Modelling and Evaluating the Ripple Consensus Protocol

An Agent-Based Simulation in Python

Vardan Tandon

MEng Computer Science

Submission Date: 2nd May 2017

Supervisors: Dr. Paolo Tasca/Dr. Claudio J. Tessone

Abstract

This projects aims to investigate and evaluate the consensus protocol of a popular digital payment system, Ripple, now known as Ripple Labs. Ripple has recently been receiving some criticism for a certain assumption it takes for one of its building blocks that contradicts with their claim of being a decentralised payment system. In particular, Ripple assumes that their underlying network topology needs to be constructed in a certain restricted way for their consensus algorithm to prevent any forks in the system and achieve conflicting consensus results. Subsequently, the project models the ripple consensus protocol through an agent-based simulation approach and runs multiple implementations that look at gradually improving upon their existing simulation of this consensus protocol and probing the underlying assumptions used by the Ripple consensus protocol.

Contents

1 Introduction **1**

1.1 Distributed Payment Systems and Digital Currencies 1

1.2 Motivation behind the Problem 2

1.3 Aim and Objectives 3

1.4 Approach Overview 3

1.5 Report Structure 5

2 Background Research and Related Work **6**

2.1 Bitcoin 6

2.1.1 Bitcoin as a Distributed Network 7

2.1.2 Bitcoin as a Consensus Protocol 8

2.2 Ripple 9

2.2.1 Ripple as a Payment System 9

2.2.2 Ripple as a Distributed Network 9

2.2.3 Ripple as a Consensus Protocol 10

2.3 Ripple Protocol Consensus Algorithm (RPCA) 11

2.4 Soundness of the Ripple Consensus Protocol 13

2.4.1 Byzantines Generals Problem 13
2.4.2 Correctness 14
2.4.3 Agreement 15
2.4.4 Utility 16
2.5 Uniqueness of the Ripple Consensus Protocol 17
2.6 Criticism faced by Ripple Labs and its Consensus Protocol 17
2.7 Tools Employed 20
2.7.1 Python 3.0 21
2.7.2 NetworkX 21

3 Simulation Studies and Requirements Gathering 22

3.1 Simulation Study 1: Modelling Nakamoto Consensus 22
3.1.1 Nakamoto consensus Description 22
3.1.2 Nakamoto Model Building Blocks 25
3.1.3 Nakamoto Model Performance Measures 25
3.2 Simulation Study 2: Modelling Ripple Consensus Protocol 26
3.2.1 Ripple Consensus Model Description 27
3.3 Requirements Gathering 29
3.3.1 Ripple Consensus Model Building Blocks 29

3.3.2 Ripple Consensus Model Performance Measures 30

4 Design and Implementation **31**

4.1 Design Choice 1: Current Ripple Simulation Model 31

4.1.1 Implementation Technicalities 31

4.1.2 Design Problems: 34

4.2 Design Choice 2: Multi-threading and More Simulations 35

4.2.1 Implementation Technicalities: 35

4.2.2 Design Problems: 37

4.3 Design Choice 3: Interconnected sub-network of Cliques 37

4.3.1 Implementation Technicalities: 38

4.3.2 Design Problems: 39

5 Results Evaluation **40**

5.1 First Implementation Results and Analysis 40

5.1.1 Second Implementation Results and Analysis 42

5.1.2 Varying Network Size 42

5.1.3 Varying Malicious Servers 43

5.1.4 Varying Outbound Connections 44

5.1.5 Varying UNL size range 45

5.1.6 Varying Network Latencies range 46

5.2 Third Implementation Results and Analysis 46

6 Testing **49**

6.1 First and second Implementation Testing 49

6.2 Third Implementation Testing 49

7 Conclusions and Future Work **51**

8 Bibliography **52**

9 Appendix **55**

9.1 Appendix A: Project Plan 55

9.2 Appendix B: Interim Report 57

9.3 Appendix C: Visualisation of RPCA 58

9.4 Appendix G: Code Listing 66

9.4.1 First Implementation of Ripple Consensus Protocol 66

9.4.2 Second Implementation of Ripple Consensus Protocol 86

9.4.3 Third Implementation of Ripple Consensus Protocol 95

9.4.4 Testing Code 108

Acknowledgements

I am deeply appreciative of my supervisors, **Dr. Paolo Tasca** and **Dr. Claudio J. Tessone** for their constant support and encouragement throughout the course of the project despite their busy schedule.

My main supervisor, **Dr. Paolo Tasca** motivated me to gain a general insight into the cryptocurrency space and admiration towards the field of Blockchain.

My co-supervisor, **Dr. Claudio J Tessone**, was extremely supportive in providing the relevant reading material and tools that would help in development and technical side of the project.

1 Introduction

1.1 Distributed Payment Systems and Digital Currencies

Technological advancements in payment systems have led to the advent of Bitcoin and other alternative digital currencies. When the traditional payment systems were first automated, the processes underpinning such systems were not significantly transformed. The key difference between traditional and modern payment systems has been the evolution in the underlying technologies employed for recording transactions and performing exchanges between two entities. Despite the involvement of novel technologies, the centralized structure constructing these modern payment systems was still retained. For example, mobile based payments have considerably reduced transaction time and improved accessibility to incorporate a wider range of users, however it is still dependent on central authorities such as banks for validating the transactions. More recently, there has been one significant innovation that has gained popularity over other innovations. Distributed Ledger Technology was introduced to allow payment systems facilitate transactions between two parties without the need for any intermediaries like banks by operating on a distributed peer-2-peer(P2P) network. The application of distributed ledger technology has had a considerable impact on how payment systems evolved structurally from being centralized to decentralized resulting in the emergence of Distributed payment systems and digital currencies.

Following the launch of Bitcoin [1], there has been an influx of alternative digital payment systems and their corresponding digital currencies. Some of these popular alternative digital currencies include Ethereum [2], Primecoin [3], Litecoin [4] Dash [5] and Ripple [6] among many others. Although these digital currencies are just a mere extension of Bitcoin with the primary aim of circumventing Bitcoin's downsides, they are ultimately derived from Bitcoin and share the same inherent structure apart from Ripple. For example, Ethereum extends bitcoin by offering Turing complete contracts. Primecoin operates on a different

proof-of-work involving prime number sets. Litecoin offers less time for block generation (approximately 2.5 minutes) as opposed to Bitcoin (approximately 10 minutes). Dash is based on complete anonymity and allows instantaneous payments without the need for any confirmations. Ripple is a distributed payment system that facilitates trade between different fiat currencies such as USD, GBP and Euro unlike Bitcoin which has a payment system involving only its own currency. Most of these digital currencies are often referred to as 'crypto-currencies' as they rely on cryptographic techniques to validate transactions like Bitcoin. However, Ripple is one such exception which that is fundamentally different to Bitcoin and uses non-cryptographic techniques to achieve consensus.

1.2 Motivation behind the Problem

While most crypto-currencies have been explored thoroughly in the past owing to Bitcoin's popularity [7–12], there haven't been much studies surrounding Ripple's payment System and consensus algorithm. More Recently, Ripple, now known as Ripple Labs, has received some criticism on a certain assumption it takes in its consensus algorithm making it centralized and less viable to be implemented in the real-world[13, 14]. Another aspect that contributes against Ripple's claim of being decentralized is the fact that majority of servers or nodes participating in the consensus process are operated by Ripple itself since the real incentive behind an outsider to participate as a validator seems to be ambiguous.

Despite the criticism, Ripple remains to be one of the most popular digital payment systems ranked third in the world by market capitalization following Bitcoin and Ethereum [15]. Recently, Ripple partnered with major financial institutions like Bank of America Merrill Lynch, Santander, Standard Chartered and many others to form 'Global Payments Steering Group'(GPSG) which aims at creating a network of banks employing distributed ledger technology for carrying out high volume, low value global payments [16]. Furthermore, Santander collaborated with Ripple to become the first bank in the UK to implement blockchain-driven

technology for allowing cross border payments through an app [17]. These examples highlight the significance and impact that Ripple has made in the financial industry making It an important area for further study and research.

1.3 Aim and Objectives

The project aims at investigating and modelling the key building blocks and performance measures surrounding the underlying consensus protocol of the Ripple system. Through this study, we expect to gather valuable insights on the assumptions raising the controversy [13, 14] around the centralized aspect of Ripple. Although we first examine the consensus protocol solely based on the constraints specified in the Ripple white paper [6], it is important to acknowledge the reasoning behind the criticism and further probe this reasoning.

The Ripple Consensus Protocol is modelled using the concept of agent-based modelling inspired from that of Bitcoin. Prior to implementing an agent-based model for the Ripple system, we first gain an understanding of the agent-based model for the Bitcoin consensus protocol, developed by my supervisor Dr. Paolo Tasca and co-supervisor Dr. Claudio J. Tessone. Subsequently, the Ripple network is re-constructed using an agent based model with the aim of making it as close to the real implementation as possible. Following the implementation of the simulation model, analysis of the same is conducted to determine how the model captures the dynamics of the real system and how the network performs under different circumstances taking a range different values for the building blocks.

1.4 Approach Overview

The general approach followed over the course of this project was an iterative-based approach and can be split into 4 stages.

Stage One

The first stage predominantly comprised of a little introduction into the field of crypto-currencies and Blockchain. This stage was intended to familiarize myself with the underlying terminologies that are widely encountered in the digital currency and Blockchain space. Subsequently, more emphasis was put into the study of two crypto-currencies, Bitcoin and Ripple given their heavy usage in the upcoming stages.

Stage Two

On getting a thorough idea into the two crypto-currencies, the second stage dived right into the consensus mechanism used in Bitcoin and how it has been modelled by my supervisor and co-supervisor. This stage involved an in-depth understanding and analysis of the implemented agent-based model which was presented to my supervisors.

Stage Three

Following this stage the third stage primarily covered Ripple and its consensus protocol mechanism. The core part of the project was executed in this stage including the implementation and analysis of the existing agent-based simulation in Python, construction of the revised Ripple network topology and finally the integration of this revised topology with the agent-based model to generate multiple simulations.

Stage Four

The fourth and final stage of the project incorporated an evaluation of the simulation results obtained from the previous stage and documentation of the entire report.

During the initial stages of the project there was constant communication delivery, and feedback from the supervisors to gain a strong foundation on the background and scope of the project. As the project progressed in its later stages the interaction got less to encourage less dependency on the supervisors since the deliverables of the project were clear

and primarily involved implementation of the Ripple consensus process.

1.5 Report Structure

The remaining part of the report is structured as follows. Section 2 section introduces the terminologies, consensus protocol components and algorithms, building blocks and potential risks involving the Ripple and Bitcoin payment systems. Furthermore, this section also discusses briefly the criticism surrounding the Ripple network and certain ambiguous assumptions taken in its consensus protocol. Section 3 conducts simulation studies by first formulating the concept of agent-based modelling used for constructing the existing Bitcoin agent-based model. We then present a formulation for the agent-based model simulating the Ripple network and subsequently gathering certain initial requirements. Section 4 builds on top of the previous chapter by further probing those requirements to build an improved model taking into account the inadequacies of the existing implementation. Section 5 compares and analyses the results of the simulation models on the basis of performance metrics that were obtained. Section 6 summarizes the project in general and illustrates the conclusions. An appendix is also provided in the end following the report which includes the initial project plan, interim report,a system manual, the code implementation and the results log from the simulations of the Ripple agent-based model.

2 Background Research and Related Work

Prior to gaining an understanding of the Ripple system, how it achieves consensus and how we will be looking to model the system, it is important to review the Bitcoin system, its consensus protocol and how this protocol is simulated using agent-based modelling. Subsequently, this section seeks to motivate our project by describing the Ripple system, the relevant terminologies and building blocks which lay the foundation of the Ripple network and the underlying consensus protocol that makes it fundamentally deviant from Proof-of-Work and other Bitcoin-derived consensus protocols. Lastly, the section summarizes the potential challenges faced by consensus systems in general and how Ripple overcomes these problems given its current protocol.

2.1 Bitcoin

Bitcoin can be thought of as a combination of various definitions and concepts which eventually relate towards forming a digital environment that allows transactions between different entities. Fundamentally, the term Bitcoin can be defined as either a unit of account such as euro, an electronic payment system, a decentralised network or a consensus protocol.

The main purpose of bitcoin as a currency unit is essentially to quantify, store and transfer value between different parties within the network. This in turn also justifies the definition of bitcoin as an electronic payment system with transfer of an electronic currency taking place. However, the underlying Bitcoin network and consensus protocol driving the digital payment mechanism is a bit more complex in terms of their structure and function.

2.1.1 Bitcoin as a Distributed Network

Bitcoin is built on top of a peer-2-peer decentralised network which lay the basis for Bitcoin as a consensus protocol. Prior to discussing the Bitcoin protocol, it is important to define some core ingredients responsible for constructing the network.

End Users

People who send transactions to other users or receive transactions from other users and vice versa.

Miners

People who participate in the consensus process and are ultimately responsible for performing the users' transactions in blocks.

Validators

People who consistently monitor the miners to check that the added transactions are sound.

Blockchain

A decentralized ledger which is shared among all miners in the network and gets updated with new blocks of transactions regularly.

Mining

A process routinely carried by all miners which requires them to solve by brute force a cryptographic function demanding high computational power. Some miners also chose to combine their computational power to increase their chances of solving the function by forming mining pools.

Gossiping

A process again carried out by all miners where they communicate with their respective neighbours to broadcast a new block.

Hashing Power/ Hash Rate

Every miner has a certain hashing power that determines the miner's ability to compute the hashing function.

Gossip rate

The rate at which a miner communicates with its neighbours about a new block.

Blocktree

A tree with its braches representing chains of blocks because of the occurrence of forks with evolution in time. Each node has its own internal blocktree i.e. constructed by all the blocks it received during this evolution of the system. The longest chain inside of this block tree is what defines the block chain for this node.

2.1.2 Bitcoin as a Consensus Protocol

The main role of the bitcoin consensus protocol, also known as Nakamoto consensus, is to achieve consensus such that each miner agrees on the same block(s) to be applied to the blockchain. This is carried out by the miners through the process of mining as follows.

1. The difficulty of solving the cryptographic function by brute force is adjusted in such a way that given the computational power of all the miners inside the network the mean time it takes for all these miners to find a new block is roughly 10 mins. This holds true regardless of the changing number of miners.
2. To determine which miner gets to add this new block, each miner has a certain probability of being the winning miner depending on its hashing power.
3. Subsequently, when a miner mines a new block, the miners communicate and gossip with other miners about the new block.
4. The block propagates and is added to all the internal blocktrees' corresponding to nodes that it reaches. However, it has been proved, mathematically that in certain conditions

this implies that if 50% of the network has included a new block in their internal blocktree then a consensus will be reached for that block and the block will be included in the shared blockchain neglecting all the other internal blocktrees.

2.2 Ripple

Like Bitcoin, Ripple is a payment system, a distributed network and a consensus protocol but the difference lies in the medium of transfer between the users and the dynamics by which consensus is reached. Although Ripple uses a native currency XRP, it plays the primary role of paying transaction fees and stimulating liquidity within the system and rather than a medium of transfer between its users. Now we will look at the role of Ripple as a payment system, a distributed network and a consensus protocol respectively.

2.2.1 Ripple as a Payment System

Ripple, when thought of as a payment system, allows its users to transfer all kinds of fiat currencies such as USD, GBP etc. across the world, supports cross-asset payments and uses a native currency XRP to facilitate these transactions.

2.2.2 Ripple as a Distributed Network

The Ripple network is distributed peer-2-peer network with some added constraints surrounding its connectivity to support the consensus mechanism in Ripple. Some core ingredients constructing the Ripple network topology are as follows:

Server
Nodes or Agents participating in the consensus process.

Ledger

Common Ledger between the all servers which gets updated with transactions which pass the consensus process.

Last Closed Ledger

Current stable state of the network or the most recent ledger validated by the consensus process.

Open Ledger

Current operating status of a server (each server maintains its own open ledger). Transactions are added to the open ledger but not considered final until they pass the consensus process.

Unique Node List (UNL)

Each server has a UNL i.e. a set of other servers that this server can communicate with to reach consensus.

Proposal

Set of transactions to be considered to apply on any ledger (from UNL). Proposals from servers not in the UNL are simply ignored.

Proposer

Any server can volunteer to broadcast transactions included in the consensus process. During the consensus process, only proposals from servers on the UNL of a server are considered by the server.

2.2.3 Ripple as a Consensus Protocol

Ripple operates on a consensus protocol involving collectively-trusted cliques or subnetworks such that the minimum trust is required from these subnetworks to reach consensus. This protocol can be understood of as a distributed agreement protocol in which agreement be-

tween the subnetworks is sufficient to determine the transactions to be added onto the ledger and achieve consensus. Accordingly, the Ripple Protocol Consensus Algorithm(RPCA) is iterated for a few rounds until consensus is achieved. The next subsection enlists the individual processes involved in the Ripple Protocol Consensus Algorithm. A visualization of RPCA has been provided in the Appendix under 9.3

2.3 Ripple Protocol Consensus Algorithm (RPCA)

Given that the goal of the consensus if for each server to apply the same set of transactions to the last closed ledger, the RPCA proceeds in the following steps:

1. Servers receive transactions from end-users continuously. These transactions form a candidate set (A pool of transactions waiting to be added to the ledger).
2. At the same time, server receives proposals from other servers in the network. There are simply, sets of transactions to be considered to apply to the ledger. Proposals from servers outside of the UNL are rejected.
3. In case the incoming proposal is from a server in the UNL, the transactions in the incoming proposal is compared against the candidate set. When a transaction in the incoming proposal matches a transaction in the candidate set, that transaction receives one vote.
4. The server continues to check incoming proposals against the candidate set until the timer expires. Once the timer expires, the server takes the transactions that have received at least 50% approval rating and packages them into a new proposal. This proposal goes out to the other servers in the network.
5. The process now repeats but with a different approval rating which has been increased to 60%. The server again continues to receive proposals and compares them against the candidate set until the timer expires again.

6. Now transactions with an approval rating of 60% or more are packaged into a new proposal and sent out to the network. Subsequently the approval rating rises to 70%.
7. With each such iteration, the proposals will have a greater similarity and the votes will increasingly agree.
8. As the dissimilarity tends to moves towards 0, the network has reaches consensus when 80% of the votes for every transaction in the proposal are either a "yes" or "no".
9. When the proposal hits the 80% threshold, the server validates the proposal, alerts the network, closes its consensus process and resets the approval rating threshold to 50% for the next consensus round. Any invalid transactions are discarded at this stage and a new Last closed ledger is created.
10. Transactions not taken in this ledger stay in the candidate set while all transactions received during the consensus process are added to the candidate set and the process starts again.

Although these steps encapsulate the consensus mechanism in Ripple, it is important to realise that the protocol takes into consideration certain assumptions that guarantees the consensus algorithm to achieve consensus on a set of transactions and do so in a trustworthy way. These assumptions are a consequence of certain challenges that distributed payment systems like Ripple and Bitcoin encounter and it is of the utmost importance that these payment systems overcome these challenges else the purpose of the payment system could be put into serious jeopardy. For instance, there is no point of a distributed payment system that allows a user to maliciously send the same value to two separate accounts, also called the double-spend problem. The next section describes such challenges, how Ripple overcomes these challenges and what makes the Ripple consensus algorithm different from other consensus algorithms.

2.4 Soundness of the Ripple Consensus Protocol

After close examination of the consensus protocols, there might be some immediate questions such as how the consensus systems deal with the double-spend problem or how to stimulate trust within the network as unlike traditional banking systems the users will remain anonymous to each other. Another potential question could be concerning forks whether the system can handle conflicting transactions that need to be applied to the ledger. Most of the former challenges have been previously explored in Byzantines Generals problem [18].

2.4.1 Byzantines Generals Problem

Byzantines Generals problem(BGP) is a hypothetical situation that can be imagined with a group of generals surrounding the enemy territory such that each general controls a part of army and all generals must co-ordinate an attack on the enemy territory by sending messengers to each other. However, there can be some miscommunication among the generals due to some generals being corrupt or the messengers not reaching their destination. The Byzantines Generals problem is synonymous to a Distributed payment system as follows.

Byzantines Generals Problem	Distributed Payment Systems
General	Node or end-user
Messenger may fail to reach their destination because the generals are in an unfamiliar territory and the messengers might get lost in the way.	Data may fail to reach the destination nodes.
Some of the generals may turn out to be traitors or conspiring together. Messages may have a false plan resulting in miscommunication between the loyal generals.	Malicious nodes (individually or mining pools) may attempt to convince the system into accepting fraudulent transactions or double spending transactions.

Table 1: Comparison between BGP and Distributed Payment systems

So, in conclusion Distributed payment systems must be robust in handling:

- **The standard failures**: These include maliciously increasing value.
- **Byzantines failures**: These include multiple sources in the network colluding with each other. Byzantines failures can also act as a measure of robustness of a consensus algorithm by estimating the tolerance of the network in the advent of malicious nodes or faulty processes. [18] verifies that no solution to the Byzantines Generals Problem can tolerate more than 33% malicious nodes in the network or $\frac{n-1}{3}$ byzantine faults.

To answer how Bitcoin overcomes the challenges proposed by the Byzantines General Problem is outside the scope of the project, however, it is crucial to understand how Ripple resolves these challenges as the model for the Ripple system needs to have the suitable assumptions that take this into consideration.

The ripple whitepaper (Schwartz et al., 2014) classifies the challenges encountered by distributed payment systems into three categories namely correctness, agreement and utility.

2.4.2 Correctness

Correctness with respect to a payment system can be defined as the ability of a system to distinguish between a malicious and a trustworthy transaction. In other words, correctness is representative of trust within the network. Unlike conventional payment systems, trust between 2 parties is non-existent as the identity of all users in the network is unknown. This suggests the need for trust in the network such that the system prohibits any fraudulent transactions to take place.

Ripple system achieves correctness from the assumption that a transaction is applied to the ledger only when 80% of the servers in a UNL approve the transaction. This follows that there can be no fraudulent transactions are applied to the ledger given that 80% of

the servers in a UNL are honest. In terms of the Ripple protocol strength, the system can tolerate only 20% malicious nodes in a UNL or $\frac{n-1}{5}$ Byzantines faults [6].

2.4.3 Agreement

Agreement follows from correctness in the sense that even though the system does not authorize any fraudulent transactions there could be a possibility in which multiple correct transactions independent of each other combine to fabricate a malicious act altogether such as the double spend problem. Another slight variation of this problem could be in the advent of forks when multiple correct transactions that conflict one another are to be applied to the ledger at the same time and the system should determine which transaction to consider

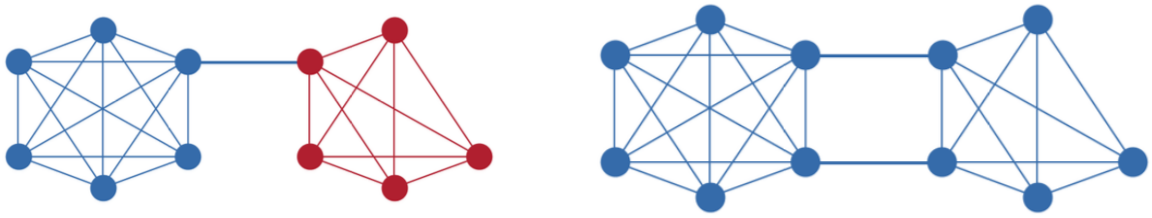
Ripple answers the former question regarding the double spend problem by warranting that all servers in the Ripple network apply the transactions deterministically. For instance, if two or more correct transactions that combine to perform a double spend act are approved by the consensus protocol, the second transaction would only take place after the first transaction has been completed. So, following the first transaction if there are no more funds available in the account the second transaction would fail [6].

To resolve the second problem concerning forks in the system being caused due to two different UNL's achieving multiple correct consensuses at the same time, the Ripple algorithm needs to make sure that there is a certain amount of overlap between all UNL's to prevent such forking incidents. [6] claims that forks can be prevented in the network by restricting the connectivity of the network such that:

$$\boxed{\mid UNL_i \cap UNL_j \mid \geq \frac{1}{5} max(\mid UNL_i \mid, \mid UNL_j \mid) \quad \forall i, j}$$

However, the above restriction is considered under the assumption that the underlying network takes the form of a graph comprising of cliques that are connected to each other. Each

clique can be imagined as a UNL such that each server contains all the other server of the clique in its UNL. Subsequently, [6] guarantees the non-existence of forks if there is sufficient overlap between any two cliques and this overlap must greater than or equal to $0.2 * n_{total}$ where n_{total} is the total number of servers in the two cliques as shown in Figure 16.



(a) Disjoint UNLs that fork **(b) Sufficient overlap that prevents fork**

Figure 1: An example of fork and non-fork situations [1]

It was this restriction surrounding the overlap of UNL's that received criticism from various papers [13, 14] and will be looked at in the following subsections.

2.4.4 Utility

A payment system that satisfies both correctness and agreement could suffer from high latency or lag. Therefore, it is vital that the consensus process is terminated in quantifiable time else the system could be extremely inefficient and not serve as a feasible solution. Another aspect of utility could be the level of computing power required by a user to avoid getting some value illegally or perform illegal transactions. For instance, in bitcoin, 51% of computing power could lead to double spending or fraudulent transactions being accepted. Fortunately, in case of Ripple, the only harm inflicted on the system if a malicious server(s) controls more than 20% of the UNL would be to delay the forward progress. This would result in the exclusion of the malicious server(s) from the UNL of the honest servers.

To satisfy the latency aspect of utility, [6] proves the convergence of the Ripple Protocol Consensus Algorithm by monitoring the time of response for all servers. Given that consensus is reached in 't' number of rounds, if the response-time for a server exceeds a bound 'b' the

server is automatically disconnected from the all UNLs to ensure that consensus terminates in finite time with an upper bound '$t * b$'.

2.5 Uniqueness of the Ripple Consensus Protocol

Although there are existing algorithms that solve the Byzantines Generals Problem, one common drawback that most of these algorithms suffer from is high latency arising due to the requirement that all nodes in the network need to be communicating synchronously.

Ripple consensus algorithm overcomes the issues concerning high latency by exploiting the collectively-trusted subnetworks within the larger network and removing any nodes that tend to slow forward progress. However, to achieve consensus the minimal trust and connectivity is required. Therefore, unlike other consensus algorithms, the Ripple Consensus Algorithm is designed to be a low latency algorithm whilst being tolerant to Byzantines failures.

2.6 Criticism faced by Ripple Labs and its Consensus Protocol

The criticism encountered by Ripple Labs is two-fold in the sense that there were accusations surrounding both the business side of the company and the technical side concerning its consensus protocol. While the controversies relating to the business side of Ripple are not of much significance to this project, there is however one conflicting aspect concerning their valuation of market capitalization since it was mentioned in Chapter 1. Although Ripple initially created approximately 100 billion Ripple tokens but there is only a fraction of this amount (approx. 37 billion) which is being currently circulated [19]. Therefore, there is no real understanding on how to estimate their market capitalization. [20] lists the market cap of Ripple taking all the total supple while the [15] only taken the amount of Ripple tokens in circulation. This can be perceived as a common scam to artificially drive the market cap.

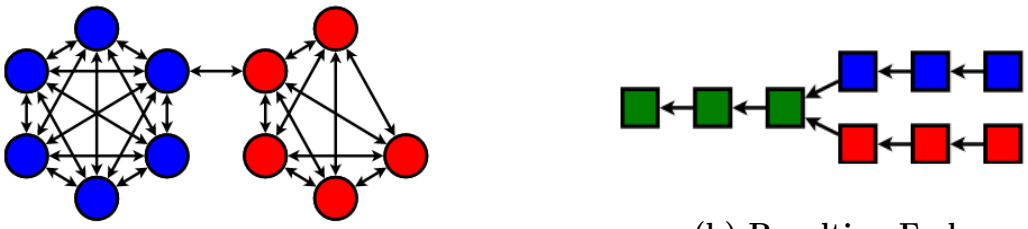
The more significant criticism which directly relates to our project is concerning the Ripple

consensus protocol. [14] extensively provides an analysis of Ripple and its consensus protocol taking the Ripple whitepaper as a reference. Some important points that were discussed in [14] questioning certain aspects of the Ripple protocol can be summarised as follows:

1. **Existence of Ripple XRP:** As specified in the Ripple white paper, XRP token is primarily used in Ripple for anti-spam purposes by raising the transaction fees. This could lead to a scenario where a malicious user can intentionally drive up the transaction fees to diminish the Ripple user base.

2. **UNL size:** UNL for a validating server is a subset of public keys of servers in the network that the validating server trusts and believes are unique. Although Ripple Labs suggest that there needs to be more than 100 servers in the UNL they publish a set of only 5-8 public keys for a new server who volunteers to participate in the consensus process. To add to this, Ripple currently only has around 105 nodes and it would seem impractical to add all these nodes in the start given that you initially have limited or in fact no knowledge about most of these nodes.

3. **UNL overlap:** The Ripple whitepaper states that it is essential for any two UNLs to have sufficient overlap of at least 20% to ensure that no two UNL's arrive at a different consensus that each other. However, it cannot be guaranteed that two UNL's will always be overlapping sufficiently resulting in the possibility of forks.



(a) Insufficient overlap

(b) Resulting Fork

Figure 2: Insufficient overlap between two UNLs leading to a fork condition [1]

[0]Source: Ripple Protocol Consensus Algorithm Review (Todd, 2015), [14]

4. **Fork handling:** Following the previous issue, there is no mechanism put in place that could effectively re-organise the ledger on detection of longer forks. Currently, when a fork is detected in the Ripple network one of the sides must discard all its history and transactions leading to double-spends and heavy financial losses.

5. **Lack of Incentive:** There is no apparent means to incentivize any new servers for participating in the consensus process unlike mining other than explicitly telling them to do so in exchange for money. Although there is some transaction fee but if the validating nodes misjudge the consensus it could lead to potential losses and no one would voluntarily take that amount of risk unless there is some additional incentive.

6. **Centralized protocol:** If no new servers are willing to volunteer as a validating node and take on the added risk they could simply trust the default servers provided by Ripple believing that other new servers would have also gone with these default servers. In this scenario, the entire consensus protocol would be controlled by Ripple itself making it a centralised system.

Most of these accusations against Ripple goes to show that Ripple makes certain assumptions in its consensus protocol that could practically cause regulatory problems and violate its existence as a decentralised consensus protocol.

Another matter of concern consequential to their assumption of at least 20% overlap in all UNLs was further explored in [13]. This paper claims that even in the situation where the overlap between the UNL's is 20% there could still be a possibility of forks and subsequently suggest a revised assumption of this overlap being more than 40%. The proof was formulated as follows:

Ripple suggests,

$$\boxed{\mid UNL_i \cap UNL_j \mid \geq \frac{1}{5} max(\mid UNL_i \mid, \mid UNL_j \mid) \quad \forall i, j} \tag{1}$$

This formula could be re-written as:

$$\boxed{\mid UNL_i \cap UNL_j \mid \geq \theta_{i,j} max(\mid UNL_i \mid, \mid UNL_j \mid) \quad \forall i, j} \tag{2}$$

Where $\theta_{i,j} = 0.2$

Now if we consider a scenario where $\mid UNL_i \mid = \mid UNL_i \mid = 5$ and $\mid UNL_i \cap UNL_j \mid = 2$. The above conditions (1) and (2) hold for this as $\mid UNL_i \cap UNL_j \mid = 0.4 max(\mid UNL_i \mid, \mid UNL_j \mid)$. Now we assume that one of the common server's votes for a transaction T_1 to be included in the ledger and the other conflicting server votes T_2 to be included instead of T_1. Now, we assume that all servers in UNL_i but not in UNL_j (remaining 3 servers in UNL_i) vote for T_1 to be included and all servers which are in UNL_j but not in UNL_i (remaining 3 servers in UNL_j) vote for T_2 to be included in the ledger. Therefore, we find ourselves in a fork situation where 80% of UNL_i vote for T_1 and 80% of UNL_j vote for T_2.

So, the revised condition for prevention of forks should be $\mid UNL_i \cap UNL_j \mid > 0.4$.

Considering some if not most of these issues, we will look at the Ripple's consensus protocol from a more practical standpoint by simulating the protocol and testing the outcome of consensus by first investigating the default scenario provided in Ripple whitepaper and then adding constraints on top to even the variance in the outcome.

2.7 Tools Employed

To construct the Ripple consensus protocol, there were several tools employed keeping in mind the ease with which the agent-based simulation can be set-up ran and visualised. Another aspect considered was efficiency in the construction of the model since we are looking to run the model on many different values.

2.7.1 Python 3.0

As a pre-requisite to this project, we looked at Bitcoin's agent-based implementation in depth which was developed in Python 3.0 and having fully understood the simulation code it was established that Python 3.0 could also be extended in Ripple's scenario following the general skeleton used in Bitcoin. There was additionally a combination of various features in Python that encouraged the use of Python in agent based simulations. Python3.0 provides an extensive standard library that encompasses many popular programming tasks thereby increasing productivity and substantially reducing the programmer's effort. The language also has a straightforward syntax that is easy to understand and offers data structures like dictionaries and lists with the added support of dynamic dictionaries which is crucial in the implementation. Lastly, the enormous resources available for Python resulted in significant reduction of coding time.

2.7.2 NetworkX

Python library NetworkX was employed for network construction, visualisation and manipulation. The software package provides mechanisms to create basic graphs and subsequently experiment with different network topologies. These topologies could also be visualised to check if they coincide with what is required. For instance, the library was extremely beneficial in investigating the topology of cliques and simultaneously checking though basic visualizations if the required overlap is taking place between the cliques. The basic graph construction could be extended to create more complex real-world graphs with many nodes and edges while retaining the underlying constraints.

3 Simulation Studies and Requirements Gathering

This chapter follows a similar structure as the previous chapter where we first investigated the Bitcoin system except in this chapter we dig a little deeper into how the consensus mechanism in Bitcoin is modelled using an agent-based simulation. This analysis of the Bitcoin agent-based model was essential in capturing the mapping between the real-world application and the simulation study. With adequate understanding of the Bitcoin consensus simulation, the chapter then closely examines the existing agent-based simulation of the Ripple consensus mechanism.

3.1 Simulation Study 1: Modelling Nakamoto Consensus

As previously mentioned in Section 1.3, the modelling of the consensus mechanism in Bitcoin was performed by my supervisors. The agent-based model constructed consequently was further reviewed as a part of this simulation study. The study involved understanding the simulation code and subsequently producing a description summarising the inherent agent-based model in the code. Moreover, the building blocks and performance measures or results used to construct and evaluate the model respectively are also captured from the study.

3.1.1 Nakamoto consensus Description

The Nakamoto model is a simplified agent-based model that replicates a certain restricted proportion the of actual implementation by only primarily focussing on how the miners operate within the network overlooking the users and validators.

The agent based model is built on top a distributed network which comprises of a fixed set of nodes 'k' which will be participating in the mining process. The number of connections

or degree of each node is dependent on the chosen network topology. Subsequently, the underlying distributed network can be created by considering the two network topologies:

1. Random Degree Sequence Graph $G_1 = (d, d, d \ldots d)$ i.e. list of 'k' nodes each having a degree 'd'.

2. Barabasi-Albert Graph $G_2 = (k, d)$ where 'k' is the total number of nodes in the network and 'd' is the number of connections that a new node is going to preferentially have with existing nodes that possess high degrees..

Although in a real setup the network size and degree varies with nodes entering and leaving the network, we assume the network size and degree to be fixed as the dynamics of how new nodes enter and leave the network is irrelevant for the model.

The bitcoin consensus protocol,also referred to as Proof-of-Work, is modelled by considering the two main processes of mining and gossiping. In case of mining, every node is given a random hashing power which is derived from the from either Power law distribution(H_{pl}) or Exponential distribution (H_{ex}).

1. Power law distribution(H_{pl}) can be defined as follows:

$$\boxed{P(x) = Cx - a, \text{ for } x \geq xmin}$$

Where, P is the power law function

C is the normalization constant

a is a constant

x is the random variable

2. Exponential distribution (H_{ex}) can be defined as:

$$\boxed{P(x) = \lambda e - \lambda x,\ x \geq 0}$$

Where, P is the exponential function

λ is the rate of the distribution i.e. $\frac{1}{mean}$

x is the random variable

Miners cannot join hash power from one unique mining pool so we keep them separated unlike reality.

This implies that the rate at which a node mines a new block also called the mining rate λ_m is randomly distributed and can be calculated as follows:

$$\boxed{\lambda_m = \frac{1}{T_m}}$$

Where, λ_m is the mining event rate.

T_m is the mining time

Mining time is given by:

$$\boxed{T_m = \frac{Diff * 232}{H}}$$

Where, $Diff$ is the underlying difficulty for adding a new block at a given time.

H is the Average Hash rate of network at a given time.

For gossiping, whenever a node comes across a block it compares the length of the chain of blocks that it witnessed to the lengths of the chain of blocks witnessed by its neighbours and it subsequently tries to gossip with a neighbour at a certain rate λ_g known as the gossip rate. λ_g is assumed to be constant for all the nodes in the network. In case of a collision, when one node wants to transmit a block such that the length of the internal blocktree seen by its neighbours is the same as the gossiping node then no action takes place and the node must wait for another block until the blocktree lengths mismatch. These two processes are

iterated multiple times till the running time of the model is reached. At this point we record the resulting global blocktree and extract certain elements like longest branch or chain as the blockchain, number of branches, the number of discarded block called orphaned blocks and the total number of blocks.

3.1.2 Nakamoto Model Building Blocks

The building blocks are the core ingredients that are required to construct the agent-based model. Following the study of the simulation code and Nakamoto model, a number of building blocks can be deduced from the model:

1. **Simulation Time:** Number of days over which we want the simulation to take place.
2. **Network Size:** Total Number of nodes in the network that are participating in the consensus process.
3. **Network Degree:** Degree of the nodes depending on the Network topology as discussed in 3.1.1.
4. **Network Topology:** The topology of the network which could either be a Random Degree Sequence Graph or Barabasi-Albert Graph.
5. **Network Delay:** Time taken to deliver a block from one node to another.
6. **Hashing Power Distribution:** Distribution could be either Power Law or exponential.
7. **Block or Mining Interval:** Time interval between addition of a new blocks to the blockchain.

3.1.3 Nakamoto Model Performance Measures

Performance measures are the resulting outcomes that we intended to produce following the execution of the agent-based simulation.

1. **Number of Blocks:** The total number of blocked that were mined over the course of the simulation.
2. **Number of Orphaned blocks:** Orphaned blocks are those blocks which were mined but not attached to the main Blockchain. This situation tends to take place during forks when the longest chain wins and one of the blocks is discarded, which is the orphaned block. This measure gives the total number of Orphaned blocks obtained after the simulation is ended.
3. **Orphaned Block Rate:** This metric gives the number of orphaned blocks over the total number of blocked mined.
4. **Number of Branches:** Total number of branches that are formed on occurrence of forks.
5. **Longest Branch:** The branch that forms the longest chain in the Bitcoin network, which is eventually considered for mining. This branched is termed as the 'Blockchain' while all the other branches are ignored.
6. **Average Propagation time:** This is the average time it took for all nodes in the network to be informed about a new block.
7. **Maximum Propagation time:** This is the maximum time it took for all nodes to be informed about a new block.

3.2 Simulation Study 2: Modelling Ripple Consensus Protocol

The second part of the simulation study involved the investigation of the Ripple consensus simulation which is open source on Github and provided by Ripple[1]. The simulation code on Github has been developed in C++ and for the purposes of this project was re-developed in Python 3.0. The Python version of the simulation code has been provided in 9.4.1.

[1]Github repository: `https://github.com/ripple/simulator`

Consequently, a description of this model was documented and the underlying Building Blocks and expected performance were captured as initial requirements.

3.2.1 Ripple Consensus Model Description

Like the Nakamoto model, for the Ripple Consensus model is also a simplification of the real implementation which translates certain elements of the actual model that are significant for our research. As our project primarily revolves around the consensus mechanism of Ripple we only focus on the operation of the servers within the network overlooking its users.

The Ripple agent based model is built on top a random network comprising of a fixed set of servers 'k', each having 'd' number of outbound connections. The number of trusted servers in the UNL for each server and the response time of the node is derived from a uniform distribution whose density function within an interval ul and ub, can be defined as follows:

$$U(x) = \begin{cases} \dfrac{1}{ub - lb} & \text{if } lb \leq x \leq ub \\ 0 & \text{otherwise} \end{cases}$$

Where, ul and ub are lower and upper bound respectively.

Unlike the real setting where these values could be dynamically changing, the model is constructed under the assumption that these values are fixed once they are set given that the conditions in which the new servers enter the network, existing servers leave the network, servers modify their trusted servers and response times is not relevant for the study of the core consensus protocol.

The consensus protocol is modelled by first initializing the network with 50/50 consensus split meaning that 50% of the servers agree with a set of transactions to be added onto the ledger while 50% of the servers disagree with the set of transactions to be added onto the

ledger. Then the model triggers all servers in the network to broadcast the initial set of messages that comprise of their position concerning agreement or disagreement with the set of transactions to be considered for the ledger. With the initial broadcast of messages taking place, the response times of these transmissions are recorded and sorted in an ascending order as the simulation would be performed in that order. To better understand why this sorting takes place, response time can be thought of as the time at which a message is expected to be delivered. So, a message with lesser response time will be delivered prior to the messaged with a greater response time.

Once the initial positions of the servers are recorded by all servers the simulation starts as the transmission of the updated messages containing the modified server positions after the initial broadcast takes place. The simulation proceeds with those message transfers that have a less response time than the other messages. Once the messages having a certain response time have been sent, they are erased from the system and replaced by the new set of messages containing the updated server positions and updated response times. The process carries on with more and more messages being queued in the system.

As the delivery of messages is taking place with time, each server also updates the positions of the servers in its UNL if the incoming message requires the server to do so. This way the server keeps a track of the trusted UNL servers. If a certain threshold number of trusted UNL servers back either an 'agree' or 'disagree' position with the consensus, and the server contradicts with that position then it required to change its own position in accordance with its UNL. However, if the server is a malicious one then the server's decision is based on complete disagreement with all servers in its UNL. A bias in favour of disagreement with the consensus is added with increments as the simulation time advances.

Finally, the total number of serves that agree and disagree with the consensus are recorded at every stage in the transmission of messages. As soon as the network contains a super-majority of servers, taken to be 80%, that either agree or disagree on the consensus, the consensus process terminates with that decision.

3.3 Requirements Gathering

The simulation studies were performed as pre-requisites to gain an insight and hands-on experience with agent-based modelling. More importantly, the second simulation study revealed some initial core requirements in the form of inputs or building blocks to the Ripple simulation model and performance measures or results that are expected to be evaluated from the simulation.

3.3.1 Ripple Consensus Model Building Blocks

The following are the core ingredients essential for building the simulation model for the Ripple consensus mechanism:

1. **Network size:** Total Number of servers in the network that are participating in the consensus process.
2. **Network Degree:** Number of outbound links or connections for each server.
3. **Network Topology:** Completely random graph in which servers are randomly connected among one another.
4. **Network Latencies:** : These are of two types namely, end to core latency and core to core latency. End to core latency is time taken for a server to reach a decision and this latency is possessed by all the servers. Core to core latency is the added time it takes to transmit that decision to the connecting servers. Given these two latencies, the total latency along a link can be determined which is simply the sum of the two end to core latencies possessed by a pair of connected nodes and the core to core latency along the link they are connected.
5. **UNL size:** Number of servers in the network trusted by a server.

6. **UNL threshold:** Number of servers in the UNL of a server that could influence the decision of the server.
7. **Consensus Percentage:** The percentage threshold required by the network to achieve consensus.
8. **Malicious servers:** The number of malicious servers present in the network.

3.3.2 Ripple Consensus Model Performance Measures

The following are performance measures or results that we intend to obtain from the simulation of the Ripple consensus mechanism:

1. **Consensus time:** The duration over which consensus is reached by a super-majority (more than 80%) of servers.
2. **Positive servers:** The number of servers in the network that vote in accordance with the consensus.
3. **Negative servers:** The number of servers that vote in disagreement with the consensus.

It is important to note the above-mentioned ingredients or building blocks for the model is just a preliminary list that has been a directly gathered from the simulation study performed on Ripple. These requirements will be further revised in the next chapter as we look to extend and modify the existing implementation of this Ripple consensus model. Nevertheless, they serve the purpose of providing a good starting point that can be further investigated and improved.

4 Design and Implementation

This chapter seeks to consider and build on top of the present implementation of the Ripple Agent-Based model while maintaining the basic skeleton of this model. The chapter includes the design choices made along the way and the reasoning behind those choices. Moreover, implementation details have also been specified on how these design choices were incorporated in the simulations to deliver an improved agent-based model.

4.1 Design Choice 1: Current Ripple Simulation Model

The first design choice follows from the second simulation study of the preceding chapter as we look to evaluate the existing design of the Ripple agent-based simulation model. A description of how the model is currently designed has already been discussed in section 3.2.1 , so we will directly proceed to the implementation details of how this model has been implemented.

4.1.1 Implementation Technicalities

The agent-based simulation code for the Ripple Consensus Protocol is implemented taking 7 different classes namely Server, Link, Network, Event, Message, ServerState and Default-OrderedDict and a main function. Each of these classes and main function have their own distinctive purpose, instance attributes and methods that contribute towards the execution of the agent-based Ripple model described in 3.2.1

Server Class

The Server class is a representative of the agents or servers that make up the Ripple Network. A server object maintains a record of certain elements that are unique to a server such as

the id by which it is known, the end to core latency or time taken by a server to dispatch a message to another neighbouring server, a list of servers in its UNL, a list of servers that are directly connected to the server, time stamps of all servers that are known to this server (initially set to 0 for all servers except itself), positions of all servers that are known to this server (initially set to 0 for all servers except itself) , the number of messages that have been sent by the server and the number of messages that the server receives.

On receiving a message from other servers, the server updates its existing knowledge on servers' positions and time stamps and further broadcasts that updated information to the network. As the consensus protocol proceeds the server continues to receive more server positions some of which could also be about the servers in its UNL. Once the server has enough information on the servers in its UNL, it could choose to change its current position given that a certain threshold of UNL's servers hold a contradicting position from that of its own position. The position change is further broadcasted to the network along with its current knowledge of other server positions.

Link Class

The Link class is a representative of an outbound connection from a server to another server. A link object possesses attributes such as the id of the server to which the link is directed, the message that needs to be sent through that link, the total latency across the link which is taken to be a combination of the end to core latencies of the two servers and the additional core to core latency required to transmit the message across the link, the time at which a message is expected to be sent across the link and the time at which a message is expected to be received at the other end.

Network Class

The network class is responsible for the executing any message transfer across a link and updating the event log resulting due to the message transfer. The instance attributes possessed by a network object comprise of the current simulation time of the consensus protocol and a log of events that's need to be performed paired with the event times at which the events

are expected to be executed.

Once a message needs to be broadcasted, the network class is responsible for updating the attributes of the link through which the message is needs to be sent. The message is either appended to an existing event in the event log with the time at which the event is expected to take place or added to a newly created event in the event log in case the current event log does not have that event time.

Event Class

The event class maintains a list of messages that are need to be sent at a specific time. If a message is created that needs to be sent at a certain time, either an event object appends the message into its existing list of messages.

Message Class

The Message Class represents a message that needs to be transmitted across a link from one server to another. A message object is an abstraction containing the id of the server which sends the message, the id of the server to which the message needs to be sent and the message in the form of the sending server's knowledge on the server states of all servers in the network.

The receiving server updates its knowledge on the server states based on the message received and this updated server information is then broadcasted by the receiving server to the network. If the message from the sending server is directed to itself then it does not update its current knowledge on server states but the server still broadcasts the unchanged server information to the network.

ServerState Class

ServerState class captures the id of a server, its current time stamp and its current position which could either be 1 or -1 depending on whether the agrees or disagrees on a transaction to be added to the ledger.

DefaultOrderedDict Class

DefaultOrderedDict[1] is primarily used for conversion of the event ordered dictionary in which the keys are sorted to a default ordered dictionary so that key-value pairs can be dynamically added to the event dictionary and in ascending order. This is essential as we traverse the event dictionary in an ascending order of the time in which they are expected to be performed.

Main Function

The main initializes the Ripple network and runs the consensus protocol until either more than 80% servers reach an agreement on a transaction, or more than 80% servers disagree with the transaction or the consensus process reaches a state where there is no clear majority to determine consensus in which case the consensus process simply aborts. As the consensus process is terminated performance measures specified in 3.3.2 are captured.

The first implementation of the Ripple Consensus protocol in Python 3.0 has been provided in under the appendix 9.4.1 which illustrates how the above mentioned functionalities of all classes are being implemented to run a simulation of the protocol using the input values specified by the default Ripple simulator[2].

4.1.2 Design Problems:

After the initial implementation of the Ripple consensus mechanism, there were some immediate design issues which were apparent from the simulation code:

1. One of the major problems include the lack of simulations or input values considered for the consensus process with the current implementation just considering one range of values for the building blocks. Although the agent-based does not look to replicate the dynamically changing inputs such as Network size, there can be a range of values that

[1]Source: `http://stackoverflow.com/a/6190500/562769`

[2]Github repository: `https://github.com/ripple/simulator`

need to be considered to evaluate how the performance measures vary with different set of inputs or building blocks.

2. Another drawback which was directly implied from the existing design is the incompatibility of the input values for network size and UNL size with the real-setting. Ripple Charts, a website that gives a real-time visualization of the Ripple topology show around 100 servers on the network[1] while the simulation takes 1000 servers i.e. 10 times the number of servers in a real network. The same inconsistency of the numbers also hold true for the UNL size with Ripple providing only 5-8 servers[2] as UNL to a new server while the simulation take a range of 20-30 servers as UNL size.

In addition to the design problems mentioned above, there were a few more downsides to this design which will be considered in the next section.

4.2 Design Choice 2: Multi-threading and More Simulations

The second design choice aims at improving the previous design by considering the design flaws lists in 4.1.2. The first problem is resolved by simply considering a range of different input values and subsequently running multiple simulations. The second problem is implicitly solved by incorporating values that reflect the real-setting of the Ripple consensus protocol.

4.2.1 Implementation Technicalities:

The addition of several simulations led to another issue pertaining to the efficiency of running those simulations that had to be taken into consideration. Given that we were running hundreds of simulations, an iterative approach of execution where we run one simulation after another would not be viable as it would require a lot of time. So, there needed to be

[1]Source: `https://charts.ripple.com/#/topology`

[2]Source: `https://ripple.com/ripple.txt`

some synchronised way of processing the simulations keeping in mind that the simulations do not affect each other.

Multiprocessing with Locks

To introduce synchronisation to the simulations, Pythons 'multiprocessing' [1] library was imported which allows synchronised handling of simulations as processes, with the use of locks to prohibit any instance of simulation from affecting the other. Although this improves the processing time of the simulations to some extent but it is still not optimal as an instance of simulation must wait until it can acquire the lock.

Multiprocessing with Locks

For better optimization of these simulation processes, a 'Pool' object is created using the 'multiprocessing'[1] library that generates a pool of simulation processes with different input values and creating a mapping the input values with the outputs of these processes while parallelizing the execution of these processes. However, given our current set-up of the simulation the implementation of Pool object would yield the results in a way that simulations would interrupt each other because of the presence of print statements in each simulation instance that are not 'thread-safe'. To avoid this mix-up of results, all print statements were removed from the simulation with all results being returned as a list rather than being printed. Only once the results from all simulation processes are captured and mapped to their input values using the pool object, then the simulation results are sequentially printed along with their input values. sound

The second implementation is provided under appendix 9.4.2 and henceforth includes the execution of multiple simulations using the pool object to parallelize the execution. Since the core Ripple classes remain unchanged so they have not been included in the second implementation.

[1]Source: `https://docs.python.org/2/library/multiprocessing.html`

4.2.2 Design Problems:

Although the multiple simulations give a more complete representation of the Ripple consensus protocol but we have yet to explore the underlying network on which this protocol is currently running. As we can recall from sections 2.4.3 and 2.6 for the ripple network to prevent any forks and avoid double spends, the UNL's of any two servers need to be having at least more than 20% or more precisely more than 40% overlap.

To check whether the above assumption of more than 20% overlap holds true, we could run a simple test iterating through all possible pairs of server UNL's and breaking if the overlap happens to be 20% or less. On testing the validity of the above assumption, it was found that the simulation contradicts this assumption of more than 20% between two UNL's. Furthermore, there was an additional test to check if there the overlap assumption holds between the UNL of a server and the UNL's of the servers in its UNL. The second test case generated similar results as first test with insufficient overlap between the UNL's. The results obtained from the test cases have has been specified in the next chapter under 6 and the test cases are listed under appendix 9.4.4.

The insufficiency of the overlap between UNL's can be inferred directly from the process with governs the construction of the underlying network topology network. Currently, the network is constructed in a completely random fashion by randomly assigning links and randomly allocating servers as UNLs which neither guarantees overlap between UNL's nor takes into consideration the assumption of a network of cliques as proposed in the [6].

4.3 Design Choice 3: Interconnected sub-network of Cliques

The third design choice builds on top of the previous design by overcoming the design issues stated in 4.2.2. The underlying network topology had to be transformed such that it represents a network of sub-networks that connected to each other in a way that sufficient over-

lapping takes place. Although there are some existing network topologies such as stochastic block models that could be applied but they would be unable to completely guarantee the overlapping assumption. Subsequently, a novel graph topology that completely captures the overlapping assumption was developed and implemented as the primary network topology.

4.3.1 Implementation Technicalities:

Stochastic Block Model

To better examine the overlapping problem, the Ripple network can be thought of as being constructed from a stochastic block model[1]. This model is typically used to generate social network graphs which are partitioned into many communities. Entities within the same community are more likely to have a connection with each other than two entities which belong to different communities. Social network graphs produced from stochastic block models are analogous to that of a Ripple network where each community can be translated to an instance of a 'UNL clique'. A 'UNL clique' is simply a set of servers in which each server contains every other server from that set in its UNL. The servers within a UNL clique are more likely to trust each other than the servers outside of this UNL clique because all servers in that UNL clique contain each other in their UNL while the same cannot be guaranteed for servers outside this UNL clique. This is equivalent to entities within a community having a higher probability of being connected to each other than entities outside that community.

Problems with Stochastic Model

Although stochastic block model for generation of Ripple network would seem like a perfect replacement to the random generation, there is one small caveat. Imagine a scenario where say we have two UNL cliques A and B each containing 5 nodes. To guarantee sufficient overlap there needs to be at least 2 connections (more than 0.2*5 = 1 connections) as per Ripple [6] or 3 connections (more than 0.4*5 = 2 connections) as per paper [13]. Although

[1]Wiki source: `https://en.wikipedia.org/wiki/Stochastic_block_model`

it is less likely but there could be the case where the 2 or 3 servers from UNL clique A are connected to only one of the servers of UNL clique B. This would give us the illusion that the two UNL cliques have overlapping but if for instance that server in UNL clique B breaks down suddenly we reach a situation where there is no overlap between the UNL cliques. To fully guarantee that the overlapping, a revised network topology was introduced which follows from the stochastic block model approach but this time we enforce that the overlapping takes place on distinct nodes between two UNL cliques.

The third and final implementation listed under appendix 9.4.3 includes the algorithm for producing the revised network topology of interconnected network of cliques used in the final simulation of the Ripple Consensus Protocol.

4.3.2 Design Problems:

Eventhough the revised network topology imitates the clique-like configuration of UNL's as the Ripple white paper suggested [6], but the stochastic block model gives a more accurate representation of how real-world complex networks can be simulated. Currently, the stochastic block model would result in the possible breakdown of the consensus protocol due to the creation of forks as explained in section 2.6. The revised network topology, on the other hand, enforces the overlap between the UNL's so that the underlying assumptions are met rather than relying on the natural occurrence of overlapping servers. This also justifies the argument against the clique-like configuration of UNL's by Peter Todd [14] that the model takes assumptions that are would deem as impractical upon implementation in the real-world.

5 Results Evaluation

This chapter showcases and discusses the results obtained as consequence from running the simulations of the three implementations. The results are then further evaluated and probed to develop a more concrete understanding of the underlying implementation producing these results. Finally, this chapter concludes specifying the test cases used to check the validity of the implementations.

5.1 First Implementation Results and Analysis

The simulation results generated from running the first implementation were as follows:

```
INPUT VALUES

1) Network Size : 1000
2) Malicious Servers : 15
3) Outbound Links : 10
4) (Min UNL size, Max UNL size, UNL threshold) : (20, 30, 10)
5) (Min e2c latency, Max e2c latency) : (5, 50)
6) (Min c2c latency, Max c2c latency) : (5, 200)

RUNNING SIMULATION

1) Initialing Servers
2) Initialing Links
3) Broadcasting initial messages
4) 271 Events created and 20000 Messages created
5) Running Ripple Consensus protocol (might take some time)

Time : 100 ms  Servers+/Servers- : 500/500
Time : 200 ms  Servers+/Servers- : 490/510
Time : 300 ms  Servers+/Servers- : 417/583
Time : 400 ms  Servers+/Servers- : 321/679

PERFORMANCE MEASURES

1) Consensus outcome: Majority of servers disagree on the transaction
2) Positive servers: 199
3) Negative servers: 801
4) Consensus time : Consensus reached in 460 ms with 74095 messages on the wire.
5) Message rate : An average server sent 175.464 messages.
```

Figure 3: Simulation results from Implementation 1

The simulation first prints the numeric values of the input parameter used for running the Ripple consensus protocol which were the same as the default values used by the Ripple simulator [1].. Then the simulation outputs the consensus algorithm steps as they are being executed by the simulation code. The number of events could vary depending on the number of distinct event times at which the events are expected to happen. Initially there are 271 events and 20000 messages in the Ripple network. Given that there are 1000 servers in the network and each server has 10 outbound links so each server can send 10 messages and at the same time receives each server can receive 10 messages through the same links thereby creating 1000*10*2 = 20,000 messages.

The execution of the core consensus protocol is illustrated by printing the number of positive servers or servers in agreement and negative servers or servers in disagreement after every 100ms where ms is taken to be a hypothetical time unit and not milliseconds. The simulation reaches consensus after 400ms and therefore displays 4 occurrences of positive and negative. At the start of the consensus protocol, the network is evenly split with 500 positive and 500 negative servers but these change as we progress with the protocol. Moreover, Initially, around 200 ms, there is very little change in the number of positive and negative servers (490/500) but then these numbers shoot up in no time. This happens because the servers in the network become increasingly aware of the server states in their UNLs with time which triggers the position change of these servers.

Finally, the results of the simulation are expressed by printing the performance measures of the consensus protocol. For this instance of the consensus protocol, the outcome of consensus results in the exclusion of a transaction with majority of servers disagreeing on the transaction. As the consensus process terminates, there were 199 positive and 801 negative servers implying that as soon as 80% of the network either agree or disagree with the transaction to be added to the ledger, the consensus is reached. More precisely, the consensus is reached in 460ms with 74095 messages remaining to be broadcasted. This

[1]Github repository: `https://github.com/ripple/simulator`

justifies the utility aspect of Ripple explained in 2.4.4 as the consensus protocol does not wait for the entire network to reach consensus unlike Bitcoin but just a super-majority of servers (80%). Following the consensus, the message rate or the average number of messages sent by each server is calculated which for this instance turns out to be approximately 175.

5.1.1 Second Implementation Results and Analysis

The simulation results collected from the second implementation can be split into different categories depending of on the inputs parameters that we were looking to vary while keeping the other parameter fixed. Due to the design issues stated in 4.1.2, the standard fixed values for network size is 100 servers and UNL range is 5-15 servers as they give a more accurate representation of the Ripple network.

5.1.2 Varying Network Size

This simulation analyses the effect of changing network sizes on the Ripple consensus protocol. With 5 outbound links and 10 malicious servers in the network, the Network size was made to vary from 100 to 1000 servers with a step size of 100. Subsequently, the resulting consensus times were captured and plotted onto a graph with the varying network sizes.

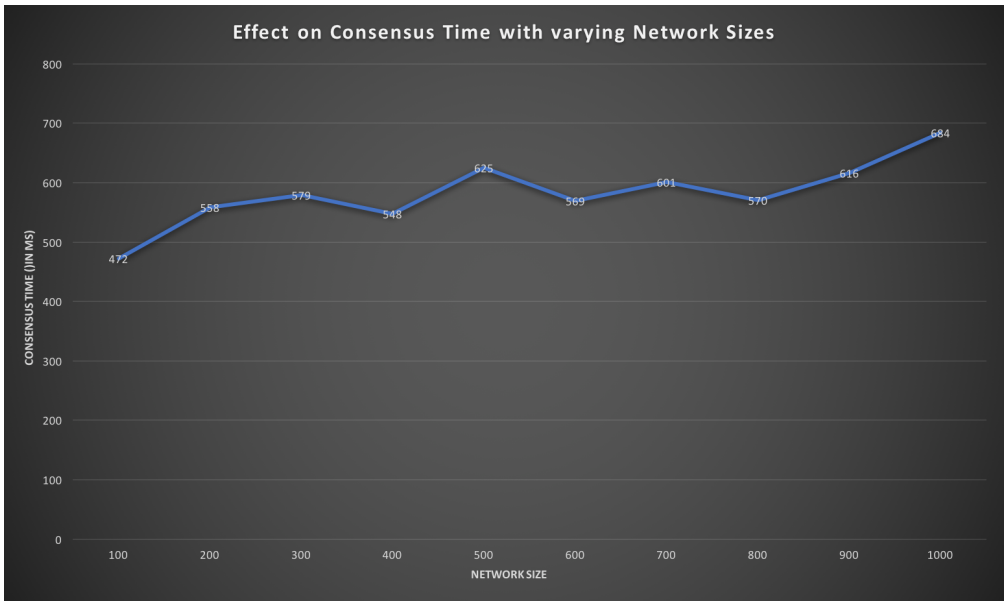


Figure 4: Effect on Consensus with varying Network sizes

The graph shows that generally the time taken to reach consensus increases with increase in the number of servers. As the network size increases, it can be implied that the servers take more time to update their knowledge on the existing servers in the network. This is not the case every time as the consensus times are also dependant on the inherent network topology which could either delay as observed with network sizes 500 and 700 the consensus or accelerate the consensus as witnessed with network sizes 400, 600 and 800.

5.1.3 Varying Malicious Servers

As the number of malicious servers have a direct impact on the breakdown of the consensus protocol, this simulation was designed to study the effect of changing malicious servers on the consensus protocol. They made to vary from 5 to 30 in step sizes of 5 keeping all other input parameters fixed to check if a break down takes place. Interestingly, the consensus protocol reaches a halting state when the number of malicious servers were 25 and 30. To closely examine its effect, the step size was reduced to 1 to see if check if the breakdown

could occur with lesser number of malicious servers but 25 proved to be the least number which triggers a breakdown. This shows that once the network is controlled by more than 25% of malicious servers, they tend to slow down the system by preventing the forward progress of the consensus protocol. These results coincide with what Ripple have stated in the whitepaper where the number of malicious server in the network need to be less the 20% to be Byzantine fault tolerant.

```
Time : 100 ms  Servers+/Servers- : 49/51
Time : 200 ms  Servers+/Servers- : 46/54
Time : 300 ms  Servers+/Servers- : 39/61
Time : 400 ms  Servers+/Servers- : 34/66
Time : 500 ms  Servers+/Servers- : 23/77
Time : 600 ms  Servers+/Servers- : 20/80
Time : 700 ms  Servers+/Servers- : 22/78
Time : 800 ms  Servers+/Servers- : 21/79
Time : 900 ms  Servers+/Servers- : 21/79
Time : 1000 ms  Servers+/Servers- : 20/80
Time : 1100 ms  Servers+/Servers- : 20/80
Time : 1200 ms  Servers+/Servers- : 20/80
Time : 1300 ms  Servers+/Servers- : 20/80
Time : 1400 ms  Servers+/Servers- : 20/80
Time : 1500 ms  Servers+/Servers- : 20/80
Time : 1601 ms  Servers+/Servers- : 20/80
```

(a) Running consensus protocol

```
PERFORMANCE MEASURES

1) Consensus outcome: No messages to send. Aborting Consensus.
2) Positive servers: 20
3) Negative servers: 80
4) Consensus time : Consensus reached in 1613 ms with 0 messages
5) Message rate : An average server sent 292 messages.
```

(b) Simulation results

Figure 5: Simulation results with 25 malicious servers

5.1.4 Varying Outbound Connections

To study how the consensus protocol is affected by the changing number of outbound connections, we vary the number of outbound connection from 5 to 30 in step sizes of 5 with 10 malicious servers and other parameters being fixed. An intriguing result observed from running this simulation was the consensus being triggered due a super-majority of positive servers. Prior to this simulation, there were almost no simulations that resulted in the attainment of consensus from the positive servers. This is justifiable as the simulation code introduces a time-dependant bias into the network favouring the negative servers as the simulation proceeds. This bias is added on purpose to alter the initial configuration of a perfectly even split of negative and positive servers to avoid halting situations that could result in the servers just backing their position as time progresses. With increasing number

of connections all servers get instantaneous updates of other server positions in the network and thus the time-dependant negative bias has a minimal effect on the consensus protocol. Instances of 25 and 30 outbound connections in the simulation resulted in the consensus due to the positive servers.

```
PERFORMANCE MEASURES

1) Consensus outcome: Majority of Servers agree on the transaction
2) Positive servers: 81
3) Negative servers: 19
4) Consensus time : Consensus reached in 365 ms with 17120 messages on the wire.
5) Message rate : An average server sent 328 messages.
```

Figure 6: Simulation results with 25 outbound connections

5.1.5 Varying UNL size range

There were two characteristics of UNL size range that were taken into consideration for this simulation. First the simulation was made to run on a smaller UNL size range that evenly increases from (10,15) to (30,40) in step sizes of 5. Then the UNL size range was doubled so that that increases from (5,25) to (30,50) in step sizes of 5. The variance in the UNL size ranges does not reveal any substantial evidence on any dependency with the consensus times given the randomised nature under which the UNL sizes are generated from these simulations.

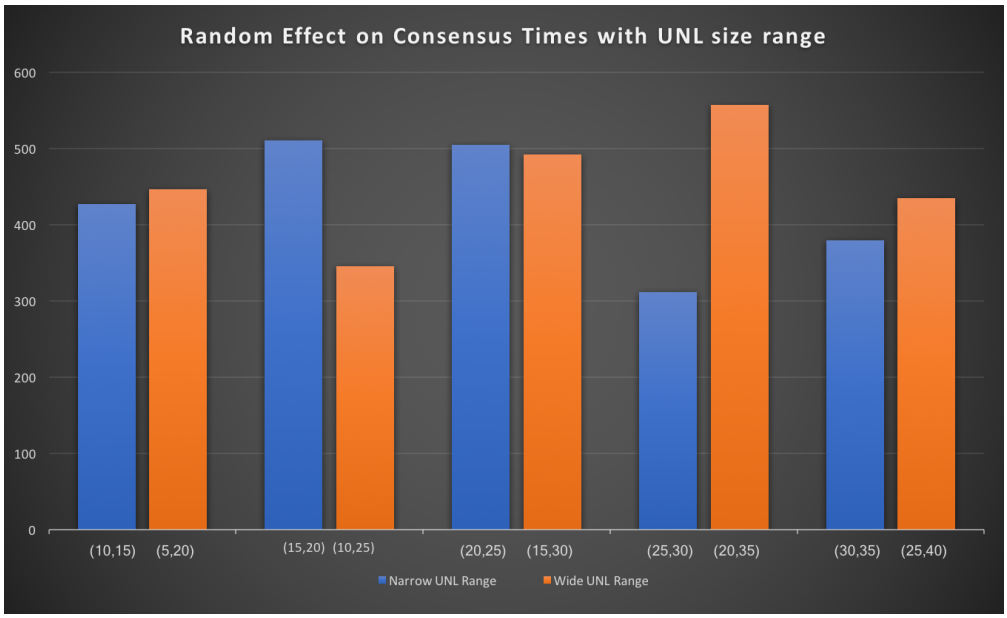


Figure 7: Consensus with narror and wide UNL ranges

5.1.6 Varying Network Latencies range

Like UNL size range, varying end to end core latency range and core to core latency range would not provide any deterministic dependency with the consensus times as the inherent means by which the Network latencies are generated is completely random.

5.2 Third Implementation Results and Analysis

The third implementation ran simulations of the Ripple consensus protocol on a revised network structure that represents an inter-connected network of cliques. The simulation results consider varying input values like the previous implementation and use multiprocessing to accelerate the generation of results. The revised network structure is much more restricted than the random networks used in earlier implementations so a smaller subset of input values are taken for the simulations. There was a total of 64 simulations that were run and the

results gathered from these simulations. The following is an example simulation run from the 64 simulations that were executed:

```
Simulation 1 :
--------------------------------------------------------------------------------

INPUT VALUES

1) Network Size : 100
2) Malicious Servers : 10
3) Network Topology (1: Random 2: Normal 3: Lognormal) : 1
4) (Min clique size, Max clique size) : (4, 6)
5) (Min e2c latency, Max e2c latency) : (5, 50)
6) (Min c2c latency, Max c2c latency) : (5, 200)

RUNNING SIMULATION

1) Initialing Servers
2) Initialing Links
3) Broadcasting initial messages
4) 256 Events created and 2896 Messages created
5) Running Ripple Consensus protocol

Time : 20 ms  Servers+/Servers- : 50/50
Time : 40 ms  Servers+/Servers- : 50/50
Time : 60 ms  Servers+/Servers- : 51/49
Time : 80 ms  Servers+/Servers- : 60/40
Time : 100 ms  Servers+/Servers- : 67/33
Time : 120 ms  Servers+/Servers- : 79/21

PERFORMANCE MEASURES

1) Consensus outcome: Majority of Servers agree on the transaction
2) Positive servers: 82
3) Negative servers: 18
4) Consensus time : Consensus reached in 120 ms with 8710 messages on the wire.
5) Message rate : An average server sent 80 messages.
```

Figure 8: Example simulation run

It can be seen from the output of this simulation instance that the input parameters have now been updated to construct the revised network of interconnected-cliques. The implementation now considers the minimum clique and maximum clique sizes which represent a subnetwork of servers with every server having every other server in the subnetwork in its UNL. Another notable addition in the input parameters is the type of Network Topology that suggests the process by which these clique sizes are chosen from the range of clique sizes which could either be at random or from a normal or lognormal distribution of that range. After the initial input values are provided the consensus protocol is then simulated based on

those values.

Following the execution of the consensus protocol, it was seen that the entire run of 64 simulations comprised of all possible consensus outcomes which could trigger the termination of a consensus process. For instance, it was seen that on multiple occasions that consensus protocol reaches a halting state with no deterministic result. This situation is triggered due the restrictive nature of the underlying network topology.

```
Time : 100 ms  Servers+/Servers- : 52/48
Time : 200 ms  Servers+/Servers- : 69/31
Time : 300 ms  Servers+/Servers- : 69/31
Time : 400 ms  Servers+/Servers- : 69/31
Time : 500 ms  Servers+/Servers- : 69/31
Time : 600 ms  Servers+/Servers- : 69/31
Time : 700 ms  Servers+/Servers- : 69/31
Time : 800 ms  Servers+/Servers- : 69/31
Time : 907 ms  Servers+/Servers- : 69/31
```

(a) Halting State of consensus

```
PERFORMANCE MEASURES

1) Consensus outcome: No messages to send. Aborting Consensus.
2) Positive servers: 69
3) Negative servers: 31
4) Consensus time : Consensus reached in 907 ms with 0 messages
5) Message rate : An average server sent 177 messages.
```

(b) Results

Figure 9: Exampple simulation run

Another interesting observation conflicting to the results of the previous implementations was a relatively higher number of instances where consensus has been achieved due to the positive servers. This follows directly from our explanation on the negative bias discussed in 5.1.4, given that the revised network topology is much more regulated in terms of the UNL overlap, consensus results are achieved much faster and the negative bias becomes ineffective to the network. Consequently, execution results from the core consensus protocol are presented with a step size of 20ms instead of 100ms.

6 Testing

As we were primarily looking to study the controversial existence of the 20% overlap among all UNLs and cliques, there were test snippets created to check for the validity of this assumption 9.4.4.

6.1 First and second Implementation Testing

For checking the validity of the first and second implementation we look at the two tests, first test iterates through all the possible pairs of UNL's in the network to see if any such pair has insufficient overlap that could lead to conflicting consensus results. The second test looks at the UNLs of the UNLs of all servers to see if the overlap holds. Following test results were produced for the first implementation confirming forks in the system due to insufficient overlap.

```
PERFORMANCE MEASURES

1) Consensus outcome: No messages to send. Aborting Consensus.
2) Positive servers: 69
3) Negative servers: 31
4) Consensus time : Consensus reached in 907 ms with 0 messages
5) Message rate : An average server sent 177 messages.
```

Figure 10: Example simulation run

6.2 Third Implementation Testing

For checking the validity of the third implementation we look at one test case which iterates through all the possible pairs of UNL cliques in the network to see if any such pair has insufficient overlap that could lead to conflicting consensus results. Following test results

were produced for the third implementation confirming sufficient overlap.

```
------------------------------------------------------------

CHECKING UNL OVERLAP CONDITION (TEST 1)
------------------------------------------------------------

Possibility of fork: Insufficient UNL overlap between servers 0 and 5

UNL of server 0 : [54, 41, 43, 55, 96, 22, 88, 8, 86, 1, 76, 18, 15, 77, 51, 84, 29, 64, 10, 58, 5, 21, 95, 30, 42]

UNL of server 5 : [90, 76, 94, 15, 50, 65, 31, 79, 10, 78, 83, 93, 32, 68, 35, 82, 29, 89, 59, 66, 9, 46]

UNL overlap [10, 76, 29, 15]
------------------------------------------------------------

CHECKING UNL OVERLAP CONDITION (TEST 1)
------------------------------------------------------------

Possibility of fork: Insufficient UNL overlap between servers 0 and 5

UNL of server 0 : [54, 41, 43, 55, 96, 22, 88, 8, 86, 1, 76, 18, 15, 77, 51, 84, 29, 64, 10, 58, 5, 21, 95, 30, 42]

UNL of server 5 : [90, 76, 94, 15, 50, 65, 31, 79, 10, 78, 83, 93, 32, 68, 35, 82, 29, 89, 59, 66, 9, 46]

UNL overlap [10, 76, 29, 15]
```

Figure 11: Example simulation run

7 Conclusions and Future Work

This chapter summaries the project results and suggests potential future work that can be carried out beyond the scope of this project.

This project seeks to evaluate, model and improve the Ripple consensus protocol though an Agent-Based simulation in Python. Following the initial implementation of the existing simulation provided by Ripple, it was found that the simulation uses certain input values to the model differ significantly with the real-world implementation of the consensus protocol. Consequently, efforts were made to take a range of input values, including values that represent the real-setting of Ripple, to gain a deeper understanding of the consensus protocol on the level of the building blocks which construct the Ripple network. On further evaluation of the extended simulation model, it was deduced that the underlying network on which the consensus protocol is run fails to satisfy the assumption of a 20% UNL overlap between any two UNLs resulting in possible forks in the system that could lead to double spending. The last stage of the project proposes and implements a revised network topology involving a network of interconnected cliques which extends the stochastic block model. This revised topology also coincides with the clique-like structure proposed by Ripple in their whitepaper [6] and with the cliques achieving sufficient overlapping to prevent forks in the system.

Although we did end up merging the consensus protocol with the revised network topology of cliques but have still to conduct an extensive evaluation of on how a range of different values of the updated inputs to this topology can affect the consensus protocol just like it was done with the previous random network topology. Moreover, techniques such as Agent-based modelling is transferable and can be implemented for studying the consensus protocols of other popular cryptocurrencies like Ethereum.

8 Bibliography

[1] Satoshi Nakamoto. “Bitcoin: A peer-to-peer electronic cash system”. In: (2009).

[2] Gavin Wood. “Ethereum: A secure decentralised generalised transaction ledger”. In: *Ethereum Project Yellow Paper* 151 (2014).

[3] Sunny King. “Primecoin: Cryptocurrency with prime number proof-of-work”. In: *July 7th* (2013).

[4] Business Insider. *WHAT IS LITECOIN: Here’s What You Need to Know About The Digital Currency Growing Faster Than Bitcoin*. 2013. URL: `http://www.businessinsider.com/introduction-to-litecoin-2013-11?IR=T`.

[5] Evan Duffield and Daniel Diaz. *Dash: A privacy-centric crypto-currency*. 2014.

[6] David Schwartz, Noah Youngs, and Arthur Britto. “The Ripple protocol consensus algorithm”. In: *Ripple Labs Inc White Paper* 5 (2014).

[7] Michael Fleder, Michael S Kester, and Sudeep Pillai. “Bitcoin transaction graph analysis”. In: *arXiv preprint arXiv:1502.01657* (2015).

[8] Christian Decker and Roger Wattenhofer. “Information propagation in the bitcoin network”. In: *Peer-to-Peer Computing (P2P), 2013 IEEE Thirteenth International Conference on*. IEEE. 2013, pp. 1–10.

[9] Elli Androulaki, Ghassan O Karame, Marc Roeschlin, Tobias Scherer, and Srdjan Capkun. “Evaluating user privacy in bitcoin”. In: *International Conference on Financial Cryptography and Data Security*. Springer. 2013, pp. 34–51.

[10] Julie Fortuna, Ben Holtz, and Jocelyn Neff. “Evolutionary Structural Analysis of the Bitcoin Network”. In: (2013).

[11] Annika Baumann, Benjamin Fabian, and Matthias Lischke. “Exploring the Bitcoin Network.” In: *WEBIST (1)*. 2014, pp. 369–374.

[12] AMG Lopez, B Alvarez-Pereira, S Gorsky, and M Ayres. "Network and Conversation Analyses of Bitcoin". In: (2014).

[13] Frederik Armknecht, Ghassan O Karame, Avikarsha Mandal, Franck Youssef, and Erik Zenner. "Ripple: Overview and outlook". In: *International Conference on Trust and Trustworthy Computing*. Springer. 2015, pp. 163–180.

[14] Peter Todd. "Ripple Protocol Consensus Algorithm Review". In: (2015).

[15] Coinmarketcap. *CryptoCurrency Market Capitalizations*. 2017. URL: https://coinmarketcap.com/.

[16] Ripple. *Announcing Ripple's Global Payments Steering Group*. 2017. URL: https://ripple.com/insights/announcing-ripples-global-payments-steering-group/.

[17] Ripple. *Santander Becomes the First U.K. Bank to Use Ripple for Cross-Border Payments*. 2017. URL: https://ripple.com/insights/santander-becomes-first-uk-bank-use-ripple-cross-border-payments/.

[18] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine generals problem". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4.3 (1982), pp. 382–401.

[19] Coinmarketcap. *CryptoCurrency Market Capitalizations*. 2017. URL: https://coinmarketcap.com/currencies/ripple/.

[20] Bravenewcoin. *Market Cap*. 2017. URL: https://bravenewcoin.com/market-cap/.

[21] *What is PySPG?* URL: https://github.com/tessonec/PySPG/wiki.

9 Appendix

9.1 Appendix A: Project Plan

TITLE

Agent-based model for analysing the performance of Ripple as a Distributed Ledger Protocol

SUPERVISOR

Paolo Tasca

CO-SUPERVISOR

Claudio J. Tessone

AIMS

To investigate the key building blocks and performance measures of a popular Distributed Ledger Protocol, Ripple and gather valuable insights on conditions under which the protocol could potentially break or be vulnerable.

If possible perform a similar study for other Distributed Ledger Protocols, such as Ethereum.

OBJECTIVES

1. Review Bitcoin Nakamoto protocol, which would serve good starting point for the studying the Ripple consensus protocol.
2. Conduct research on the Ripple consensus protocol, which would be essential in determining the potential building blocks and performance metrics for Ripple.
3. Gain an understanding on the agent based model of the Bitcoin network developed by my supervisor, Paolo.
4. Implement an agent-based model, which simulates the ripple network in a reduced scale using the building blocks.
5. Conduct a performance analysis of the underlying building blocks to determine how these building blocks perform under different circumstances.
6. Could be extended for Ethereum, which is another distributed ledger protocol.

DELIVERABLES

1. Specification with regards to the chosen building blocks and performance metrics for Ripple, which would be taken into account for the agent, based model.
2. Agent-based model for simulating the ripple network.
3. Performance analysis on the building blocks of ripple and a summary of implications based on the analysis.

WORK PLAN

The work plan for this project is split into months with the aim of accomplishing the objective mentioned. Each month will further be divided into weekly meetings with my supervisor and bi-weekly meetings with my co-supervisor where I'll be discussing the progress so far, reviewing the objectives and future work for the coming week. Expected timeframe of the work plan will be as follows:

- October:
  Initial discussion with my supervisor regarding the project and what is to be expected from me in terms of delivery of the project.

- November:
  Start November – Mid November:
  Develop a firm understanding on Blockchain and the underlying concepts which are essential for the project. Read about the Bitcoin Nakamoto consensus protocol useful for building a base prior to studying about Ripple.
  Mid November – End November:
  Perform literature search and review on the building blocks and performance metrics of Ripple and carry out some analysis on the key building blocks which would be a subset of all building blocks. Showcase findings to my supervisor and finalise on the requirements.

- December:
  Begin developing the agent based model for the Ripple consensus protocol seeking some intuition from the developed Bitcoin model.

- January:
  Start January – Mid January:
  Continue developing on the model.
  Mid January – End January:
  Check the validity of the model through suitable testing and evaluation, confirming that everything is sound. Commence on the performance analysis of the building blocks.

- February:
  Start February – Mid February:
  Complete the performance and gather the results of the analysis.
  Mid February – End February:
  Start working on the Final Report.

- March:
  Continue working on the Final Report

- April:
  Submission of the Final report.

9.2 Appendix B: Interim Report

NAME
Vardan Tandon

PREVIOUS PROJECT TITLE
Agent-based model for analysing the performance of Ripple as a Distributed Ledger Protocol

CURRENT PROJECT TITLE
Agent-based model for analysing the performance of Ripple as a Distributed Ledger Protocol

INTERNAL SUPERVISOR
Paolo Tasca

EXTERNAL SUPERVISOR
Claudio J. Tessone

PROGRESS MADE

- Developed a firm understanding on Blockchain and the underlying concepts which are essential for the project. Read about the Bitcoin Nakamoto consensus protocol useful for building a base prior to studying about Ripple.
- Understanding the Nakamoto consensus code and documenting my understanding including the Gillespie model which is crucial in agent based model and will be applicable to Ripple's model as well.
- Recording the inputs or building blocks and the constraints of those building blocks required for constructing the agent based model for Ripple.
- Submitted weekly or bi-weekly reports to my supervisor updating him with my progress on the project.

REMAINING WORK

- Development on the Ripple ABM given the inputs and conduct the required performance and risk analysis on Ripple Consensus Protocol.
- Drafting the final report for the study of Ripple as a Distributed Ledger.

9.3 Appendix C: Visualisation of RPCA

This appendix gives a detailed visualisation of how the Ripple Protocol Consensus Algorithm discussed in section 2.3 is performed by considering a server 0 which contains servers 1,2,3,4 and 5 in its UNL.

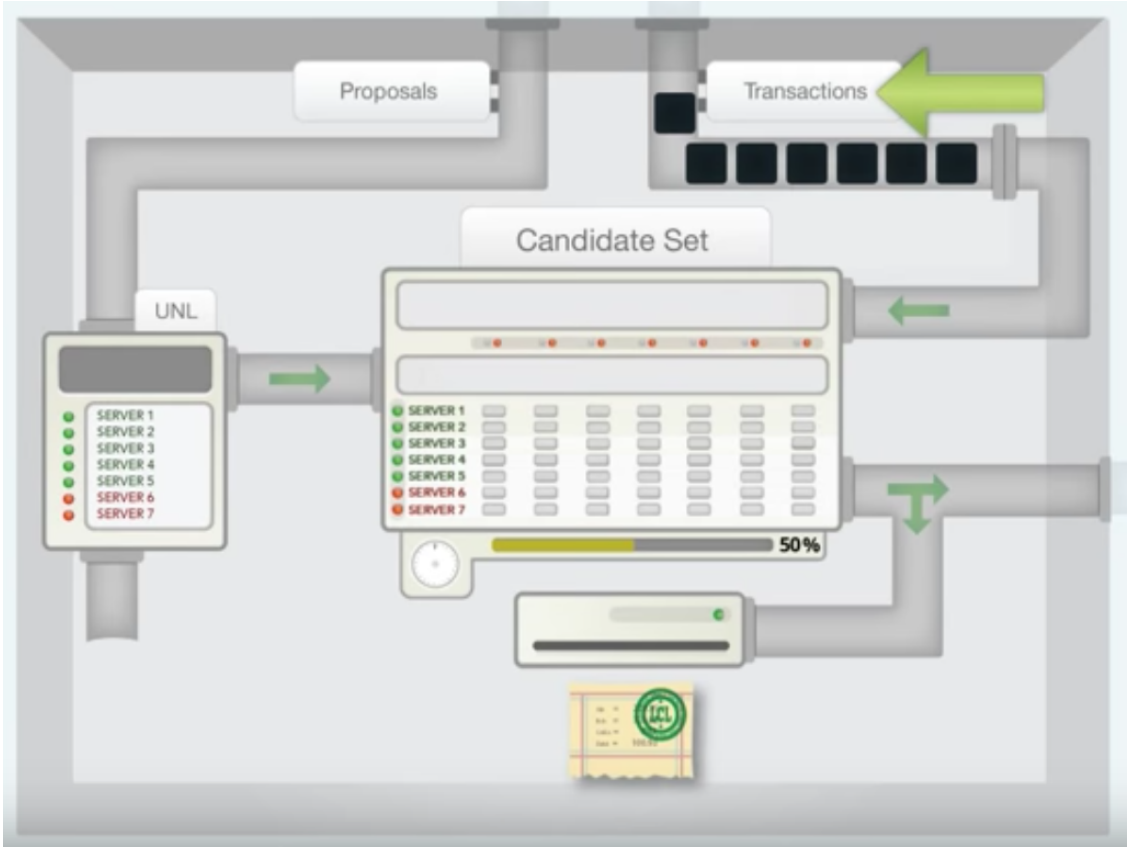


Figure 1: Transactions entering Server 0 [3]

[3]Source: How Ripple Works - The Consensus Process (ADVANCED), `https://vimeo.com/64405422`

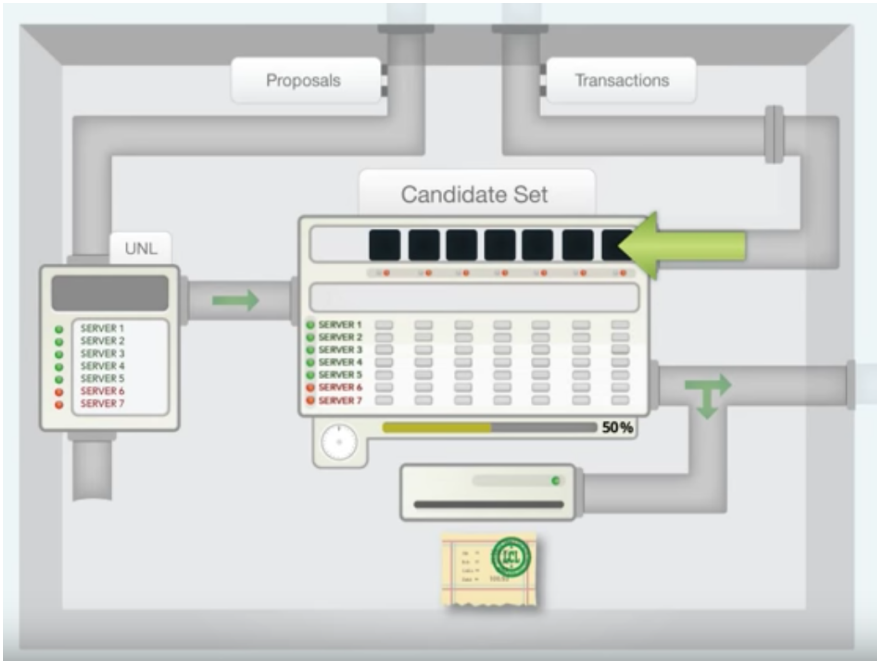


Figure 2: Incoming transactions forming a candidate set [1]

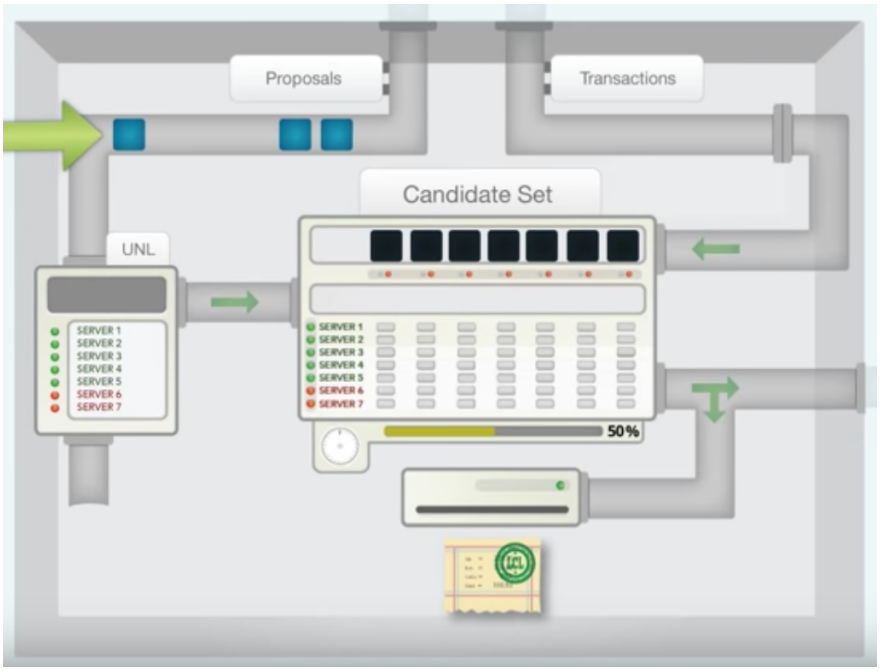


Figure 3: Proposals entering Server 0 [1]

[3]Source: How Ripple Works - The Consensus Process (ADVANCED), `https://vimeo.com/64405422`

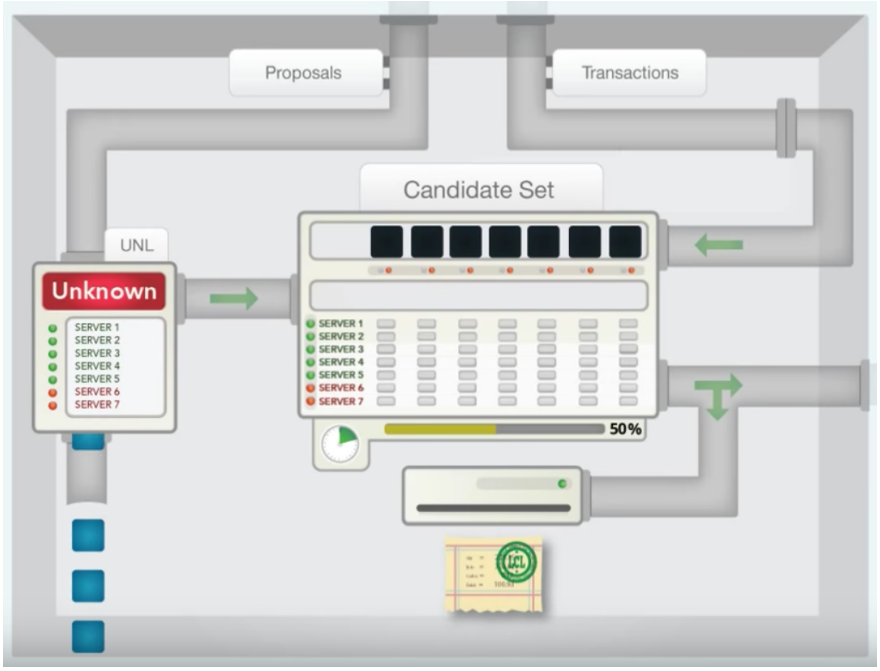


Figure 4: Proposal from an unknown server 6 not in UNL [1]

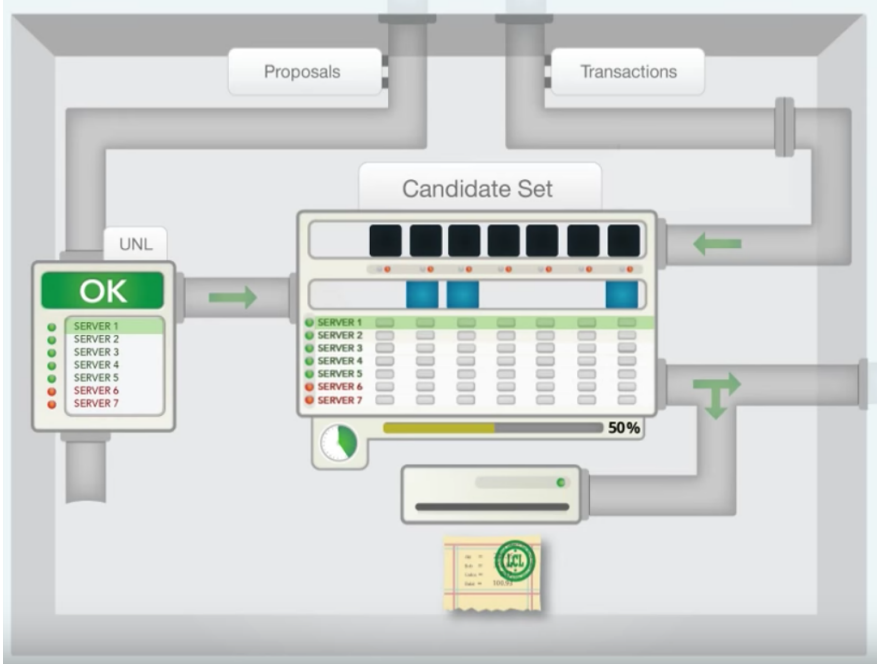


Figure 5: Incoming transactions from server 1's proposal [1]

[3]Source: How Ripple Works - The Consensus Process (ADVANCED), `https://vimeo.com/64405422`

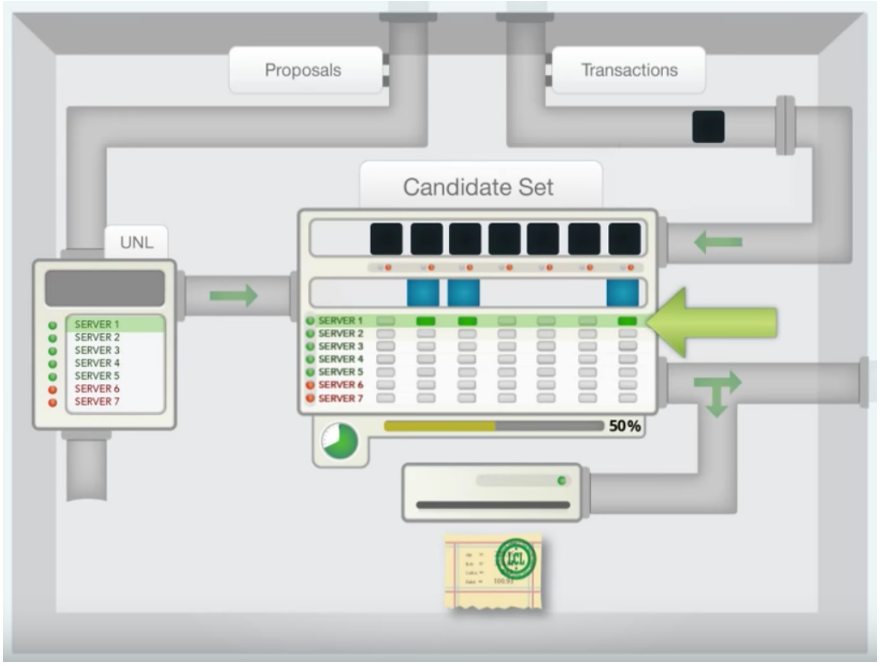


Figure 6: Matching transactions receive one vote [1]

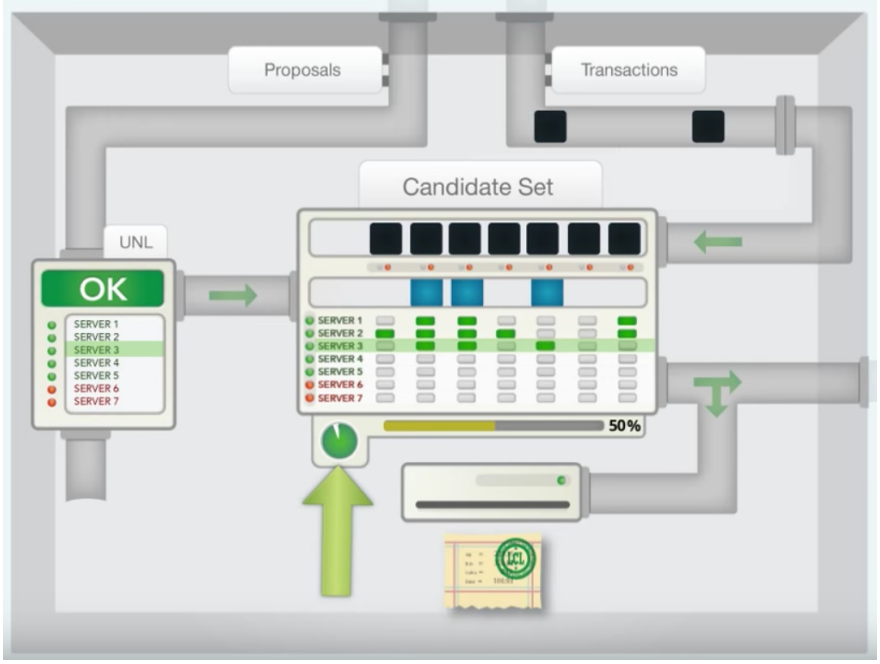


Figure 7: Server 0 continues to receive proposals [1]

[3]Source: How Ripple Works - The Consensus Process (ADVANCED), `https://vimeo.com/64405422`

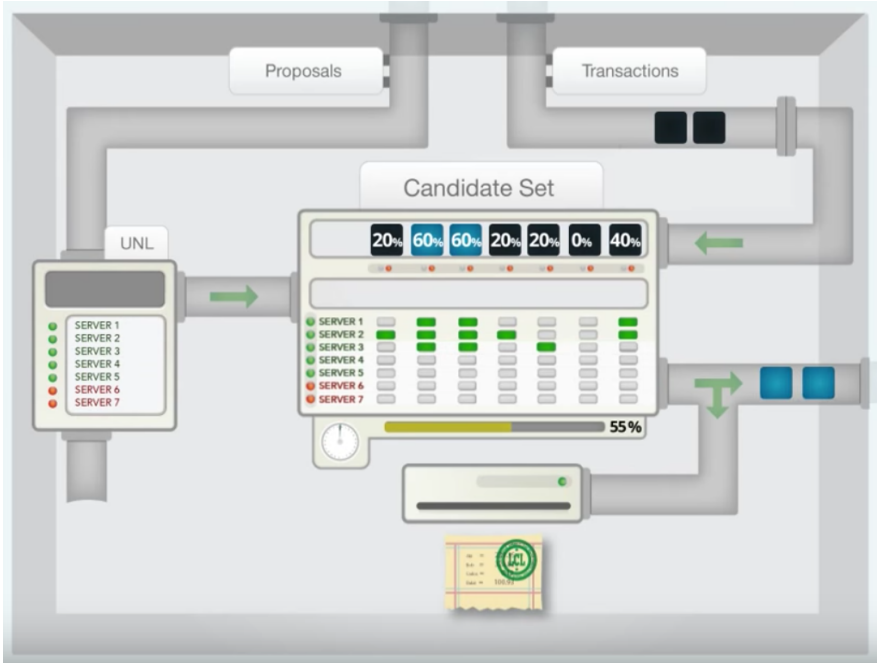


Figure 8: Server 0 broadcasts the new proposal [1]

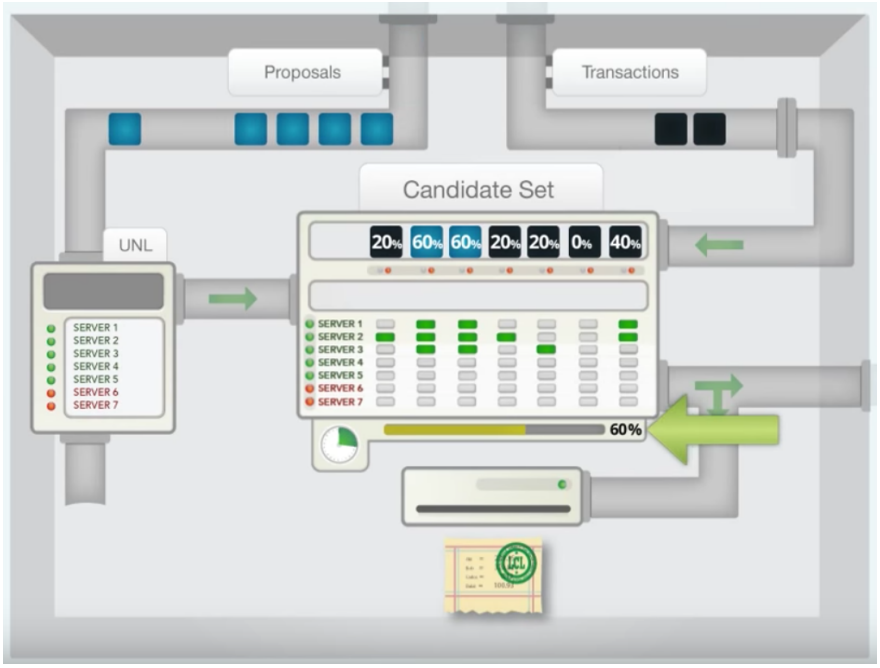


Figure 9: Approval rating rises to 60% [1]

[3]Source: How Ripple Works - The Consensus Process (ADVANCED), `https://vimeo.com/64405422`

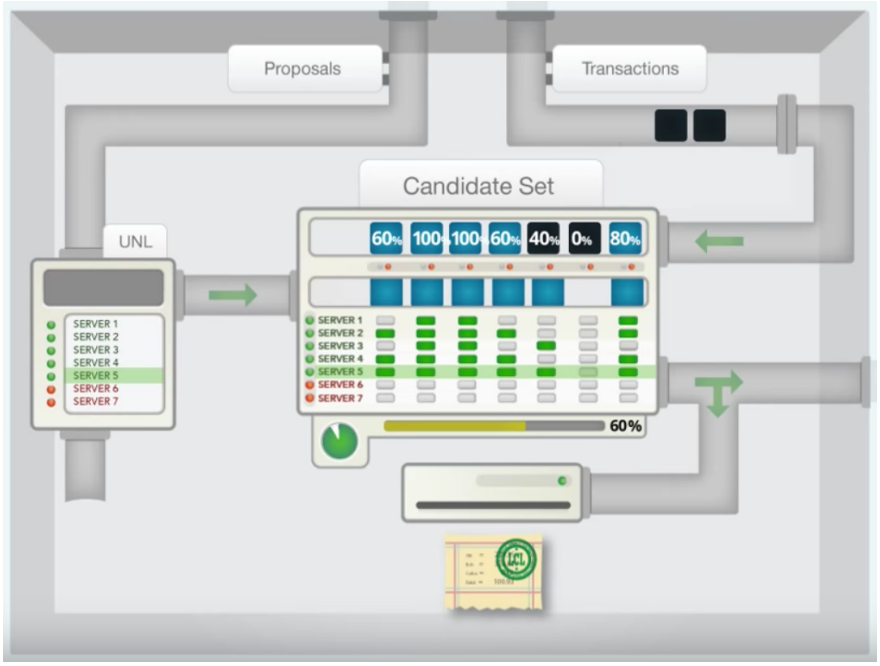


Figure 10: Server continues to receive proposals [1]

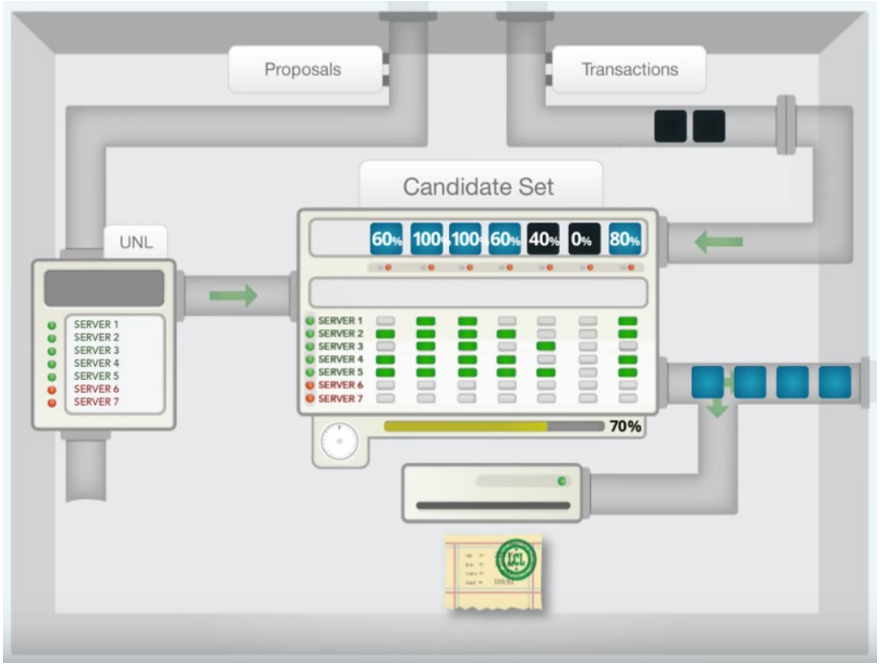


Figure 11: New proposal is broadcasted [1]

[3]Source: How Ripple Works - The Consensus Process (ADVANCED), `https://vimeo.com/64405422`

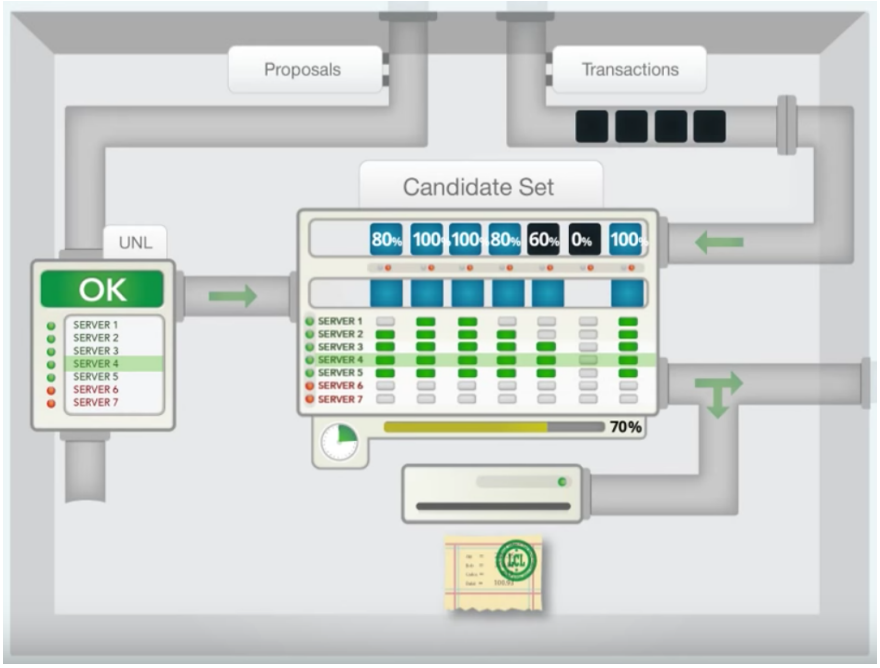


Figure 12: Similarity increases [1]

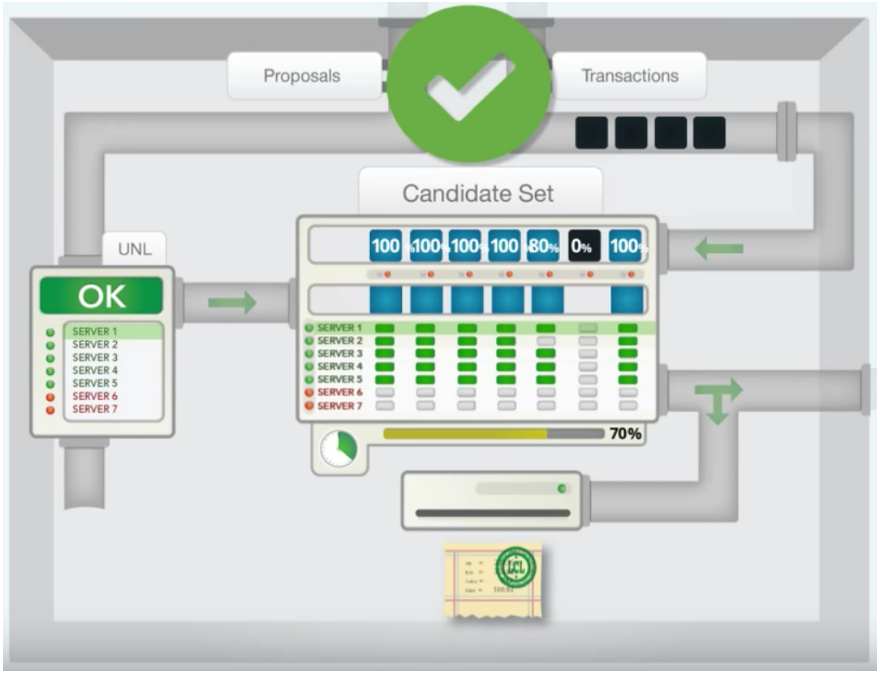


Figure 13: Server reaches consensus [1]

[3]Source: How Ripple Works - The Consensus Process (ADVANCED), `https://vimeo.com/64405422`

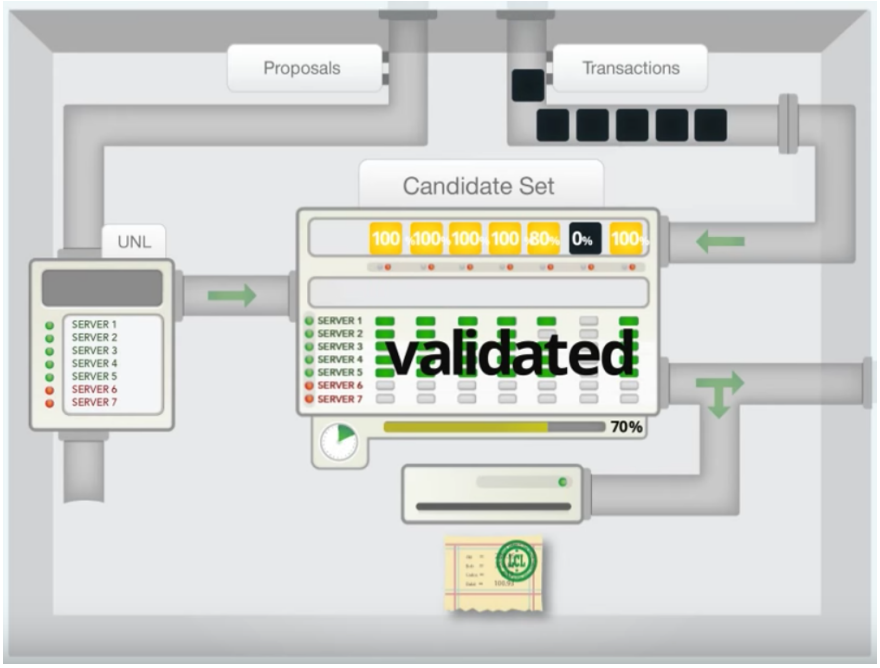


Figure 14: Proposal Validated [1]

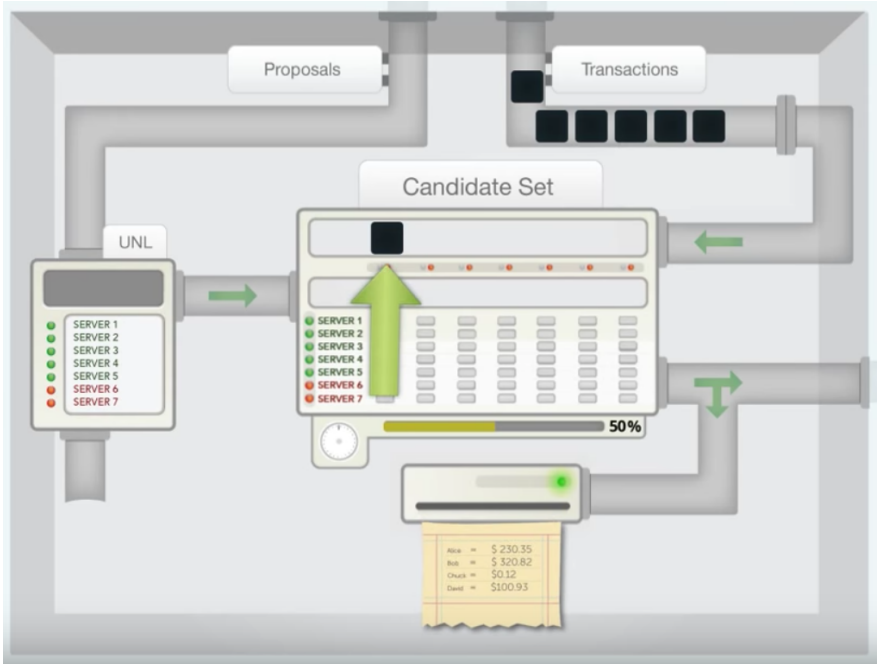


Figure 15: New consensus process [1]

[1]Source: How Ripple Works - The Consensus Process (ADVANCED), `https://vimeo.com/64405422`

9.4 Appendix G: Code Listing

9.4.1 First Implementation of Ripple Consensus Protocol

```
import networkx as nx
import numpy as np
from collections import OrderedDict, Callable
import warnings
import collections

warnings.filterwarnings('ignore')

'''
Server Class: An Agent or server in the Ripple network
'''

class Server:

    def __init__(self, server_id):

    '''
    Server attributes
    1. server_id: Unique id for Server/Agent
    2. e2c_latency: End to Core Latency describing the latency from a Server
   to a                    neighbouring Server
    3. unl: List of trusted servers, also known as the Unique Node List
    4. links: List of connected servers
    5. server_time_stamps: Dictionary of known server time stamps
    6. server_positions: Dictionary of known server positions
    7. messages_sent: Number of messages sent by the server
    8. messages_received: Number of messages received by the server
    '''

```

```
29          self.server_id = server_id
30          self.e2c_latency = 0
31          self.unl = []
32          self.links = []
33          self.server_time_stamps = collections.defaultdict(int)
34          self.server_positions = collections.defaultdict(int)
35          self.messages_sent = 0
36          self.messages_received = 0
37
38      def check_unl(self, check_server):
39
40      '''
41      Server method check_unl:
42      Return True if the server belongs to the UNL of this server else
43      returns False
44      '''
45
46          for server in self.unl:
47              if check_server == server:
48                  return True
49          return False
50
51      def check_link(self, check_server):
52
53      '''
54      Server method check_link:
55      Return True is the server is connected or linked to this server else
56      returns False
57      '''
58
59          for link in self.links:
60              if check_server == link.link_to:
61                  return True
62          return False
```

```
63
64      def receive_message(self, message, network):
65
66      '''
67      Server method receive_message
68      The server updates its knowledge of server states concerning the network
69      based on the message received, undergoes state change depending on its
70      position and intent(honest/malicious) and further broadcasts its
71      updated state to the network.
72      '''
73
74          global positive_servers, negative_servers
75
76          self.messages_received += 1
77
78          # If the message was being sent to itself, the server ignores the data
        but updates the outbound message.
79          for link in self.links:
80              if (link.link_to == message.message_from) and (link.
    message_send_time >= network.master_time):
81                  link.message.del_states(message.server_data)
82                  break
83
84          # Server updates its existing knowledge of the Server States and
    Server Time stamps.
85          broadcast_changes = collections.defaultdict(ServerState)
86
87          for server_id, server_state in message.server_data.items():
88              if (server_id != self.server_id) and (
89                          self.server_positions[server_id] != server_state.
    server_position) and (
90                          server_state.time_stamp > self.server_time_stamps[
    server_id]):
```

```
91                  self.server_positions[server_id] = server_state.
    server_position
92                  self.server_time_stamps[server_id] = server_state.time_stamp
93                  broadcast_changes[server_id] = server_state
94
95          # No changes to be broadcasted to the network
96          if len(broadcast_changes) == 0:
97              return
98
99          # Choose our Position change
100         unl_count = 0
101         unl_balance = 0
102
103         # Calcuates server votes from the UNL
104         for server_id in self.unl:
105
106             if (self.server_positions[server_id] == 1):
107                 unl_count += 1
108                 unl_balance += 1
109
110             if (self.server_positions[server_id] == -1):
111                 unl_count += 1
112                 unl_balance -= 1
113
114         # Malicious server completely disagrees with the votes
115         if self.server_id < malicious_servers:
116             unl_balance = -unl_balance
117
118         # Adding a negative bias with time to favour 'no'
119         unl_balance -= int(network.master_time / 250)
120
121         pos_change = False
122
123         # Enough data to make choose a position
```

```
124         if (unl_count >= unl_threshold):
125 
126             if (self.server_positions[self.server_id] == -1) and (unl_balance
    > self_weight):
127 
128                 # If enough servers in the unl on aggregate vote in favour of
    'yes', the server switches from 'no' to 'yes'
129                 self.server_positions[self.server_id] = 1
130                 positive_servers += 1
131                 negative_servers -= 1
132                 self.server_time_stamps[self.server_id] += 1
133                 broadcast_changes[self.server_id] = ServerState(self.server_id
    , self.server_time_stamps[self.server_id],
134                                                                 self.
    server_positions[self.server_id])
135                 pos_change = True
136 
137             elif (self.server_positions[self.server_id] == 1) and (unl_balance
     < (-self_weight)):
138 
139                 # If enough servers in the unl on aggregate vote in favour of
    'no', the server switches from 'yes' to 'no'
140                 self.server_positions[self.server_id] = -1
141                 positive_servers -= 1
142                 negative_servers += 1
143                 self.server_time_stamps[self.server_id] += 1
144                 broadcast_changes[self.server_id] = ServerState(self.server_id
    , self.server_time_stamps[self.server_id],
145                                                                 self.
    server_positions[self.server_id])
146                 pos_change = True
147 
148         # Broadcast the message
149         for link in self.links:
```

```
150
151             if (pos_change) or (link.link_to != message.message_from):
152
153                 # Update a message that wasn't sent or broadcasted
154                 if link.message_send_time > network.master_time:
155                     link.message.update_states(broadcast_changes)
156
157                 # Broadcast the changes
158                 else:
159                     send_time = network.master_time
160
161                     # Delay the message to coalesce
162                     if not (pos_change):
163                         send_time += base_delay
164
165                         # Delay more if a packet is on the wire
166                         if link.message_receive_time > send_time:
167                             send_time += int(link.total_latency /
    packets_on_wire)
168
169                     # Construct the message that needs to be sent/broadcasted
170                     new_message = Message(self.server_id, link.link_to)
171                     new_message.server_data = broadcast_changes
172
173                     # Send/Broadcast the message
174                     network.send_message(new_message, link, send_time)
175                     self.messages_sent = self.messages_sent + 1
176
177
178 '''
179 Link Class: Connection or link from one server to another.
180 '''
181
182 class Link:
```

```
183
184     def __init__(self, link_to, total_latency):
185
186     '''
187     Link attributes
188     1. link_to: Server id of the connected server.
189     2. total_latency: Total latency is the combined end to core latencies of
        the two connected servers and the additional core to core latency as the
        connected servers might be far away.
190     3. message_send_time: Time at which the message is sent across the link.
191     4. message_receive_time: Time at which the message is received across the
        link, equals the send time plus the total latency across the link.
192     5. message: Message being sent across the link.
193     '''
194
195         self.link_to = link_to
196         self.total_latency = total_latency
197         self.message_send_time = 0
198         self.message_receive_time = 0
199         self.message = None
200
201
202 '''
203 Network Class: Message Transfer and Events Update
204 '''
205
206 class Network:
207
208     def __init__(self):
209
210     '''
211     Network attributes
212     1. master_time: Indicates the time at which the simulation is currently
213                     running.
```

```
214     2. events: Dictionary of events that need to be executed as the time or
                    master_time progresses
215     '''
216
217         self.master_time = 0
218         self.events = collections.defaultdict(Event)
219
220     def send_message(self, message, link, send_time):
221
222     '''
223     Network method send_message:
224     The network sends the message through the link by updating the sending and
        receiving times of the message along with the message being sent across
225     the link. It also updates its events dictionary as it receives multiple
226     messages at a given time.
227     '''
228
229         link.message_send_time = send_time
230         link.message_receive_time = send_time + link.total_latency
231         # The message is appended to the event dictionary along with the
       expected receive time.
232         self.events[link.message_receive_time].add_message(message)
233         link.message = self.events[link.message_receive_time].return_message()
234
235
236 '''
237 Event Class: An event or collection of messages
238 '''
239
240 class Event:
241
242     def __init__(self):
243
244     '''
```

```
245     Event attributes
246     1. messages: List of messages that need to be sent at a particular
247                  instance
248     '''
249
250         self.messages = []
251
252     def add_message(self, message):
253
254     '''
255     Event method add_message
256     This method appends a message to its list of messages that are associated
257     with a particular time in the simulation.
258     '''
259
260         self.messages.append(message)
261
262     def return_message(self):
263
264     '''
265     Event method return_message
266     This method returns the most recenly added message from its list of
267     messages.
268     '''
269
270         return list(reversed(self.messages))[0]
271
272
273 '''
274 Message Class: Representing a message that needs to be sent.
275 '''
276
277 class Message:
278
```

```
279     def __init__(self, message_from, message_to):
280 
281     '''
282     Message attributes
283     1. message_from: Server id of the server that sends to message.
284     2. message_to: Server id of the server that receives the message.
285     3. server_data: Dictionary of all server positions known to the sending
286                     server.
287     '''
288 
289         self.message_from = message_from
290         self.message_to = message_to
291         self.server_data = collections.defaultdict(ServerState)
292 
293     def update_states(self, updated_server_data):
294 
295     '''
296     Message method update_states
297     This method either updates the existing server positions known to the
    server or adds the newly transmitted server position
298     '''
299 
300         server_data_keys = []
301         for server_key, server_value in self.server_data.items():
302             server_data_keys.append(server_key)
303 
304         for server_id, updated_state in updated_server_data.items():
305 
306             if server_id != self.message_to:
307 
308                 if server_id in server_data_keys:
309 
310                     if updated_state.time_stamp > self.server_data[server_id].
    time_stamp:
```

```
311                         self.server_data[server_id].time_stamp = updated_state
    .time_stamp
312                         self.server_data[server_id].server_position =
    updated_state.server_position
313
314                 else:
315                     self.server_data[server_id] = updated_state
316
317     def del_states(self, received_server_data):
318
319     '''
320     Message method del_states
321     This method deletes the data from the message that was sent to itself.
322     '''
323
324         server_data_keys = []
325         for server_key, server_value in self.server_data.items():
326             server_data_keys.append(server_key)
327
328         for server_id, server_state in received_server_data.items():
329
330             if server_id != self.message_to:
331
332                 if server_id in server_data_keys:
333
334                     if server_state.time_stamp >= self.server_data[server_id].
    time_stamp:
335                         # Data not needed by the server itself
336                         del self.server_data[server_id]
337
338
339 '''
340 ServerState Class: Representing a server's state with respect to the consensus
    .
```

```
'''

class ServerState:

    def __init__(self, server_id, time_stamp, server_position):

    '''
    ServerState attributes
    1. server_id: Server id of the server.
    2. time_stamp: Time stamp of the server as we proceed with the consensus
    process.
    3. server_position: State of the server which could either be a 1 if the
    server agrees with the transaction or -1 if it disagrees with the
    transaction.
    '''

        self.server_id = server_id
        self.time_stamp = time_stamp
        self.server_position = server_position


'''
DefaultOrderedDict Class: Converts an OrderedDict to a default OrderedDict
'''

class DefaultOrderedDict(OrderedDict):

    '''
    Source: http://stackoverflow.com/a/6190500/562769
    '''

    def __init__(self, default_factory=None, *a, **kw):
        if (default_factory is not None and
                not isinstance(default_factory, Callable)):
```

```
372             raise TypeError('first argument must be callable')
373         OrderedDict.__init__(self, *a, **kw)
374         self.default_factory = default_factory
375 
376     def __getitem__(self, key):
377         try:
378             return OrderedDict.__getitem__(self, key)
379         except KeyError:
380             return self.__missing__(key)
381 
382     def __missing__(self, key):
383         if self.default_factory is None:
384             raise KeyError(key)
385         self[key] = value = self.default_factory()
386         return value
387 
388     def __reduce__(self):
389         if self.default_factory is None:
390             args = tuple()
391         else:
392             args = self.default_factory,
393         return type(self), args, None, None, self.items()
394 
395     def copy(self):
396         return self.__copy__()
397 
398     def __copy__(self):
399         return type(self)(self.default_factory, self)
400 
401     def __deepcopy__(self, memo):
402         import copy
403         return type(self)(self.default_factory,
404                           copy.deepcopy(self.items()))
405 
```

```
406     def __repr__(self):
407         return 'OrderedDefaultDict(%s, %s)' % (self.default_factory,
408                                                OrderedDict.__repr__(self))
409
410
411 '''
412 main() function initializes the network and runs the Ripple simulation until
        consensus is reached.
413 '''
414
415 def main():
416
417     # Declaring all the global variables used in other classes and methods
418     global positive_servers, negative_servers, num_servers, malicious_servers,
         unl_threshold, self_weight, base_delay, packets_on_wire,
        consensus_percent
419
420     # Initalizing the global and local variables (Values taken from the Ripple
         sim code on Github)
421     positive_servers = 0
422     negative_servers = 0
423
424     num_servers = 1000
425     malicious_servers = 15
426
427     server_outbound_links = 10
428     unl_min = 20
429     unl_max = 30
430     unl_threshold = int(unl_min / 2)
431
432     min_e2c_latency = 5
433     max_e2c_latency = 50
434     min_c2c_latency = 5
435     max_c2c_latency = 200
```

```
436
437     self_weight = 1
438     base_delay = 1
439     packets_on_wire = 3
440     consensus_percent = 80
441     seed_counter = 0
442
443     G = nx.Graph()
444
445     print('INPUT VALUES\n')
446     print('1) Network Size : ' + str(num_servers))
447     print('2) Malicious Servers : ' + str(malicious_servers))
448     print('3) Outbound Links : ' + str(server_outbound_links))
449     print('4) (Min UNL size, Max UNL size, UNL threshold) : (' + str(unl_min)
    + ', ' + str(unl_max) + ', ' + str(
450         unl_threshold) + ')')
451     print('5) (Min e2c latency, Max e2c latency) : (' + str(min_e2c_latency) +
     ', ' + str(max_e2c_latency) + ')')
452     print('6) (Min c2c latency, Max c2c latency) : (' + str(min_c2c_latency) +
     ', ' + str(max_c2c_latency) + ')')
453
454     # Initializing a list of all servers in the network
455     servers_list = []
456     for i in range(num_servers):
457         new_server = Server(i)
458         servers_list.append(new_server)
459
460     print('\nRUNNING SIMULATION\n')
461
462     # Initializing server attributes: latency, server positions, server time
    stamp and unl
463     print('1) Initialing Servers')
464
465     for i in range(num_servers):
```

```
466
467         # Comment the random seed function to generate different end to core
    latency values on each run
468         np.random.seed(i)
469         servers_list[i].e2c_latency = round(np.random.uniform(min_e2c_latency,
     max_e2c_latency))
470
471         for j in range(num_servers):
472             if i == j:
473                 if (i % 2):
474                     servers_list[i].server_positions[i] = 1
475                     servers_list[i].server_time_stamps[i] = 1
476                     positive_servers += 1
477                 else:
478                     servers_list[i].server_positions[i] = -1
479                     servers_list[i].server_time_stamps[i] = 1
480                     negative_servers += 1
481             else:
482                 servers_list[i].server_positions[j] = 0
483                 servers_list[i].server_time_stamps[j] = 0
484
485         # Comment the random seed function to generate different UNL sizes on
    each run
486         np.random.seed(i)
487         unl_count = round(np.random.uniform(unl_min, unl_max))
488
489         while unl_count > 0:
490
491             # Comment the random seed function to generate different UNL
    servers on each run
492             np.random.seed(i + seed_counter)
493             unl_server = round(np.random.uniform(0, num_servers - 1))
494
495             if unl_server != i and not servers_list[i].check_unl(unl_server):
```

```
                servers_list[i].unl.append(unl_server)
                unl_count -= 1
            seed_counter += 1

    L = [i for i in range(num_servers)]
    G.add_nodes_from(L)

    # Initializing Links to all servers in the graph above
    print('2) Initialing Links')
    for i in range(num_servers):

        num_links = server_outbound_links

        while num_links > 0:

            # Comment the random seed function to generate different server links on each run
            np.random.seed(i + num_servers + seed_counter)
            link_server = round(np.random.uniform(0, num_servers - 1))

            if link_server != i and not servers_list[i].check_link(link_server) and not servers_list[
                link_server].check_link(i):
                # Comment the random seed function to generate different core to core latency on each run
                np.random.seed(i + num_servers + seed_counter)
                c2c_latency = round(np.random.uniform(min_c2c_latency, max_c2c_latency))
                tot_latency = servers_list[i].e2c_latency + servers_list[link_server].e2c_latency + c2c_latency
                servers_list[i].links.append(Link(link_server, tot_latency))
                servers_list[link_server].links.append(Link(i, tot_latency))
                num_links -= 1
            seed_counter += 1
```

```
525
526     # Adding the links generated
527     for i in range(num_servers):
528         for j in range(len(servers_list[i].links)):
529             G.add_edges_from([(i, servers_list[i].links[j].link_to)])
530
531     network = Network()
532
533     # Broacasting initial positions to the connected servers
534     print('3) Broadcasting initial messages')
535
536     for i in range(num_servers):
537         link_num = 0
538         for link in servers_list[i].links:
539             m = Message(i, link.link_to)
540             m.server_data[i] = ServerState(i, 1, servers_list[i].
    server_positions[i])
541             network.send_message(m, link, 0)
542             link_num += 1
543
544     messages_created = 0
545     for event_time, event in network.events.items():
546         messages_created += len(event.messages)
547
548     print('4) ' + str(len(network.events)) + ' Events created and ' + str(
    messages_created) + ' Messages created')
549     print('5) Running Ripple Consensus protocol (might take some time) \n')
550
551     # Starts the consensus process until a final decision is reached
552     while True:
553
554         # Majority servers agree with the transactions
555         if positive_servers > (num_servers * (consensus_percent / 100)):
556             print('\nPERFORMANCE MEASURES\n')
```

```
557             print('1) Consensus outcome: Majority of Servers agree on the
    transaction')
558             print('2) Positive servers: ' + str(positive_servers))
559             print('3) Negative servers: ' + str(negative_servers))
560             break
561 
562         # Majority servers disagree with the transactions
563         if negative_servers > (num_servers * (consensus_percent / 100)):
564             print('\nPERFORMANCE MEASURES\n')
565             print('1) Consensus outcome: Majority of servers disagree on the
    transaction')
566             print('2) Positive servers: ' + str(positive_servers))
567             print('3) Negative servers: ' + str(negative_servers))
568             break
569 
570         ordered_network_events = collections.OrderedDict(sorted(network.events
    .items()))
571         network.events = DefaultOrderedDict(Event)
572 
573         for event_time, event in ordered_network_events.items():
574             network.events[event_time] = event
575 
576         # Events are sequantially iterated
577         event_iter = iter(network.events)
578 
579         try:
580             event_time = event_iter.__next__()
581             event = network.events[event_time]
582 
583         except StopIteration:
584             print('\nPERFORMANCE MEASURES\n')
585             print('1) Consensus outcome: No messages to send. Aborting
    Consensus.')
586             print('2) Positive servers: ' + str(positive_servers))
```

```
587             print('3) Negative servers: ' + str(negative_servers))
588             break
589 
590         if int(event_time / 100) > int(network.master_time / 100):
591             print('Time : ' + str(event_time) + ' ms ' + ' Servers+/Servers- :
    ' + str(positive_servers) + '/' + str(
592                 negative_servers))
593 
594         network.master_time = event_time
595 
596         # Transmission of messages
597         for m in event.messages:
598             if len(m.server_data) == 0:
599                 servers_list[m.message_from].messages_sent -= 1
600             else:
601                 servers_list[m.message_to].receive_message(m, network)
602 
603         del network.events[event_time]
604 
605     message_count = 0
606     for event_time, event in network.events.items():
607         message_count += len(event.messages)
608 
609     # Final Consensus results are captured
610     print('4) Consensus time : Consensus reached in ' + str(network.
    master_time) + ' ms with ' + str(
611         message_count) + ' messages on the wire.')
612 
613     total_messages_sent = 0
614 
615     for i in range(num_servers):
616         total_messages_sent += servers_list[i].messages_sent
617 
```

```
618     print('5) Message rate : An average server sent ' + str(
        total_messages_sent / num_servers) + ' messages.\n')
619
620 if __name__ == "__main__":
621     main()
```

Listing 1: First Implementation code

9.4.2 Second Implementation of Ripple Consensus Protocol

```
1
2 '''
3 main() function receives the input values of the building blocks which are
      used to construct the network and then run the simulation of the Ripple
      consensus protocol on those set of values
4 '''
5
6 def main(input_values):
7
8     global positive_servers, negative_servers, num_servers, malicious_servers,
       unl_threshold, self_weight, base_delay, packets_on_wire,
      consensus_percent
9     positive_servers = 0
10    negative_servers = 0
11
12    # The building blocks are initialized with the input values passed for
      this simulation
13    num_servers, malicious_servers, server_outbound_links, unl_range,
      e2c_latency_range, c2c_latency_range = input_values
14
15    unl_min, unl_max = unl_range
16    unl_threshold = int(unl_min / 2)
17
18    min_e2c_latency, max_e2c_latency = e2c_latency_range
19    min_c2c_latency, max_c2c_latency = c2c_latency_range
```

```
20
21      self_weight = 1
22      base_delay = 1
23      packets_on_wire = 3
24      consensus_percent = 80
25      seed_counter = 0
26      simulation_summary = []
27
28      G = nx.Graph()
29
30      # Initializing a list of all servers in the network
31      servers_list = []
32      for i in range(num_servers):
33          new_server = Server(i)
34          servers_list.append(new_server)
35
36      # Initializing server attributes: latency, server positions, server time
        stamp and unl
37      for i in range(num_servers):
38
39          # Comment the random seed function to generate different end to core
        latency values on each run
40          np.random.seed(i)
41          servers_list[i].e2c_latency = round(np.random.uniform(min_e2c_latency,
         max_e2c_latency))
42
43          for j in range(num_servers):
44              if i == j:
45                  if (i % 2):
46                      servers_list[i].server_positions[i] = 1
47                      servers_list[i].server_time_stamps[i] = 1
48                      positive_servers += 1
49                  else:
50                      servers_list[i].server_positions[i] = -1
```

```
                    servers_list[i].server_time_stamps[i] = 1
                    negative_servers += 1
            else:
                servers_list[i].server_positions[j] = 0
                servers_list[i].server_time_stamps[j] = 0

        # Comment the random seed function to generate different UNL sizes on each run
        np.random.seed(i)
        unl_count = round(np.random.uniform(unl_min, unl_max))

        while unl_count > 0:

            # Comment the random seed function to generate different UNL servers on each run
            np.random.seed(i + seed_counter)
            unl_server = round(np.random.uniform(0, num_servers - 1))

            if unl_server != i and not servers_list[i].check_unl(unl_server):
                servers_list[i].unl.append(unl_server)
                unl_count -= 1
            seed_counter += 1

    L = [i for i in range(num_servers)]
    G.add_nodes_from(L)

    # Initializing Links to all servers in the graph above
    for i in range(num_servers):

        num_links = server_outbound_links

        while num_links > 0:

            # Comment the random seed function to generate different server
```

```
    links on each run
83              np.random.seed(i + num_servers + seed_counter)
84              link_server = round(np.random.uniform(0, num_servers - 1))
85
86              if link_server != i and not servers_list[i].check_link(link_server
    ) and not servers_list[
87                  link_server].check_link(i):
88                  # Comment the random seed function to generate different core
    to core latency on each run
89                  np.random.seed(i + num_servers + seed_counter)
90                  c2c_latency = round(np.random.uniform(min_c2c_latency,
    max_c2c_latency))
91                  tot_latency = servers_list[i].e2c_latency + servers_list[
    link_server].e2c_latency + c2c_latency
92                  servers_list[i].links.append(Link(link_server, tot_latency))
93                  servers_list[link_server].links.append(Link(i, tot_latency))
94                  num_links -= 1
95              seed_counter += 1
96
97      # Adding the links generated
98      for i in range(num_servers):
99          for j in range(len(servers_list[i].links)):
100             G.add_edges_from([(i, servers_list[i].links[j].link_to)])
101
102     network = Network()
103
104     # Broacasting initial positions to the connected servers
105
106     for i in range(num_servers):
107         link_num = 0
108         for link in servers_list[i].links:
109             m = Message(i, link.link_to)
110             m.server_data[i] = ServerState(i, 1, servers_list[i].
    server_positions[i])
```

```
111             network.send_message(m, link, 0)
112             link_num += 1
113 
114     num_events = len(network.events)
115     messages_created = 0
116     for event_time, event in network.events.items():
117         messages_created += len(event.messages)
118 
119     simulation_summary.extend([num_events, messages_created])
120     consensus_summary = []
121 
122     # Starts the consensus process until a final decision is reached
123     while True:
124 
125         # Majority servers agree with the transactions
126         if positive_servers > (num_servers * (consensus_percent / 100)):
127             simulation_summary.append(1)
128             simulation_summary.append(positive_servers)
129             simulation_summary.append(negative_servers)
130             break
131 
132         # Majority servers disagree with the transactions
133         if negative_servers > (num_servers * (consensus_percent / 100)):
134             simulation_summary.append(-1)
135             simulation_summary.append(positive_servers)
136             simulation_summary.append(negative_servers)
137             break
138 
139         ordered_network_events = collections.OrderedDict(sorted(network.events
    .items()))
140         network.events = DefaultOrderedDict(Event)
141 
142         for event_time, event in ordered_network_events.items():
143             network.events[event_time] = event
```

```
144
145         # Events are sequantially iterated
146         event_iter = iter(network.events)
147
148         try:
149             event_time = event_iter.__next__()
150             event = network.events[event_time]
151
152         except StopIteration:
153             simulation_summary.append(0)
154             simulation_summary.append(positive_servers)
155             simulation_summary.append(negative_servers)
156             break
157
158         if int(event_time / 100) > int(network.master_time / 100):
159             iteration_summary = [event_time, positive_servers,
    negative_servers]
160             consensus_summary.append(iteration_summary)
161
162         network.master_time = event_time
163
164         # Transmission of messages
165         for m in event.messages:
166             if len(m.server_data) == 0:
167                 servers_list[m.message_from].messages_sent -= 1
168             else:
169                 servers_list[m.message_to].receive_message(m, network)
170
171         del network.events[event_time]
172
173     simulation_summary.append(consensus_summary)
174
175     message_count = 0
176     for event_time, event in network.events.items():
```

```
177         message_count += len(event.messages)
178 
179     # Final Consensus results are captured
180     simulation_summary.extend([network.master_time, message_count])
181 
182     total_messages_sent = 0
183 
184     for i in range(num_servers):
185         total_messages_sent += servers_list[i].messages_sent
186 
187     message_rate = int(total_messages_sent / num_servers)
188     simulation_summary.append(message_rate)
189 
190     #all other prints statements are removed
191     print('Ran a simulation for ' + str(num_servers) + ' servers.')
192 
193     return simulation_summary
194 
195 if __name__ == "__main__":
196 
197     # Example range of Input values that can be experimented
198     num_nodes_list = list(range(100,300,100)) # Taking 100 and 200 servers
199     mal_node_list = list(range(10,20, 5)) # Taking 10 and 15 malicious servers
200     nodes_conn_list = list(range(10, 20, 5)) # Taking 10 and 15 output links
201     unl_min = list(range(5, 15, 5)) # Taking min unl 5 and 10
202     unl_max = list(range(15, 25, 5)) # Taking max unl 15 and 20
203     unl_list = list(zip(unl_min, unl_max)) # Taking two unl ranges (5,15) and
    (10,20)
204     min_e2c_latency = list(range(5, 15, 5)) # Taking min e2c latency 5 and 10
205     max_e2c_latency = list(range(50, 60, 5)) # Taking max e2c latency 50 and
    55
206     e2c_latecy_list = list(zip(min_e2c_latency, max_e2c_latency)) # Taking two
     e2c latency ranges (5,50) and (10,55)
207     min_c2c_latency = list(range(5, 15, 5)) #Taking min c2c latency 5 and 10
```

```
208     max_c2c_latency = list(range(200, 210, 5)) # taking max c2c latency 200
    and 210
209     c2c_latecy_list = list(zip(min_c2c_latency, max_c2c_latency)) # Taking two
     c2c latency ranges (5,200) and (10,205)
210
211     param_list = [num_nodes_list, mal_node_list, nodes_conn_list, unl_list,
    e2c_latecy_list, c2c_latecy_list]
212
213     # Prepares a list of input values with all possible combination of inputs
214     input_list = list(itertools.product(*param_list))
215
216     print('\nTotal number of Simulations: ' + str(len(input_list)))
217     print('
    ----------------------------------------------------------------------------\n
    ')
218
219     print('\nProcessing all Simulations via multithreading(This might take a
    few minutes)')
220     print('
    ----------------------------------------------------------------------------\n
    ')
221
222     # Pool object that maps the input values and to produce the corresponding
    results
223     pool = multiprocessing.Pool()
224     results = pool.map(main, input_list)
225
226     print('\nSimulation Summary')
227     print('
    ----------------------------------------------------------------------------\n
    ')
228
229     # The results obtains from all simulations are now presented
230     for i in range(len(input_list)):
```

```
231
232         print('\nSimulation ' + str(i+1) + ' : ')
233         print('
    ----------------------------------------------------------------------------\n
    ')
234         print('INPUT VALUES\n')
235         print('1) Network Size : ' + str(input_list[i][0]))
236         print('2) Malicious Servers : ' + str(input_list[i][1]))
237         print('3) Outbound Links : ' + str(input_list[i][2]))
238         print('4) (Min UNL size, Max UNL size) : ' + str(input_list[i][3]))
239         print('5) (Min e2c latency, Max e2c latency) : ' + str(input_list[i
    ][4]))
240         print('6) (Min c2c latency, Max c2c latency) : ' + str(input_list[i
    ][5]))
241         print('\nRUNNING SIMULATION\n')
242         print('1) Initialing Servers')
243         print('2) Initialing Links')
244         print('3) Broadcasting initial messages')
245         print('4) ' + str(results[i][0]) + ' Events created and ' + str(
    results[i][1]) + ' Messages created')
246         print('5) Running Ripple Consensus protocol \n')
247
248         for j in range(len(results[i][5])):
249             print('Time : ' + str(results[i][5][j][0]) + ' ms ' + ' Servers+/
    Servers- : ' + str(
250                 results[i][5][j][1]) + '/' + str(results[i][5][j][2]))
251
252         print('\nPERFORMANCE MEASURES\n')
253
254         if results[i][2] == 1:
255             print('1) Consensus outcome: Majority of Servers agree on the
    transaction')
256             print('2) Positive servers: ' + str(results[i][3]))
257             print('3) Negative servers: ' + str(results[i][4]))
```

```
258
259         elif results[i][2] == -1:
260             print('1) Consensus outcome: Majority of servers disagree on the
    transaction')
261             print('2) Positive servers: ' + str(results[i][3]))
262             print('3) Negative servers: ' + str(results[i][4]))
263
264         else:
265             print('1) Consensus outcome: No messages to send. Aborting
    Consensus.')
266             print('2) Positive servers: ' + str(results[i][3]))
267             print('3) Negative servers: ' + str(results[i][4]))
268
269         print('4) Consensus time : Consensus reached in ' + str(results[i][6])
    + ' ms with ' + str(
270             results[i][7]) + ' messages on the wire.')
271         print('5) Message rate : An average server sent ' + str(results[i][8])
    + ' messages.\n')
```

Listing 2: Second Implementation code (Multiprocessing of multiple simulations)

9.4.3 Third Implementation of Ripple Consensus Protocol

```
1  """
2  generate_rand_values(n,min_size,max_size):
3  function returns a list of random clique sizes from a range
4  of minimum and maximum clique size such that it sums up to
5  the total number of servers
6  """
7
8  def generate_rand_values(n, min_size, max_size):
9      values = []
10     while n != 0:
11         np.random.seed(n)
12         value = randint(min_size, max_size)
```

```
        values.append(value)
        n -= value
        if n == 0:
            break
        if n < min_size:
            del values[-1]
            n += value
    return values

"""
generate_normal_values(n,mean_size, sd_size, min_size, max_size):
function returns a list of random clique sizes generated using a
normal distribution from a range of minimum and maximum clique size
such that it sums up to the total number of servers
"""

def generate_normal_values(n, mean_size, sd_size, min_size, max_size):
    values = []
    while n != 0:
        value = int(np.random.normal(mean_size, sd_size))
        if value < min_size or value > max_size:
            continue
        values.append(value)
        n -= value
        if n == 0:
            break
        if n < min_size:
            del values[-1]
            n += value
    return values


"""
generate_lognormal_values(n,mean_size, sd_size, min_size, max_size):
```

```
function returns a list of random clique sizes generated using a lognormal
distribution from a range of minimum and maximum clique size
such that it sums up to the total number of servers
"""

def generate_lognormal_values(n, mean_size, sd_size, min_size, max_size):
    values = []
    while n != 0:
        value = int(np.random.lognormal(mean_size, sd_size))
        if value < min_size or value > max_size:
            continue
        values.append(value)
        n -= value
        if n == 0:
            break
        if n < min_size:
            del values[-1]
            n += value
    return values


"""
generate_rand_values(n, min_overlap, max_overlap):
function returns a list of random overlaps from a range of minimum and maximum
    overlap
such that it sums up to the total number of combinations for all
cliques.
"""

def generate_rand_overlaps(n, min_overlap, max_overlap):
    values = []
    for i in range(n):
        np.random.seed(i)
        value = randint(min_overlap, max_overlap)
```

```
        values.append(value)
    return values


"""
generate_normal_overlaps(n, mean_overlap, sd_overlap, min_size, max_size):
function returns a list of random overlaps generated using normal distribution
from a range of minimum and maximum overlap such that it sums up to the
total number of combinations for all cliques.
"""

def generate_normal_overlaps(n, mean_overlap, sd_overlap, min_size, max_size):
    values = []
    decrement = n
    while decrement != 0:
        value = int(np.random.normal(mean_overlap, sd_overlap))
        if value < min_size or value > max_size:
            continue
        decrement = decrement - 1
        values.append(value)
    return values

"""
generate_rand_values(n,min_size,max_size) function returns a list of
random overlaps from a range of minimum and maximum overlap
such that it sums up to the total number of combinations for all
cliques.
"""

def generate_lognormal_overlaps(n, mean_overlap, sd_overlap, min_size,
    max_size):
    values = []
    decrement = n
    while decrement != 0:
```

```
113         value = int(np.random.lognormal(mean_overlap, sd_overlap))
114         if value < min_size or value > max_size:
115             continue
116         decrement = decrement - 1
117         values.append(value)
118     return values
119 
120 
121 """
122 function clique_topology(network_size, clique_seq, overlap_seq, col_list):
123 returns the Ripple network in the form of an interconnected network of cliques
124 satisfying atleast a 20% overlap between all UNL pairs.
125 """
126 
127 def clique_topology(network_size, clique_seq, overlap_seq, col_list):
128     num_cliques = len(clique_seq)
129     L = [i for i in range(0, network_size)]
130     G = nx.Graph()
131     clique_list = []
132     color_included = []
133     from_node = []
134     to_node = [0]
135     seed =0
136 
137     for i in range(0, num_cliques):
138 
139         clique_size = clique_seq[i]
140         np.random.seed(i + seed)
141         clique_color = random.choice(col_list)
142         while clique_color in color_included:
143             seed += 1
144             np.random.seed(i + seed)
145             clique_color = random.choice(col_list)
146             seed += 1
```

```
147
148         color_included.append(clique_color)
149
150         from_node.append(to_node[i])
151         to_node.append(to_node[i] + clique_size)
152
153         for j in range(from_node[i], to_node[i + 1]):
154             G.add_node(L[j], clique_group=i, color=clique_color)
155
156         edges = itertools.combinations(L[from_node[i]:to_node[i + 1]], 2)
157         G.add_edges_from(edges)
158         clique_list.append(L[from_node[i]:to_node[i + 1]])
159
160     clique_edges = [x for x in itertools.combinations(clique_list, 2)]
161
162     for i in range(0, len(clique_edges)):
163         edges_included = []
164         for j in range(0, overlap_seq[i]):
165             successful = False
166             while not successful:
167                 edges = (random.choice(clique_edges[i][0]), random.choice(
    clique_edges[i][1]))
168                 counter = 0
169                 for edge in edges_included:
170                     if edges[0] in edge:
171                         counter = 1
172                     if edges[1] in edge:
173                         counter = 1
174                         break
175                 if edges not in edges_included and not counter:
176                     G.add_edges_from([edges])
177                     edges_included.append(edges)
178                     successful = True
179                 else:
```

```
180                     successful = False
181     return G
182
183 '''
184 main() function receives the input values of the building blocks which are
    used to construct the network and
185 then run the simulation of the Ripple consensus protocol on those set of
    values
186 '''
187
188 def main(input_values):
189     global positive_servers, negative_servers, num_servers, malicious_servers,
     unl_threshold, self_weight, base_delay, packets_on_wire,
    consensus_percent
190
191     # Initalizing the global and local variables (Values taken from the Ripple
     sim code on Github)
192     positive_servers = 0
193     negative_servers = 0
194
195     # The building blocks are initialized with the input values passed for
    this simulation
196     num_servers, malicious_servers, network_topology, clique_range,
    e2c_latency_range, c2c_latency_range = input_values
197
198     min_clique_size, max_clique_size = clique_range
199
200     min_e2c_latency, max_e2c_latency = e2c_latency_range
201     min_c2c_latency, max_c2c_latency = c2c_latency_range
202
203     min_overlap = min_clique_size/2
204     max_overlap = min_clique_size
205
206     color_list = []
```

```
207     unl_sizes = []
208     seed_counter = 0
209 
210     self_weight = 1
211     base_delay = 1
212     packets_on_wire = 3
213     consensus_percent = 80
214     simulation_summary = []
215 
216     for i in range(num_servers):
217         successful = False
218         while not successful:
219 
220             random.seed(i+seed_counter)
221             rand_color = (random.random(), random.random(), random.random(),
    random.random())
222 
223             if rand_color not in color_list:
224                 color_list.append(rand_color)
225                 break
226             else:
227                 successful = True
228                 seed_counter += 1
229 
230     mean_clique_size = (min_clique_size + max_clique_size) / 2
231     sd_clique_size = mean_clique_size - min_clique_size
232 
233     mean_overlap = (min_overlap + max_overlap) / 2
234     sd_overlap = mean_overlap - min_overlap
235 
236     clique_list_1 = generate_rand_values(num_servers, min_clique_size,
    max_clique_size)
237     clique_list_2 = generate_normal_values(num_servers, mean_clique_size,
    sd_clique_size, min_clique_size,
```

```
                                                max_clique_size)
    clique_list_3 = generate_lognormal_values(num_servers, mean_clique_size,
sd_clique_size, min_clique_size,
                                                    max_clique_size)

    overlap_num_1 = len([x for x in itertools.combinations(clique_list_1, 2)])
    overlap_num_2 = len([x for x in itertools.combinations(clique_list_2, 2)])
    overlap_num_3 = len([x for x in itertools.combinations(clique_list_3, 2)])

    overlap_list_1 = generate_rand_overlaps(overlap_num_1, min_overlap,
max_overlap)
    overlap_list_2 = generate_normal_overlaps(overlap_num_2, mean_overlap,
sd_overlap, min_overlap, max_overlap)
    overlap_list_3 = generate_lognormal_overlaps(overlap_num_3, mean_overlap,
sd_overlap, min_overlap, max_overlap)

    ### initialize Ripple network
    if network_topology == 1:
        Ripple_net = clique_topology(num_servers, clique_list_1,
overlap_list_1, color_list)
    elif network_topology == 2:
        Ripple_net = clique_topology(num_servers, clique_list_2,
overlap_list_2, color_list)
    elif network_topology == 3:
        Ripple_net = clique_topology(num_servers, clique_list_3,
overlap_list_3, color_list)

    #nx.draw(Ripple_net, node_color=[Ripple_net.node[node]['color'] for node
in Ripple_net])
    #plt.show(Ripple_net)

    # Initializing server attributes: latency, server positions, server time
stamp and unl
    servers_list = []
```

```
263     for i in range(num_servers):
264         new_server = Server(i)
265         servers_list.append(new_server)
266 
267     for i in range(num_servers):
268 
269         # Comment the random seed function to generate different end to core
    latency values on each run
270         np.random.seed(i)
271         servers_list[i].e2c_latency = round(np.random.uniform(min_e2c_latency,
     max_e2c_latency))
272 
273         for j in range(num_servers):
274             if i == j:
275                 if (i % 2):
276                     servers_list[i].server_positions[i] = 1
277                     servers_list[i].server_time_stamps[i] = 1
278                     positive_servers += 1
279                 else:
280                     servers_list[i].server_positions[i] = -1
281                     servers_list[i].server_time_stamps[i] = 1
282                     negative_servers += 1
283             else:
284                 servers_list[i].server_positions[j] = 0
285                 servers_list[i].server_time_stamps[j] = 0
286 
287         servers_list[i].unl = Ripple_net.neighbors(i)
288         unl_sizes.append(len(servers_list[i].unl))
289         links = Ripple_net.neighbors(i)
290 
291         for k in range(len(Ripple_net.neighbors(i))):
292             link_server = links[k]
293             np.random.seed(k)
294             c2c_latency = round(np.random.uniform(min_c2c_latency,
```

```
    max_c2c_latency))
295             tot_latency = servers_list[i].e2c_latency + servers_list[
    link_server].e2c_latency + c2c_latency
296             servers_list[i].links.append(Link(link_server, tot_latency))
297             servers_list[link_server].links.append(Link(i, tot_latency))
298
299     unl_threshold = min(unl_sizes)
300
301     network = Network()
302
303     # Broacasting initial positions to the connected servers
304
305     for i in range(num_servers):
306         link_num = 0
307         for link in servers_list[i].links:
308             m = Message(i, link.link_to)
309             m.server_data[i] = ServerState(i, 1, servers_list[i].
    server_positions[i])
310             network.send_message(m, link, 0)
311             link_num += 1
312
313     num_events = len(network.events)
314     messages_created = 0
315     for event_time, event in network.events.items():
316         messages_created += len(event.messages)
317
318     simulation_summary.extend([num_events, messages_created])
319     consensus_summary = []
320
321     # Starts the consensus process until a final decision is reached
322     while True:
323
324         # Majority servers agree with the transactions
325         if positive_servers > (num_servers * (consensus_percent / 100)):
```

```
326             simulation_summary.append(1)
327             simulation_summary.append(positive_servers)
328             simulation_summary.append(negative_servers)
329             break
330 
331         # Majority servers disagree with the transactions
332         if negative_servers > (num_servers * (consensus_percent / 100)):
333             simulation_summary.append(-1)
334             simulation_summary.append(positive_servers)
335             simulation_summary.append(negative_servers)
336             break
337 
338         ordered_network_events = collections.OrderedDict(sorted(network.events
    .items()))
339         network.events = DefaultOrderedDict(Event)
340 
341         for event_time, event in ordered_network_events.items():
342             network.events[event_time] = event
343 
344         # Events are sequantially iterated
345         event_iter = iter(network.events)
346 
347         try:
348             event_time = event_iter.__next__()
349             event = network.events[event_time]
350 
351         except StopIteration:
352             simulation_summary.append(0)
353             simulation_summary.append(positive_servers)
354             simulation_summary.append(negative_servers)
355             break
356 
357         if int(event_time / 20) > int(network.master_time / 20):
358             iteration_summary = [event_time, positive_servers,
```

```
    negative_servers]
359             consensus_summary.append(iteration_summary)
360
361         network.master_time = event_time
362
363         # Transmission of messages
364         for m in event.messages:
365             if len(m.server_data) == 0:
366                 servers_list[m.message_from].messages_sent -= 1
367             else:
368                 servers_list[m.message_to].receive_message(m, network)
369
370         del network.events[event_time]
371
372     simulation_summary.append(consensus_summary)
373
374     message_count = 0
375     for event_time, event in network.events.items():
376         message_count += len(event.messages)
377
378     # Final Consensus results are captured
379     simulation_summary.extend([network.master_time, message_count])
380
381     total_messages_sent = 0
382
383     for i in range(num_servers):
384         total_messages_sent += servers_list[i].messages_sent
385
386     message_rate = int(total_messages_sent / num_servers)
387     simulation_summary.append(message_rate)
388
389     # all other prints statements are removed
390     print('Ran a simulation for ' + str(num_servers) + ' servers.')
391
```

```
392     return simulation_summary
```

Listing 3: Third Implementation code (Revised Network clique-like structure)

9.4.4 Testing Code

```
1
2  def UNL_Overlap_test_One(num_servers,servers_list):
3
4      print('-------------------------------------------------------------------------------\n')
5      print('CHECKING UNL OVERLAP CONDITION (TEST 1)')
6      print('-------------------------------------------------------------------------------')
7
8      for i in range(num_servers):
9          unl_1 = servers_list[i].unl
10         unl_1_size = len(unl_1)
11
12         for j in range(num_servers):
13
14             if i!=j:
15                 unl_2 = servers_list[j].unl
16                 unl_2_size = len(unl_2)
17                 unl_overlap = len(list(set(unl_1).intersection(unl_2)))
18                 if(unl_1_size>unl_2_size):
19                     if (unl_overlap>= 0.2*unl_1_size):
20                         pass
21                     else:
22                         print('\nPossibility of fork: ' + 'Insufficient UNL
    overlap between servers ' + str(i) + ' and ' + str(j))
23                         print('\nUNL of server ' + str(i) + ' : ' + str(unl_1)
    )
24                         print('\nUNL of server ' + str(j) + ' : ' + str(unl_2)
    )
25                         print('\nUNL overlap ' + str(list(set(unl_1).
    intersection(unl_2))))
```

```
                        return

                else:
                    if (unl_overlap>= 0.2*unl_2_size):
                        pass
                    else:
                        print('\nPossibility of fork: ' + 'Insufficient UNL overlap between servers ' + str(i) + ' and ' + str(j))
                        print('\nUNL of server ' + str(i) + ' : ' + str(unl_1))
                        print('\nUNL of server ' + str(j) + ' : ' + str(unl_2))
                        print('\nUNL overlap ' + str(list(set(unl_1).intersection(unl_2))))
                        return

    return
```

Listing 4: First Test snippet for UNL overlap

```

def UNL_Overlap_test_Two(num_servers,servers_list):

    print('-------------------------------------------------------------------------------\n')
    print('CHECKING UNL OVERLAP CONDITION (TEST 2)')
    print('-------------------------------------------------------------------------------')
    for i in range(num_servers):
        unl_1 = servers_list[i].unl
        unl_1_size = len(unl_1)
        for j in range(unl_1_size):
            server_id = unl_1[j]
            print(server_id)
            unl_2 = servers_list[server_id].unl
            unl_2_size = len(unl_2)
            unl_overlap = len(list(set(unl_1).intersection(unl_2)))
```

```
16            if (unl_1_size > unl_2_size):
17                if (unl_overlap >= 0.2 * unl_1_size):
18                    pass
19                else:
20                    print('\nPossibility of fork: ' + 'Insufficient UNL
    overlap between servers ' + str(
21                        i) + ' and ' + str(server_id))
22                    print('\nUNL of server ' + str(i) + ' : ' + str(unl_1))
23                    print('\nUNL of server ' + str(server_id) + ' : ' + str(
    unl_2))
24                    print('\nUNL overlap ' + str(list(set(unl_1).intersection(
    unl_2))))
25                    return
26            else:
27                if (unl_overlap >= 0.2 * unl_2_size):
28                    pass
29                else:
30                    print('\nPossibility of fork: ' + 'Insufficient UNL
    overlap between servers ' + str(
31                        i) + ' and ' + str(server_id))
32                    print('\nUNL of server ' + str(i) + ' : ' + str(unl_1))
33                    print('\nUNL of server ' + str(server_id) + ' : ' + str(
    unl_2))
34                    print('\nUNL overlap ' + str(list(set(unl_1).intersection(
    unl_2))))
35                    return
36
37     return
```

Listing 5: Second Test snippet for UNL overlap

```
def Clique_Overlap_test_One(G, clique_list):

    print('-------------------------------------------------------------------------\n')
    print('CHECKING Clique OVERLAP CONDITION (TEST 1)')
    print('-------------------------------------------------------------------------')

    for i in range(len(clique_list)):
        clique_1 = clique_list[i]
        clique_1_size = len(clique_1)

        for j in range(len(clique_list)):

            if i != j:
                clique_2 = clique_list[j]
                clique_2_size = len(clique_2)
                clique_overlap = 0
                for k in range(clique_1_size):
                    for l in range(clique_2_size):
                        if G.has_edge(k,l):
                            clique_overlap+=1

                if (clique_1_size > clique_2_size):
                    if (clique_overlap >= 0.2 * clique_1_size):
                        pass
                    else:
                        print('\nPossibility of fork: ' + 'Insufficient
   overlap between cliques' + str(
                            clique_1) + ' and ' + str(clique_2))
                        return

                else:
                    if (clique_overlap >= 0.2 * clique_2_size):
                        pass
                    else:
```

```
34                      print('\nPossibility of fork: ' + 'Insufficient
    overlap between cliques ' + str(
35                          clique_1) + ' and ' + str(clique_2))
36                      return
37     print('Test Passed')
38     return
```

Listing 6: Test snippet for clique overlap