Distributed Fault-Tolerant Chat Project

CHOUKROUN Simon

July 23, 2023

Contents

1 Introduction

The purpose of this report is to outline the implementation details and architecture of a decentralized P2P messaging system that utilizes the User Datagram Protocol (UDP) for communication. The system incorporates an authentication server, existing nodes, and various protocols to achieve secure and efficient messaging capabilities.

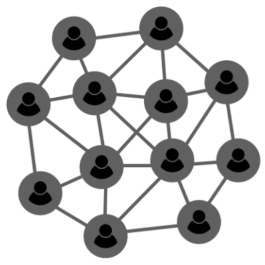
2 What is Peer-to-Peer Messaging

In peer-to-peer messaging, participants exchange messages directly between their devices without the aid of a centralized server or middleman. Each member in a peer-to-peer messaging system serves both as a client and a server, allowing for direct communication between them.

Peer-to-peer messaging enables direct communication between devices, in contrast to conventional client-server architectures where messages are routed through a central server. This decentralized strategy can provide advantages including enhanced scalability, decreased reliance on a single point of failure, and increased privacy.

Each participant's device in a peer-to-peer messaging system has the ability to create a direct link with other devices using the network.

Peer-to-peer messaging allows users to communicate directly with one another without the use of a centralized server, potentially providing benefits in terms of privacy, fault tolerance, and scalability.



3 Message Transmission

In P2P networks, there are multiple message transmission algorithms that help with secure and efficient transfer of events. Here are some common ones :

▶ **Flooding Algorithm :** In this technique, a node broadcasts the message to all its connected peers and then they broadcast the same message again if they haven't seen it before. This goes on until the message reaches the required host. Although easy to implement, this approach can lead to excessive load on the network due to multiple re-broadcasts of the same message. It can be optimized by integrating Time-to-live or duplicate message suppression.

▶ **Gossip Protocol :** Similar in ways to flooding, the only difference being that the receiver node will broadcast the message to a subset of its peers. The node selects these peers at random. Overtime the message will spread throughout the network as nodes broadcast the message randomly. Hence, creating a more optimized way for message transmission.

► **DHTs :** DHT helps in indexing and locating a resource in the p2p environment without the need of a central routing server. Nodes can find the location of other nodes by querying the DHT, making it useful for peer discovery in P2P systems.

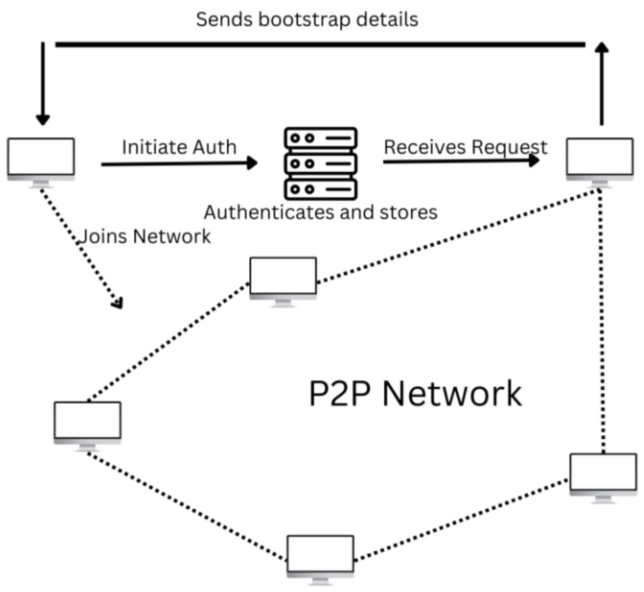
4 Message Ordering Causality

In a distributed P2P chat application, ensuring correct ordering of messages among multiple nodes is essential to maintain a coherent conversation. Here are some common techniques :

► **Vector Clocks :** Each node maintains its own vector clock. And it is updated whenever it sends or receives new messages. When a node sends a message it includes its own vector clock in it. Upon receiving a vector clock, the nodes compare it with their own clocks to determine the order of events.

► **Lamport Timestamps :** Lamport Timestamps are maintained by logical clocks that can be used to establish the ordering of events. Whenever a node sends a message, it sends its clock as a timestamp with it. The ordering of events is based on these timestamps.

5 Flow Diagram



6 Authentication Server

The authentication server acts as a central component for managing signup and sign in requests. It stores all the credentials of different users and restricts nodes from connecting if the number of nodes that are alive in the network exceeds a set limit.

Furthermore, it verifies any new connection requests and stores the connection request in a queue, awaiting connection to the P2P network via bootstrap nodes.

7 Existing Nodes and Communication

Each existing node functions as a separate entity responsible for handling incoming connection requests and facilitating network connectivity. The nodes communicate with each other and the authentication server using UDP, ensuring lightweight and efficient communication without the need for sticky sessions.

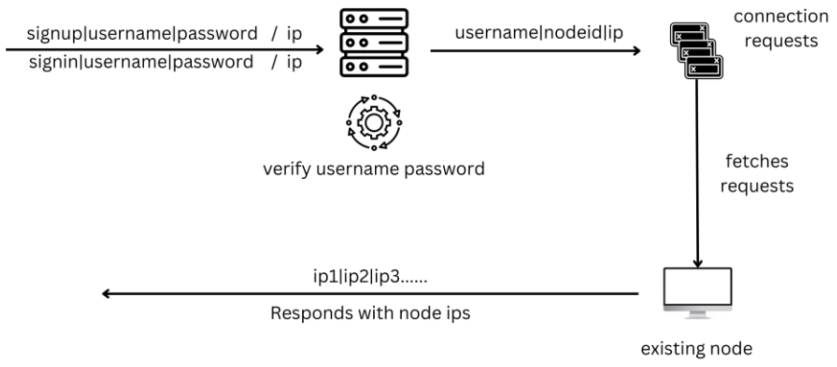
8 Authentication Process

When a new node seeks to join the network, it sends its username, password, and a signup or signin flagto the authentication server over UDP.

In the case of signup, the server stores the node Id and password in its local storage. Then pushes thenode details to a queue awaiting connection to the network.

In the case of sign in, the server will query its local storage and verify the password. Then push the nodedetails to the same queue awaiting connection to the network.

An existing node will ping the server incrementally for requests that are awaiting connection and sendthem the list of addresses of nodes that are in the network. Hence, acting as a bootstrap node.



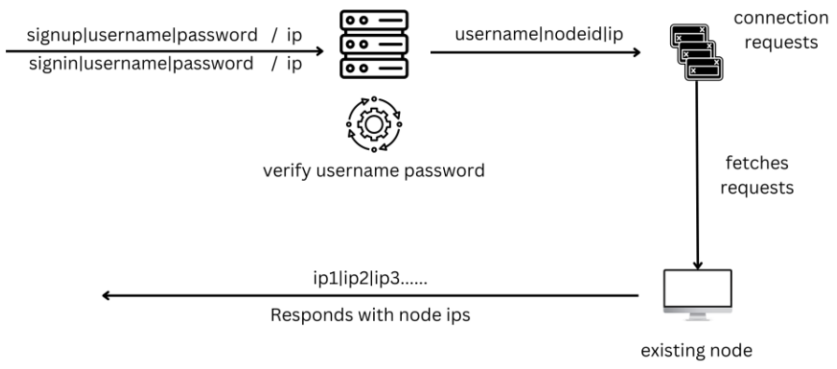
9 Network Connectivity and Discovery

After successful authentication, the new node broadcasts a message to all other nodes in the network using UDP multicast. This message notifies the network about the addition of the new node and triggersupdates in each node's local storage. Nodes update their internal data structures to reflect the current network topology, ensuring accurate routing and connectivity.

10 Messaging Features and Chat History

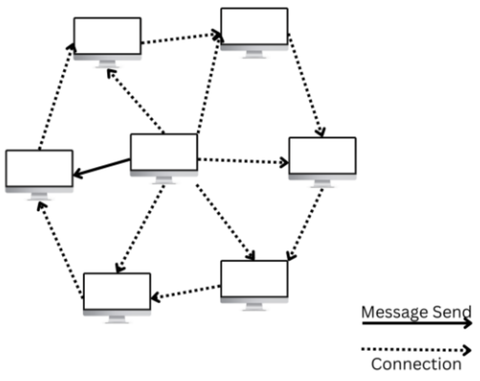
10.1 Broadcast Messaging

We will be using Gossip Protocol to send broadcast messages. In a system with requirements that are relatively small, we don't need congesting algorithms like a flooding algorithm as well as a complex algorithm like DHT. Gossip Protocol seems like a good fit.



10.2 Private Messaging

For private messaging, we are utilizing direct routing. In direct routing, a sender node knows the specific address for the recipient node. The sender uses the specific address to send the message directly to thereceiver with no transit nodes.
This approach is efficient for one-to-one communication and ideal for situations where both nodes know each other's addresses. Such is the case for our project.



10.3 Message Ordering

To maintain message order and causality tracking, vector clocks are employed. Each node includes its vector clock with every message it sends. A vector clock captures the local knowledge of event ordering for each node and enables accurate sequencing and identification of causal relationships between messages. This ensures consistency and integrity within the messaging system. The size of the vector clock is typically equal to the number of nodes in the network.

10.3.1 Message Sending

When a node sends a message to another node, it includes its own vector clock along with the message. The vector clock represents the node's knowledge of events up until that point in time.

10.3.2 Vector Clock Update

Upon receiving a message, the receiving node compares its own vector clock with the vector clock included in the received message. It then updates its vector clock based on the comparison.

10.3.3 Ordering Determination

To determine the order of two messages, the following rules are applied :
If the vector clock of message A is causally before the vector clock of message B (i.e., all counters in A are less than or equal to the corresponding counters in B), message A is considered to have occurred before message B.

If the vector clock of message B is causally before the vector clock of message A, message B is considered to have occurred before message A.

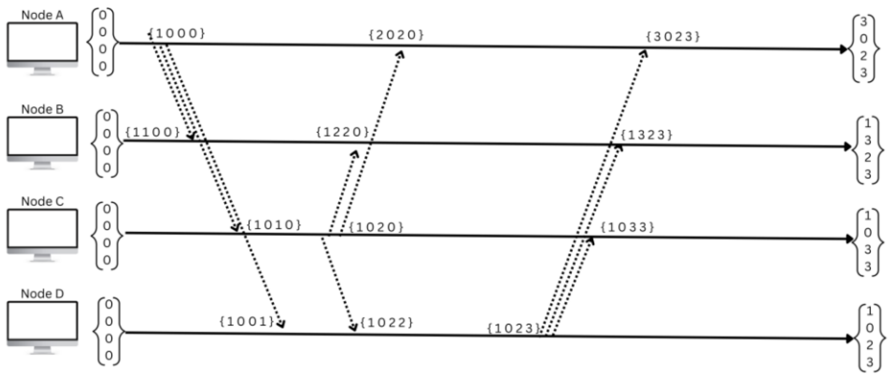
If there is no causality between the vector clocks (i.e., the counters in the vector clocks are incomparable), the messages are considered concurrent.

10.3.4 Maintaining Vector Clocks

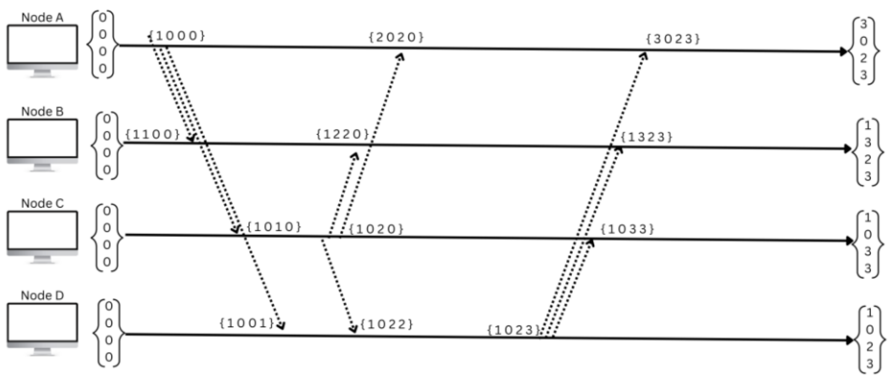
Each node updates its vector clock as it sends and receives messages. Whenever a node sends a message, it increments its own counter in the vector clock. When a node receives a message, it updates its vector clock by taking the maximum value for each counter from its own vector clock and the received vector clock and increments its own counter for the corresponding node.

10.3.5 Examples

Vector Clocks in group chat :



Vector Clocks in peer-to-peer communication :



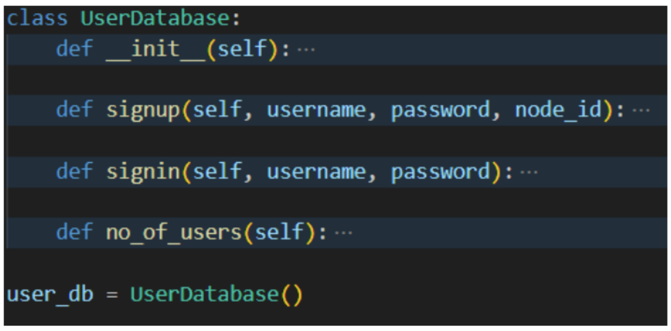
Message Retrieval : In both scenarios, the messages are ordered such that messages with the smaller vector clocks are retrieved first.

11 Code

We have in total 4 classes hosted in this project.

11.1 UserDatabase

It has the following properties :

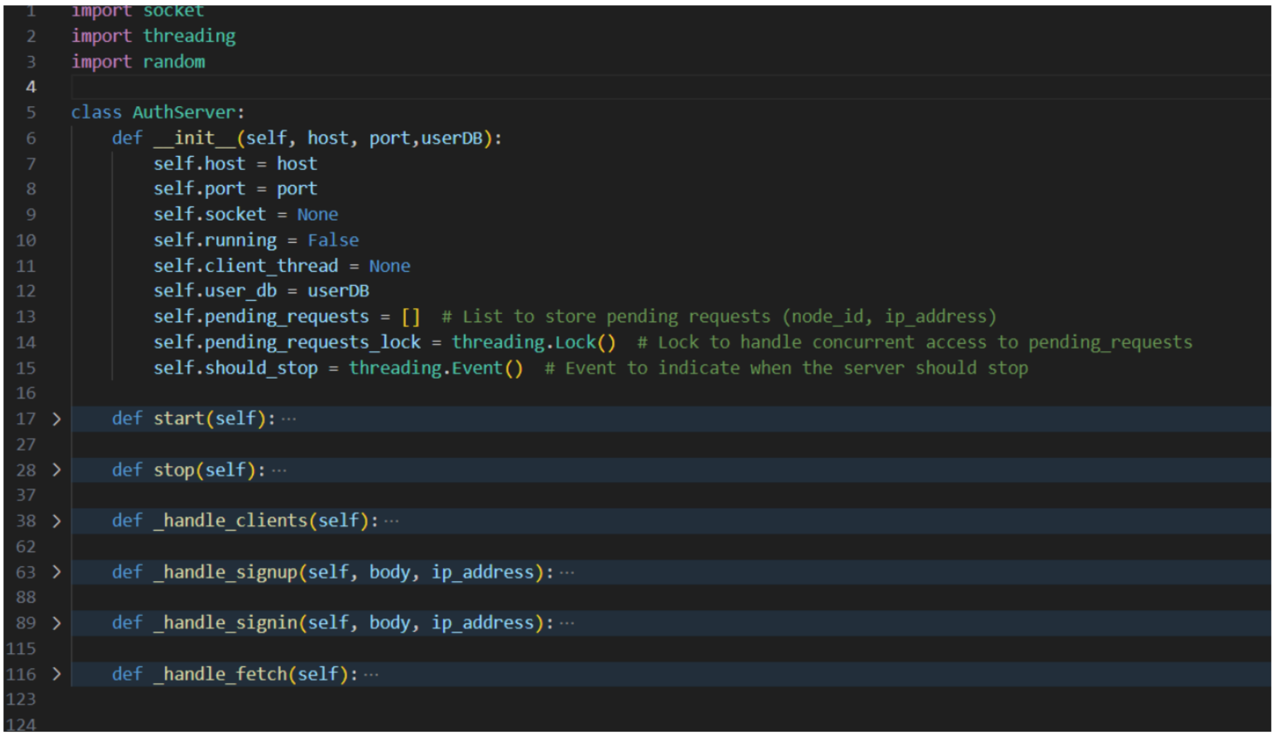


```
class UserDatabase:
    def __init__(self): ···

    def signup(self, username, password, node_id): ···

    def signin(self, username, password): ···

    def no_of_users(self): ···

user_db = UserDatabase()
```

▶ This class represents a simple user database that stores user information (username, password,and node ID).
▶ The signup method allows users to sign up with a new username, password, and node ID. It returns True if the signup is successful and False if the username already exists.
▶ The signin method checks if a given username and password match the stored data and returns the associated node ID if the signin is successful, or None if the credentials are invalid.
▶ The no_of_users method returns the number of registered users in the database.

11.2 Auth Server

The Auth Server acts as an authentication server that listens for UDP packets on a specified host and port. It handles signup, signin, and fetch requests from clients.

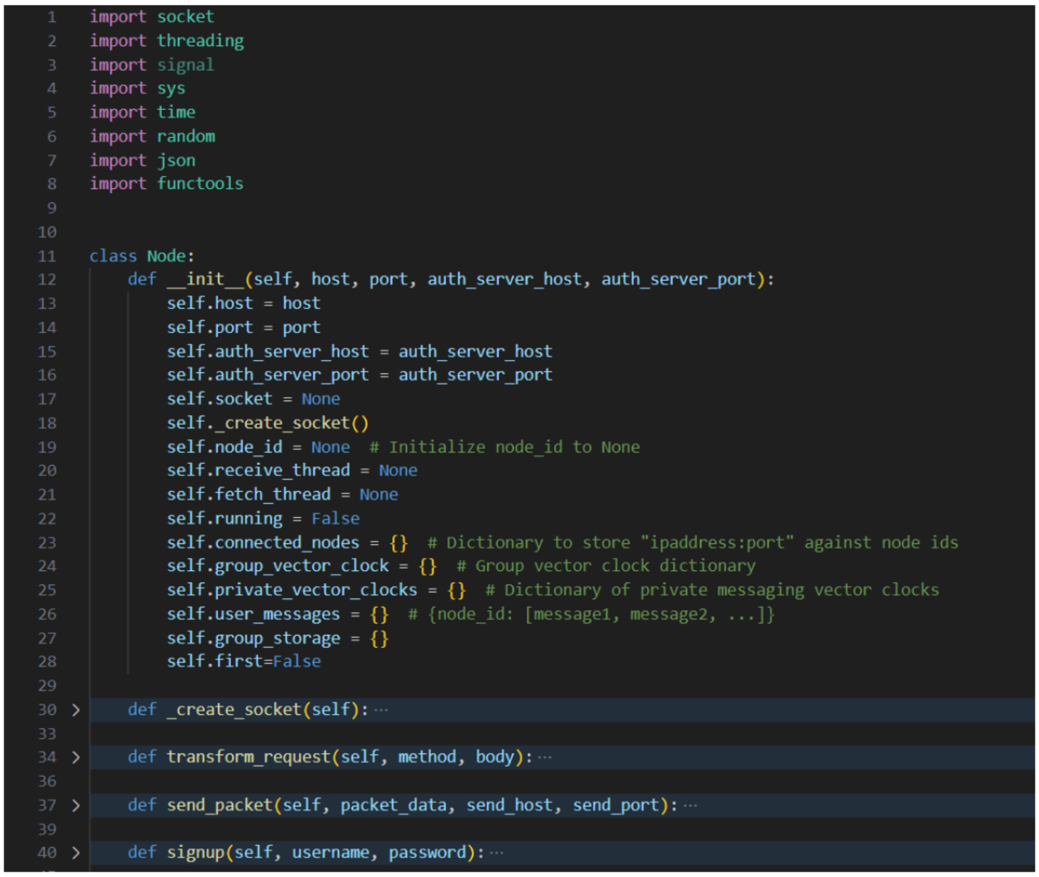


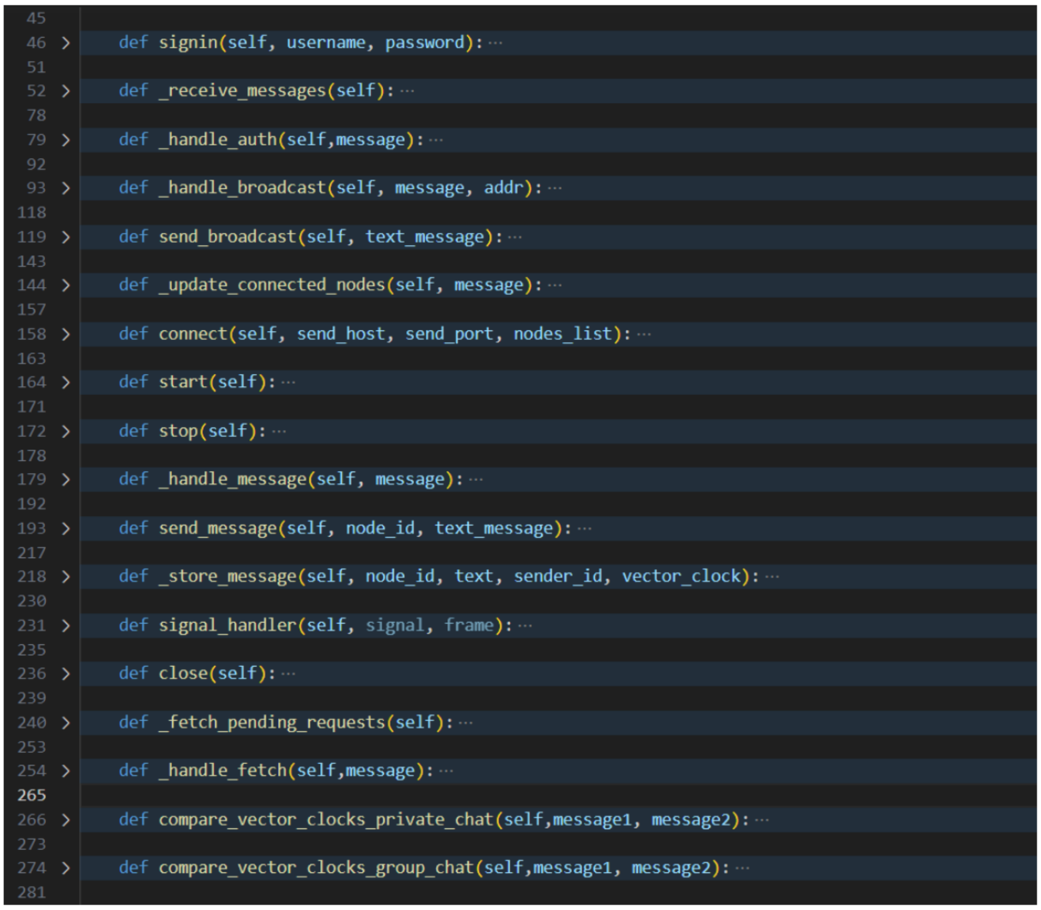
```
import socket
import threading
import random

class AuthServer:
    def __init__(self, host, port,userDB):
        self.host = host
        self.port = port
        self.socket = None
        self.running = False
        self.client_thread = None
        self.user_db = userDB
        self.pending_requests = []  # List to store pending requests (node_id, ip_address)
        self.pending_requests_lock = threading.Lock()  # Lock to handle concurrent access to pending_requests
        self.should_stop = threading.Event()  # Event to indicate when the server should stop

    def start(self):···

    def stop(self):···

    def _handle_clients(self):···

    def _handle_signup(self, body, ip_address):···

    def _handle_signin(self, body, ip_address):···

    def _handle_fetch(self):···

```

▶ The start method starts the server and creates a thread to handle client requests.
▶ The _handle_clients method processes incoming client requests and responds accordingly.
▶ The _handle_signup method processes signup requests, adds the new user to the UserDatabase, and maintains a list of pending requests.
▶ The _handle_signin method processes signin requests and maintains a list of pending requests.
▶ The _handle_fetch method responds to fetch requests by sending the list of pending requests to the client.

11.3 Node

This class represents a node in the chat network that can communicate with other nodes through UDP. It stores information such as host, port, authentication server host, port, node ID, connected nodes, group vector clock, private vector clocks, and user messages.


```python
import socket
import threading
import signal
import sys
import time
import random
import json
import functools


class Node:
    def __init__(self, host, port, auth_server_host, auth_server_port):
        self.host = host
        self.port = port
        self.auth_server_host = auth_server_host
        self.auth_server_port = auth_server_port
        self.socket = None
        self._create_socket()
        self.node_id = None  # Initialize node_id to None
        self.receive_thread = None
        self.fetch_thread = None
        self.running = False
        self.connected_nodes = {}  # Dictionary to store "ipaddress:port" against node ids
        self.group_vector_clock = {}  # Group vector clock dictionary
        self.private_vector_clocks = {}  # Dictionary of private messaging vector clocks
        self.user_messages = {}  # {node_id: [message1, message2, ...]}
        self.group_storage = {}
        self.first=False

    def _create_socket(self): ...

    def transform_request(self, method, body): ...

    def send_packet(self, packet_data, send_host, send_port): ...

    def signup(self, username, password): ...
```

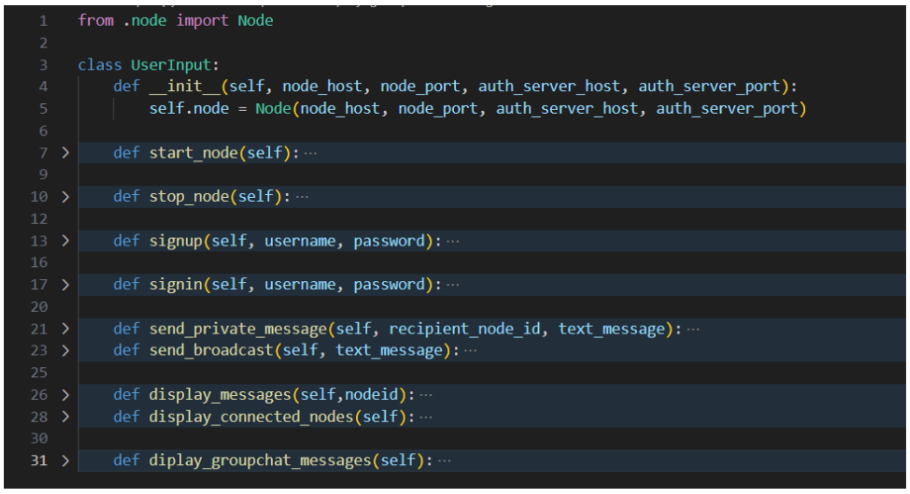

```python

    def signin(self, username, password): ...

    def _receive_messages(self): ...

    def _handle_auth(self,message): ...

    def _handle_broadcast(self, message, addr): ...

    def send_broadcast(self, text_message): ...

    def _update_connected_nodes(self, message): ...

    def connect(self, send_host, send_port, nodes_list): ...

    def start(self): ...

    def stop(self): ...

    def _handle_message(self, message): ...

    def send_message(self, node_id, text_message): ...

    def _store_message(self, node_id, text, sender_id, vector_clock): ...

    def signal_handler(self, signal, frame): ...

    def close(self): ...

    def _fetch_pending_requests(self): ...

    def _handle_fetch(self,message): ...

    def compare_vector_clocks_private_chat(self,message1, message2): ...

    def compare_vector_clocks_group_chat(self,message1, message2): ...

```

```
    def get_groupchat_messages(self):···

    def get_private_messages(self,nodeid):···
```

▶ **__init__(self, host, port, auth_server_host, auth_server_port):** Initializes a new node with its host, port, and authentication server's host and port.
▶ **start(self):** Starts the node, creating threads for receiving messages and fetching pending requests.
▶ **stop(self):** Stops the node and closes the socket.
▶ **signup(self, username, password):** Sends a signup request to the authentication server.
▶ **signin(self, username, password):** Sends a signin request to the authentication server.
▶ **send_broadcast(self, text_message):** Sends a broadcast message to all connected peers.
▶ **send_message(self, node_id, text_message):** Sends a private message to a specific node.
▶ **_receive_messages(self):** Handles incoming messages.
▶ **_handle_auth(self, message):** Handles the response from the authentication server.
▶ **_update_connected_nodes(self, message):** Updates the list of connected nodes.
▶ **_handle_broadcast(self, message, addr):** Handles incoming broadcast messages.
▶ **_handle_message(self, message):** Handles incoming private messages.
▶ **_store_message(self, node_id, text, sender_id, vector_clock):** Stores a private message.
▶ **_fetch_pending_requests(self):** Fetches pending requests from the authentication server.
▶ **_handle_fetch(self, message):** Handles the response of pending fetch requests.
▶ **compare_vector_clocks_private_chat(self, message1, message2):** Compares vector clocks forprivate messages.
▶ **compare_vector_clocks_group_chat(self, message1, message2):** Compares vector clocks for group messages.
▶ **get_groupchat_messages(self):** Retrieves and displays group chat messages.
▶ **get_private_messages(self, nodeid):** Retrieves and displays private messages for a specific node.
▶ **signal_handler(self, signal, frame):** Handles SIGINT signal for graceful termination.
▶ **close(self):** Closes the node gracefully.

11.4 User Input

This class acts as an interface to the **Node** class and sends various commands such as signup, signin, send private messages, send broadcast messages, and display messages.



```
from .node import Node

class UserInput:
    def __init__(self, node_host, node_port, auth_server_host, auth_server_port):
        self.node = Node(node_host, node_port, auth_server_host, auth_server_port)

    def start_node(self):···

    def stop_node(self):···

    def signup(self, username, password):···

    def signin(self, username, password):···

    def send_private_message(self, recipient_node_id, text_message):···
    def send_broadcast(self, text_message):···

    def display_messages(self,nodeid):···
    def display_connected_nodes(self):···

    def diplay_groupchat_messages(self):···
```

▶ **__init__(self, node_host, node_port, auth_server_host, auth_server_port):** Initializes UserInput with node and authentication server details.
▶ **start_node(self):** Starts the node.
▶ **stop_node(self):** Stops the node.
▶ **signup(self, username, password):** Sends a signup request to the node.
▶ **signin(self, username, password):** Sends a signin request to the node.
▶ **send_private_message(self, recipient_node_id, text_message):** Sends a private message to a specific node.
▶ **send_broadcast(self, tex_message):** Sends a broadcast message to all connected peers.
▶ **display_messages(self, nodeid):** Displays private messages with a specific node.
▶ **display_connected_nodes(self):** Displays the list of connected nodes.
▶ **display_groupchat_messages(self):** Displays group chat messages.

12 Starter Scripts

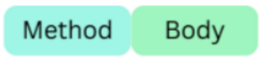
The code contains 2 scripts. 1 for the auth server. And the other for the nodes. Their initiating commands are :

▶ **Python index.py** ▶ **Python node_client.py ¡port¿ ¡username¿ ¡password¿** The index.py file just initiates the server on localhost with the port of 12345.

The node_client.py file initiates the node with args given by the user. All the test cases were implemented by applying changes in this file.

12.1 Message Structures

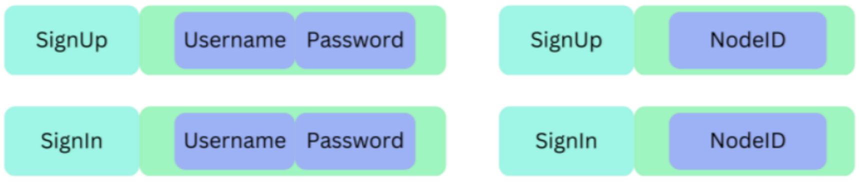
The general message protocol is as follows :



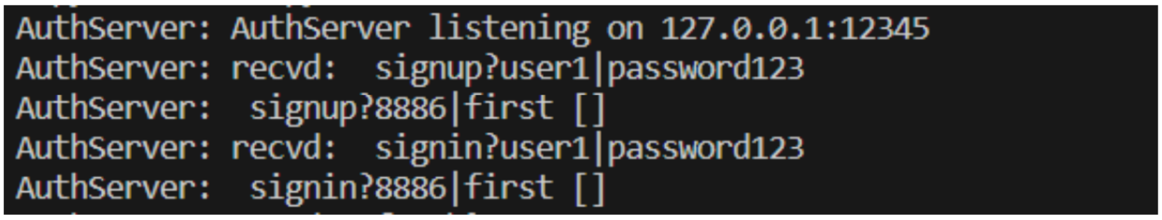
The Method and the Body are separated by "?".
And each field in the body is separated by "—".
By using this approach, we can define custom handlers for each method on the sender and receiver sides. Hence providing a great level of control over the systems communication.

12.2 Signup/SignIn

Request/Response-To Auth Server

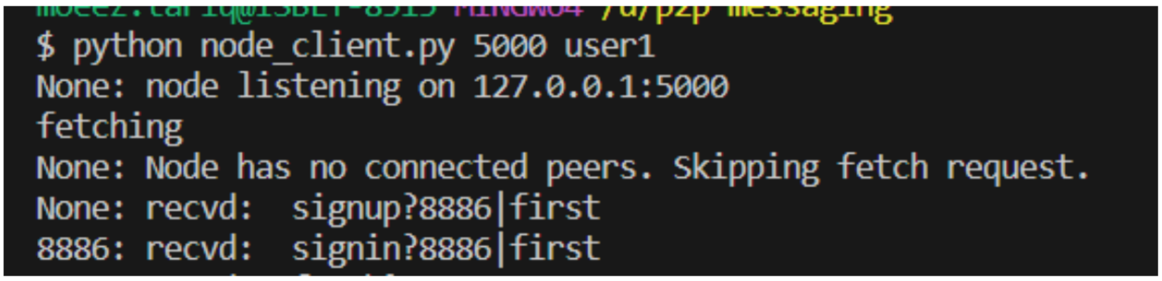


▶ **Auth Server side**



```
AuthServer: AuthServer listening on 127.0.0.1:12345
AuthServer: recvd:  signup?user1|password123
AuthServer:  signup?8886|first []
AuthServer: recvd:  signin?user1|password123
AuthServer:  signin?8886|first []
```

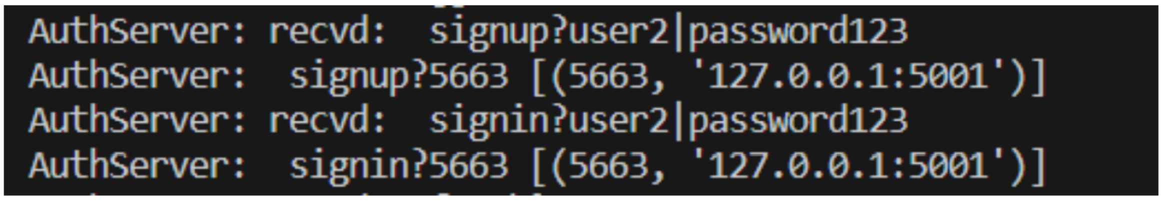
▶ **Node Side**



```
$ python node_client.py 5000 user1
None: node listening on 127.0.0.1:5000
fetching
None: Node has no connected peers. Skipping fetch request.
None: recvd:  signup?8886|first
8886: recvd:  signin?8886|first
```

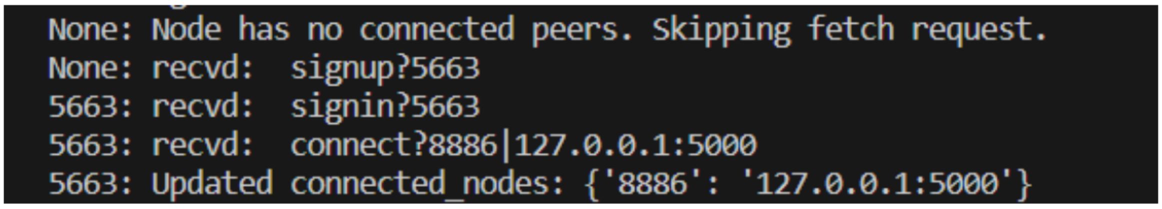
The **"first"** keyword is optionally added to the first node. Remainder do not receive that. Like:

▶ **Auth Server side**



```
AuthServer: recvd:  signup?user2|password123
AuthServer:  signup?5663 [(5663, '127.0.0.1:5001')]
AuthServer: recvd:  signin?user2|password123
AuthServer:  signin?5663 [(5663, '127.0.0.1:5001')]
```

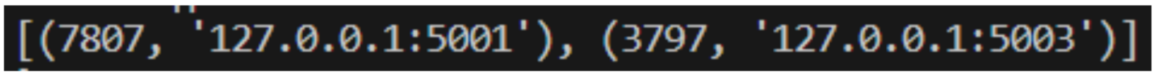
▶ **Node Side**



```
None: Node has no connected peers. Skipping fetch request.
None: recvd:  signup?5663
5663: recvd:  signin?5663
5663: recvd:  connect?8886|127.0.0.1:5000
5663: Updated connected_nodes: {'8886': '127.0.0.1:5000'}
```

The second node then waits for the bootstrap node. And it's pending request is stored in the auth server.

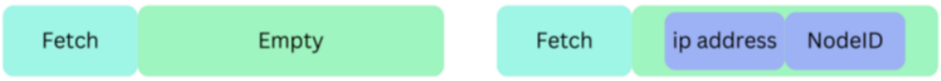
▶ **Pending Requests (nodeid,IP)**



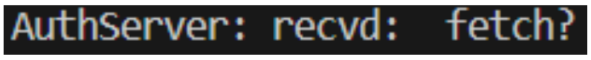
```
[(7807, '127.0.0.1:5001'), (3797, '127.0.0.1:5003')]
```

12.3 Fetching Pending Requests

Existing Node querying Auth server for Pending Requests.
Request/Response :

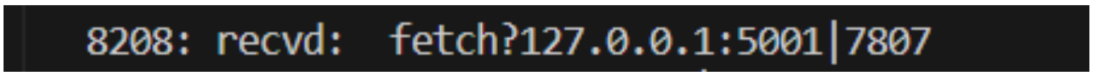


▶ **Auth Server side**



```
AuthServer: recvd:  fetch?
```

▶ **Node Side**

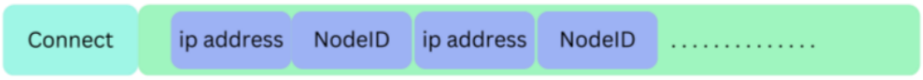


```
8208: recvd:  fetch?127.0.0.1:5001|7807
```

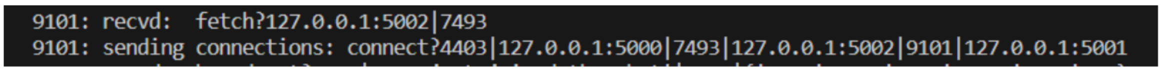
12.4 Bootstrapping

In Bootstrapping step, the existing node transfers its existing connections to the new node that's awaiting connection to the network.

Request/Response

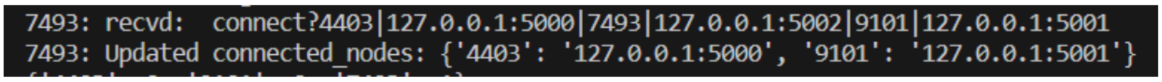


▶ **Sender Node Side**



```
9101: recvd:  fetch?127.0.0.1:5002|7493
9101: sending connections: connect?4403|127.0.0.1:5000|7493|127.0.0.1:5002|9101|127.0.0.1:5001
```

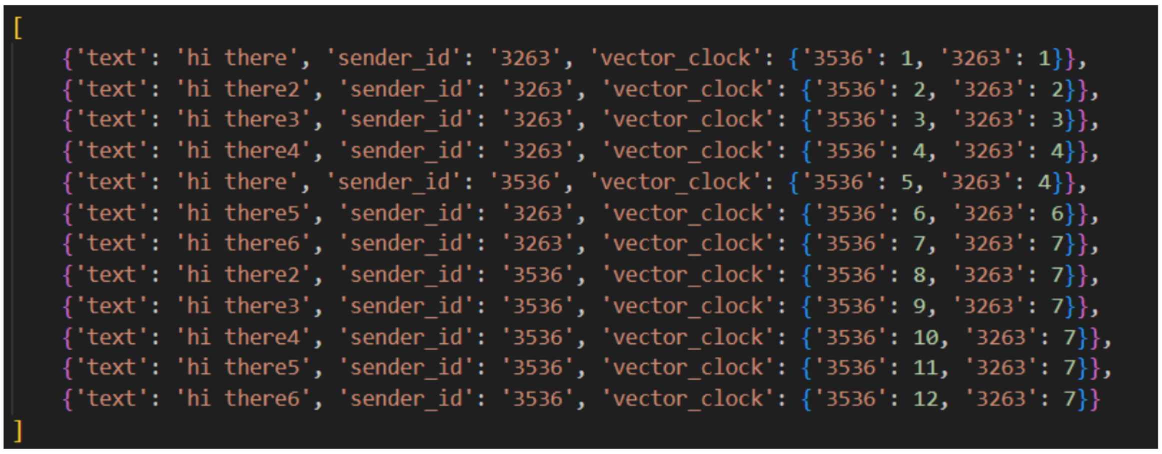
▶ **Receiver Node Side**



```
7493: recvd:  connect?4403|127.0.0.1:5000|7493|127.0.0.1:5002|9101|127.0.0.1:5001
7493: Updated connected_nodes: {'4403': '127.0.0.1:5000', '9101': '127.0.0.1:5001'}
```

▶ **Private Messages History**

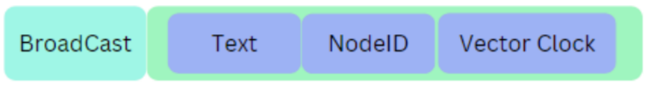
For this, 2 nodes were set up. And they sent messages to each other after a set time. So that a sense of chat between two people could be imagined. The message history is retrieved as :



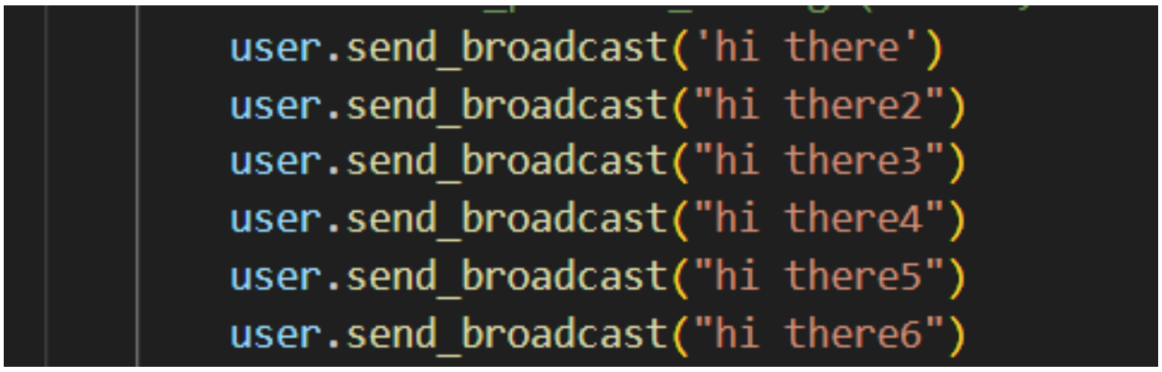
```
[
    {'text': 'hi there', 'sender_id': '3263', 'vector_clock': {'3536': 1, '3263': 1}},
    {'text': 'hi there2', 'sender_id': '3263', 'vector_clock': {'3536': 2, '3263': 2}},
    {'text': 'hi there3', 'sender_id': '3263', 'vector_clock': {'3536': 3, '3263': 3}},
    {'text': 'hi there4', 'sender_id': '3263', 'vector_clock': {'3536': 4, '3263': 4}},
    {'text': 'hi there', 'sender_id': '3536', 'vector_clock': {'3536': 5, '3263': 4}},
    {'text': 'hi there5', 'sender_id': '3263', 'vector_clock': {'3536': 6, '3263': 6}},
    {'text': 'hi there6', 'sender_id': '3263', 'vector_clock': {'3536': 7, '3263': 7}},
    {'text': 'hi there2', 'sender_id': '3536', 'vector_clock': {'3536': 8, '3263': 7}},
    {'text': 'hi there3', 'sender_id': '3536', 'vector_clock': {'3536': 9, '3263': 7}},
    {'text': 'hi there4', 'sender_id': '3536', 'vector_clock': {'3536': 10, '3263': 7}},
    {'text': 'hi there5', 'sender_id': '3536', 'vector_clock': {'3536': 11, '3263': 7}},
    {'text': 'hi there6', 'sender_id': '3536', 'vector_clock': {'3536': 12, '3263': 7}}
]
```

The trends of the vector clocks clearly show that the causality of the messages is maintained.

12.5 Broadcast Message

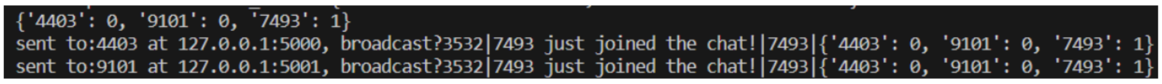


This case is demonstrated by sending 6 broadcast messages.



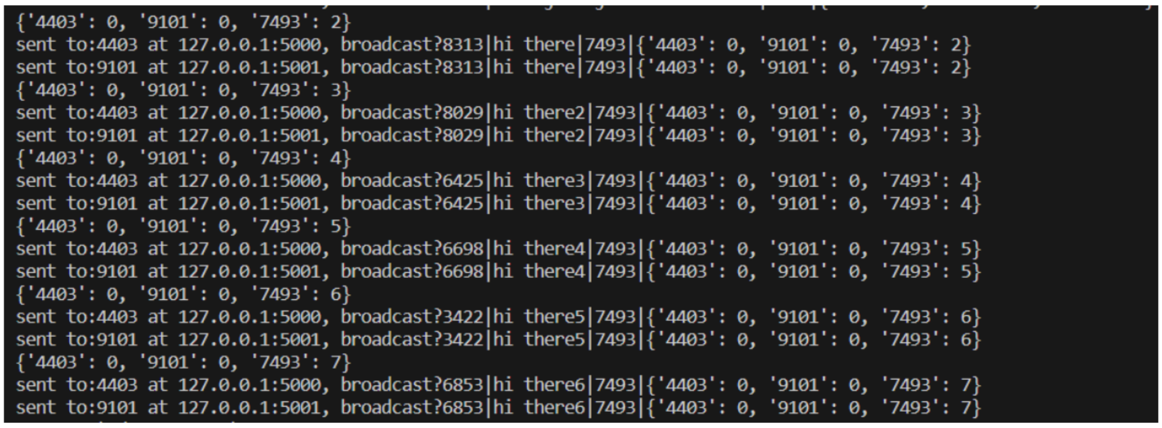
```
user.send_broadcast('hi there')
user.send_broadcast("hi there2")
user.send_broadcast("hi there3")
user.send_broadcast("hi there4")
user.send_broadcast("hi there5")
user.send_broadcast("hi there6")
```

▶ **Sender Node Side**
First Broadcast Message



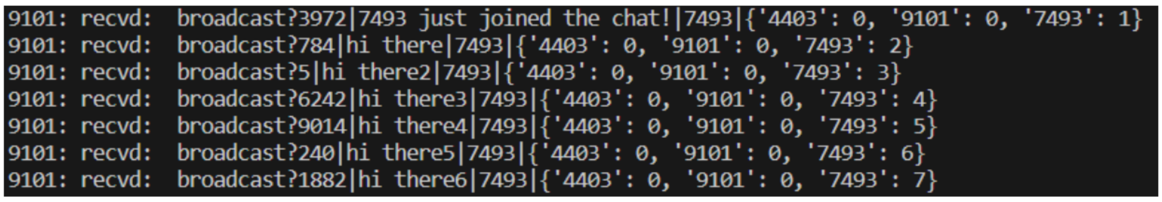
```
{'4403': 0, '9101': 0, '7493': 1}
sent to:4403 at 127.0.0.1:5000, broadcast?3532|7493 just joined the chat!|7493|{'4403': 0, '9101': 0, '7493': 1}
sent to:9101 at 127.0.0.1:5001, broadcast?3532|7493 just joined the chat!|7493|{'4403': 0, '9101': 0, '7493': 1}
```

The First Broadcast message is sent when the new node discovers new nodes by the help of the bootstrap node. It sends this message to tell all the other nodes about its connectivity so that they can update their connections database. **Normal Broadcast Message**



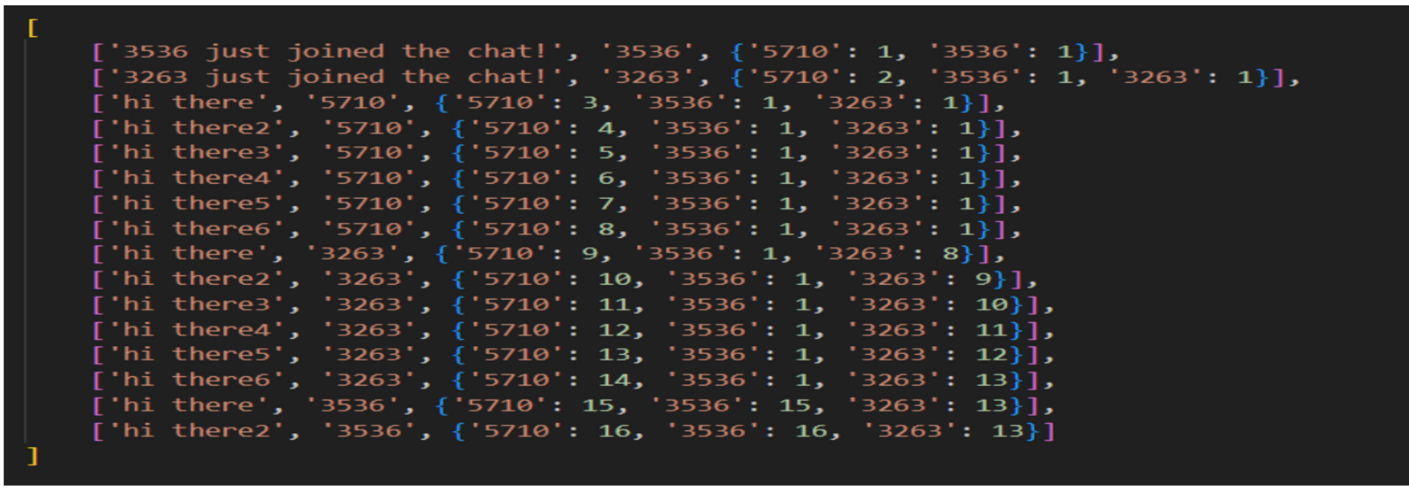
```
{'4403': 0, '9101': 0, '7493': 2}
sent to:4403 at 127.0.0.1:5000, broadcast?8313|hi there|7493|{'4403': 0, '9101': 0, '7493': 2}
sent to:9101 at 127.0.0.1:5001, broadcast?8313|hi there|7493|{'4403': 0, '9101': 0, '7493': 2}
{'4403': 0, '9101': 0, '7493': 3}
sent to:4403 at 127.0.0.1:5000, broadcast?8029|hi there2|7493|{'4403': 0, '9101': 0, '7493': 3}
sent to:9101 at 127.0.0.1:5001, broadcast?8029|hi there2|7493|{'4403': 0, '9101': 0, '7493': 3}
{'4403': 0, '9101': 0, '7493': 4}
sent to:4403 at 127.0.0.1:5000, broadcast?6425|hi there3|7493|{'4403': 0, '9101': 0, '7493': 4}
sent to:9101 at 127.0.0.1:5001, broadcast?6425|hi there3|7493|{'4403': 0, '9101': 0, '7493': 4}
{'4403': 0, '9101': 0, '7493': 5}
sent to:4403 at 127.0.0.1:5000, broadcast?6698|hi there4|7493|{'4403': 0, '9101': 0, '7493': 5}
sent to:9101 at 127.0.0.1:5001, broadcast?6698|hi there4|7493|{'4403': 0, '9101': 0, '7493': 5}
{'4403': 0, '9101': 0, '7493': 6}
sent to:4403 at 127.0.0.1:5000, broadcast?3422|hi there5|7493|{'4403': 0, '9101': 0, '7493': 6}
sent to:9101 at 127.0.0.1:5001, broadcast?3422|hi there5|7493|{'4403': 0, '9101': 0, '7493': 6}
{'4403': 0, '9101': 0, '7493': 7}
sent to:4403 at 127.0.0.1:5000, broadcast?6853|hi there6|7493|{'4403': 0, '9101': 0, '7493': 7}
sent to:9101 at 127.0.0.1:5001, broadcast?6853|hi there6|7493|{'4403': 0, '9101': 0, '7493': 7}
```

You can see that the Vector clock against 7493 keeps on increasing with each message. Indicating backto-back broadcast messages. ▶ **Receiver Node Side**



```
9101: recvd:  broadcast?3972|7493 just joined the chat!|7493|{'4403': 0, '9101': 0, '7493': 1}
9101: recvd:  broadcast?784|hi there|7493|{'4403': 0, '9101': 0, '7493': 2}
9101: recvd:  broadcast?5|hi there2|7493|{'4403': 0, '9101': 0, '7493': 3}
9101: recvd:  broadcast?6242|hi there3|7493|{'4403': 0, '9101': 0, '7493': 4}
9101: recvd:  broadcast?9014|hi there4|7493|{'4403': 0, '9101': 0, '7493': 5}
9101: recvd:  broadcast?240|hi there5|7493|{'4403': 0, '9101': 0, '7493': 6}
9101: recvd:  broadcast?1882|hi there6|7493|{'4403': 0, '9101': 0, '7493': 7}
```

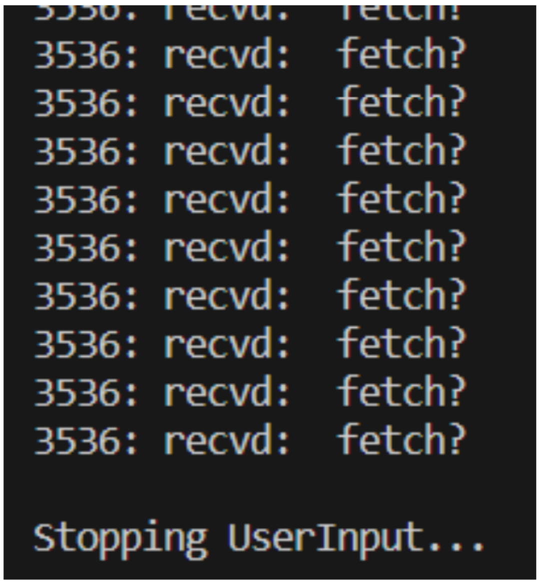
▶ **Broadcast Messages Display**
We connected 3 nodes and all of them sent broadcast messages at different rates. This is the ordered display of messages from the 1st node.



```
[
    ['3536 just joined the chat!', '3536', {'5710': 1, '3536': 1}],
    ['3263 just joined the chat!', '3263', {'5710': 2, '3536': 1, '3263': 1}],
    ['hi there', '5710', {'5710': 3, '3536': 1, '3263': 1}],
    ['hi there2', '5710', {'5710': 4, '3536': 1, '3263': 1}],
    ['hi there3', '5710', {'5710': 5, '3536': 1, '3263': 1}],
    ['hi there4', '5710', {'5710': 6, '3536': 1, '3263': 1}],
    ['hi there5', '5710', {'5710': 7, '3536': 1, '3263': 1}],
    ['hi there6', '5710', {'5710': 8, '3536': 1, '3263': 1}],
    ['hi there', '3263', {'5710': 9, '3536': 1, '3263': 8}],
    ['hi there2', '3263', {'5710': 10, '3536': 1, '3263': 9}],
    ['hi there3', '3263', {'5710': 11, '3536': 1, '3263': 10}],
    ['hi there4', '3263', {'5710': 12, '3536': 1, '3263': 11}],
    ['hi there5', '3263', {'5710': 13, '3536': 1, '3263': 12}],
    ['hi there6', '3263', {'5710': 14, '3536': 1, '3263': 13}],
    ['hi there', '3536', {'5710': 15, '3536': 15, '3263': 13}],
    ['hi there2', '3536', {'5710': 16, '3536': 16, '3263': 13}]
]
```

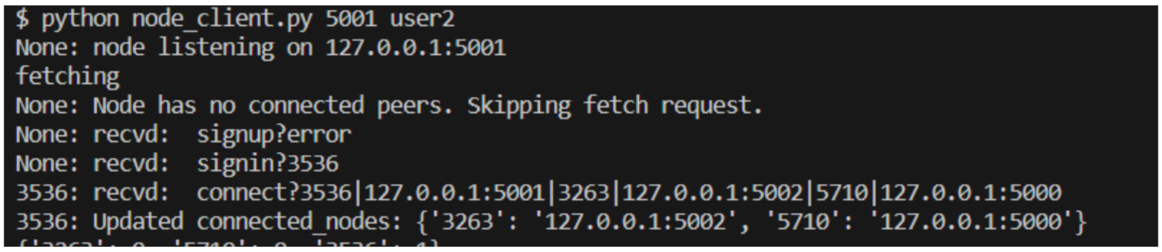
12.6 Disconnect/Reconnection of nodes

If the server is up and running, the node will resume its nodeid. ▶ **Before Disconnect**



```
3536: recvd:  fetch?
3536: recvd:  fetch?
3536: recvd:  fetch?
3536: recvd:  fetch?
3536: recvd:  fetch?
3536: recvd:  fetch?
3536: recvd:  fetch?
3536: recvd:  fetch?
3536: recvd:  fetch?
3536: recvd:  fetch?

Stopping UserInput...
```

▶ **Reconnection**

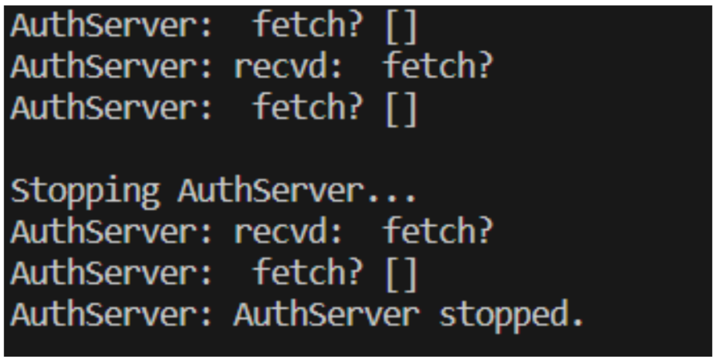


```
$ python node_client.py 5001 user2
None: node listening on 127.0.0.1:5001
fetching
None: Node has no connected peers. Skipping fetch request.
None: recvd:  signup?error
None: recvd:  signin?3536
3536: recvd:  connect?3536|127.0.0.1:5001|3263|127.0.0.1:5002|5710|127.0.0.1:5000
3536: Updated connected_nodes: {'3263': '127.0.0.1:5002', '5710': '127.0.0.1:5000'}
```

Signup gives an error because a user with the same credentials already exists. Sign in gave the node its original nodeid. Then the pending requests mechanism helped the node regain connections from existing nodes.

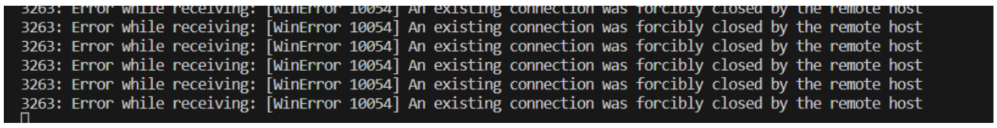
12.7 Disconnect/Reconnection of Auth Server

▶ **Before Disconnect**



```
AuthServer:  fetch? []
AuthServer: recvd:  fetch?
AuthServer:  fetch? []

Stopping AuthServer...
AuthServer: recvd:  fetch?
AuthServer:  fetch? []
AuthServer: AuthServer stopped.
```

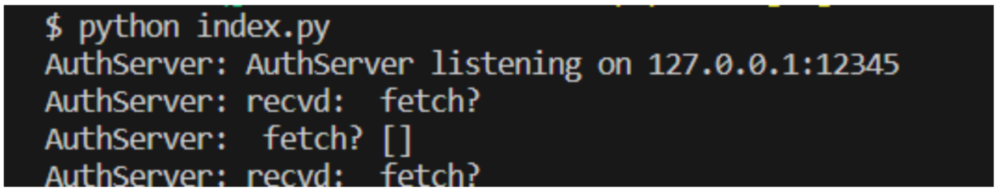
▶ **Node Reaction**



```
3263: Error while receiving: [WinError 10054] An existing connection was forcibly closed by the remote host
3263: Error while receiving: [WinError 10054] An existing connection was forcibly closed by the remote host
3263: Error while receiving: [WinError 10054] An existing connection was forcibly closed by the remote host
3263: Error while receiving: [WinError 10054] An existing connection was forcibly closed by the remote host
3263: Error while receiving: [WinError 10054] An existing connection was forcibly closed by the remote host
3263: Error while receiving: [WinError 10054] An existing connection was forcibly closed by the remote host
```

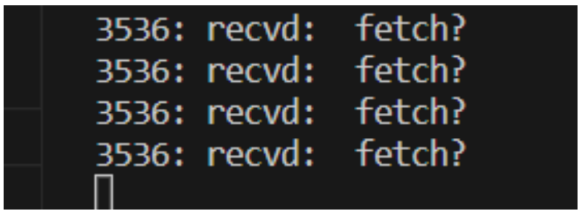
As the nodes keep trying to hit the auth server for pending requests that they can handle, they start getting errors.

▶ **Reconnection of Auth Server**



```
$ python index.py
AuthServer: AuthServer listening on 127.0.0.1:12345
AuthServer: recvd:  fetch?
AuthServer:  fetch? []
AuthServer: recvd:  fetch?
```

▶ **Node Reaction**



```
3536: recvd:  fetch?
3536: recvd:  fetch?
3536: recvd:  fetch?
3536: recvd:  fetch?
```

After the auth server gets back online. The nodes start getting responses. But without the server, they did not go down. Hence keeping minimal dependency on the auth server except authentication and partial bootstrapping.

13 Conclusion

The implementation and architecture of the decentralized P2P messaging system described in this report highlights the utilization of UDP for lightweight and efficient communication. The system incorporates an authentication server, existing nodes, and relevant protocols to enable secure messaging, network connectivity, dynamic node updates, and features like private messaging, broadcast messaging, chat history, and message ordering using vector clocks. The use of UDP ensures a scalable and responsive messaging platform without requiring sticky sessions, contributing to the system's efficiency and effectiveness.