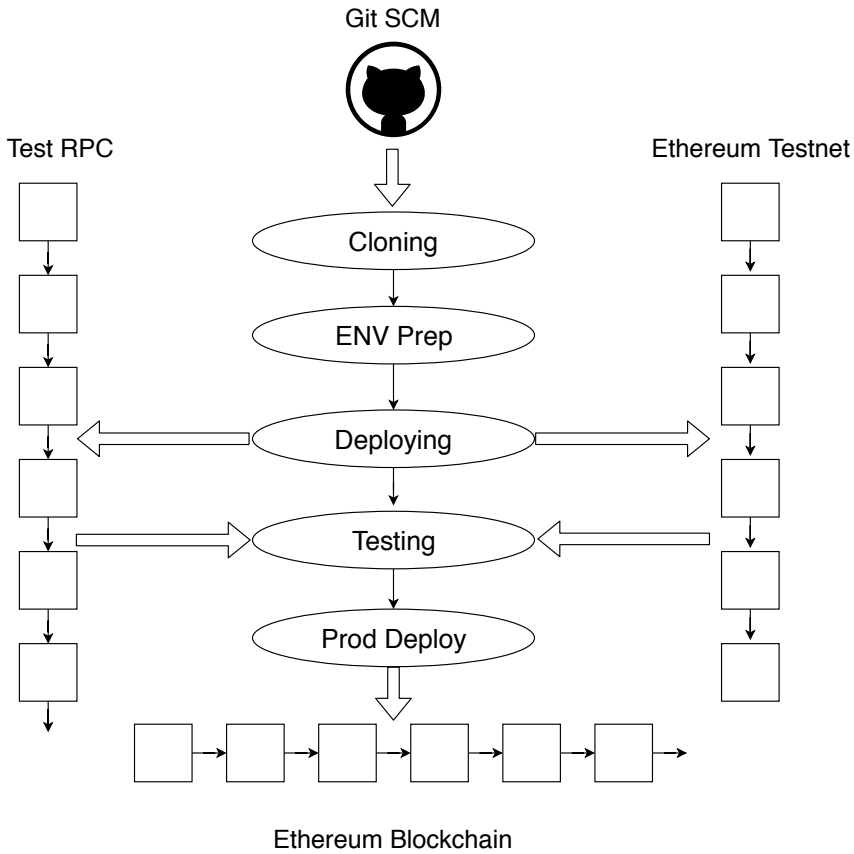

Figure 5.1: This diagram illustrates the Jenkins delivery steps in Jenk-Thereum.

Figure 5.1 illustrates the Jenkins architecture for this one step build and deployment process. The first step is to clone the Git SCM repository that contains

our smart contract code. Upon pulling this code, Jenkins will enter the *ENV Prep* or Environment Preperation step. In this step, Jenkins runs "npm install" on the folder that it pulled from Git. This installs the needed npm package, Jenk-Thereum, that contains the modules used for the deployment steps in the pipeline. Once the package install is complete, jenkins will deploy one copy of the smart contract to the Test RPC blockchain on the user's local machine and one copy of the smart smart contract to the Ethereum testnet of the users choosing. If these contracts deploy to each blockchain correctly, Jenkins will run the user defined tests from the "test" folder on each contract. Getting a passing responce from these tests will prompt the user if they want to deploy their contract to the main Ethereum blockchain. With approval from the user, Jenkins will deploy this contract onto the Ethereum blockchain, putting this contract into a production environment. These are all of the steps that a smart contract will go through in the Jenk-Thereum delivery pipeline.

Contract Monitoring

A second tool built on Ethereum-Easy is a smart contract health monitoring utility. Using the Listener module of Ethereum-Easy, we are able to watch for calls to a deployed contract. When we monitor a smart contract we can see how many times it gets called, who is calling it, and as well as other metadata related to the contract invocation. Monitoring a smart contract in this way gives helpful insight to the developer about frequency of use of the contract in the production environment.

Invoking the Monitor module in the Jenk-Thereum package will automatically start watching the Ethereum blockchain for calls to the most recently deployed smart contract. Figure 5.2 describes how this process works. Currently the Monitor will display the number of times it recognizes an Ethereum block with the subscription address to the user.

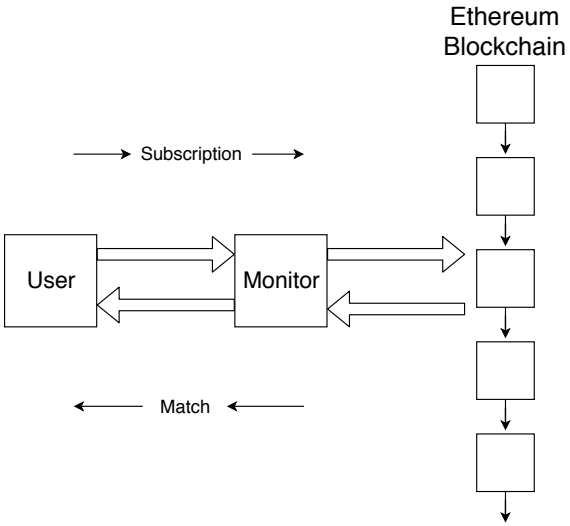

Figure 5.2: This figure depicts the Monitor function of Jenk-Thereum. The Monitor pulls the address information tied to the most recently deployed smart contract stored in local memory on the user's machine. The Monitor subscribes to this address and begins to scan each new block on the Ethereum blockchain for matching addresses. Each time it finds a matching address, the Monitor copies the content of that block.

RESULTS

Our goal with Ethereum-Easy and it's accompanying products, RPS and Jenk-Thereum, is to simplify operations on the Ethereum blockchain. To demonstrate how Ethereum-Easy simplifies blockchain operations, we looked at the code volume and complexity of steps for implementing Ethereum-Easy in Web3.js, the current industry standard library for blockchain operations.

Complexity and Code Volume

The first way we measure simplicity of our library is by looking at the complexity of object building when interacting with the Ethereum blockchain. Each function in the Ethereum-Easy library processes a specific order of steps to create an object that Ethereum miners accept and propose to the network. Here is a list of the steps needed for each method in our library:

`sendTx`:

1. `Create provider`
2. `Build Web3.jsobject`
3. `Calculate transaction gas`
4. `Get wallet transaction number`
5. `Build raw transaction object`
6. `Sign transaction`
7. `Serialize transaction to hex`
8. `Send transaction to miner pool`

`deployContract`:

1. `Compile contract`
2. `Create provider`

3. Build Web3.js object
4. Calculate transaction gas
5. Get wallet transaction number
6. Build raw transaction object from compiled contract
7. Sign transaction
8. Send transaction to miner pool

contractCall:

1. Compile contract
2. Create provider
3. Build Web3.js object
4. Calculate transaction gas
5. Get wallet transaction number
6. Build raw transaction object from compiled contract
7. Sign transaction
8. Send transaction to miner pool

contractSend:

1. Compile contract
2. Create provider
3. Build Web3.js object
4. Calculate transaction gas
5. Get wallet transaction number
6. Build raw transaction object from compiled contract
7. Sign transaction
8. Send transaction to miner pool

listen:

1. Create provider

2. `Build Web3.js object`
3. `Request blocks from miners`

Most of these functions in our Ethereum-Easy have eight steps required to properly build and send out an Ethereum object that can operate on the blockchain network. The Ethereum-Easy library abstracts away all of these steps and their ordering, decreasing the complexity of the object building for the user.

Table 6.1: Code Volume

	EE	W3
sendTx	1	25
deployContract	1	88
contractCall	1	42
contractSend	1	77
listen	1	140

Next, we examined the lines of code saved when using Ethereum-Easy. In order to do this, we implement each function of the Ethereum-Easy library separately with the Web3.js library. In Table 6.1 we show the lines of code for the Ethereum-Easy (EE) functions versus the Web3.js (W3) implementations we created for each function. In Table 6.1 we notice that the functions dealing with smart contracts (deployContract, contractCall, contractSend) have more code inflation than the sendTx function. This is due to the necessity of compiling smart contract code before sending it to the blockchain. We also notice listen has the highest code inflation. This is due to the amount of parsing needed to examine an entire Ethereum block. It demonstrates that the listen function is a complex feature under the hood.

Table 6.2: RPS Code Volume

	EE	W3
Lines of code	521	868

Finally, we examined the code required to implement RPS with Ethereum-Easy compared to Web3. To do this, we added a module to RPS that facilitated all of the Ethereum-Easy functions. This module itself was 347 lines of code. Table 6.2 shows the total lines code for each implementation. With Ethereum-Easy we reduce our code volume by 40% in RPS.

We see two patterns emerge from analyzing the Ethereum-Easy library. First, we see a reduction in code volume when using the Ethereum-Easy library. This is true for the Ethereum-Easy functions themselves and implementing an application on the Ethereum blockchain. The second pattern is an abstraction of the complexities of working with Web3.js when using Ethereum-Easy. Reducing code volume and abstracting complexities demonstrates how Ethereum-Easy reduces barriers for developers in working with the Ethereum blockchain.

Speed of Execution

Another important metric to consider is the run time of the Ethereum Easy functions compared to other methods of execution. To understand this metric, we compared the run time of two of our stateful functions on three different methods of execution: Ethereum-Easy, Web3, and the Remix online smart contract integrated development environment (IDE). Remix is a web application that allows a user to deploy smart contract to the Ethereum blockchain. It also provides a web interface for calling functions on deployed smart contracts. While it is a common tool in the

Ethereum developer's tool belt, it does not provide a library interface for working with smart contracts, like what we provide with Ethereum-Easy.

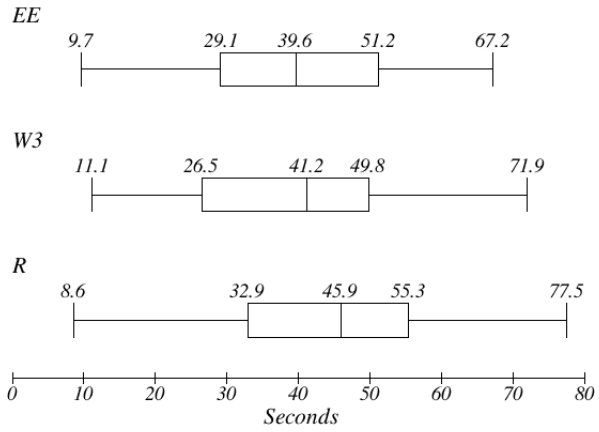

Figure 6.1: This box and whisker plot shows our deployContract run time data. We used three platforms to measure for run time: Etheruem-Easy (EE), Web3.js (W3), and the Remix online IDE (R). On the x-axis we have the sends to successful run time execution. The whiskers of our plot represent the minimum and maximum values recorded for each execution method.

To compare the three tools listed, we ran the deployContract and contractSend functions in the Ethereum-Easy library and recorded the run time to successfully complete each operation. We did this 25 times for both functions. We then tested the run time of both functions on Web3.js and the Remix editor's implementation of the same operations. We did not test the sendTx and listen functions because these do not have similar implementations in Remix. We also did not test contractCall because this

is a stateless operation in which the run time is only bound by network latency. The contractCall function is near instantanious compared to the other functions because it reads a static variable on the blockchain and returns this value.

Figure 6.1 shows our run time data gathered for the deployContract function. There is a variation of almost 6 seconds in the average run time for each execution method, with Ethereum-Easy being fastest and Remix being slowest. With each execution method we can see very similar minimum and maximum run time values. We do notice that the Remix execution method tends to be slower on almost all of the quartiles of the plot. We speculate that this is due to the application interface between the user and the blockchain in this online IDE. Where in both Ethereum-Easy and Web3.js, we are displaying data as soon as we receive the function confirmation message from the blockchain miner.

Figure 6.2 displays our run time data for the contractSend function. The results for this measurement are even more similar than our deployContract run times. Again, we see the Ethereum-Easy execution method being faster on average, with Web3.js only slightly behind Ethereum-Easy, and Remix only a few seconds behind that. The ranges of our data are very similar as well, with only 1 second of variation in the lower bound of our plot and about 3 seconds of variation in the upper bound.

Ethereum-Easy library is very comparable in run time with other current methods of function execution in the Ethereum space. This is especially true for Ethereum-Easy and Web3.js. While Ethereum-Easy is the fastest for both function calls in our measurements, we do not make the claim that our library is faster than other options. We believe running the tests for Ethereum-Easy and Web3.js more times would converge the data making the results look nearly identical.

Finally, we would like to note that these tests took place on the Ropsten Ethereum testnet. This is important to the findings because block times and system

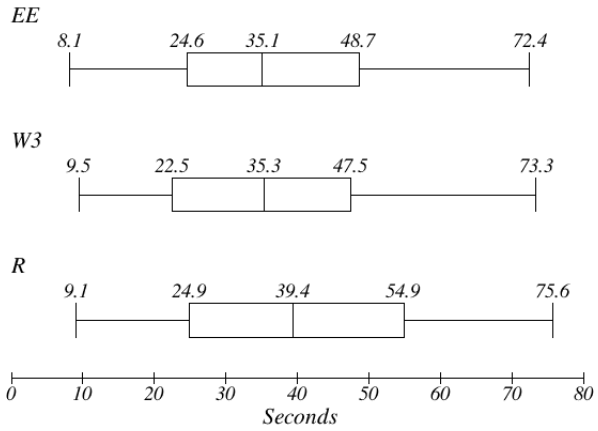

Figure 6.2: This box and whisker plot shows our conractSend run time data. Three methods were chosen for recording run time: Etheruem-Easy (EE), Web3.js (W3), and the Remix online IDE (R). On the x-axis we have the sends to successful run time execution. The whiskers of our plot represent the minimum and maximum values recorded for each execution method.

performance on testnets can be highly variable compared to the live Ethereum blockchain. Testnets do not have the same mining resources as Ethereum, which makes them act differently to the main chain in some cases. We believe that if we used the live Ethereum chain for these tests, run times would be more consistent with less range in the minimum and maximum values in our plots.

Our time measurements show us that there is not a large variation in the run times between the Ethereum-Easy operations compared to other implementations of these operations. We believe with more trials, all run time measurements would

converge to very similar values, demonstrating little difference in the run time of the function calls between each method of execution. This indicates that the reduction in code volume and complexity with the Ethereum-Easy library does not lead to increased overhead in function run time.

CONCLUSION

In this thesis, we identify the costs associated to incorporating blockchain solutions into business appellations. Currently, the required concepts for working with Ethereum are numerous and complex. The tooling around the ecosystem is sparse. Documentation is often inaccurate, outdated, or entirely missing. With these challenges in mind, we create a new library for Ethereum software developers. Ethereum-Easy abstracts away the hard concepts of Ethereum blockchain interaction. With Ethereum-Easy we built a demonstration to help educate users of this library. This demonstration, RPS, is a rock, paper, scissors game which uses the functions of Ethereum-Easy. We also built tools to aid in Ethereum software development. Jenk-Thereum is a CI/CD pipeline for testing and deploying Ethereum smart contracts. Using Jenkins, a common tool for CI/CD in software engineering, Jenk-Thereum is a product any software developer could pickup to aid in their smart contract development. The RPS game and CI/CD pipeline are well documented so any developer can learn to use them without the pitfalls of poorly documented software. Based on our analysis of Ethereum-Easy we found that the library decreases the volume of code needed for Ethereum applications while not increasing function response time. With Ethereum-Easy, RPS, and Jenk-Thereum we can reduce the cost of adoption in the Ethereum coding ecosystem and provide much needed industry standard coding tools for software developers.

REFERENCES

[1] U. Blog. The hottest freelance skills on upwork: Q3 2017. URL: https://www.upwork.com/blog/2017/11/freelance-skills-upwork-q3-2017/, Nov. 2017.

[2] V. Buterin et al. A next-generation smart contract and decentralized application platform. URL: ethereum.org, Jan. 2014.

[3] C. M. Cap. All cryptocurrencies, June 2018.

[4] B. Cohen. Bittorrent protocol 1.0. *URL: http://www. bittorrent. org.*, Jan. 2002.

[5] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, Sept. 1985.

[6] J. Garay, A. Kiayias, and N. Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 281–310. Springer, Apr. 2015.

[7] H. Garcia-Molina. Elections in a distributed computing system. *IEEE transactions on Computers*, Jan. 1982.

[8] S. G. Gohwong. The state of the art of top 20 cryptocurrencies. *Asian Administration & Management Review*, May 2018.

[9] V. Gramoli. From blockchain consensus back to byzantine consensus. *Future Generation Computer Systems*, Sept. 2017.

[10] I. Grigg. Eos - an introduction. URL: https://github.com/EOSIO/, July 2017.

[11] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, July 1978.

[12] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE transactions on computers*, (9):690–691, Sept. 1979.

[13] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, May 1998.

[14] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, July 1982.

[15] M. Maekawa. An algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems (TOCS)*, May 1985.

[16] R. Merkle. Secrecy, authentication, and public key systems. *Ph. D. Thesis, Stanford University*, Jan. 1979.

[17] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. academia.edu, Jan. 2008.

[18] D. Ongaro and J. K. Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, June 2014.

[19] J. Peterson. Augur white paper. *http://www.augur.net/whitepaper.pdf*, Jan. 2018.

[20] S. Popov. The tangle. URL: iota.org, Apr. 2016.

[21] Ripple. Ripple.com. URL: https://ripple.com/, Oct. 2018.

[22] D. Schwartz, N. Youngs, A. Britto, et al. The ripple protocol consensus algorithm. *Ripple Labs Inc White Paper*, 5, Sept. 2014.

[23] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, Oct. 2001.