



NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

BLOCKCHAIN SCALABILITY IN SOFTWARE DEFINED NETWORKS

by

Stephen J. Workman

March 2020

Thesis Advisor:
Second Reader:

Geoffrey G. Xie
Robert Beverly

Approved for public release. Distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			<i>Form Approved OMB No. 0704-0188</i>	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE March 2020	3. REPORT TYPE AND DATES COVERED Master's thesis	
4. TITLE AND SUBTITLE BLOCKCHAIN SCALABILITY IN SOFTWARE DEFINED NETWORKS			5. FUNDING NUMBERS	
6. AUTHOR(S) Stephen J. Workman				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			10. SPONSORING / MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.			12b. DISTRIBUTION CODE A	
13. ABSTRACT (maximum 200 words) Recent optimizations in blockchain technology and increased utilization of software defined networking (SDN) have led to new opportunities in data center network management. The research problem presented in this study is whether a blockchain protocol can be efficiently used in east/west communications between geographically separated SDN domains. To evaluate blockchain geographic scalability in SDNs, experiments were performed utilizing latency measurements on a GENI testbed coupled with a blockchain transaction discrete-event simulator modeling an open-source permissioned blockchain system. Results reveal that additional network latency imposed by increased geographic separation between SDNs does not introduce a significant detriment to blockchain-enabled SDN communication because the current blockchain systems typically incur relatively high processing delays. However, the relative newness and poor quality of available permissioned blockchain software impose significant challenges in terms of ease of adoption and deployment. Experience gained in this research suggests that such software likely needs to mature more through real-world deployment and testing in order to support further efforts of integrating blockchain and SDN.				
14. SUBJECT TERMS SDN, software defined networking, blockchain, data center			15. NUMBER OF PAGES 101	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UU	

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

BLOCKCHAIN SCALABILITY IN SOFTWARE DEFINED NETWORKS

Stephen J. Workman
Lieutenant Commander, United States Navy
BS, University of Florida, 2009

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

**NAVAL POSTGRADUATE SCHOOL
March 2020**

Approved by: Geoffrey G. Xie
Advisor

Robert Beverly
Second Reader

Peter J. Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Recent optimizations in blockchain technology and increased utilization of software defined networking (SDN) have led to new opportunities in data center network management. The research problem presented in this study is whether a blockchain protocol can be efficiently used in east/west communications between geographically separated SDN domains. To evaluate blockchain geographic scalability in SDNs, experiments were performed utilizing latency measurements on a GENI testbed coupled with a blockchain transaction discrete-event simulator modeling an open-source permissioned blockchain system. Results reveal that additional network latency imposed by increased geographic separation between SDNs does not introduce a significant detriment to blockchain-enabled SDN communication because the current blockchain systems typically incur relatively high processing delays. However, the relative newness and poor quality of available permissioned blockchain software impose significant challenges in terms of ease of adoption and deployment. Experience gained in this research suggests that such software likely needs to mature more through real-world deployment and testing in order to support further efforts of integrating blockchain and SDN.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

1	Introduction	1
1.1	Problem Statement	3
1.2	Research Questions	4
1.3	Major Findings	4
1.4	Thesis Organization	4
2	Background	7
2.1	Software Defined Networks	7
2.2	Global Environment for Network Innovations	16
2.3	Related Work	18
3	Design	21
3.1	Experiment Design	21
3.2	Experiment 1: Co-located Blockchain Components	28
3.3	Experiment 2: Two Co-located SDNs	29
3.4	Experiment 3: Factor of Geographic Distance	30
3.5	Evaluation Process	31
4	Implementation and Analysis	33
4.1	Configure SDN	33
4.2	Experiment 1 Analysis	37
4.3	Experiment 2 Analysis	39
4.4	Experiment 3 Analysis	40
4.5	Further Experimentation with Heavy-Tailed Distribution of Network Latency	43
4.6	Further Experimentation with Performance Optimizations	44
4.7	Analysis Summary	45

5	Conclusions and Future Work	47
5.1	Conclusions	47
5.2	Future Work	48
	Appendix: Configuration Files, Scripts, Code	51
A.1	GENI RSPEC Example	51
A.2	GENI Experiment SDN Instance	58
A.3	ONOS Install and Configuration	62
A.4	OVS Switch Configuration	65
A.5	Hyperledger Fabric Installation	66
A.6	Hyperledger Fabric Discrete Event Simulator	69
A.7	Blockchain Analysis	72
	List of References	79
	Initial Distribution List	83

List of Figures

Figure 2.1	Example ONOS GUI.	8
Figure 2.2	Example Chaincode and Related Smart Contracts.	9
Figure 2.3	Hyperledger Fabric Relationships.	13
Figure 2.4	GENI Resource Map.	16
Figure 2.5	GENI Project and Slices	18
Figure 3.1	SDN Instance.	22
Figure 3.2	Two SDNs Separated By Predetermined Geographical Distance. .	23
Figure 3.3	Blockchain Timing Diagram.	25
Figure 4.1	Experiment 1 Transaction Throughput.	37
Figure 4.2	Experiment 1 Transaction Processing Times.	38
Figure 4.3	Experiment 2 Transaction Throughput.	39
Figure 4.4	Experiment 2 Transaction Processing Times.	40
Figure 4.5	Experiment 3 Transaction Throughput Cross-State.	40
Figure 4.6	Experiment 3 Transaction Processing Times Cross-State.	41
Figure 4.7	Experiment 3 Transaction Throughput Cross-Country.	42
Figure 4.8	Experiment 3 Transaction Processing Times Cross-Country. . . .	42
Figure 4.9	Comparison of Normal and Heavy-Tailed Distributions	43
Figure 4.10	Heavy-tailed Distribution Latency Results	44
Figure 4.11	Validation And Ledger Commit Optimization	44
Figure 4.12	Combined Experiments 1-3	45

Figure A.1 GENI RSPEC Visualization. 51

List of Tables

Table 3.1	Experiment 1 Variables	29
Table 3.2	Experiment 2 Variables	30
Table 3.3	Experiment 3 Variables	31

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

AM	Aggregate Manager
API	Application Programming Interface
AS	Autonomous Systems
BASH	Bourne-Again SHell
BGP	Border Gateway Protocol
CLI	Command Line Interface
CURL	Client URL Request Library
DES	Discrete Event Simulator
DoD	Department of Defense
DRM	Domain Relationships Management
GENI	Global Environment for Network Innovations
GUI	Graphical User Interface
HCM	Heterogeneous Controller Management
IP	Internet Protocol
LTS	Long Term Support
MSP	Membership Service Provider
NPS	Naval Postgraduate School
ONOS	Open Network Operating System
OVS	Open vSwitch

REST REpresentational State Transfer

SDN Software Defined Network

SFP SDN Federation Protocol

TPS Transactions Per Second

TCP Transmission Control Protocol

URL Uniform Resource Locator

USN U.S. Navy

VM Virtual Machine

Acknowledgments

I would like to thank Dr. Geoff Xie and Dr. Robert Beverly. Without their guidance and patience over the last two years this body of work would not have been possible.

I would also like to thank my wife, De'Anna, for her continued and unwavering support in all things.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:

Introduction

A computer network acts as a fabric by which computers and computing devices communicate. The largest of such networks is the internet, an ever-growing global infrastructure that encompasses an untold number of smaller networks with billions of hosts that communicate via the internet protocol (IP) while encapsulating data in a sequence of standard messaging units called packets. The smaller networks connected to the internet are managed by businesses, governments, academic institutions, or even individuals. How these individual networks are managed is a decision the administrators must make, though how they communicate between each other over the internet is governed by the border gateway protocol (BGP).

Two key aspects of computer networking are the concepts of routing and forwarding. Routing dictates the path of IP routers by which packets will traverse in order to reach their destination. Forwarding is a per-device action by a router or switch to move a packet from an inbound link interface to an outbound link interface [1]. Routing takes place in the control plane while forwarding takes place in the data plane. In other words, the control plane layer is the network-wide logic that decides, through coordination among routers, how specific packet flows carrying user data are best routed from the source network to the destination network. The data plane is the per-device function that determines how data arriving at a device input is forwarded to that same device's output [1]. There are two approaches to routing worth discussing that directly pertain to the focus of this study and are broadly grouped under the categorization of “per-router control” versus “logically centralized control” [1]. Per-router control is when a part of the routing logic is executed on every router and forwarding decisions are eventually made locally in each router. A logically centralized control design is when a logically centralized decision element, commonly referred to as a controller, determines packet forwarding paths on behalf of the routers [1]. That is, routers under this routing paradigm no longer execute any routing logic. This separation of control and data planes marks a key difference between traditional router-centric routing protocols and software defined networking (SDN).

Traditional network management, which often requires administrators to perform meticulous per-router configuration changes to ensure desired IP packet routing, is often viewed as challenging, error-prone, and tedious to update and manage [2]. SDNs, or “programmable networks” as they are sometimes referred, aim to greatly simplify network management for the administrator while enhancing network flexibility and providing for increased customization. In an SDN, the forwarding hardware is physically separated from the hardware hosting the controller. The intelligence of the network resides in the SDN controller and directs network traffic from a logically centralized decision point. One of the most significant benefits is that network behavior can be directed from the application level to the controllers, which in turn define routing policy for the forwarding devices. Documented use cases for SDN include network slicing in cloud data centers [3], traffic engineering in Google’s wide area network [4], and potential deployment at military tactical networks [5], [6].

The benefits of decoupling the control and data planes can be illustrated by a brief description of the three different SDN controller interfaces. These interfaces are commonly referred to as northbound, southbound, and east/west. The northbound interface allows for communication between the controller and a variety of high level network control applications such as those implementing network-wide routing policy while the southbound interface provides a vendor agnostic means of communication between the controller and the lower level network switches [2]. The focus of this study is on the east/west interface which enables controller-to-controller communication between multiple SDN networks and thereby allows inter-domain routing. SDN inter-domain routing is still a research concept and no formal standard of communication exists yet today. BGP is the internet standard for inter-domain routing. However, BGP suffers from numerous deficiencies in the SDN context. Specifically, according to Le et al., BGP suffers from scalability, stability, and efficiency issues that stem from a single dimension information model (i.e., the IP destination) and full instantiation information exchange protocol (i.e., network decisions must be completely instantiated as data) [7]. The destination-based approach to routing which BGP relies on does not afford the fine grain flow level control for which SDNs require. In view of the BGP challenges and citing a need to establish an SDN east/west interface standard, Le et al. proposed the SDN Federation Protocol (SFP) [7]. The goal of SFP is to support multi-dimension routing (i.e., IP addresses, port information, and protocols),

enable intelligent resource sharing, support autonomy, and enforce privacy [7]. In recent work, Tollefson proposed an east/west protocol utilizing blockchain to satisfy some of the SFP requirements listed above [8].

A blockchain is a shared and distributed ledger (i.e., record keeping) technology that records transactions in the form of a series of blocks. The chain of hashes guarantees the immutability of the ledger and preserves the order of transactions. Broadly, there are two types of blockchains: *permissionless* and *permissioned*. Permissionless blockchain models are founded on the idea that the nodes in the network do not trust each other and so every node in the network maintains its own identical copy of the ledger and is responsible for checking the validity of transactions on its own [9]. In a permissioned blockchain model, a concept of trust exists in that nodes may participate only if they have been authorized to be a part of the network [10]. Each block in the ledger can be viewed as a series of transactions and includes a hash of the previous block. One such benefit of blockchain use in SDN networks is the immutable nature of the ledger which would provide guaranteed record keeping of all transactions between separately managed SDNs. Additionally, the permissioned nature of Hyperledger Fabric enforces privacy and is a required feature of SFP [7]. These qualities and others will be discussed in greater detail in Chapter 2.

1.1 Problem Statement

Data centers continue to grow in size and complexity as user demand continues to grow at increasing rates [2]. The explosion of cloud-based services, on-demand computing power, and dynamic scaling requirements illustrates the need for data centers to share resources at the granularity of virtual machines (VMs). Network management of this scale requires careful consideration of many factors. Software defined networks have showed promise in recent years in aiding in the management of data center-sized networks. However, there currently exists no standardized method for network communication between separately managed SDNs. This research aims to apply a proof-of-concept model using latency measurements taken from real-world hardware for use in a custom blockchain transaction simulation tool. We will use commercially supported networking components and protocols whenever possible while specifically investigating the scalability aspects of blockchain operations in a blockchain-enabled SDN. It will build upon Tollefson's recent work that demonstrated a proof-of-concept that utilized blockchain in an SDN east/west

protocol [8], while focusing on aspects of blockchain scalability under different network latency conditions.

1.2 Research Questions

Tollefson's work focused solely on the viability of utilizing blockchain in an SDN east/west interface. This was performed in a simulated/emulated environment [8]. The focus of this research is to take this proof-of-concept and perform analysis with a combination of real-world hardware and simulation tools. This research seeks to answer the following questions:

1. Can a suite of tools and software scripts be developed to rapidly experiment with blockchain-enabled SDN related research?
2. Is the current state of Hyperledger Fabric a viable blockchain solution for implementing a blockchain-enabled SDN?
3. How does blockchain performance (in terms of end-to-end latency) scale as SDN sites become geographically dispersed resulting in different network latency conditions?

1.3 Major Findings

This study concluded with the following major findings:

1. Although network latency impacts the end-to-end individual transaction processing time in a blockchain-enabled SDN, delays associated with blockchain component processing have significantly more impact on transaction throughput.
2. Distance between SDNs may not be a critical cause for concern in determining blockchain-enabled SDN feasibility.
3. Permissioned blockchain software likely needs to mature more through real-world deployment and testing in order to support further efforts of integrating blockchain and SDN.

1.4 Thesis Organization

This thesis is organized into five chapters. Chapter 1 provides an introduction to networking concepts and technologies relevant to this research while providing justification of the need

for exploring new approaches to network management. Chapter 2 provides background information on tools and technologies used in this research while additionally reviewing previous work that makes this research possible. Chapter 3 covers the experiment designs used to answer the research questions above. Chapter 4 details implementation specifics and analysis while Chapter 5 provides details on conclusions and future work. An appendix can also be found following Chapter 5 that provides all original source code used throughout this research.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2: Background

The goal of this chapter is to provide a more detailed description of relevant technologies and software used in this thesis. We will look more closely at specific software defined networking (SDN) features relevant to this research, the concept of blockchain-based SDN coordination, and the Global Environment for Network Innovations (GENI) infrastructure. The blockchain discussion will focus on the Hyperledger technology, and the associated system scaling challenges. We will conclude with a review of prior work that has motivated this study.

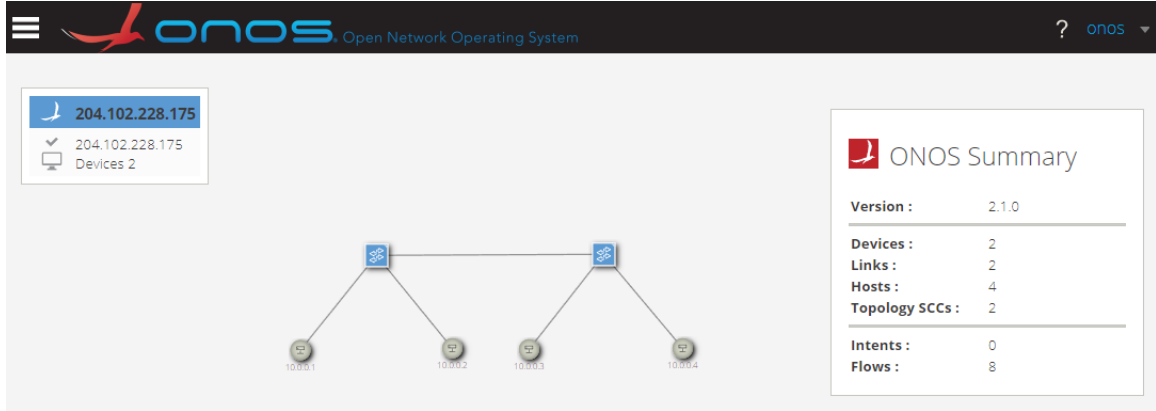
2.1 Software Defined Networks

Software defined networks (SDNs) provide incredibly flexible software solutions to complex networking challenges. Chapter 1 introduced many benefits SDN provides. Of these, programmability is the main factor motivating this research. We will analyze blockchain components allowing resource sharing between independently managed SDN networks, and furthermore, we will perform a series of experiments involving multiple geographically distributed SDN test-bed networks to evaluate the scalability of the blockchain system. Critical to any SDN is the selection of SDN controller and switch solutions to be used as building blocks for the network. The Open Network Operating System (ONOS) and Open vSwitch (OVS) software packages were chosen as the controller and switch solutions, respectively, because they are open-source and they have a relatively stable code base and large user community.

2.1.1 ONOS and OVS

ONOS contains multiple qualities that led to its selection as the software controller of choice. ONOS is production quality, open source, has an active developer community, and advertises scalability, high performance, and resiliency [11], [12]. Most modern Linux 64-bit operating systems can be made into an ONOS controller with relative ease. For our experiment we used 64-bit Ubuntu 18.04 LTS (long term support). This ease of setting up an ONOS controller on Linux motivated us to construct a testbed infrastructure consisting

Figure 2.1. Example ONOS GUI



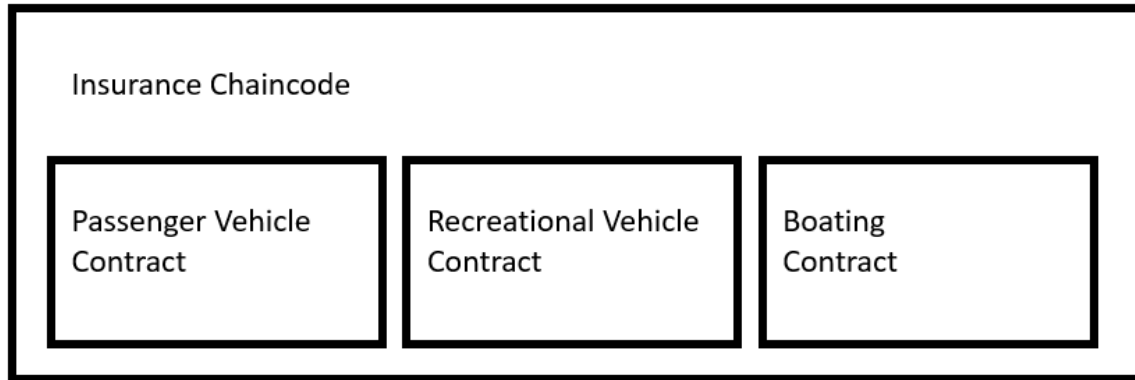
Example ONOS GUI displaying simple SDN composed of two switches and four hosts. Source: [12].

of Linux virtual machines (VMs) running at different geographic locations. An example installation and configuration script for ONOS can be found in Appendix A.3. In addition to a full-featured command line interface (CLI), ONOS also provides a graphical user interface (GUI) which can be accessed by entering the following Uniform Resource Locator (URL) into a web browser: [http://\[host IP address\]:8181/onos/ui/index.html](http://[host IP address]:8181/onos/ui/index.html). Figure 2.1 provides an example of what the ONOS GUI looks like with two switches and four hosts associated. The ONOS northbound interface supports REST (REpresentational State Transfer), which is an architectural style for distributed systems that simplifies the deployment and configuration of network devices and will be discussed in further detail in Chapter 4 [11]. ONOS supports all major versions of OpenFlow, the only standard SDN southbound interface protocol [11]. OpenFlow is open source and continues to be a popular choice for SDNs [13]. Therefore, we used OpenFlow in this work. Our switch solution of choice for this research, OVS, supports all major versions of OpenFlow and runs on Linux without need for any modification.

2.1.2 Blockchain-based SDN Coordination

Central to this research is the exploration of a more secure means of SDN-to-SDN communication that preserves the SFP ideas proposed by Le et al. [7]. The properties inherent to blockchain technologies and features specific to Hyperledger Fabric fill these requirements [14]. Previous work demonstrated the viability of utilizing a blockchain to establish an east/west interface between separately managed SDNs in a simulated environment [8].

Figure 2.2. Example Chaincode and Related Smart Contracts



Chaincode can be thought of as a container of smart contracts available to participants. In this example the passenger vehicle smart contract would be executed when a client requests a new insurance policy for their vehicle. Source: [14].

This research will further this idea and explore scalability aspects of the blockchain-enabled SDN on real-world hardware with analysis focused on blockchain components.

Hyperledger Fabric Blockchain

Hyperledger Fabric is a blockchain variant under the larger Hyperledger framework maintained by the Linux Foundation. Other variants include Besu, Burrow, Indyu, Iroha, and Sawtooth [15]. The Hyperledger blockchains listed above are designed with certain goals in mind. For example, Besu is a Java-based Ethereum (crypto-currency) client. Hyperledger Fabric differs itself from other blockchains by enforcing network participation privacy [16]. In the Fabric network, instead of allowing unknown entities to join and participate the members must first be validated through a Membership Service Provider (MSP) [16]. Hyperledger Fabric also includes a new feature which allows for the creation of channels. These channels allow for private and scalable communications between participants of the same channel. Every channel has a unique ledger and only participants of the channel are able to write, store, and read that ledger.

Hyperledger Fabric supports smart contracts and chaincode. Smart contracts are pieces of code stored in the blockchain that execute when specific conditions are met and multiple smart contracts can be defined within a single chaincode [17]. Chaincode can be thought of as a container of smart contracts that are available to blockchain participants. An

example high level design of chaincode for an insurance application can be found in Figure 2.2. Hyperledger Fabric chaincode can be written in both Go and NodeJS programming frameworks [16]. Smart contracts interact with two different parts of the ledger. These two components are the blockchain, which is the immutable history of all transactions on the network and the world state, which is the cache of current state values. Smart contracts can get, put, and delete states in the world state while also performing queries on the ledger to retrieve information on previous transactions. The “get” function is used to query the current state of an object. A “put” function is used to either modify an existing object in the ledger world state or to create an entirely new one. Lastly, a “delete” function is used to remove an object from the ledger while still retaining the history. It is important to observe that regardless of the actions performed on the world state, the ledger shall always retain an immutable record of all of these changes to the world state [9].

Another key characteristic of blockchains is the order in which records are chained together to form the blockchain. Transaction ordering is the result of the consensus algorithm used within the blockchain. Hyperledger Fabric currently supports three different consensus algorithms which includes Solo, Kafka, and Raft [16]. Solo is simplistic in design and only has a single “order” node. Kafka and Raft both provide enhanced crash fault tolerance, or CFT, but have a much higher administrative overhead and are complex to manage [16]. When performing experimentation with proof-of-concept networks, the Linux Foundation recommends using Solo. This is because the Solo ordering service processes transactions in exactly the same manner as Raft and Kafka, but the researcher will not have the additional burden of managing and upgrading multiple clusters and nodes [16].

The Building Blocks of Hyperledger Fabric: A Test Network

The above section describes the higher level functionality of the Hyperledger Fabric blockchain technology. In this section we will work through a test network highlighting the features described above while focusing on the major software installation and configuration steps. This research models the Hyperledger Fabric blockchain network in a simulation tool, but we will detail the steps necessary to install and run an actual blockchain test network to both provide insight into the transaction process and additionally aid future research as the installation process is not trivial. To do so, we’ll use the basic network configuration provided by the Hyperledger Fabric development team [9]. The following

assumes a standard 64-bit Ubuntu 18.04 LTS operating system. The first challenge in setting up a Hyperledger Fabric blockchain network involves installing all of the required software dependencies needed for the Fabric to function. The first such piece of required software is cURL(Client URL Request Library). Simply stated, cURL is a CLI tool that allows for sending or receiving files using standard URL syntax [18]. Next, docker needs to be installed. Docker represents a set of services and products that use virtualization to manage software in independent packages called containers [19]. The use of containers allows the software to be isolated from one another and to keep their own software libraries and configuration files protected from each other. In this example, different nodes of the blockchain network will operate in different docker containers. Lastly, because the test chaincode is written in Go, we will need to ensure that Go is installed and configured. Go, or Golang, as it is sometimes referred is an open source language that advertises conciseness, efficiency, and flexibility while tailoring itself well to multi-core and networked machines [20]. A script was written for this thesis that performs the installation and configuration of all required software dependencies and can be found in Appendix A.5.

Now that all required software is installed, the building blocks necessary to create a simple Hyperledger Fabric blockchain network are available. The following steps are important as they verify the correct versions and configurations of previously installed software. The first step involves installing the latest stable Hyperledger Fabric binaries and samples. This can be done by executing the following:

```
curl -sSL http://bit.ly/2ysb0FE | bash -s
```

The above command uses a script developed by the Hyperledger Farbric team to download and install the latest version of Fabric. This is a good test of cURL. An error here likely indicates that the version of cURL installed is too old and no longer supported. The binaries that were downloaded using the above script were placed in a bin sub-directory of the current working directory. To enter the directory where the simple network components are installed, simply execute the following:

```
cd fabric-samples/first-network
```

The Hyperledger Fabric sample code contained in this folder is used to test all software components necessary to run a simple blockchain network. The number of software dependencies and complexity of interrelated components make this basic network very useful for the researcher. The first step in bringing up this network involves generating certificates and a genesis block for the channel and is done so by executing the following:

```
./byfn.sh generate -c mychannel
```

The above line will begin the process of building the blockchain network. Specifically it will generate all of the certificates and keys for the different nodes on the network while also configuring a channel for communication (above called “mychannel”). It also starts the ordering service, which accepts endorsed transactions and will then “order” them into a block. The default ordering service is “Solo”, the details of which were described in the above section. Once the above bootstrapping is complete, the network is ready to be brought up. This can be done by executing the following:

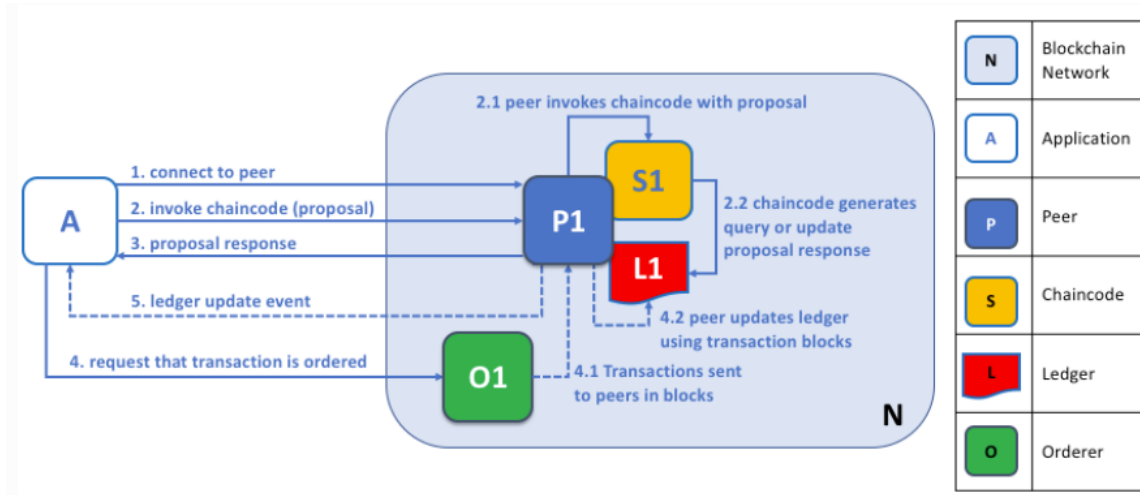
```
byfn.sh up -c mychannel -s couchdb
```

Here, the blockchain network is brought up on the “mychannel” channel and will use “couchdb” as the backend database. Numerous images will be executed in docker containers and transactions will be executed on the network. For this basic network, no further interaction on the part of the user is required. Eventually, the transactions will cease and the end of the scenario will be reached. Logs will flow across the screen detailing specific events. The goal here is to reach the end. Once the desirable “End” result is achieved (“End” is printed to the terminal screen), the only thing left to do is to bring down the network. This can be done by executing the following:

```
./byfn.sh down
```

This concludes the basic functionality test and explanations of all interrelated software components required to build and run a simple Hyperledger Fabric blockchain network.

Figure 2.3. Hyperledger Fabric Relationships



This graphic illustrates the relationships between different entities within the Hyperledger Fabric architecture. Source: [9].

For this study, Hyperledger Fabric blockchain software challenges will be discussed in greater detail in Chapter 3.

The Building Blocks of Hyperledger Fabric: Entity Relationships

The previous section described the required tools and actions necessary to install and launch a test blockchain network. Because much of this research is being performed on real-world hardware and not with pre-configured images supplied by vendors, this is an especially important step to perform as it helps verify numerous software package installations and configurations. Here, we aim to explain the relationship between the various entities that make up the above network. These relationships are common to any Hyperledger Fabric network, regardless of developer implementation. Figure 2.3 will be used to help demonstrate entity relationships and walk through the steps necessary to perform a transaction on the network.

Peer entities make up the majority of the Hyperledger Fabric blockchain network. Peers host smart contracts (chaincode) and all peers maintain a copy of the blockchain ledger. It is through peers that applications are able to access these smart contracts and ledgers. Figure 2.3 illustrates how an application interacts with the Hyperledger Fabric blockchain through a peer node. Additionally, channels are used to allow for a collection of applications,

peers, and orderers to communicate privately with one another in a Hyperledger Fabric blockchain network. Finally, blockchain networks are governed by a group of organizations and organizations own and contribute resources (like peers) to the overall network. These organizations both form the network while additionally providing the resources necessary to provide the blockchain functionality. The decentralized structure allows for the network to shrink or grow in size and as long as at least one organization remains then the network will continue to exist [21].

Now, we will detail the steps necessary to conduct a transaction on the network and what actions the different entities described above perform to complete the transaction.

1. Propose Transaction
2. Order Transaction Into a Block
3. Validate and Commit Transaction

At the beginning of Phase 1, the application submits a transaction proposal to a peer or set of peers (as defined) for endorsement. These peers respond to the application with signed transaction proposal responses. Phase 1 is complete once the originating application has received the predetermined number of signed responses. Phase 2 involves ordering and packaging of the proposed Phase 1 transaction(s). After receiving the signed responses from Phase 1, the application then sends a request to the orderer. The orderer is responsible for defining an order for all proposed transactions that it has received from one or more applications and placing them into blocks. Phase 3 involves the distribution of transactions to peers where the individual transactions will be validated by receiving peers and updates applied to the ledger at each peer. The validation policy at the peer is processed according to the endorsement policy of the smart contract that produced the specific transaction. Once validated, the transaction is written to the ledger and the transaction process is complete [21].

Blockchain Scaling Challenges

Le et al. proposed SFP in an effort to define requirements for an SDN east/west protocol that addressed efficiency, scalability, and stability concerns between separately managed SDNs [7]. By design, SDNs are highly programmable and allow for nearly limitless customization. This property allowed for the development of a proof-of-concept model that utilized a blockchain in an east/west interface [8]. We argue that the Hyperledger

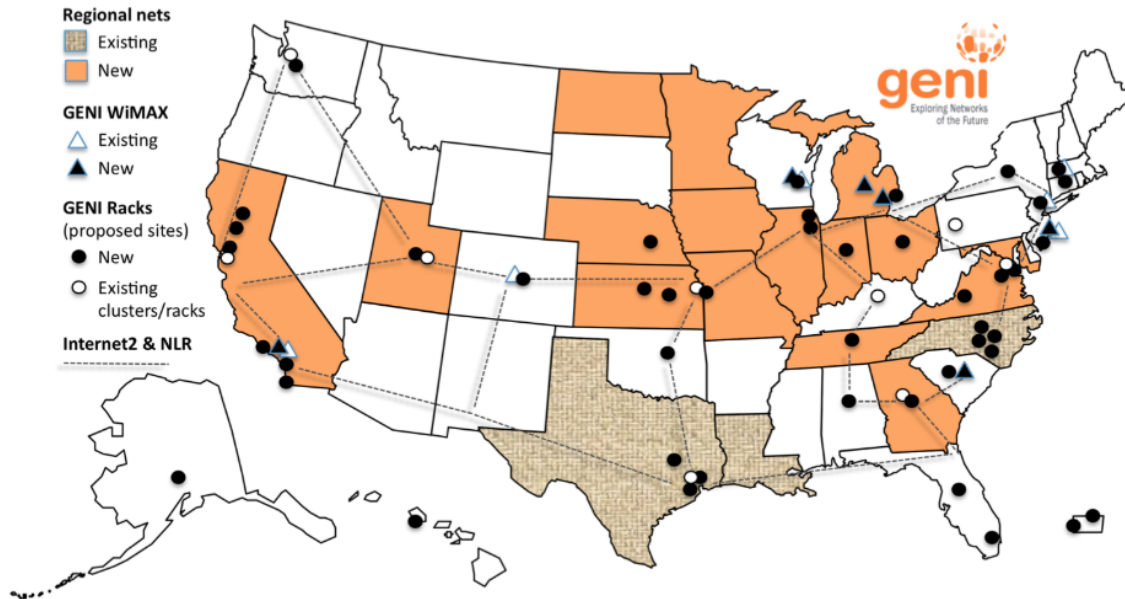
Fabric blockchain with advertised properties of confidentiality, resiliency, scalability, and flexibility preserve the requirements of SFP as stated above [9]. Therefore, the Hyperledger Fabric blockchain will be used for experimentation purposes with a focus on scalability aspects, particularly for the cases where the SDN networks are geographically dispersed.

As this research utilizes the Hyperledger Fabric blockchain, our focus on scaling challenges and performance issues will be focused on this specific blockchain technology. As previously discussed, Hyperledger Fabric is modular in design and supports numerous configurable properties. A study by Thakkar et al. attempted to identify Hyperledger Fabric performance bottlenecks and subsequently suggest performance optimizations [22]. In performing this analysis, Thakkar et al. considered five parameters which included block size, endorsement policy, channel use, resource allocation, and the type of ledger database. Results of this study will be summarized below.

In a study that conducted thousands of test cases, Thakkar et al. discovered a “saturation point” which they defined as a transaction arrival rate of 140 TPS (transactions per second). It was at this point that latency increased from 100s of milliseconds to 10s of seconds. What they additionally found was that when the transaction arrival rate was below the saturation point defined above, one should use a smaller block size. When the transaction rate is expected to be higher, one should use a larger block size [22]. The impact of the endorsement policy of choice also proved to be extremely impactful. To increase performance in this area, endorsement policies should have as few sub-policies and signatures as possible.

The use of channels helps to parallel operations. Thakkar et al. found that performance greatly benefited from allocating at least one vCPU per channel [22]. In terms of database selection, it was found that the GoLevelDB performed much better than CouchDB, but CouchDB provided better rich-query support. Therefore, if performance is the focus, GoLevelDB should be utilized [22]. Scalability can be measured in multiple ways and presents for interesting optimizations for Hyperledger Fabric. In terms of resource consumption and specifically CPU usage, the single largest factor is the complexity of the endorsement policy. If the endorsement policy only requires a few signatures from organization peers, then an organization composed of a large number of peers would not prohibitively impact the endorsement phase of the network [22]. Additionally, geographic dispersion of nodes may greatly impact scalability and will be investigated in this study.

Figure 2.4. GENI Resource Map



Resource map showing the distribution of GENI nodes across the United States.
Source: [23].

2.2 Global Environment for Network Innovations

GENI, or the Global Environment for Network Innovations, was chosen as the test bed architecture for this study. There were multiple real-world architectures considered for this research, but the flexibility and the ability to completely customize the network environment made GENI the most attractive option. GENI, which is supported by the National Science Foundation, focuses on distributed systems and network research and education [23]. Compute nodes, the resources utilized by virtual machine instances, can be found across the United States as illustrated in Figure 2.4. In terms of flexibility, GENI allows custom software and operating systems to be deployed across the compute nodes [23]. In terms of this research, this allows for rapid deployment of end hosts, switches, and controllers. Geographically, any combination of resources can be obtained across the United States. One can choose to work within one physical site or distribute a system across multiple states. GENI also provides measurement and instrumentation tools that can provide probes for passive and active measurements in addition to visualization and analysis tools [23]. For these reasons, GENI presents a very attractive option for this research.

2.2.1 GENI Concepts

GENI accounts are most commonly managed by participating organizations (sites which host GENI server racks). However, it is possible to register an account with the National Center for Supercomputer Applications (NCSA) if your organization or academic institution is not a participating organization. Regardless of your organizational status, an account request can be made through the GENI web portal (<https://portal.geni.net>). Once you have a GENI account, you are able to begin work. Within GENI, research is organized inside a “project.” A project exists at the top of the hierarchy and includes both experiments and the people conducting the experiments. A project may contain many researchers and researchers may be a part of many different projects. Because GENI is a shared testbed, there are likely many experiments being run simultaneously across the infrastructure. The concept of the “slice” makes this possible. A slice is a container for resources that are used within a specific experiment. A project may contain multiple slices. Resources, such as compute nodes, network links, etc., can all be added to a slice. A slice also acts as a layer of access control. The researcher that creates a slice manages access to the slice and thereby controls any experimentation on that slice [23].

A GENI aggregate assigns resources to slices for use in experiments. For example, the GENI rack at NPS provides resources and is an aggregate. Not all aggregates are equal and some aggregates provide different resources from others. For example, some aggregates only provide compute resources while others may also provide networking resources such as WiMAX base stations. A detailed listing of aggregate details can be found on the GENI website [23]. Once a researcher is part of a slice within a project, they are ready to request resources from an aggregate. An example project with corresponding slices can be found in Figure 2.5.

Researchers request resources from aggregates through the GENI Aggregate Manager (AM) Application Programming Interface (API). Aside from just requesting resources, this interface allows researchers to list all resources available at a physical site, find the status of resources allocated to their slice, and delete resources no longer needed from their slice [23]. In order to request resources, the API requires a formatted XML-style file called an RSPEC (resource specification) to describe the requested resources. The GENI website provides format details and examples for constructing specification files [23]. A complete example RSPEC with corresponding graphical representation can be found in Appendix A.1.

Figure 2.5. GENI Project and Slices

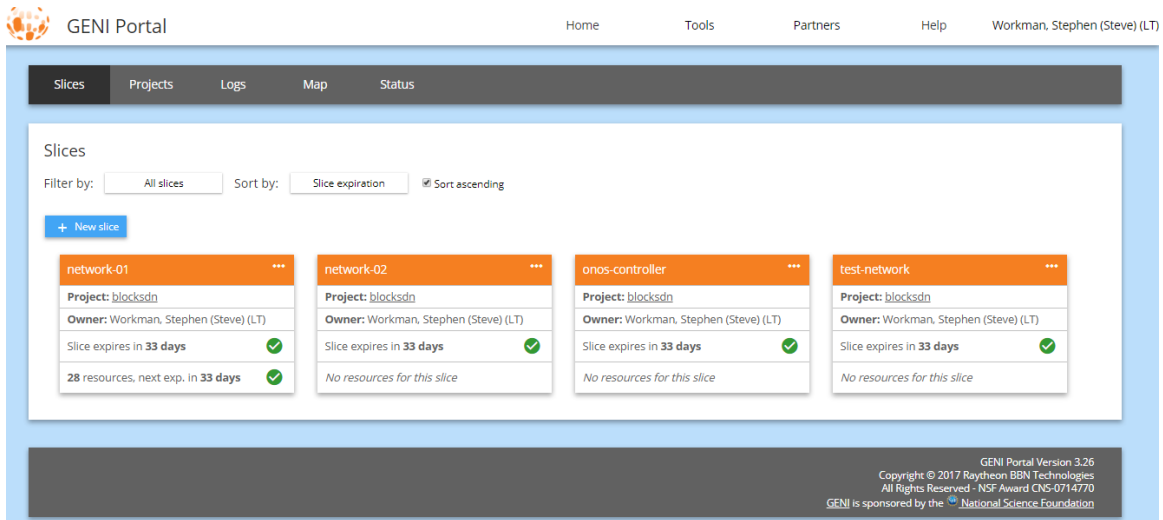


Image depicting the “blocksdn” project with assigned slices. A project may have multiple researchers and slices assigned. In this example, only network-01 has resources reserved. Source: [23].

2.3 Related Work

The research outlined in this thesis follows previous successes within the field of blockchains and software defined networks. Specifically, recent contributions have demonstrated the viability of utilizing blockchain technology to develop an east/west protocol for negotiating traffic between software controllers [8]. In Tollefson’s thesis, he successfully demonstrated establishing a connection between separately managed SDNs and data flows between the two networks using the Hyperledger Fabric blockchain [8]. The study was additionally able to show a linear relationship between the Hyperledger Fabric processing time and network latency, which suggests that this combination of technologies may scale and be viable for larger applications [8]. Because this previous work relied on emulated and simulated aspects, much work remains to be done to explore scalability on real-world hardware. This is the focus of this research and the specific details will be further explained in Chapter 3.

2.3.1 Other SDN East/West Approaches

Though SDN technology solutions have been quickly adopted in a variety of academic and commercial environments, the specific area of communication between separately managed controllers is still largely an area of academic study. Two such approaches to SDN-to-SDN

communication will be discussed below. Unfortunately, because the area of research is still relatively new and no standard has been set, the solutions proposed below have not been put through the same rigorous testing as other protocol standards.

Novel SDN-to-SDN Solution

The Novel SDN architecture proposed by Helebrandt and Kotuliak seeks to solve the SDN-to-SDN communication challenge by utilizing the SDN controller INT manager [24]. They envision an architecture organized into three layers to include:

- Controller Interconnection Session Control
- Capabilities Information Exchange
- Path Setup

Key to this organization is overall path determination and the ability to exchange network capabilities and properties between disparate controllers. The first layer, Controller Interconnection Session Control, is used to establish and maintain connection sessions between controllers. The next layer is responsible for exchanging network characteristics to include capabilities, potential limitations, and possibly path metrics. The final layer, path setup, uses network capabilities information to route traffic along a specific route [24]. Therefore, creating a path for a packet to follow involves establishing a connection between controllers, exchanging network capabilities between controllers, and finally the controller of the network in which the packet must traverse must determine the traffic route according to requirements as specified by contacting controller [24]. In their study, Helebrandt and Kotuliak demonstrated successful negotiation of path setup and flow installation which allowed for traffic to move across separately managed SDNs. However, this study was limited to an emulated network environment and scalability issues require additional research [24].

Zebra SDN-to-SDN Solution

Zebra is another approach to SDN-to-SDN inter-domain networking proposed by Yu et al. [25]. In Zebra, Yu et al. propose a four-layer architecture to include:

- Application
- Decision
- Dissemination and Discovery
- Data

The application layer is viewed as a customized demand layer where network applications, such as in-network storage and visualization tools operate. The decision layer governs all routing determinations between the interconnected domains. This layer represents the largest component of Zebra, and is where two necessary modules as described by Yu et al. reside. The HCM (Heterogeneous Controller Management) module is tasked with determining routing decisions inside a singular domain. The DRM (Domain Relationships Management) module is then tasked with determining routing information between separately managed SDNs [25]. The third layer, dissemination and discovery, provides the necessary communication links that connect decision and control information to routers and switches while additionally sending network metrics back to the decision layer. The fourth and final layer, data, is primarily charged with simple processing and forwarding operations [25]. Testing demonstrated a successful negotiation of route determination and flow installation. Further studies will need to be conducted to better analyze scalability aspects [25].

CHAPTER 3: Design

The goal of Chapter 3 is to introduce the experiment and analysis design. We will start with SDN and blockchain design specifics toward an integrated testbed that can be instantiated on GENI. Next, we will describe a set of experiments designed for evaluating the scalability of the blockchain component of the testbed. Finally, we will define metrics to be gathered from the experiments and the process that will be used to capture the data necessary for performance analysis and evaluation.

A significant amount of thesis efforts focused on fully integrating Hyperledger Fabric blockchain into SDN. However, we concluded that due to its poor quality and lack of documentation, the Hyperledger Fabric software is not ready for such integration. Therefore, the rest of thesis research has adopted a hybrid methodology by building a discrete event simulator of Hyperledger Fabric while leveraging real-world network latency measurements between SDN networks instantiated on GENI.

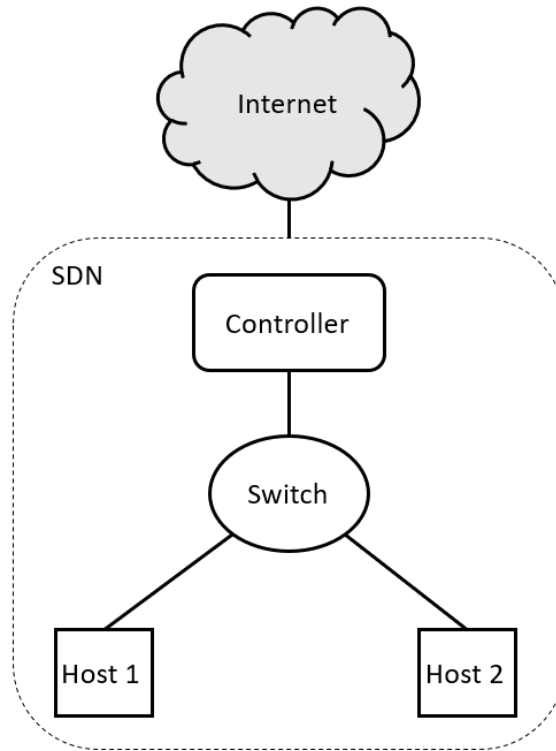
3.1 Experiment Design

Our experimental methodology follows a multi-step process. The steps are as follows:

1. Design and launch network topology with GENI.
2. Install and configure SDN components (controllers, switches) on GENI VMs.
3. Perform latency measurements on GENI network.
4. Use latency measurements from Step 3 and known blockchain component latencies to run the blockchain discrete event simulator (DES).
5. Conduct analysis of DES results.

The goal of this research is to test scalability in a blockchain-enabled SDN while also enabling future experimentation through the production of automation scripts to rapidly deploy a test environment. Ideally, we would construct clustered SDNs with full blockchain integration and perform whole-system testing. However, to do so would require significantly more time and resources than are available. As mentioned above, we will experiment using

Figure 3.1. SDN Instance



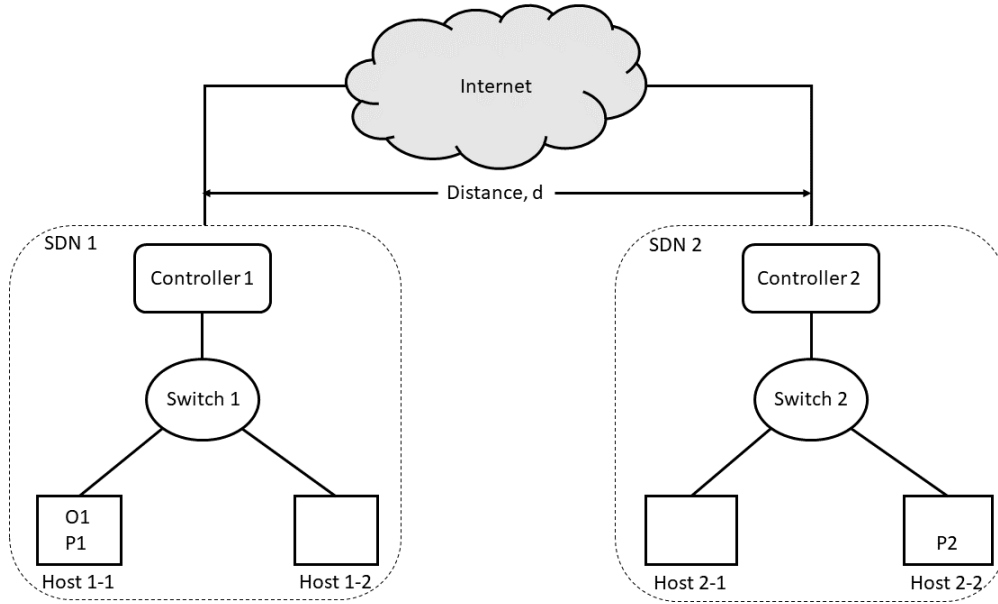
Example SDN instance used to represent a single SDN location.

the SDNs created in GENI and apply blockchain performance analysis using a discrete event simulator which will model the processing and communication latency incurred by a blockchain transaction. The tools necessary to perform testing for our research can be directly applied to future efforts so we will additionally detail methods for constructing SDN testing environments for future research. To start, we will need a configurable test-bed environment, a simple SDN topology that can be rapidly deployed, and an SDN controller and switch.

3.1.1 Software Defined Network Design

Our configurable test-bed environment will be GENI for reasons discussed in Chapter 2. GENI allows for efficient and rapid deployment of SDNs. Using an XML format, we can describe the topology best suited to our testing needs. Because the SDN is not the main contributing factor of overhead and the focus of this research is on blockchain-specific

Figure 3.2. Two SDNs Separated By Predetermined Geographical Distance.



Experiment 1 will assume all blockchain components are on Host 1-1 in SDN 1. Experiment 2 will use the design above and model two separately managed SDNs co-located at NPS where distance “d” is 0.0 miles. Experiment 3 will vary the distance “d” from 100s of miles (cross-state) to 1000s of miles (cross-country). For blockchain analysis purposes for Experiment 2 and 3, orderer(O) is on Host 1-1 and two peers (P) are distributed between Host 1-1 and Host 2-2.

components we can keep the SDN design simple. Additionally, a simple SDN design will allow us to rapidly launch and configure SDNs for different experiments. The design of the instance for use within our experiments can be seen in Figure 3.1. Here, a very simple SDN composed of one controller, one switch, and two hosts is depicted. The XML that describes this network for use within GENI can be found in Appendix A.2.

Once our network instance is launched and active within the GENI environment, we can begin installing the various software packages required to enable a network composed of a controller, switch, and hosts. These components are the basic building blocks of our simple SDN topology and will allow for data transmission within and between SDNs. The ONOS controller will provide the routing and control plane functions while the OVS switch will perform data plane forwarding functions for the hosts. GENI provides SSH login credentials to allow for secure and efficient access to the different VMs. Chapter 4 details

the exact steps necessary to install and configure all SDN-related components on these VMs. These steps lay the groundwork to begin experimentation. Before the individual experiments are defined, it is worth further describing blockchain-specific design details for these experiments.

3.1.2 Blockchain Design

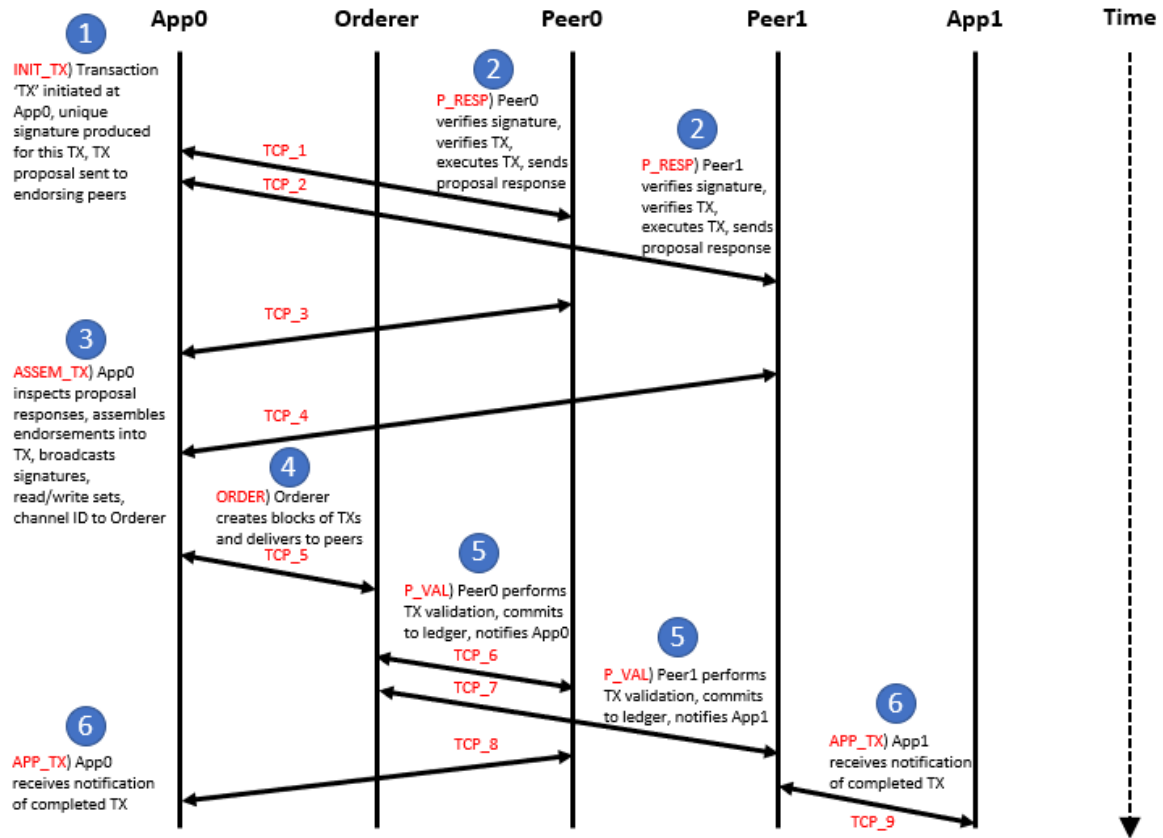
As discussed in Chapter 2, a blockchain network is composed of peer(s), orderer(s), and applications. The applications are used to communicate with peers and orderers. The peers contain the ledgers and smart contracts and orderers perform transaction ordering. Using Figure 3.2 as a reference, Host 1-1 will contain one peer and one orderer. Host 2-2 will contain one peer. These two peers and one orderer will be on the same channel so each peer will maintain a copy of the same ledger. When chaincode is invoked on a peer, the orderer will perform transaction ordering before updates are sent to peers on the network for validation and commit. In the envisioned blockchain-enabled SDN, the chaincode is responsible for negotiating network paths and requesting the installation of flow rules on network controllers. Ideally, there is a control application that provides flow updates to the controller for full integration. However, this is not necessary for the purposes of blockchain performance evaluation. To analyze blockchain performance, we will model blockchain transactions and use real world data to calculate network latency.

For example, if Host 1-1 on “Network A” wished to pass traffic to Host 2-2 on “Network B” an agreement between SDNs “Network A” and “Network B” will need to be established. Invoking chaincode on a peer on Host 1-1 will initiate the negotiation process. This transaction process looks like the following:

1. Application connects to peer and invokes chaincode with proposal.
2. Peers provide proposal response to application.
3. Application requests transaction to be ordered.
4. Orderer orders transaction into a block and distributes transaction to peers.
5. Peers perform validation and commit transaction to ledger.
6. Application is informed of event completion.

This summarizes the transaction process of the blockchain-enabled SDN and is the focus of this study. The diagram representing this transaction flow can be found in Figure 3.3.

Figure 3.3. Blockchain Timing Diagram



Example ledger update transaction flow within a Hyperledger Fabric environment assuming two peer endorsement policy.

3.1.3 Blockchain Transaction Analysis

Ideally, a Hyperledger Fabric blockchain-enabled network would be built and distributed across the SDN environment deployed through GENI. However, because of full blockchain integration challenges, we will instead perform a mathematical analysis of the transaction process and use real world measurements to build a discrete event simulator(DES). The DES is particularly useful for this analysis because it allows us to efficiently test the impact of different parameters on the blockchain system.

Using Figure 3.3 as the basis for this mathematical model, the latency of a single blockchain transaction in our network can be represented as the following (RED text in Figure 3.3 represents variables used below):

$$\text{INIT_TX} + \text{MAX}(\text{TCP_1} + \text{P_RESP} + \text{TCP_3}, \text{TCP_2} + \text{P_RESP} + \text{TCP_4}) + \text{ASSEM_TX} \\ + \text{ORDER} + \text{MAX}(\text{TCP_6} + \text{P_VAL} + \text{TCP_8}, \text{TCP_7} + \text{P_VAL} + \text{TCP_9}) + \text{APP_TX}$$

Further, many of the TCP variables can be reduced. TCP variables represent two types of latency: those internal to a host and those between the two hosts. Reducing TCP variables produces the following:

$$\text{TCP_1} = \text{TCP_3} = \text{TCP_5} = \text{TCP_6} = \text{TCP_8} = \text{TCP_9}$$

- Communication within the same host.
- These network delays can be assumed to be zero, as the network delay with a single host machine is negligible. Therefore, these TCP variables shall be removed.

$$\text{TCP_2} = \text{TCP_4} = \text{TCP_7}$$

- Communication between two hosts.
- This can be reduced to a single variable, TCP. These latency costs are associated with communication between Host 1-1 and Host 2-2.

Using the above TCP variable reductions, we can now update the single blockchain transaction formula to the following:

$$\text{INIT_TX} + \text{MAX}(\text{P_RESP}, 2 * \text{TCP} + \text{P_RESP}) + \text{ASSEM_TX} + \text{ORDER} + \text{MAX}(\text{P_VAL}, \\ \text{TCP} + \text{P_VAL}) + \text{APP_TX}$$

Additionally, for the MAX functions above the dominating expression will always be the expression with network latency cost (TCP), so the formula can be updated again to remove the factor without TCP cost:

$$\text{INIT_TX} + (2 * \text{TCP} + \text{P_RESP}) + \text{ASSEM_TX} + \text{ORDER} + (\text{TCP} + \text{P_VAL}) + \text{APP_TX}$$

Reordering this equation to preserve transaction order results in the following:

$$\text{INIT_TX} + \text{TCP} + \text{P_RESP} + \text{TCP} + \text{ASSEM_TX} + \text{ORDER} + \text{TCP} + \text{P_VAL} + \text{APP_TX}$$

It is import to clarify that this mathematical model is strictly for the two-peer Hyperledger Fabric blockchain design for this research. It assumes a two peer endorsement policy and zero network latency between blockchain components which reside on the same host. A

consequence of this formula reduction is the elimination of latency cost associated with P_RESP and P_VAL phases for Peer0. This follows the logic above that the transaction latency cost associated with Peer1 will always exceed that of Peer0. Additionally, the ASSEM_TX phase will always wait for responses from both endorsing peers before it can continue. Since the endorsements by Peer0 and Peer1 are conducted in parallel, the latency with Peer0 can simply be disregarded as the latency associated with Peer1 will always dominate as discussed above. The final formula can be adapted to a single machine configuration by simply removing the TCP variable. The first experiment will do exactly this. It will assume all components are co-located on a single host and provide a baseline for which further experiments will follow.

3.1.4 Linking Real-World Data and Blockchain Model

As discussed above, we are going to use real-world and research data to model transactions in a blockchain-enabled SDN. The SDNs built within the GENI environment will serve to provide network latency measurements. Blockchain component latency for a simple two peer network is sourced from a separate study focused on Hyperledger Fabric performance conducted by researchers at Duke University and IBM [26]. Previous research focused primarily on benchmarking Hyperledger Fabric or optimizing blockchain transaction operations to improve system throughput. This study focuses on SDN challenges and specifically the effect that network latency will have on blockchain transactions in an SDN-to-SDN environment where the geographic distance can vary. To perform an experiment, we will first conduct transfer time measurements between Host 1-1 (on SDN 1) and Host 2-2 (on SDN 2). We will use a Linux CLI tool called ‘iperf’ to measure the latency of sending a 512KB message between Host 1-1 and Host 2-2. The 512KB size is the preferred default message size in a Hyperledger Fabric network [9].

An iperf server shall be started on Host 2-2 by performing the following:

```
$ iperf -e -s
```

Then, from Host 1-1 a 512KB message shall be sent to Host 2-2 by performing the following:

```
$ iperf -e -n 512KB -c <Host 2-2 IP Address>
```

Network latency measurements shall be taken over multiple runs and used as the TCP cost between Host 1-1 and Host 2-2 for the blockchain discrete event simulator. Since we already possess real world blockchain component latency data from previous research and we now have network latency measurements we have all the data necessary to run the discrete event simulator and perform blockchain transaction analysis. Of note, it appears that due to how iperf is implemented it likely achieves slightly better performance than what an application would given the same environment [27]. However, because we are concerned with how the different geographic SDN distributions affect overall throughput when compared to each other a better performing iperf does not affect results. The DES is programmed in Python and generates simulation processes for each transaction and then runs them through a route that models the transaction mathematical model drawn from Figure 3.3. A key aspect of this model is the queues that form on each blockchain component during the simulation. By design, a processing component cannot process additional transactions until it is done with the current transaction. Therefore, queues may build in instances where transactions arrive at components faster than the components can process them. Additional details can be found in Appendix A.6.

3.2 Experiment 1: Co-located Blockchain Components

Experiment 1 will be used to validate a working blockchain DES model and determine appropriate blockchain transaction inter-arrival times. Table 3.1 shows initial transaction mean inter-arrival times to be tested in this experiment. Here, it is assumed that all blockchain components are co-located on Host 1-1 so an iperf latency measurement will not be performed. Desired metrics include transaction throughput (transactions per second) and the amount of time transactions spend in the blockchain network. This data will educate choices for transaction inter-arrival times for Experiment 2 and Experiment 3. Experiment 1 will also serve to define a theoretical maximum efficiency in terms of blockchain transaction throughput.

Table 3.1. Experiment 1 Variables

Experiment 1			
Experiment Number	Geographic Locations	Geographic Distance (miles)	Inter-SDN Connection Request Arrival Interval
1	co-located	0	1
2	co-located	0	2
3	co-located	0	3
4	co-located	0	4
5	co-located	0	5
6	co-located	0	10
7	co-located	0	15
8	co-located	0	20
9	co-located	0	50
10	co-located	0	100

Inter-SDN connection request is a Poisson mean inter-arrival time (milliseconds). For example, a connection request value of 5 indicates that on average a transaction will be initiated approximately every 5 milliseconds.

3.3 Experiment 2: Two Co-located SDNs

The previous experiment was performed to validate a working blockchain DES model and explore the effects of different transaction inter-arrival times. We will now model blockchain performance across network devices. The two SDNs will be co-located at NPS as depicted in Figure 3.2. The goal of this test is to model processing performance for two separately managed SDNs that are co-located. The variables for this experiment can be seen in Table 3.2. Here, we will maintain the same locations and distance between SDNs throughout the range of experiments. We will then vary the inter-SDN connection request inter-arrival time. The inter-SDN connection request inter-arrival time is the mean time between transaction requests. Transaction requests represent the negotiation between

separately managed SDNs. In this example, it would be between Host 1-1 on SDN 1 and Host 2-2 on SDN 2.

Experiment 1 provides baseline metrics to help determine appropriate transaction inter-arrival times for Experiments 2 and 3. The exact values for inter-arrival times for Experiments 2 and 3 will be detailed in Chapter 4.

Table 3.2. Experiment 2 Variables

Experiment 2			
Experiment Number	Geographic Locations	Geographic Distance (miles)	Inter-SDN Connection Request Arrival Interval
1	co-located	0	to be evaluated

Experiment 2 will examine the performance of geographically co-located SDNs. Poisson mean inter-interval times shall be determined after completion of Experiment 1.

3.4 Experiment 3: Factor of Geographic Distance

The goal of Experiment 3 is to incrementally test different blockchain scalability aspects by increasing the geographic distance between SDNs. First, we will move the location of SDN 2 from Experiment 2 hundreds of miles away in the same U.S. state. We will then incrementally adjust the inter-SDN connection request inter-arrival time until a saturation point occurs. This saturation point is the point at which the arrival rate of connection requests exceeds the rate at which these requests can be processed. This results in a connection request queue that grows without bound. Next, we will move SDN 2 across the country so there is communication between East and West coasts. We will then adjust the inter-SDN connection request inter-arrival time in the same fashion as above. A table of variables can be found in Table 3.3.

Table 3.3. Experiment 3 Variables

Experiment 3			
Experiment Number	Geographic Locations	Geographic Distance (miles)	Inter-SDN Connection Request Arrival Interval
1	cross-state	100s	to be evaluated
2	cross-country	1000s	to be evaluated

Experiment 3 will vary distance from 100s of miles between SDNs to 1000s of miles between SDNs. Poisson mean inter-interval times shall be determined after completion of Experiment 1.

3.5 Evaluation Process

The purpose of these experiments is to evaluate the scalability of a blockchain-enabled SDN. Therefore, we need to consider the effect that different parameters will have on certain performance characteristics. The metrics collected and analyzed here will focus on transaction throughput given geographic distance between blockchain components while varying transaction arrival rates. Throughput is the number of completed blockchain transactions per second. Given the complexity of blockchain transactions, this research will also examine the relationship between blockchain component processing delays and networking delays.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4:

Implementation and Analysis

In Chapter 4 we will discuss the experiment implementation specifics as well as provide analysis of the results upon experiment completion. We will begin by detailing the process necessary to bring up the participating SDNs and then review how we will model blockchain transactions on the network. Finally, we will analyze results for the different scenarios described in Chapter 3.

4.1 Configure SDN

Configuring the SDN and readying the network for experimentation requires three steps:

1. Define and configure network topology using GENI
2. Install and configure ONOS on specified controllers
3. Configure OVS software switches

Upon completion, the environment will be ready to for latency testing. A series of BASH scripts for use in Linux VMs will be used to automate the process. This process is the same for all experiments, so will only be detailed once. Individual experiment specifics shall be described further in their own sections.

4.1.1 Define and Configure Network Topology Using GENI

The first step in preparing the experimentation environment is launching and configuring the network. As discussed in Chapter 3, GENI is used to host the VMs needed to accomplish this task. After creating a slice for SDN 1, a topology can be described using the XML provided in Appendix A.2. For all experiments, NPS shall be used as the site location for SDN 1 (see Figure 3.2). Site 2 shall be chosen according to the experiment design description as defined in Chapter 3. Note that the XML used to define the above SDN instance for SDN 1 can be used for SDN 2 in all experiments. Now that SDN 1 and SDN 2 are allocated resources at the chosen sites, the software controller is ready to be installed. Note that the following steps for controller and switch configuration are required for eventual full blockchain integration into the network, but not for the purposes of the

blockchain simulator that we will use in our experiments. The following is provided to aid in further blockchain-enabled SDN experimentation.

4.1.2 Install and Configure ONOS on Specified Controllers

Each SDN instance contains one VM intended for use as the ONOS controller. The installation and configuration of the controller is simplified by use of a BASH script described in Appendix A.3. This same script is used for every instance of ONOS installation and configuration. To install ONOS using the BASH script, one needs only to log into the desired host, download the script, and run the script as root:

```
$ sudo ./onos_install.sh
```

The ONOS installation script installs all required software packages, modifies the firewall, configures various functions, and finally launches the ONOS controller upon completion. Once the install script finishes, the ONOS controller can be accessed via a command line or web interface. Access details can be found as notes at the beginning of the script.

Once the controller is up and running, OpenFlow needs to be enabled. Log into the ONOS CLI by executing the following on the controller host:

```
$ ssh -p 8101 karaf@localhost
```

Next, bring up the ONOS interface by typing:

```
$ onos
```

Then, enable OpenFlow by executing the following:

```
$ app activate org.onosproject.openflow
```

For testing purposes (for example, pinging Host 2 from Host 1), you may want to briefly allow automatic forwarding for hosts on the network by the switch. This will install flow rules as needed. This can be accomplished by executing the following:

```
$ app activate org.onosproject.fwd
```

To disable this feature, simply deactivate it:

```
$ app deactivate org.onosproject.fwd
```

At this point the ONOS controller is properly configured and the network is ready for switch configuration.

4.1.3 Configure OVS Software Switches

Fortunately, GENI specifications allow for OVS installation at the time of resource allocation, so no special installation script is needed. One only needs to configure the OVS switch. Appendix A.4 provides a BASH script for the configuration of the OVS switch. To configure the OVS switch, download the BASH script and run it as root on the host machine (verify controller IP address in config_switch.sh before running):

```
$ sudo ./config_switch.sh
```

The OVS configuration script creates a bridge for the various network interfaces on the VM and establishes a connection to the specified ONOS controller. Once the script is finished, it shall print to the screen a status update revealing connection details and bridge/interface information. At this point you can ping between hosts within each SDN to confirm a properly functioning network.

After the controllers and switches are configured for the two SDN networks (SDN 1 and SDN 2), a cluster shall be formed by linking the controllers of the two SDNs to each other.

Once clustered, the controllers will share network topologies. This is optional, but useful for the researcher because it enhances network visibility. This can be done from either controller and only requires the following to be executed from the standard Linux CLI (not the ONOS interface CLI):

```
$ sudo /opt/onos/bin/onos-form-cluster <IP_1> <IP_2>
```

where <IP_1> and <IP_2> are the external (internet facing) IP addresses of the two ONOS controllers. Once complete, the blockchain software is ready to be installed and configured on the host machines. The network is now ready for network latency measurements.

4.1.4 Perform Network Latency Measurements

As discussed in Chapter 3, latency measurements using iperf need to be conducted between Host 1-1 and Host 2-2. Below reiterates the steps for performing network latency measurements.

An iperf server shall be started on Host 2-2 by performing the following:

```
$ iperf -e -s
```

Then, from Host 1-1 a 512KB message shall be sent to Host 2-2 by performing the following:

```
$ iperf -e -n 512KB -c <Host 2-2 IP Address>
```

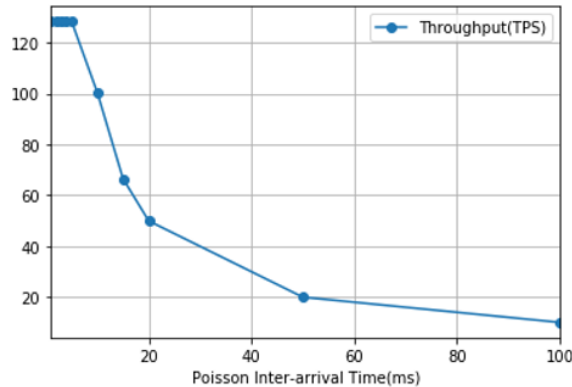
Appendix A.6 details the latency measurements gathered for all experiments. To reiterate, for every blockchain simulation experiment 1000 blockchain transactions were processed in the network. Each experiment was conducted 10 times and the throughput results were averaged. Cumulative distribution function graphs show individual blockchain transaction times in the system for a single experiment. After taking network latency measurements from GENI, we will use that data to inform a normalized distribution. This was done so we

could induce randomness into the simulator and generate the tens of thousands of latency costs required to perform experimentation. Blockchain transaction mean inter-arrival times were drawn from a Poisson distribution and are detailed in the analysis of each experiment. We will abbreviate “mean inter-arrival time” as “interval” in figures.

4.2 Experiment 1 Analysis

The objectives of Experiment 1 are to validate a working blockchain simulator, inform decisions on appropriate Poisson mean inter-arrival times for Experiments 2 and 3, and determine a theoretical maximum efficiency in terms of transaction throughput. As seen in Figure 4.1, the maximum throughput is approximately 128.27 transactions per second (TPS).

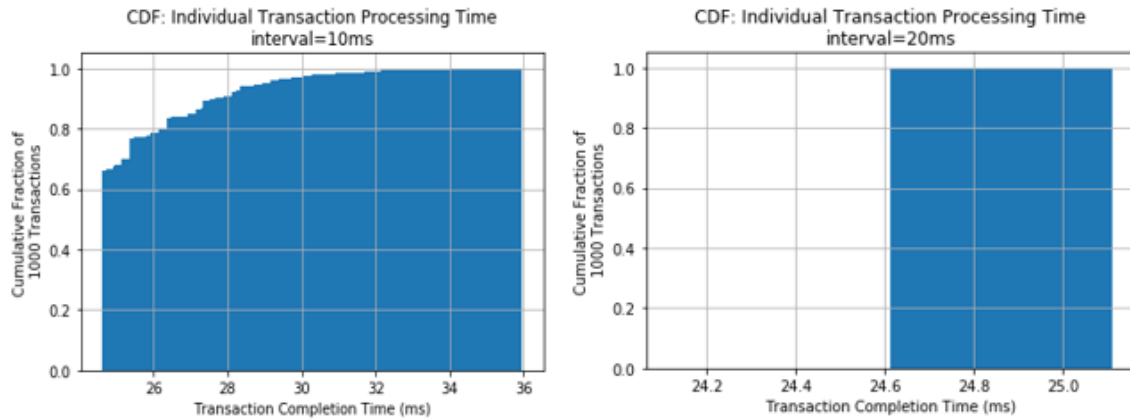
Figure 4.1. Experiment 1 Transaction Throughput



For co-located blockchain processing components, network latency is approximated to be zero. Maximum throughput plateaued at 128.27 TPS (transactions per second).

Also of interest is approximately how long transactions spend processing within the blockchain network. This figure gives an indication of network congestion. The longer the individual component queues are, the longer the transaction will have to wait to be processed. On a real network, this could quickly result in processing component overload and dropped packets while also increasing the overall transaction completion times due to retransmission of dropped packets. Figure 4.2 shows the cumulative distribution functions (CDFs) of the transaction completion times collected from experiments with the mean transaction inter-arrival times set to 10ms and 20ms, respectively.

Figure 4.2. Experiment 1 Transaction Processing Times



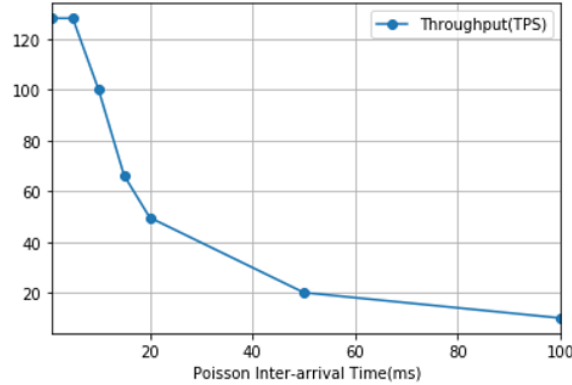
Average transaction processing time improves significantly at interval=10ms. At interval=20ms, no queuing occurred in the system and all transactions proceeded unimpeded by other transactions.

Mean inter-arrival times of 5ms and less resulted in a linearly increasing transaction processing time. Transactions entered the network at such a rate that a bottleneck formed causing increased wait times for each subsequent transaction. At inter-arrival times of 5ms, the last transaction took approximately 2.6s to process from end to end in the blockchain network. Therefore, the rate at which transactions were entering the network is not supportable long term as transaction wait queues would grow without bound. Processing time significantly improved at inter-arrival times of 10ms and greater. Experiments 2 and 3 examined if increased latency associated with growing geographic separation had a significant impact on overall system throughput and individual transaction processing time.

4.3 Experiment 2 Analysis

Experiment 2 is the first experiment to consider network latency impact on the blockchain performance. SDN 1 and SDN 2 were co-located on the same server rack, so performance impact was estimated to be minimal. Figure 4.3 shows throughput for various transaction inter-arrival times. The small latency cost imposed by the network made little impact on overall throughput.

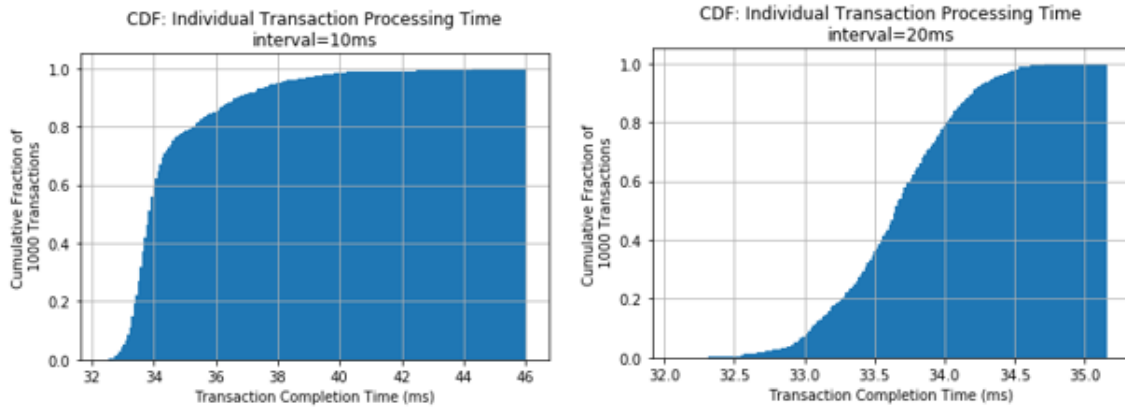
Figure 4.3. Experiment 2 Transaction Throughput



Experiment 2 observes blockchain transaction impact when two SDNs are co-located. As expected, the performance is very similar to that of Experiment 1. Maximum throughput for Experiment 2 is approximately 128.13 TPS (For comparison, Experiment 1 was 128.27 TPS).

Also like Experiment 1, network congestion is evident at transaction mean inter-arrival times of 5ms and less. At these levels the queuing in the system grows without bound and a linearly increasing wait is imposed on each subsequent transaction to enter the network. The last transaction for interval=1ms took approximately 6.75 seconds to process. So, while throughput is maximized at interval=5ms and less, the blockchain component processing queues grow without bound and the system becomes unstable. At interval=10ms the whole blockchain system is able to keep pace with the arrival rate of transactions. This makes sense because the single greatest latency cost in this network is the transaction validation and ledger commit stage, which averages 7.78ms. Experiment 2 observed very little change from Experiment 1. This is expected due to the co-location of the two SDNs. This design is representative of many SDN environments that communicate within the same geographic location.

Figure 4.4. Experiment 2 Transaction Processing Times

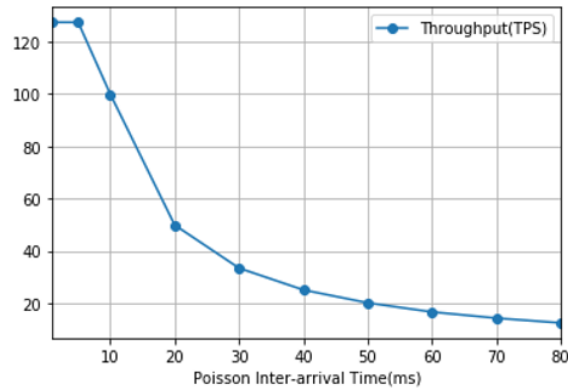


Results are nearly identical to Experiment 1. Processing time significantly improved at inter-arrival times of 10ms and greater.

4.4 Experiment 3 Analysis

Experiment 3 begins by moving SDN 2 from Monterey, CA to San Diego, CA. This represents an increase in distance of approximately 380 miles as a crow flies. Network latency for these transactions is much more significant. However, Figure 4.5 shows a nearly optimal throughput graph. Even though the network latency cost was nearly five times that as the network latency between co-located SDNs, the overall transaction throughput is still approximately 127.45 TPS. These are excellent results, as they suggest that geographic distance will not be a significant detriment to blockchain-enabled SDN operations.

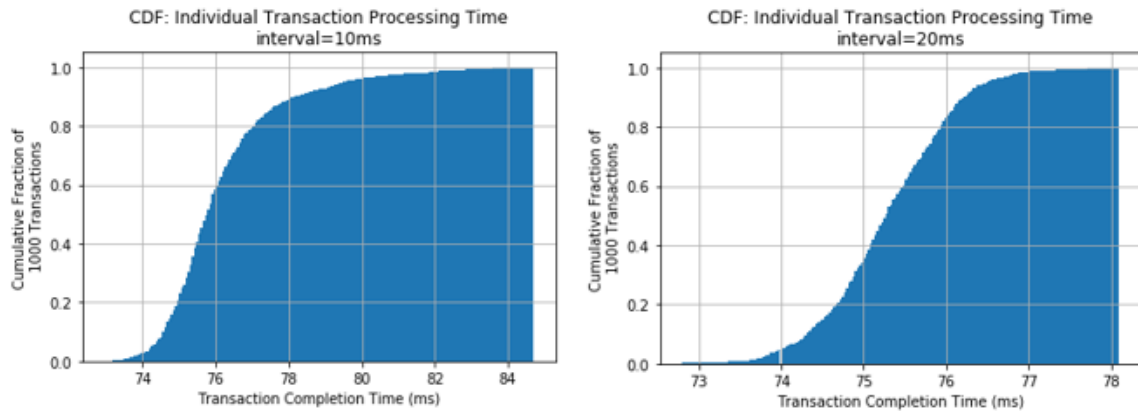
Figure 4.5. Experiment 3 Transaction Throughput Cross-State



Maximum throughput is approximately 127.45 TPS (For comparison, Experiment 1 was 128.27 TPS).

Figure 4.6 shows blockchain transaction processing times in the cross-state experiment. A stark difference can be observed in the average time a transaction spends in the network as compared to transactions in Experiment 1. In this cross-state experiment, transactions spend on average more than twice as much time being processed in the blockchain network. However, transaction throughput is nearly the same. This is very telling of transaction behavior. In Experiment 2, much of the time a transaction spends in the blockchain network is actually spent in transit. So, even though the latency is greatly increased by the extra distance travelled, the throughput is still largely determined by the blockchain processing functions and queues associated with transactions waiting to be processed in order. When transiting the network, a blockchain transaction does not need to wait for a preceding transaction to reach its destination before it begins to transit.

Figure 4.6. Experiment 3 Transaction Processing Times Cross-State

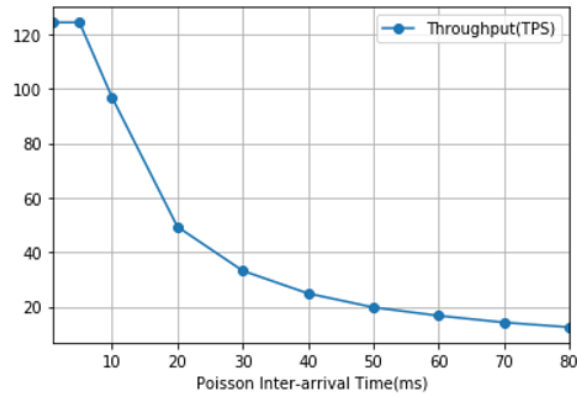


Transactions spend significantly more time in the blockchain network as compared to Experiments 1 and 2. At interval=20ms, we see similar behavior as Experiment 2 where almost no queuing occurs in the system.

The cross-state experiment showed very promising results. Network latency showed significant impact to individual blockchain transaction times, but minimal impact to overall transaction throughput. We will now examine the impact that network latency imposes on blockchain transactions where SDNs are now separated by 1000s of miles.

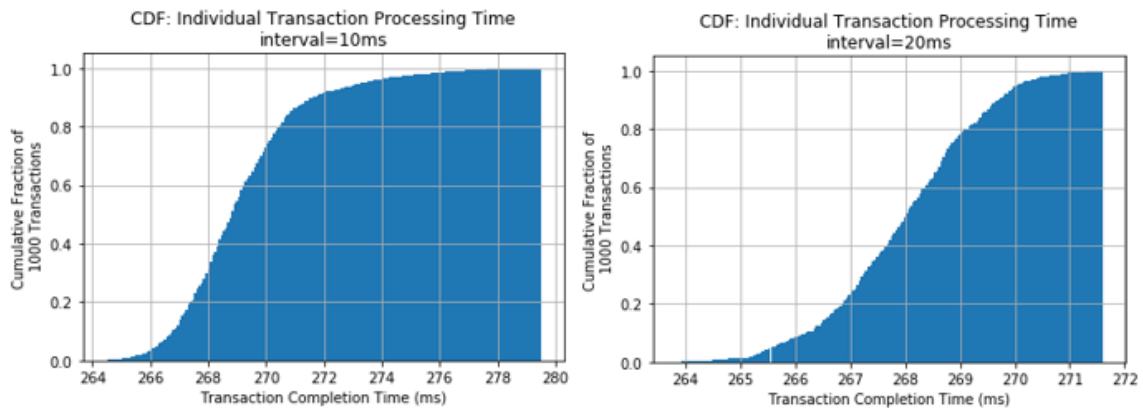
Figure 4.7 shows the throughput for the cross-country experiment. Here, we see a slightly decreased throughput for given transaction inter-arrival times as compared to Experiment 1. Figure 4.8 shows the blockchain transaction processing times in the system. The individual transaction processing times increased greatly. However, just like in the cross-

Figure 4.7. Experiment 3 Transaction Throughput Cross-Country



This experiment begins with moving SDN 2 from San Diego, CA to Burlington, VT (2,577 miles as the crow flies). Maximum throughput is approximately 124.36 TPS (For comparison, Experiment 1 was 128.27 TPS).

Figure 4.8. Experiment 3 Transaction Processing Times Cross-Country



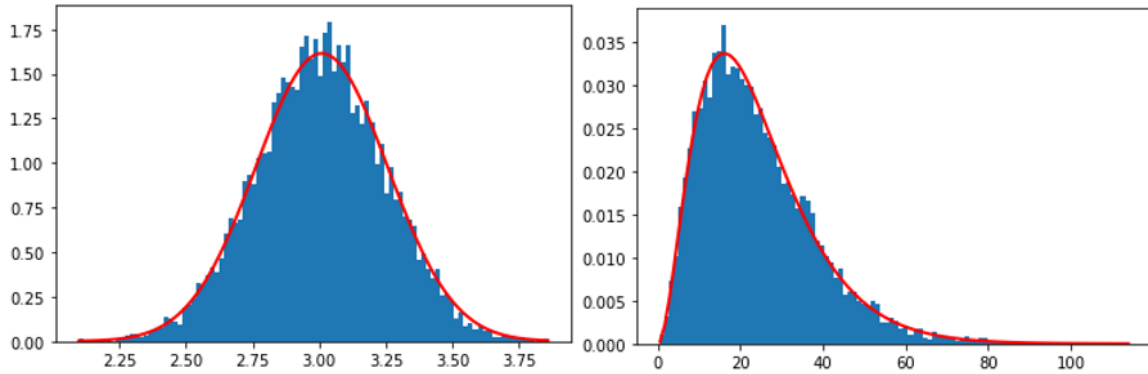
For cross-country SDNs, individual transactions spend significantly more time in the blockchain network, but throughput suffers little impact.

state experiment, we saw overall transaction throughput minimally impacted. The cross-country experiment continued the positive trend observed in the cross-state experiment. Blockchain transaction throughput is not significantly impacted by network latency caused by increasing geographic distances between SDNs.

4.5 Further Experimentation with Heavy-Tailed Distribution of Network Latency

A massive advantage in conducting this research with a simulator is the adaptability in designing and testing new experiments. The Hyperledger Fabric blockchain discrete event simulator created for this research allows for a great deal of flexibility in terms of parameter modification. Here, we demonstrate the ease with which one can make changes to the DES to explore other performance impacts to the blockchain-enabled SDN. The latency measurements taken on GENI for Experiments 1-3 experienced “normal” traffic load and can be represented visually in the left graph of Figure 4.9. This is representative of a best-case network scenario. However, it is often the case that network traffic becomes congested at inopportune times. Here we repeat Experiment 2, but induce extra latency with a gamma distribution where shape and scale are set to 3.009 and 8, respectively. The shape and scale parameters for this heavy-tailed distribution were chosen intentionally to impose significant latency cost in keeping with congested networks. Figure 4.10 shows the results.

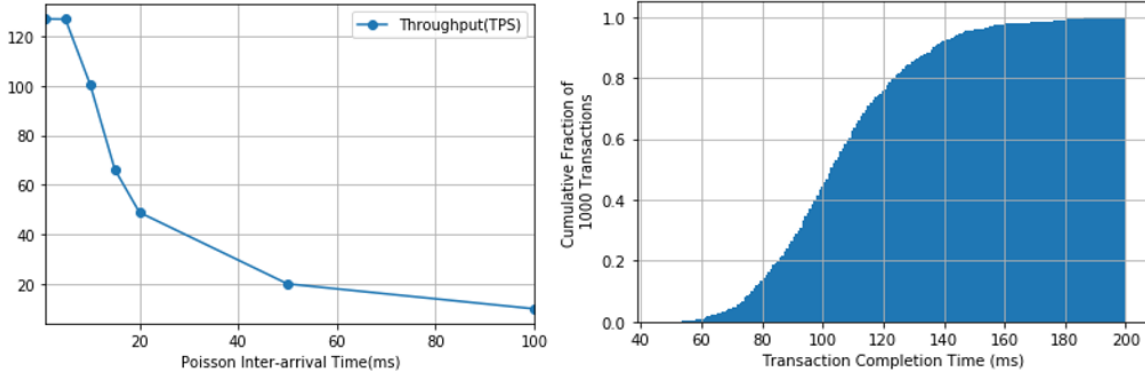
Figure 4.9. Comparison of Normal and Heavy-Tailed Distributions



Visual examples of what normal (left) and heavy (right) latency costs are due to network traffic load. The x-axis represents mean inter-arrival times.

In comparison to the results reported in Figure 4.3 and Figure 4.4, we can observe a reduction of two transactions per second (or 1% decrease in transaction throughput), but more than a 500% increase in the individual transaction processing time. This experiment was easily added using the tools developed for this research. To do so without the model simulator and instead on a real blockchain would be significantly more resource intensive (people and time). Therefore, before moving to blockchain deployment, it is likely beneficial to first experiment with a simulator such as the one developed for this research.

Figure 4.10. Heavy-tailed Distribution Latency Results

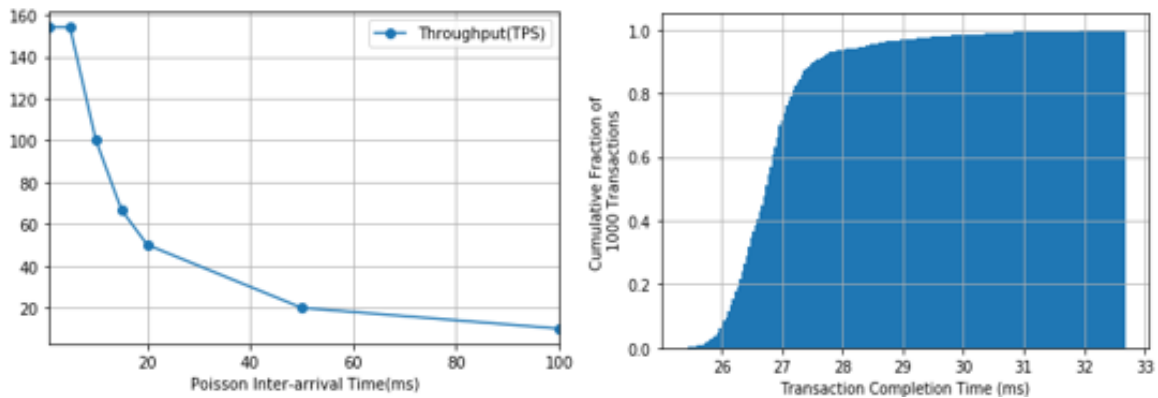


Co-located SDNs, but with a heavy-tailed distribution for network latency costs. Maximum throughput is 126.95 TPS. The inter-arrival time for the right graph depicting individual transaction completion times is 10ms.

4.6 Further Experimentation with Performance Optimizations

Research conducted by Thakkar et. al. resulted in performance optimizations of the validation and ledger commit stage of nearly one order of magnitude [22]. Using the DES created for this research we can quickly make adjustments to that specific stage in the blockchain network and analyze the results. Using the co-located SDN model as a basis and updating the validation and commit stage of the DES produces the results seen in Figure 4.11.

Figure 4.11. Validation And Ledger Commit Optimization



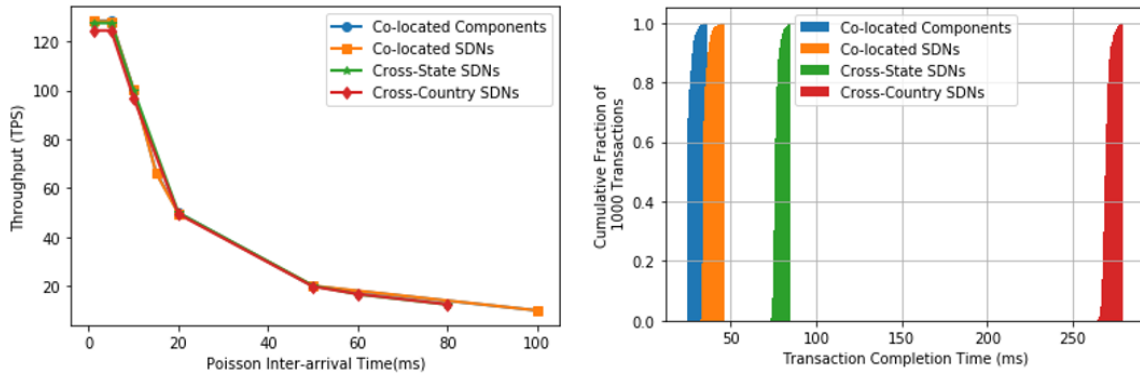
Maximum throughput in left graph is 154.09 TPS. The time interval chosen for the right graph is a Poisson mean inter-arrival time of 10ms.

The optimization produces an increase in throughput of nearly 25 transactions per second (a nearly 20% increase). Additional analysis of the blockchain model reveals that the processing bottleneck within the network has moved from the validation and ledger commit stage to the transaction assembly stage. Implementing this change in a fully integrated blockchain network would not be trivial. However, doing so in this simulator allows us to analyze results in minutes. The last two experiments served as examples of some of the benefits of conducting these experiments in a simulator.

4.7 Analysis Summary

A important observation made in this study is how the different latency delays impact blockchain performance. Figure 4.12 summarizes the results of Experiments 1 through 3. In the blockchain network, processing at a component (like the peer validation stage) must occur in order and a new transaction cannot begin processing until the current transaction is completed. The network delays do not function in this manner. A transaction does not need to complete its transit to a new blockchain component before another transaction starts its transit. The parallelism in how transactions transit a network results in blockchain-enabled SDN functionality that is not significantly impacted by network delays associated with geographic separation.

Figure 4.12. Combined Experiments 1-3



Regardless of geographic separation and individual transaction completion times, transaction throughput for various mean inter-arrival times was nearly the same. The transaction mean inter-arrival time chosen for the right graph depicting individual transaction completion times is 10ms.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5:

Conclusions and Future Work

The goal of this chapter is to summarize conclusions following the analysis presented in Chapter 4. Additionally, we will describe limitations and propose future work that may build upon insights and lessons learned in this research.

5.1 Conclusions

Although network latency impacts the end-to-end individual transaction processing time in a blockchain-enabled SDN, delays associated with blockchain component processing operations have significantly more impact on transaction throughput. This research observed nearly identical blockchain transaction throughput performance across varying geographic distributions. This suggests that distance between SDNs may not be a critical cause for concern in determining blockchain-enabled SDN feasibility. The automation scripts produced as a result of this study should greatly aid research that follows. Additionally, the simulator developed here is highly adaptable to future research and provides many opportunities to analyze “what if” scenarios.

5.1.1 Limitations

This research used a custom Python discrete event simulator to model Hyperledger Fabric blockchain transactions. It relies on the accuracy of reported processing measurements of blockchain components. Many factors play into the rate with which blockchain transactions occur such as block size, endorsement policy, number of endorsing peers, etc. The current simulator does not yet incorporate this level of complexity. However, it does provide insight into how these different blockchain components may contribute to the overall system performance. This study used transactions per second as a metric by which to analyze blockchain performance. Though we found that geographic separation did not significantly impact blockchain performance, we cannot confirm without an integrated blockchain if the throughput found in this study is sufficient to support east/west communication between SDNs. Additionally, the blockchain simulator in the current state does not account for

disconnection of peers or different endorsement policies. There are numerous features that could be added which are left to future research.

5.1.2 Lessons Learned

Many challenges were overcome through the course of completing this research. In this section, we hope to provide some important lessons learned to help those that follow. There are significant challenges in deploying custom Hyperledger Fabric blockchain networks. One of the most critical areas lacking in Hyperledger Fabric documentation is how to best design and build a network from scratch. Much of the documentation covers tutorials for modifying what already exists, but there is very little in the way of how to construct your own network. Hyperledger Fabric 2.0 looks very promising. Hopefully the documentation will eventually be able to keep pace with the rapid development process. ONOS documentation is significantly out of date. Official install guidelines are missing important steps and data. Newer ONOS releases have incorporated a new framework for clustering SDNs. At the time of this writing there is no official ONOS guidance on how to employ it. In contrast, GENI is an outstanding resource that is both well documented and extremely reliable. The ease with which one can launch VMs and begin research cannot be overstated.

5.2 Future Work

The simulator developed for this research does not fully address all of the complexities inherent to a Hyperledger Fabric blockchain network. However, this simulator is highly adaptable and can be modified further to address these shortcomings. Potential upgrades to the simulator include:

1. Support of more complex endorsement policies
2. Blockchain components which may disconnect from the network
3. Increase or decrease number of peer nodes
4. Allow transactions to originate from multiple sources
5. Support multiple simultaneous geographic locations of blockchain components

Following this, the next step is the development of a fully integrated blockchain-enabled SDN. However, based upon experience gained in this research Hyperledger Fabric likely

needs to mature a lot more through real-world deployment and testing in order to support such integration efforts.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX: Configuration Files, Scripts, Code

A.1 GENI RSPEC Example

The below RSPEC was developed for this research and reserves resources for a network composed of one controller, two switches, and four hosts. Figure A.1 is defined using the A.1.1 specification.

Figure A.1. GENI RSPEC Visualization

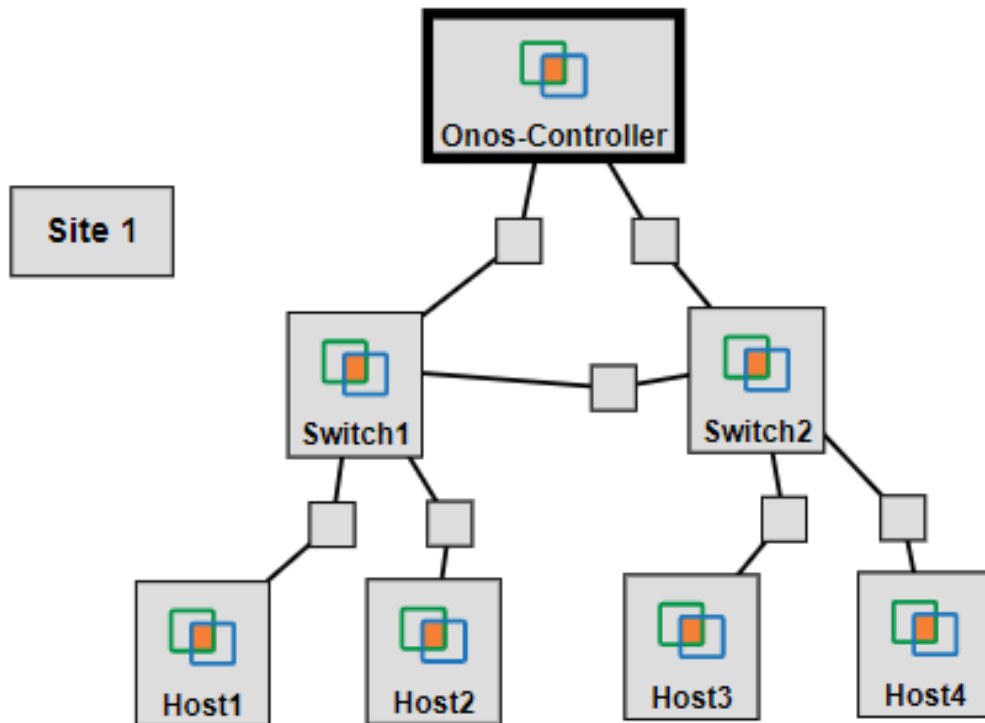


Image depicts SDN defined by RSPEC developed through the course of this research below.

A.1.1 rspec_example.xml

```
<rspec xmlns="http://www.geni.net/resources/rspec/3" xmlns:emulab="http://www.protonet.net/resources/rspec/ext/emulab/1" xmlns:tour="http:
```

```

//www.protopeni.net/resources/rspec/ext/apt-tour/1" xmlns:jacks="
http://www.protopeni.net/resources/rspec/ext/jacks/1" xmlns:xsi="
http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http:
//www.geni.net/resources/rspec/3 http://www.geni.net/resources/
rspec/3/request.xsd" type="request">
2 <node xmlns="http://www.geni.net/resources/rspec/3" client_id="Host1">
3   <jacks:icon xmlns="http://www.protopeni.net/resources/rspec/ext/
jacks/1" url="https://portal.geni.net/images/Xen-VM.svg"/>
4   <sliver_type xmlns="http://www.geni.net/resources/rspec/3" name="
emulab-xen">
5     <disk_image xmlns="http://www.geni.net/resources/rspec/3" name="
urn:publicid:IDN+emulab.net+image+emulab-ops:UBUNTU18-64-STD"/>
6   </sliver_type>
7   <services xmlns="http://www.geni.net/resources/rspec/3"/>
8   <jacks:site xmlns="http://www.protopeni.net/resources/rspec/ext/
jacks/1" id="Site 1"/>
9   <interface xmlns="http://www.geni.net/resources/rspec/3" client_id="
S1toH1">
10     <ip xmlns="http://www.geni.net/resources/rspec/3" address="
10.0.0.1" type="ipv4" netmask="255.255.255.0"/>
11   </interface>
12   <emulab:routable_control_ip xmlns="http://www.protopeni.net/
resources/rspec/ext/emulab/1"/>
13 </node>
14 <node xmlns="http://www.geni.net/resources/rspec/3" client_id="Host2">
15   <jacks:icon xmlns="http://www.protopeni.net/resources/rspec/ext/
jacks/1" url="https://portal.geni.net/images/Xen-VM.svg"/>
16   <sliver_type xmlns="http://www.geni.net/resources/rspec/3" name="
emulab-xen">
17     <disk_image xmlns="http://www.geni.net/resources/rspec/3" name="
urn:publicid:IDN+emulab.net+image+emulab-ops:UBUNTU18-64-STD"/>
18   </sliver_type>
19   <services xmlns="http://www.geni.net/resources/rspec/3"/>
20   <jacks:site xmlns="http://www.protopeni.net/resources/rspec/ext/
jacks/1" id="Site 1"/>
21   <interface xmlns="http://www.geni.net/resources/rspec/3" client_id="
S1toH2">
22     <ip xmlns="http://www.geni.net/resources/rspec/3" address="
10.0.0.2" type="ipv4" netmask="255.255.255.0"/>
23   </interface>

```

```

24     <emulab:routable_control_ip xmlns="http://www.protopeni.net/
resources/rspec/ext/emulab/1"/>
25 </node>
26 <node xmlns="http://www.geni.net/resources/rspec/3" client_id="Switch1
">
27     <jacks:icon xmlns="http://www.protopeni.net/resources/rspec/ext/
jacks/1" url="https://portal.geni.net/images/Xen-VM.svg"/>
28     <sliver_type xmlns="http://www.geni.net/resources/rspec/3" name="
emulab-xen">
29         <disk_image xmlns="http://www.geni.net/resources/rspec/3" name="
urn:publicid:IDN+emulab.net+image+emulab-ops:UBUNTU14-OVS2.31"/>
30     </sliver_type>
31     <services xmlns="http://www.geni.net/resources/rspec/3"/>
32     <jacks:site xmlns="http://www.protopeni.net/resources/rspec/ext/
jacks/1" id="Site 1"/>
33     <interface xmlns="http://www.geni.net/resources/rspec/3" client_id="
H1toS1">
34         <ip xmlns="http://www.geni.net/resources/rspec/3" address="
10.0.0.11" type="ipv4" netmask="255.255.255.0"/>
35     </interface>
36     <interface xmlns="http://www.geni.net/resources/rspec/3" client_id="
H2toS1">
37         <ip xmlns="http://www.geni.net/resources/rspec/3" address="
10.0.0.12" type="ipv4" netmask="255.255.255.0"/>
38     </interface>
39     <interface xmlns="http://www.geni.net/resources/rspec/3" client_id="
S2toS1">
40         <ip xmlns="http://www.geni.net/resources/rspec/3" address="
10.0.0.16" type="ipv4" netmask="255.255.255.0"/>
41     </interface>
42     <interface xmlns="http://www.geni.net/resources/rspec/3" client_id="
CtoS1">
43         <ip xmlns="http://www.geni.net/resources/rspec/3" address="
10.0.0.17" type="ipv4" netmask="255.255.255.0"/>
44     </interface>
45     <emulab:routable_control_ip xmlns="http://www.protopeni.net/
resources/rspec/ext/emulab/1"/>
46 </node>
47 <node xmlns="http://www.geni.net/resources/rspec/3" client_id="Host3">

```

```

48 <jacks:icon xmlns="http://www.protogeni.net/resources/rspec/ext/
jacks/1" url="https://portal.geni.net/images/Xen-VM.svg"/>
49 <jacks:site xmlns="http://www.protogeni.net/resources/rspec/ext/
jacks/1" id="Site 1"/>
50 <sliver_type xmlns="http://www.geni.net/resources/rspec/3" name="
emulab-xen">
51 <disk_image xmlns="http://www.geni.net/resources/rspec/3" name="
urn:publicid:IDN+emulab.net+image+emulab-ops:UBUNTU18-64-STD"/>
52 </sliver_type>
53 <services xmlns="http://www.geni.net/resources/rspec/3"/>
54 <interface xmlns="http://www.geni.net/resources/rspec/3" client_id="
S2toH3">
55 <ip xmlns="http://www.geni.net/resources/rspec/3" address="
10.0.0.3" type="ipv4" netmask="255.255.255.0"/>
56 </interface>
57 <emulab:routable_control_ip xmlns="http://www.protogeni.net/
resources/rspec/ext/emulab/1"/>
58 </node>
59 <node xmlns="http://www.geni.net/resources/rspec/3" client_id="Switch2
">
60 <jacks:icon xmlns="http://www.protogeni.net/resources/rspec/ext/
jacks/1" url="https://portal.geni.net/images/Xen-VM.svg"/>
61 <jacks:site xmlns="http://www.protogeni.net/resources/rspec/ext/
jacks/1" id="Site 1"/>
62 <sliver_type xmlns="http://www.geni.net/resources/rspec/3" name="
emulab-xen">
63 <disk_image xmlns="http://www.geni.net/resources/rspec/3" name="
urn:publicid:IDN+emulab.net+image+emulab-ops:UBUNTU14-OVS2.31"/>
64 </sliver_type>
65 <services xmlns="http://www.geni.net/resources/rspec/3"/>
66 <interface xmlns="http://www.geni.net/resources/rspec/3" client_id="
S1toS2">
67 <ip xmlns="http://www.geni.net/resources/rspec/3" address="
10.0.0.15" type="ipv4" netmask="255.255.255.0"/>
68 </interface>
69 <interface xmlns="http://www.geni.net/resources/rspec/3" client_id="
H3toS2">
70 <ip xmlns="http://www.geni.net/resources/rspec/3" address="
10.0.0.13" type="ipv4" netmask="255.255.255.0"/>
71 </interface>

```

```

72 <interface xmlns="http://www.geni.net/resources/rspec/3" client_id="
H4toS2">
73 <ip xmlns="http://www.geni.net/resources/rspec/3" address="
10.0.0.14" type="ipv4" netmask="255.255.255.0"/>
74 </interface>
75 <interface xmlns="http://www.geni.net/resources/rspec/3" client_id="
CtoS2">
76 <ip xmlns="http://www.geni.net/resources/rspec/3" address="
10.0.0.19" type="ipv4" netmask="255.255.255.0"/>
77 </interface>
78 <emulab:routable_control_ip xmlns="http://www.protoneni.net/
resources/rspec/ext/emulab/1"/>
79 </node>
80 <node xmlns="http://www.geni.net/resources/rspec/3" client_id="Host4">
81 <jacks:icon xmlns="http://www.protoneni.net/resources/rspec/ext/
jacks/1" url="https://portal.geni.net/images/Xen-VM.svg"/>
82 <jacks:site xmlns="http://www.protoneni.net/resources/rspec/ext/
jacks/1" id="Site 1"/>
83 <sliver_type xmlns="http://www.geni.net/resources/rspec/3" name="
emulab-xen">
84 <disk_image xmlns="http://www.geni.net/resources/rspec/3" name="
urn:publicid:IDN+emulab.net+image+emulab-ops:UBUNTU18-64-STD"/>
85 </sliver_type>
86 <services xmlns="http://www.geni.net/resources/rspec/3"/>
87 <interface xmlns="http://www.geni.net/resources/rspec/3" client_id="
S2toH4">
88 <ip xmlns="http://www.geni.net/resources/rspec/3" address="
10.0.0.4" type="ipv4" netmask="255.255.255.0"/>
89 </interface>
90 <emulab:routable_control_ip xmlns="http://www.protoneni.net/
resources/rspec/ext/emulab/1"/>
91 </node>
92 <node xmlns="http://www.geni.net/resources/rspec/3" client_id="Onos-
Controller">
93 <jacks:icon xmlns="http://www.protoneni.net/resources/rspec/ext/
jacks/1" url="https://portal.geni.net/images/Xen-VM.svg"/>
94 <jacks:site xmlns="http://www.protoneni.net/resources/rspec/ext/
jacks/1" id="Site 1"/>
95 <emulab:routable_control_ip xmlns="http://www.protoneni.net/
resources/rspec/ext/emulab/1"/>

```

```

96 <sliver_type xmlns="http://www.geni.net/resources/rspec/3" name="
emulab-xen">
97 <disk_image xmlns="http://www.geni.net/resources/rspec/3" name="
urn:publicid:IDN+emulab.net+image+emulab-ops:UBUNTU18-64-STD"/>
98 </sliver_type>
99 <services xmlns="http://www.geni.net/resources/rspec/3">
100 <execute xmlns="http://www.geni.net/resources/rspec/3" command="
sudo wget --load-cookies /tmp/cookies.txt &quot;https://docs.google.
com/uc?export=download&amp;confirm=$(wget --quiet --save-cookies /
tmp/cookies.txt --keep-session-cookies --no-check-certificate '
https://docs.google.com/uc?export=download&amp;id=1
NjKJJ0RkdCdibus2c44rxx2bxHZVKlfd' -O- | sed -rn 's/.*confirm=([0-9A-
Za-z_]+).*/\1\n/p')&amp;id=1NjKJJ0RkdCdibus2c44rxx2bxHZVKlfd&quot; -
O /opt/onos_install.sh &amp;&amp; rm -rf /tmp/cookies.txt" shell="/
bin/sh"/>
101 </services>
102 <interface xmlns="http://www.geni.net/resources/rspec/3" client_id="
S1toC">
103 <ip xmlns="http://www.geni.net/resources/rspec/3" address="
10.0.0.18" type="ipv4" netmask="255.255.255.0"/>
104 </interface>
105 <interface xmlns="http://www.geni.net/resources/rspec/3" client_id="
S2toC">
106 <ip xmlns="http://www.geni.net/resources/rspec/3" address="
10.0.0.20" type="ipv4" netmask="255.255.255.0"/>
107 </interface>
108 </node>
109 <link xmlns="http://www.geni.net/resources/rspec/3" client_id="link-1"
>
110 <emulab:link_attribute xmlns="http://www.protogeni.net/resources/
rspec/ext/emulab/1" key="nomac_learning" value="yep"/>
111 <jacks:site xmlns="http://www.protogeni.net/resources/rspec/ext/
jacks/1" id="undefined"/>
112 <interface_ref xmlns="http://www.geni.net/resources/rspec/3"
client_id="S1toH1"/>
113 <interface_ref xmlns="http://www.geni.net/resources/rspec/3"
client_id="H1toS1"/>
114 </link>
115 <link xmlns="http://www.geni.net/resources/rspec/3" client_id="link-2"
>

```

```

116 <emulab:link_attribute xmlns="http://www.protoneni.net/resources/
rspec/ext/emulab/1" key="nomac_learning" value="yep"/>
117 <jacks:site xmlns="http://www.protoneni.net/resources/rspec/ext/
jacks/1" id="undefined"/>
118 <interface_ref xmlns="http://www.geni.net/resources/rspec/3"
client_id="S1toH2"/>
119 <interface_ref xmlns="http://www.geni.net/resources/rspec/3"
client_id="H2toS1"/>
120 </link>
121 <link xmlns="http://www.geni.net/resources/rspec/3" client_id="link-0"
>
122 <interface_ref xmlns="http://www.geni.net/resources/rspec/3"
client_id="S2toS1"/>
123 <interface_ref xmlns="http://www.geni.net/resources/rspec/3"
client_id="S1toS2"/>
124 <jacks:site xmlns="http://www.protoneni.net/resources/rspec/ext/
jacks/1" id="undefined"/>
125 <emulab:link_attribute xmlns="http://www.protoneni.net/resources/
rspec/ext/emulab/1" key="nomac_learning" value="yep"/>
126 </link>
127 <link xmlns="http://www.geni.net/resources/rspec/3" client_id="link-3"
>
128 <interface_ref xmlns="http://www.geni.net/resources/rspec/3"
client_id="H3toS2"/>
129 <interface_ref xmlns="http://www.geni.net/resources/rspec/3"
client_id="S2toH3"/>
130 <jacks:site xmlns="http://www.protoneni.net/resources/rspec/ext/
jacks/1" id="undefined"/>
131 <emulab:link_attribute xmlns="http://www.protoneni.net/resources/
rspec/ext/emulab/1" key="nomac_learning" value="yep"/>
132 </link>
133 <link xmlns="http://www.geni.net/resources/rspec/3" client_id="link-4"
>
134 <interface_ref xmlns="http://www.geni.net/resources/rspec/3"
client_id="H4toS2"/>
135 <interface_ref xmlns="http://www.geni.net/resources/rspec/3"
client_id="S2toH4"/>
136 <jacks:site xmlns="http://www.protoneni.net/resources/rspec/ext/
jacks/1" id="undefined"/>

```

```

137 <emulab:link_attribute xmlns="http://www.protogeni.net/resources/
    rspec/ext/emulab/1" key="nomac_learning" value="yep"/>
138 </link>
139 <link xmlns="http://www.geni.net/resources/rspec/3" client_id="link-5"
    >
140 <interface_ref xmlns="http://www.geni.net/resources/rspec/3"
    client_id="CtoS1"/>
141 <interface_ref xmlns="http://www.geni.net/resources/rspec/3"
    client_id="S1toC"/>
142 <jacks:site xmlns="http://www.protogeni.net/resources/rspec/ext/
    jacks/1" id="undefined"/>
143 <emulab:link_attribute xmlns="http://www.protogeni.net/resources/
    rspec/ext/emulab/1" key="nomac_learning" value="yep"/>
144 </link>
145 <link xmlns="http://www.geni.net/resources/rspec/3" client_id="link-6"
    >
146 <interface_ref xmlns="http://www.geni.net/resources/rspec/3"
    client_id="CtoS2"/>
147 <interface_ref xmlns="http://www.geni.net/resources/rspec/3"
    client_id="S2toC"/>
148 <jacks:site xmlns="http://www.protogeni.net/resources/rspec/ext/
    jacks/1" id="undefined"/>
149 <emulab:link_attribute xmlns="http://www.protogeni.net/resources/
    rspec/ext/emulab/1" key="nomac_learning" value="yep"/>
150 </link>
151 </rspec>

```

A.2 GENI Experiment SDN Instance

The below RSPEC was developed for this research and reserves resources for a network composed of one controller, one switch, and two hosts. This instance was used throughout the experimentation phase to rapidly deploy SDNs across the GENI environment.

A.2.1 sdn_instance.xml

```

1 <!--
2 Network: 10.0.0.X
3 Topology: 1 ONOS Controller(C), 1 OVS Switch(S), 2 Hosts(H):
4           C

```

```

5         *
6         S1
7         *****
8         *           *
9         H1         H2
10    -->
11    <rspec xmlns="http://www.geni.net/resources/rspec/3" xmlns:emulab="http:
//www.protoneni.net/resources/rspec/ext/emulab/1" xmlns:tour="http:
//www.protoneni.net/resources/rspec/ext/apt-tour/1" xmlns:jacks="
http://www.protoneni.net/resources/rspec/ext/jacks/1" xmlns:xsi="
http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http:
//www.geni.net/resources/rspec/3 http://www.geni.net/resources/
rspec/3/request.xsd" type="request">
12    <node xmlns="http://www.geni.net/resources/rspec/3" client_id="Host1">
13    <jacks:icon xmlns="http://www.protoneni.net/resources/rspec/ext/
jacks/1" url="https://portal.geni.net/images/Xen-VM.svg"/>
14    <sliver_type xmlns="http://www.geni.net/resources/rspec/3" name="
emulab-xen">
15    <disk_image xmlns="http://www.geni.net/resources/rspec/3" name="
urn:publicid:IDN+emulab.net+image+emulab-ops:UBUNTU18-64-STD"/>
16    </sliver_type>
17    <services xmlns="http://www.geni.net/resources/rspec/3"/>
18    <jacks:site xmlns="http://www.protoneni.net/resources/rspec/ext/
jacks/1" id="Site 1"/>
19    <interface xmlns="http://www.geni.net/resources/rspec/3" client_id="
S1toH1">
20    <ip xmlns="http://www.geni.net/resources/rspec/3" address="
10.0.0.1" type="ipv4" netmask="255.255.255.0"/>
21    </interface>
22    <emulab:routable_control_ip xmlns="http://www.protoneni.net/
resources/rspec/ext/emulab/1"/>
23    </node>
24    <node xmlns="http://www.geni.net/resources/rspec/3" client_id="Host2">
25    <jacks:icon xmlns="http://www.protoneni.net/resources/rspec/ext/
jacks/1" url="https://portal.geni.net/images/Xen-VM.svg"/>
26    <sliver_type xmlns="http://www.geni.net/resources/rspec/3" name="
emulab-xen">
27    <disk_image xmlns="http://www.geni.net/resources/rspec/3" name="
urn:publicid:IDN+emulab.net+image+emulab-ops:UBUNTU18-64-STD"/>
28    </sliver_type>

```

```

29 <services xmlns="http://www.geni.net/resources/rspec/3"/>
30 <jacks:site xmlns="http://www.protogeni.net/resources/rspec/ext/
jacks/1" id="Site 1"/>
31 <interface xmlns="http://www.geni.net/resources/rspec/3" client_id="
S1toH2">
32 <ip xmlns="http://www.geni.net/resources/rspec/3" address="
10.0.0.2" type="ipv4" netmask="255.255.255.0"/>
33 </interface>
34 <emulab:routable_control_ip xmlns="http://www.protogeni.net/
resources/rspec/ext/emulab/1"/>
35 </node>
36 <node xmlns="http://www.geni.net/resources/rspec/3" client_id="Switch1
">
37 <jacks:icon xmlns="http://www.protogeni.net/resources/rspec/ext/
jacks/1" url="https://portal.geni.net/images/Xen-VM.svg"/>
38 <sliver_type xmlns="http://www.geni.net/resources/rspec/3" name="
emulab-xen">
39 <disk_image xmlns="http://www.geni.net/resources/rspec/3" name="
urn:publicid:IDN+emulab.net+image+emulab-ops:UBUNTU14-OVS2.31"/>
40 </sliver_type>
41 <services xmlns="http://www.geni.net/resources/rspec/3"/>
42 <jacks:site xmlns="http://www.protogeni.net/resources/rspec/ext/
jacks/1" id="Site 1"/>
43 <interface xmlns="http://www.geni.net/resources/rspec/3" client_id="
H1toS1">
44 <ip xmlns="http://www.geni.net/resources/rspec/3" address="
10.0.0.11" type="ipv4" netmask="255.255.255.0"/>
45 </interface>
46 <interface xmlns="http://www.geni.net/resources/rspec/3" client_id="
H2toS1">
47 <ip xmlns="http://www.geni.net/resources/rspec/3" address="
10.0.0.12" type="ipv4" netmask="255.255.255.0"/>
48 </interface>
49 <interface xmlns="http://www.geni.net/resources/rspec/3" client_id="
CtoS1">
50 <ip xmlns="http://www.geni.net/resources/rspec/3" address="
10.0.0.17" type="ipv4" netmask="255.255.255.0"/>
51 </interface>
52 <emulab:routable_control_ip xmlns="http://www.protogeni.net/
resources/rspec/ext/emulab/1"/>

```

```

53 </node>
54 <node xmlns="http://www.geni.net/resources/rspec/3" client_id="Onos-
    Controller">
55   <jacks:icon xmlns="http://www.protogeni.net/resources/rspec/ext/
    jacks/1" url="https://portal.geni.net/images/Xen-VM.svg"/>
56   <jacks:site xmlns="http://www.protogeni.net/resources/rspec/ext/
    jacks/1" id="Site 1"/>
57   <emulab:routable_control_ip xmlns="http://www.protogeni.net/
    resources/rspec/ext/emulab/1"/>
58   <sliver_type xmlns="http://www.geni.net/resources/rspec/3" name="
    emulab-xen">
59     <disk_image xmlns="http://www.geni.net/resources/rspec/3" name="
    urn:publicid:IDN+emulab.net+image+emulab-ops:UBUNTU18-64-STD"/>
60   </sliver_type>
61   <services xmlns="http://www.geni.net/resources/rspec/3">
62   </services>
63   <interface xmlns="http://www.geni.net/resources/rspec/3" client_id="
    S1toC">
64     <ip xmlns="http://www.geni.net/resources/rspec/3" address="
    10.0.0.18" type="ipv4" netmask="255.255.255.0"/>
65   </interface>
66 </node>
67 <link xmlns="http://www.geni.net/resources/rspec/3" client_id="link-1"
    >
68   <emulab:link_attribute xmlns="http://www.protogeni.net/resources/
    rspec/ext/emulab/1" key="nomac_learning" value="yep"/>
69   <jacks:site xmlns="http://www.protogeni.net/resources/rspec/ext/
    jacks/1" id="undefined"/>
70   <interface_ref xmlns="http://www.geni.net/resources/rspec/3"
    client_id="S1toH1"/>
71   <interface_ref xmlns="http://www.geni.net/resources/rspec/3"
    client_id="H1toS1"/>
72 </link>
73 <link xmlns="http://www.geni.net/resources/rspec/3" client_id="link-2"
    >
74   <emulab:link_attribute xmlns="http://www.protogeni.net/resources/
    rspec/ext/emulab/1" key="nomac_learning" value="yep"/>
75   <jacks:site xmlns="http://www.protogeni.net/resources/rspec/ext/
    jacks/1" id="undefined"/>

```

```

76 <interface_ref xmlns="http://www.geni.net/resources/rspec/3"
   client_id="S1toH2"/>
77 <interface_ref xmlns="http://www.geni.net/resources/rspec/3"
   client_id="H2toS1"/>
78 </link>
79 <link xmlns="http://www.geni.net/resources/rspec/3" client_id="link-5"
   >
80 <interface_ref xmlns="http://www.geni.net/resources/rspec/3"
   client_id="CtoS1"/>
81 <interface_ref xmlns="http://www.geni.net/resources/rspec/3"
   client_id="S1toC"/>
82 <jacks:site xmlns="http://www.protogeni.net/resources/rspec/ext/
   jacks/1" id="undefined"/>
83 <emulab:link_attribute xmlns="http://www.protogeni.net/resources/
   rspec/ext/emulab/1" key="nomac_learning" value="yep"/>
84 </link>
85 </rspec>

```

A.3 ONOS Install and Configuration

The below BASH (Bourne-Again SHell) script was developed for this research and installed and configured an ONOS software controller onto a 64-bit Linux OS. Once installed it automatically launched and was accessible via a web browser (<http://XX.YY.WW.ZZ:8181/onos/ui/index.html>) or from the command line on the host machine itself (`ssh -p 8101 karaf@localhost`).

A.3.1 onos_install.sh

```

1 #!/bin/sh
2
3 #To make this executable: $ sudo chmod u+x onos_install.sh
4 #Launch with: $ sudo ./onos_install.sh
5
6 #Accessing the controller from browser:
7 #http://XX.YY.WW.ZZ:8181/onos/ui/index.html
8 #The default username and password is onos, rocks.
9
10 #From Target Machine:
11 #ssh -p 8101 karaf@localhost (password karaf)
12 #/opt/onos/bin/onos

```

```

13
14 if [ $USER != "root" ]; then
15     echo "This install script must be run as root. (Try 'sudo ./
        onos_install.sh')"
16     exit 1
17 fi
18
19 #ONOS: https://wiki.onosproject.org/display/ONOS/Downloads
20 #ONOS requires the following ports to be open, in order to make the
        corresponding functionalities available:
21 #Port 8181 for REST API and GUI
22 #Port 8101 to access the ONOS CLI
23 #Port 9876 for intra-cluster communication (communication between target
        machines)
24 #Port 6653 optional, for OpenFlow
25 #Port 6640 optional, for OVSD
26
27 echo "Opening ONOS ports..."
28 iptables -A INPUT -p tcp --dport 8181 -j ACCEPT
29 iptables -A OUTPUT -p tcp --dport 8181 -j ACCEPT
30 iptables -A INPUT -p tcp --dport 8101 -j ACCEPT
31 iptables -A OUTPUT -p tcp --dport 8101 -j ACCEPT
32 iptables -A INPUT -p tcp --dport 9876 -j ACCEPT
33 iptables -A OUTPUT -p tcp --dport 9876 -j ACCEPT
34 iptables -A INPUT -p tcp --dport 6653 -j ACCEPT
35 iptables -A OUTPUT -p tcp --dport 6653 -j ACCEPT
36 iptables -A INPUT -p tcp --dport 6640 -j ACCEPT
37 iptables -A OUTPUT -p tcp --dport 6640 -j ACCEPT
38
39 apt-get update
40 apt-get -y install iptables-persistent
41
42 #Display rules
43 iptables -S
44 echo "...done"
45
46 #ONOS shouldn't run as root. Create an sdn user and sdn group.
47 echo "Creating sdn user and sdn group..."
48 adduser sdn --system --group
49 echo "...done"

```

```

50
51 #Need to install Java 8
52 #Shareable Link on my Google Drive: Jre-8u212-linux-x64.tar.gz:
53 #https://drive.google.com/file/d/1x4GIj0P27Tdi90aaxmRmtBk0vnPUty0e/view?
    usp=sharing
54 echo "Downloading Java 8..."
55 wget --load-cookies /tmp/cookies.txt "https://docs.google.com/uc?export=
    download&confirm=$(wget --quiet --save-cookies /tmp/cookies.txt --
    keep-session-cookies --no-check-certificate 'https://docs.google.com
    /uc?export=download&id=1x4GIj0P27Tdi90aaxmRmtBk0vnPUty0e' -O- | sed
    -rn 's/.*confirm=([0-9A-Za-z_]+).*/\1\n/p')&id=1
    x4GIj0P27Tdi90aaxmRmtBk0vnPUty0e" -O jre-8u212-linux-x64.tar.gz &&
    rm -rf /tmp/cookies.txt
56 echo "...done"
57
58 echo "Installing Java 8..."
59 mkdir /opt/java-jdk
60 tar -C /opt/java-jdk -zxvf jre-8u212-linux-x64.tar.gz
61
62 #Set defaults:
63 sudo update-alternatives --install /usr/bin/java java /opt/java-jdk/jre1
    .8.0_212/bin/java 1
64 echo "...done"
65
66 #Verify Install:
67 java -version
68
69 #Need to install Curl
70 echo "Installing curl..."
71 apt-get -y install curl
72 echo "...done"
73
74 #Installing ONOS
75 echo "Installing ONOS..."
76 cd /opt
77 wget -c http://repo1.maven.org/maven2/org/onosproject/onos-releases
    /2.1.0/onos-2.1.0.tar.gz
78 tar -zxvf onos-2.1.0.tar.gz
79
80 #Rename the directory:

```

```

81 mv onos-2.1.0 onos
82 echo "...done"
83
84 #Running ONOS using its startup script:
85 echo "Starting ONOS..."
86 /opt/onos/bin/onos-service start &
87 echo "...ready"
88 echo "Access controller from browser: http://XX.YY.WW.ZZ:8181/onos/ui/
    index.html"
89 echo "Access controller from CLI: ssh -p 8101 karaf@localhost"

```

A.4 OVS Switch Configuration

The below BASH script was developed for this research and configured the OVS software switch which was already installed on the specified switch host per the requirements as outlined in the GENIRSPEC SDN instance per A.2. Once installed, the switch automatically connected to the designated SDN controller.

A.4.1 config_switch.sh

```

1 #!/bin/sh
2
3 # To make this executable:  chmod u+x config_switch.sh
4 # Launch with:  ./config_switch.sh
5
6 if [ $USER != "root" ]; then
7     echo "This install script must be run as root. (Try 'sudo ./
    config_switch.sh')"
8     exit 1
9 fi
10
11 # Create an ethernet bridge:
12 ovs-vsctl add-br br0
13
14 # Bring down all Host 1/2 and Switch 2 interfaces
15 ifconfig eth1 0
16 ifconfig eth2 0
17 ifconfig eth3 0
18 ifconfig eth4 0

```

```

19
20 # Add data interfaces to bridge
21 ovs-vsctl add-port br0 eth1
22 ovs-vsctl add-port br0 eth2
23 ovs-vsctl add-port br0 eth3
24 ovs-vsctl add-port br0 eth4
25
26 # Trust but verify
27 ovs-vsctl list-ports br0
28
29 # Point Switch 1 to controller
30 ovs-vsctl set-controller br0 tcp:WWW.XXX.YYY.ZZZ:6653
31
32 # Set switch to fail-safe-mode to secure (controller down = packets
    dropped)
33 ovs-vsctl set-fail-mode br0 secure
34
35 # Trust but verify all settings
36 ovs-vsctl show

```

A.5 Hyperledger Fabric Installation

The below BASH script was developed for this research and installed the Hyperledger Fabric blockchain and required software dependencies on the designated host.

A.5.1 fabric_install.sh

```

1 #!/bin/sh
2
3 # To make this executable: $ sudo chmod u+x fabric_install.sh
4 # Launch with: $ sudo ./fabric_install.sh
5 # Will need to update user 'sjworkma' with user of choice
6
7 if [ $USER != "root" ]; then
8     echo "This install script must be run as root. (Try 'sudo ./
    fabric_install.sh')"
9     exit 1
10 fi
11

```

```

12 # Opening ports
13 iptables -A INPUT -p tcp --dport 5984 -j ACCEPT
14 iptables -A OUTPUT -p tcp --dport 5984 -j ACCEPT
15 iptables -A INPUT -p tcp --dport 8080 -j ACCEPT
16 iptables -A OUTPUT -p tcp --dport 8080 -j ACCEPT
17 iptables -A INPUT -p tcp --dport 7051 -j ACCEPT
18 iptables -A OUTPUT -p tcp --dport 7051 -j ACCEPT
19 iptables -A INPUT -p tcp --dport 7052 -j ACCEPT
20 iptables -A OUTPUT -p tcp --dport 7052 -j ACCEPT
21 iptables -A INPUT -p tcp --dport 7053 -j ACCEPT
22 iptables -A OUTPUT -p tcp --dport 7053 -j ACCEPT
23 iptables -A INPUT -p tcp --dport 8051 -j ACCEPT
24 iptables -A OUTPUT -p tcp --dport 8051 -j ACCEPT
25 iptables -A INPUT -p tcp --dport 8053 -j ACCEPT
26 iptables -A OUTPUT -p tcp --dport 8053 -j ACCEPT
27 iptables -A INPUT -p tcp --dport 7054 -j ACCEPT
28 iptables -A OUTPUT -p tcp --dport 7054 -j ACCEPT
29
30 apt-get update
31 apt-get -y install iptables-persistent
32
33 # Install needed software
34 apt-get -y install vim curl wget git apt-transport-https ca-certificates
    gnupg-agent software-properties-common
35
36 # Add docker's key (and verify it)
37 echo "Adding docker key..."
38 curl -fsSL https://download.docker.com/linux/ubuntu/gpg | apt-key add -
39 apt-key fingerprint 0EBFCD88
40 echo "...done"
41
42 # Set up the stable repository
43 add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/
    ubuntu $(lsb_release -cs) stable"
44
45 # Update the package index and install latest stable docker engine
46 echo "Installing latest stable docker engine..."
47 apt-get -y install docker-ce docker-ce-cli containerd.io
48 echo "...done"
49

```

```

50 # Create docker group and add myself (need to logout for this to take
    effect)
51 groupadd docker
52 usermod -aG docker sjworkma
53
54 # Enable docker to start on boot
55 systemctl enable docker
56
57 # Download and install latest stable docker compose
58 curl -L "https://github.com/docker/compose/releases/download/1.24.1/
    docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-
    compose
59 chmod +x /usr/local/bin/docker-compose
60
61 # Download and install GO
62 wget https://dl.google.com/go/go1.13.6.linux-amd64.tar.gz
63 tar -C /usr/local -xzf go1.13.6.linux-amd64.tar.gz
64 mkdir -p /users/sjworkma/go/src/github.com/hyperledger
65 echo "PATH=$PATH:/usr/local/go/bin" >> /users/sjworkma/.bashrc
66 echo "GOPATH=/users/sjworkma/go" >> /users/sjworkma/.bashrc
67 echo "PATH=$PATH:$GOPATH/bin" >> /users/sjworkma/.bashrc
68
69 # Download and install latest stable Hyperledger Fabric
70 echo "Installing Hyperledger Fabric..."
71 curl -sSL https://raw.githubusercontent.com/hyperledger/fabric/master/
    scripts/bootstrap.sh | bash -s
72 echo "PATH=$PATH:/users/sjworkma/fabric-samples/bin" >> /users/sjworkma
    /.bashrc
73 cd /users/sjworkma/go/src/github.com/hyperledger/
74 git clone https://github.com/hyperledger/fabric-chaincode-go.git
75 git clone https://github.com/hyperledger/fabric-protos-go.git
76 echo "...done"
77
78 #Fix some directory Permissions
79 chown -R sjworkma /users/sjworkma/
80 echo "Need to logout for changes to take effect"

```

A.6 Hyperledger Fabric Discrete Event Simulator

The below Python program was developed for this research and simulated the execution of blockchain transactions in a Hyperledger Fabric network. The design below is specific to the peer organization discussed in Chapter 3. TCP variable values are from real world measurements taken on the GENI environment. Adjusting different parameters (transaction arrival intervals and TCP latency) produced output for use in experiment analysis.

A.6.1 blockchain_des.py

```
1 import random
2 import simpy
3 import pandas as pd
4 import numpy as np
5
6 class Blockchain(object):
7
8     # Initialize Blockchain object
9     def __init__(self, env):
10         self.env = env
11
12     # Generate transactions for blockchain network
13     def gen_tx(self, counter, interval):
14         for i in range(counter):
15             t = self.process_tx(i)
16             env.process(t)
17             # Wait next_tx time until next transaction
18             next_tx = np.random.poisson(interval)
19             yield env.timeout(next_tx)
20
21     # Process transaction in blockchain network
22     # Each stage maintains its own queue of transactions
23     # Stages correspond to variables in mathematical model
24     # Stages process transactions in FIFO (first in, first out)
25     def process_tx(self, name):
26
27         # INIT_TX (stage 1)
28         data.append((name, 'start', env.now))
29         with app0.request() as req:
30             yield req
```

```

31         yield env.timeout(init_tx)
32
33     # TCP
34     yield env.timeout(np.random.normal(tcp_mean, tcp_std))
35
36     # P_RESP (stage 2)
37     with peer0.request() as req:
38         yield req
39         yield env.timeout(p_resp)
40
41     # TCP
42     yield env.timeout(np.random.normal(tcp_mean, tcp_std))
43
44     # ASSEM_TX (stage 3)
45     with app1.request() as req:
46         yield req
47         yield env.timeout(assem_tx)
48
49     # ORDER (stage 4)
50     with orderer.request() as req:
51         yield req
52         yield env.timeout(order)
53
54     # TCP
55     yield env.timeout(np.random.normal(tcp_mean, tcp_std))
56
57     # P_VAL_APP_TX (stage 5 and 6)
58     with peer1.request() as req:
59         yield req
60         yield env.timeout(p_val_app_tx)
61     data.append((name, 'done', env.now))
62
63 # Component latency
64 # Times are in milliseconds
65 # NPS mean/standard deviation: 3.009 / 0.247
66 # University of California, San Diego, CA mean/standard deviation:
67     16.890 / 0.446
68 # University of Vermont, Burlington, VT mean/standard deviation: 81.149
69     / 0.851
70 tcp_mean      = 3.009

```

```

69 tcp_std      = 0.247
70 init_tx      = 6.47
71 p_resp       = 3.25
72 assem_tx     = 5.22
73 order        = 1.894
74 p_val_app_tx = 7.779
75
76 # Set up the environment
77 # Times are in milliseconds
78 # 1 transaction approx every int_tx milliseconds
79 data         = []
80 sim_time     = 10000000
81 num_tx       = 1000
82 int_tx       = 1
83 env          = simpy.Environment()
84 app0         = simpy.Resource(env, capacity=1)
85 app1         = simpy.Resource(env, capacity=1)
86 peer0        = simpy.Resource(env, capacity=1)
87 peer1        = simpy.Resource(env, capacity=1)
88 orderer      = simpy.Resource(env, capacity=1)
89 blockchain   = Blockchain(env)
90
91 # Generate transactions
92 env.process(blockchain.gen_tx(num_tx, int_tx))
93
94 # The process is run for the first sim_time milliseconds
95 # or until num_tx is reached
96 env.run(until = sim_time)
97
98 # Look at results
99 pd_data = pd.DataFrame(data, columns=['tx_id', 'tx_phase', 'time'])
100 print(pd_data.head())
101 print(pd_data.tail())
102
103 csv_file = 'des_out' + '_tcp_' + str(tcp_mean) + '_int_tx_' + str(int_tx
    ) + '.csv'
104 pd_data.to_csv(csv_file, index=False)

```

A.7 Blockchain Analysis

The below Python program was developed for this research and performed analysis of the blockchain transaction output produced by the discrete event simulator. Specifically, transaction throughput and cumulative distribution functions were produced. Below, we looked at the first round of experiments which observed activity on a network where all blockchain components were co-located. Additional experiments used similar code as below, but with different file names as input and other small changes, so for brevity they will not be included.

A.7.1 blockchain_analysis.py

```
1 import random
2 import simpy
3 import pandas as pd
4 import numpy as np
5 import matplotlib.pyplot as plt
6
7 # Read in all DES (tcp=0) csv files
8 pd_tcp0_int1 = pd.read_csv('des_out_tcp_0_int_tx_1.csv')
9 pd_tcp0_int2 = pd.read_csv('des_out_tcp_0_int_tx_2.csv')
10 pd_tcp0_int3 = pd.read_csv('des_out_tcp_0_int_tx_3.csv')
11 pd_tcp0_int4 = pd.read_csv('des_out_tcp_0_int_tx_4.csv')
12 pd_tcp0_int5 = pd.read_csv('des_out_tcp_0_int_tx_5.csv')
13 pd_tcp0_int10 = pd.read_csv('des_out_tcp_0_int_tx_10.csv')
14 pd_tcp0_int15 = pd.read_csv('des_out_tcp_0_int_tx_15.csv')
15 pd_tcp0_int20 = pd.read_csv('des_out_tcp_0_int_tx_20.csv')
16 pd_tcp0_int50 = pd.read_csv('des_out_tcp_0_int_tx_50.csv')
17 pd_tcp0_int100 = pd.read_csv('des_out_tcp_0_int_tx_100.csv')
18
19 # Look at throughput (transactions per second) for given tcp latency and
    interval
20 # Each experiment conducted 1000 transactions (num_tx / tot_time_ms) *
    1000s = TPS
21 tput_tcp0_int1 = (1000 / pd_tcp0_int1.at[1999, 'time']) * 1000
22 tput_tcp0_int2 = (1000 / pd_tcp0_int2.at[1999, 'time']) * 1000
23 tput_tcp0_int3 = (1000 / pd_tcp0_int3.at[1999, 'time']) * 1000
24 tput_tcp0_int4 = (1000 / pd_tcp0_int4.at[1999, 'time']) * 1000
25 tput_tcp0_int5 = (1000 / pd_tcp0_int5.at[1999, 'time']) * 1000
```

```

26 tput_tcp0_int10 = (1000 / pd_tcp0_int10.at[1999, 'time']) * 1000
27 tput_tcp0_int15 = (1000 / pd_tcp0_int15.at[1999, 'time']) * 1000
28 tput_tcp0_int20 = (1000 / pd_tcp0_int20.at[1999, 'time']) * 1000
29 tput_tcp0_int50 = (1000 / pd_tcp0_int50.at[1999, 'time']) * 1000
30 tput_tcp0_int100 = (1000 / pd_tcp0_int100.at[1999, 'time']) * 1000
31 print('Throughput for co-located blockchain components (transactions per
    second), \
32 \ngiven Poisson inter-arrival time:')
33 print('interval = 1ms: ', tput_tcp0_int1)
34 print('interval = 2ms: ', tput_tcp0_int2)
35 print('interval = 3ms: ', tput_tcp0_int3)
36 print('interval = 4ms: ', tput_tcp0_int4)
37 print('interval = 5ms: ', tput_tcp0_int5)
38 print('interval = 10ms: ', tput_tcp0_int10)
39 print('interval = 15ms: ', tput_tcp0_int15)
40 print('interval = 20ms: ', tput_tcp0_int20)
41 print('interval = 50ms: ', tput_tcp0_int50)
42 print('interval = 100ms: ', tput_tcp0_int100)
43
44 df_tput_tcp0 = pd.DataFrame({
45     'Poisson Inter-arrival Time(ms)': [1, 2, 3, 4, 5, 10, 15, 20, 50,
46     100],
47     'Throughput(TPS)': [tput_tcp0_int1, tput_tcp0_int2, tput_tcp0_int3,
48     tput_tcp0_int4, tput_tcp0_int5, tput_tcp0_int10, tput_tcp0_int15,
49     tput_tcp0_int20, tput_tcp0_int50, tput_tcp0_int100]
50 })
51 df_tput_tcp0.plot(kind='line', grid=True, marker='o', x='Poisson Inter-
    arrival Time(ms)', y='Throughput(TPS)', title='Co-located Blockchain
    Components:\nVarying Poisson Inter-arrival Time')
52
53 # Compute total transaction times for individual tx_id, store in new
    column
54 # tx_id 'start' row will contain the total amount of time transaction
    was within blockchain network
55
56 # Formats floats to display 6 digits after decimal point
57 pd.set_option('display.float_format', lambda x: '%.6f' % x)
58
59 # TCP=0, int=1
60 pd_tcp0_int1.sort_values(by='tx_id', kind='mergesort', inplace=True)

```

```

58 pd_tcp0_int1['tot_time'] = np.NaN
59
60 for i in range(pd_tcp0_int1.shape[0]-1):
61     if pd_tcp0_int1.iloc[i, 1] == 'start':
62         pd_tcp0_int1.iloc[i, 3] = np.round((pd_tcp0_int1.iloc[i+1, 2] -
        pd_tcp0_int1.iloc[i, 2]), decimals=3)
63
64 # TCP=0, int=2
65 pd_tcp0_int2.sort_values(by='tx_id', kind='mergesort', inplace=True)
66 pd_tcp0_int2['tot_time'] = np.NaN
67
68 for i in range(pd_tcp0_int2.shape[0]-1):
69     if pd_tcp0_int2.iloc[i, 1] == 'start':
70         pd_tcp0_int2.iloc[i, 3] = np.round((pd_tcp0_int2.iloc[i+1, 2] -
        pd_tcp0_int2.iloc[i, 2]), decimals=3)
71
72 # TCP=0, int=3
73 pd_tcp0_int3.sort_values(by='tx_id', kind='mergesort', inplace=True)
74 pd_tcp0_int3['tot_time'] = np.NaN
75
76 for i in range(pd_tcp0_int3.shape[0]-1):
77     if pd_tcp0_int3.iloc[i, 1] == 'start':
78         pd_tcp0_int3.iloc[i, 3] = np.round((pd_tcp0_int3.iloc[i+1, 2] -
        pd_tcp0_int3.iloc[i, 2]), decimals=3)
79
80 # TCP=0, int=4
81 pd_tcp0_int4.sort_values(by='tx_id', kind='mergesort', inplace=True)
82 pd_tcp0_int4['tot_time'] = np.NaN
83
84 for i in range(pd_tcp0_int4.shape[0]-1):
85     if pd_tcp0_int4.iloc[i, 1] == 'start':
86         pd_tcp0_int4.iloc[i, 3] = np.round((pd_tcp0_int4.iloc[i+1, 2] -
        pd_tcp0_int4.iloc[i, 2]), decimals=3)
87
88 # TCP=0, int=5
89 pd_tcp0_int5.sort_values(by='tx_id', kind='mergesort', inplace=True)
90 pd_tcp0_int5['tot_time'] = np.NaN
91
92 for i in range(pd_tcp0_int5.shape[0]-1):
93     if pd_tcp0_int5.iloc[i, 1] == 'start':

```

```

94         pd_tcp0_int5.iloc[i, 3] = np.round((pd_tcp0_int5.iloc[i+1, 2] -
95         pd_tcp0_int5.iloc[i, 2]), decimals=3)
96
97 # TCP=0, int=10
98 pd_tcp0_int10.sort_values(by='tx_id', kind='mergesort', inplace=True)
99 pd_tcp0_int10['tot_time'] = np.NaN
100
101 for i in range(pd_tcp0_int10.shape[0]-1):
102     if pd_tcp0_int10.iloc[i, 1] == 'start':
103         pd_tcp0_int10.iloc[i, 3] = np.round((pd_tcp0_int10.iloc[i+1, 2]
104         - pd_tcp0_int10.iloc[i, 2]), decimals=3)
105
106 # TCP=0, int=15
107 pd_tcp0_int15.sort_values(by='tx_id', kind='mergesort', inplace=True)
108 pd_tcp0_int15['tot_time'] = np.NaN
109
110 for i in range(pd_tcp0_int15.shape[0]-1):
111     if pd_tcp0_int15.iloc[i, 1] == 'start':
112         pd_tcp0_int15.iloc[i, 3] = np.round((pd_tcp0_int15.iloc[i+1, 2]
113         - pd_tcp0_int15.iloc[i, 2]), decimals=3)
114
115 # TCP=0, int=20
116 pd_tcp0_int20.sort_values(by='tx_id', kind='mergesort', inplace=True)
117 pd_tcp0_int20['tot_time'] = np.NaN
118
119 for i in range(pd_tcp0_int20.shape[0]-1):
120     if pd_tcp0_int20.iloc[i, 1] == 'start':
121         pd_tcp0_int20.iloc[i, 3] = np.round((pd_tcp0_int20.iloc[i+1, 2]
122         - pd_tcp0_int20.iloc[i, 2]), decimals=3)
123
124 # TCP=0, int=50
125 pd_tcp0_int50.sort_values(by='tx_id', kind='mergesort', inplace=True)
126 pd_tcp0_int50['tot_time'] = np.NaN
127
128 for i in range(pd_tcp0_int50.shape[0]-1):
129     if pd_tcp0_int50.iloc[i, 1] == 'start':
130         pd_tcp0_int50.iloc[i, 3] = np.round((pd_tcp0_int50.iloc[i+1, 2]
131         - pd_tcp0_int50.iloc[i, 2]), decimals=3)
132
133 # TCP=0, int=100

```

```

129 pd_tcp0_int100.sort_values(by='tx_id', kind='mergesort', inplace=True)
130 pd_tcp0_int100['tot_time'] = np.NaN
131
132 for i in range(pd_tcp0_int100.shape[0]-1):
133     if pd_tcp0_int100.iloc[i, 1] == 'start':
134         pd_tcp0_int100.iloc[i, 3] = np.round((pd_tcp0_int100.iloc[i+1,
135             2] - pd_tcp0_int100.iloc[i, 2]), decimals=3)
136
137 # Sanity check the data
138 # pd_tcp0_int50.tail()
139
140 # CDF Results: TCP=0, int=1
141 pd_tcp0_int1['tot_time'].dropna().hist(density=True, cumulative=True,
142     bins=200, label='CDF')
143 plt.title('CDF: Individual Transaction Processing Time\ninterval=1ms')
144 plt.xlabel('Transaction Completion Time (ms)')
145 plt.ylabel('Cumulative Fraction of\n 1000 Transactions')
146
147 # CDF Results: TCP=0, int=2
148 pd_tcp0_int2['tot_time'].dropna().hist(density=True, cumulative=True,
149     bins=200, label='CDF')
150 plt.title('CDF: Individual Transaction Processing Time\ninterval=2ms')
151 plt.xlabel('Transaction Completion Time (ms)')
152 plt.ylabel('Cumulative Fraction of\n 1000 Transactions')
153
154 # CDF Results: TCP=0, int=3
155 pd_tcp0_int3['tot_time'].dropna().hist(density=True, cumulative=True,
156     bins=200, label='CDF')
157 plt.title('CDF: Individual Transaction Processing Time\ninterval=3ms')
158 plt.xlabel('Transaction Completion Time (ms)')
159 plt.ylabel('Cumulative Fraction of\n 1000 Transactions')
160
161 # CDF Results: TCP=0, int=4
162 pd_tcp0_int4['tot_time'].dropna().hist(density=True, cumulative=True,
163     bins=200, label='CDF')
164 plt.title('CDF: Individual Transaction Processing Time\ninterval=4ms')
165 plt.xlabel('Transaction Completion Time (ms)')
166 plt.ylabel('Cumulative Fraction of\n 1000 Transactions')
167
168 # CDF Results: TCP=0, int=5

```

```

164 pd_tcp0_int5['tot_time'].dropna().hist(density=True, cumulative=True,
    bins=200, label='CDF')
165 plt.title('CDF: Individual Transaction Processing Time\ninterval=5ms')
166 plt.xlabel('Transaction Completion Time (ms)')
167 plt.ylabel('Cumulative Fraction of\n 1000 Transactions')
168
169 # CDF Results: TCP=0, int=10
170 pd_tcp0_int10['tot_time'].dropna().hist(density=True, cumulative=True,
    bins=200, label='CDF')
171 plt.title('CDF: Individual Transaction Processing Time\ninterval=10ms')
172 plt.xlabel('Transaction Completion Time (ms)')
173 plt.ylabel('Cumulative Fraction of\n 1000 Transactions')
174
175 # CDF Results: TCP=0, int=15
176 pd_tcp0_int15['tot_time'].dropna().hist(density=True, cumulative=True,
    bins=200, label='CDF')
177 plt.title('CDF: Individual Transaction Processing Time\ninterval=15ms')
178 plt.xlabel('Transaction Completion Time (ms)')
179 plt.ylabel('Cumulative Fraction of\n 1000 Transactions')
180
181 # CDF Results: TCP=0, int=20
182 pd_tcp0_int20['tot_time'].dropna().hist(density=True, cumulative=True,
    bins=200, label='CDF')
183 plt.title('CDF: Individual Transaction Processing Time\ninterval=20ms')
184 plt.xlabel('Transaction Completion Time (ms)')
185 plt.ylabel('Cumulative Fraction of\n 1000 Transactions')
186
187 # Plot Results: TCP=0, int=50
188 pd_tcp0_int50['tot_time'].dropna().hist(density=True, cumulative=True,
    bins=200, label='CDF')
189 plt.title('CDF: Individual Transaction Processing Time\ninterval=50ms')
190 plt.xlabel('Transaction Completion Time (ms)')
191 plt.ylabel('Cumulative Fraction of\n 1000 Transactions')
192
193 # CDF Results: TCP=0, int=100
194 pd_tcp0_int100['tot_time'].dropna().hist(density=True, cumulative=True,
    bins=200, label='CDF')
195 plt.title('CDF: Individual Transaction Processing Time\ninterval=100ms')
196 plt.xlabel('Transaction Completion Time (ms)')
197 plt.ylabel('Cumulative Fraction of\n 1000 Transactions')

```

THIS PAGE INTENTIONALLY LEFT BLANK

List of References

- [1] J. Kurose and K. Ross, *Computer Networking: A Top-Down Approach*, 7th ed. Hoboken, NJ: Pearson Education Inc., 2017.
- [2] B. Nunes, M. Mendonca, X. Nguyen, K. Obraczka, and T. Turletti, "A survey of software-defined networking: Past, present, and future of programmable networks," *IEEE Communications Surveys Tutorials*, vol. 16, no. 3, pp. 1617–1634, Mar. 2014.
- [3] K. Bakshi, "Considerations for software defined networking (sdn): Approaches and use cases," in *2013 IEEE Aerospace Conference*, Mar. 2013, pp. 1–9.
- [4] S. Jain, et al., "B4: Experience with a globally-deployed software defined wan," in *SIGCOMM 2013*, 2013, pp. 3–14.
- [5] K. Poularakis et al., "SDN-enabled tactical ad hoc networks: Extending programmable control to the edge," *IEEE Communications Magazine*, vol. 56, no. 7, 2018.
- [6] B. Logan and G. Xie, "Automating distributed firewalls: A case for software defined tactical networks," in *Military Communications Conference*, Norfolk, VA, USA, 2019.
- [7] F. Le, C. Leet, C. Makaya, M. Rio, X. Wang, Y. and R. Yang, "Sfp: Toward a scalable, efficient, stable protocol for federation of software defined networks," in *2017 IEEE SmartWorld, Ubiquitous Intelligence Computing, Advanced Trusted Computed, Scalable Computing Communications, Cloud Big Data Computing, Internet of People and Smart City Innovation*, 2017, pp. 1–6.
- [8] S. Tollefson, "Utilizing blockchain to design an east/west interface for federated software defined networks," M.S. thesis, Dept. Computer Science, Naval Postgraduate School, Dec. 2018.
- [9] "Hyperledger Fabric: A blockchain platform for the enterprise." [Online]. Available: <https://hyperledger-fabric.readthedocs.io/en/release-1.4/index.html>. Accessed: Aug. 14, 2019.
- [10] "Checking the ledger: Permissioned vs. permissionless blockchains." [Online]. Available: <https://www.ibm.com/blogs/think/2016/07/checking-the-ledger-permissioned-vs-permissionless-blockchains/>. Accessed: Aug. 14, 2019.
- [11] "Open Network Operating System." [Online]. Available: <https://onosproject.org/>. Accessed: Sep. 6, 2019.

- [12] "ONOS wiki." [Online]. Available: <https://wiki.onosproject.org/display/ONOS/ONOS>. Accessed: Aug. 22, 2019.
- [13] A. Lara, A. Kolasani, and B. Ramamurthy, "Network innovation using openflow: A survey," in *IEEE Communications Surveys Tutorials*, Quarter 1 2014, pp. 493–512.
- [14] "Hyperledger Fabric: Smart contracts and chaincode." [Online]. Available: <https://hyperledger-fabric.readthedocs.io/en/release-1.4/smartcontract/smartcontract.html>. Accessed: Oct. 06, 2019.
- [15] "Hyperledger." [Online]. Available: <https://hyperledger.org>. Accessed: Oct. 06, 2019.
- [16] "Hyperledger Fabric: Introduction." [Online]. Available: <https://hyperledger-fabric.readthedocs.io/en/release-1.4/blockchain.html>. Accessed: Oct. 06, 2019.
- [17] "What are smart contracts on blockchain?" (2018, Jul.). IBM. [Online]. Available: <https://www.ibm.com/blogs/blockchain/2018/07/what-are-smart-contracts-on-blockchain/>. Accessed Feb. 22, 2019.
- [18] "cURL frequently asked questions." [Online]. Available: <https://curl.haxx.se/docs/faq.html>. Accessed: Oct. 22, 2019.
- [19] "Docker documentation." [Online]. Available: <https://docs.docker.com>. Accessed: Oct. 22, 2019.
- [20] "Go documentation." [Online]. Available: <https://golang.org/doc/>. Accessed: Oct. 22, 2019.
- [21] "Hyperledger Fabric: Peers." [Online]. Available: <https://hyperledger-fabric.readthedocs.io/en/release-1.4/peers/peers.html>. Accessed: Dec. 22, 2019.
- [22] P. Thakkar, S. Nathan, and B. Viswanathan, "Performance benchmarking and optimizing hyperledger fabric blockchain platform," in *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2018, pp. 264–276.
- [23] "GENI: Exploring networks of the future." [Online]. Available: <https://www.geni.net/>. Accessed: Aug. 14, 2019.
- [24] P. Helebrandt and I. Kotuliak, "Novel sdn multi-domain architecture," in *12th IEEE International Conference on Emerging eLearning Technologies and Applications (ICETA)*, Dec. 2014, pp. 139–143.

- [25] H. Yu, K. Li, H. Qi, W. Li, and X. Tao, “Zebra: An east-west control framework for sdn controllers,” in *2015 44th International Conference on Parallel Processing*, 2015, pp. 610–618.
- [26] H. Sukhwani, N. Wang, K. S. Trivedi and A. Rindos, “Performance modeling of hyperledger fabric (permissioned blockchain network),” in *2018 IEEE 17th International Symposium on Network Computing and Applications*, 2018, pp. 1–8.
- [27] J. Bresnahan, R. Kettimuthu, and I. Foster, “XIOPerf: A Tool for Evaluating Network Protocols,” in *2006 3rd International Conference on Broadband Communications, Networks and Systems*. IEEE, 2006, pp. 1–9.

THIS PAGE INTENTIONALLY LEFT BLANK

Initial Distribution List

1. Defense Technical Information Center
Ft. Belvoir, Virginia
2. Dudley Knox Library
Naval Postgraduate School
Monterey, California