Technische Dokumentation: Dezentrales Peer-to-Peer Chatprogramm

BSRN – Sommersemester 2025

Fabian Matzollek, Michael Nguyen, Jin Huh, Muhammed Can Sahin, Burim Rashica

Frankfurt University of Applied Sciences
Nibelungenplatz 1
60318 Frankfurt am Main

22. Juni 2025

Zusammenfassung

Diese Dokumentation beschreibt die Entwicklung eines dezentralen Chatprogramms, das in Python implementiert wurde. Die Anwendung läuft über die Kommandozeile und ermöglicht das Versenden von Text- und Bildnachrichten zwischen mehreren Nutzern im lokalen Netzwerk – ganz ohne zentralen Server. Die Kommunikation erfolgt über ein eigenes Protokoll (SLCP), das auf UDP basiert. Das System besteht aus drei Prozessen, nutzt eine Konfigurationsdatei im TOML-Format und setzt Interprozesskommunikation zur Koordination ein.

Das Projekt wurde im Rahmen einer Portfolioprüfung im Modul Betriebssysteme und Rechnernetze an der University of Applied Sciences in Frankfurt durchgeführt. Ziel war es, die grundlegenden Konzepte der Interprozesskommunikation und der Python-Programmierung zu erlernen und zu vertiefen. Die folgende Dokumentation gibt einen Überblick über die Architektur des Spiels, die verwendeten Mittel und Bibliotheken sowie die Herausforderungen und Lösungen, die während der Entwicklung auftraten.

Inhaltsverzeichnis

1 Einleitung

Diese Dokumentation beschreibt die Konzeption und technische Umsetzung eines dezentralen Chatprogramms, das im Rahmen des Moduls *Betriebssysteme und Rechnernetze* an der Frankfurt University of Applied Sciences entwickelt wurde. Die Anwendung basiert auf dem Simple Local Chat Protocol (SLCP) und ermöglicht den Austausch von Text- und Bildnachrichten im lokalen Netzwerk.

2 Ziele und Anforderungen

In erster Linie soll das Spiel funktionsfähig sein, d.h. ausführbar sein und keine Fehler in der Nutzung beinhalten. Außerdem soll es folgende Funktionen besitzen:

- Peer-to-Peer Kommunikation ohne zentrale Serverinstanz
- Unterstützung für Text- und Bildnachrichten
- Implementierung des SLCP-Protokolls
- Nutzung von UDP (Text) und TCP (Bildübertragung)
- Drei-Prozess-Architektur: UI, Client, Discovery
- Interprozesskommunikation über Sockets (IPC)
- Konfigurierbar über eine zentrale `config.toml`
- Kommandozeilenschnittstelle (CLI)
- Dokumentation mit `doxygen`

3 Teamorganisation

Die Teamarbeit erfolgte nach dem Prinzip einer schrittweisen Übergabe: Jedes Teammitglied versuchte zunächst, ein bestimmtes Modul oder eine Funktion möglichst weitgehend selbstständig zu entwickeln. Anschließend wurde der jeweilige Zwischenstand dokumentiert, übergeben und vom nächsten Mitglied übernommen oder erweitert. Dadurch entstand ein fließender Entwicklungsprozess, bei dem Aufgaben nicht strikt getrennt, sondern nacheinander im Team bearbeitet wurden.

Dieser kollaborative Ansatz ermöglichte es, Probleme frühzeitig gemeinsam zu erkennen und zu lösen. Zudem förderte er das Verständnis für das Gesamtprojekt, da jedes Teammitglied Einblicke in alle zentralen Komponenten erhielt. Die Verantwortung für einzelne Module lag somit nicht exklusiv bei einzelnen Personen, sondern wurde bewusst geteilt.

4 Verwendete Bibliotheken

Zur Umsetzung des Chatprogramms kamen ausschließlich Standardbibliotheken sowie verbreitete externe Bibliotheken zum Einsatz:

- `socket` – Netzwerkkommunikation über UDP und TCP
- `threading` – Nebenläufigkeit zur parallelen Verarbeitung von Nachrichten
- `toml` – Parsen und Bearbeiten der Konfigurationsdatei `config.toml`
- `sys, os` – Prozesssteuerung, Dateioperationen, Cleanup-Skripte
- `time` – Zeitgesteuerte Wiederholungen (z. B. WHO-Broadcasts)

Für die Steuerung unter Linux/WSL wurde zusätzlich ein Bash-Skript (`bash_start.sh`) genutzt, das automatisch Konfiguration und Prozesse startet.

5 Bedienungsanleitung

5.1 Start unter Linux / WSL

Für Linux- und WSL-Umgebungen steht das Startskript `bash_start.sh` zur Verfügung. Es automatisiert die Initialisierung des Systems. Vor dem ersten Ausführen muss das Skript mit dem Befehl `chmod +x bash_start.sh` ausführbar gemacht werden.

1. Die Datei `config_wsl.toml` wird automatisch nach `config.toml` kopiert.
2. Der `broadcast_server.py` wird im Hintergrund gestartet.
3. Drei Clients (`client.py 1`, `2`, `3`) werden im Hintergrund gestartet.
4. In drei separaten Terminals müssen die zugehörigen Kommandozeilen-Schnittstellen manuell ausgeführt werden:
   - `python3 cli.py 1`
   - `python3 cli.py 2`
   - `python3 cli.py 3`

5.2 Beenden und Aufräumen unter Linux

Das Skript `cleanup.sh` terminiert alle laufenden Prozesse (`cli.py`, `client.py`, `broadcast_server.py`) und gibt belegte Ports frei. Auch dieses Skript muss einmalig ausführbar gemacht werden:

- `chmod +x cleanup.sh`
- Ausführung: `./cleanup.sh`

5.3 Start unter Windows (2 Hosts)

Für den Einsatz unter Windows wird die Anwendung auf zwei physische Geräte im selben lokalen Netzwerk verteilt. Auf jedem Host ist eine angepasste `config.toml`-Datei notwendig, die zuvor aus einer Vorlage (z. B. `config_windows.toml`) kopiert wird.

- **Host A** (z. B. `192.168.2.205`): Startet `client.py 1` und `client.py 2` automatisch über eine eigene Batch-Datei. Anschließend werden manuell die zugehörigen CLIs gestartet:
  - `python cli.py 1`
  - `python cli.py 2`
- **Host B** (z. B. `192.168.2.209`): Startet `client.py 3` über eine zweite Batch-Datei. Danach wird CLI 3 manuell gestartet:
  - `python cli.py 3`

Hinweis: Die Windows-Firewall muss eingehende und ausgehende Verbindungen für die verwendeten UDP- und TCP-Ports explizit zulassen. Andernfalls kann die Kommunikation zwischen den Hosts blockiert werden.

5.4 Dual-Konfigurationsarchitektur

Das Projekt unterstützt zwei Betriebsmodi mit jeweils angepasster Netzwerkkonfiguration. Die Startmechanismen und Konfigurationsdateien unterscheiden sich je nach Umgebung:

Umgebung	**Kommunikation (Netzwerk)**	**Startmethode**	**Konfiguration**
WSL/Linux	Localhost (127.0.0.1, UDP/TCP)	`bash_start.sh`	`config_wsl.toml`
Windows (2 Geräte)	IP-Netzwerk (UDP/TCP)	Batch-Dateien + manuelle CLI	`config_windows.toml`

Tabelle 1: Unterstützte Umgebungen und zugehörige Konfiguration

6 Verfügbare CLI-Befehle

- `JOIN` – Chat beitreten
- `MSG <Handle> <Text>` – Textnachricht senden
- `IMG <Handle> <Pfad>` – Bild versenden
- `WHO` – aktive Nutzer ermitteln
- `AWAY`, `BACK` – Abwesenheitsstatus setzen/zurücksetzen
- `LEAVE` – Chat verlassen

- `HANDLE <NeuerName>` – eigenen Namen ändern
- `AUTOREPLY <Text>` – automatische Antwort setzen

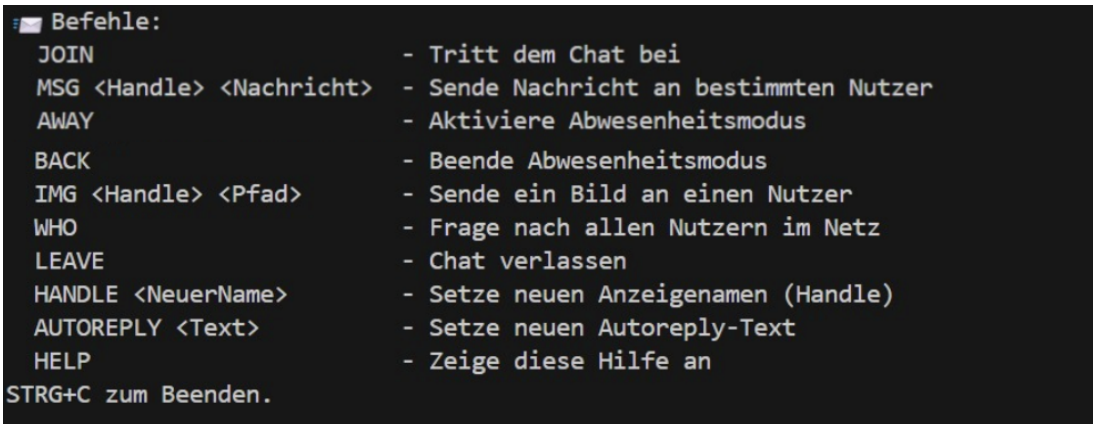

Abbildung 1: Beispielhafter Ablauf von CLI-Kommandos in der Anwendung

7 Systemarchitektur

Die Architektur besteht aus drei logisch getrennten Prozessen:

1. **CLI (cli.py)**: Bietet dem Benutzer eine interaktive Kommandozeile zur Eingabe von Chatbefehlen. Die Verbindung zum zugehörigen Client erfolgt über einen lokalen TCP-Socket (IPC).
2. **Client (client.py)**: Nimmt Nachrichten von der CLI entgegen, verarbeitet sie und übernimmt die gesamte Kommunikation mit anderen Teilnehmern über SLCP (JOIN, MSG, IMG etc.).
3. **Discovery (broadcastserver.py)**: Lauscht auf Broadcast-Port 4000 und verwaltet aktive Teilnehmer. Reagiert auf WHO-Anfragen mit einer KNOWUSERS-Nachricht.

Die Interprozesskommunikation zwischen CLI und Client erfolgt über dedizierte TCP-Verbindungen pro Nutzer (`ipc_port` in `config.toml`). UDP wird zur Nachrichtenübertragung genutzt, TCP für den sicheren Bildaustausch.

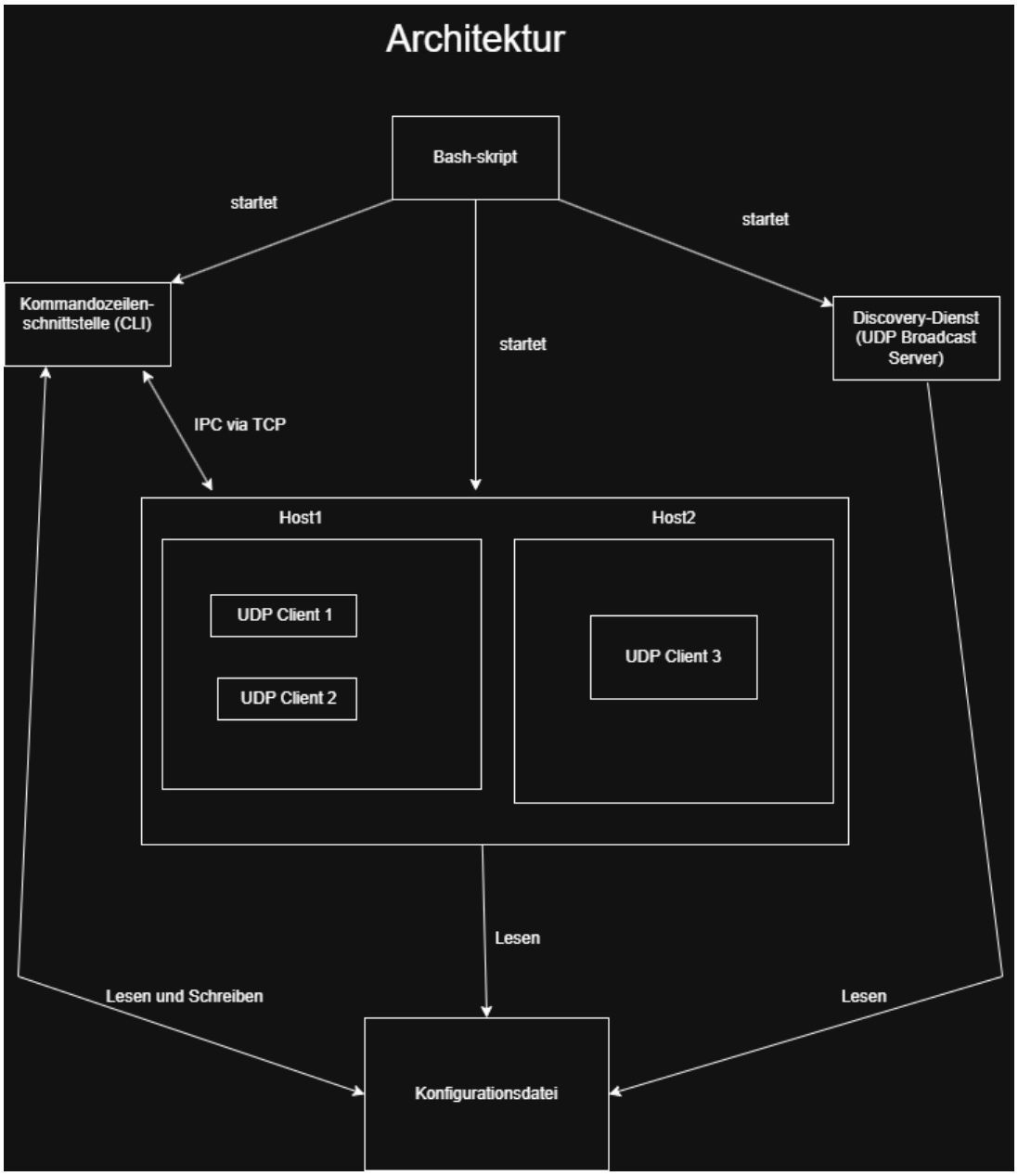

Abbildung 2: Systemarchitektur des Chatprogramms (CLI, Client, Discovery-Service,Netzwerkkommunikation)

8 Kommunikationsprotokoll (SLCP)

- **JOIN**: Registrierung im Netzwerk per Broadcast
- **LEAVE**: Abmeldung per Broadcast und Unicast
- **MSG**: Textnachricht per UDP-Unicast
- **IMG**: Bildnachricht (Header per UDP, Daten per TCP)
- **WHO / KNOWUSERS**: Nutzererkennung per Broadcast + Antwort

8.1 Bildübertragungsprotokoll

Die Übertragung von Bildern erfolgt in einem zweistufigen Verfahren, das die Vorteile beider Protokolle kombiniert:

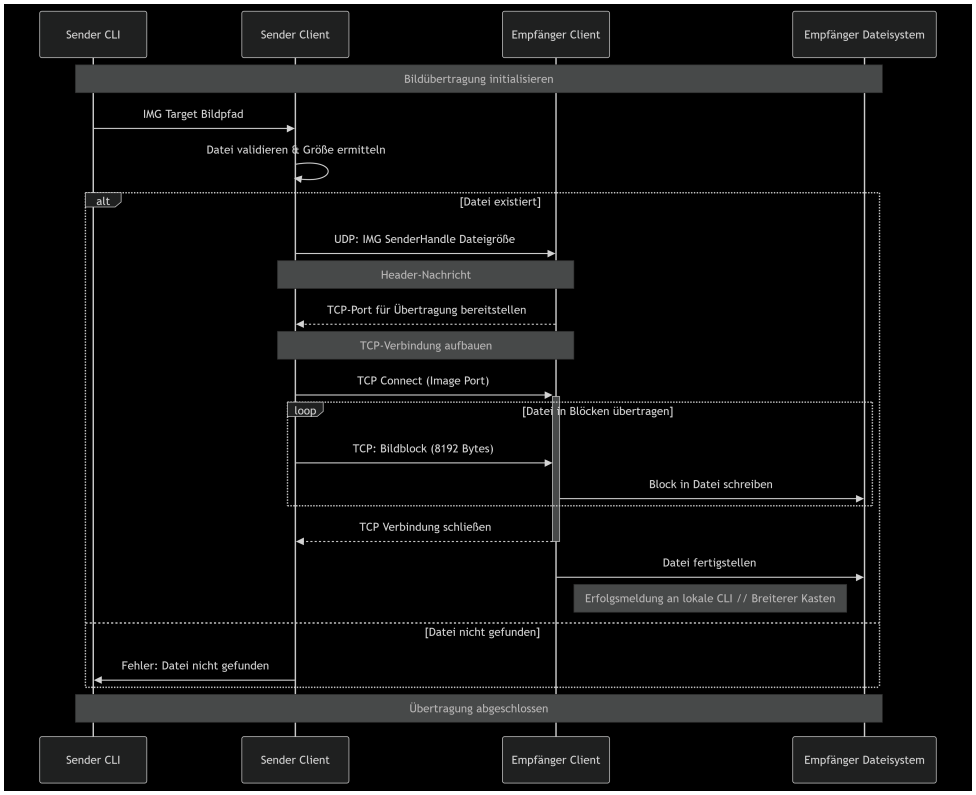

Abbildung 3: Bildübertragung - Header per UDP, Daten per TCP

Zunächst wird über UDP ein Header mit Dateiinformationen gesendet. Anschließend erfolgt die eigentliche Datenübertragung über eine dedizierte TCP-Verbindung, um Datenverluste zu vermeiden.

8.2 SLCP-Protokollablauf

Das folgende Sequenzdiagramm veranschaulicht den typischen Ablauf einer Chat-Session von der Anmeldung bis zur Nachrichtenübertragung. Der Ablauf zeigt die klare Trennung zwischen Discovery-Nachrichten (Broadcast) und direkter Peer-to-Peer-Kommunikation (Unicast). Textnachrichten werden über UDP übertragen, während Bilddaten die Zuverlässigkeit von TCP nutzen.

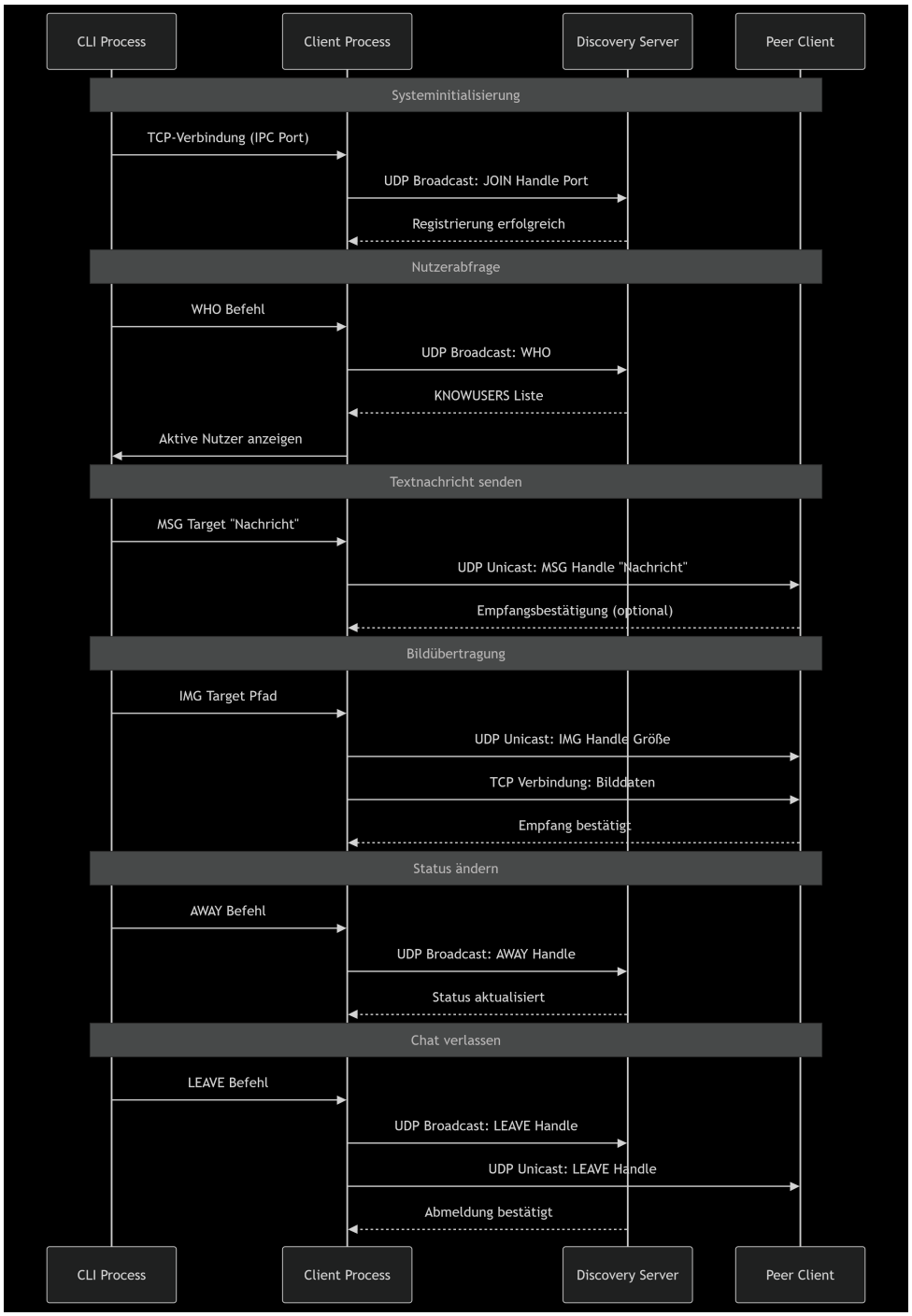

Abbildung 4: SLCP Protokollablauf

9 Konfiguration

Die zentrale Datei `config.toml` enthält folgende Parameter:

- `handle`: Benutzername
- `port`: Lokaler Portbereich
- `whoisport`: Port für Discovery-Service

- `autoreply`: Automatische Antwortnachricht
- `imagepath`: Speicherort für empfangene Bilder

10 Umgesetzte Funktionen

Das entwickelte System erfüllt alle im Projektkontext gestellten funktionalen Anforderungen. Im Folgenden werden die wesentlichen implementierten Funktionen aufgeführt, die das Kommunikationssystem und seine Kernfunktionen darstellen.

- Umsetzung aller SLCP-Kommandos: JOIN, LEAVE, MSG, IMG, WHO, KNOWUSERS, AWAY, BACK
- Discovery über Broadcast und dynamische Nutzerverwaltung
- Echtzeit-Kommunikation über UDP (Text) und TCP (Bild)
- CLI zur Steuerung mit interaktiven Benutzerprompts
- Autoreply-Funktion bei Abwesenheit (AWAY)
- Verwaltung mehrerer Clients über individuell konfigurierbare Ports
- Bildversand und -empfang mit automatischer Dateispeicherung
- Interprozesskommunikation über TCP zwischen CLI und Client
- Laufzeitänderung von Handle und Autoreply-Text via CLI
- Hilfebefehl (HELP) in CLI zur Übersicht aller verfügbaren Kommandos

Die Funktionen bilden die technische Grundlage für die dezentrale Kommunikation zwischen mehreren Peers. Sie wurden modular umgesetzt und sind eindeutig voneinander getrennt.

11 Herausforderungen und Lösungen

Im Verlauf der technischen Umsetzung traten verschiedene Herausforderungen auf, die sowohl konzeptueller als auch praktischer Natur waren. Der folgende Abschnitt beschreibt ausgewählte Probleme und deren jeweilige Lösung im endgültigen Systemdesign.

- **UDP ist nicht verbindungssicher:** Da UDP keine garantierte Zustellung bietet, wurden Bilddaten über TCP übertragen, um Datenverlust zu vermeiden.
- **Nebenläufigkeit und Prozessarchitektur:** Empfang, CLI-Steuerung und Discovery laufen in getrennten Threads, um Reaktionsfähigkeit und Parallelität sicherzustellen.
- **Trennung von CLI und Client:** Ursprünglich erfolgte die Kommunikation zwischen CLI und Client durch direkte Funktionsaufrufe. Zur Umsetzung echter Interprozesskommunikation wurde TCP verwendet.

- **Plattformkompatibilität (WSL vs. Windows):** UNIX-Domain-Sockets funktionieren nicht auf Windows. TCP als IPC ermöglichte identischen Code für beide Betriebssysteme. Konfigurationsdateien wurden für beide Umgebungen angepasst.
- **Port-Kollisionen und Ressourcenfreigabe:** Wiederholte Starts ohne ordnungsgemäßes Beenden führten zu belegten Ports. Das Skript `cleanup.sh` wurde eingeführt, um Prozesse und Ports zuverlässig zu bereinigen.
- **Fehlkonfiguration bei Netzwerkwechsel:** Bei aktiviertem WLAN wurden falsche IP-Adressen übernommen. Erst ein Systemneustart aktualisierte die Netzwerkschnittstellen korrekt.
- **Firewall-Blockaden unter Windows:** Eingehende Verbindungen wurden standardmäßig blockiert. Die Ports mussten manuell in der Windows-Firewall freigegeben werden.
- **Fehlende Rückmeldung bei Netzwerkproblemen:** Verbindungsfehler wurden vom System nicht immer erkannt oder angezeigt. Erst systematische Portanalysen und Neustarts lieferten Klarheit.
- **Discovery-Instabilität bei verzögertem JOIN:** Teilnehmer, die später beitraten, wurden nicht immer erkannt. Die Wiederholung von WHO-Abfragen und eine erweiterte JOIN-Verarbeitung verbesserten die Stabilität.
- **UTF-8 und Sonderzeichenverarbeitung:** Textnachrichten mussten korrekt kodiert und Leerzeichen maskiert oder zitiert werden, um Übertragungsfehler zu vermeiden.
- **Konfigurationsrobustheit:** Bei fehlerhaften oder fehlenden Konfigurationswerten wurden Fallback-Werte gesetzt, um Laufzeitfehler zu vermeiden.

Die aufgeführten Lösungen gewährleisteten die Funktionsfähigkeit des Systems unter verschiedenen Bedingungen und ermöglichten eine robuste, plattformunabhängige Ausführung.

12 Erweiterungspotenzial

Das entwickelte System ist modular aufgebaut und bietet damit eine solide Grundlage für zukünftige Erweiterungen. Im Folgenden sind mögliche Weiterentwicklungen aufgeführt, die die Funktionalität und Benutzerfreundlichkeit des Programms verbessern würden:

- **Gruppenchats und Broadcast-Nachrichten:** Möglichkeit, Nachrichten gleichzeitig an mehrere Nutzer zu senden.
- **GUI-Frontend:** Ein grafisches Interface (z. B. mit `tkinter` oder `PyQt`) würde die Bedienung für weniger technisch versierte Nutzer erleichtern.
- **Verschlüsselung:** Absicherung der Kommunikation mittels symmetrischer oder asymmetrischer Verschlüsselungsverfahren (z. B. mit `cryptography`).
- **Persistente Chat-Historie:** Speicherung von Nachrichtenverläufen lokal oder in einer Datei, um Nachrichten auch nach einem Neustart wieder einsehen zu können.

- **Benutzerstatus in Echtzeit:** Erweiterung des Discovery-Dienstes zur Übertragung von Online-, AFK- und Offline-Status.
- **Dateitransfer allgemein:** Neben Bildern könnten auch andere Dateitypen übertragen werden (z. B. PDFs oder Textdateien).
- **Benachrichtigungssystem:** Akustische oder visuelle Hinweise bei neuen Nachrichten.

Diese Vorschläge zeigen, dass das Chatprogramm nicht nur als studentisches Projekt funktioniert, sondern als technische Basis für ein vollwertiges Kommunikationssystem dienen kann.

13 Fazit

Das Projekt zeigt, wie sich ein vollständig dezentrales Chatprogramm mit modularer Architektur, Interprozesskommunikation und eigener Protokollumsetzung in Python realisieren lässt. Neben Textnachrichten wurde auch eine stabile Bildübertragung implementiert. Die klare Trennung in Prozesse und der flexible Aufbau der Konfiguration ermöglichen einfache Erweiterbarkeit und gute Wartbarkeit. Die Teamarbeit verlief kollaborativ und iterativ – jedes Mitglied trug zur Gesamtumsetzung bei.