IOTA Implementation Specification Document

Python-Based Tangle Protocol Development

Misbah Khan

University of New South Wales
School of System and Computing

October 20, 2023

1 Introduction

The purpose of this document is to outline the implementation and testing of the IOTA protocol, based on the IOTA white paper. The IOTA protocol is a distributed ledger technology that operates on a structure called the ”Tangle”, a type of directed acyclic graph (DAG). Unlike traditional blockchain technologies, IOTA’s Tangle architecture allows for free and almost instantaneous transactions, making it ideal for Internet-of-Things (IoT) applications.

1.1 IOTA Overview

IOTA is a distributed ledger technology (DLT) designed for the Internet of Things (IoT). It offers secure data transfer and fee-less real-time micro-transactions for this ecosystem, providing a foundation for the effective automation of machines and other IoT devices. Unique to IOTA is its use of a directed acyclic graph (DAG) structure for its ledger, known as the Tangle, as opposed to the traditional blockchain used in other DLTs.

IOTA operates in a non-linear data structure, where each new transaction is attached to two previous transactions. This creates a web of transactions, known as the Tangle, rather than a linear chain found in typical blockchain architectures. The scalability and speed of this network increase as more participants join the network and conduct transactions, making it especially suitable for micro-transactions in IoT devices.

The white paper details the steps a node takes to issue a transaction in the IOTA network:

- ”The node chooses two other transactions to approve according to an algorithm. In general, these two transactions may coincide.”
- ”The node checks if the two transactions are not conflicting, and does not approve conflicting transactions.”
- ”For a node to issue a valid transaction, the node must solve a cryptographic puzzle similar to those in the Bitcoin blockchain. This is achieved by finding a nonce such that the hash of that nonce concatenated with some data from the approved transaction has a particular form. In the case of the Bitcoin protocol, the hash must have at least a predefined number of leading zeros.” - IOTA White Paper [Pop18]

1.2 Key Metrics in the IOTA Tangle

The IOTA Tangle defines several key metrics for transactions, including the transaction weight, the cumulative weight, height, depth, and score.

Transaction Weight: This is proportional to the amount of work a node has invested in issuing a transaction. It serves as a deterrent against spamming and other attacks by ensuring that no entity can generate an abundance of transactions with ”acceptable” weights quickly. Weights can only assume certain values, according to the rules defined in the IOTA implementation.

Cumulative Weight: The cumulative weight of a transaction is the sum of its own weight plus the weights of all transactions that approve it, either directly or indirectly. It is a critical metric that represents the overall ”importance” of a transaction in the network.

Tips: Tips are unapproved transactions in the Tangle. When a new transaction arrives and approves existing tips, it becomes the only tip, and the cumulative weight of all other transactions increases by the new transaction’s own weight.

Height: The height of a transaction site in the Tangle is defined as the length of the longest path to the genesis.

Depth: The depth of a transaction site in the Tangle is the length of the longest reverse-oriented path to some tip.

Score: The score of a transaction is defined as the sum of the weights of all transactions approved by this transaction, plus its own weight. It is a valuable metric in certain contexts.

"In order to understand the arguments presented in this paper, one may safely assume that all transactions have an own weight equal to 1. From now on, we stick to this assumption. Under this assumption, the cumulative weight of transaction X becomes 1 plus the number of transactions that directly or indirectly approve X, and the score becomes 1 plus the number of transactions that are directly or indirectly approved by X. Let us note that, among those defined in this section, the cumulative weight is (by far!) the most important metric, although height, depth, and score will briefly enter some discussions as well." - IOTA White Paper [Pop18]

1.3 MCMC RandomWalk Algorithm

The IOTA Tangle uses a Markov Chain Monte Carlo (MCMC) algorithm to select the two tips to reference. The process starts by defining H_x as the current cumulative weight of a site. Given our assumption that all own weights equal to 1, the cumulative weight of a tip is always 1, while the cumulative weight of other sites is at least 2.

The algorithm places particles, also known as random walkers, on sites of the Tangle and lets them walk towards the tips randomly. The tips chosen by these walks become the candidates for approval. The algorithm can be described as follows:

1. Consider all sites on the interval [W, 2W], where W is reasonably large.
2. Independently place N particles on sites in that interval.
3. Let these particles perform independent discrete-time random walks "towards the tips". A transition from x to y is possible if and only if y approves x.
4. The two random walkers that reach the tip set first will sit on the two tips that will be approved. However, it may be wise to discard those random walkers that reached the tips too fast as they may have ended on one of the "lazy tips".
5. The transition probabilities of the walkers are defined in the following way: if y approves x ($y \to x$), then the transition probability P_{xy} is proportional to $\exp(\alpha(H_y))/\sum_{z:z\to x}\exp(\alpha(H_z))$ where $\alpha > 0$ is a parameter to be chosen.

Here, if $\alpha = 0$, then we have Uniform Random Walk (URW), and if $\alpha > 0$, then we have Biased Random Walk (BRW).

1.4 Implementation of IOTA Protocol

Our implementation of the IOTA protocol is constructed using Python, selected for its ease-of-use, clear syntax, and comprehensive libraries that align well with the specific needs of distributed ledger technology. Leveraging Python's asyncio and Threading libraries, the asynchronous network communication between nodes in the Tangle has been effectively handled to ensure smooth and concurrent data transfer.

Python's hashlib and RSA libraries, known for their robust cryptographic hash functions, have been used to create unique and secure transaction identifiers. To manage and operate on the Directed Acyclic Graph (DAG) structure of the Tangle, we've employed powerful data handling libraries like pandas and NumPy, simplifying the complex task of data manipulation and processing.

To visually analyze and debug the Tangle structure, we have used NetworkX, a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. In conjunction, matplotlib and graphviz have been employed to provide a flexible way to visualize the Tangle data and observe the state of our network at any given point, aiding in the debugging and optimization process.

A critical aspect of our implementation is adherence to the IOTA protocol as defined by the IOTA white paper. It ensures that our software can interact seamlessly with other nodes in the Tangle network, thereby maintaining network-wide consistency. We've implemented key features such as transaction validation, acumulative weight calculations, random walk, Coordinator, Genesis milestone, and node synchronization as described in the white paper, ensuring our Python-based IOTA node is fully compliant with the established IOTA protocol (as per the white paper).

1.5 Testing the Implementation

The testing of this implementation will be conducted in accordance with the table provided in the subsequent section of this document. Each key feature of the IOTA protocol, as mentioned in the IOTA white paper, has been tabulated along with their expected behavior, current status, and test results. This approach ensures that all components of the system are exhaustively tested and validated for their functionality, resulting in a robust and reliable software system.

2 System Architecture

The IOTA Tangle protocol implementation leverages a modular and scalable architecture, represented in the class diagram, sequence diagram, and network model. This architecture fosters robustness, adaptability, and facilitates network operations.

In this system, the Network serves as the foundation, comprising several nodes including a special one called the coordinator. Each node possesses its own list of transactions and the ability to communicate and broadcast these transactions to other nodes in the network, ensuring continuous and efficient propagation. The coordinator node plays a crucial role, generating the genesis block and regular milestones. Network latency or delay is factored into the design, making the propagation of transactions across the network more realistic.

The class diagram depicts the various classes, including Node, Coordinator, Network, Transaction, IOTA_DAG, and RandomWalker, each embodying distinct attributes and methods to serve specific functionalities within the system. The encapsulation of these functionalities within respective classes makes the system independently manageable and efficient.

The sequence diagram complements this by providing a visual representation of transaction propagation in the network, starting from transaction creation at an individual node, through validation and propagation to peers and peers of peers.

In summary, the system architecture presents a comprehensive overview of the working model of the implemented IOTA Tangle protocol, facilitating an understanding of its functionalities and processes, and paving the way for its potential enhancements and debugging.

2.1 Network

In the current implementation of the IOTA Tangle protocol, the network is composed of multiple nodes. The quantity of these nodes can be adjusted in the code, providing flexibility to adapt to different scenarios and use case. These nodes are strongly interconnected, forming a comprehensive, resilient network. .

A special type of node, known as the coordinator, plays a important role within the network. The coordinator is responsible for the generation of the genesis block or transaction — the very first transaction in the network. Furthermore, the coordinator periodically produces milestones, which serve as checkpoints in the transaction history. The frequency of these milestone generations is modifiable in the code, providing the opportunity to set the rate of checkpoint creation as needed.

Despite the fact that only one coordinator is present in the current simulations, it maintains connections with every node within the network. Although this might not be the most practical approach for larger, more complex networks, it is an efficient setup for initial simulations.

The nodes within the network operate in a highly cooperative manner. When a node generates a transaction, it broadcasts this message to its peers. In turn, these peers propagate the transaction to their peers, creating a ripple effect that quickly permeates through the entire network.

Network latency is a crucial aspect of the network's dynamics. This term refers to the delay that occurs when transaction is transmitted from one node to another. For example, consider two nodes, Node A and Peer 1 Node B. The delay between these nodes might be 2.49 seconds, implying that any transaction generated by Node A will reach Node B 2.49 seconds later. Similarly, when Node B initiates a transaction, it will arrive at Node A after the same duration. This symmetric delay holds true for all pairs of nodes within the network.

The coordinator, despite its unique role, also experience these delay principles. It experiences different delay times when communicating with different nodes, just like any other node in the network. The network latency is represented in a delay matrix within the Network class. Figure 1 represent the Network consisting on Nodes and Coordinator with associated delays.

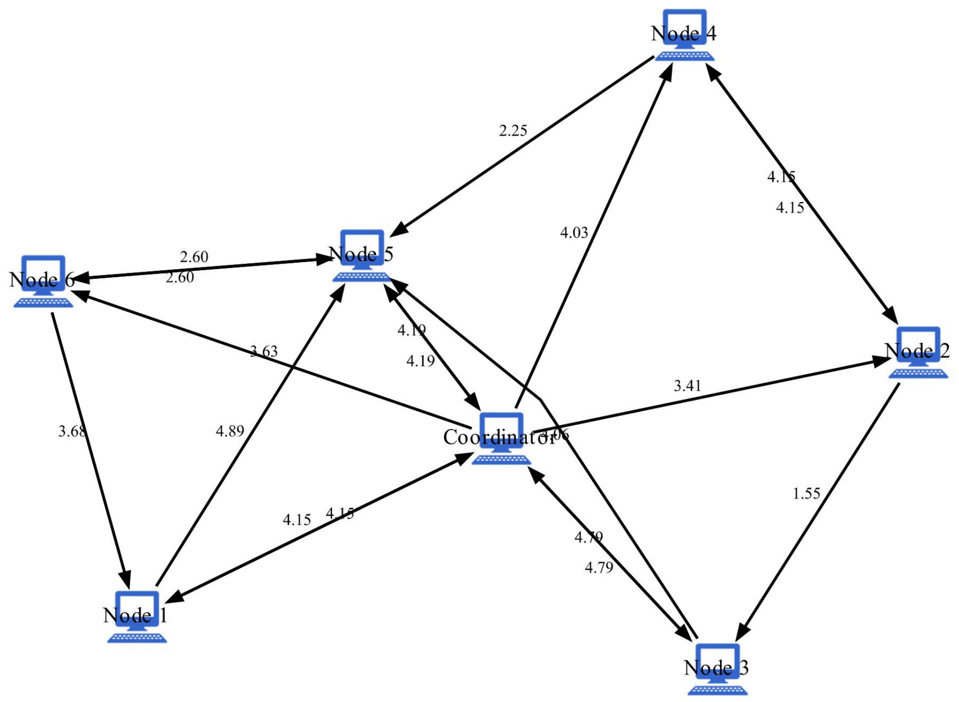

Figure 1: Network diagram of the implemented IOTA Tangle protocol

2.2 Class Diagram

In the following subsection, a class diagram, Figure 2, representing the structure of the implemented IOTA Tangle protocol is provided. This diagram showcases the interrelationships between the different classes in the system, their attributes, and methods, providing a clear overview of the program's structure and functionality. The classes represented include the 'Node'', 'Coordinator'', 'Network'', 'Transaction(s)'', 'IOTA_DAG'', and 'RandomWalker''.

2.3 Class Diagram Description

The implementation of the IOTA Tangle protocol is divided into multiple classes, each with specific attributes and methods. Below is a brief overview of these classes:

- **Node:** This class represents an individual node in the network. It includes attributes such as Name (of type string), and methods including 'receive_genesis_milestone()'', 'create_and_sign_transaction()'', 'broadcast_transaction()'', 'receive_transaction()'', 'is_double_spent()'', 'receive_milestone()'', and 'validate_milestone_signature()''.

- **Coordinator:** This class is a special node that has the added responsibility of issuing milestones. It includes attributes such as Name (of type string), and methods including 'coordinator_view()'', 'get_recent_transactions()'', 'generate_milestone()'', 'sign_milestone()'', and 'broadcast_milestone()''.

- **Network:** This class represents the entire network of nodes. It contains methods such as 'create_peer()'', 'generate_delay_matrix()'', 'configure_nodes_with_coordinator()'', and 'draw_network()''.

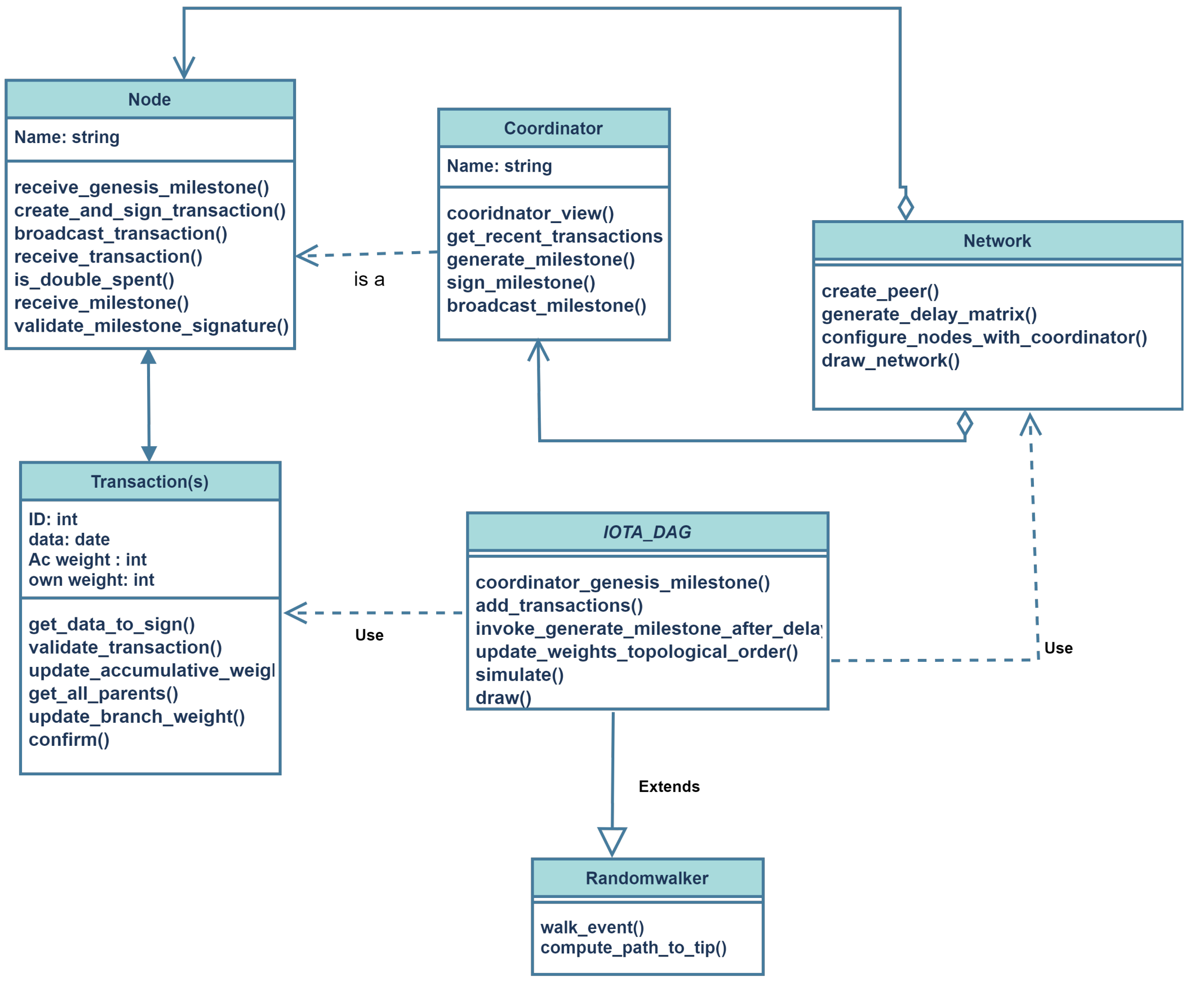

Figure 2: Class diagram of the implemented IOTA Tangle protocol

- **Transaction(s):** This class represents individual transactions in the network. It includes attributes such as ID (of type integer), Data (of type date), Ac weight and Own weight (both of type integer), and methods including ‘get_data_to_sign()‘, ‘validate_transaction()‘, ‘update_accumulative_weight()‘, ‘get_all_parents()‘, ‘update_branch_weight()‘, and ‘confirm()‘.
- **IOTA_DAG:** This class represents the Directed Acyclic Graph of transactions in the IOTA protocol. It contains methods such as ‘coordinator_genesis_milestone()‘, ‘add_transactions()‘, ‘invoke_generate_milestone_after_delay()‘, ‘update_weights_topological_order()‘, ‘simulate()‘, and ‘draw()‘.
- **RandomWalker:** This class implements the random walker algorithm for tip selection. It contains methods like ‘walk_event()‘ and ‘compute_path_to_tip()‘.

The ‘Network‘ class incorporates instances of the ‘Node‘ and ‘Coordinator‘ classes, where ‘Coordinator‘ is a special type of ‘Node‘. The ‘IOTA_DAG‘ class uses instances of the ‘Transaction‘ class and extends the ‘RandomWalker‘ class. Each ‘Node‘ has associated transactions, which are either in its own transaction list or received transactions.

2.4 Sequence Diagram

Figure 3 sequence diagram of the Tangle protocol as implemented in the simulation. It provides a high-level overview of the process flow during the creation and propagation of transactions in the network.

2.5 Sequence Descriptions

In the *IOTA Tangle* protocol, transactions are generated and propagated throughout a network of nodes, using a decentralized peer-to-peer model. The process begins with a single node and flows through multiple stages, including transaction creation, tip selection, transaction validation, and ultimately, transaction propagation. This process is designed to ensure the integrity, consistency, and robustness of the Tangle, which is a *Directed Acyclic Graph* (DAG) that represents all the confirmed transactions in the IOTA network. The sequence diagram above provides a high-level overview of the transaction creation and propagation process, with key components representing the primary stages.

- **Node:** This represents an individual node within the network. Each node is responsible for generating transactions, which are then propagated throughout the network. The node initiates a transaction by executing the *createTransaction()* function. This function encapsulates the logic needed to generate a new transaction.
- **RandomWalker:** This component symbolizes the algorithm used to choose two tips from the Tangle for a new transaction to approve. After the node creates a transaction, it calls the *getTips()* function, which is handled by the RandomWalker. This function uses a random walk algorithm to select two tips that will be referenced by the new transaction.
- **DirectPeers:** These are the nodes within the network that directly receive transactions from the originating node. Upon receipt of a new transaction, each DirectPeer runs the *checkConflictAndDuplicate()* function, which confirms whether or not the transaction conflicts with existing transactions and whether it is a duplicate of an existing transaction. If the transaction passes this validation process (i.e., it is not conflicting and not a duplicate), the DirectPeer accepts it.
- **PeersOfPeers:** These are the secondary nodes within the network that receive transactions from the DirectPeers. Just like the DirectPeers, upon receipt of a transaction, each PeerOfPeer executes the *checkConflictAndDuplicate()* function to validate the transaction. If the transaction passes the validation (i.e., it is not conflicting and not a duplicate), the PeerOfPeer accepts it.

Finally, each PeerOfPeer propagates the validated transaction back to the originating Node. This propagation helps ensure the robustness and redundancy of the network, as well as the eventual consistency of the Tangle across all nodes.

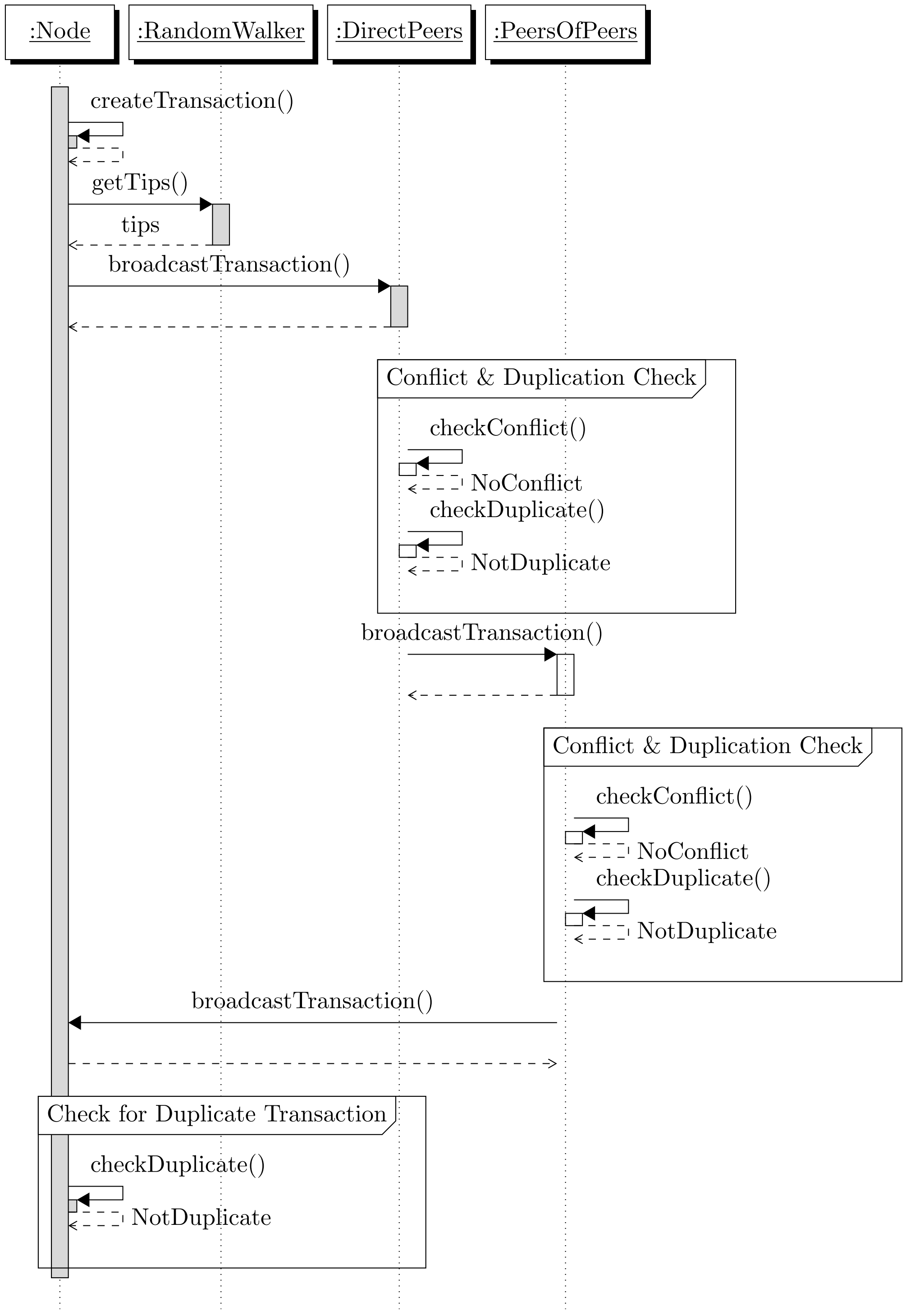

Figure 3: Sequence diagram of implemented IOTA Tangle protocol

3 Implementation Details

3.1 Overview

The implementation of the IOTA Tangle protocol is an intricate process that involves the careful coordination of multiple components and the effective use of various tools, technologies, and strategies. This section (Table 1) aims to provide an overview of the critical elements involved in the implementation process, including the programming language, development environment, frameworks and libraries, data structures, algorithms, system architecture, and testing methods. In addition to detailing what was used in the implementation, we will also discuss why these specific elements were chosen and how they contribute to the overall functionality and efficiency of the protocol. By exploring these components, we aim to provide a holistic view of the implementation process, highlighting the decisions made during the development and the considerations that guided these choices.

Table 1: Implementation details of the IOTA Tangle protocol

Aspect	**Details**
Programming Language	***Python 3.10*** was used for its supportive libraries, simplicity, and dynamic natures ideal for developing interactive simulations like our IOTA Tangle protocol.
Development Environment	***PyCharm***: Chosen for its comprehensive features such as intelligent code assistance, linting, debugging, and version control integration, aiding in efficient development and debugging processes.
Frameworks and Libraries	The implementation leverages several Python libraries: • **NumPy**: Used for efficient numerical computations. • **NetworkX**: Employed for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. • **Matplotlib**: Utilized for graphical visualization of the network and transaction propagation. • **SimPy**: A framework used to build and simulate discrete event systems. • **hashlib**:This library is used to implement a cryptographic hash function for the proof-of-work process, aiding in maintaining the integrity and security of the transaction data. • **Graphviz**: Assisted in network visualization. • **threading**: Used for creating threads and performing parallel computations. • **datetime**: Used for manipulating dates and times. • **rsa**: Utilized for the generation and verification of RSA signatures for transactions. • **ThreadPoolExecutor**: From the concurrent.futures module, used for creating a pool of worker threads. • **asyncio**: Used for writing single-threaded concurrent code.

Continued on next page

Table 1 – continued from previous page

Aspect	**Details**
Data Structures and Algorithms	Key data structures and algorithms in the implementation include: • **Directed Acyclic Graph (DAG)**: Used to represent the Tangle, with nodes representing transactions and edges representing approvals. • **Random Walk Algorithm**: Used for tip selection, guiding the choice of which transactions a new transaction should approve. • **Validation Algorithms**: Various algorithms are used for validating transactions, checking for conflicts, and resolving double-spend issues. • **Proof of Work**: Algorithm where a nonce value is iterative incremented and hashed until a hash with a predetermined difficulty level (number of leading zeros) is found. • **Simulation Algorithms**: Various algorithms are used to simulate aspects of the network, such as network delays.
System Architecture	The system architecture comprises several classes and components that interact in a structured manner to simulate the functioning of the IOTA Tangle protocol. It includes the Node, Coordinator, Network, Transaction(s), IOTA_DAG, and RandomWalker classes, which play different roles in the system. The interaction and flow of data among these components are detailed in the Sequence Diagram, Class Diagram, and Network Diagram (Refer to Section 2), 1, 2, 3

3.2 Detailed Class Descriptions

3.3 Transactions

- ***init***: This is the constructor method of the 'Transaction' class. It initializes various attributes, including 'txid' (transaction ID), 'parent_txids' (IDs of parent transactions), 'children' (child transactions), and others. One notable attribute is 'is_confirmed', which is a flag indicating whether the transaction has been confirmed or not.

```
def __init__(self, txid, parent_txids, node, data=None):
    ...
    self.is_confirmed = False  # Transaction confirmation check
    ...
```

- ***confirm***: This method is used to confirm a transaction. It sets the 'is_confirmed' attribute to 'True'.

```
def confirm(self):
    self.is_confirmed = True
```

- ***get_data_to_sign***: This method computes the data that needs to be signed for this transaction. This is composed of the 'txid', 'timestamp', 'parent_txids', and a SHA-256 hash of the transaction data.

```
def get_data_to_sign(self):
        data_hash = sha256(self.data.encode()).hexdigest()
        return f'{self.txid}{self.timestamp}{self.parent_txids}{data_hash}'.encode()
```

- ***validate_transaction***: This method checks if a transaction is valid. It first verifies if the transaction is not a double spend, second it verify the proof of work, then verifies the signature of the transaction data.

```
def validate_transaction(self):
    ...
    if self.node.is_double_spent(self):
    ...
    if not self.verify_proof_of_work(DIFFICULTY):
    ...
    try:
        rsa.verify(
            self.get_data_to_sign(),
            self.signature,
            self.node.public_key
        )
        return True
    except rsa.VerificationError:
        return False
```

- ***update_accumulative_weight***: This method recursively updates the accumulative weight of a transaction and all its children.

```
def update_accumulative_weight(self):
    ...
    for child in self.children:
        child.update_accumulative_weight()
    ...
```

- ***verify_proof_of_work***: This function verifies the proofofwork by ensuring that the hashed transaction data and nonce starts with the required number of leading zeros, specified by the difficulty level. This process provides a means of verifying that the required computational effort was expended in the generation of the transaction.

```
def verify_proof_of_work(self, DIFFICULTY):
    message = self.get_data_to_sign() + str(self.nonce).encode()
    hash_result = hashlib.sha256(message).hexdigest()
    prefix = '0' * DIFFICULTY
    return hash_result.startswith(prefix)
```

3.3.1 Node

The Node class is a fundamental component of the simulation, representing a participant in the network. This participant can generate transactions, broadcast them to its peers, and validate incoming transactions. Below are the primary attributes and methods contained within this class:

- *name*: The name of the node.
- *network*: The network that the node is part of.
- *delay_range*: The range of time delays for broadcasting transactions.
- *public_key, private_key*: The public and private keys used for signing transactions.
- *transaction_list*: A list of all transactions that the node has created.

- *unconfirmed_transactions, confirmed_transactions*: Lists of unconfirmed and confirmed transactions respectively.
- *peers*: A list of the node's peers in the network.
- *milestones*: A list of the milestones the node has received.
- *genesis_milestone*: The initial transaction or "genesis" milestone that the node receives.
- *nodes_received_transactions*: A list of transactions that the node has received from others.
- *transaction_queue*: A queue for incoming transactions.
- *tips*: A list of transactions that do not have any child transactions yet.
- *broadcasted_transactions*: A list of transactions that the node has broadcasted to its peers.
- *is_coordinator*: A boolean indicating whether this node is a coordinator node.
- *coordinator_public_key*: The public key of the coordinator node.
- *transaction_lock, transaction_list_lock, tips_lock*: Locks to ensure thread-safe operations on transactions and tips.
- *create_and_sign_transaction*: To generate a transaction node solve a cryptographic puzzle, which is Proof of work algorithm. The PoW algorithm involves finding a nonce (a one-time numeric value) that, when hashed with the transaction data, produces a hash that starts with a certain number of zeroes, specified by the difficulty. The nonce starts at zero and is incremented by one until a suitable nonce is found.

  ```
  prefix = '0' * DIFFICULTY
  nonce = 0
  while True:
      message = transaction.get_data_to_sign() + str(nonce).encode()
      hash_result = hashlib.sha256(message).hexdigest()
      if hash_result.startswith(prefix):
          break
      nonce += 1
  ```

  After that node generates a new transaction, signs it with the node's private key using the rsa.sign() method, and adds it to the node's transaction list and tips. The signing of the transaction is achieved using the RSA library as shown below:

  ```
  transaction.signature = rsa.sign(
      transaction.get_data_to_sign(),
      self.private_key,
      'SHA-256'
  )
  ```

  After creating and signing the transaction, the method updates the node's tips. If a new transaction refers to a previous tip as a parent, it removes the parent from the tip list:

  ```
  with self.tips_lock:
      self.tips.append(transaction)
  for parent in transaction.parent_txids:
      if parent in self.tips:
          with self.tips_lock:
              self.tips.remove(parent)
  ```

- *broadcast_transaction*: Broadcasts a transaction to all of its peers with a certain time delay.Below is the most critical part of this function:

  ```
  if transaction.txid not in self.transaction_list:
  thread = threading.Thread(target=peer.receive_transaction,
  args=(transaction, self, delay))
  ```

  This approach uses multithreading, allowing the node to concurrently notify all its peers about the new transaction. In other words, instead of waiting for each peer to receive the transaction before moving on to the next, the node can broadcast the transaction to multiple peers at the same time.

- *receive_transaction*: This method allows a node to receive a transaction from another node. After receiving the transaction, the node performs several steps to process it:

  ```
  if transaction in self.nodes_received_transactions:
  ```

  This condition checks if the node has already received this transaction. If it has, the method ends and no further action is taken.

  ```
  if transaction in self.transaction_list:
  ```

  This condition checks if the transaction is already in the node's own list of transactions. If it is, the method ends and no further action is taken.

  ```
  if not transaction.validate_transaction():
  ```

  This line calls the validate_transaction() method on the transaction object. If the transaction does not pass validation, the method ends and no further action is taken.

  ```
  self.nodes_received_transactions.append(transaction)
  ```

  If the transaction passes all the above checks, it is added to the list of transactions that the node has received from other nodes.

  ```
  with self.tips_lock:
  if not transaction.children and transaction not in self.tips:
  self.tips.append(transaction)
  ```

  If the transaction does not have any child transactions and it is not already in the node's tips, it is added to the tips.

  ```
  for parent in transaction.parent_transactions:
  if parent in self.tips:
  self.tips.remove(parent)
  ```

  If any of the transaction's parent transactions are in the node's tips, they are removed from the tips. This is because a tip cannot be a parent of another transaction.

  ```
  self.broadcast_transaction(transaction, sender)
  ```

  Finally, the node broadcasts the received transaction to all of its peers, allowing them to receive and process the transaction in the same manner by recursively calling broadcast_transaction method.

- *is_double_spent*: This method checks whether a transaction is a double spend. In other words, it checks if the transaction attempts to use funds that have already been spent in another transaction.

  ```
  for tx in self.transaction_list:
  if tx != transaction and tx.node ==
  transaction.node and tx.txid ==
  transaction.txid:
  ```

  The above loop iterates through all transactions in the node's transaction list. For each transaction in the list, it checks three conditions:

  1. The transaction in the list is not the same as the transaction being checked. This ensures a transaction cannot double spend itself.
  2. The transaction in the list and the transaction being checked were created by the same node.
  3. The transaction ID of the transaction in the list is the same as the transaction ID of the transaction being checked. If two transactions with the same ID were created by the same node, one of them is a double spend.

  After that, it checks the conflict between and parents and parents of parents. This code performs a depth-first search on the parents of a given transaction, and their parents, and so on, to look for the conflicting transaction.

  ```
  queue = [parent for parent in transaction.parent_txids]
  while queue:
      current = queue.pop(0)
      if current == transaction.txid:
          return True
      for tx in self.transaction_list:
          if tx.txid == current:
              queue.extend(tx.parent_txids)
              break
  ```

- *receive_milestone*: Receives a milestone, validates it, and if valid, adds it to the node's list of milestones and confirms the transactions contained within it.

- *validate_and_confirm*: Validates and confirms a transaction, marking it as confirmed.

- *validate_milestone_signature*: This method is responsible for validating the signature of a milestone.

  ```
  rsa.verify(
  milestone.get_data_to_sign(),
  milestone.signature,
  self.coordinator_public_key
  )
  ```

  This code snippet invokes the rsa.verify() method from the RSA library. The get_data_to_sign() method of the milestone object is used to obtain the data that was originally signed. The signature attribute of the milestone is the signature to be verified, and self.coordinator_public_key is the public key of the coordinator, which is used to perform the verification. If the verification process does not raise an exception, it implies that the signature is valid.

- *receive_genesis_milestone*: Receives the genesis milestone, validates it, and if valid, confirms the transaction within it and adds it to the list of received transactions.

3.4 Coordinator

- ***init***: This is the constructor method of the 'Coordinator' class. It initializes the attributes and inherits from the 'Node' class, with additional parameters specific to the coordinator node, such as 'milestones_interval', 'milestones', 'coordinator_public_key', and 'coordinator_private_key'.

```
def __init__(self, name, network, milestones_interval, is_coordinator=False):
    ...
    self.coordinator_public_key, self.coordinator_private_key = rsa.newkeys(512)
    ...
```

- ***coordinator_view***: This method is used to update the list of transactions seen by the coordinator.

```
def cooridnator_view(self, transaction):
    self.transaction_list.append(transaction)
```

- ***generate_milestone***: This method generates a milestone. It includes recent transactions and signs the milestone.

```
def generate_milestone(self):
    ...
    milestone = {
        'index': milestone_index,
        'timestamp':  datetime.now().strftime('%H:%M:%S.%f'),
        'signature': self.sign_milestone(milestone_index),
        'validated_transactions': []
    }
    ...
    self.milestones.append(milestone)
    self.broadcast_milestone(milestone)
    ...
```

- ***sign_milestone***: This method is used to sign the milestone using the coordinator's private key.

```
def sign_milestone(self, milestone_index):
    ...
    signature = rsa.sign(milestone_data, self.coordinator_private_key, 'SHA-256')
    return signature
```

- ***broadcast_milestone***: This method is used to send the milestone to all peers in the network.

```
def broadcast_milestone(self, milestone):
    ...
    for peer in self.peers:
        ...
        if isinstance(milestone, Transaction):
            if peer.receive_genesis_milestone(milestone):
                valid_count += 1
    ...
```

- ***get_recent_transactions***: This method is used to get recent transactions, i.e., transactions created within the recent interval.

```
def get_recent_transactions(self):
    ...
    if (current_time - tx_time).total_seconds() <= self.milestones_interval:
        recent_transactions.append(tx)
    ...
```

3.5 Network

- ***init***: This initializes the network with the specified number of nodes, a new instance of a DAG (Directed Acyclic Graph), and the coordinator node. All nodes are connected and a delay matrix is generated.

```
def __init__(self, num_nodes):
    ...
    self.dag = DAG(poisson_rate=0.5, milestones_interval=10)
    ...
    self.coordinator = Coordinator("Coordinator", self,
    milestones_interval=150, is_coordinator=True)
    ...
```

- ***get_random_node***: This method randomly selects a node from the network.

```
def get_random_node(self):
    ...
    nodes_without_coordinator = [node for node in self.nodes
    if not isinstance(node, Coordinator)]
    node = random.choice(nodes_without_coordinator)
    return node
```

- ***create_peers***: This method connects each node in the network to a subset of the other nodes.

```
def create_peers(self):
    ...
    for node in self.nodes:
        potential_peers = [peer for peer in self.nodes if peer !=
        node and peer not in node.peers]
        node.peers = random.sample(potential_peers,
        min(num_peers, len(potential_peers)))
    ...
```

- ***generate_delay_matrix***: This method generates a unique delay range for each pair of nodes.

```
def generate_delay_matrix(self):
    ...
    for i in range(len(self.nodes)):
        for j in range(i + 1, len(self.nodes)):
            ...
            delay = random.uniform(min_delay, max_delay)
            self.delay_matrix[(current_node, other_node)] = delay
            self.delay_matrix[(other_node, current_node)] = delay
    ...
```

- ***configure_nodes_with_coordinator***: This method sets the public key of the coordinator for all nodes.

```
def configure_nodes_with_coordinator(self, coordinator_public_key):
    ...
    for node in self.nodes:
        node.coordinator_public_key = coordinator_public_key
    ...
```

- ***draw_network***: This method visualizes the network with nodes and edges using Graphviz.

3.6 RandomWalker

- ***init***: This is the constructor method of the ‘Random-Walker‘ class. It initializes the attributes ‘N‘, which is number of the site in interval and ‘alpha‘ representing the degree of biased random walk .

  ```
  def __init__(self, N, alpha):
  ```

- ***walk_event***: This method is used to start the walk process.

  ```
  def walk_event(self, dag, prev_tips):
  ```

  This function, ‘walk_event‘, is a member function of the RandomWalker class. This function carries out the process of a random walk through the DAG to select the tips. It accepts two arguments:

  - ‘dag‘: an object of the DAG class which represents the Directed Acyclic Graph, - ‘prev_tips‘: a list of previously known tips of the DAG.

  ```
  current_time = datetime.now()
  w = 30
  wd = w * 2
  ```

  First, the function captures the current time with ‘datetime.now()‘. Then it defines an interval ‘W‘ to ‘2W‘ seconds with ‘W=30‘ seconds. Therefore, ‘wd = 60‘ seconds, indicating the interval ends 60 seconds ago from the current time. The interval is used to determine which transactions will be considered for the random walk.

  ```
  if current_time - dag.creation_time < timedelta(seconds=wd):
      wd = int((current_time - dag.creation_time).total_seconds())
  ```

  This block checks if the age of the DAG is less than the interval. If the DAG is younger than the specified interval (‘wd‘ seconds), the function adjusts ‘wd‘ to match the DAG’s age. This is done to ensure that the random walk does not include transactions that were not yet present.

  ```
  start_time = current_time - timedelta(seconds=wd)
  end_time = current_time - timedelta(seconds=w)
  ```

  Next, the function defines the start and end times for the interval. This interval will be used to filter the transactions that are considered for the random walk.

  ```
  interval_transactions = [tx for tx in
  dag.transactions.values()
  if start_time <= tx.timestamp <= end_time]
  ```

  This line filters out transactions that occurred within the defined interval. These transactions are saved in ‘interval_transactions‘ and are the candidates for the start of the random walk.

  ```
  if len(interval_transactions) < self.N:
      start_transactions = interval_transactions
  else:
      start_transactions = random.sample(interval_transactions, self.N)
  ```

  The function then checks if the number of interval transactions is less than ‘N‘. If true, all the transactions are selected as starting points for the random walk. If not, the function randomly selects ‘N‘ transactions from the interval transactions.

```
        for i, start_transaction in enumerate(start_transactions):
              t = threading.Thread(target=self.walk_from,
              args=(start_transaction, reached_tips, tip_paths),
                                    name=f"Thread-{i}")
```

Furthermore, to run the walkers concurrently on selected sites, N threads starts walk simultaneously and call walk_from method and pass start_transaction, reached_tips, tip_paths and name (Thread name) arguments. walk_from method returns the reached_tips by the walkers.

```
while len(reached_tips) < 2:
    ...
```

In case fewer than two unique tips are reached, the function enters a loop to select additional tips randomly until there are at least two unique tips reached.

```
reached_tips.sort(key=lambda tup: tup[1])
        selected_tips = [tup[0] for tup in reached_tips[:2]]
```

Finally, the function sorts the reached tips based on their transaction IDs and selects the first two transactions that reached the tip set. These selected tips are then returned as the result of the random walk.

- ***walk_from***: This is helping method of walk_event, called for every thread.

```
while current_transaction.children
    ...
```

This function loops over each selected starting transaction, letting each perform independent random walks towards the tips. This is where the 'random walk' part of the algorithm happens. The function uses either an unbiased or biased random walk, depending on the 'alpha' value. An unbiased random walk has all children with equal probability, whereas a biased random walk's transition probability is proportional to the exponentiated accumulative weight of each child.

  - The above while loop continues as long as the current transaction has children, i.e., it's not a tip in the DAG. This loop performs a random walk towards a tip in the DAG.

```
if self.alpha== 0:
    probabilities = [1 / len(current_transaction.children)]
    * len(current_transaction.children)
```

  - If the parameter 'alpha' is 0, the random walk is unbiased. This means that all children of the current transaction have an equal probability to be chosen as the next transaction in the walk. Therefore, the probabilities of the children are set as 1/number of children for all children.

```
else:
    probabilities = [np.exp(self.alpha *
    child.accumulative_weight)
    for child in current_transaction.children]
    probabilities /= np.sum(probabilities)
```

  - If 'alpha' is not 0, then the random walk is biased. In this scenario, the transition probability for each child is proportional to $e^{\text{alpha}\times\text{accumulative\_weight of the child}}$. The accumulative weight represents the total weight the transaction and its descendants have accumulated in the DAG. This calculation means that children with more weight have a higher probability of being chosen in the walk. After calculating the raw probabilities, they are normalized by dividing them by the sum of the probabilities. This ensures that the probabilities form a valid probability distribution.

The calculated probabilities will then be used in the next step to decide which child transaction to transition to next in the random walk. The higher the probability for a child, the more likely it is to be chosen as the next transaction. This approach allows the random walk to be biased towards certain paths in the DAG, depending on the weights of the transactions.

- ***compute_path_to_tip_event***: This method computes the path to the tip for each transaction.

  ```
  def compute_path_to_tip_event(self, dag, tip, interval_transactions):
      ...
      current_transaction = tip
      W_batch_txids = set(tx.txid for tx in interval_transactions)
      ...
  ```

3.7 IOTA_DAG

The IOTA_DAG class is responsible for creating and managing the Directed Acyclic Graph (DAG) used in the IOTA protocol. Below are the descriptions for each method:

- **__init__**: This is the constructor for the class, initializing various attributes, including the network, transaction details, and a lock for multi-threaded scenarios. In the initialization method of the 'IOTA_DAG' class, a new thread is started to invoke and continuously generate milestones after given intervals. The current time is recorded to keep track of the last milestone time:

  ```
  self.last_milestone_time = datetime.now()
  ```

  Then, a new thread is initialized with the target function set to 'self.invoke_generate_milestone_after_delay', which will be responsible for generating milestones after the specified intervals:

  ```
  self.milestone_thread = threading.Thread(target=
  self.invoke_generate_milestone_after_delay)
  ```

  Finally, the newly created thread is started, initiating the process of milestone generation:

  ```
  self.milestone_thread.start()
  ```

- **coordinator_genesis_milestone**: This method generates the first transaction (the genesis transaction) of the graph and broadcasts it to all the nodes in the network.

  - **Generate Random Data**: The method first generates a random data string of size 0.5 MB.

    ```
    data = os.urandom(500000)  # generates 500,000 bytes = 0.5 MB
    data = data.decode('latin1')  # decode bytes to string using 'latin1' encoding
    ```

  - **Create Genesis Transaction**: Then, it creates the genesis transaction with no parent transactions and with the data just created.

    ```
    genesis_milestone = Transaction("0", [], None, data)
    ```

  - **Sign Transaction**: The transaction's data is signed using the Coordinator's private key.

    ```
    genesis_milestone.signature = rsa.sign(
        genesis_milestone.get_data_to_sign(),
        self.coordinator.private_key,
        'SHA-256'
    )
    ```

  - **Broadcast Milestone**: Next, the Coordinator's method is used to broadcast the milestone to all peers.

    ```
    self.coordinator.broadcast_milestone(genesis_milestone)
    ```

  - **Add Genesis Milestone**: Finally, the genesis milestone is added to the transactions of DAG_Event.

    ```
    self.transactions[genesis_milestone.txid] = genesis_milestone
    ```

- **process_node**: This function simulates a delay in processing a node after receiving a transaction from the coordinator.

- **add_transactions**: This function adds new transactions to the graph from a specific node. It also performs a random walk to select parent transactions, validates them, creates a new transaction, updates various attributes of the transactions, and broadcasts them. This function also invokes the creation of milestones when the milestone interval reaches a certain threshold.

  - **Instantiate Random Walker**: The method first instantiates the random walker to perform a random walk over the tips of the Directed Acyclic Graph (DAG).

    ```
    random_walker = RandomWalker(N=2, alpha=0.01)
    random_walk_tips = random_walker.walk_event(self, nodes_tips)
    ```

  - **Validate Signatures**: Then, it validates the signatures of parent transactions to ensure they are not counterfeit. The invalid signatures are skipped.

    ```
    for parent in tips:
        if parent.txid == "0":
            continue
        if not parent.validate_transaction():
            valid = False
            break
    ```

  - **Create and Sign Transaction**: The new transaction is created and signed by the node.

    ```
    tx = node.create_and_sign_transaction(txid, parent_txids)
    ```

  - **Update Parent Transactions**: The parent transactions are updated for the new transaction.

    ```
    tx.parent_transactions = [self.transactions[parent_id]
    for parent_id in parent_txids]
    ```

  - **Add Transaction**: The new transaction is added to the network's transactions.

    ```
    with self.transactions_lock:
        self.transactions[txid] = tx
    ```

  - **Broadcast Transaction**: The new transaction is broadcast to all peers in the network.

    ```
    node.broadcast_transaction(tx, self)
    ```

  - **Update Tips**: The tips of the DAG are updated.

    ```
    with self.tips_lock:
        self.tips = [tx for tx in self.tips +
        list(self.transactions.values())
        if not tx.children]
    ```

  - **Update Weights**: The cumulative weights and branch weights of the DAG are updated.

    ```
    self.update_weights_topological_order()
    for tx in new_transactions:
        tx.update_branch_weight()
    ```

- **invoke_generate_milestone_after_delay**: This method runs in a separate thread and periodically instructs the coordinator to generate a new milestone.

- **update_weights_topological_order**: This function updates the weights of the transactions in a topological order of the graph.

- **simulate_node**: This function simulates the operation of a specific node in the DAG for a specified time, and transactions are added to the DAG.
  - Capturing the current time and initializing the transactions list.

    ```
    current_time = time.time()
    transactions = []
    ```

  - Getting the Poisson rate specific to the node and starting a loop to continue until the simulation time has elapsed.

    ```
    node_poisson_rate = self.poisson_rate[node.name]
    while current_time - start_time < sim_time * 60:
    ```

  - Adding transactions to the DAG, updating the branch weights, and determining the delay using the exponential distribution.

    ```
    tx = self.add_transactions(node, network)
    if tx and tx.parent_txids:
        transactions.append(tx)
    for tx in list(self.transactions.values()):
        tx.update_branch_weight()
    delay = random.expovariate(node_poisson_rate)
    time.sleep(delay)
    ```

  - Returning the transactions list at the end of the simulation.

    ```
    return transactions
    ```

- **simulate**: This function manages the entire simulation, coordinating the simulation of each node and collating the results.
  - Start by capturing the current time and initializing the transactions list.

    ```
    start_time = time.time()
    transactions = []
    ```

  - Create a list of all nodes excluding the Coordinator node, and then randomize the order.

    ```
    nodes_without_coordinator = [node for node in network.nodes if not isinstance(node, Coor
    random.shuffle(nodes_without_coordinator)
    ```

  - Create a ThreadPoolExecutor and within its context, start a parallel simulation for each node, waiting for all to complete.

    ```
    with ThreadPoolExecutor() as executor:
        futures = [executor.submit(self.simulate_node, sim_time, node, network, start_time)
        transactions = [future.result() for future in futures]
    ```

  - Flatten the transactions list and return it.

    ```
    transactions = [tx for sublist in transactions for tx in sublist]
    return transactions
    ```

- **print_weights**: This function prints the updated accumulative weights and branch weights of each transaction in the graph.

- **draw**: Though not fully implemented in the provided code, it would typically be used to draw or visualize the graph.

In general, this class represents a DAG-based data structure in the IOTA network, providing various functions to operate and manipulate it according to the IOTA protocol. Multi-threading is used in several methods to simulate the asynchronous and distributed nature of a real-world network.

3.8 Main Function

The main function of the code acts as a driver function for the entire simulation of the IOTA DAG. This function primarily configures the simulation parameters, initializes the network and DAG, starts the simulation and finally prints the output transactions generated during the simulation. It is this function where the overall execution of the system takes place, bringing together all the components and functionalities defined in the other parts of the code.

- The method begins by setting the parameters of the simulation, such as the number of nodes, the Poisson rate for generating transactions, the milestone interval, and the simulation time.

  ```
  num_nodes = 6
  poisson_rate = 0.5
  milestones_interval = 30
  sim_time = 0.3
  ```

- It then creates a network of nodes and an instance of the IOTA_DAG class with the set parameters.

  ```
  network = Network(num_nodes)
  IOTA_DAG = IOTA_DAG(poisson_rate, milestones_interval, network)
  ```

- After that, the method sets the coordinator of the IOTA_DAG and generates the genesis milestone transaction. This is the first transaction in the DAG.

  ```
  IOTA_DAG.coordinator = network.coordinator
  IOTA_DAG.coordinator_genesis_milestone()
  ```

- The state of the network is visualized using the ‘network.draw_network()‘ method.

  ```
  network.draw_network()
  ```

- The DAG object of the coordinator is then set to the DAG of the network.

  ```
  IOTA_DAG.coordinator.dag = network.dag
  ```

- The simulation then starts by calling the ‘simulate‘ method on the IOTA_DAG instance.

  ```
  transactions = IOTA_DAG.simulate(sim_time=2, network=network)
  ```

- After the simulation, the final state of the DAG is visualized using the ‘draw‘ method.

  ```
  IOTA_DAG.draw()
  ```

- Finally, the transactions that have been created during the simulation are printed.

  ```
  print(transactions)
  ```

4 Test Specifications

This section presents the test specifications for the primary features and components of the IOTA protocol as implemented in simulation. Each feature has been broken down into an individual test specification that includes the objective of the test, the required setup and configuration, the step-by-step testing procedure, and the acceptance criteria for the test to be considered successful.

The testing process is crucial for validating the functionality of our IOTA implementation, and ensuring it behaves as expected under various conditions. The specifications listed in this section provide a comprehensive road-map for effectively conducting this validation process. This includes features such as transaction approval, conflict checking, proof-of-work, site consideration, particle placement, random walks, tip approval, Accumulative weight, and tip update.

4.1 Transaction Approval

The process of Transaction Approval forms the core of IOTA protocol, with each node approving two prior transactions whenever it creates a new one. It is, therefore, critical that this process operates as expected. The following table 2 outlines the test specification for this key aspect of the system.

Table 2: Test Specification: Transaction Approval

Item	**Description**
Test Objective	To confirm that a node correctly selects and approves two other transactions according to the prescribed algorithm.
Test Setup	Requires a functioning node in the Tangle with existing transactions and the ability to generate and propagate new transactions.
Test Configurations	A functioning IOTA node with the following configurations: - Node: Latest IOTA reference implementation - Tangle DAG: Populated with at least 10 transactions
Test Procedure	1. Generate a new transaction from the node. 2. Note the transactions that the new transaction approves. 3. Confirm that these approved transactions were either tips or approved by the node's algorithm.
Acceptance Criteria	The test passes if the node correctly selects and approves two transactions. If the node does not select or approve correctly, the test fails.

4.2 Conflict Checking

In the context of IOTA Tangle, ensuring that conflicting transactions are not simultaneously approved is vital for maintaining the system's integrity. This test specification is designed to verify that the implemented conflict checking mechanisms function as expected in table 3.

Table 3: Test Specification: Conflict Checking

Item	**Description**
Test Objective	To verify that the node does not approve conflicting transactions.
Test Setup	Requires a functioning node in the Tangle with existing transactions and the ability to generate and propagate new transactions.
Test Configurations	A functioning IOTA node with the following configurations: - Node: Latest IOTA reference implementation - Tangle DAG: Populated with at least 10 transactions
Test Procedure	1. Generate two new transactions from the node that conflict with each other. 2. Check if the node approves both conflicting transactions.
Acceptance Criteria	The test passes if the node does not approve both conflicting transactions. If the node approves both, the test fails.

4.3 Proof of Work

Proof of Work (PoW) is a fundamental aspect of many blockchain technologies, including IOTA Tangle. The mechanism helps to prevent spam and denial-of-service attacks. This test specification in table 4 focuses on ensuring that the node correctly solves a cryptographic puzzle before issuing a transaction.

Table 4: Test Specification: Proof of Work

Item	**Description**
Test Objective	To confirm that the node correctly solves a cryptographic puzzle before issuing a transaction.
Test Setup	Requires a functioning node in the Tangle with the ability to generate and propagate new transactions.
Test Configurations	A functioning IOTA node with the following configurations: - Node: Latest IOTA reference implementation - Tangle DAG: Populated with at least 10 transactions
Test Procedure	1. Generate a new transaction from the node. 2. Check if the hash of the nonce and transaction data has the required number of leading zeros (difficulty level).
Acceptance Criteria	The test passes if the transaction meets the difficulty requirement. If it does not, the test fails.

4.4 Random-Walk:Consider Sites

As part of the tip selection process, the algorithm must consider all sites within a certain interval. This test specification in table 5 aims to ensure that this critical step is performed correctly.

Table 5: Test Specification: Consider Sites

Item	**Description**
Test Objective	To confirm that the algorithm correctly considers all sites within the specified interval.
Test Setup	Requires a functioning node in the Tangle with the ability to generate and propagate new transactions.
Test Configurations	A functioning IOTA node with the following configurations: - Node: Latest IOTA reference implementation - Tangle DAGe: Populated with at least 10 transactions
Test Procedure	1. Generate a new transaction. 2. Verify that the algorithm considers all sites within the specified interval.
Acceptance Criteria	The test passes if the algorithm successfully considers all sites within the specified interval. If not, the test fails.

4.5 Random-Walk: Place Particles

The algorithm's task of distributing a predetermined number of particles across sites within a certain interval is a crucial aspect of its functionality. This test specification in table 6is designed to ensure the accurate execution of this process.

Table 6: Test Specification: Place Particles

Item	**Description**
Test Objective	To confirm that the algorithm correctly places a predefined number of particles on the sites within the specified interval.
Test Setup	Requires a functioning node in the Tangle with the ability to generate and propagate new transactions.
Test Configurations	A functioning IOTA node with the following configurations: - Node: Latest IOTA reference implementation - Tangle DAGe: Populated with at least 10 transactions
Test Procedure	1. Generate a new transaction. 2. Place N particles on the sites within this interval. 3. Verify that the particles are correctly distributed across the specified sites.
Acceptance Criteria	The test passes if the particles are correctly distributed across the specified sites. If not, the test fails.

4.6 Random-Walk: Random Walks (URW/BRW)

Each particle within the system performs a deterministic random walk, either of Uniform Random Walk (URW) type or Biased Random Walk (BRW) type, depending on the value of the parameter α. This test specification in table 7 aims to ensure this process operates as expected.

Table 7: Test Specification: Random Walks (URW/BRW)

Item	**Description**
Test Objective	To confirm that each particle performs a deterministic random walk towards the tips according to cumulative weight.
Test Setup	Requires a functioning node in the Tangle with the ability to generate and propagate new transactions.
Test Configurations	A functioning IOTA node with the following configurations: - Node: Latest IOTA reference implementation - Tangle DAGe: Populated with at least 10 transactions
Test Procedure	1. Generate a new transaction from the node. 2. Let each particle perform an independent random walk towards the tips. 3. Depending on the value of α, the walk should be either Uniform Random Walk (URW) or Biased Random Walk (BRW). 4. Verify that the walk is performed correctly according to the cumulative weight and the type of walk (URW/BRW).
Acceptance Criteria	The test passes if the walk is performed correctly according to the cumulative weight and the type of walk (URW/BRW). If not, the test fails.

4.7 Random-Walk: Tip Approval

The Tangle protocol involves a selection process whereby the first two random walkers reaching the tips of the transaction graph are selected. The purpose of this test specification in table 8 is to ensure the integrity and correctness of this selection process.

Table 8: Test Specification: Tip Approval

Item	**Description**
Test Objective	To confirm that the first two random walkers reaching the tips are correctly selected.
Test Setup	Requires a functioning node in the Tangle with the ability to generate and propagate new transactions.
Test Configurations	A functioning IOTA node with the following configurations: - Node: Latest IOTA reference implementation - Tangle DAGe: Populated with at least 10 transactions
Test Procedure	1. Generate a new transaction from the node. 2. Let each particle perform an independent random walk towards the tips. 3. The first two reaching walkers should be selected as tips. 4. Verify that the selected tips are those reached first by the walkers.
Acceptance Criteria	The test passes if the selected tips are those reached first by the walkers. If not, the test fails.

4.8 Key Metrics:Cumulative Weight

This section focuses on the testing of the Cumulative Weight Calculation, a critical metric in the IOTA protocol. The cumulative weight of a transaction is drive the random walk, making it a fundamental part of the protocol's functioning. The specifications for testing this metric are as detailed in the following table 9.

Table 9: Test Specification: Cumulative Weight Calculation

Item	**Description**
Test Objective	To confirm that the cumulative weight of a transaction in the Tangle is calculated accurately.
Test Setup	Requires a Tangle with several existing transactions and an ability to introduce new transactions to it.
Test Configurations	A functioning IOTA node with the following configurations: - Node: Latest IOTA reference implementation - Tangle DAG: Populated with at least 10 transactions
Test Procedure	1. Note the cumulative weight of all transactions in the Tangle. 2. Introduce a new transaction to the Tangle, noting its own weight and the transactions that directly approves it. 3. Calculate the expected cumulative weight for this transaction. 4. Compare the expected cumulative weight to the actual cumulative weight assigned to the transaction in the Tangle.
Acceptance Criteria	The test passes if the calculated and actual cumulative weights match. Any deviation is a test failure.

4.9 Key Metrics: Tips

Within the Tangle network, unapproved transactions are known as tips. A newly introduced transaction, upon approving existing tips, becomes a tip itself. This test specification in table 10 is designed to validate this behavior.

Table 10: Test Specification: Tips

Item	Description
Test Objective	To confirm that a new transaction becomes a tip after approving existing tips.
Test Setup	Requires a functioning node in the Tangle with the ability to generate and propagate new transactions.
Test Configurations	A functioning IOTA node with the following configurations: - Node: Latest IOTA reference implementation - Tangle DAG: Populated with at least 10 transactions
Test Procedure	1. Generate a new transaction from the node, making sure it approves existing tips. 2. Verify that the new transaction becomes a tip.
Acceptance Criteria	The test passes if the new transaction becomes a tip after approving existing ones. If not, the test fails.

5 Feature Validation

In this section, we validate and verify the functionality of the various features of our IOTA Tangle implementation against the specifications outlined in the IOTA white paper. It follow a systematic approach, examining each feature one-by-one, providing a detailed description of its function, stating our implementation assumptions, and then reporting on the status of its implementation. Furthermore, describes the expected behavior of each feature, in alignment with the white paper's specifications, and provide a status update on whether the actual behavior aligns with our expectations during testing. This rigorous process helps ensure the integrity and reliability of our IOTA Tangle implementation.

5.1 Steps for a Node to issue a new transaction Testing and validation

Table 11 presents the basic features of steps required by node to issue a new transaction.

Table 11: Testing and Validation of Components: Steps for a Node to issue a new transaction

Feature	Description (from IOTA white paper)	Assumptions	Implementation Status	Expected Behavior	Test Status
Transaction Approval (Two)	The node chooses two other transactions to approve according to an algorithm.	1- Two transaction may coincide. 2- Select two tips if available, else get one tip. (Further details of tip selection is presented in table X)	Implemented 3.6	The node should be able to choose two/one transactions to approve.	Passed 2

Continued on next page

Table 11 – continued from previous page

Feature	Description (from IOTA white paper)	Assumptions	Implementation Status	Expected Behavior	Test Status
Conflict Checking	The node checks if the two transactions are not conflicting, and does not approve conflicting transactions.	1-Transaction ID is not in the own transaction list of Node, nor in selected tips and their ancestors. [1] 2- Signature check. 3- PoW nonce check 3.3	Implemented 3.3.1	The node should not approve conflicting transactions	No Conflict designed (Yet) 3
Proof of Work	the node must solve a cryptographic puzzle similar to those in the Bitcoin blockchain.	The hash of the nonce and transaction data must have at least a predefined number of leading zeros (Difficulty)	Implemented 3.3.1	Nodes should solve a cryptographic puzzle (proof of work) before issuing a transaction.	Passed 4

5.2 Random-Walk Algorithm Validation

The Random-Walk algorithm's testing and validation involves a thorough examination of its various components, as outlined in the Table 12. Each feature is evaluated against its description from the IOTA white paper, assumptions made during its implementation, its current implementation status, expected behavior, and testing status.

Table 12: Testing and Validation of Components of RandomWalk

Feature	Description (from IOTA white paper)	Assumptions	Implementation Status	Expected Behavior	Test Status

Continued on next page

[1]The IOTA White paper lacks a precise definition of what constitutes a conflict. Therefore, for our implementation, we've assumed that each transaction in the network has a unique identifier. Conflict detection has been incorporated via two checks: 1) The transaction a node generates cannot already be in that node's own transaction list; 2) There cannot exist a transaction with the same ID among the parent transactions or their ancestors. Note that, in actual blockchain networks, these kinds of checks typically based on the balance against each node/user or UTXO. However, in our current context, these two measures form the basis for conflict detection.

Table 12 – continued from previous page

Feature	Description (from IOTA white paper)	Assumptions	Implementation Status	Expected Behavior	Test Status
Consider Sites	Consider all sites on the interval [W, 2W], where W is reasonably large.	Some Interval i.e. Transaction generated between t1 to t2. [2]	Implemented	The algorithm should successfully consider all sites within the specified range.[3]	Passed 5
Place Particles	Independently place N particles on sites in that interval.	The number of particles, N, is predefined and sufficient.	Implemented	Particles should be distributed across the specified sites.[4]	Passed 6
Random Walks (URW/BRW)	Let these particles perform independent discrete-time random walks "towards the tips", meaning that a transition from x to y is possible if and only if y approves x	1- Tips approval is a deterministic process. 2- If $\alpha = 0$, we have Uniform Random Walk (URW), if $\alpha > 0$, we have Biased Random Walk (BRW).	Implemented	Each particle performs a (deterministic) random walk towards the tips according to accumulative weight. Based on the value of α, the system should follow either URW or BRW. 3.6	Passed 7

Continued on next page

[2]

- W, set as W seconds, signifies the start of the interval for site consideration.
- The end of the interval, 2W, should fall within the DAG's age.
- If the DAG's age is less than 2W, the end interval adjusts to the DAG's age.
- The age of the DAG must be at least 2W seconds for the algorithm's optimal operation.

[3]The line of code that selects interval transactions is `interval_transactions = [tx for tx in dag.transactions.values() if start_time <= tx.timestamp <= end_time]`

[4]`if len(interval_transactions) < self.N:` - The code checks if the number of transactions in the interval is less than the number of particles N to be placed.

`start_transactions = interval_transactions` - If there are fewer transactions than N, all transactions from the interval are chosen as starting points for the particles.

`start_transactions = random.sample(interval_transactions, self.N)` - If there are N or more transactions in the interval, a random subset of N transactions is selected as starting points for the particles.

Table 12 – continued from previous page

Feature	**Description (from IOTA white paper)**	**Assumptions**	**Implementation Status**	**Expected Behavior**	**Test Status**
Tip Approval	The two random walkers that reach the tip set first will sit on the two tips that will be approved. However, it may be wise to modify this rule in the following way: first discard those random walkers that reached the tips too fast because they may have ended on one of the "lazy tips".	1- Current implementation does not include Lazy tip. 2- if two unique tips were reached, selected tips randomly. 3- Return the two transactions that reached the tip set first.	Implemented	The first two reaching walkers are selected [5]	Passed 8

5.3 Key Metrics in the IOTA Tangle Testing and Validation

Table 13 presents the validation results for a select set of key metrics in the IOTA Tangle. These include Cumulative Weight and Tips, each summarized with their definition, underlying assumptions, implementation status, expected behavior, and test status. While other metrics like height, depth, and score are also described in the IOTA white paper 1.2, they are not directly used in the Random Walk or the steps for a node to generate a transaction, and therefore haven't been included in the current implementation and this specific validation.

Table 13: Key Metrics in the IOTA Tangle Testing and Validation

Feature	**Description (from IOTA white paper)**	**Assumptions**	**Implementation Status**	**Expected Behavior**	**Test Status**
Cumulative Weight	The cumulative weight of a transaction is the sum of its own weight plus the weights of all transactions that approve it.	Each transaction is approved by other transactions, either directly or indirectly.	implemented	Cumulative weights updates when new transaction added 3.7 .	Passed 9

Continued on next page

[5]

1. The loop continues until at least two unique tips have been reached.
2. If fewer tips are available, the loop breaks. This could occur if there is only one tip in the DAG.
3. If more tips are available, a random tip is selected, its path computed, and the tip is added to the list of reached tips.
4. The paths to the randomly reached tips are stored.

Table 13 – continued from previous page

Feature	Description (from IOTA white paper)	Assumptions	Implementation Status	Expected Behavior	Test Status
Tips	Tips are unapproved transactions in the Tangle.	Every Node when generates or receive the transaction updates it transaction list.[6]	implemented	When a new transaction arrives and approves existing tips, it becomes the new tip. 3.3.1.	Passed 10

6 Unaddressed Components and Possible Extensions

This document primarily focuses on the main components of the IOTA protocol. Certain key metrics mentioned in the IOTA white paper, such as height, depth, and score, were not covered in depth. These metrics, while easily calculable, do not play a fundamental role in the basic operations of the IOTA protocol, and hence were not specifically included in the testing and validation processes.

Moreover, certain sections of the IOTA white paper, namely "Stability of the system, and cut-sets" and "How fast does the cumulative weight typically grow", were not tested in the current version of the protocol.

Future versions of this specification may expand on these components based on their relevance to the operation of the protocol. As the protocol continues to develop and evolve, it may become necessary to devise new tests and validation procedures for these additional components.

7 Conclusion

In conclusion, this document presents a comprehensive specification for the implementation of the IOTA protocol, focusing specifically on key components such as the Random Walk algorithm, various key metrics like cumulative weight, tips, etc., and the processes that a node follows to generate new transactions. The class and sequence diagram further provided a visual representation of the process flow during the creation and propagation of transactions within the network.

This implementation specification lays the foundation for developers and researchers to understand the fundamental operations of the IOTA Tangle protocol and provides a starting point for future enhancements and optimizations. The details included in the document also enable comprehensive testing and validation to ensure the correct operation of the IOTA protocol.

[6] if not transaction.children: self.tips.append(transaction) .If the transaction does not have any children, it is added to the 'tips' list. for parent in transaction.parent_transactions: This line iterates over all parent transactions of the current transaction. if parent in self.tips: Then it checks if these parent transactions are in the 'tips' list. self.tips.remove(parent).If a parent transaction is in the 'tips' list, it gets removed.

References

[Pop18] Serguei Popov. The tangle. *White paper*, 1(3):30, 2018.