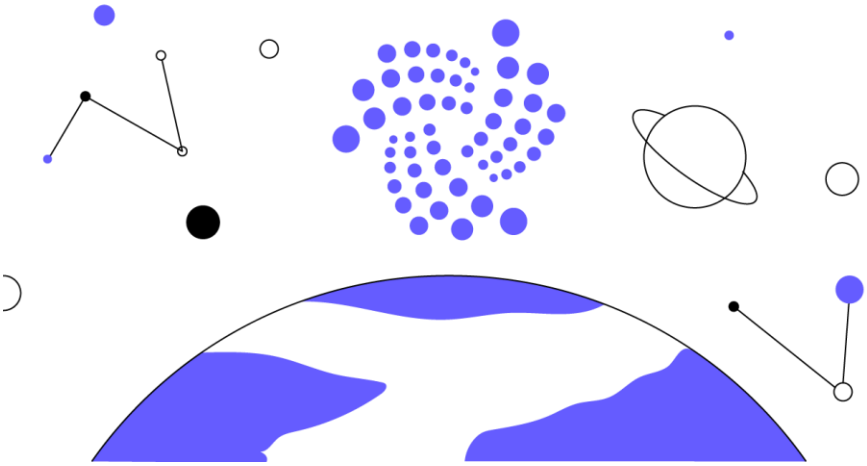
IOTA Consensus Algorithm Implementation



Negin Jafari

Helia Ghahraman

Minoo Mohaghegh

What Is IOTA?

IOTA is the first distributed ledger built for the “Internet of Everything” - a network for exchanging value and data between humans and machine entities.

IOTA is designed entirely without fees. When a node sends an IOTA transaction you validate two other transactions. This allows IOTA to overcome the cost and scalability limitations of blockchain.

The Tangle is IOTA's network. It immutably records the exchange of data and value. It ensures that the information is trustworthy and cannot be tampered with nor destroyed.

IOTA Is highly scalable, IOTA uses a DAG data structure allowing transactions to be added in parallel, unlike blockchain alternatives.

IOTA has fast transactions which are confirmed within minutes. Honest messages are approved very quickly and efficiently.

Google Colab Notebook

♦ Overview

Step 1: Calculating Hash

In this step, the algorithm involves finding a nonce that satisfies a defined target, resulting in the calculation of a cryptographic hash. This process is fundamental to ensuring the security and integrity of transactions within the IOTA consensus algorithm.

Step 2: Selecting Two Transactions

To issue a transaction, users must work to approve other transactions. The second step focuses on the careful selection of two transactions. These selection is from tips.

Step 3: Checking Transaction Conflict

Before proceeding, the algorithm checks if the two selected transactions are non-conflicting.

Step 4: Applying Required Changes

After selecting two transactions, cumulative weight of involved transactions, will be incremented, the algorithm checks if the cumulative weight surpasses or equals the specified threshold (confirmed_threshold). If it does, the transaction is marked as confirmed

Step 5: Updating Cumulative Weights and Confirmation

In the final step, the algorithm updates the cumulative weights of transactions and performs checks to confirm certain transactions.

♦ Classes

Entity

The Entity class represents a participant or node in the IOTA consensus algorithm. Each entity is identified by a unique id and maintains a list of transactions associated with it.

```
class Entity:
    def __init__(self, id):

        self.id = id
        self.entityTransactions = []
```

Transaction

The Transaction class models a transaction in the IOTA consensus algorithm. Transactions are created with a unique id, a timestamp, and the sender entity's information. Each transaction includes data, weight, cumulative weight, and cryptographic hash. It also keeps track of transactions it approves and transactions that approve it.

```
class Transaction:
    def __init__(self, id, timestamp, sender_entity):
        self.id = id
        self.timestamp = timestamp
        self.sender_entity = sender_entity

        self.data = self.get_random_string()
        self.weight = 1
        self.cumulative_weight = self.weight
        self.hash = None
        self.approved_transactions = []
        self.approved_by_transactions = []
        self.previous_hashes = []
```

♦ Methods

send_transaction

This method represents the process of sending a new transaction within the IOTA consensus algorithm. It follows the defined steps for transaction creation, selection, validation, and updating of cumulative weights.

```
def send_transaction(self):
    timestamp = time.time()
    tx_id = random.randint(0, 100)
    transaction = Transaction(tx_id, timestamp, self.id)

    # step1
    transaction.hash = transaction.calculate_hash()
    print('Transaction hash: ', transaction.hash)

    # step2
    selected_transactions = self.transaction_selection(tipsList, unconfirmedList, 2)
    print("\nSelected transactions' IDs:")
    for tx in selected_transactions:
        print(tx.id)

    # step3
    self.validate_transactions(selected_transactions)

    # step4
    for tx in selected_transactions:
        tx.approved_by_transactions.append(transaction)
        transaction.approved_transactions.append(tx)

        # add the current transaction hash to the selected transaction header
        tx.previous_hashes.append(transaction.hash)

        # update tips list
        if tx in tipsList:
            tipsList.remove(tx)
            unconfirmedList.append(tx)
```

```
    # add the transaction to the transaction list of the current node
    self.entityTransactions.append(transaction)

    tipsList.append(transaction)
    print("\nTransaction is added to the tangle.")

    # step5
    transaction.update_cumulative_weights()

    # check if the cumulative weight of unconfirmed transactions reached the threshold
    for tx in unconfirmedList:
        if tx.cumulative_weight >= confirmed_threshold:
            unconfirmedList.remove(tx)
            confirmedList.append(tx)

    # print transaction details
    print(transaction)
    print("\nTip List: ", [tx.id for tx in tipsList])
    print("\nUnconfirmed List: ", [tx.id for tx in unconfirmedList])
    print("\nConfirmed List: ", [tx.id for tx in confirmedList])
```

Steps:

Step 1: Calculating Hash

Generates a new transaction with a unique ID, timestamp, and sender entity ID. Calculates the hash of the transaction and prints the result.

Step 2: Selecting Two Transactions

Calls the transaction_selection method to select two transactions from the tipsList and unconfirmedList based on the defined criteria. Prints the IDs of the selected transactions.

Step 3: Checking Transaction Conflict

Invokes the validate_transactions method to ensure the selected transactions are non-conflicting.

Step 4: Applying Required Changes

Updates the approved transactions for both the selected transactions and the newly created transaction. The hash of the new transaction is added to the selected transactions' headers (previous_hashes list). Finally, the status of transactions in the tipsList and unconfirmedList is adjusted.

Step 5: Updating Cumulative Weights and Confirmation

Adds the new transaction to the current node's transaction list. Updates the tipsList and checks if any unconfirmed transactions have reached the cumulative weight threshold for confirmation.

transaction_selection

It initially selects transactions from the end of the tipsList, representing the latest tips in the tangle.

If the number of selected transactions is less than the desired count (num_txs), it completes the selection from the unconfirmedList.

If there are enough transactions in the unconfirmedList to meet the requirement, it adds them to the selection.

If the unconfirmedList is insufficient to meet the desired count, it selects all available transactions. Finally it returns selected transactions.

```
def transaction_selection(self, tipsList, unconfirmedList, num_txs):
    selected_transactions = tipsList[-num_txs:]
    num_selected_txs = len(selected_transactions)

    if num_selected_txs < num_txs:
        num_remaining = num_txs - num_selected_txs

        if len(unconfirmedList) >= num_remaining:
            selected_transactions.extend(unconfirmedList[-num_remaining:])
        else:
            selected_transactions.extend(unconfirmedList[:])

    return selected_transactions
```

validate_transactions

This method validates a list of selected transactions, ensuring their integrity within the IOTA consensus algorithm. If any transaction is found to be invalid, the method initiates a reselection process to find a replacement transaction. The reselection involves creating temporary lists excluding the initially selected transactions and searching for a valid replacement using the transaction_selection method.

If a valid replacement is found, the method updates the list of selected transactions by removing the invalid transaction and appending the replacement. The process is repeated until all selected transactions are valid.

```
def validate_transactions(self, selected_transactions):
    new_selection = False

    for tx in selected_transactions:
        valid = True

        if not self.is_transaction_valid(tx):
            new_selection = True
            valid = False

            # create temp tip list and unconfirmed list to search for valid transactions
            temp_tipsList = [x for x in tipsList if x not in selected_transactions]
            temp_unconfirmedList = [x for x in unconfirmedList if x not in selected_transactions]

            print(f'\nTransaction {tx.id} is not valid. finding another transaction ...')

            while not valid:
                new_tx = self.transaction_selection(temp_tipsList, temp_unconfirmedList, 1)

                if not new_tx:
                    valid = True
                    print("\nThere is no other transaction to choose.")

                elif self.is_transaction_valid(new_tx):
                    valid = True
                    selected_transactions.remove(tx)
                    selected_transactions.append(new_tx)

                else: # if new transaction is also invalid
                    # remove the invalid transaction form tip/unconfirmed list not to choose it again
                    if new_tx in temp_tipsList:
                        temp_tipsList.remove(new_tx)
                    else:
                        temp_unconfirmedList.remove(new_tx)

    if new_selection:
        print("\nNewly selected transactions' IDs:")
        for tx in selected_transactions:
            print(tx.id)
```

is_transaction_valid

To check whether or not a transaction selected by a new transaction is valid, the method checks if the hash of the selected transaction is identical to the hash stored in the previous_hashes list in all of the transactions approved by this transaction. This ensures that no changes have been made in the selected transaction. If a node attempts to modify the data within a transaction, then a new hash should be calculated for the transaction and thus, the hash of the transaction in each of the approved transactions no longer match the current hash, in which case the method returns the False value.

```
    def is_transaction_valid(self, transaction):
        valid = True

        for tx in transaction.approved_transactions:
            if transaction.hash not in tx.previous_hashes:
                valid = False

        return valid
```

calculate_hash

This method computes the hash for a transaction within the IOTA consensus algorithm. The hash is generated by concatenating the SHA-256 hash of the transaction data and a nonce. The process involves repeatedly generating a random nonce, hashing it, and combining it with the hashed transaction data until the resulting block hash meets a specified target prefix (in this case, "0000").

```
def calculate_hash(self):
    block_hash = ""
    target = "0000"

    data_hash = hashlib.sha256(self.data.encode()).hexdigest()
    # Convert data_hash hexadecimal strings to bytes for concatination later on
    data_hash_bytes = bytes.fromhex(data_hash)

    while not block_hash.startswith(target):
      nonce = self.get_random_string()
      nonce_hash = hashlib.sha256(nonce.encode()).hexdigest()

      # Convert nonce_hash hexadecimal strings to bytes for concatination
      nonce_hash_bytes = bytes.fromhex(nonce_hash)

      # Concatenate the bytes
      concatenated_bytes = data_hash_bytes + nonce_hash_bytes
      # Hash the concatenated bytes using SHA-256
      block_hash = hashlib.sha256(concatenated_bytes).hexdigest()

    return block_hash
```

calculate_cumulative_weight

This method calculates the cumulative weight of a transaction within the IOTA consensus algorithm. The cumulative weight is determined by traversing the transactions in the approval graph starting from the current transaction. It uses a breadth-first search (BFS) approach to visit and sum the weights of all transactions that approve the current transaction.

```
def calculate_cumulative_weight(self):
    visited = set()
    total_weight = 0

    queue = deque([self])
    visited.add(self)

    while queue:
        current_transaction = queue.popleft()
        total_weight += current_transaction.weight

        for approved_transaction in current_transaction.approved_by_transactions:
            if approved_transaction not in visited:
                visited.add(approved_transaction)
                queue.append(approved_transaction)

    self.cumulative_weight = total_weight
```

update_cumulative_weights

update cumulative weights of transactions after adding a new transaction to the tangle

```
def update_cumulative_weights(self):
    visited = set()
    for tx in self.approved_transactions:
        queue = deque([tx])
        visited.add(tx)

    while queue:
        current_transaction = queue.popleft()
        current_transaction.cumulative_weight += self.weight

        for tx in current_transaction.approved_transactions:
            if tx not in visited:
                visited.add(tx)
                queue.append(tx)
```

♦ Output

First we create a list of nodes and Genesis transaction.

```
tipsList = []  # Global FIFO list of tips
unconfirmedList = []  # Global list of unconfirmed transactions
confirmedList = []  # Global list of confirmed transactions
confirmed_threshold = 5  # threshold of cumulative weight for confirming a transaction

# genesis
genesis = Transaction(0, time.time(), 0)
tipsList.append(genesis)

# other nodes
node_1 = Entity(1)
node_2 = Entity(2)
node_3 = Entity(3)
node_4 = Entity(4)
node_5 = Entity(5)
```

Adding transactions:

```
node_1.send_transaction()
```

```
Transaction hash:  00005317394faafaac66daf969750706164e0b82a968a71df77c15f8f8c04115

Selected transactions' IDs:
0

Transaction is added to the tangle.

**********
Transaction details
**********
Transaction ID: 48
Transaction data: 9xzcy
Transaction weight: 1
Transaction cumulative weight: 1
Approved Transactions: 0

Tip List:  [48]

Unconfirmed List:  [0]

Confirmed List:  []
```

```
node_3.send_transaction()
```

```
Transaction hash:  0000711b05a5c764670a8e3d8ed38320ac4c4dfdba346793e80345f530286d65

Selected transactions' IDs:
48
0

Transaction is added to the tangle.

**********
Transaction details
**********
Transaction ID: 54
Transaction data: uzm31
Transaction weight: 1
Transaction cumulative weight: 1
Approved Transactions: 48, 0

Tip List:  [54]

Unconfirmed List:  [0, 48]

Confirmed List:  []
```

```
node_5.send_transaction()
```

```
Transaction hash:  000098ba18bf8393b2b34fd1109d3180469ea003cbf51029971275a731bc4137

Selected transactions' IDs:
54
48

Transaction is added to the tangle.

**********
Transaction details
**********
Transaction ID: 6
Transaction data: 783x1
Transaction weight: 1
Transaction cumulative weight: 1
Approved Transactions: 54, 48

Tip List:  [6]

Unconfirmed List:  [0, 48, 54]

Confirmed List:  []
```

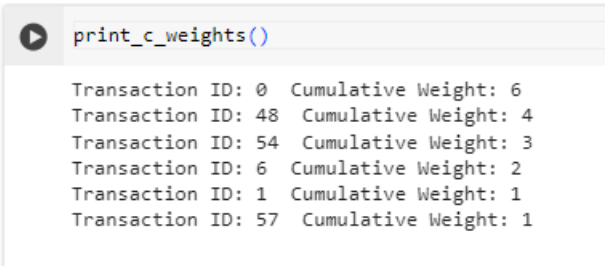
```
node_2.send_transaction()
```

```
Transaction hash:  000029a2fd30059bdaa09b97f6c40617e2db444b76bf0e5cb1519e954d0b06e7

Selected transactions' IDs:
6
54

Transaction is added to the tangle.

**********
Transaction details
**********
Transaction ID: 1
Transaction data: ef3vd
Transaction weight: 1
Transaction cumulative weight: 1
Approved Transactions: 6, 54

Tip List:  [1]

Unconfirmed List:  [48, 54, 6]

Confirmed List:  [0]
```

```
node_4.send_transaction()
```

```
Transaction hash:  00004b237670a9a6a4400f511588da40be543f355130cf951d380d1e791ac758

Selected transactions' IDs:
1
6

Transaction is added to the tangle.

**********
Transaction details
**********
Transaction ID: 57
Transaction data: shdqy
Transaction weight: 1
Transaction cumulative weight: 1
Approved Transactions: 1, 6

Tip List:  [57]

Unconfirmed List:  [48, 54, 6, 1]

Confirmed List:  [0]
```

DAG information after adding multiple transactions:



```
print_c_weights()

Transaction ID: 0  Cumulative Weight: 6
Transaction ID: 48  Cumulative Weight: 4
Transaction ID: 54  Cumulative Weight: 3
Transaction ID: 6  Cumulative Weight: 2
Transaction ID: 1  Cumulative Weight: 1
Transaction ID: 57  Cumulative Weight: 1
```