Disclaimer: Parts of this report code debugging was performed using A.I.

1. Slykord System Overview

Slykord (Slack, Skype & Discord) is a leader based peer to peer chat system (influenced from 2010 Call of Duty Lobbies, which were peer-to-peer with 1 host that would be transferred to another user upon disconnecting). Each running instance is a node. Nodes can create rooms, join existing rooms and exchange text messages and files over a local network.

There is no central server. Any node can start a new room. Once a room exists, other nodes discover it by listening for small UDP broadcast packets on a set of discovery ports. When you join a room, your node connects directly to the other peers in that room using UDP sockets.

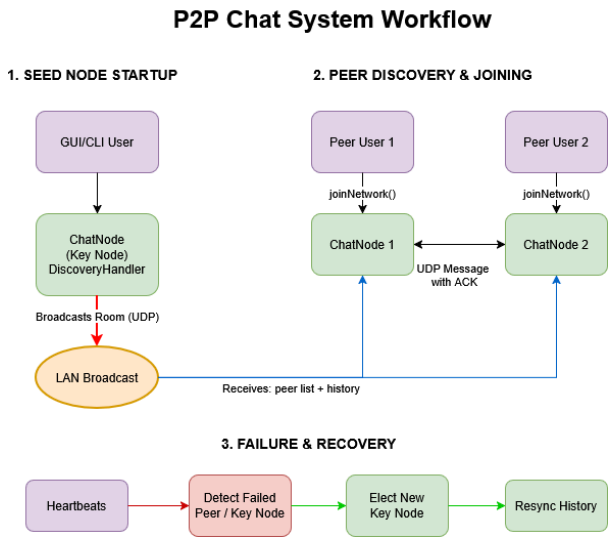
Inside each room, one node becomes the key node. The key node is simply the node that has been in the room the longest. The system tracks this using a join time on each node and peer, stored in objects like PeerInfo. If the key node leaves or times out, the remaining peers elect a new key based on who joined first. That election is automatic and does not rely on any outside service.

All chat messages and file fragments travel directly between peers in that room. Messages carry a Lamport timestamp and a unique message ID so that every node can place them in a consistent order. Each node stores its local copy of the room history and also logs it to disk so that it can redraw the chat from its own files, even if no other peers are online.

Slykord has two user interfaces that share the same networking core:

- A command line client for tests and debugging
- A Swing based GUI that looks and feels more like a classic desktop chat app

Both use the same middleware object, `ChatNode`, which hides all the networking, discovery, history sync and file transfer logic.



2. Which Requirements are Met

Slykord meets all the “must have” and “should have” requirements, and all of the “could have” features listed in the assignment brief.

For the basic requirements, the system lets at least two nodes connect, choose human readable nicknames and exchange messages. Each node gets a random UUID at startup that uniquely identifies it across the network. When a message is sent, every node in the same room eventually shows that message, in the same order, using Lamport clocks. UDP is the only communication protocol used between peers. When a node disconnects cleanly, it sends a leave notification and closes its sockets. When it disappears without warning, other nodes detect the missing heartbeats, remove it from their peer lists and elect a new key node if needed.

For the intermediate requirements, Slykord handles more than two nodes per room. You can start several clients and join them all into the same room. A new node that joins an existing room does not start from an empty view. It contacts the key node, requests the current history and applies a snapshot of all messages seen so far. Messages and files are all tagged with unique IDs. Messages are timestamped with Lamport clocks, and each node updates its logical clock on send and on receive. This keeps the message ordering consistent across peers.

The system has an automated robot chat mode. When you toggle it on, a background thread in the client periodically generates canned messages and sends them like a normal user. Malformed packets and unknown commands are handled defensively with try catch blocks and error messages, rather than crashing the process. The key node model fits the “true peer to peer” definition: any node can be a seed for a new room, and any node can become key if the previous key disconnects or fails.

Slykord includes explicit support for simulating network failure. The /netloss command lets you set a percentage of simulated packet loss. The networking layer checks this setting for both outbound and inbound messages and drops packets based on a random number generator. This lets you show what happens when messages or history snapshots do not always arrive.

For the advanced requirements, the system supports two forms of discovery. First, nodes discover each other via peer lists when joining a room. Second, they discover rooms themselves using UDP broadcast discovery packets on a range of ports. A new node does not need to know any IP address up front. It can simply wait for discovery beacons and then choose from the list of visible rooms.

If a node has missed messages due to packet loss or being offline, it can request a full history snapshot from its peers using a resync command. This clears or replaces its local in memory history with the version from another node. This covers the “obtain missing data” requirement.

The file transfer feature sends binary files directly between peers in the same room using UDP, not a TCP based client or server. Files are read into memory, split into chunks that are small enough to avoid IP fragmentation, and each chunk is wrapped in a message fragment with an ID and index. On the receiving side, a fragment assembler groups fragments by file ID, counts how many have arrived, and when it has the full set, writes the reassembled file to disk and notifies the user.

For clearing and rebuilding from the network, the resync feature requests and applies a fresh history snapshot, effectively rebuilding the chat view from data on the network. For redrawing from local data, you added a log based recovery path. Each message is appended to a per node log file as it is seen. When a user issues the “resync local” action, the client reads that log, reconstructs the messages in file order, feeds them back into the in memory history and redraws the chat window. This works even with no peers connected.

The second interface, the GUI (which is inspired by the design of Slack and Discord), counts as an external interface to the system. The underlying middleware does not know or care whether it is driven by the CLI or the GUI. Both call the same methods on `ChatNode` to send messages, join rooms, resync history and send files.

3. Middleware Design

The middleware is centered around the `ChatNode` class. This class owns the node identity, the UDP sockets, the current room state and the list of peers. It exposes a simple API to the rest of the application:

- Start or stop the node
- Create or join rooms
- Send chat messages or files
- Resync history from peers or from local logs
- `ChatNode` delegates lower level networking to helper classes.

`UDPHandler` owns a DatagramSocket and a worker thread pool. It sends and receives raw UDP packets. When it receives a datagram, it hands the byte array to a serializer.

`PacketSerialiser` is responsible for translating between Java objects and byte arrays. It tags each packet with a message type and applies a length prefix. For large payloads, especially file content, it splits the data into fragments below a chosen maximum packet size. Each fragment carries a file ID, fragment index and total fragment count.

On the receiving side, `FragmentAssembler` collects these fragments in a concurrent structure per file ID. It uses an atomic counter to track how many fragments have arrived. Once all fragments for a file are present, it reassembles them in order and hands the complete file back to `ChatNode`.

Discovery is handled by `DiscoveryHandler`. This component sends small UDP broadcast packets on a known port range. Each broadcast advertises the room ID, room name, member count and key node address. Other nodes listen on the same ports and update a local list of `RoomInfo` objects. This list feeds the CLI `/rooms` command and the GUI room browser.

Peer tracking uses `PeerInfo` objects, one per known node. Each `PeerInfo` holds the peer's node ID, nickname, IP and port, room ID, join time and a flag that notes if it is currently considered the key node. `ChatNode` maintains maps of all known peers, peers in the current room and known key nodes in other rooms. When a heartbeat arrives from a peer, the node updates the `lastSeen` timestamp on the corresponding `PeerInfo`. A background task periodically scans these timestamps and removes peers that have timed out.

Leader election is built on top of this metadata. When the current key leaves or times out, `LeaderGroupManager` scans the peers in the room, compares their join times and selects the earliest. If there is a tie, it breaks it by comparing UUIDs so that every node makes the same choice. The new key announces itself with a key node status message that other nodes respect unless their own records show that they joined earlier.

Ordering and consistency are handled by a Lamport clock stored inside `ChatNode`. Before a node sends any message, it increments its local clock and stamps the message. When it receives a

message, it sets its clock to one greater than the maximum of its own clock and the received clock. All nodes then sort messages by clock value and, if needed, by message ID to break exact ties. This gives a consistent total order of chat events across the network, independent of the actual arrival times over UDP.

The clients, both CLI and GUI, subscribe to events from `ChatNode`. When the node receives a new message, a history snapshot, or a file, it calls listener methods that update the terminal or the Swing components. The GUI keeps some extra state to show pending messages in grey and flip them back to white when an acknowledgement arrives, by matching message IDs. The CLI does something similar in text: it records outgoing messages as pending, prints them when an acknowledgement arrives, and marks them as unconfirmed if no acknowledgement shows up after a timeout.
Overall, the middleware follows a clear separation of concerns:

`ChatNode` orchestrates room logic, peers, messages and files
UDPHandler, PacketSerialiser and FragmentAssembler hide raw networking details
DiscoveryHandler handles room discovery
LeaderGroupManager and PeerInfo manage leadership and membership
The CLI and GUI sit on top of this layer and do not reimplement any networking logic.

4. Extra Features

On top of the core middleware, the project includes a set of features that make the system feel closer to a real chat application like Slack or Skype and help demonstrate distributed behaviour. One feature is room renaming. The key node for a room can change the room name at runtime. When the user issues the rename command, the key node updates its local RoomInfo and sends a control message to every peer in the room. Each peer updates its own copy of the room metadata and the discovery layer also refreshes its view. As a result, the new room name appears correctly in both the CLI /rooms listing and in the GUI room list. If other nodes later rediscover the same room through UDP discovery, they see the latest name instead of an outdated default. This shows that metadata for a distributed group can change over time and remain consistent.

Another feature is visual message delivery feedback. The middleware sends a lightweight acknowledgement for each chat message. When a node receives a message, it stores it in its history and sends back a `MESSAGE_ACK` acknowledgement that includes the original message ID. The sender tracks its own outgoing messages as “pending” until it sees that acknowledgement. The GUI uses this to drive visual feedback: when you send a message, it first appears in grey to show that it is not yet confirmed. When the acknowledgement arrives, the GUI finds the matching message by ID and repaints that line in white. If the acknowledgement never comes due to simulated packet loss or a failed peer, the grey state remains and the user can see that this message might not have reached anyone.

The CLI uses the same idea in text form. It keeps a map of pending messages keyed by ID. When the acknowledgement arrives, it prints the message into the local transcript as a normal sent line. If no acknowledgement appears before a timeout, the CLI prints the message with an “[UNCONFIRMED]” tag. This pushes some of the reality of unreliable networks into the user experience and shows that the system is tracking delivery at the application level, even though it is built on unreliable UDP.

History synchronisation has a few enhancements beyond the minimum. When a node joins an existing room, it does not just start from the next live message. It requests a full history snapshot from the key node or another peer. That snapshot includes all known messages with their Lamport timestamps and IDs. The joining node applies this snapshot locally and then switches to live streaming of new messages. Later, if a node suspects that it has missed messages due to network loss, it can trigger a resync from peers. The node sends a history request, receives another snapshot and merges it with its existing state. You also added a local log–based resync path. Each node writes every message it sees to a per-node log file on disk. If there are no peers available, the user can ask the node to rebuild its chat view from that log file alone. The node parses the file line by line, reconstructs message objects and redraws the view in logical order.

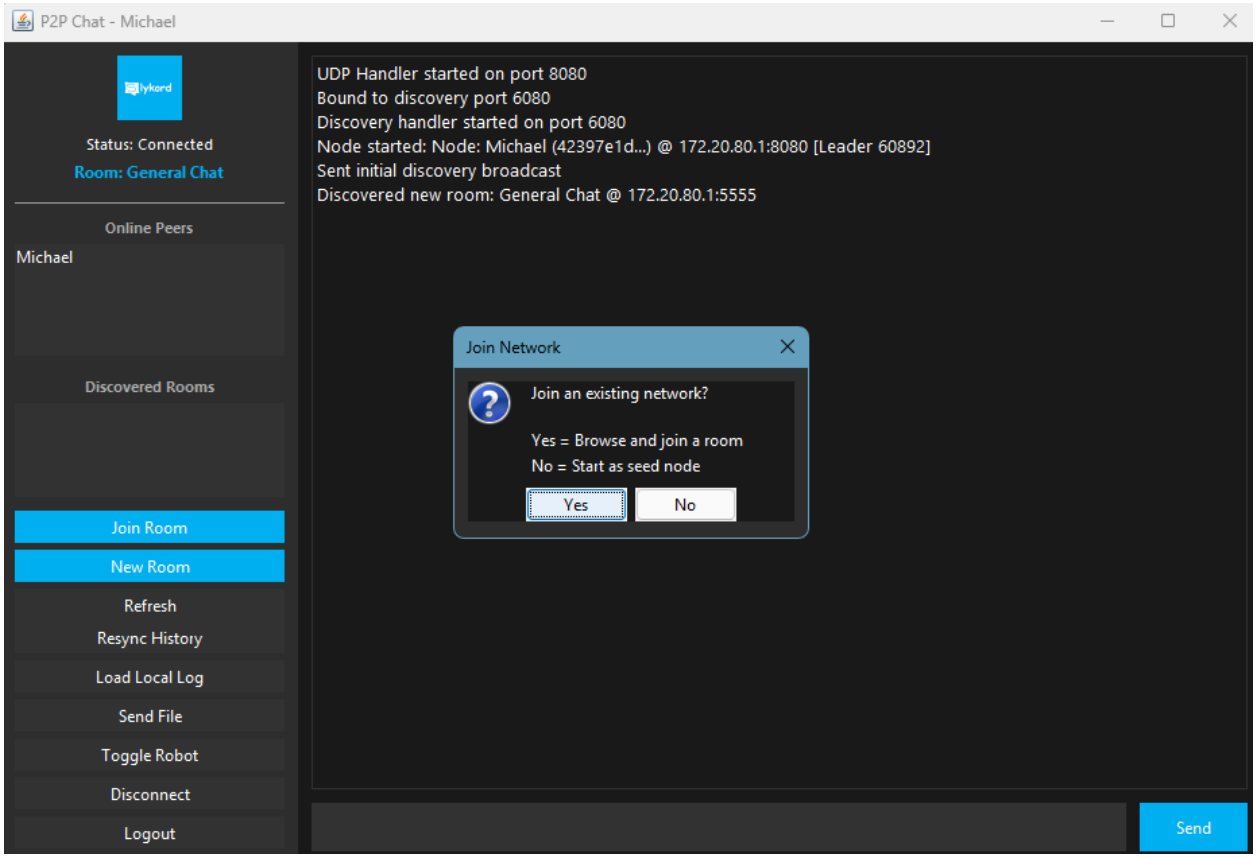
Room and network discovery are also more user friendly than a minimal solution. Discovery runs in the background and maintains a dynamic list of visible rooms on the LAN. Both interfaces expose this list in a clear way. In the CLI, the /rooms command prints a numbered list of rooms with names, member counts and addresses. In the GUI, a sidebar shows available rooms as a list you can click on. This means a new user on the network does not need to know any IP addresses. They can simply start the client, wait a moment and join one of the discovered rooms.

The project includes a network simulation feature that is integrated with the rest of the system rather than being an external test harness. The /netloss command adjusts a shared network conditions object that both the send and receive paths consult. Before sending or after receiving a packet, the code draws a random number and decides whether to drop the packet based on the configured loss percentage. Dropped packets are logged with a clear "[Simulated] Dropped …" message. Because message delivery feedback, resync and history snapshots all build on top of the same middleware, you can see realistic behaviour: grey messages that never confirm, out-of-date histories that need resync, and nodes that continue to function despite large amounts of loss.

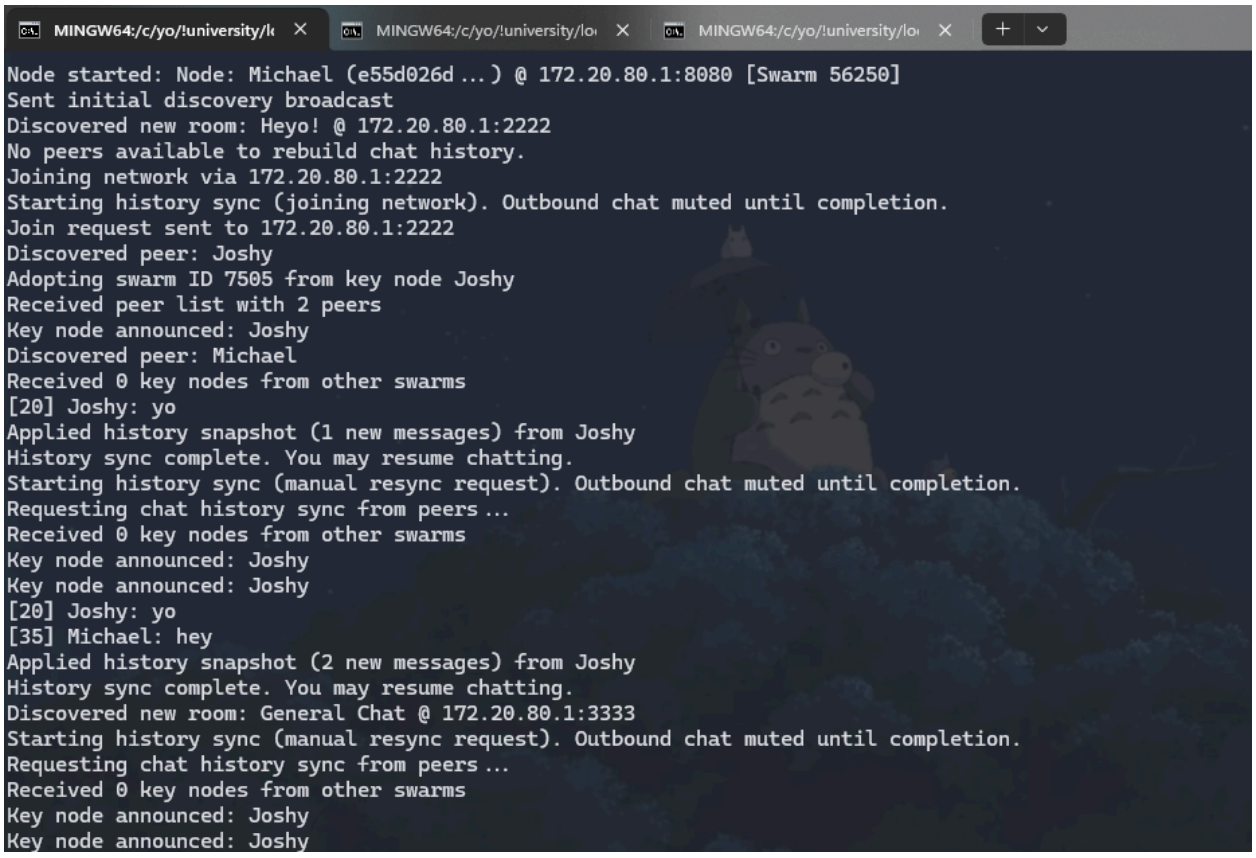
The GUI itself is a significant extra feature. It wraps the same `ChatNode` middleware as the CLI but presents a more familiar chat experience. It shows the list of peers in the current room, highlights the key node, exposes discovery and room switching as buttons and menus, and incorporates file sending and robot chat through simple controls. It also updates its connect button to switch between "Disconnect" and "Reconnect" based on the node's current state, and it clears or repopulates the peer list when joining or leaving networks. These details make it much easier to demonstrate the underlying distributed behaviour to a non-technical audience, while the CLI remains available for low-level testing and debugging.

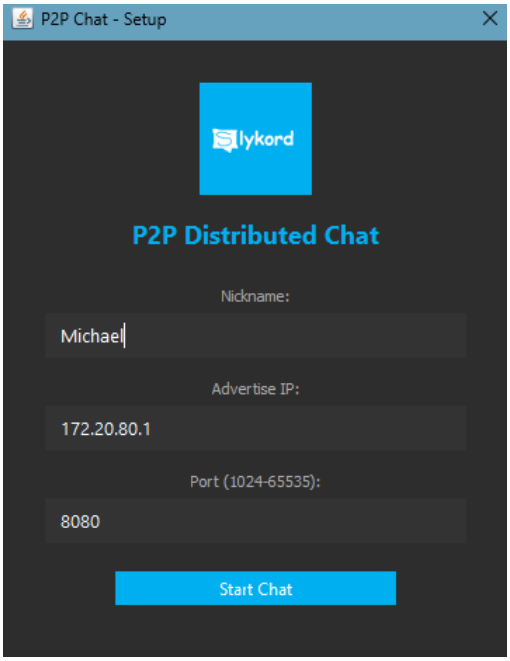
On top of the GUI and CLI interfaces, the project also includes a small HTTP broadcast API that acts as an external integration point. A separate BroadcastService process hosts a minimal HttpServer and internally runs a normal `ChatNode` that joins a room like any other peer. External tools can call endpoints such as GET /message?text=... or POST /message to inject chat messages, while GET /status and GET /peers expose the current node state and peer list as JSON. Because this service reuses the same middleware—UDP transport, Lamport clocks, history, discovery and key-node logic—API-triggered messages appear in the transcript, participate in ordering and are delivered with the same reliability guarantees as user-typed messages. This demonstrates how the middleware can be embedded behind different front-ends and used by automated systems (for example, CI pipelines or monitoring bots) without needing to speak the custom UDP protocol directly.

Screenshots of the Skykord below:



(GUI) Initial Screen upon login.


MINGW64:/c/yo/!university/k
MINGW64:/c/yo/!university/lo
MINGW64:/c/yo/!university/lo
Node started: Node: Michael (e55d026d ...) @ 172.20.80.1:8080 [Swarm 56250]
Sent initial discovery broadcast
Discovered new room: Heyo! @ 172.20.80.1:2222
No peers available to rebuild chat history.
Joining network via 172.20.80.1:2222
Starting history sync (joining network). Outbound chat muted until completion.
Join request sent to 172.20.80.1:2222
Discovered peer: Joshy
Adopting swarm ID 7505 from key node Joshy
Received peer list with 2 peers
Key node announced: Joshy
Discovered peer: Michael
Received 0 key nodes from other swarms
[20] Joshy: yo
Applied history snapshot (1 new messages) from Joshy
History sync complete. You may resume chatting.
Starting history sync (manual resync request). Outbound chat muted until completion.
Requesting chat history sync from peers ...
Received 0 key nodes from other swarms
Key node announced: Joshy
Key node announced: Joshy
[20] Joshy: yo
[35] Michael: hey
Applied history snapshot (2 new messages) from Joshy
History sync complete. You may resume chatting.
Discovered new room: General Chat @ 172.20.80.1:3333
Starting history sync (manual resync request). Outbound chat muted until completion.
Requesting chat history sync from peers ...
Received 0 key nodes from other swarms
Key node announced: Joshy
Key node announced: Joshy


P2P Chat - Setup
lykord
P2P Distributed Chat
Nickname:
Michael
Advertise IP:
172.20.80.1
Port (1024-65535):
8080
Start Chat