University of Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study programme/specialisation: Computer Science: Reliable And Secure Systems	Spring , 2023 Open
Author: Adnan Ahmed	
Programme coordinator: Leader Jehl Supervisor(s): Leander Jehl, Arian Baloochestani Asl	
Title of master's thesis: Simulating and comparing Tangle 2.0 and PoW	
Credits: 30	
Keywords: Blockchain, Tangle 2.0, Proof-of-work(POW), Bitcoin, DAG	Number of pages: 54 + supplemental material/other: ………… Stavanger, 14.06.2023 date/year

Title page for master’s thesis
Faculty of Science and Technology

Simulating and comparing Tangle 2.0 and PoW

Adnan Ahmed

June 2023

Acknowledgements

I would like to express my sincere gratitude to my supervisors, Leander Jehl and Arian Baloochestani Asl, for their guidance, support, and invaluable contributions throughout the course of this research project. Their expertise, insightful feedback, and continuous encouragement have been instrumental in shaping the direction and quality of this work. I am truly grateful for their dedication and commitment to my academic and personal development.

Abstract

Tangle 2.0[5] is a leaderless probabilistic consensus protocol that is based on a DAG called Tangle. The way consensus is found is in the heaviest DAG and not in the longest chain which is popular in Blockchain. POW is a consensus protocol most known for being used in Bitcoin. The nodes in a POW system have to solve a complex mathematical puzzle that satisfies a difficulty threshold before a new block can be published to the network. Because Tangle 2.0 is such a complex protocol we wish to find out if it has an edge over a traditional consensus protocol such as PoW. We modify the Tangle 2.0 simulating tool[6] by adding POW as the consensus protocol. We compare the original Tangle 2.0 with POW by metrics such as the confirmation time of network messages. Adding POW as the consensus protocol in Tangle 2.0 creates a greedy-heaviest sub-tree structure which has led to forks. We have found that because of forks in the POW network, slowing down the network throughput will eventually rid the network of forks. But leads to a higher confirmation time. So the comparison shows a significant benefit for Tangle 2.0.

Contents

Chapter 1

Introduction

Blockchain technology has become a topic of significant interest in recent years due to its potential to disrupt various industries by enabling decentralized and trustless systems. The original Tangle protocol[1] is a directed acyclic graph (DAG) structure that served as the underlying technology for IOTA cryptocurrency. It was designed to overcome the limitations and scalability issues present in traditional blockchain systems. In the Tangle transactions are not organized into blocks, but rather a network of interconnected vertices where each transaction references two previous transactions. Tangle 2.0[5] is an enhanced version of the original Tangle that also uses a Directed Acyclic Graph (DAG) instead of a linear chain to store transactions. Tangle 2.0 addresses some of the limitations of the original version. Some of the key differences between the Tangle and Tangle 2.0 are scalability, Tangle 2.0 introduces sharding which allows for parallel processing of transactions. A new consensus mechanism is also implemented, the original Tangle relied on the cumulative weight of transactions to determine consensus while Tangle 2.0 called the Tip Selection Algorithm. Proof-of-work(POW) is a consensus protocol widely used in traditional blockchain systems to achieve consensus in blockchain networks. In a blockchain system using the POW protocol participants are called miners. The miners compete to solve a complex mathematical puzzle, the goal of the puzzle is to find a solution that satisfies a certain criteria. This criterion is usually a specific target or difficulty value. POW is widely used and is most known as being used in the Bitcoin cryptocurrency system.

TangleSim[6] is a new simulation tool for the Tangle, which allows researchers to study the behavior of the Tangle under different conditions. There is a necessity for simulation tools for blockchain systems to understand the behavior of the blockchain systems, evaluate the performance of the system and other important reasons such as optimizing the protocol and finding faults and security flaws in the system. With a simulation tool, one can also gain information about the cost and resource usage of the system instead of finding this type of information while the system is live and online. Some of the good features of the TangleSim are the strong and well-implemented network structure and the node-to-node interconnectivity. Building a consensus layer on top of this solid foundation and maybe making it generalized so to work with any kind of blockchain would be interesting.

In this thesis, we propose remodeling TangleSim to accommodate Proof-of-Work (POW) consensus protocol and its existing consensus algorithm. The addition of POW consensus will provide researchers with the ability to compare and analyze the performance of the Tangle under different consensus algorithms. Currently, there are several simulation tools[9][10][11] available for studying blockchain technology. However, most of these tools focus on the traditional blockchain and lack support for DAG-based blockchains like the Tangle. Therefore, the proposed remodeling of TangleSim will contribute to the existing literature by providing a powerful tool for researchers to study the behavior of Tangle 2.0 implemented with POW. Our research is published to a GitHub repository and will be available there[3].

1.1 Motivation and Problem Description

The expected contribution of this thesis is to provide a detailed analysis of the performance of the Tangle under POW consensus in addition to its existing consensus algorithm. The findings of this research will have important implications for the development and deployment of blockchain systems. The scope of this research will be limited to the remodeling of TangleSim to accommodate POW consensus. We remodel this simulation tool because we wish to find out if there is a possibility of running other consensus protocols in a Tangle 2.0 setting, in our case, POW is the chosen consensus protocol because of its wide use. The limitations of this research include the fact that it is based on simulation results, which may not accurately reflect the behavior of a real-world blockchain system.

The comparison of Tangle 2.0 with POW against the original Tangle protocol will provide valuable insights into the trade-offs, advantages, and limitations of incorporating POW in the Tangle. By analyzing performance metrics such as transaction confirmation times, throughput, security against various attacks, and network stability, we can better understand the impact of POW on the Tangle's overall performance and its suitability for real-world applications. In summary, the motivation behind this study is to explore and evaluate the Tangle 2.0 simulation tool and layering POW on top as a consensus mechanism. By extending the TangleSim simulation tool to support POW-based simulations, we aim to compare the performance and behavior of Tangle 2.0 with POW as the consensus protocol against the original Tangle 2.0 protocol simulation. This analysis will provide valuable insights into the potential benefits and challenges associated with the integration of POW in the Tangle, ultimately contributing to the advancement of decentralized ledger technologies.

Chapter 2

Background

2.1 Blockchain

The technology known as *Blockchain* was first introduced in Bitcoin[13]. It was proposed as a peer-to-peer ($P2P$) payment system that goes from a sending party, directly to the recipient without needing to go through a financial institution. This paper laid out the mathematical groundwork for the Bitcoin cryptocurrency. Blockchain technology is the foundation of cryptocurrency but it also opens a new door for the financial industry. Trust in distributed systems has always been a major issue and in blockchain systems. Because for there to be a secure transaction or distribution of any kind of information trust is the first component. This problem was solved in *Sompolinsky et al*[14] was the problem of establishing trust in a distributed system. In the paper, there is highlighted the inherited limitations of centralized systems, limitations such as the need for trust in intermediaries and the potential for censorship and control. The solution proposed by the authors was to implement Greedy-Heaviest Observed Sub-tree(GHOST). The GHOST protocol addresses this problem by introducing a novel way to select which blocks should be considered as part of the blockchain. We explain the $GHOST$ protocol in more detail in the GHOST section later in this chapter.

To address the other issues, the authors propose a decentralized solution based on blockchain, which is a public ledger that records transactions in a transparent and immutable manner. A transaction is defined as a transfer of bitcoins from one Bitcoin address to another. Each transaction contains input and output. Input in a transaction refers to the Bitcoins being spent which are sourced from previous transactions or mined, while the outputs represent the recipient address and the amount being transferred. What we explained was a fundamental form of the transactions. In the real implementation, the sender has to digitally sign the transaction with their private key to prove ownership. This signature ensures the authenticity and integrity of the transaction.

The transactions are published in blocks, the block structure proposed by *Nakamoto et alt*[13] in this paper refers to a data structure that is containers for the transactions. It is a vital component of the Blockchain hence the name. Bitcoin defines a block as being a collection of transactions, timestamps along with important metadata. Each block has a header and a list of transactions, the header contains the hash of the block which is unique, the hash of the previous block, the timestamp, and the nonce value used for the proof-of-work process. We explain POW in more detail in section 2.2. Once a block has been mined by a miner, it broadcasts it to the network and other mines will validate the content of the block and the proof-of-work. Once validated, the block is added to the Blockchain. The block and the transactions become a permanent part of history. The purpose of the Blockchain is to provide a means of ordering and organizing transactions in a way that makes them verifiable and tamper-resistant.

2.2 Proof-of-work (POW)

Proof-of-work is a consensus algorithm used in blockchain systems to decide which that should be included in the ledger and attain network consensus. It's one of the most used consensus algorithms in blockchain systems, it's commonly associated with Bitcoin and other cryptocurrencies. POW was originally introduced to the blockchain world as a solution to the double-spending problem that has been a common problem in blockchain systems. The Double-spending problem is a problem that refers to a unit of currency spent twice, hence the name double-spending. The POW algorithm is designed to require enormous computational prowess to add new blocks to the chain and also validate transactions. This is intended to discourage malicious actors to carry out double-spend. [4]

The way POW works in blockchain-based systems is that the miners compete with each other to solve a complex mathematical test, this is where the computation power comes in as the miner with greater computational power is faster than the others. The first miner to solve the puzzle is rewarded with an amount of cryptocurrency unit which is also an incentive for the mining to continue. The difficulty of the mathematical puzzle is continuously adjusted to keep the block time consistent. Despite its good qualities, POW has received criticism concerning its high energy consumption[15] and the issue with mining power centralization. Mining power centralization is the issue of a few miners having a disproportionate amount of the total amount of computational power, and the high energy consumption is also a huge problem because the hardware used to run the mining operation for solving the puzzle is expensive.

2.3 GHOST

Greedy-Heaviest-Observed Sub-tree (GHOST) is a blockchain consensus protocol first introduced in [14]. This protocol was introduced as a modification to the original Bitcoin protocol with the aim of enhancing the scalability and transaction processing capabilities of the network.

In the original Bitcoin Blockchain, the new blocks are added to the main chain based on the longest chain rule. The main chain is the chain with the most accumulated proof-of-work and is considered the valid chain. This approach can lead to the reduction of transaction throughput and gives rise to longer confirmation time as the network grows.

The $GHOST$ protocol addresses these limitations by introducing a different way of selecting the main chain and determining valid blocks. $GHOST$ takes into consideration the branches and sub-trees that contain a significant amount of computational effort instead of only the longest chain. In $GHOST$ when a new block is received, the selection of the main chain is based on the $heaviest-other$ rule. The protocol includes the blocks from other sub-trees that have accumulated a substantial amount of computational work in addition to the longest chain. This allows for more utilization of the network's computational power, increasing the transaction processing capabilities. The $GHOST$ protocol also makes a modification to how the block reward is distributed. In Bitcoin, the block reward is given to the miner that mines the block, but in $GHOST$ the reward is distributed among the miner of the block, but also the miners of the blocks that were included from other branches. This distribution tactic incentivizes other miners to include blocks from other branches, this also promotes more efficient use of computational power.

The key benefit of $GHOST$ is its ability to handle higher transaction rates compared to the original Bitcoin, also by including blocks from other branches $GHOST$ reduces the creation of orphaned blocks and increases overall transaction confirmation speed.

2.4 Tangle

The Tangle is a concept introduced first in $TheTangle$[1]. The author presents the concept and principles, which serve as the underlying technology for $IOTA$ cryptocurrency.

The Tangle is presented as an alternative to the traditional Blockchain architecture. It aims to address some of the limitations and scalability issues found in traditional Blockchain systems. It is introduced as a decentralized, scalable, and feeless distributed ledger. The Tangle makes use of a directed acyclic graph (DAG), where each transaction is a node and they are connected through a "web" of references.

The Tangle consensus protocol is at its core a directed acyclic graph (DAG), where every transaction is represented as a node. It is different from traditional Blockchain systems where blocks are ordered in a sequential manner, the Tangle allows for multiple transactions to be referenced by a single transaction, creating a web-like structure. The main innovation introduced in the Tangle is the new approach to consensus and validation. In contrast to the traditional Blockchain systems that rely on miners and validators, the Tangle makes use of a consensus mechanism called $Markos\ Chain\ Monte\ Carlo$ ($MCMC$) to determine transaction validity and ordering. This mechanism is based on a cumulative weight, attributed to each transaction, this attribute reflects the level of trustworthiness in the network.

Participants in the Tangle are called nodes, these nodes validate transactions by approving two previous transactions referenced by the transaction. This action allows nodes to indirectly contribute to the confirmation of earlier transactions, this creates a network of inter-dependencies (See Figure 2.1). This consensus structure makes transactions become more secure as they gain more approvals from subsequent transactions.

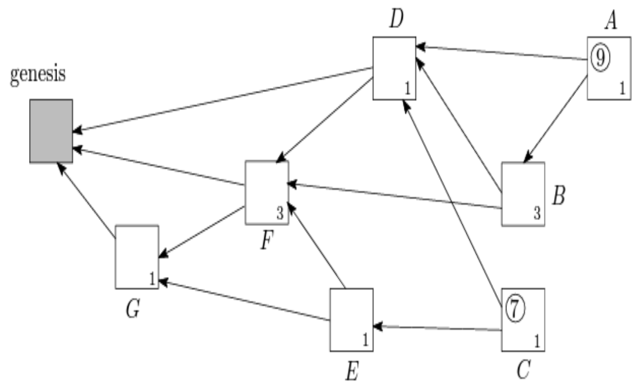


Figure 2.1: DAG with weights assigned to each site, and scores calculated for A and C. [1]

In the paper, the author discusses various properties and features of the Tangle. A notable property is the Tangles' potential for being highly scalable because the structure allows for parallelization and efficient transaction processing.

The Tangle at its core is fee-less, this is a notable feature that makes micro-transactions and other types of payments more feasible and cost-effective.

2.5 Tangle 2.0

Tangle 2.0 is a concept proposed in [5] and is a new approach to attaining consensus in a distributed ledger technology (DLT). It's based on the concept of directed acyclic graphs (DAG), this concept uses DAG to represent transactions in a ledger. In contrast with traditional blockchain-based DLTs, Tangle 2.0 does not have the standard miner/validator or stakers system to validate or create blocks. Tangle 2.0 uses instead a new and original consensus mechanism called the Leaderless Nakamoto Consensus on the Heaviest DAG, this consensus mechanism allows for a completely decentralized and permissionless system. A new concept in Tangle 2.0 introduced in[5] is something called approval weight, which is a measurement assigned to each transaction in the DAG, and what this metric measures is the number of transactions approving a given transaction, (in)directly. The approval weight is used for validating transactions, basically, the transactions with a higher approval weight have a higher chance of being included in the heaviest DAG, This approach makes the system more efficient and secure.

A different but important aspect of Tangle 2.0 is the aspect of confirmation. The process of a transaction being confirmed ends in a given transaction being added to the DAG. So confirmation refers to the process of checking the validity of a transaction and adding it to the DAG. The confirmation process itself is carried out by the nodes in the network, the nodes themselves have to approve a transaction before it can be considered confirmed. To further secure the network and make it resilient to double-spending Tangle 2.0 introduces and incorporates a concept called Synchronized Random Reality Selection (SRRS). SRRS ensures that the nodes in the network can't manipulate the transactions to be confirmed, because the selection is done randomly, and is agreed upon by all the nodes it prevents actions like double-spending.

In addition to SRRS Tangle 2.0 also introduces a concept called On Tangle Voting(OTV). This mechanism allows the nodes in the network to vote for a specific transaction by attaching their vote directly to a given transaction. The votes are then propagated throughout the network and the nodes have the reach a consensus regarding the validity of the transaction. This mechanism provides an additional layer of security to the network.

In summary, Tangle 2.0 is a significant upgrade to the previous Tangle technology. It offers solutions to some of the challenges that were faced by its predecessor. The protocol uses a weight system to select tips for approving new transactions, we will explain this in depth later. The design of the protocol work in a way such that nodes that have contributed more to the security of the network are given more access to the network's resources, in more voting power. The changes stated above allow Tangle 2.0 to handle higher transaction throughput while also maintaining the security of the network.

2.5.1 DAG

The Tangle is the DAG that stores transactions of the distributed ledger. In current systems running the traditional blockchain system, there is an established total ordering of transactions along with a lack of parallel confirmation. In the Tangle there is no total ordering but rather partial ordering in the sets of vertices in the DAG. Partial ordering in the context of Tangle means that the transactions in the DAG are not strictly ordered in a linear sequence but instead can have different ordering within a subset or branches of the graph. In partial ordering, some of the transactions might have a direct relationship like parent and child while others might have an indirect relationship making their order ambiguous. Sybil protection plays an important role in a permissionless system where there is no need for permission and everyone can participate. Bitcoin's POW consensus protocol where an enormous amount of energy is wasted along with other negative properties leads to the development of more sustainable avenues. Proof of stake (POS) was one of the more sustainable consensus protocols but it also like POW has issues like giving proportional control over the blockchain network to stakeholders making it impartial. Some of the advantages DAG has over traditional blockchain systems that make it more ideal are, lower fees or even fee-less, much more scalable because of concurrent processing of transactions, and fast confirmation because of said scalability.

The Tangle 2.0 protocol includes a novel transaction model known as the "transaction model with colored coins," in contrast to conventional blockchain systems that make use of the Unspent Transaction Output (UTXO) model. The Tangle network can now track ownership and financial transactions thanks to this creative technique. Transactions can carry additional information, such as asset types or particular features related to the transferred value, by using colored coins. The inclusion of this sophisticated transaction model in Tangle 2.0 improves the distributed ledger's flexibility and functionality in terms of how transactions are represented and managed. A more detailed representation of digital assets is possible because of it, and a variety of use cases beyond straightforward money transfers are made possible.

2.5.2 Approval Weight and Confirmation

The core element of the consensus protocol is Approval Weight (AW). It allows for the measuring of endorsement of a given block and its corresponding transaction. By summing up the cumulative weight of all issuers of blocks in the future cone of the given block that contains the transaction one can find the confirmation of a transaction.

2.5.3 Synchronised Random Reality Selection & Asynchronous On Tangle Voting

Tangle 2.0 consensus protocol consists of two components: the Synchronised Random Reality Selection (SRRS), and the asynchronous On Tangle Voting. Because of the OTV asynchronous nature, there is a lack of synchronization process between nodes. To overcome the FLP (Fischer, Lynch, and Paterson) Impossibility result. The reason the FLP result needs to be overcome is because of the impossibility of creating a consensus algorithm that guarantees both $safety$ and $liveness$. This result is significant because it shows the fundamental limitations of achieving consensus in an asynchronous system. So synchronization of the nodes is done in SSRS using a shared random number generated periodically. A period D (epoch) time and of each period D every node in the system receives the newly generated number, this common number (coin) is used to synchronize the opinions of the nodes. Tangle is both a record of the communication between the nodes and also facilitates the data structure for the voting scheme. The structure of the Tangle is split into the referenced blocks' parents and the approving blocks' children. All blocks referenced directly or indirectly the past cone of a given block, and the blocks that refer to the block directly or indirectly are called the block's future cone. Only the first block on the Tangle is called the Genesis block and has no parents.

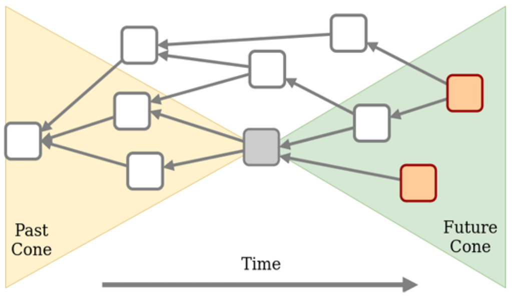


Figure 2.2: Past and future cone of a block, they are highlighted in the green and yellow triangles respectively.[6]

The On Tangle Voting (OTV) is the component of Tangle that allows voting on double spends or transactions that are conflicting, A and B. A node can have an opinion on a promising block "A" by issuing a block in the future cone of the block and later if a conflicting block "B" that is better with regards to Approval weight(AW) later revokes its weight from block "A". In any pair of conflicting transactions, the weight of a node is counted only once. A transaction is finally confirmed when its AW reaches θ, even if later the AW of the transaction falls below this threshold it will stay confirmed.

2.6 Tangle 2.0 simulation tool

The Robustness of the Tangle 2.0 Consensus paper [6] provides a comprehensive analysis of the Tangle 2.0 consensus algorithm's robustness against several attack scenarios, including the double-spend and eclipse attacks. The paper's findings suggest that the Tangle 2.0 consensus algorithm is highly resilient to these attacks, making it a robust and secure alternative to traditional blockchain-based DLTs. The paper itself is a study and implementation of the original Tangle 2.0 Leaderless Nakamoto Consensus on the Heaviest DAG [5]. That paper outlines the Tangle 2.0, the DAG structure, and the theory and definitions behind the voting, and voting sets. In the paper node participation and block structure along with estimated confirmation time and confirmation is detailed and explained.

Tips

We will first define what a strong(weak) tip is. In the TangleSim simulation tool, a *tip* refers to a *message* that has yet to be directly or indirectly approved by any other *messages*. A *Strong tip* is a tip that has a high number of indirect approvals, this means it has been approved indirectly through a chain of *messages*. To give an example, suppose a *message A* approves *message B*, and *message B* approves *message C* this we can say that *message A* indirectly approved *message C*.

On the other hand, a *weak tip* is a tip that has fewer indirect approvals or none at all. A *message* such as that has a low probability of being confirmed in the system in the future.

2.6.1 UTXO and Sybil Protection

Tracking of funds and how they change ownership in nodes is done through Unspent Transaction Output (UTXO) model. In this model, a transaction is specified as the output of the previous transaction as input, and the expenditure as output. This allows for the verification of a transaction without knowing the global state, in contrast to account-based models. Because consistency is the main requirement for the ledger, the nodes have to eventually resolve any conflicts and double-spends. The Tangle 2.0 consensus protocol relies on identity-based Sybil protection to decide between conflicting expenditures.

Every node in the system has a weight, which serves as a Sybil protection mechanism. The weight is employed voting power and corresponds to the amount of access to the network's resources. In this implementation of Tangle, we assume that the node weights stay constant and that all the nods use their total share of the network throughput, also the total weight of the network is the sum of all the node weights.

2.6.2 Colors and Conflicts

As we described earlier in section 2.5.3 nodes can reference transactions that are conflicting and this is shown through the use of colors in the implementation. In the original paper [5] there were two types of references, block reference, and transaction reference. In this simulation tool, only the basic block reference is considered and referencing blocks not on the same branch is not allowed. Looking at Figure 2.3 if one considers the state of the Tangle until time $t1$ (dashed line), we can count the support for the branches by summing up the block issuers weights.

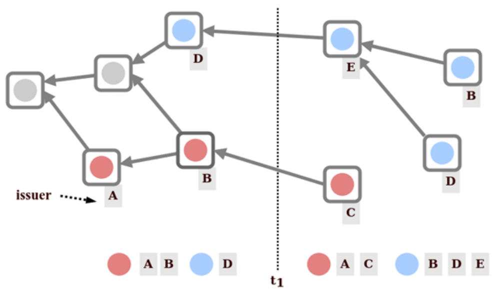


Figure 2.3: Calculation of AW done by tracking supports.[6]

2.6.3 Network Layer and Weight distribution

The environment that is simulated is a reflection of a peer-peer network topology where each node has k numbers of neighbors. In this environment, the nodes can receive blocks, request missing blocks, and state opinions in case of conflict. This network is set up using a Watts-Strogatz network [7], this network mimics reality and allows for the specification of network delays and loss of packets.
The weight of all the nodes in the network is distributed with the use of Zipf empirical law. This law is proven to govern an asymptotic distribution of weight close to its upper tail [8]. The advantage of Zipf law is that by changing the parameter s, one can model the behavior of a network for different degrees of (de)centralization. Its to be noted that in the simulation there is no change in the node weights due to transactions and other events.

In the simulation, nodes act as different and independent asynchronous agents. This means that the nodes have different perceptions of the Tangle, thus the nodes have their own local Tangle that they use to calculate weights and run the protocol.

Chapter 3

Related Works

3.1 Chain Sim

ChainSim:[9] is a paper that discusses the motivation for the research and the limitations of the current set of existing simulation tools. A problem with a majority of the existing tools is that they are designed with P2P networks, but this creates not-so-accurate reflection when used on hybrid blockchain networks. So with this motivation, the researchers argue that a simulation tool for hybrid blockchain networks is needed to further the knowledge of their performance and behavior.

To address this need, the authors have proposed a simulation framework that models the behavior of hybrid blockchain networks. The design for this framework is modular and extensible, this allows for the customization of the tool to fit the researcher's needs. The framework is implemented in Python and uses Pycharm.
The architecture of the ChainSim tool consists of four main modules: the blockchain module, the client-server module, the P2P network module, and the simulation module. The blockchain module models the creation of a block, the behavior of a block, block confirmation, and block propagation. The client-server module models how the client-server behaves, client requests, server responses, and data storage. The P2P network module models the P2P network while also modeling message propagation, peer discovery, and peer participation. The last module of the main modules is the simulation module and in this module, the coordination and interaction between the other three modules is done along with generating simulation results.

In ChainSim, the authors describe some of the experiments they conducted using the simulation tool and evaluated the performance of several blockchain protocols in a hybrid architecture. By conducting these experiments they demonstrate the flexibility of the framework and its ability to be customized according to different needs.

In conclusion, ChainSim: A P2P Blockchain Simulation Framework is a valuable addition to the blockchain research field. The simulation framework proposed in the paper which is designed specifically for hybrid blockchain networks, addresses a missing gap in the current field of simulation tools.

3.2 DAGSim

DAGsim[10] is a framework for directed acyclic graph (DAG) based distributed ledger protocols and was published in the proceedings of the IEEE International Conference.

The paper discusses the need for simulation tools for the simulation of DAG-based distributed protocols. The scalability and throughput advantages over traditional blockchain systems have gained DAG-based protocols popularity in newer times. The evaluation of the performance of DAG-based protocols using analytical models and(or) empirical measurements can be challenging due to the complex structure of such protocols. Therefore simulation tools provide better accurate and efficient means of evaluating DAG-based protocols. These points make up the motivation behind this research paper.

To address these issues and needs, the authors of this paper propose DAGsim. DAGsim is a simulation framework for DAG-based distributed ledger protocols, the framework is designed to be both modular and extensible in its uses. This allows the users to customize according to their specific needs.

The modular-based framework is described by the authors as a design that allows for easy integration of new components. The DAGsim framework consists of 4 main modules, modules are the DAG module, the node module, the transaction module, and the simulation module. The DAG module models the behavior of the DAG structure, and the node module models how the nodes behave in the system along with the propagation of transactions and verification of transactions. The transaction module models the transaction's behavior along with the creation and validation of transactions. The four modules model a different part of the system and it's the simulation module that coordinates the interactions between the modules, this module is also responsible for

The authors have conducted several experiments using the framework to find how well it performs under different DAG-based protocols and to find out how flexible and extensible it is. They also discussed the insights they had gained through the experiments they conducted, they discussed the impact of transaction arrivals on the throughput, the impact network size had on the validation of transactions, and the impact of different consensus mechanisms on network security.

They examine the impact the network size had on validation, they discussed in the paper that as the network grew in size, the chance of a consensus being reached decreased due to the increased amount of conflicting transactions in the network. The reason for this is that the validation system of a DAG-based ledger relies on a concept called approval weight (AW), which is a measure of how many earlier transactions have been approved for a given transaction. The paper attributes this to the fact that DAG-based ledgers don't have a global ordering of transactions like traditional blockchain systems have. In the case of the impact on the transaction validation had on the throughput of the network, the paper shows that as the rate of transaction increases the throughput of the network decreases. This is due to the fact that each new transaction needs validation by a number of previous transactions before it's confirmed. As the number of unconfirmed transactions increases, the time required to validate them increases, leading to a decrease in network throughput.

The simulation results show that the high rate of transaction arrivals leads to a lower throughput due to the increased number of transactions waiting to be validated. To summarize, the DAGsim paper highlights the importance of evaluating the security and performance of DAG-based distributed ledger protocols using simulation tools. It also provides insights into the impact of network size and transaction arrivals on the validation of transactions and throughput, which can be useful in the development and optimization of DAG-based distributed ledger protocols.

In conclusion, the DAGsim paper presents a simulation tool that can be used to evaluate the performance of DAG-based distributed ledger protocols. The paper highlights the importance of network security and presents a simulation approach that allows for the evaluation of network security in various network conditions. The paper also investigates the impact of network size on the validation of transactions. The results show that larger networks require more time to validate transactions and, therefore, lower transaction throughput. This suggests that scalability is a critical challenge that must be addressed in DAG-based distributed ledger protocols. Finally, the paper examines the impact of transaction arrivals on the throughput of DAG-based distributed ledger protocols.

3.3 BlockSims

BlockSim is a comprehensive simulation framework developed for blockchain systems, proposed in [11]. The paper presents an overview of the BlockSim framework, architecture, design, and key features.

The paper discusses the limitations of the existing simulation tools for blockchain systems. The authors argue that most existing simulation tools are specifically designed for specific blockchain protocols or can't accurately reflect other protocols' behavior. They argue that a simulation tool that has the capability to model different blockchain systems is needed to better understand the behavior and performance of different types of blockchain systems. The motive the authors had in mind when developing BlockSim was to create a reliable and efficient platform for evaluating performance and security in a blockchain system.

BlockSim is proposed by the authors to address this need and this framework is designed as a modular framework. The BlockSim framework is modular and flexible, this allows the researcher to use the framework to customize it according to their specifications.

BlockSim's architecture is based on a modular design that allows for easy integration of new components. The framework includes several modules: blockchain, network, user, transaction, and simulation. The blockchain module models block creation, validation, and propagation. The network module models message propagation, peer discovery, and peer churn. The user module models transaction creation and validation. The transaction module models transaction creation, validation, and propagation. The simulation module coordinates interactions between modules and generates simulation results. The framework supports various types of blockchain systems both public and private networks. Some of the systems BlockSim can simulate are, proof-of-work (POW), proof-of-stake (POS), and also hybrid consensus algorithms.

The authors conducted experiments using BlockSim to evaluate different blockchain systems. The paper details several experiments conducted using the BlockSim framework, the experiments are done with the purpose of evaluating the performance and security of the system under various conditions. One such experiment evaluates the blockchain system performance by varying the number of nodes in the system. Another experiment evaluates the effect of block time (the time it takes for a block to be added to the chain) and block size on network throughput. In the paper security-related metrics are presented, i.e. Fork rate, 51% attack resistance, and byzantine fault tolerance. These metrics are used in concert with experiments in order to test the resilience of the system, use these metrics to find out how blockchain systems like POW and POS perform and how they achieve consensus and ensure security.

BlockSim is a valuable contribution to blockchain research, addressing a gap in existing simulation tools. The framework is flexible, scalable, and easy to use, making it useful for researchers. The experiments demonstrate the potential of BlockSim to generate insights and inform new protocols.

Chapter 4

Methodology

4.1 Design Background

This research is done using the simulation tool proposed in[6]. The simulation tool TangleSim is designed to model the Tangle protocol that underlies the IOTA cryptocurrency. The tool is designed to allow researchers and developers to test different scenarios and evaluate the performance and security of the Tangle protocol. This simulation tool consists of several different components that work together to provide a comprehensive simulation tool. In this section, we introduce and explain the different components of the tool and discuss how they work together and how to alter or modify them according to different needs. Our own modification to the tool for our research will be discussed in detail later in the Methodology section.

There are five main modules and a configuration module for the simulation settings. The three main modules are the Adversary module, the multiverse module, the network, the scripts module, and the simulation module.

4.1.1 Adversary Module

The adversary module is the module where the malicious nodes are defined. In the simulation tool, the authors have created 3 different adversary types, namely the *no_gossip* node, the *same_opinion* node, and the *shifting_opinion* node. These nodes are meant to simulate different aspects of the real-life scenario, where a node may be inactive and not answer any request or gossip, a node may also be the type of node that just follows the wave and always aligns its opinion with the masses, and lastly the normal occurrence of a node that is always shifting opinion and is chaotic in nature. Each adversary type is defined in its own file and each adversary type has its own unique functions defined to regulate behavior consistent with the node type. To give a few examples, the *no_gossip* node has no outside communication with other nodes or the system, it does not answer any request from other nodes for missing messages and does not issue any message. The *shifting_opinion* node has functions like *formOpinion* and *getMaxOpinion*, the first function is used for updating the opinion of the node when a new there is a better opinion available. The second function is responsible for finding the opinion in the network(the nodes view of the network) and selecting the opinion with the highest weight, then the node forms a new opinion and updates the weights. As for the *same_opinion* node it has functions for forming an opinion and updating the weight but in the function for updating the weight, it's blank.

```
func CastAdversary(node network.Node) NodeInterface {
   s := reflect.ValueOf(node)
   switch s.Interface().(type) {
   case *ShiftingOpinionNode:
      return node.(*ShiftingOpinionNode)
   case *SameOpinionNode:
      return node.(*SameOpinionNode)
   case *NoGossipNode:
      return node.(*NoGossipNode)
}
         return nil
}
```

Caption : A node from the network is cast to either of 3 *types of Adversary nodes, adversary.go*

In the adversary module, there is a file for casting nodes to either of the three adversary types (see *Figure* 4.2). This function is used when simulating the double-spend attack in the network. The number of adversary nodes is specified in the configuration, but the default value is 1.

```
    func SimulateDoubleSpent(testNetwork *network.Network) {
     time.Sleep(time.Duration(config.DoubleSpendDelay*config.SlowdownFactor) * time.Second)
     // Here we simulate the double spending
     dsIssuanceTime = time.Now()

     switch config.SimulationMode {
     case "Accidental":
     for i, node := range network.GetAccidentalIssuers(testNetwork) {
     color := multiverse.ColorFromInt(i + 1)
     go sendMessage(node, color)
     log.Infof("Peer %d sent double spend msg: %v", node.ID, color)
     }
     case "Adversary":
     for _, group := range testNetwork.AdversaryGroups {
     color := multiverse.ColorFromStr(group.InitColor)
     for _, nodeID := range group.NodeIDs {
     peer := testNetwork.Peer(nodeID)
     // honest node does not implement adversary behavior interface
     if group.AdversaryType != network.HonestNode {
     node := adversary.CastAdversary(peer.Node)
     node.AssignColor(color)
     }
     go sendMessage(peer, color)
     log.Infof("Peer %d sent double spend msg: %v", peer.ID, color)
     }
     }
     }
}
```

Caption : In the case of Adversary, honest nodes do not implement adversary behavior, a node is cast to Adversary. main.go

4.1.2 Multiverse Module

The multiverse module is the module that is responsible for a majority of the processes that are being used in the simulation tool. This module is responsible for the important aspects of the tool. We explain all the parts of this module. This module is compromised of 11 files, we explain in short and concise words what part of the system each file is responsible for.

approvalweight_manager.go and booker.no The first file in this module is *approvalweight_manager.go*, the file is responsible for managing the approval weights of the nodes in the network. It determines the level of power a node has in validating a transaction. Its task is to keep track of the approval weights of nodes, it keeps track of the number of incoming and outgoing transactions of a node and the cumulative weight of the node, and the tips it approves. The approval weight of a node is an important factor in determining the probability that the node's message will be selected for approval and confirmed by other nodes. The method that does this is the *ApproveMessages*(), this method starts of by retrieving the message from the message database, then using a *byte map* we get the weight of the issuer of the message. After the weight is checked and information is logged a *Walker* is initiated to traverse the message *DAG* down to *genesis* to properly calculate the approval weight.

The next file in the multiverse module is *booker.go*, this file is responsible for initiating new transactions and the color inheritance for the messages and their parents.

messagefactory.go *messagefactory.go* is as the name states the place where messages are created, and have their fields filled in. The function *CreateMessage* is called upon in *node.go*, where the nodes handle different types of messages arriving.

models.go The *models.go* file contains the definitions of different data models used throughout the simulation. It defines *structs* such as, a *Message*, the *MessageMetaData*, a *MessageRequest*, the *MessageID*(s) and some methods to go along with *MessageIDs*, and the *color* structure.

node.go In the *node.go*, the behavior of a single node is defined. This includes the setup of a node, the becoming a part of the network as a peer, the function *IssuePayload* for sending color to the socket for creating new messages, and the function *HandleNetworkMessage* which works with functions from both the *messagefactory.go/CreateMessage* and *tangle.go/ProcessMessage*. This *HandleNetworkMessage* function receives messages of 3 different types, namely a *MessageRequest* which is a request for re-sending a missing message to a peer, the *Message* which is a new message from the network that the nodes need to process, and the last type is *color* which is a color the nodes receives and attaches to a new message as an opinion.

opinion_manager.go The *opinion_manager.go* is the component in the system that manages the opinions of nodes and the tips they see in the Tangle. Everything from forming an *opinion* and updating weights, to updating the *confirmation* for tips that the nodes assign an opinion to. The *opinion manager* updates the *opinion score* of the tips as the nodes in the network issue new messages. It also checks and confirms the colors of messages accordingly to find the opinion with *max* score in the color branches.

requester.go The *Requester.go* is where the requesting of missing messages is defined. This component of the system is vital as it deals with filling missing gaps in the local Tangle of nodes. It has a few functions defined, there is *StartRequest*, which checks if the missing the *messageID* is in the queued elements, if this is the case nothing will happen but if the *messageID* is not in the list of queued elements it will trigger a request for the missing *messageID* and schedules a retry for the request. In the other function *StopRequest*, this function does the opposite of *StartRequest*. In this function there is a check in the list of queued elements, if the *messageID* in question is in the queue of pending requests it cancels the pending and deletes the element from the queue. Both these functions acquire a lock on the *Requester mutex* which is a synchronized mechanism to prevent concurrent access to the shared resource. The last function in this component is *triggerRequestAndScheduleRetry* is the function used by the *StartRequest* to schedule a re-send of the missing message, it schedules a retry of the request for the missing *messageID* and uses a *timedExecutor* to schedule the retries.

solidifier.go In contrast to the *approvalweight_manager.go*, *solidifier.go* implements the solidification process where in the network. What the solidification process does is that it confirms messages in the Tangle, this means that the message is a permanent and immutable part of the Tangle. There are three functions defined for the solidification process in the system. The *Solidify* function takes in a *messageID* as the argument, the function then traverses the tangle from that *messageID* and examines the *messages* and *metadata* along the way. Then it sets the *Message* with the given *messageID* as *solid* if it already was not. If the *message* is flagged as *solid* a *MessageSolid* event is triggered notifying listeners in the network. This is the main function and uses *messageSolid* and *parentSolid* functions to flag the *message* and *parent* as solid.

storage.go and tangle.go The last three components in this module are *storage.go*, *tangle.go*, and *tipmanager.go*. The *storage.go* is where the *storage* object in the system is defined and created, this object is accompanied by the function for storing *messages* and the functions for looking up stores *messages* and *MessageMetaData*. The *Store* function implements the process of storing a message but first checks if the *message* is already stored, then if this is not the case it stores the *message* and triggers an event to notify the network. The helper functions for this component are *Message*, which looks up a *message* based on its *messageID* in the *messageDB* which is a map with *messageID* and *message* as *key*, *value*. The same setup is also implemented for the look-up of *MessageMetaData*, *weakChildren*, *strongParent*, and *childReferences*.

tangle.go is probably the most important part of the entire simulation tool as this is where the Tangle structure is defined. The Tangle structure is defined and the *Setup* function is implemented with *peer* and *weightDistribution* is put in as arguments. This is done with a single peer because each node runs its own local version of the Tangle and the *weightDistribution* object which represents the consensus weight distribution of the network. In this *Setup* function the other components that were defined in the rest of the module are set up and initialized. These include the *Solidfier*, the *Requester*, the *Booker*, the

OpinionManager, *TipManager*, and the *ApprovalManager*. There is also the *ProcessMessage* function which takes in a *message* object as an argument, stores it in *Storage*, and starts the processing of this *message* by the other components of the Tangle.

```
func (t *Tangle) Setup(peer *network.Peer, weightDistribution *network.ConsensusWeightDistri
t.Peer = peer
t.WeightDistribution = weightDistribution

t.Solidifier.Setup()
t.Requester.Setup()
t.Booker.Setup()
t.OpinionManager.Setup()
t.TipManager.Setup()
t.ApprovalManager.Setup()
}
```

Caption : The Tangle is set up with all the components of the multiverse module. tangle.go

tipmanager.go The last component of this module is the *tipmanager.go* which is responsible for analyzing a given message by looking it up in the *storage* and determining whether it should be added as a strong tip or a weak tip. This process is implemented in the *AnalyzeMessage* function, this function looks up the message based on *messageID* in storage and then first determines the color of the message based on its parents. It uses this information to decide which *tipset(s)* the message should be added to. The function fetches the tip set corresponding to the color using the *TipSet* function and checks the pool size before potentially adding it to the tip set as a new *strong tip* using the *AddStrongTip* method. If any *tips* remain they will be put into other tip sets as a *weak tip*. The method also keeps a counter to keep track of processed messages of each color and triggers an *MessageProcessed* event to notify the network. There are definitions for tip sets and the initialization of new tip sets, and methods for adding strong/weak tips. The method *Tips*() returns a slice of strong and weak tips from the tip set of the current opinion of the *OpinionManager*. From the *tipset* object, the *StrongTips*() method is called on it to retrieve the set of *StrongTips*. The method *StrongTips*() takes in two arguments, the number of parent messages to consider indirect approval calculation and a *TimeSourceAdapter* object to calculate the time-based weight of a message.

The latter part of *tipmanager.go* is where the *TipSelector* is defined and the different types of *TipSelectors* are defined and initialized. There are two types of *TipSelectors*, there is the *URTS* type which implements the uniform random tip selection algorithm. The other type is *RURTS* which implements a restricted uniform random tip selection algorithm where any *messages* is valid up to an age D. There is a method defined for both types of $TipSelectors$, for the $URTS$ the tip selector takes in a random map of tips and max amount as arguments. The *random map* is a data structure used in the IOTA protocol to store and manage a set of data items with a unique identifier. It's similar to hash tables or dictionaries where $key - value$ pairs are stored and retrieved but it adds an element of randomization. The *maxAmount* argument is the number of tips to be retrieved from the *random map.* The tip selection method for the $RURTS$ has an additional requirement for the tips. Its starts by getting a *maxAmount* from by calling the *RandomUniqueEntries* method defined in the $RandomMap$ which returns a $amountLeft$ slice of randomly selected unique tips. Then it checks the issuance time of each tip in the slice and if they are older than a predefined value $DeltaURTS$ they are deleted from the map else it appends the tip to the $tipsToReturn$ slice and decreases the $amountLeft$. This process is repeated until the *maxAmount* tips are appended to $tipsToReturn$ or until the *tips* map is empty.

4.1.3 Network Module

This module is compromised of 5 components and is responsible for the weight distribution for the system and also the Adversary nodes. In this module, the network is defined and implemented along with the $Watts\ Strogats$ network. The creation of peers and connecting them is also done here and the overall configuration structure is defined in this module.

```
func ZIPFDistribution(s float64) WeightGenerator {
return func(nodeCount int, totalWeight float64) (result[]uint64) {
rawTotalWeight := uint64(0)
rawWeights := make([]uint64, nodeCount)
for i := 0; i < nodeCount; i++ {
weight := uint64(math.Pow(float64(i+1), -s) * totalWeight)
rawWeights[i] = weight
rawTotalWeight += weight
}
normalizedTotalWeight := uint64(0)
result = make([]uint64, nodeCount)
for i := 0; i < nodeCount; i++ {
normalizedWeight := uint64((float64(rawWeights[i]) / float64(rawTotalWeight)) * totalWeight
result[i] = normalizedWeight
normalizedTotalWeight += normalizedWeight
}
result[0] += uint64(totalWeight) - normalizedTotalWeight
return
}
}
```

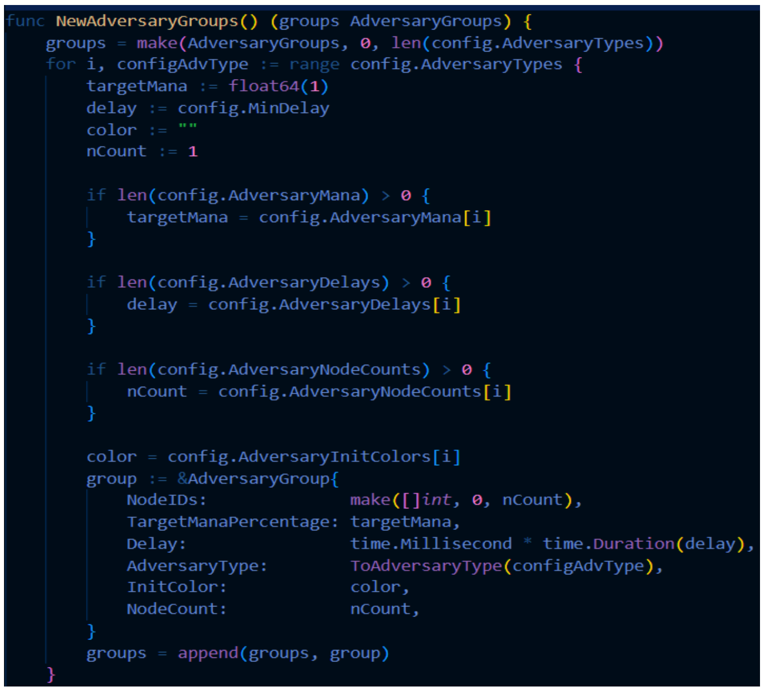
Caption : Implementation of Zipf's law, returns a weight generator based on this law. consensus_weight.go

consensus_weight.go The first component of the network module is the *consensus_weight.go*. This component handles the network consensus weight distribution and defines the *Zipf Distribution* function. As we mentioned earlier in the paper, the node weights in the network are distributed with the use of Zipf empirical law. The *Zipf Distribution* function implements this law (*Figure* 4.4) and the parameter for the distribution is specified in the configuration file. The other part of this component is the *ConsensusWeightDistribution*, which is the object in the system that keeps track of the weights of the nodes in the network. The structure of this object has the parameters, a weight map of all the nodes in the network, the total weight of the network, and the weight of the node with the highest weight. The *NewConsensusWeightDistribution* is a constructor function that creates a new *ConsensusWeightDistribution* object. This object is used to keep track of the node weights in the network as they are updated. It uses the method *SetWeight* to update the weight of a given node. The method receives *PeerID* and *weight* and starts the process of updating the weight by first checking if the *peer* already has an existing weight. This step is done to make sure that the total weight of the distribution is updated accurately and correctly. If the weight already exists then the existing weight is subtracted from the total weight, then a check is done to if the largest weight parameter needs to be changed. Then it ends the method by assigning the *peer* the new *weight* that was given as a parameter. There are also some helper methods to get the weight of a given peer, the total weight of the network, the largest weight, and a method for calculating the largest weight.

groups.go The *groups.go* is the component in the system that defines the structure and methods for groups of nodes in the simulation. This simulation component allows users to define the characteristics of nodes such as consensus weights, number of peers, and delays. The creation of such groups allows for a realistic simulation and can be used to create a heterogeneous network topology where nodes with different characteristics can be connected to each other. The *groups.go* file contains structures and functions for creating groups that can be a mixture of adversary nodes and honest nodes. The file defines both *AdversaryGroup* and *AdversaryGroups*, the latter just being a slice of the type *AdversaryGroup*. *AdversaryGroup* type is defined with parameters such as *NodeIDs* slice of the nodes in the group, *GroupMana* which is a measurement used in *IOTA's* consensus mechanism for measuring a node's influence on the network. The *GroupMana* is the total amount of *mana* that is owned by the group. There is also the *TargetManaPercentage* parameter which is a target measurement for how much *mana* of the total *mana* in the network the group should control. Other parameters such as the *AdversaryType* of the group and the *nodeCount* are also defined in this *AdversaryGroup* type structure. The functions and methods implemented in this file are used for i.e. initializing the groups for each of the types of adversaries that are specified in the configuration. The methods are implemented for purposes like adding nodes to groups and updating weights. The function for initializing the groups is the *NewAdversaryGroups*. This function creates an empty *groups* slice with the length being the number of adversary types from the configuration, then it loops through all of the types and sets the parameters we mentioned earlier. Then it checks the lengths of the *mana*, *delays*, and *nodeCount* to check if they are all above 0, then *mana*, *delays*, and *nodeCount* are set to their appropriate value corresponding to their type (see *Figure* 4.1). The way it's implemented is that the *config.AdversaryTypes*, *config.AdversaryMana*, *config.AdversaryDelays*, and *config.AdversaryNodeCounts* are slices, and for each type of Adversary, there is a corresponding number. So when the *config.AdversaryTypes* is being looped through for each i, $i = 0$ is an *honest* node, and so forth of each of the four types of nodes. Then each type is initialized with its corresponding starting parameters from the configuration using the other slices we mentioned earlier. Then when a group has been initialized, the group is put in the *groups* slice.

The rest of the methods in this component is for calculating weights, updating weights, updating mana, and also connecting the adversaries in each group to their peers. The authors also implement the function *GetAccidentalIssuers* for the accidental issuing of messages. This is done with the intention of modeling a real-life scenario where messages are not intentionally generated but somewhat random events or spam.

network.go The *network.go* component defines the network structures and methods for simulating the network of peers in the Tangle. It's in this component the network is initialized, and the management of the peers is done. The *Network* structure provides methods for selecting random peers and managing the peer connection.
This component is split into three parts, they are namely the definition of the network, its creation, and methods and functions for this type, then there is the definition of the configuration, the creation of a new configuration, and the methods for this type. The last part is the part for the node specification for the behavior and peering strategy which is the *Watts Strogatz* network topology.
The *network.go* structure is defined with the parameters, a slice of the peers in the network, the weight distribution, and all the adversary groups. A network is created with these parameters being filled with information from the configuration, also the function used in *GetAccidentalIssuers* is defined here. This function gets random peers from the network and gives them to *GetAccidentalIssuers*



```go
func NewAdversaryGroups() (groups AdversaryGroups) {
    groups = make(AdversaryGroups, 0, len(config.AdversaryTypes))
    for i, configAdvType := range config.AdversaryTypes {
        targetMana := float64(1)
        delay := config.MinDelay
        color := ""
        nCount := 1

        if len(config.AdversaryMana) > 0 {
            targetMana = config.AdversaryMana[i]
        }

        if len(config.AdversaryDelays) > 0 {
            delay = config.AdversaryDelays[i]
        }

        if len(config.AdversaryNodeCounts) > 0 {
            nCount = config.AdversaryNodeCounts[i]
        }

        color = config.AdversaryInitColors[i]
        group := &AdversaryGroup{
            NodeIDs:              make([]int, 0, nCount),
            TargetManaPercentage: targetMana,
            Delay:                time.Millisecond * time.Duration(delay),
            AdversaryType:        ToAdversaryType(configAdvType),
            InitColor:            color,
            NodeCount:            nCount,
        }
        groups = append(groups, group)
    }
```

Figure 4.1: The different types of Adversaries are initialized with parameters. *groups.go*[6]

as a parameter. The methods used in the *main.go* for starting and shutting down the network are also defined in this system component.
The next part of the network component defines the structure used for holding the various configuration options for the simulation tool. The parameters defined are used for inter-network communication, parameters like i.e. minimum delay, maximum delay, and the peering strategy used for nodes to select peers. Methods for generating random network delays and packet losses are also implemented. The two most important methods in this part of the network component are the *CreatePeers* and *ConnectPeers*. The former creates and configures the peers that will form the network and the latter is responsible for connecting the peers in the network using the specified peering strategy. Both of the methods also log their activity at key points of the process. The *Watts Strogatz* network topology is implemented here as the *PeeringStrategy* type.

Node This *node.go* component is different from the component with the same name from the *Multivers* module. The *node.go* component in the previous module is responsible for the definition, setup, network participation, and network message handling, while *node.go* in this module is an extension of *peer.go*. The node type is defined here with *Setup* and *HandleNetworkMessage* from *peer.go* are initialized and used. The *NodeFactory* type definition along with the helper function $NodeClosure$ is the last part of this component. $NodeFactory$ type is a function type that is a factory function for creating nodes in the simulation tool, it does not take any arguments and returns a node. The helper function $NodeClosure$ is a function that takes in closure function $Closure$ as an argument. It returns a $NodeFactory$ function that wraps the closure and converts its return value to a $Node$. The returned $NodeFactory$ function can then be used to create nodes in the simulation.

Peer.go This last component of the $Network$ module is where a $Peer$ is actually defined. $Peer$ and $Peers$ are used throughout the simulation tool but it's here that it's defined and created. This component is split into two parts, there is the *node* type definition, creation, and the methods for this type, and the other part is the node $Configuration$ definition and its corresponding methods. The parameters of a *peer* are ID, $Neighbors$, *Socket* channel, $Node$, $AdversarySpeedup$, and also some parameters for *shutdown* and *shutdown signal* channel. The methods for this structure are of a functional nature, i.e. *setup*(), $Start()$, $Shutdown()$, *ReceiveNetworkMessage*, and *Run*(). The setup method sets up the peer node with the *Setup*() function from *Node.go* and with the parameters p for the peer and the consensus weight distribution. The $Start()$ starts up the peer, the method is executed only once using the *StartOnce sync.Once* object, then the *run* method is executed as a *goroutine* for concurrency. The $Shutdown()$ method is used to shut down the peer, it ensures that the shutdown signal channel is closed only once with the use of the same *sync.Once* object as the $Start()$ method. *ReceiveNetworkMessage* gets a *message* as an argument, then it send this *message* into the *socket* channel. Both the shutdown signal and the message that is sent into the socket channel are handled in *run*(). In the *run*() method there is an endless loop that does two things, the first is checking if the shutdown signal is sent if that is the case just end the *run*() by returning, the other case is if a network message has been received through the socket channel, in this case, the peer calls the *p.Node.HandleNetworkMessage*() where the argument is the received message.
The peer's network connection is defined in the second part of this module. The parameters set for a peer are defined in the *Connection* structure. Parameters such as the *socket* for sending the messages received, the *network delay*, *packetloss* percentage, *shutdownOnce* signal, and $configuration$ parameter which is other configurations from the $config.go$ that can be specified. The methods implemented for this structure are methods for getting and setting a delay, packet loss and shutting down the peer. There is also the method *send*() used to send a message through the socket connection, but first, it checks if the packet loss probability *c.packetLoss* is greater than a random value generated from $crypto.Randomness.Float64()$. If the random value is less than the packet loss probability then the packet is dropped and if not, then the packet is scheduled to be executed by the use of a *timedExecutor* after a random network delay.

4.1.4 Scripts Module

This module serves as a location for scripts and programs that automate the execution of the simulation and perform additional analysis of the simulation results. It provides a convenient way to alter or extend the simulation tool. It's split into 6 components that cover this description, the components are *config.py*, *constant.py*, *main.py*, *parsing.py*, *plotting.py*, and *utils.py*.

config.py This component serves as the configuration file for defining the various parameters and settings for running simulations. It provides a centralized place for specifying simulation parameters, such as node configuration, network topology, peering strategy, simulation duration, and more. The main purpose of this component is to allow users of the simulation tool to easily configure and customize the simulation by changing the parameters. Some of the key functionalities that can be modified are :

1. Network Topology:

   The network topology can be modification allows users to define the structure of the simulated network. This includes the number of nodes, their connections, and other network topology-related properties.

2. Node Configuration:

   Users can specify the behavior of nodes, initial opinions of nodes, mana values, node delays, packet loss probabilities, and other parameters. These configurations specify how the nodes in the network interact and communicate.

3. Simulation Duration:

   The parameters for setting the duration of the simulation can be modified. Parameters such as the total number of simulation rounds, and the simulation time in seconds. This allows users to control how long the simulation runs and also how the simulated network changes over time.

4. Peering Strategies:

   Users can select and configure different peering strategies that determine how nodes in the network establish connections and form neighbors. The simulation is run with the Watts Strogatz peering strategy, but users can implement other types of peering strategies and use them as a base for their simulation, this provides an opportunity for comparing peering strategies.

5. Adversary types:

   The component allows the users to define different types of adversaries that be present in the network. The user can configure the behavior of the adversaries, the mana distribution, and other parameters.

6. Statistical Analysis and Output:

   The simulation tool allows for the specification of various options for statistical analysis and output generation. This means that the users can define metrics that will be tracked throughout the simulation run, also specifying output files and enabling/disabling certain logging or debugging information. The plotting and visual representation of the acquired result can also be modified to show the results in the way that the user specifies.

constant.py This component of the simulation tool is responsible for defining and keeping track of constant values that are used throughout the simulation. Constant values typically do not change during the run of the simulation and are used to configure different parts of the simulation.
Some of the purposes this component can serve are:

1. Parameter Configuration:

   The *constant.py* component is used to define and store configurable parameters that affect the simulation behavior. Some of the parameters in question are network-like parameters (i.e., delays and packet loss), node parameters (i.e., mana values and weights), and (or) simulation parameters (i.e., max simulation time and number of rounds)

2. Code readability and maintainability:

   Centralizing the constant values in a single separate file makes reading and maintaining the code easier, defining these constant values in a single place makes it easier to locate and modify if necessary.

3. Consistent naming:

   This touches upon the same issue as the previous section. The single placement of all the constant values in the simulation tool allows for a consistent naming convention. With the use of descriptive names and following consistent naming conventions, the code becomes much more readable and understandable.

4. Flexibility and Customization:

   Because the constant values are separated from the main code logic, the simulation becomes easier to customize according to user(s) preferences by simply modifying the values in this file.

Overall the *constant.py* component manages the execution and setting up of the simulation and provides a place for easy configuration of constant values.

main.py The *main.py* component serves as the entry point and coordinates the running of the simulation. It is responsible for setting up the simulation environment, configuring the parameters, executing the simulation loop, and generating output. The main areas this component is responsible for and executes are:

1. Simulation Setup:

   The initialization of the necessary simulation parts and resources required for running the simulation. This covers the creation of the network, defining the nodes, parameter configuration such as network topology, and simulation duration.

2. Simulation Loop Execution:

   Executing the simulation when everything is set up and configured. The execution consists of a loop that iterates over each simulation round where then nodes perform their action and communicate with each other based on the specified algorithms.

3. Interfacing with External tools:

   The *main.go* component also interacts with external tools and libraries to perform analysis on the results or make the implementation easier by the use of existing functions and not having to implement them themselves.

parsing.py The *parsing.py* component in the simulation tool is responsible for the parsing and processing of the input data and configuration. This file provides functions to read and interpret the input, but also extract the useful information and then perform the conversion into a format understandable to the simulation. Some of the work that is done in this component are:

1. Parsing Configuration file:

   Configuration files that specify various parameters, settings, and initial conditions for the simulation are parsed in this component. The format of these configuration files can be in different formats. The parsing file extracts the information that is required from the configuration file and passes it to the parts of the simulation for further processing if needed and setup.

2. Data extraction and Converting:

   The simulation requires input data such as network topology and node initial states, and other relevant data. The parsing component implements the function to parse and extract input data from external sources, reads them, and converts them to a format that can be used by the simulation.

3. Data Validation: The data used by the simulation needs to be validated to make sure that the input data is in the right format, that the data is structured correctly, and that it satisfies any constraints it might have.

4. Error Handling:

   In any simulation there are errors and it's in this component of the system that any error that occurs in the parsing process is also handled. Checking errors, raising exceptions, and logging error messages to provide the user with about any errors that may take place.

parsing.py plays a crucial role in reading, interpreting, and preparing the data for use in the simulation. It ensures that the simulation runs smoothly and received data in the correct format. In the case of any errors or incompatibility of the input data.

plotting.py The main purpose of this component is to generate plots and create visuals based on the simulation output results. By making these plots and visuals it makes it easier to understand and interpret the simulation outcome. To go into detail about the main aspects of this component we can describe some of the work that happens in this component. Some of the aspects are:

1. Data Visualization The primary purpose of this component is to make visuals of the simulation results. The output data from the simulation such as network statics and performance metrics are used to make plots. Some of the types of plots used in this simulation are *Box Plot* and *Violet Plot* (see *Figure* 4.6 for examples).

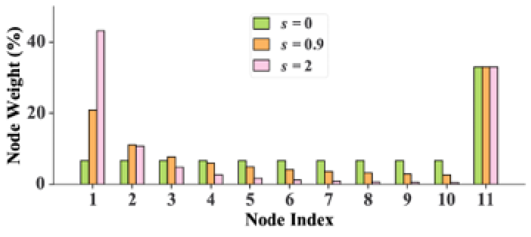


Figure 3: Node weight distribution example with $q = 33\%$, $N = 11$. The node index of the adversary is 11.

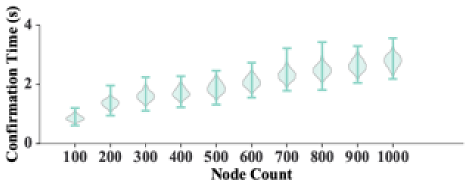


Figure 6: Confirmation time distributions with different N's ($s = 0.9$).

Figure 4.2: Two types of plots are used in TangleSim, Box Plot (Left) and Violet Plot (Right). [6]

2. Output Generation The visuals generated from the simulation results can also be generated in formats such as image files (e.g., JPEG and PNG). These image files can be used further for analysis and included in reports or research papers.

The *plotting.py* component is crucial for visualization and interpreting the simulation results in a graphical and understandable way. It provides a way to analyze the outcome of the simulation and also provides an easy way of communicating any findings with other researchers and other peers.

utils.py The *utils.py* component is mainly a utility file that contains helper methods used for the plotting of figures. Overall the *utils.py* is a toolbox for the plotting part of the simulation and handles the fetches information from the simulation like node weights and network delays and plots the row and column information.

4.1.5 Simulation Module

The *counter.go* component serves as the system clock and provides mechanisms for counting and tracking the different metrics or events that happen in the simulation. There are two types of counters defined and implemented, there is the *AtomicCounters* and *ColorCounters*. The former counter is a counter used for atomic operations to provide safe thread access to do modifications. The latter counter is specifically designed for this simulation tool. It's responsible for tracking multiverse color occurrences. The *counter* is defined here along with helper methods for various tasks.

Atomic Counter The atomic counter is defined as a *AtomicCounters* type, which is implemented with a map of counters. This is a map of $(key, value)$ where counter keys are associated with their corresponding atomic counters. For security purposes so as not to override and access shared resources, there is also defined a mutex *countersMutex* which is *sync.RWMutex* object to protect against concurrent access to the counters map. The method *CreateAtomicCounter* is used for creating new atomic counters. The way its implemented is straightforward, first, the mutex lock is called to prevent concurrent creation, then check if the key is passed as an argument exists in the map, and if not add to the map, and the mutex is unlocked. The rest of the methods implemented for this *AtomicCounters* type are helper methods for getting, adding, or setting a given counter.

```
type ColorCounters struct {
        counts map[string]map[multiverse.Color]int64
        mu     sync.RWMutex
}

func NewColorCounters() *ColorCounters {
     return &ColorCounters{
     counts: make(map[string]map[multiverse.Color]int64),
     }
}
```

Color Counter struct definition and associated implementation. [6]

Color Counter This specifically designed counter is of the type *ColorCounters*. The counter structure is initialized with a nested *counts* map. The nested map associates a counter string *key* to another map that maps the multiverse colors to their assigned counter values (See *Figure* 4.7). For the same reason as the *AtomicCounters* type, a mutex is also defined as a parameter for this structure. The function *NewColorCounters*() is implemented to create this counter, then the method *CreateCounter* for creating an individual counter. This method takes in three parameters, a *counterKey* string, a *colors* slice of multiverse colors, and a slice of *initial values*. This method first acquires the lock to ensure exclusive access, then checks if the length of the slice of initial values is empty. This check is done for flexibility, as some of the counters may not need initial values for all the colors. Then the method checks if the given key does not exist in the map. If that is the case, then a new inner map is created. It then iterates over the *colors* slice and assigns the initial values from the *initValue* slice to each color in the inner map. Then finally the inner map is added to the *counts* map with the provided *counterKey*. As with the *AtomicCounters* type, helper functions are implemented for *ColorCounters* to provide easy access and modification to the color map.

4.1.6 Configuration Module

This module consists of a single *config.go* file and is responsible for setting up the simulation parameters for all the previous main modules. This file categorizes the parameters after the main components of the simulation. It's split into five categories, the simulator settings, the network setup, the weight setup, the tip selection algorithm, and the Adversary setup which is only triggered when simulating double spending in the network.

Simulator Settings This section of the configuration file specifies the target for the simulation, and monitoring the *AW* of nodes while the simulation runs. This is also where a threshold for ending the simulation is put in, this threshold value is calculated using $SimulationStopThreshold * NodesCount$ is larger or equal to confirmed nodes. The two metrics that are being simulated are double spending in the network and consensus time in the network. These two metrics can be set in the *SimulationTarget* field as either *DS* for double spending or *CT* for consensus time. Other parameters are *ConsensusMonitorTick* which tracks the consensus time in milliseconds and *MonitoredWitnessWeightMessageID*, a parameter that tracks the witness weights.

Network Setup In the network setup part of the configuration file there are parameters for the number of nodes, network throughput, and other network-related parameters. To not go into detail we will briefly mention the network parameters and give a short explanation.

1. NodesCount:

   This field specifies the total number of nodes in the simulated network. So it specifies the actual size of the network.

2. TPS:

   Transaction per second is a measurement of the network throughput and this parameter defines the desired network throughput rate.

3. ParentsCount:

   This parameter defines the number of parents a new message should select from the tip pool. Parents refer to previous messages in the network.

4. NeighborCountWS:

   Every node in the network is connected to a peer, this parameter determines the number of neighbors each node is connected to. In this simulation tool, we use the Watts Strogatz network topology model which is a random graph generation model that creates networks.

5. RandomnessWS:

   This parameter is the Watts-Strogatz randomness parameter and determines the degree of randomness in the network topology. The higher value of this parameter the more random the network topology becomes.

6. IMIF:

   $IMIF$ stands for Inter Message Issuance Function is the function that determines the specific interval between message activity in the network. In this simulation, it can be set to either $Poisson$ or $Uniform$ distribution. The difference between the distributions is that with $Poisson$ models a random event occurrence while the $Uniform$ represents a linear constant time delay between events.

7. PacketLoss, MinDelay, MaxDelay:

   The packet loss parameter specifies the probability in the network for packet loss. Min and Max delay parameters define the minimum and maximum network delays that can happen in the network in milliseconds.

8. SlowDownFactor:

   This parameter is used to control the simulation speed. It can be slowed down or sped up, depending on the user's needs. The time dilation makes observation and analysis of the simulation results easier.

Weight Setup This part of the configuration file control the weight distribution among nodes and the confirmation threshold for nodes based on the AW. We will now explain shortly the parameters in this part.

1. Nodes Total Weight:

   This parameter defines the total weight of all nodes in the network. Weight in this simulation represents the voting power of the nodes, so this parameter is the total voting power of the network. In the simulation, this value is set to 100000 by the authors.

2. Zipf Parameter

   This specifies the parameter s for the Zipf distribution used to model the weight distribution among the nodes. The parameter s controls the shape of the distribution. When s is set to 0, all nodes have equal weight, and as s increases, the weight distribution becomes more centralized, with a few nodes having significantly higher weight than others.

3. Confirmation Threshold: This parameter defines the threshold for message confirmation. It measures the $Active\ Walk$, which is a measure of how the messages spread in the network. If the AW exceeds the confirmation threshold it indicates how much confidence is in the confirmation of a message.

4. Confirmation Threshold Absolute

   $ConfirmationThresholdAbsolute$ determines whether the confirmation threshold is counted from zero or from the next peer's weight. If this is set to $true$ which it is in this simulation, then the threshold is counted from zero. This means that the weight collected in AW must exceed the threshold to be counted confirmed. If set to $false$, then it means that the weight collected is counted from the next peer's weight, so the AW in addition to the next peer's weight must exceed the threshold of confirmation.

5. Relevant Validation Weight:

   This parameter represents a threshold for whether a node will issue any messages. If the product of a node's $weight$ with $RelevantValidatorWeight$ is less than or equal to the weight of the largest node it will not issue any messages. Currently simulation, this is set to 0, which means that any nodes can issue messages regardless of weight.

Tip Selection Algorithm Setup The settings in this part of the setup determine the characteristics and behavior of the Tip Selection Algorithm in the simulation. They influence the selection of tips for transaction confirmation and play a role in the overall dynamics and performance of the simulated network.

1. Tip Selection Algorithm:

   Tip selection algorithm (TSA), is the parameter where it's defined which TSA to use during the simulation. In the simulation, there are defined two types of $TSA's$ as we explained earlier in this thesis. The two types are uniform random tip selection ($URTS$) and restricted uniform random tip selection ($RURTS$).

2. Delta URTS:

   It defines the time delta in seconds for the Unreferenced Tip Selection (URTS) algorithm. The $DeltaURTS$ parameter determines the time window within which tips are considered unreferenced and eligible for selection.

3. Weak Tip Ratio

   This sets the ratio of weak tips to strong tips in the tip pool. Weak tips are tips that have a lower chance of being selected as the trunk and branch transactions in the Tangle. This is set to 0.0, which means every tip has an equal chance of being chosen, while a non-zero value indicates a bias towards strong tips.

Adversary Setup The settings in the Adversary setup configure the behavior and parameters of the adversary in the simulation, specifically for double-spending (DS). This setup part has some important parameters that we will explain in short detail.

1. Simulation Mode:

   The simulation model specifies what type of double spending scenario that will be run in the simulation. There are two types of scenarios that can be used in the simulation, the scenarios are *Accidental* and $Adversary$. The *Accidental* scenario simulates accidental double-spending done by nodes in the network. On the other hand, the *Adversary* scenario simulates a malicious attempt at double-spending done by the adversary nodes in the network.

2. Double-Spend Delay:

   This parameter defines the delay in seconds between the issuance of double-spend transactions. So in principle the interval between double-spending attempts done by the nodes.

3. Accidental Mana

   The accidental mana parameter specifies the node that will be used for the accidental double-spend attempt. Selection of the accidental DS node is based on the mana values, the actual selection is done by using three parameters for mana value. The parameters that can be used in this simulation is Min, Max, or *Random*. Min refers to the node with the least amount of mana in the network, Max is the node with the most amount of mana, and *Random* refers to choosing a random node independent of mana value.

4. Adversary Delay:

   This parameter's purpose is to introduce some variability to the actions of the adversary nodes. These delays introduce an artificial delay because, in reality, the actions done by adversary nodes are random and not scheduled. So in theory what these delays do is affect the interactions the adversaries have with the network.

5. Adversary Types

   As we mentioned earlier in the thesis, there are different types of adversaries. This parameter specifies the type(s) of adversaries that will be used in each run of the simulation. The values that can be put into this parameter are 0 (honest node behavior), 1 (shifts opinion), 2 (keeps the same opinion), and 3 (nodes not gossiping anything, even double spends). By using these adversary types in different combinations we can get different views on how the network will operate.

6. Adversary Node Counts and Adversary Initial Colors:

   The $AdversaryNodeCount$ specifies the number of adversary nodes in the network, if this parameter is left as empty the default value will be 1. The $AdversaryInitColors$ defines the initial colors of the adversary groups. The colors available in this simulation are R(red), G(green), and B(blue). The reason for this is that every adversary group needs a specified initial color at the start of the simulation.

7. Adversary Peering All and Adversary Speedup

   The *AdversarySpeedup* is a flag that is set for the adversaries indicating if the adversaries should be able to send messages to all the nodes in the network, effectively bypassing the peering algorithm for the network. If the flag is set to *true* then the adversaries will be able to communicate with any node in the network.
   In the same fashion as the simulation speed-up parameter for the simulation, the *AdversarySpeedup* parameter specifies the speed-up factor for the adversaries. This factor determines how many more messages the adversaries can issue compared to the other nodes.

Chapter 5

Research Design

In this section of the thesis, we introduce our own version of the *TangleSim* simulation tool. We explain in detail what modifications we have done and the purpose behind each of them. As we mentioned in our introduction section, our hypothesis is that it is easier to implement proof-of-work in a Tangle 2.0 simulator than implement Tangle 2.0 in a proof-of-work simulating tool. We make this hypnotizes because we believe that implementing POW on top of the Tangle 2.0 simulator as the consensus mechanism is easier, because of the fact that POW is a consensus mechanism and not a blockchain system like Bitcoin or Tangle. We also wish to show that other types of consensus mechanisms can be implemented in the Tangle 2.0 simulating tool. We will approach the modifications we implemented one file at a time, and conclude our work at the end.

5.0.1 models.go

The *Message* object defined in *Modesl.go* that we mentioned briefly earlier in the thesis is where we started modifying the code base. For *POW* we need the longest chain to add blocks onto. We created a new parameter for the *Message* structure. We created the *height* parameter for the messages so we could keep track of the longest chain of messages. This parameter will come into use later in the process of updating message heights.

5.0.2 messagefactory.go

As we mentioned in the design background (ref section 4.1.2), the message factory component handles the creation of messages and passing in the message fields. The modification we did here is fourfold. We mainly changed the *CreateMessage* method in this file. The first thing that happens in this method is retrieving the strong and weak tips from the tip manager. The tip manager has a function called *Tips*() which selects tips from the tip set according to the nodes' current opinion.

The *Tips*() returns the strong and weak tips, in the message factory we only want the strong tips and look away from the weak tips because the strong tips provide the tips with the most references and are most likely to be confirmed. The next step of our process is to find the parent of the strong tip we can update the height for the new message that will be created and published. We find the parent of the strong tip by using the *GetTip*() method we defined in *TipManager.go*. We will explain the implementation of this method in the next section also. But this method returns an integer *height* and a boolean which we use to confirm successful retrieval of the parent height. Then we increment the *height* field of the message object with parent height + 1.

5.0.3 tipmanager.go

The tip manager component is an important part of the process of integrating proof-of-work into the simulation tool. Because in *POW* the blocks published by the miners are added to the main chain, there can only be one block at a time. So we have to make it so that the tip provided by the tip manager must be the one with the longest chain, but in our case, we use a strong tip. The way we filter out the strongest tip is by finding the tip from the strong tips set that has the highest height. We will explain the different parts of our process in detail in the next steps.

POW Tip Selector For *POW* we created a custom tip selector algorithm in the same way the authors created the other tip selectors. First, we create a new type of tip selector algorithm type *POW* under the *TSA* interface *TipSelector*. The way our tip selector algorithm for *POW* works is that we first define a maximum height variable to store the *maxHeight* of the tips in the random map *randommap.RandomMap*, a message variable for storing the message we retrieve, and we also define a *tipsToReturn* interface slice.

As we mentioned in the design background section, the tips are stored in a *randommap.RandomMap* object. This map has a method implemented for different types of retrieval, we used the *ForEach*() method that iterates over each element in the map and calls the *consumer* function. By using this method we can iterate over the elements in the *RandomMap* and apply custom logic to each element without having to manually handle the synchronization of concurrent access to the map. When we iterate over the elements we check if the message height is larger than the maximum height which we defined as 0 in the beginning. And as we iterate through the map we continuously update the *maxHeight* and *msg* variables. When the loop is finished, we den we update the *tipsToReturn* slice with the message height and then return *tipsToReturn*.

This implementation is our attempt at integrating *POW* into the tip selection algorithm. To form it to *POW* we only choose one tip, which has to be the one with the highest height. This conforms to the *POW* system where only one block is published to the chain. We also make the *POW* tip selection algorithm a part of the *swithc* − *case* in the *NewTipManager*() function, this is where the tip manager knows which of the *TSA's* is specified in the configuration to use for the simulation run.

```
 func (POW) TipSelect(tips *randommap.RandomMap, maxAmount int) []interface{} {
     maxHeight := uint64(0)
     var msg *Message
     var tipsToReturn []interface{}
     tips.ForEach(func(key, value interface{}) {
     if value.(*Message).height > int(maxHeight) {
     tipsToReturn = make([]interface{}, 1)
     maxHeight = uint64(value.(*Message).height)
     msg = value.(*Message)
     }
     tipsToReturn[0] = msg
     })return tipsToReturn}
```

Proof-of-work tip selector. tipmanager.go[3]

GetTip(): The *GetTip*() method is the method we defined in the *TipManager* structure. The purpose of this method is to find the message height of a given message. The first thing that we do is to get the tip set of the current opinion of the calling node. Then use that tip set to access the *StrongTips*, then we use the method *Get*() from the *randommap.RandomMap* object. This *Get*() method returns a value to which the given key is mapped. After retrieving the message from the *RandomMap* we first check if the message is not empty, then we cast the *msg* as a slice of the type *Message* with the length 1. We create this slice because both *messageID* and the *msg* returned from the *Get*() method are both of the type *interface*. We then return the *height* of the message by indexing in the slice and a boolean *true*. This method is the same *GetTip*() method that we mentioned in the *messagefactory.go* to return the height of the *strongParent* (See Below).

```
func (t *TipManager) GetTip(messageID interface{}) (height int, true bool) {
     tipSet := t.TipSet(t.tangle.OpinionManager.Opinion())
     msg, _ := tipSet.strongTips.Get(messageID)
     if msg == nil {
     // return 0 and false if msg interface is empty
     println("msg is nil")
     return 0, false
     } else {
     // else cast interface to Message type, index and return height
     msg := make([]Message, 1)
     return msg[0].height, true
     }
}
```

The method for retrieving parent tip from the *randomMap*, *tipmanager.go* [3]

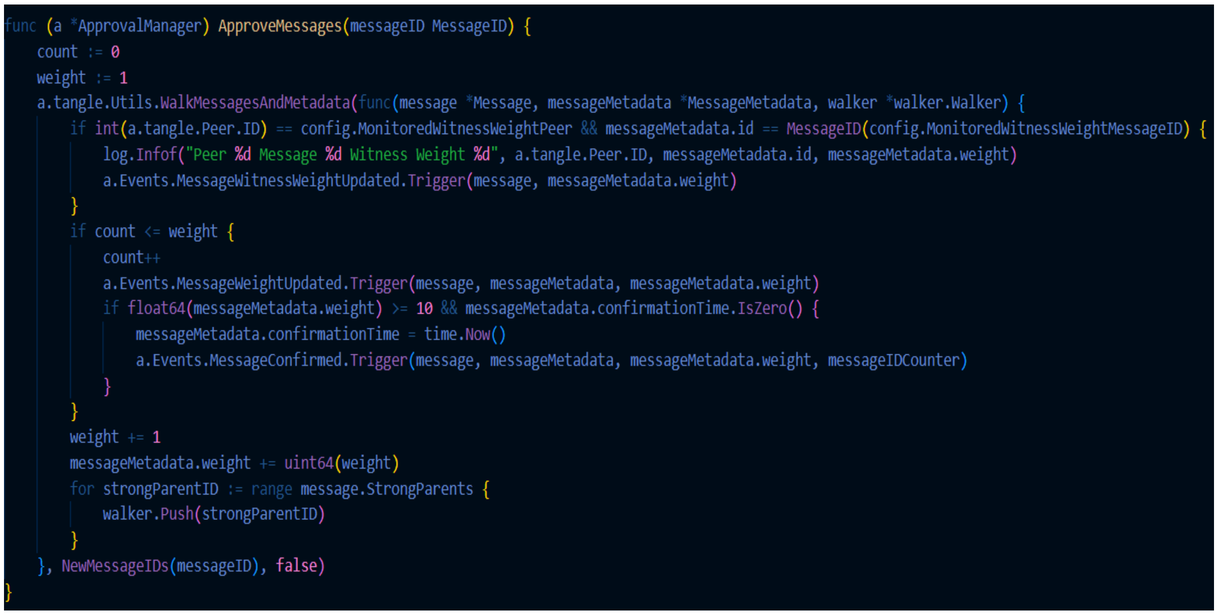
5.0.4 approvalweight_manager.go

In *POW* we mentioned earlier that the difficulty of publishing blocks is the main component of the consensus protocol. This keeps the network going forward at a relatively same rate of publishing. In our implementation of the protocol in this simulation tool, we used some other unorthodox methods to simulate the same effect of the *POW* ever-changing difficulty.

The way we simulate the same effect as *POW* difficulty is by making use of the random nature of the *Poisson* distribution. As we mentioned earlier, the *Poisson* distribution models random event occurrences, so we use this property to make the message publishing a random event similar to the randomness in block publishing in Blockchain where the consensus protocol used is *POW*. In addition to the random event occurrence of the *poisson*, we have made modifications to the *approvalweight_manager.go*. This component which is a part of the *Multiverse* module is responsible for determining the level of power a node has in validating a transaction. Its task is to keep track of the approval weights of nodes, it keeps track of the number of incoming and outgoing transactions of a node, and the cumulative weight of the node. We made a modification of the weights of the messages, where all messages start off with a weight of 0. Using a counter that is always increasing we compare the message weight with the counter weight. We do this because we want the message weights to be ever-increasing as the simulation runs and all messages have the same starting point in terms of weight. After this counter check, we then have another check where we ascertain if the message weight is larger than a set *confirmation* value along with checking if the message has been given a weight before.

5.0.5 config.go

In the configuration file, we did not do many modifications, but the ones we did have an important role in the process. The first thing we did is to change the *TSA* parameter to reflect the change to *POW* instead of the other tip selection algorithms. The second modification we made is to reduce the *parentCount* parameter from the original number of 8 to instead 1. This makes sure that each message can only have one parent, so a type of Greedy-Heaviest-Observed Sub-tree (GHOST) is established instead of the *DAG* structure which it was originally.



```go
func (a *ApprovalManager) ApproveMessages(messageID MessageID) {
    count := 0
    weight := 1
    a.tangle.Utils.WalkMessagesAndMetadata(func(message *Message, messageMetadata *MessageMetadata, walker *walker.Walker) {
        if int(a.tangle.Peer.ID) == config.MonitoredWitnessWeightPeer && messageMetadata.id == MessageID(config.MonitoredWitnessWeightMessageID) {
            log.Infof("Peer %d Message %d Witness Weight %d", a.tangle.Peer.ID, messageMetadata.id, messageMetadata.weight)
            a.Events.MessageWitnessWeightUpdated.Trigger(message, messageMetadata.weight)
        }
        if count <= weight {
            count++
            a.Events.MessageWeightUpdated.Trigger(message, messageMetadata, messageMetadata.weight)
            if float64(messageMetadata.weight) >= 10 && messageMetadata.confirmationTime.IsZero() {
                messageMetadata.confirmationTime = time.Now()
                a.Events.MessageConfirmed.Trigger(message, messageMetadata, messageMetadata.weight, messageIDCounter)
            }
        }
        weight += 1
        messageMetadata.weight += uint64(weight)
        for strongParentID := range message.StrongParents {
            walker.Push(strongParentID)
        }
    }, NewMessageIDs(messageID), false)
}
```

Figure 5.1: This method traverses(Walks) down the message chain to determine approval weight. [3]

5.0.6 main.go

We made some modifications to the *main.go* file so we could gain some other types of results from the output. What we did was make an additional header to the header fields for the output, we added a *parentID* field in the *main.go* and also created a simple loop in the information output method, where we retrieved the *parentID* for the message that was being outputted. The reason we did this was to gain an idea of the heritage of the message blocks, we will explain more in the results and discussion chapters.

Chapter 6

Results and Discussion

In this chapter, we present and give a brief explanation of the results of our research. We then discuss the results, the entirety of the thesis, and describe the knowledge we have gained and if there is anything we would wish to make different or improve upon.

6.1 Results

6.1.1 Tangle 2.0 with POW results

Confirmation Time

In this section, we present the confirmation time results obtained from our experiments conducted using the extended TangleSim simulation tool incorporating Proof-of-Work (PoW) in the Tangle protocol. The objective of this analysis is to evaluate the impact of PoW on the confirmation time of transactions in the Tangle network.

Confirmation time is a crucial metric that measures the time it takes for a transaction to be deemed finalized and included in a confirmed state by the network. A shorter confirmation time is desirable as it indicates faster transaction settlement and improved user experience. By integrating PoW into the TangleSim simulation tool, we can observe how the computational effort required for transaction validation affects the confirmation time in the Tangle network.

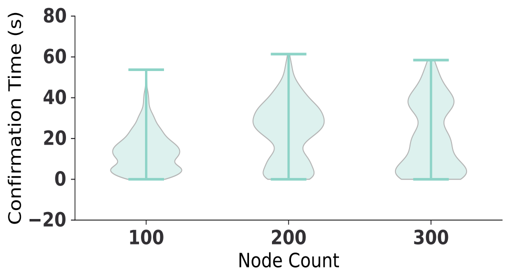


Figure 6.1: Confirmation time for different numbers of Nodes. With Zipf = 0.8, Approval Threshold = 50. Simulation run with python3 main.py -rs -pf -v N -vv 100 200 300 -df 1 -rt 5 -st CT [3]

In the figure above we can see how long it takes for 100,200, and 300 nodes. The x-axis shows the number of nodes in the network, the y-axis shows the confirmation time in seconds. The simulation was run with 5 iterations with different thresholds for approval of messages. The threshold used in our simulation runs varied from 10-50. The Python command to run the simulation specifies what types of variation we wish to simulate, in this case, we want to have a variety of the number of nodes hence we use the N parameter which is node. After that, we specify the different number of nodes and how many repetitions we want the simulation to run for each of the node counts. At the very end, we specify that the simulation target is confirmation time hence the CT in the end.

Forks in the Network

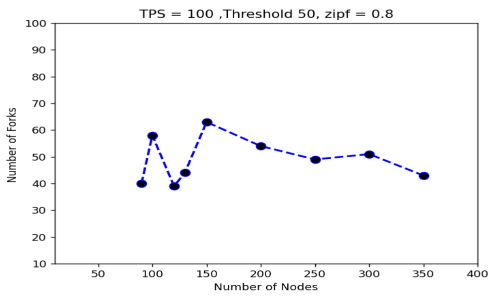


Figure 6.2: This figure shows the number of forks in the network in different network sizes.

This figure shows the rate of change in the number of forks in the network. Forks in our case where there is more than one branch going from one block to another. So it means that there is not a main chain from Genesis, but many branches, this can be associated with *GHOST* consensus protocol where there is not 1 main chain but can be different branches one can build blocks on top of. This figure is made using the results from the same simulation run as in the previous figure. The figure itself does not give any conclusive answer to the correlation between forks and the number of nodes, and we have observed that it has a certain variability to it.

The simulation that produced the results of the plot was run with a confirmation threshold of 50 and zipf parameter of 0.8 which means that it's not a completely decentralized network. What we can see from this figure is that there is a steady decline in the number of forks in the network as the network size grows. A temporary decline was seen between 100 and 150 network size but it was just a slight dip in forks that can be due to different reasons. Figure 5.3 shows a different perspective, which shows the correlation between

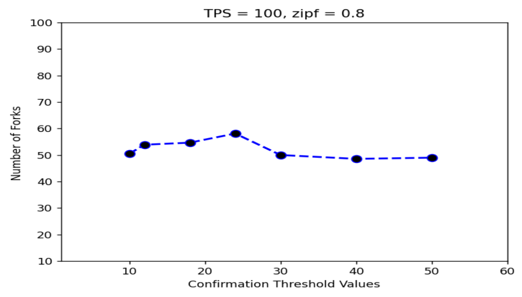


Figure 6.3: This figure shows the number of forks in the network for the different confirmation threshold values.

the number of forks in the network for different confirmation threshold values. We found the mean number of forks throughout all of our simulation runs and plotted them against the different values for the threshold. It shows that the number of forks in the network is a steady line and it increases slightly around threshold value = 25, then evens out for the remainder of the values. This shows that there is no definitive answer to either a turn point where the number increases downwards or upwards.

6.1.2 TangleSim results

In this section, we will present our simulation results from running the original Tangle simulator[6]

Confirmation Time

The figure above illustrates the relationship between confirmation time and the number of nodes in the TangleSim simulation with POW. The figure provides insights into how the confirmation time varies as the network size increases.

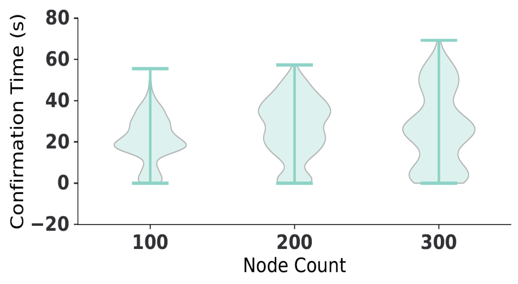


Figure 6.4: Confirmation time for different numbers of Nodes. With Zipf = 0.8. Simulation run with python3 main.py -rs -pf -v N -vv 100 200 300 -df 1 -rt 5 -st CT [6]

Figure X illustrates the results obtained from our experiments, where we varied the number of nodes in the network and measured the corresponding confirmation time. The y-axis represents the confirmation time, which indicates the time taken for a transaction to be confirmed in the Tangle. The x-axis represents the number of nodes in the network, indicating the scale of the simulated Tangle ecosystem.

What we can see from this figure is mainly the correlation between the number of confirmed messages and the confirmation time. On the left of the figure we can see that for a network size of 100, we can see that there is a slow start and a majority of the messages start getting confirmed between $20s$ and $40s$ into the simulation run. In the middle of the figure, we can see that similar to the previous network size, when the network size is 200 the confirmation of messages starts off slow, but in this case, the confirmations continue in an increasing fashion from between $20s$ into the simulation and all the way to the end of the simulation. In the last case of the network size of 300, we can see that it differs from the previous two cases. It instead shows a continuous and steady confirmation rate of the messages.

6.2 Discussion

In this section, we will discuss our results in a general manner while also comparing both the results from the two simulations and also comparing Tangle and proof-of-work.

6.2.1 Comparison of results

The results we presented represented only one of the simulations we performed. We will include the rest of our results in our GitHub repository. But what we have observed is that the results have varied and do not give a conclusive view that can be stated. Our results about the number of forks in our version of the Tangle with POW, we analyzed the results from the simulation, by finding the number of parents blocks and adding them up we found the total number of nodes. In a traditional blockchain, every block has only one parent, in our case, it's more similar to *GHOST* where a block can be parent to a number of different blocks creating forks. So after calculating the number of forks for an entire simulation run, we plotted the results. What we have noticed in our results is that there are more forks and the number of these forks does not reduce that much when the confirmation threshold is low. This was to be expected, but we also noticed that the number of forks is higher when the network size is smaller and reduces slightly when the network size is larger. We performed similar calculations for the original Tangle simulator using the same commands and observed a much higher number of "forks". We cannot call these forks because as we explained the Tangle creates a *DAG* structure so it's a part of the design and we confirmed it works as we expected. The forks in the network can be removed by adjusting the network throughput, this will lead to a longer simulation but also a longer confirmation time for the messages in the network.

When looking at the results from both simulators, we can see that there is a varying degree of confirmation when talking about the number of messages getting confirmed. We see that there is either a majority of messages get confirmed within the first $20s$ and $30s$, but also the latter part of the confirmation time i.e. from $20s$ to $80s$. We also see in some of the results that there are also instances where there is a steady rate of confirmed messages throughout the simulation. In the case of the network size being 100 and 200, there is no conclusive answer, but in almost all of the cases of network size of 300, there is a steady confirmation rate in all of our simulation runs.

6.2.2 Comparison between Tangle 2.0 and Tangle 2.0 with POW

This research has shown that incorporating *POW* into a *Tangle* setting provides better handling of confirmation and approval compared to the original Tangle 2.0. The results have shown that there is not a major reduction in important metrics such as confirmation time. Any reduction in confirmation can be credited to the more mathematical burden of proof-of-work and this exchange of confirmation time to a more secure network that *POW* provides is to be expected. An unexpected discovery was that incorporating *POW* produced a greedy-heaviest sub-tree type of network topology with many branches in the network instead of the originally intended traditional blockchain network topology. Comparing *Tangle* and *POW* strictly then there are aspects of the two consensus protocols such as the consensus algorithm, the network topology, and network participation/participants.

The Tangle topology is a *DAG* structure where the blocks can be published by every participating node. The blocks have two parents that they reference upon issuing. In contrast, proof-of-work in traditional Blockchain systems has one main-chain structure where from genesis there is one block as a parent creating a chain of blocks. The consensus mechanism used in the Tangle 2.0 is called *Tip Selection Algorithm.* The tip selection algorithm in Tangle 2.0 is based on a concept known as the *Markov Chain Monte Carlo* (*MCMC*) *Random Walk.* The MCMC random walk algorithm selects tips based on a probabilistic approach, considering various factors to determine the most suitable tips for approval. In contrast, POW participants, known as miners, compete to solve a computational puzzle or mathematical problem. The solution to this problem requires a significant amount of computational power and time. If we look at the process of block issuing we can see that POW has a barrier of entry that is hardware needed for computational power in contrast to Tangle where there is no need for particular hardware. This leads us to the participation of nodes in the two networks, in Tangle there is a direct entry, by that we mean that one does not need anything in particular before joining the network while in POW you have to take into consideration that if one wants to publish a block there will be a roadblock one has to overcome before joining the network.

6.3 Limitations and Future Improvements

This research was limited by factors such as time but also not having a deep enough understanding of the original TangleSim simulation tool. The tool was both very complicated in terms of unraveling the core functionality but also tailored to Tangle 2.0 by people of far more knowledge and expertise than we. If we could change some of the ways we proceeded with this research is that we would split the components such as the network and the tip selection and do more in-depth changes to make the POW protocol more ingrained in the simulation tool and not shallow as we currently have.

Chapter 7

Conclusion

In conclusion, we have simulated Tangle 2.0 with the TangleSim simulating tool and analyzed it before making a simulator of our own. We have simulated how POW works in a Tangle setting by changing the simulating tool in different key places to make it a fit for POW. This has led us to make observations and given us insight into how a POW consensus protocol operates in a Tangle environment. The results of our research have made it possible for us to make the conclusion that POW does not give a definitive improvement over the original Tangle in a Tangle environment. Inserting the POW protocol into Tangle changes the original DAG structure and creates a structure similar to $GHOST$. So we conclude that this research has not yielded much in terms of value.

Bibliography

[1] The Tangle Serguei Popov, October 1, 2017. Version1.3 http://cryptoverze.s3.us-east-2.amazonaws.com/wp-content/uploads/2018/11/10012054/IOTA-MIOTA-Whitepaper.pdf

[2] Silvano, W. F., Marcelino, R. (2020). Iota Tangle: A cryptocurrency to communicate Internet-of-Things data. Future Generation Computer Systems, 112, 307-319. https://doi.org/10.1016/j.future.2020.05.047

[3] https://github.com/AdnanAhmed12/Multiverse.git

[4] On the Security and Performance of Proof of Work Blockchains Arthur Gervais ETH Zurich,Switzerland arthur.gervais@inf.ethz.ch Ghassan O.Karame NEC Laboratories,Europe ghassan@karame.org Karl Wüst ETH Zurich, Switzerland kwuest@student.ethz.ch Vasileio Glykantzis ETH Zurich, Switzerland glykantv@student.ethz.ch Hubert Ritzdorf ETH Zurich, Switzerland hubert.ritzdorf@inf.ethz.ch Srdjan Capkun ETH Zurich, Switzerland srdjan.capkun@inf.ethz.ch

[5] Tangle2.0 Leaderless Nakamoto Consensus on the Heaviest DAG Sebastian Müller†, Andreas Penzkofer†, Nikita Polyanskii†, Jonas Theis†, William Sanders†, Hans Moog† Aix Marseille Université, CNRS,Centrale Marseille,I2M-UMR7373,13453 Marseille, France †IOTA Foundation, Berlin,Germany `https://arxiv.org/abs/2205.02177`

[6] Robustness of the Tangle 2.0 Consensus Bing-Yang Lin,Daria Dziubałtowska,Piotr Macek,Andreas Penzkofer,Sebastia nMüller† IOTA Foundation,Berlin,Germany †AixMarseille Université,CNRS,Centrale Marseille,I 2M-UMR7373,13453 Marseille,France `https://arxiv.org/abs/2208.08254` `https://github.com/iotaledger/TangleSim.git`

[7] D.J.Watts and S.H.Strogatz, "Collective dynamics of 'small world' networks, "nature, no.6684, pp.440–442,1998.

[8] D.M.W.Powers,"Applications and explanations of Zipf's law," in New Methods in Language Processing and Computational Natural Language Learning, 1998.[Online]. Available:https: //aclanthology.org/W98-1218

[9] ChainSim: A P2P Blockchain Simulation Framework Bozhi Wang1,2(B), Shiping Chen, Lina Yao,and Qin Wang University of New South Wales, Sydney,NSW 2062, Australia bozhi.wang@student.unsw.edu.au 2 CSIRO Data61, Sydney,NSW2110, Australia 3 Swinburne University of Technology, Melbourne, VIC3122, Australia

[10] DAGsim: Simulation of DAG-based distributed ledger protocols Manuel Zander Department of Computing Imperial College London London,UK manuel.zander17@ic.ac.uk, Tom Waite Department of Computing Imperial College London London, UK thomas.waite14@ic.ac.uk, Dominik Harz Department of Computing Imperial College London London,UK d.harz@ic.ac.uk

[11] BlockSim:A Simulation Framework for Blockchain Systems Maher Alharby,Aadva nMoorsel School of Computing, Newcastle University,UK m.w.r.alharby2, aad.vanmoorsel@ncl.ac.uk

[12] M.Di Pierro, "What Is the Blockchain?," in Computing in Science Engineering, vol. 19, no. 5, pp. 92-95, 2017, doi: 10.1109/MCSE.2017.3421554.

[13] Bitcoin: A Peer-to-Peer Electronic Cash System, Satoshi Nakamoto, satoshin@gmx.com, www.bitcoin.org

[14] Sompolinsky, Y., Zohar, A. (2015). Secure High-Rate Transaction Processing in Bitcoin. In: Böhme, R., Okamoto, T. (eds) Financial Cryptography and Data Security. FC 2015. Lecture Notes in Computer Science(), vol 8975. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-662-47854-7$_3 2 AlexdeVries, Bitcoin'senergyconsumptionisunderestimated : Amarketdynamicsapproach, EnergyResearchSocialScience, Volume70, 2020, 101721, ISSN2214-$ 6296, $https : //doi.org/10.1016/j.erss.2020.101721.$

List of Figures