

Peer-to-Peer Network System: Comprehensive Report

Introduction

This Peer-to-Peer (P2P) network system is designed as a decentralised platform for secure, efficient, and fault-tolerant communication and resource sharing among nodes. Eliminating the need for centralised control, the system leverages direct interactions between network participants.

Each node is autonomous and capable of executing various essential functions, including transaction management, peer discovery, file transfer, and state synchronisation. The system is engineered using UDP for lightweight messaging and TCP for reliable file sharing, providing the balance of speed and dependability.

Advanced features such as Lamport clocks ensure event ordering, while failure simulations and heartbeat mechanisms enhance network resilience against faults and node failures. With its robust architecture, this P2P network is well-suited for applications such as decentralised ledgers, IoT ecosystems, and file-sharing networks, offering scalability and fault tolerance in dynamic environments.

Summary of Functions

1. Transaction Management

- Handles the creation, validation, and synchronisation of transactions among nodes.
- Ensures accurate balance management and conflict resolution using distributed data consistency mechanisms.

2. File Transfer

- Supports direct peer-to-peer file sharing through metadata negotiation and TCP connections.
- Fault-tolerant design enables retries and error handling for robust file delivery.

3. Peer Management

- Dynamically discovers, adds, and removes peers in the network.
- Maintains a real-time peer list and prevents duplicate or invalid peers.

4. Synchronisation:

- Employs transaction hashes and periodic updates to ensure consistency across all network nodes.
- Handles mismatched data states by merging transactions and balancing information.

5. Fault Tolerance

- Implements simulated message drops, failure detection, and node monitoring.
- Ensures network operation continues despite peer unavailability or message loss.

Introduction

This Peer-to-Peer (P2P) network system exemplifies a decentralised approach to secure communication and resource sharing in distributed environments.

Its design eliminates the dependence on central servers, empowering nodes to communicate directly and maintain autonomy. The system incorporates key features such as transaction management, file transfer, peer discovery, synchronisation, and fault tolerance to ensure reliability and scalability in diverse scenarios.

At its core, transaction management forms the backbone of the network. Nodes can create, validate, and process transactions, maintaining distributed balances and resolving conflicts seamlessly. By leveraging Lamport clocks, the system ensures that events are correctly ordered, even in asynchronous environments. Transactions are synchronised across peers using periodic updates and hash comparisons, guaranteeing consistency despite network delays or faults.

The file transfer functionality enables nodes to share resources directly through TCP connections. Metadata negotiation precedes the actual transfer, ensuring that both parties agree on file attributes such as name and size. Fault tolerance is integral to this process, with retries and error handling mechanisms in place to manage potential disruptions during the transfer.

Peer management is another vital component, facilitating the dynamic addition, discovery, and removal of peers. The system avoids duplicates, invalid nodes, or rejoining of previously removed peers. Nodes also exchange their peer lists to maintain a cohesive and updated network topology.

Synchronisation mechanisms maintain data consistency across all nodes. The system uses transaction hashes to detect discrepancies, merging transactions and balances as needed. This approach ensures that all nodes share a unified view of the network's state, even when peers join or leave dynamically.

Fault tolerance underpins the entire system. The network simulates message drops and incorporates failure detection mechanisms to handle real-world challenges. Heartbeat messages periodically verify the availability of peers, and unresponsive nodes are removed to maintain network health. Despite these challenges, the system remains resilient, continuing its operations even in the face of partial failures.

In practical applications, this P2P network system demonstrates significant versatility. It is ideal for decentralised accounts, where transaction consistency and fault tolerance are critical. In IoT networks, it can facilitate device-to-device communication and resource sharing without centralised control. Additionally, the file transfer functionality makes it well-suited for decentralised storage and sharing solutions.

Overall, the P2P network system combines modular design, robust fault tolerance, and efficient communication protocols to deliver a scalable and reliable platform for distributed applications. Its emphasis on decentralisation and autonomy ensures adaptability across various use cases, from blockchain technology to IoT ecosystems.

Each function definition break down

Transaction Handling

1. `process_transaction(txn)`:
 - Validates a transaction (e.g., sufficient funds, sender exists).
 - Updates local balances and stores the transaction.
 - Synchronises the transaction with peers.
2. `send_transaction(receiver_address, amount)`:
 - Creates and broadcasts a new transaction.
 - Deducts the amount from the sender's balance upon success.

3. `send_custom_transaction(receiver_address, amount, **custom_fields):`
 - Sends transactions with custom metadata.
 - Rejects reserved keys (e.g., id, timestamp) in custom fields.
4. `synchronize_transactions():`
 - Synchronises transactions across peers using transaction hashes.
 - Ensures consistency by comparing hashes and merging data if mismatched.
5. `merge_transactions(peer_transactions):`
 - Incorporates peer transactions, avoiding duplicates and validating transaction IDs.

Heartbeat and Failure Detection

1. `start_heartbeat(interval=5, timeout=15):`
 - Periodically pings peers to check their availability.
 - Removes unresponsive peers from the network.

Logging and Debugging

1. `log(log_type, message):`
 - Logs events with a specific type (e.g., "Error", "Transaction Processed").
2. `list_transactions():`
 - Logs all stored transactions in a formatted style.
3. `format_transactions():`
 - Returns a string representation of all transactions.

GUI (Integration with Tkinter)

1. `get_node_details():`
 - Opens a GUI window to display node details (e.g., balances, transactions).
2. `clear_local_data():`
 - Clears all local data through a GUI confirmation dialog.

How It Works at the beginning:

1. **Node Initialisation:**
 - A node is created with a nickname and an address (IP:PORT).
 - It initialises balances, peers, and transactions.
 - A UDP socket is bound to the address for communication.
2. **Joining a Network:**
 - Nodes discover peers using the `join_network` method.
 - The network topology grows dynamically as peers discover one another.
3. **Transaction Processing:**
 - Nodes create transactions and broadcast them to peers.
 - Transactions are validated and stored locally.
 - Balances are updated upon successful processing.
4. **File Sharing:**
 - File transfers are initiated via UDP and conducted over TCP.
 - Metadata (e.g., file name, size) is shared before transfer.
5. **Synchronisation:**

- Periodic sync ensures consistent balances and transactions.
- Hash comparison detects discrepancies and resolves them.

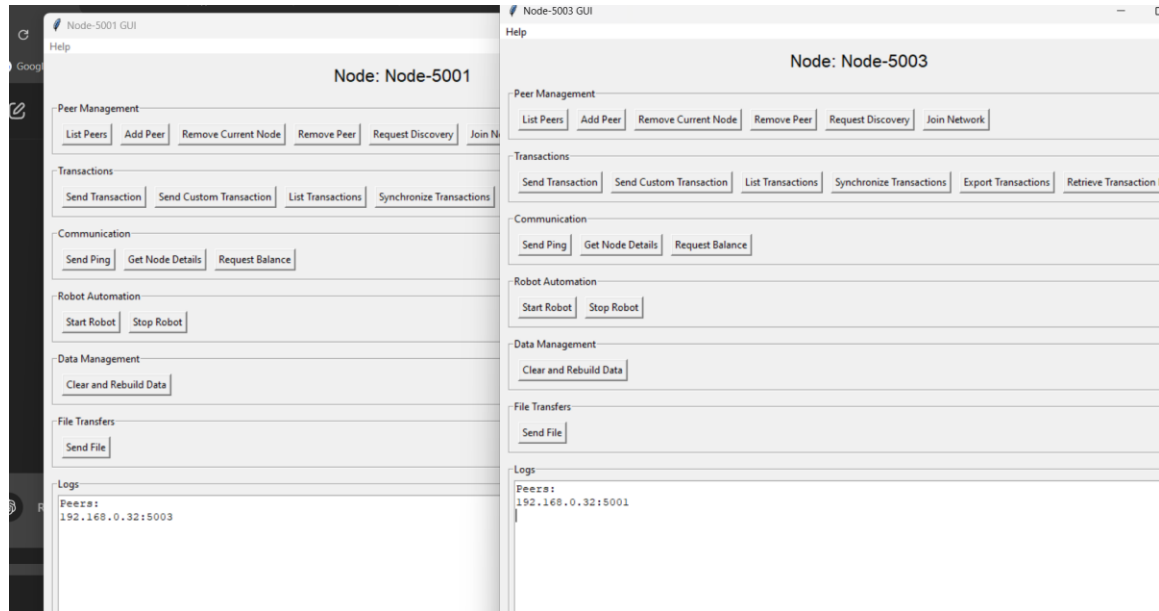
6. Fault Tolerance:

- Peers simulate message drops to test resilience.
- Heartbeats detect and remove inactive peers.

7. Logging and Debugging:

- Logs are maintained for key events (e.g., transaction processing, file transfers).

Pinging a node

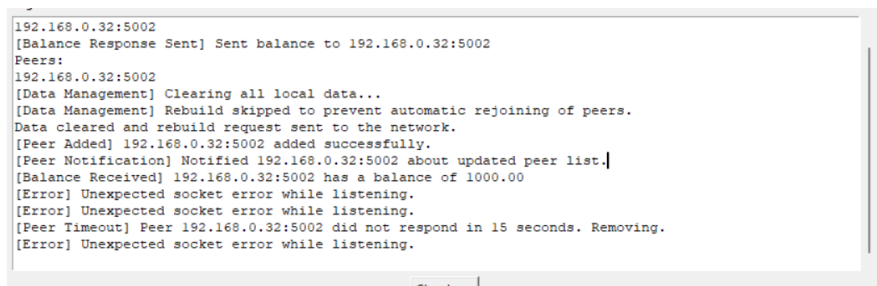


Nodes send a ping and silent pong response constantly between each other via heartbeat feature – this is done due to keep constant communication between peer nodes. If the pong response is not acknowledged whatsoever, the sending port will wait a few seconds and disconnect if it continues to fail to connect to the peer node again.

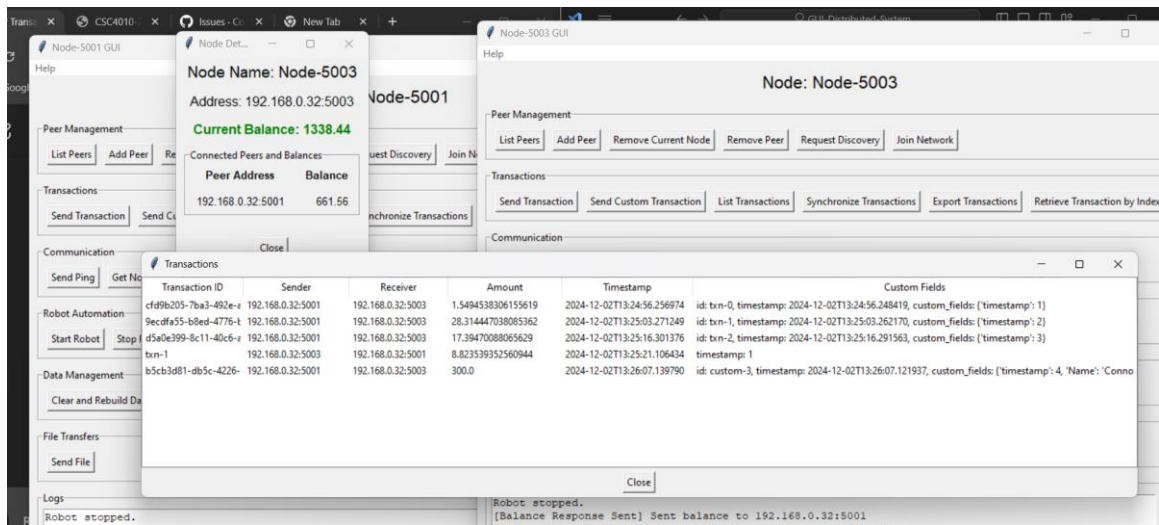
– this silent pong has implemented to maintain the clean logs as shown – or else the logs would be constantly overflowed with ping – pong responses. I believed it best to maintain this UDP mess

Automatic removal of a node

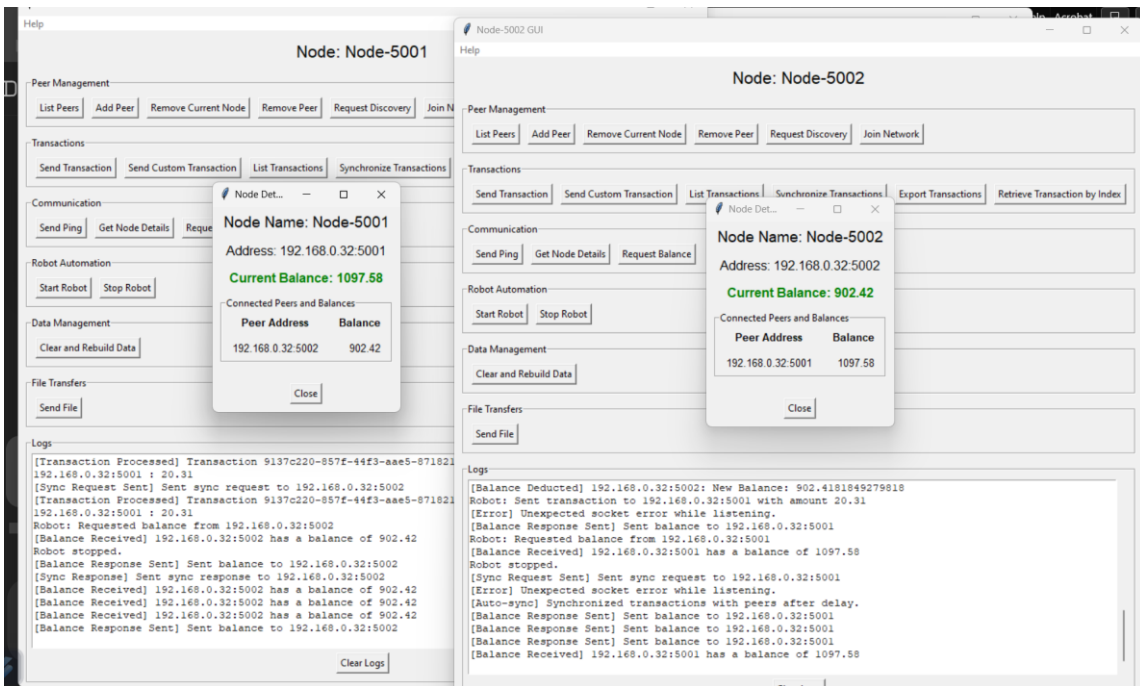
A node will automatically remove after some time a “Pong” response has not been received.



Sending a transaction



The transactions sync occurs which takes a few seconds for all the nodes to be interlinked together – We can return the details as shown through the get node details button shown on the GUI – this returns the given transactions it interacts with, via sender – receiver with the currency amount, timestamp and the custom fields – if a custom transaction has been processed.

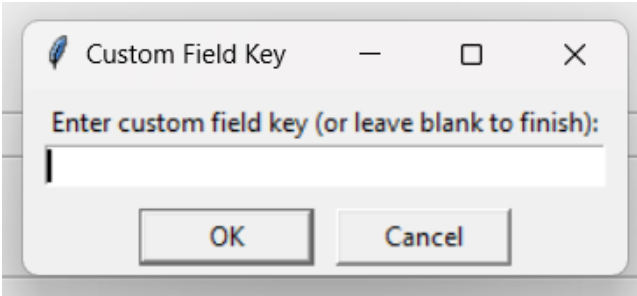


When sending transactions to one another – the peer nodes will wait a few seconds to “sync” together and we can see the balances of each node and their corresponding peer balances. As shown we can see the two nodes connected with their respective balances – acting as two independent nodes.

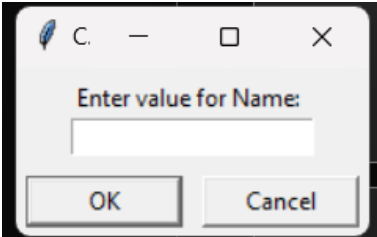
Sending a custom transaction.

Operating just as similar to send transactions, however the custom send transaction contains additional field key headers and a specific instance of the field key.

In this case, the field key is “Name” – and a name instance of “Connor” we can click Ok upon the next window and check the get node details window – to check the current transactions made.

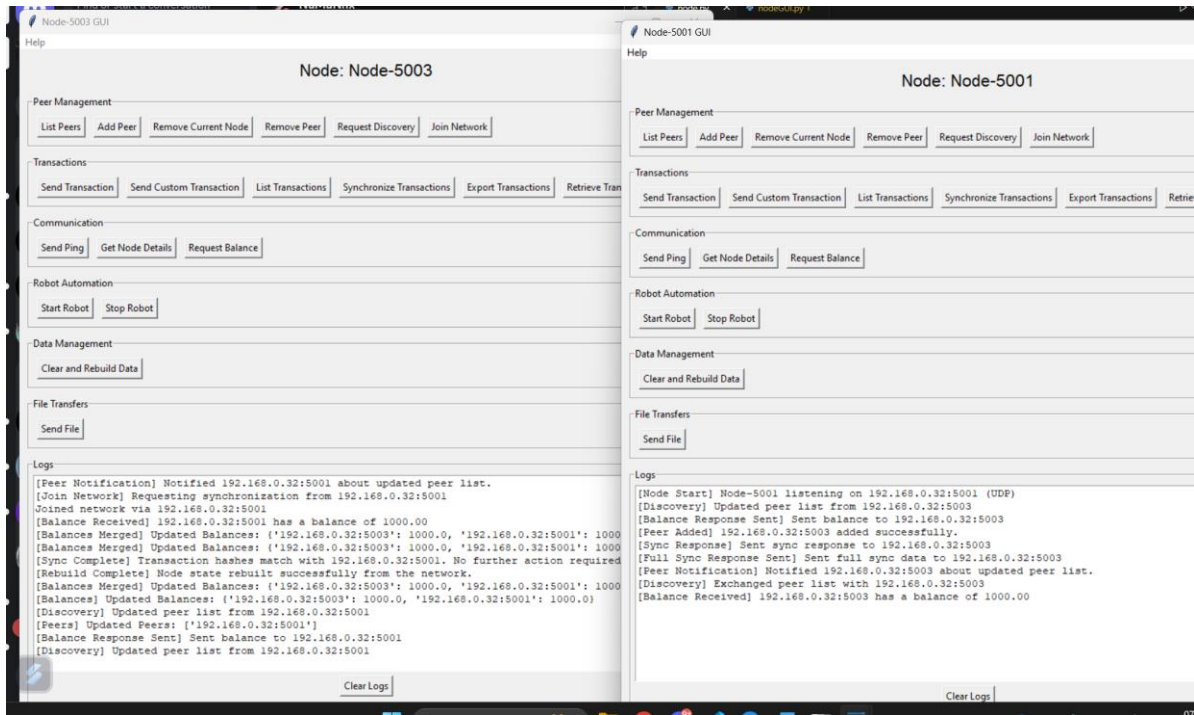


Shown below is the custom transaction custom-1 – with the amount of 200, and the custom field “Name – Connor” as described earlier, we can see this transaction is working as intended.



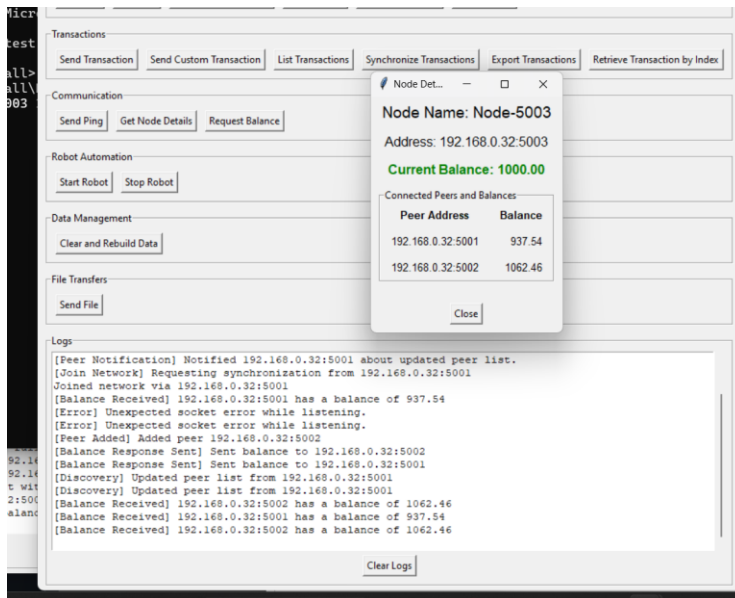
Transactions						
Transaction ID	Sender	Receiver	Amount	Timestamp	Custom Fields	
19b5609f-d037-47b7-8282-991b50a8	192.168.0.32:5002	192.168.0.32:5001	37.629667646683366	2024-12-02T14:03:47.846324	id: txn-0, timestamp: 2024-12-02T14:03:47.841224, custom_fields: {'timestamp': 1}	
13bb076f-72cb-4550-a255-2c32b693	192.168.0.32:5002	192.168.0.32:5001	39.63945528099064	2024-12-02T14:03:54.848015	id: txn-1, timestamp: 2024-12-02T14:03:54.843689, custom_fields: {'timestamp': 2}	
9137c220-857f-44f3-aae5-871821bb1	192.168.0.32:5002	192.168.0.32:5001	20.312692144344222	2024-12-02T14:04:00.856324	id: txn-2, timestamp: 2024-12-02T14:04:00.852815, custom_fields: {'timestamp': 3}	
custom-1	192.168.0.32:5001	192.168.0.32:5002	200.0	2024-12-02T14:09:33.659852	timestamp: 1, Name: Connor	

Join Network



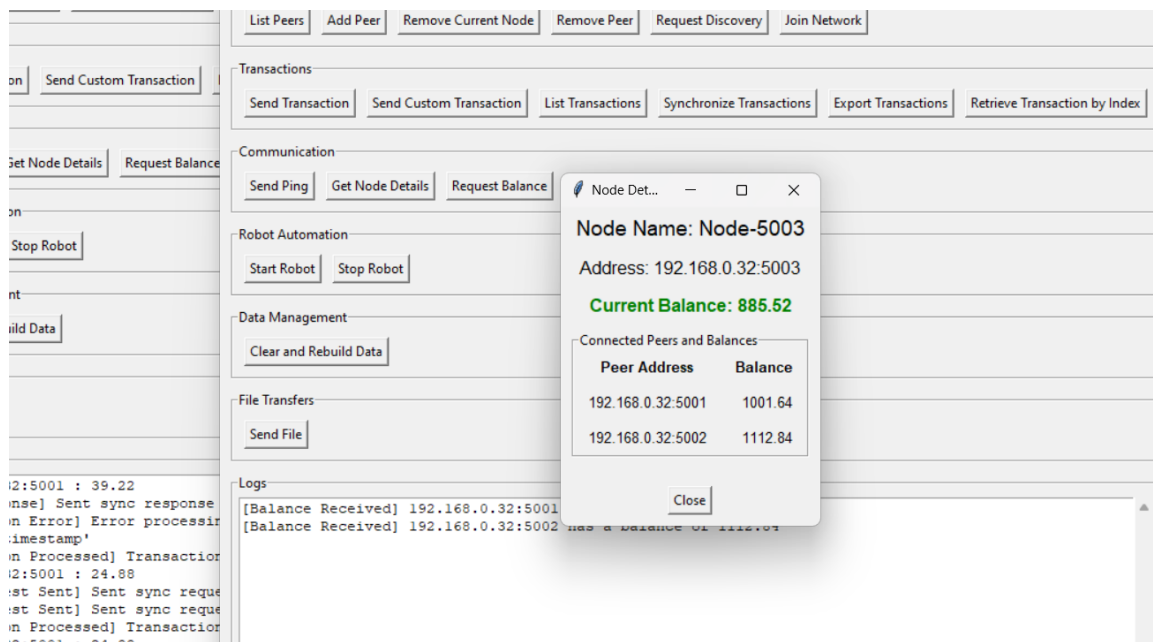
A node joining via network

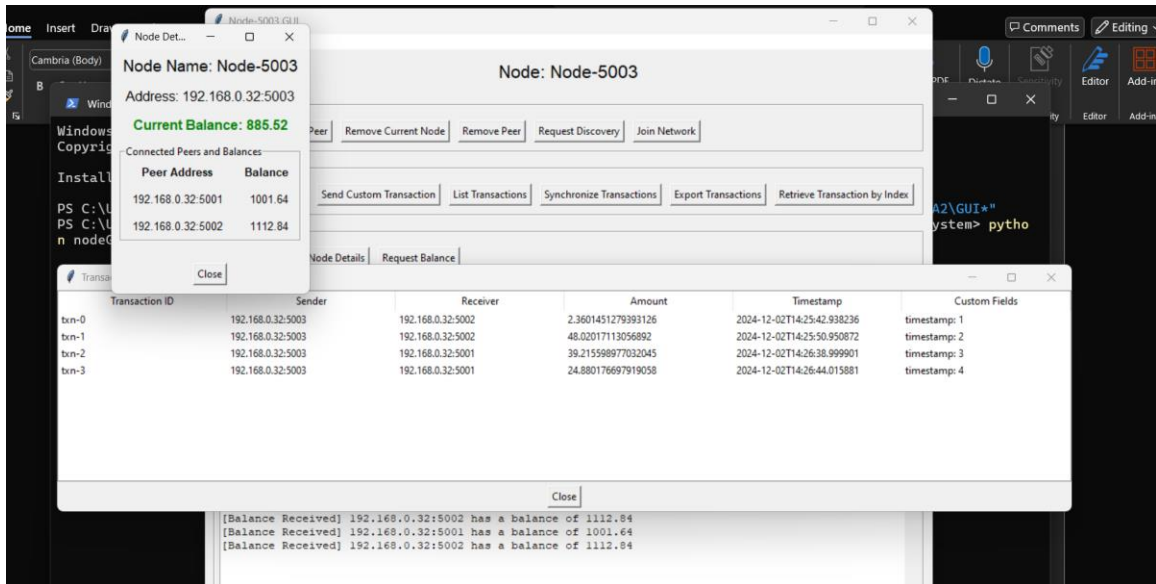
The current node will connect to a peer node network and connect to other peer nodes and balances once connected.



When two nodes have made a few transactions and a new node joins the network – it will connect to the other peer nodes and return the corresponding balances as shown here. Where Node-5003 is able to make transactions with Node-5001 and Node-5002. This allows 5003 to continue sending transactions to the peer nodes.

Just as shown – 5003 is sending transactions and updating the balances accordingly,

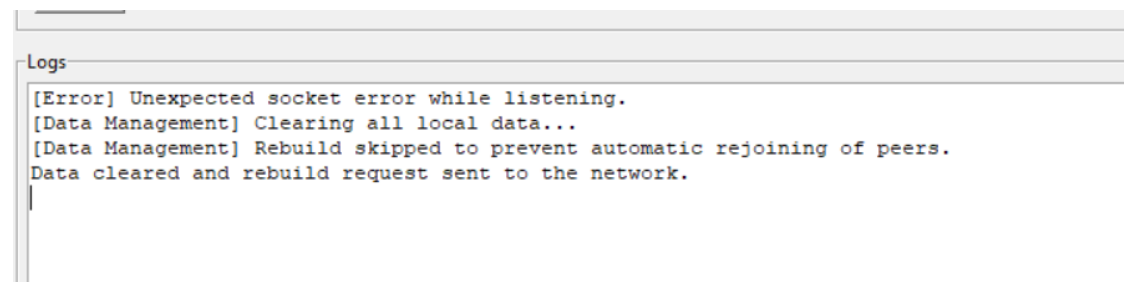
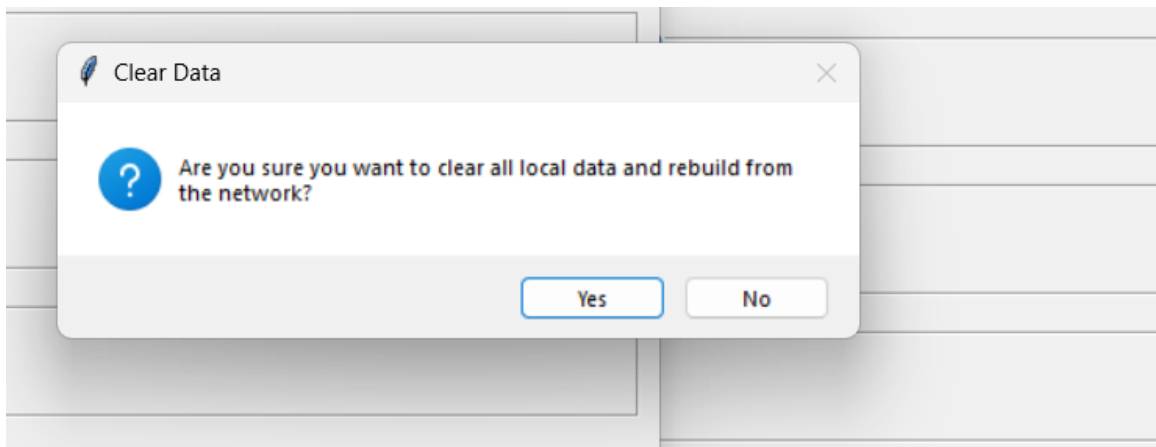




We can see the sending transactions between Node-5003 to Node-5001 and Node-5002

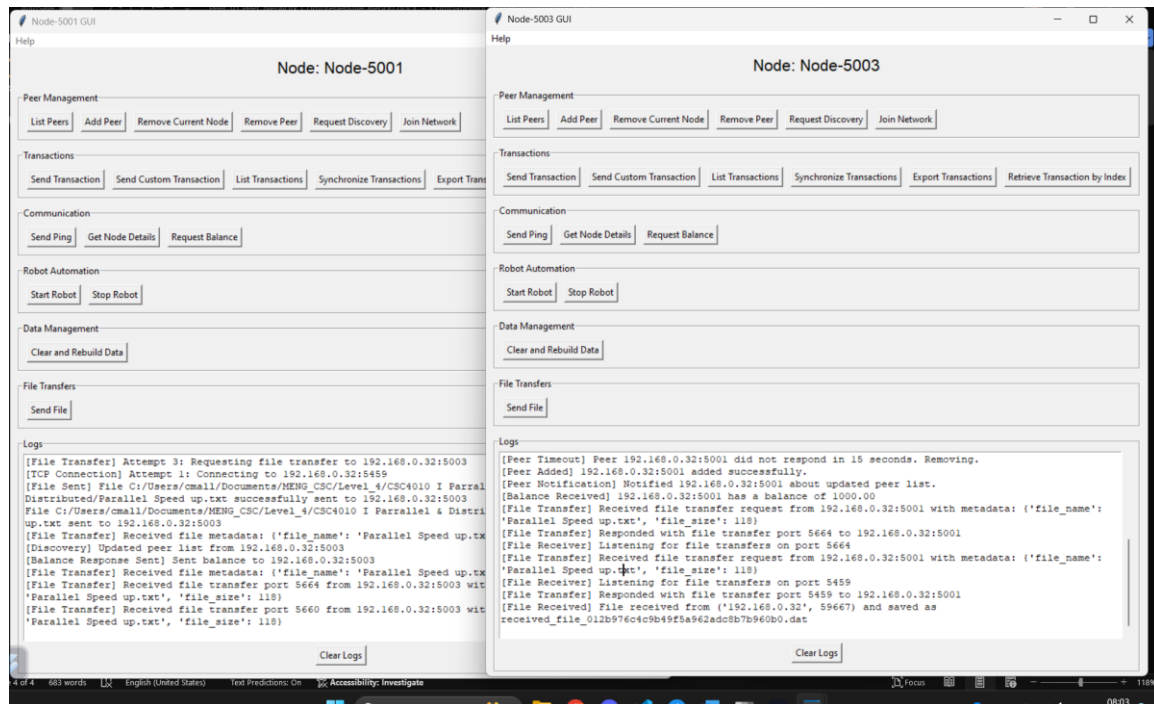
Take note that there is a waiting period for transactions to synchronise together – we can alternatively click the “synchronize Transactions” button.

Clear and Rebuild Rebuild Data



The node will disconnect, reset balances, transactions and all saved information. Once done, the peer nodes will automatically reconnect to the current node and continue operating as normal again.



Send File



The send file features brings up a window to send a new file to the peer node, The file cannot be any more than 4096 bytes or 4KB – this limitation was to send small .txt based file examples to one node to another.

The output should return a DAT file as shown – which effectively returns somewhat like a .txt file.

The receiving node shows the [File Received] Log and we can see the name it is saved under.

	received_file_012b976c4c9b49f5a962a...	02/12/2024 08:01	DAT File	1 KB
	received_file_316c456010554848a977...	02/12/2024 13:13	DAT File	1 KB

The file transfer process is built on a combination of UDP for discovery and TCP for reliability. Below is an overview of the workflow:

1. Metadata Exchange:

- Before sending the file, the sender transmits metadata (file name, size) to the receiver over UDP.
- This allows the receiver to prepare for the file transfer and validate that the incoming file meets predefined constraints (e.g., 4KB size limit).

2. File Transmission:

- The file is transferred via TCP to ensure reliable delivery.
- The system logs each stage of the process, including connection establishment, file transmission, and receipt acknowledgment.

3. Post-Transfer Handling:

- At the receiving node, the file is saved under a unique name (received_file_<hash>.dat) to prevent overwriting existing files.
- The receiving node logs the event with details such as file name, size, and timestamp.

Error Handling and Fault Tolerance

The system incorporates several mechanisms to ensure robust file transfers:

- Retries: If the transfer fails due to network issues, the system logs the failure and retries the process up to a specified limit.
- Timeouts: A timeout mechanism ensures that unresponsive nodes are identified and the transfer is terminated gracefully.
- Clear Logs: Users can clear logs to declutter the interface after addressing any issues.

Example Workflow

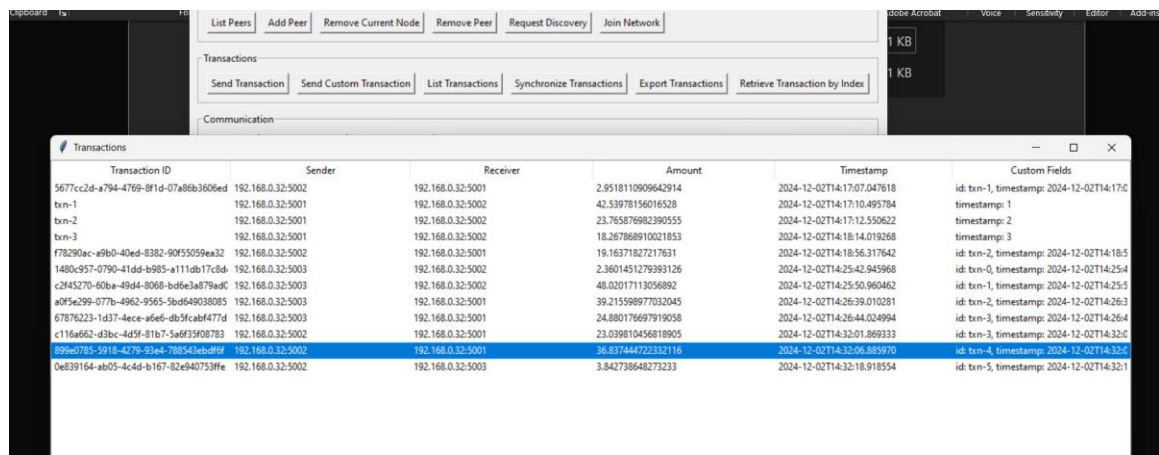
1. Sender's Actions:

- A user on Node-5001 selects a file using the GUI's "Send File" button.
- The system prompts the user to enter the destination node's address (e.g., Node-5003).
- After confirming the details, the system sends metadata and transfers the file.

2. Receiver's Actions:

- Node-5003 acknowledges the metadata and starts receiving the file via TCP.
- Upon successful receipt, Node-5003 logs the event, saves the file locally, and updates its GUI.

Export transactions – Additional Wow feature added



We can export these transactions to a .csv file for this example – we are using Node-5001, it will be independent for each Node , due to their independent nature.

We can click the Export transactions, which brings up a window to store the transactions – under name:

“Node-5001-
transactions.csv”

We get this
response.

ID	Sender	Receiver	Amount	Timestamp
5677cc2d-a794-4769-bf1d-07a86b3606ed	192.168.0.32:5002	192.168.0.32:5001	2.951811091	2024-12-02T14:17:07.047618
txn-1	192.168.0.32:5001	192.168.0.32:5002	42.53978156	2024-12-02T14:17:10.495784
txn-2	192.168.0.32:5001	192.168.0.32:5002	23.76587698	2024-12-02T14:17:12.550622
txn-3	192.168.0.32:5001	192.168.0.32:5002	18.26786891	2024-12-02T14:18:14.019268
f78290ac-a9b0-40ed-8382-90f55059ea32	192.168.0.32:5002	192.168.0.32:5001	19.16371827	2024-12-02T14:18:56.317642
1480c957-0790-41dd-b985-a111db17c8dc	192.168.0.32:5003	192.168.0.32:5002	2.360145128	2024-12-02T14:25:42.945968
c2f45270-60ba-49d4-8068-bd6e3a879ad0	192.168.0.32:5003	192.168.0.32:5002	48.02017113	2024-12-02T14:25:50.960462
a0f5e299-077b-4962-9565-5bd649038085	192.168.0.32:5003	192.168.0.32:5001	39.21559898	2024-12-02T14:26:39.010281
67876223-1d37-4ece-a6e6-dbf5cabf477d	192.168.0.32:5003	192.168.0.32:5001	24.8801767	2024-12-02T14:26:44.024994
c116a662-d3bc-4d5f-81b7-5a6f35f08783	192.168.0.32:5002	192.168.0.32:5001	23.03981046	2024-12-02T14:32:01.869333
899e0785-5918-4279-93e4-788543ebdf6f	192.168.0.32:5002	192.168.0.32:5001	36.83744472	2024-12-02T14:32:06.885970
0e839164-ab05-4c4d-b167-82e940753ffe	192.168.0.32:5002	192.168.0.32:5003	3.842738648	2024-12-02T14:32:18.918554

```
[Balance Received] 192.168.0.32:5003 has a balance of 889.37
Data exported to C:/Users/cmall/Documents/MENG_CSC/Level_4/CSC4010 I Parrallel &
Distributed/40295919_A2/GUI-Distributed-System/Node-5001-transactions.csv
```

We can see the following transactions saved as a .csv here.

Retrieve transactions by index

We can continue to use Node-5001 for this instance, with the following transactions shown above.

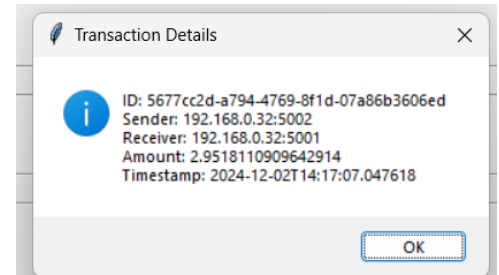
The transactions are 0 indexed – so the first instance begins 0, where the last = N - 1

By clicking this button we are prompted by the “Enter transaction index” by inputting 0, the first index. We are returned with this window:

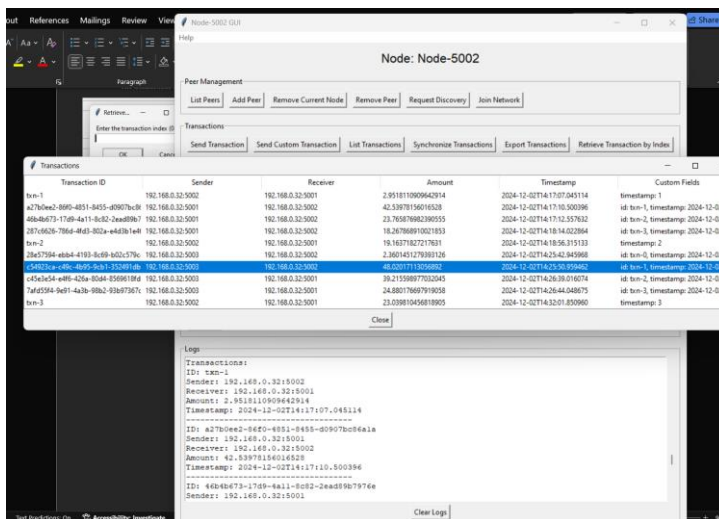
Returning the first instance transaction created,

With following ID, sender, receiver, amount and timestamp.

This would be the same for the second instance at [1], third instance at [2] and so forth onwards.



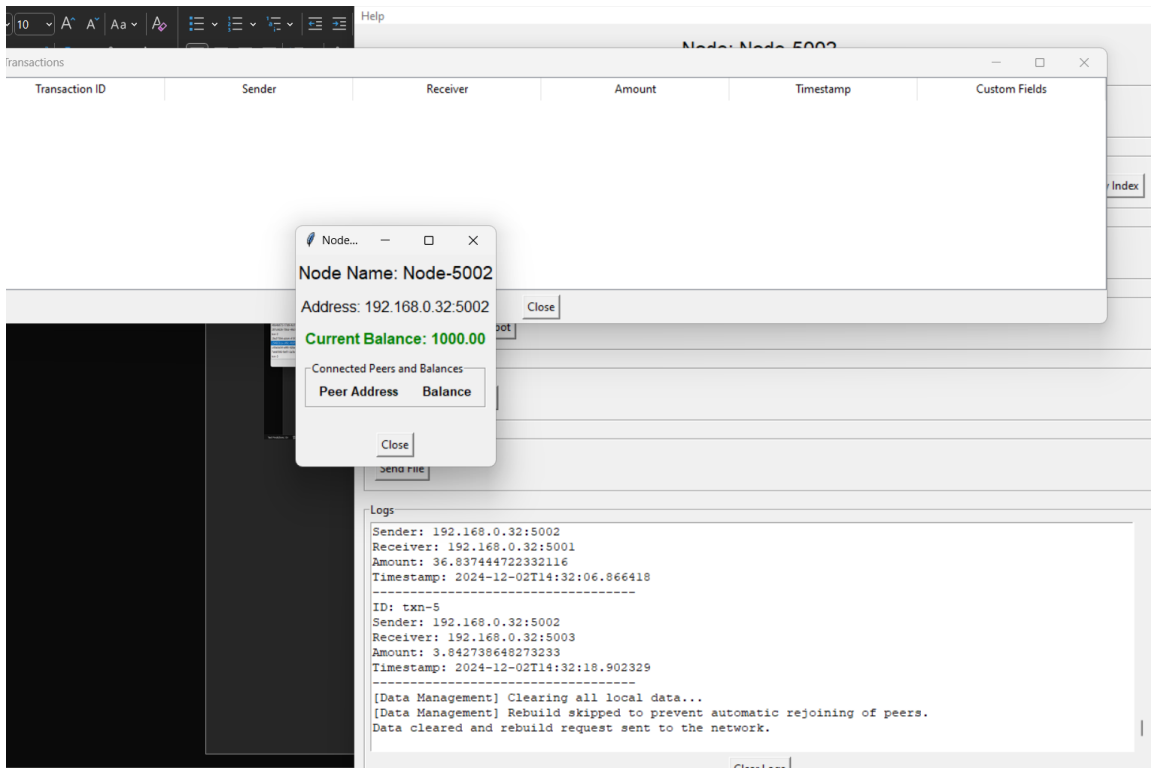
List Transactions



By listing the transactions of a specific node – we are promoted by the Transactions window , provided by Tkinter, as well as a display log as shown below, displaying the ID, Sender, Receiver, Amount and Timestamp, just as shown in the window -

I have decided to show both as it gives two different representations of the transactions for easier viewing.

Clear and rebuild instance



By clicking the clear and rebuild button in the window – we can see the transactions being reset and the balance additionally being reset – rebuilding the node as if it were just initialised once again.

We can start afresh here and begin adding new nodes once again.

Conclusion

The Node.py code is the backbone of the P2P network system, featuring several innovative mechanisms that enhance its functionality and resilience.

Advanced Peer Management

- **Dynamic Peer Discovery:** Nodes can discover and connect to peers dynamically, maintaining an updated list of peers and their statuses.
- **Removal of Peers:** Nodes can remove peers autonomously or based on user input. The `remove_peer()` function ensures a seamless disconnection and notifies other peers of the updated network topology.
- **Heartbeat Monitoring:** Periodic heartbeat messages verify the availability of peers. Unresponsive nodes are removed from the network, ensuring stability.

Fault Tolerance and Resilience

- **Failure Simulation:** The system can simulate message drops to test its robustness. By introducing artificial packet loss, developers can evaluate the network's ability to handle real-world issues like connectivity interruptions.
- **Node Reconnection:** Disconnected nodes can seamlessly reconnect to the network, resynchronising their state with peers.
- **Transaction Hashing:** Hash comparisons are used to detect discrepancies in the transaction logs, ensuring consistency across nodes.

Transaction Management

- **Custom Transactions:** Nodes support custom metadata in transactions, allowing users to include additional fields like "Name" or "Description." These custom fields enhance the versatility of transactions for specific use cases.
- **Synchronisation:** The `synchronize_transactions()` method ensures that all nodes in the network share the same transaction log, preventing data inconsistencies.
- **Validation and Processing:** Transactions are validated for sufficient balances, and duplicate or invalid transactions are ignored, ensuring data integrity.

File Transfer Capabilities

- **Direct File Sharing:** Nodes can transfer files using metadata negotiation to ensure compatibility. The actual transfer occurs over TCP for reliability.
- **Error Handling:** Robust mechanisms, such as retries and port selection, ensure successful file transfers even under challenging network conditions.

NodeGUI.py implementation

The NodeGUI.py extends the functionality of the core node by integrating a graphical user interface (GUI) for user interaction. It introduces user-friendly features that enhance the usability and monitoring of the P2P network:

Peer and Node Management

- **Interactive Peer Addition/Removal:** Users can add or remove peers directly through the GUI, simplifying network management.
- **Dynamic Network Joining:** Nodes can join existing networks by connecting to a single peer, which facilitates rapid scaling.
- The `clear_and_rebuild_data()` method allows users to reset a node's state and rebuild data from the network. This feature ensures data consistency after significant changes, such as peer removal or state corruption.

Transaction Monitoring and Management

- **Detailed Transaction Viewing:** Transactions are displayed in a structured format with details such as sender, receiver, amount, timestamp, and custom fields.
- **Custom Transaction Creation:** Users can input additional metadata for transactions, which is particularly useful in specialised use cases like supply chain tracking.
- **Transaction Export:** Transactions can be exported to CSV format, providing a snapshot of network activity for external analysis.

Automation with Robot Tasks

- **Automated Actions:** The integrated robot feature performs automated tasks like sending transactions, pings, and balance requests. This is particularly useful for stress testing and performance analysis.

- The GUI includes a "robot" feature for automated testing. The robot performs random actions, such as sending transactions, pings, and balance requests, at regular intervals. This functionality is useful for stress-testing the network.

File Transfers

- **File Sending GUI:** Users can select files to send to peers through a simple dialog box, streamlining the file transfer process.
- **File Reception Logs:** Received files are logged, providing transparency and auditability.
- The `send_file_gui()` method in the GUI allows users to select a file and send it to a peer. The backend `Node.py` handles Metadata negotiation: Ensuring both sender and receiver agree on file attributes, using TCP transfer: A Reliable delivery of the file.

Enhanced Synchronization

- **Manual Synchronisation:** Users can manually trigger synchronization of transactions and balances, ensuring real-time consistency across peers.
- **Rebuild from Network:** Nodes can reset their state and rebuild data from the network, ensuring continuity even after disruptions.

Expanded Technical Details

Logging and Debugging

The system includes comprehensive logging for debugging and monitoring:

- **Event Logs:** Each action, such as transaction processing or file reception, is logged with its type and details.
- **Silent Operations:** Pings and responses are logged silently to avoid clutter, preserving meaningful log entries.

Advanced Algorithms

- **Lamport Clocks:** Used for logical event ordering, ensuring that the sequence of operations is consistent across nodes.
- **Transaction Hashing:** Ensures that the integrity of transaction logs is maintained across the network.

GUI Setup and Initialization

The NodeGUI.py class initialises a graphical interface for managing P2P nodes. The GUI is built using Tkinter and comprises multiple sections for peer management, transactions, communication, file transfers, and logging. Upon initialisation, a Node object from Node.py is instantiated to handle backend logic, including UDP/TCP communication, transactions, and state synchronisation.

NodeGUI functions explained

Core GUI Setup

- `setup_gui()`: Initializes the GUI layout, including buttons, frames, and log display. Organizes components into categories like Peer Management, Transactions, Communication, Robot Automation, Data Management, and File Transfers.

Peer Management Functions

1. `list_peers()`:
 - Displays a list of all connected peers.
 - Logs the peer list for the user to review.
2. `add_peer()`:
 - Prompts the user to input a peer's address (IP:PORT) and adds it to the network.
 - Updates the peer list and requests discovery.
3. `remove_peer()`:
 - Removes a specific peer from the network.
 - Notifies other peers of the updated network topology.
4. `remove_current_node()`:
 - Removes the current node from the network and closes the GUI.
5. `request_discovery()`:
 - Sends a discovery request to specified IPs or peers.
 - Ensures dynamic network updates.
6. `join_network()`:
 - Allows a node to join an existing network by connecting to a peer node.
 - Requests synchronization and updates the local state.

Transaction Functions

1. `send_transaction()`:
 - Prompts the user to input a recipient's address and amount.
 - Sends a transaction to the specified peer, updates balances, and logs the event.
2. `send_custom_transaction_gui()`:
 - Allows users to add custom metadata to a transaction.
 - Users specify custom fields, which are sent alongside the transaction.
3. `list_transactions()`:
 - Displays all transactions recorded on the node.
 - Logs transactions in a detailed, formatted view.
4. `synchronize_transactions()`:
 - Synchronizes transactions with connected peers.
 - Ensures consistency across the network.
5. `retrieve_transaction_by_index()`:
 - Retrieves a specific transaction by its index in the transaction list.
 - Displays the transaction details in a separate window.
6. `export_data()`:
 - Allows the user to export all transactions to a .csv file.
 - Useful for external analysis or record-keeping.

File Transfer Functions

1. `send_file_gui()`:
 - Enables the user to select a file and specify a recipient node.
 - Initiates the file transfer process and logs the outcome.
2. `send_transaction_with_file()`:
 - Combines a transaction with a file transfer.
 - Links the file to the transaction and sends both to the specified peer.

Communication Functions

1. `send_ping()`:
 - Sends a ping to a specific peer to check connectivity.
 - Useful for monitoring network health.
2. `get_node_details()`:
 - Opens a detailed view of the current node's state.
 - Displays balances, connected peers, and transaction history.
3. `request_balance()`:
 - Requests the balance of a specific peer.
 - Displays the received balance in the logs.

Robot Automation Functions

1. `start_robot()`:
 - Starts an automated process (robot) that randomly performs tasks like sending transactions or pings.
 - Simulates node behavior in a dynamic network.
2. `stop_robot()`:
 - Stops the automated process.
 - Halts all robot-driven actions immediately.

Data Management Functions

1. `clear_and_rebuild_data()`:
 - Resets all local data (transactions, peers, and balances).
 - Optionally rebuilds the state from connected peers.

Log Management

1. `log_message()`:
 - Adds messages to the log area in the GUI.
 - Ensures real-time visibility of node events and errors.
2. `clear_logs()`:
 - Clears all logs displayed in the log area.

Help Menu

1. `show_help()`:
 - Provides a detailed overview of all available features and their usage.
 - Accessible via the "Help" menu in the GUI.