*UDC* 519.6 *http://dx.doi.org/10.30970/vam.2025.34.13635*

# TARRY'S ALGORITHM AND CONSISTENCY IN TREE-LIKE DISTRIBUTED SYSTEMS

**M. Terletskyi, G. Zholtkevych**

*Ivan Franko National University of Lviv,*
*1, Universytetska str., 79000, Lviv, Ukraine,*
*e-mail: Mykola.Terletskyi@lnu.edu.ua, Grygoriy.Zholtkevych@lnu.edu.ua*

In distributed systems, efficient and predictable network traversal is critical for coordination, broadcasting, and maintaining data consistency between nodes. This paper examines the behavior of Tarry's traversal algorithm in tree-like undirected networks under varying workloads, with a specific focus on consistency. Using simulated environments with 10, 100, and 1000 nodes, we initiated multiple transaction waves (T = 2, 5, 10) to evaluate the algorithm's ability to support consistent data propagation. Our results show that while Tarry's algorithm, due to its cycle-free and deterministic nature, ensures that all nodes are visited without duplication or omission, it alone is insufficient to guarantee data consistency. Although it provides a reliable traversal foundation, maintaining complete consistency in tree-like distributed systems requires additional synchronization or coordination mechanisms to ensure consistency. These findings highlight the strengths and limitations of the algorithm in supporting consistent state propagation in distributed environments.

*Key words*: Tarry's algorithm, Consistency, Tree-like distributed system.

## 1. Introduction

Distributed systems comprise a collection of processes (nodes) that do not share a common memory and communicate with each other by exchanging messages. Each node has its memory and state and can, in addition, receive and process messages, perform local calculations, change the state, or manipulate memory, and, crucially, send messages to other nodes [6]. Unlike centralized systems, which rely on a single central node for all operations, a distributed system utilizes multiple interconnected nodes that collaborate to achieve a common goal.

A tree-like distributed system is a network of independent computing nodes (processes or agents) organized in a logical or physical tree structure. This means that the communication topology among the nodes forms an undirected tree, an acyclic, connected graph where any two nodes are connected by exactly one path [9].

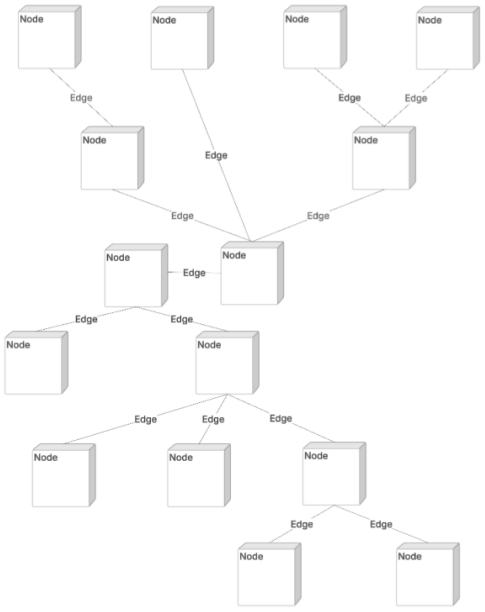


Fig. 1. Tree-like distributed system

### 1.1. Consistency

The use of distributed systems in the context of modern communication development, the Internet, and the geographical distribution of end users is growing rapidly, and this requires maintaining properties such as data consistency among nodes, the speed of message exchange between nodes, node availability, data security, the stability of reference systems, etc. The fundamental theorem of distributed systems, the CAP theorem, explains that it is possible to guarantee only two of the following three most important properties: Consistency, Availability, and Partition Tolerance [1].

This article considers data consistency, as it is one of the primary properties of distributed information systems and significantly impacts the system's reliability in providing data to the user. Data consistency, as we define it, means that all system nodes display the same data at specific points in time. Regardless of which node receives or modifies the node's data, all nodes must synchronize their data with the changes, have access to, and return the most current version of the data.

Today, consistency has many different models [2], and the models describe how nodes manage data within a distributed system, for instance:

- *Strong consistency* ensures that all nodes see the same values at the same time. When write operations are completed, subsequent reads from any node return the updated value. This model prioritizes data accuracy over availability or performance. Strong consistency is crucial for financial, medical, or e-commerce businesses where, for example, the state of the data affects client operations [11].
- *Weak consistency* means that the system can show different data, depending on the database the current service is using. In other words, the system can have multiple replicas of a single database, and after updating one of the replicas, the remaining replicas may display outdated data. This model is often used in multiplayer games or in real-time analytics. In contrast to strong consistency, weak consistency does not guarantee that all operations appear in a total order.
- *Eventual consistency* allows for temporary inconsistencies between nodes in a distributed system, with a guarantee that all nodes eventually converge to the same state. This model is prevalent in social media or networks, where it is not essential to know how many views or likes a post or news has at a given moment.
- *Sequential consistency* ensures that the results of concurrent operations appear consistent across all nodes, making it suitable for applications where the order of operations is critical but real-time constraints are not paramount. The execution results are as if all operations were executed individually, maintaining the order in which each client issues operations. The model is often used in the domain of concurrent computing, for instance, in distributed shared memory or distributed transactions.

Despite the wide range of consistency types, for the purposes of this article, we consider consistency to be either *achieved* or *failed*. Where *achieved* consistency means that all nodes return the same data, and *failed* consistency implies that at least one node returns incorrect values.

### 1.2. Algorithm

To achieve consistency in the system, we must exchange messages between its nodes, and for this, one must utilize one of the distributed algorithms [8]. In distributed systems

for broadcasting information, a special class of algorithms is used – wave algorithms. Traversal algorithms form a subclass of wave algorithms. For each algorithm of this subclass, the complete ordering of its events according to causal order is guaranteed [3, 12].

A traversal algorithm is an algorithm that adheres to three restrictions in the process of data distribution:

- Only one process is the initiator of the algorithm; that is, it is the only process whose first event is not receiving a message.
- Having received the message, the node sends the message or generates an event "decide".
- The algorithm terminates at the initiator, and at this moment, each node has sent at least one message.

This research uses Tarry's traversal wave algorithm [7] for the message exchange process. Tarry's algorithm is a traversal wave algorithm for undirected distributed networks. The algorithm is based on two rules:

1. A message is never sent twice over a single communication channel
2. A node sends a message to its parent only when it has received messages from all other neighboring nodes

The general description of how the algorithm works is as follows:

- The initiator sends a token (message) to its neighbor
- The receiving neighbor marks the sender as its parent and sends the token to all its neighbors except the parent
- If there are no neighbors except the parent, the token is returned to the parent
- When responses are received from all neighbors, the message is sent to its parent

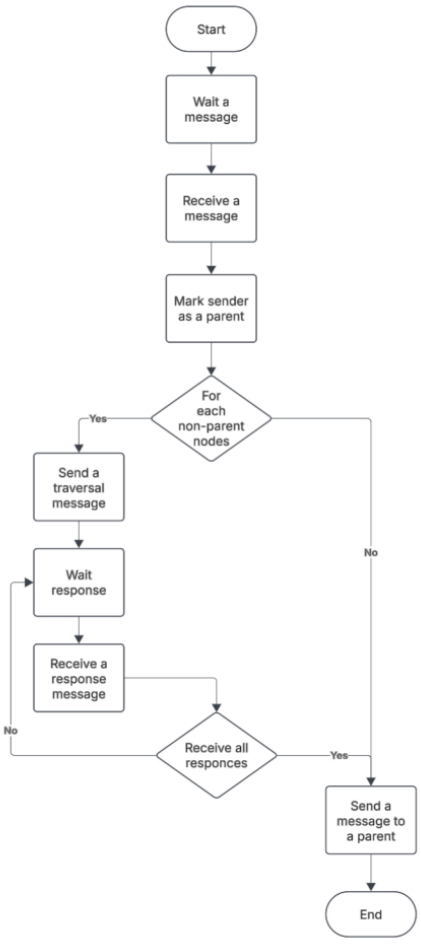


Fig. 2. Behavior of a single node in Tarry's algorithm

As a result of the algorithm, messages are sent twice over a single communication channel (once in each direction) and terminate at the initiator. Fig. 2 shows the behavior of an individual node during the execution of the algorithm.

Since Tarry's algorithm guarantees that each edge is traversed exactly twice, for a tree with N nodes, there are $N-1$ edges, resulting in $2 \cdot (N-1)$ messages per traversal.

Or simplifying, the algorithm requires $2 \cdot E$ messages, where $E$ is the number of edges in the system [3].

*Table* 1

Comparison of Wave Traversal Algorithms in Distributed Systems

| **Algorithm** | **Message Complexity** | **Structure Built** | **Termination** | **Typical Use Case** |
|---|---|---|---|---|
| Tarry's Algorithm | $2E$ (each edge twice) | Spanning tree | Token returns to initiator | Reliable network traversal, exploration |
| Tree Algorithm | $O(n-1)$ (for $n$ nodes) | Spanning tree (explicitly built) | When all nodes join the tree | Basis for broadcasting, routing, leader election |
| Depth-First Search (DFS) | $O(E)$ | DFS tree | Token returns to root after backtracking | Hierarchical exploration, spanning tree construction |
| Echo Algorithm | $2(n-1)$ | Spanning tree + echo wave | When initiator receives final echo | Distributed termination detection, information aggregation |

Above, we present the four algorithms outlined in Tabl. 1, which illustrate various approaches to wave-based traversal in distributed systems. Tarry's algorithm provides a deterministic and cycle-free method for exploring all edges, making it reliable for network exploration, though at the cost of higher message complexity. The Tree algorithm explicitly constructs a spanning tree, which then serves as a foundation for common distributed tasks such as broadcasting, routing, or leader election. In contrast, DFS traversal explores the network in depth-first order, producing a hierarchical DFS tree that is well suited for structural analysis but less efficient for broadcasting. Finally, the Echo algorithm extends the tree construction with a reverse wave that aggregates information back to the initiator, making it particularly effective for termination detection and global information collection. Together, these algorithms represent complementary strategies for traversal, each balancing determinism, communication cost, and functionality in different ways [3].

### 1.3. Goal

Tarry's algorithm ensures that each node receives a message upon completion of its execution. Therefore, we assume that after sending a message (data update transaction) to one of the system nodes, the system using Tarry's algorithm becomes eventually consistent, provided there are no network failures. This means that at the end of the algorithm's execution, every node returns the same value for the read operation request.

However, suppose that we increase the number of sent messages and the number of the algorithm's initiators accordingly. In that case, we cannot be sure that the system remains consistent after the latest message, as each algorithm execution is parallel, not sequential. Guided by this uncertain assumption, this research aims to answer the following questions:

1. Can Tarry's algorithm guarantee data consistency in tree-like distributed systems?
2. What factors influence the effectiveness of Tarry's algorithm in achieving data consistency?

## 2. Models

Our research distributed system is an undirected tree, which we understand as an undirected, connected, acyclic graph that satisfies the following characteristics [9]:

- *Undirected*: The edges do not have a direction; they connect two nodes symmetrically. If there is an edge between node A and node B, you can traverse from A to B and from B to A without restriction.
- *Connected*: Every pair of nodes is connected by exactly one simple path. There are no disconnected parts.
- *Acyclic*: The graph contains no cycles – there is no way to start at a node and return to it by traversing a sequence of edges without repeating any edge.
- *N nodes, N-1 edges*: An undirected tree with $N$ nodes always has exactly $N - 1$ edges.

Tarry's algorithm is particularly well suited for tree topologies because of its guaranteed traversal properties:

- Edge-optimal: Visits each edge exactly twice – ideal for sparse structures such as trees.
- Cycle-free operation: Naturally aligned with tree acyclicity.
- Deterministic behavior: Each traversal follows the only path between nodes.
- Local decision making: Nodes only need to know their immediate neighbors – no global knowledge is required.

For research purposes, we built several tree-like networks and sent them a certain number of messages (transactions). The following configurations are set:

1. N = 10, T = 2, with total messages: 36
2. N = 10, T = 5, with total messages: 90
3. N = 10, T = 10, with total messages: 180
4. N = 100, T = 2, with total messages: 396
5. N = 100, T = 5, with total messages: 990
6. N = 100, T = 10, with total messages: 1980
7. N = 1000, T = 2, with total messages: 3996
8. N = 1000, T = 5, with total messages: 9990
9. N = 1000, T = 10, with total messages: 19980

where $N$ is the number of nodes in a system and $T$ is the number of transactions sent to the system. The total messages are calculated using the formula $T \times 2 \times (N - 1)$.

Each configuration represents distributed systems, capturing both small- and large-scale network behaviors, shown in Tabl. 2:

*Table* 2

Types of testing distributed systems

| System size | Network size type | Real-time examples |
|---|---|---|
| 10 | Small | Sensor clusters, Classroom simulations |
| 100 | Medium | IoT systems, Small Data Centers |
| 1000 | Large-scale | Distributed databases, Peer-to-Peer and Blockchain networks |

Likewise, the variation in the number of transactions simulates the frequency or intensity of network usage. Each "transaction" represents a traversal wave initiated by a node, and the number of transactions ($T = 2, 5, 10$) can be divided into three usage groups accordingly: light or infrequent use (periodic checks), moderate use (routine system processes), high or concurrent usage (peak loads, testing robustness).

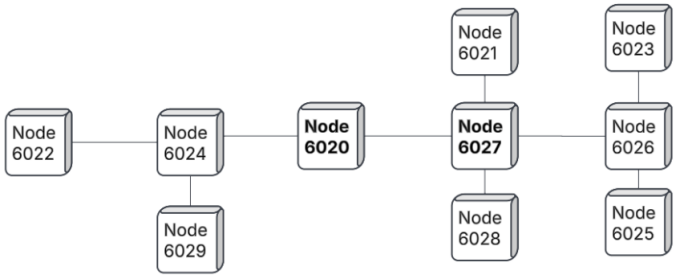


Fig. 3. Network visualization for Configuration 1

## 3. Methods

This research implements Tarry's algorithm in the dynamic functional language Elixir. In Elixir, all code runs inside processes. Processes are isolated from each other, run concurrently, and communicate by message passing. Processes are not only the basis for concurrency in Elixir, but they also provide the means to build distributed and fault-tolerant programs [4]. Therefore, this language, with its multiprocessing advantages, suits our tasks, namely, building and testing a distributed system. So, using this language allows us to interpret the algorithm as a message exchange protocol.

According to Tarry's algorithm [12], we implement the following logic to exchange messages between nodes:

- An initiator sends a message to its neighbors and waits for their responses. The initiator is the only node that does not have a parent.
- In their turn, each neighbor marks a sender as a parent and sends the message to its neighbors, except for the parent.

- If there are no neighbors, a node sends a response to its parent;
- If a node receives responses from all neighbors except the parent, it sends a response to its parent;
- The execution of the algorithms ends when the initiator receives all responses.

For clarity, we can divide the program into three separate parts:

1. *Network*. In this part, we initialize a distributed network. For current research, we set the number of nodes to 10, 100, or 1000 for testing various network sizes. To build a tree-like schema for each execution, we use a *map* data structure from the Elixir language, allowing for a maximum depth of three levels:

```
schema1 = %{parent1 => %{}}
schema1 = Enum.reduce(Enum.with_index(run1), schema1,
 fn {value, index}, schema1 ->
  update_in(schema1, [parent1], fn process ->
    schema1 = if index == 0 or (rem(index, 2) != 0
      and rem(index, 3) != 0) do
        schema1 = Map.put(process, value, %{})
    else
      schema1 = if rem(index, 3) == 0 do
        schema1 = put_in(process, [Enum.random(Map.keys(process)),
         value], %{})
      else
        level1 = Enum.random(Map.keys(process))
        schema1 = if process[level1] != %{} do
          level2 = Enum.random(Map.keys(process[level1]))
          schema1 = put_in(process, [level1, level2, value], %{})
        else
          schema1 = put_in(process, [level1, value], %{})
        end
      end
    end
  end)
end)
```

As a result, we obtain a tree-like distributed system schema with an initiator serving as the root and connections between all nodes. For instance, here we can see an example of such a schema for a system with 10 nodes:

```
%{
  #PID<0.2180.0> => %{
    #PID<0.2174.0> => %{
      #PID<0.2177.0> => %{#PID<0.2172.0> => %{}, #PID<0.2176.0> => %{}}
    },
    #PID<0.2178.0> => %{},
    #PID<0.2181.0> => %{#PID<0.2175.0> => %{}, #PID<0.2179.0> => %{}}
  }
}
```

Fig. 4 shows the schema visualization.

2. *Algorithm*. Here, we start the algorithm by randomly choosing an initiator node. Based on the configurations, we initiate 2, 5, or 10 transactions per execution. Therefore, for each execution, we choose randomly 2, 5, or 10 initiating nodes. Each node implements the significant method of the algorithm: the PUT method. The method, on receiving a message, sends, depending on its connections, messages to its neighbors to continue the traversal wave, or to a parent, signaling that there are no available nodes.

   We can see the PUT method implementation in the following code snippet:

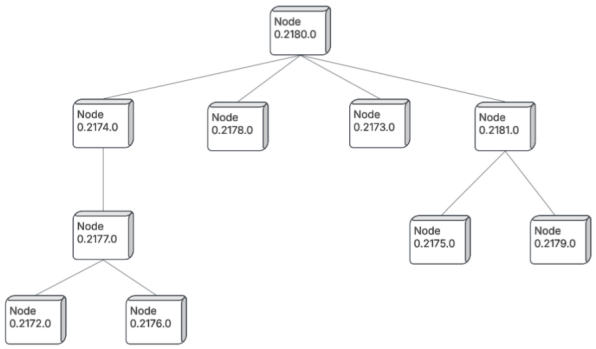


Fig. 4. Schema of distributed tree-like system

```
{:put, value, schema, printer, parent, token} ->
  map = Map.put(map, "data", value)
  map = Map.put(map, "printer", printer)
  map = Map.put(map, token, %{"parent" => parent, "servers" => nil,
   "value" => value})
  {servers} = if map_size(schema) > 0 do keys = Map.keys(schema)
    servers = Enum.map(keys, fn key -> {:"#{inspect key}", nil}
      end)
    Enum.each(keys, fn(n) -> Process.send(n, {:put, value,
      Map.get(schema, n), printer, self(), token}, [:noconnect]) end)
    {servers}
  else
    servers = []
    Process.send(parent, {:done, self(), token}, [:noconnect])
    {servers}
  end
  map = put_in(map[token]["servers"], servers)
  loop(map)
```

The second important method is the DONE method, which is responsible for processing messages sent to a parent. Upon receiving a message from a child node, the parent node marks this child node as processed and checks every node to determine if all child nodes have already sent a message. If not, the parent node waits for the next messages; if yes, it sends a message to its parent. If there is no parent, it means that this node is the initiator, so it finishes the execution of the current transaction by sending the FINISH message to the printer, which prints the results. Here, we show the DONE method implementation in the following code snippet:

```
{:done, sender, token} ->
    servers = Keyword.put(map[token]["servers"],
      :"#{inspect sender}", 1)
    map = put_in(map[token]["servers"], servers)
    printer = Map.get(map, "printer")
    parent = Map.get(map[token], "parent")
    if Enum.all?(Keyword.values(map[token]["servers"])) do
      if parent do
        send(parent, {:done, self(), token})
      else
        GenServer.call(printer, {:finish, map}, 10_000)
      end
    end
    loop(map)
```

In detail, from the node perspective, we can describe its execution behavior using the PUT and DONE methods in two main steps. First, a node receives a "PUT" call, with such parameters as: *token*, to distinguish transactions from each other; *parent*, to save it as a parent node; *printer*, to save it as a printing node; *schema*, it is part of the network schema, but limited to node neighbors only; *value*, a current transaction data value. Upon receiving the "PUT" call, a node stores a parent for the algorithm's purpose and the transaction data value in its local storage. After that, if the *schema* parameter is empty, there are no neighbors for this node, so the node sends the "DONE" call to its parent. Otherwise, it sends the "PUT" call to all its neighbors, except for the parent, with the same parameters, but sets the *parent* value to itself, and the *schema* parameter value to a receiving node's neighbors. Second, if a node receives a "DONE" call, it means that the sending node has no neighbors or has received "DONE" calls from all its neighbors. With that call, a node receives the following parameters: *sender*, an identifier of the sending node; *token*, an identifier of a transaction. When the "DONE" call is received, a node marks the sender as received and checks the status of all other neighbors. If all neighbors are marked as received, and the node has a parent, it sends a "DONE" call to its parent. If not all neighbors are marked, the node waits for the responses. If a node receives the "DONE" call and has no parent, meaning it is an initiator, it sends a final message to the printing node, which displays the current state of the node's local storage.

3. *Result.* This is a simple data output. For each program execution, it prints the latest state of the local nodes' data. The program contains a separate module, which, upon receiving the FINISH message, gathers information about the node's local state and prints it out. The module Elixir implementation is presented in the following code snippet:

```
defmodule Printer do
  use GenServer

  @impl true
  def init(elem) do
    {:ok, %{}}
  end

  @impl true
  def handle_call({:finish, all_data}, from, state) do
    sender = elem(from, 0)
    IO.puts("RECEIVE FINISH from #{inspect sender}")
    new_state = Map.put(state, sender, %{})
    Enum.each(all_data["processes"], fn(n) ->
      Process.send(n, {:get, sender}, [:noconnect]) end)
    {:reply, nil, new_state}
  end

  @impl true
  def handle_call({:data, {parent, value}}, from, state) do
    sender = elem(from, 0)
    map = put_in(state[parent][sender], value)
    if Enum.count(Map.keys(Map.get(map, parent))) == 1000 do
      egn = Enum.group_by(Map.values(Map.get(map, parent)), & &1)
      for a <- Map.keys(egn), do: IO.inspect(
        "Value #{inspect a} -> #{inspect length(Map.get(egn, a))}")
    end
    {:reply, nil, map}
  end
end
```

In general, after the printer module shows results, we expect to see the current state of the local data of each node. The result can be varied depending on the number of

transactions. As the transaction number represents the data we want to store in the node, we can expect values: 1 or 2 for the execution of the program with two transactions; 1..5 for the execution of five transactions; and finally, 1..10 for the execution of ten transactions.

## 4. Results

For each of the nine configurations we listed in Section 2, we ran five executions to obtain various results. All transactions (messages) are sent successively, and the delay between each transaction ranges approximately from 0.1 to 1 millisecond.

The result figures show the state of data consistency for each run as a percentage of all correct values. We understand 100 percent consistency as achieved, a successful state. This means that all nodes show the last transaction value stored locally. Any other value that differs from 100 percent is considered a failed state of data consistency.

To consider consistency as achieved (successful) for an execution, all nodes must show the value of local storage equal to the latest transaction value. The latest transaction value always equals the number of transactions we run. In our research, the options are 2, 5, or 10.

In Fig. 5, we demonstrate the achievement of consistency for small distributed systems. The results explicitly show us how consistency depends on the frequency of the transactions. The best results were achieved for infrequent transactions, and as the number of transactions increased, the percentage of consistency decreased dramatically. We also need to mention that even the best results did not provide consistent data across all five runs. We can see that only two of the five are consistent. Runs for the moderate usage level demonstrate only one successful result, and the high usage runs yielded zero percent success, with zero percent success in four out of the five runs.

| | N=10 | | | N=10 | | | N=10 | | |
|---|---|---|---|---|---|---|---|---|---|
| | T=2 | | | T=5 | | | T=10 | | |
| | X | K | C | X | K | C | X | K | C |
| X, final value | 1 | 0 | 100,00% | 1 | 0 | 100,00% | 6 | 5 | 0,00% |
| | 2 | 10 | | 5 | 10 | | 9 | 5 | |
| | 1 | 8 | 20,00% | 3 | 4 | 60,00% | 4 | 4 | 0,00% |
| | 2 | 2 | | 5 | 6 | | 7 | 6 | |
| | 1 | 1 | 90,00% | 4 | 5 | 50,00% | 5 | 7 | 0,00% |
| | 2 | 9 | | 5 | 5 | | 6 | 3 | |
| | 1 | 1 | 90,00% | 4 | 3 | 70,00% | 3 | 5 | 0,00% |
| | 2 | 9 | | 5 | 7 | | 6 | 5 | |
| | 1 | 0 | 100,00% | 2 | 3 | 70,00% | 3 | 5 | 50,00% |
| | 2 | 10 | | 5 | 7 | | 10 | 5 | |

Fig. 5. Results for Configurations 1, 2, and 3

The subsequent experiments were conducted for medium distributed systems, and their results are visualized in Fig. 6. Despite an increasing number of nodes in the system, the result for infrequent transactions remains very similar to that of the previous configuration of the small distributed system. However, the results for moderate and high frequencies are unacceptable in achieving data consistency when using Tarry's algorithm.

The results of the last configurations, which mimic large distributed systems, do not differ much from the previous ones. Fig. 7 shows that frequency is more important in achieving data consistency than the number of nodes: the runs for infrequent transactions are much closer to the goal, unlike those for moderate and high frequency.

| | N=100 | | | N=100 | | | N=100 | | |
|---|---|---|---|---|---|---|---|---|---|
| | T=2 | | | T=5 | | | T=10 | | |
| | X | K | C | X | K | C | X | K | C |
| X, final value | 1 | 1 | 99,00% | 4 | 8 | 92,00% | 9 | 94 | 6,00% |
| | 2 | 99 | | 5 | 92 | | 10 | 6 | |
| | 1 | 1 | 99,00% | 3 | 7 | 0,00% | 9 | 97 | 3,00% |
| | 2 | 99 | | 4 | 93 | | 10 | 3 | |
| | 1 | 0 | 100,00% | 3 | 95 | 5,00% | 6 | 96 | 0,00% |
| | 2 | 100 | | 5 | 5 | | 9 | 4 | |
| | 1 | 1 | 99,00% | 2 | 6 | 94,00% | 4 | 3 | 97,00% |
| | 2 | 99 | | 5 | 94 | | 10 | 97 | |
| | 1 | 4 | 96,00% | 3 | 7 | 0,00% | 7 | 95 | 0,00% |
| | 2 | 96 | | 4 | 93 | | 8 | 5 | |

Fig. 6. Results for Configurations 4, 5, and 6

| | N=1000 | | | N=1000 | | | N=1000 | | |
|---|---|---|---|---|---|---|---|---|---|
| | T=2 | | | T=5 | | | T=10 | | |
| | X | K | C | X | K | C | X | K | C |
| X, final value | 1 | 2 | 99,80% | 2 | 4 | 0,00% | 2 | 995 | 0,00% |
| | 2 | 998 | | 3 | 996 | | 9 | 5 | |
| | 1 | 0 | 100,00% | 4 | 11 | 98,90% | 9 | 988 | 1,20% |
| | 2 | 1000 | | 5 | 989 | | 10 | 12 | |
| | 1 | 5 | 99,50% | 2 | 4 | 0,00% | 8 | 3 | 0,00% |
| | 2 | 995 | | 4 | 996 | | 9 | 997 | |
| | 1 | 7 | 99,30% | 2 | 996 | 0,00% | 7 | 6 | 99,40% |
| | 2 | 993 | | 4 | 4 | | 10 | 994 | |
| | 1 | 6 | 99,40% | 4 | 8 | 99,20% | 8 | 994 | 0,00% |
| | 2 | 994 | | 5 | 992 | | 9 | 6 | |

Fig. 7. Results for Configurations 7, 8, and 9

Summarizing, as evident from Figs. 5, 6, and 7, data consistency is a rare case for Tarry's algorithm and a tree-like distributed system. Additionally, we must emphasize that there are cases where the value of the latest transactions is not even presented in the results (resulting in a zero percentage). In these cases, we presume that some transactions were processed longer than the latest because of the way the transactions had to traverse, and that they overwrote the newest transaction value.

Therefore, one can see that achieving data consistency for tree-like networks is almost impossible using only Tarry's algorithm for message exchange, and it cannot be used for real-time projects without additional algorithms or improvements to prevent the overwriting of local states or by implementing transaction ordering.

## 5. Conclusion

The research results, as shown in Figs. 5, 6, and 7, indicate that one cannot guarantee data consistency by using only Tarry's algorithm in the case of undirected tree-like distributed networks. Furthermore, it is not a case of proving *strong* or *eventual* consistency. It is a case where we cannot achieve consistency, neither at once nor eventually, despite seeing some successful results for the configuration of the small distributed system (Fig. 5). We conclude that Tarry's algorithm ensures that each node of the system is visited and processes each transaction. However, with an increasing frequency of transactions, we are far from achieving data consistency.

However, during program execution, we found that some additional factors have an

impact on data consistency, while others do not. Here, we list these factors and discuss their impact on data consistency:

- *node location in the system.* In other words, it is the path a transaction must traverse to complete a full traversal according to the logic of Tarry's algorithm. For our tree-like, undirected, distributed system, as shown in Figs. 5, 6, and 7, we observe that fourteen of 45 runs achieved zero data consistency. This is approximately 30 percent of the total executions. We understand that for these transactions, messages should cover a more complicated path to traverse all nodes than for the last transaction of these executions, and this is determined by overwriting the last transaction values.
- *number of transactions.* It is a very influential factor. With an increase in the number of transactions, the results decrease from one run to the next. Furthermore, even for two subsequent transactions, the algorithm cannot guarantee data consistency. The more transactions we run, the more zero percentage results we see.
- *number of nodes.* Based on the results, the number of nodes does not have an impact, such as, for example, the number of transactions. This factor influences the overall execution time of the program, but does not impact the consistency of the data.
- *time delay between transactions.* It is another influential factor. As mentioned in Section 4, the time delays between each transaction ranged from 0.1 to 1 milliseconds. As we can see from the results, such a slight delay does not have a beneficial impact on consistency. Here, we can state that the placement of the node has a much greater impact on the final result. However, increasing the delay to 10-100 milliseconds reduces the impact of node location, which is not a crucial impact compared to lower delays. With that increased delay, we can receive much better results for the research algorithm and system.

According to these factors and the research results, the most effective way to achieve data consistency in an undirected, tree-like distributed system is to implement transaction ordering based on the time they occur. This neutralizes the impact of complicated paths for some transaction initiators and helps to avoid the problem of overwriting the local state data. Ordering can be achieved by considering the following enhancements: transaction centralization, physical time, or joint agreement of all nodes. The first is outside our research, as we look for a solution for distributed systems. The second enhancement complicates our research by adding a central time server or by connecting each node to external systems. The third improvement is the one for which we chose to continue further research. Its main idea is to utilize logical clock algorithm [5] and Distributed Ledger Technology (DLT) [10] as the primary common register of transactions for all nodes, which provides ordering of transactions, and to introduce an election algorithm to establish a safe method for electing a responsible node every time the system is ready to save transactions to DLT. The election method is also considered a step away from any centralization or vulnerability.

Summarizing, we can state that, based on the results provided by our research, Tarry's algorithm, by itself, cannot guarantee data consistency in tree-like undirected distributed systems and requires additional improvements to achieve data consistency. We continue our research by incorporating DLT and introducing the election algorithm into the study.

## References

1. Aldin H.N.S. Consistency models in distributed systems: A survey on definitions, disciplines, challenges and applications / Hesam Nejati Sharif Aldin // ArXiv abs/1902.03305. – 2019. url: https://api.semanticscholar.org/CorpusID:60440625.
2. Brewer E. CAP twelve years later: How the "rules" have changed / E. Brewer // Computer. – 2012. – Vol. 45.2. – P. 23–29. url: http://dx.doi.org/10.1109/MC.2012.37.
3. Diestel R. Graph Theory /Reinhard Diestel //6th. Graduate Texts in Mathematics. – Springer, 2024. doi: 10.1007/978-3-662-7.
4. Erciyes K. Trees and Traversals / K. Erciyes // Guide to Distributed Algorithms: Design, Analysis and Implementation Using Python. – Cham: Springer Nature Switzerland. – 2025. – P. 165–189. url: https://doi.org/10.1007/978-3-031-79018-8_9.
5. Fokkink W. Distributed Algorithms: An Intuitive Approach. Second / Wan Fokkink. – Cambridge, Massachusetts: The MIT Press, 2018. https://mitpress.mit.edu/9780262037-662/distributed-algorithms/.
6. Gomes Victor B.F. Verifying strong eventual consistency in distributed systems / Victor B.F. Gomes, et al. //Proc. ACM Program. Lang. 1.OOPSLA. – Oct. 2017. url: https://doi.org/10.1145/3133933.
7. Kshemkalyani A.D. An introduction to snapshot algorithms in distributed computing / A.D. Kshemkalyani, M. Raynal, M. Singhal // Distributed Systems Engineering. – Dec. 1995. – Vol. 2.4. – P. 224. url: https://dx.doi.org/10.1088/0967-1846/2/4/005.
8. Lamport L. Time, clocks, and the ordering of events in a distributed system / Leslie Lamport // Commun. ACM. – July 1978. – Vol. 21.7. – P. 558–565. url: https://doi.org/10.1145/359545.359563.
9. Lynch N.A. Distributed Algorithms / Nancy A. Lynch. – San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996.
10. Rao N. Robot Navigation in Unknown Terrains: Introductory Survey of Non-Heuristic Algorithms / Nageswara Rao. – Mar. 1993.
11. Laurent S.St. Introducing Elixir: Getting Started in Functional Programming. 1st. /Simon St. Laurent, J. David Eisenberg. – O'Reilly Media, Inc., 2014.
12. Wattenhofer R. Distributed Ledger Technology: The Science of the Blockchain. 2nd. / Roger Wattenhofer. – North Charleston, SC, USA: CreateSpace Independent Publishing Platform, 2017.

*Article: received* 06.08.2025
*revised* 17.09.2025
*printing adoption* 15.10.2025

## АЛГОРИТМ ТЕРРІ ТА УЗГОДЖЕНІСТЬ В ДЕРЕВОПОДІБНИХ РОЗПОДІЛЕНИХ СИСТЕМАХ

**М. Терлецький, Г. Жолткевич**

*Львівський національний університет імені Івана Франка,*
*вул. Університетська 1, Львів, 79000, Україна,*
*e-mail: Mykola.Terletskyi@lnu.edu.ua, Grygoriy.Zholtkevych@lnu.edu.ua*

У розподілених системах ефективний і передбачуваний обхід мережі критично важливий для координації, розповсюдження даних і підтримки узгодженості між вузлами. Досліджуємо поведінку обхідного алгоритму Террі в деревоподібних неорієнтованих мережах за різних робочих навантажень, з особливим акцентом на узгодженість. Використовуючи змодельовані середовища з 10, 100 та 1000 вузлами,

ми ініціювали кілька хвиль транзакцій (T = 2, 5, 10), щоб оцінити здатність алгоритму підтримувати узгодженість у разі поширення даних. Наші результати підтверджують, що, з одного боку, алгоритм Террі завдяки своїй безциклічній і визначеній природі гарантує, що всі вузли відвідуються без дублювання або пропусків, проте, з іншого – його самого недостатньо для гарантування узгодженості даних. Хоча він забезпечує надійну основу для обходу системи, підтримка повної узгодженості в деревоподібних розподілених системах потребує додаткових механізмів синхронізації або координації для забезпечення узгодженості. Ці результати демонструють сильні сторони та наявні обмеження алгоритму в підтримці узгоджених станів вузлів у розподілених середовищах.

*Ключові слова*: алгоритм Террі, узгодженість даних, деревоподібні розподілені системи.