User Manual

Simulator of Distributed Algorithms

**1. Introduction**

The Distributed Network Simulator is a Python-based tool designed to help users simulate, visualize, and analyze distributed algorithms on various network topologies. With a user-friendly graphical interface, you can:

* Configure network parameters (number of nodes, topology, etc.)
* Upload and test your own distributed algorithms
* Visualize the network and simulation process
* Analyze simulation results and statistics

**2. Installation and Requirements**

**Prerequisites**

* Python 3.7 or higher (recommended)
* pip (Python package manager)

**Install Required Packages**

From your project directory, run:

This will install:

* PyQt5 (for the GUI)
* networkx (for network graph operations)
* numpy (for numerical operations)

**3. How to Launch the Simulator**

To start the Distributed Network Simulator, open a terminal and navigate to your project directory:

Run the simulator:

This will launch GUI for configuring and running simulations

Or:

Choose the in your IDE and run it.

**4. Using the GUI to Configure and Run a Simulation**

When you launch the simulator, a graphical interface will appear. Here’s how to use it step by step:

**A. Main Menu Overview**

The main window allows you to configure all simulation parameters before running.

* You will see options for:
* Number of Computers
* Topology (preset options or custom)
* ID Type
* Delay
* Display (Text or Graph)
* Root selection
* Logging level
* Algorithm file upload
* Topology file upload

Изображение выглядит как текст, снимок экрана, программное обеспечение, число

Содержимое, созданное искусственным интеллектом, может быть неверным.

**B. Step-by-Step Configuration**

Follow this sequence in the GUI to prepare and launch a simulation:

1. **Number of Computers** – type the total node count. Leave this blank if you plan to upload a custom topology; the simulator will read the count from the file.
2. **Topology** – pick a built‑in layout (Random, Clique, Line, Tree, or Star), or choose Custom and upload a .txt topology file.
3. **ID Type** – decide how node identifiers are assigned: Random, Sequential, or Custom (read directly from the uploaded file).
4. **Delay Model** – select Constant, Random Constant (one random delay reused for every edge), or Random (fresh value per edge).
5. **Display Type** – Text streams events to the console; Graph opens a live visualization (ideal for ≤ 100 nodes).
6. **Root Selection** – choose No Root, Min ID, Random, or Custom (root taken from the topology file).
7. **Logging Level** – pick Short, Medium, or Long for the amount of detail recorded.
8. **Upload Algorithm File** – click *Upload Python File* and select your script. It must implement both and .
9. **Upload Topology File** – required only when Custom was selected for Topology, ID Type, or Root. Uploading automatically switches those fields to *Custom* and overrides the *Number of Computers* input.
10. **Submit** – click Submit to save the settings and start the simulation.

Notes

* Choosing Graph with more than 500 nodes prompts you to switch to Text for better performance.
* All configuration fields are mandatory; the GUI highlights any omissions.
* Every log entry (INFO, WARNING, ERROR) is written to output.txt for post‑run review.

**5. Algorithm file**

Each client algorithm should be defined as a Python module that includes:

* Optional global dictionaries: , , and .

**5.1. Required Functions**

This function is called **once per node before the simulation begins**. It is used to:

* Initialize local state (e.g., distance, parent, color).
* Set node-specific metadata like .
* Send initial messages if necessary (especially if ).

**Example:**

def init(self: computer.Computer, communication: Communication):  
 *"""  
 Initialize the node's state before starting the synchronous rounds.  
 """* if self.is\_root:  
 self.distance = 0  
 self.parent = self.id  
 self.color = colors[0]  
 else:  
 self.distance = np.inf  
 self.parent = None

def mainAlgorithm(self: computer.Computer, communication: Communication, round, messages=None):  
 if round == 0:  
 # Send initial message if root  
 if self.is\_root:  
 communication.send\_to\_all(self.id, f"INITIALIZE {self.distance} {self.parent}", round)  
 self.state = NodeState.TERMINATED  
 else:  
 for message in messages:  
 message\_parts = message.split(" ")  
 dist = float(message\_parts[1])  
 parent = int(message\_parts[2])  
  
 if dist + 1 < self.distance:  
 self.parent = parent  
 self.distance = dist + 1  
 communication.send\_to\_all(self.id, f"UPDATE {self.distance} {self.id}", round)  
 self.color = colors[int(self.distance) % len(colors)]  
 self.state = NodeState.TERMINATED

This function defines the logic of your algorithm.

* In **synchronous mode**, the signature is:
* In **asynchronous mode**, the signature is:

It runs once per round (sync) or per received message (async) and determines:

* How to update the node’s state.
* Whether and what messages to send.

Explanation of parameters:

**The Computer Class (self-attribute) – Internal API and Semantics**

The *Computer* object represents a single vertex in the simulated network and is the self you receive inside and . Understanding its mechanics lets you write cleaner, bug‑free protocols.

|  |  |
| --- | --- |
| Attribute / Method | Purpose & Behaviour |
|  | Immutable numeric identifier (row from topology). |
|  | IDs of adjacent nodes (read‑only for clients; populated by the engine). |
|  | The node’s life‑cycle. Allowed values: |
|  | Hex/CSS token the visualizer uses. Assign freely. |
|  | Convenience flag injected by the scenario loader (e.g. BFS root). |
|  | Generic per‑algorithm you decide the schema. |
|  | Auto‑increment via update helpers; handy for collapse‑by‑traffic triggers and stats. |
|  | **Do not modify directly.** The simulator polls it to know whether to repaint the GUI / issue events. Any assignment to a **public** attribute (except bookkeeping fields like id, algorithm\_file, counters, lists) sets it automatically via an overridden |
|  | Helper that simply sets and logs. |
|  | Helper that simply sets |
|  | Internal bookkeeping; rarely needed in client code. |

**Custom Fields:**

Add whatever you like. The *Computer* class is dynamic: assigning creates that attribute on the fly. Such user‑defined fields behave exactly like the built‑in ones change‑tracking and GUI refresh are triggered automatically unless the attribute name starts with “\_”.

**The Communication Class – Internals & Hooks**

The *Communication* object is the second argument supplied to every client callback. It encapsulates all message‑passing logic.

|  |  |
| --- | --- |
| **Attribute / Method** | **Purpose & Behaviour** |
|  | This method delivers a message from one node to a single neighbor. It first confirms that the two nodes are directly connected in the topology. Next, it determines how long the message should take to arrive, either by reading a configured delay or by sampling a random value. If the link must preserve first‑in‑first‑out order, the call waits until any earlier messages on the same edge have been scheduled for delivery. A Message object is then created. When corruption or loss is requested, the content is modified accordingly before the message is placed on the global priority queue. Finally, the sender’s counter for transmitted messages is incremented, and any rules that trigger a node collapse after sending are checked. |
|  | Utility wrapper that iterates over every neighbor of the source node and calls for each one. In synchronous protocols this is the usual way to broadcast a value to all adjacent nodes. |

**Parameter Glossary**

|  |  |  |
| --- | --- | --- |
| Name | Appears in | Meaning |
|  | in asynchronous mode, , Message objects | Absolute simulation timestamp (float) at which a specific message reaches its destination. Calculated as ; used to order events in the global priority queue. |
|  | in synchronous mode | Zero‑based index of the current synchronous round. In each round the engine first delivers the batch of messages for that round, then calls every node’s algorithm with this integer. |
|  | in asynchronous mode | The single payload (str, dictionary or custom type) that just arrived at the node. |
|  | in synchronous mode | A list containing all payloads delivered in that round |

**6. Topology File (.txt)**

A topology file fully specifies the initial network that the simulator will load. Each file is plain text and **must** follow the structure below—ordered exactly as shown, each header on its own line.

|  |  |  |
| --- | --- | --- |
| Header | Required | Description |
| **IDs List:** | ✔︎ | Comma‑separated list of all node identifiers that appear elsewhere in the file. |
| **Number of Computers:** | ✔︎ | Total count of nodes. Must match the length of the IDs list. |
| **Root ID:** | ✔︎ | Either a single ID from the list (designated root) **or** the literal word Random to let the simulator pick one at runtime. |
| **Edges:** | ✔︎ | Comma‑separated list of directed (or undirected) pairs written as . All IDs must be present in the IDs list. |
| **Input:** | ✖︎ (optional) | Initial per‑node data. Begin with a JSON‑like schema row, then provide lines. Missing IDs receive an empty list. |

Example:

*IDs List:*

*1,2,3,4,5,6,7,8,9,10*

*Number of Computers:*

*10*

*Root ID:*

*Random*

*Edges:*

*(1,2),(1,3),(2,4),(2,5),(3,6),(3,7),(4,8),(4,9),(5,10)*

*Input:*

*[height, weight]*

*1:[11]*

Additional samples live under **topologyFiles/** for reference.

**7.** **Error & Fault‑Injection Module**The simulator ships with an **errorModule** that lets algorithms introduce three kinds of non‑ideal behavior: node collapse, edge reordering, and message corruption/loss.

**7.1.  Collapse Config**

Create an instance implicitly by defining **collapse\_config** in your client file.

Supported keys:

|  |  |
| --- | --- |
| Key | Meaning |
|  | Per‑node rule dictionary (see below). Node ID as string. |
|  | Fraction of nodes that should fail randomly over the first rounds. |
|  | Approximate horizon (in rounds) for the Poisson‑based scheduler that spreads the random collapses; an estimate is sufficient. |

Per‑node rule dictionary fields (any combination):

* – first round in which collapse can occur (sync mode only).
* – repeat interval in rounds (default = 1).
* – Bernoulli chance applied each time the trigger is hit (default = 1).
* – collapse after this many messages **received**.
* – collapse after this many messages **sent**.

Usage example (sync):

Where the key is ID of the node.

**7.2.  Reorder Config**

Define **reorder\_config** as where is the probability that traffic on the undirected edge becomes **unordered** (may arrive out of FIFO). During initialization the simulator rolls dice and stores unordered edges.

Calling communication. tells the engine whether it needs to enforce in‑order delivery.

**7.3 Message Loss & Corruption Helpers**

Within a call you can pass a *corruption descriptor* that uses reserved constants from :

from simulator import Constants

* **Loss** – with probability 0.6 the message is silently dropped.
* **Corruption** – with probability 0.5 the specified fields are mutated. The magic string "\_RANDOM" flips bits (numeric), changes characters (string), toggles booleans, or shuffles lists.

**7.4. Putting it together – quick patterns**

|  |  |
| --- | --- |
| Scenario | Snippet |
| Collapse node 4 exactly once at round 2 |  |
| Collapse of nodes over first 4 rounds |  |
| Make edge unordered |  |
| Corrupt *distance* field with chance, drop with |  |

**8. Running and Interpreting Simulation Results**

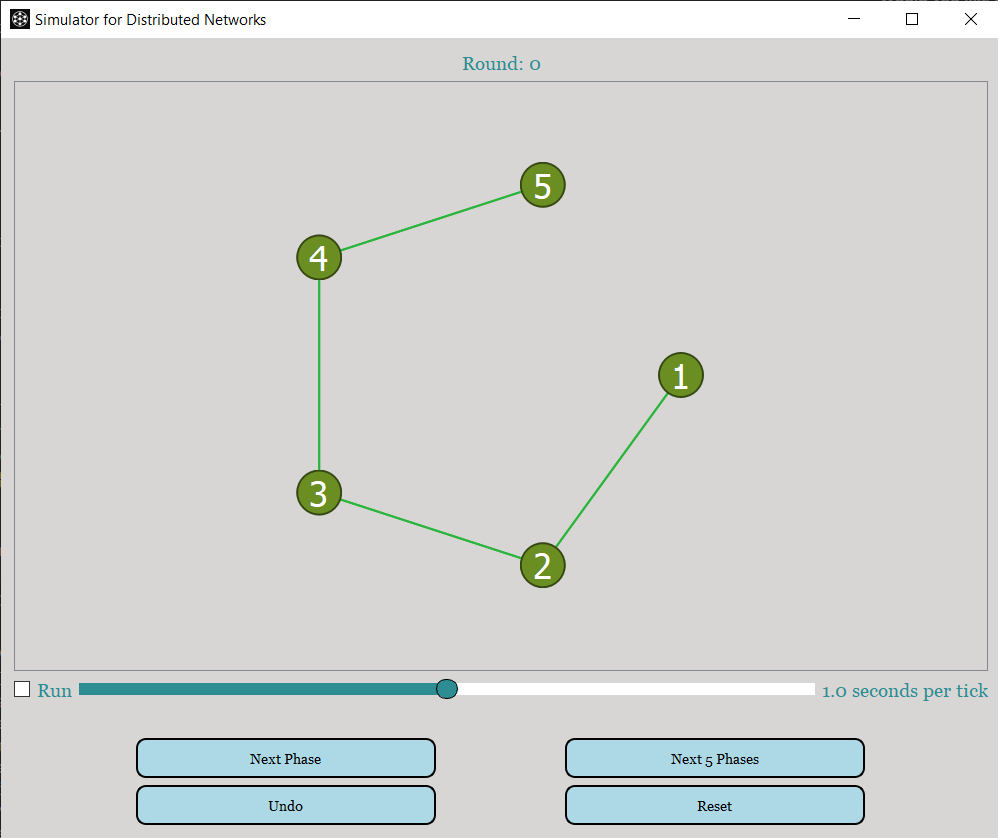
After you configure your simulation and click "Submit," the simulator will execute your chosen algorithm on the specified network.

Here’s what to expect and how to interpret the results:

**A. During the Simulation**

* **Progress Window:**

When the display type is set to **Graph**, a control panel appears beneath the network visualization. Each button advances or rewinds the simulation by *phases*:

* In **synchronous** runs a phase equals one **round**.
* In **asynchronous** runs a phase equals the delivery of **one message** to its destination node.

|  |  |
| --- | --- |
| Button | Action |
| **Next Phase** | Executes a single phase and refreshes the view. |
| **Next 5 Phases** | Executes five consecutive phases in one click. |
| **Undo** | Rewinds the simulation by exactly one phase, restoring the previous network state. |
| **Reset** | Returns the simulation to its initial state, discarding all progress and clearing the message queue. |

* **Console/Log Output:**

All simulator components share a common **Python logger** instance imported from . Your algorithm can invoke it directly to emit progress updates or flag problems:

        from utils.logger\_config import logger

        logger.info("%s is root",self.id)

        logger.warning("Received malformed message: %s",msg)

        logger.error("Node %s failed integrity check",self.id)

**Key points**

* Log levels available: **INFO**, **WARNING**, and **ERROR** (use INFO for normal milestones, WARNING for unexpected but recoverable situations, and ERROR for serious faults).
* All log entries are written to **output.txt** in the project root, so you can inspect a full trace after the simulation finishes.

Example:

2025-06-12 15:37:47,131 - INFO - Starting the simulator

2025-06-12 15:42:54,641 - INFO - file name is pathtoproject /Project/topologyFiles/line.txt

2025-06-12 16:04:21,205 - INFO - Topology file pathtoproject/topologyFiles/line.txt parsed successfully.

2025-06-12 16:04:21,208 - INFO - Computer 1 is changing is\_root to False

2025-06-12 16:04:21,208 - INFO - Computer 1 is changing inputs to {}

2025-06-12 16:04:21,208 - INFO - Computer 1 is changing outputs to {}

2025-06-12 16:04:21,208 - INFO - Computer 2 is changing is\_root to False

…

2025-06-12 16:04:21,209 - INFO - Computer 5 is changing color to olivedrab

2025-06-12 16:04:21,209 - INFO - Computer 5 is changing inputs to {}

2025-06-12 16:04:21,209 - INFO - Computer 5 is changing outputs to {}

2025-06-12 16:04:21,209 - INFO - Computer 3 is changing is\_root to True

2025-06-12 16:04:21,210 - INFO - ID 1 has input height with value 11

…

2025-06-12 16:04:21,569 - INFO - Computer 4 is changing distance to inf

2025-06-12 16:04:21,569 - INFO - Computer 5 is changing distance to inf

…

2025-06-12 16:04:21,572 - INFO - Current round: 3

2025-06-12 16:04:21,572 - INFO - sync run completed

2025-06-12 16:04:21,572 - INFO - \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

2025-06-12 16:04:21,573 - INFO - Network Statistics:

2025-06-12 16:04:21,573 - INFO - Total number of computers: 5

2025-06-12 16:04:21,573 - INFO - Total number of messages sent: 4

2025-06-12 16:04:21,573 - INFO - Total number of messages received: 4

2025-06-12 16:04:21,573 - INFO - No nodes collapsed during the simulation.

**????????????????????????????????????????????????????????????????????????????**

**B. After the Simulation**

* **Results Location:**
* **Text Output:**

Results and statistics are printed in the terminal and saved in output.txt.

* **Graph Output:**

The visualization window will display the final state of the network. You can close the window when done.

* **What’s Included in the Output:**
* Total number of computers
* Total messages sent and received
* Collapse and reorder statistics (if applicable)
* Outputs from each node (e.g., final values, states)
* Timing information (total simulation time, network creation time, algorithm run time)

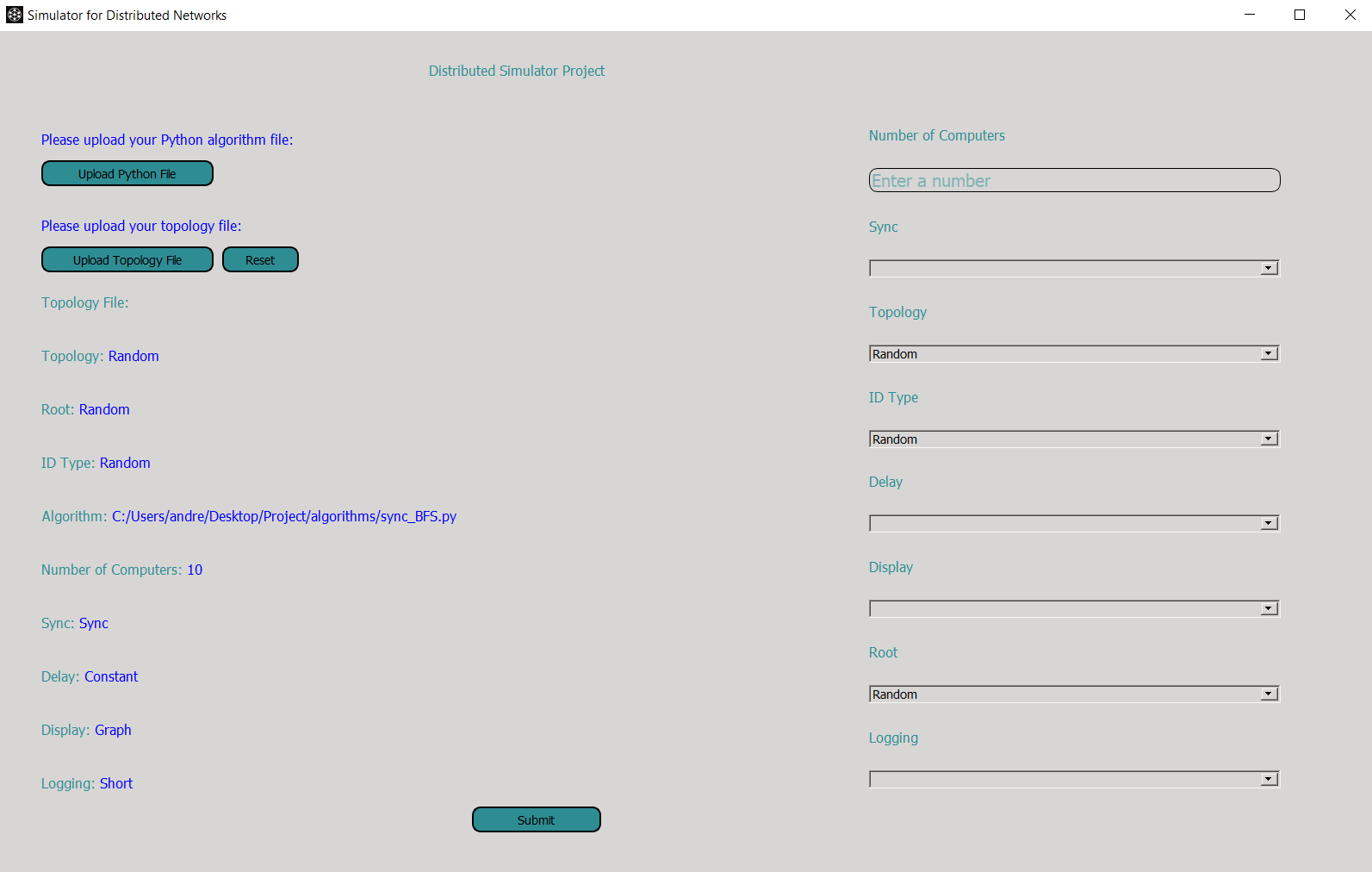
**C. Troubleshooting**

* **Missing or Incorrect Output:**
* Ensure all required fields are filled in the GUI.
* Check that your algorithm and topology files are correctly formatted.
* Review output.txt and the terminal for error messages.
* **Simulation Crashes or Freezes:**
* For very large networks, use "Text" display instead of "Graph."
* Check for errors in your custom algorithm code.

**FAQ: What are the blue settings on the left side of the main menu?**

**Question:**

What do the blue settings on the left represent (marked in a red rectangle)?



**Answer:** The blue settings on the leftside of the main menu are the saved configuration values from your  previous run of the simulator. When you launch the simulator, it automatically loads these settings as the default options for your next session. This means you do not have to reselect all the dropdown options or re-enter values each time you start the simulator—your last-used configuration is pre-filled for your convenience.

* **Where are these settings stored?**

They are saved in the  file in your project directory.

* **How does this help?**

This feature streamlines repeated experiments and makes it easy to tweak only the parameters you want to change, while keeping the rest of your previous setup intact.

* **Can I change them?**

Yes! You can modify any setting using the right-side controls. When you click "Submit," your new configuration will be saved and shown in blue the next time you open the simulator.