Version 2 – Projekt mit IPC – Branches „main“ und „IPC_P2P“ (GitHub)

Modul: Betriebssysteme und Rechnernetze | bei Prof. Dr. Oliver Hahm
bis zum 22.06.2025 | Studiengang: Wirtschaftsinformatik (B.Sc.)

BSRN-Projektdokumentation

Peer-to-Peer Chat-Programm

Von: **Ani Manukjan** (1587472), **Luca Müller** (11577156),
Sohal Haidari (1583432), **Sumaya Mohamed** (1578546)

Inhaltsverzeichnis

Unser Ansatz

Für die Umsetzung der Client-seitigen Chat-Anwendung haben wir uns für die Programmiersprache Python entschieden. Sie zeichnet sich besonders durch die simple Syntax und Code-Struktur aus, sowie die einfache Möglichkeit anhand von libraries (Bibliotheken) Command Line Interfaces (CLI) und Graphical User Interfaces (GUI) zu gestalten. Als Entwicklungsumgebung bot sich die für uns vertraute IDE „Visual Studio Code“ von Microsoft an.

Zuerst hatten wir eine Version des Projekts angefertigt, welches durch Dateien modularisiert ist, die in einem gemeinsamen Prozess, aufgeteilt durch Daemons, gemeinsam laufen.
Gegen Ende der Bearbeitungszeit haben wir dann eine weitere Version des Projekts angelegt, welche auf echt Inter Process Communication (IPC) setzt, und ebenfalls über eine grafische Benutzeroberfläche (GUI) verfügt.

In diesem Dokument finden Sie die Dokumentation für die zweite Variante des Projekts, die Version mit richtiger IPC.
Die Dokumentation für die Version ohne IPC finden Sie in einem separaten Dokument.

Zielsetzung

Das Ziel dieses Projekts ist die Entwicklung eines vollständig dezentralisierten Peer-to-Peer-Chatprogramms, das ohne zentrale Serverinstanz auskommt. Im Mittelpunkt steht die Umsetzung eines benutzerfreundlichen Chat-Systems, das sowohl Textnachrichten als auch Bildübertragungen in lokalen Netzwerken unterstützt.

Dabei sollen folgende Anforderungen erfüllt werden:

- Implementierung des textbasierten SLCP-Protokolls (Simple Local Chat Protocol), das auf UDP-Broadcast und TCP-Direktverbindungen basiert
- Verwendung von Interprozesskommunikation (IPC) zur Trennung und Synchronisation der drei Hauptkomponenten (CLI, Network, Discovery)
- Nutzung eines Konfigurationsmechanismus über eine TOML-Datei, um Benutzerparameter wie Ports, Handle und Bildspeicherpfade flexibel verwalten zu können
- Automatische Nutzererkennung über UDP-Discovery-Dienste mit Antwortlogik
- Reaktion auf Nutzerkommandos in Echtzeit mit umgehender Anzeige eingehender Inhalte und speicherbarer Bildübertragung

Das Projekt soll modular, nachvollziehbar und dokumentiert aufgebaut sein und durch saubere Prozessstrukturierung (Multiprozess-Design) nachvollziehbar zwischen Benutzeroberfläche, Netzwerkanbindung und Discovery-Dienst trennen.

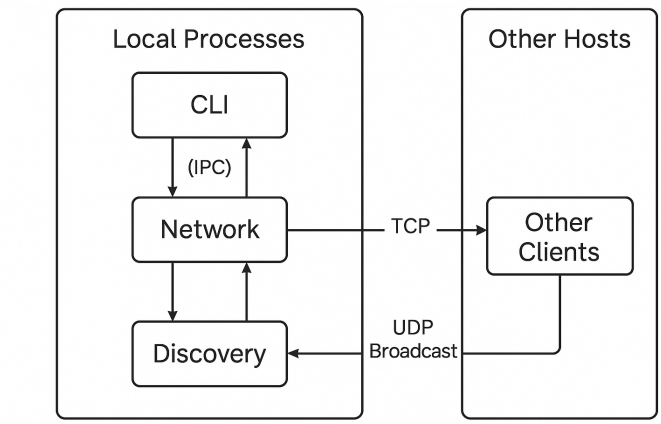
Systemarchitektur

Die Architektur des Systems basiert auf der Trennung von Zuständigkeiten in drei eigenständige Prozesse. Jeder dieser Prozesse übernimmt eine klar definierte Rolle und kommuniziert mit den anderen über Interprozesskommunikation (IPC). Ziel ist es, eine modulare, skalierbare und nachvollziehbare Struktur zu schaffen, die sich für verteilte Kommunikation eignet.

Komponente	Funktion
CLI	Kommandozeilen-Schnittstelle: Entgegennahme von Benutzereingaben, Anzeige von empfangenen Nachrichten, Auslösen von Aktionen (z. B. WHO, MSG, IMG, LEAVE)
Network	Zuständig für die eigentliche Kommunikation über das Netzwerk mittels TCP (Nachrichten, Bilder). Reagiert auf CLI-Kommandos und empfängt Daten von anderen Clients.
Discovery	Verwaltet das Teilnehmerverzeichnis und reagiert auf JOIN/LEAVE/WHO-Nachrichten über UDP-Broadcast. Gibt bekannte Benutzer an neue Clients weiter (KNOWNUSERS).

Erklärung

- **CLI** ist die Benutzerschnittstelle, nimmt Eingaben wie MSG, IMG, WHO, LEAVE entgegen
- **Network** empfängt diese Kommandos über IPC und entscheidet, ob eine TCP-Verbindung zu einem Peer geöffnet wird
- **Discovery** verarbeitet JOIN, WHO, LEAVE und verteilt Informationen über bekannte Clients
- TCP wird für gezielte Kommunikation (z. B. MSG, IMG) verwendet
- UDP-Broadcast wird zur verteilten Discovery genutzt
- Die Prozesse kommunizieren lokal über Queues/Pipes, externe Kommunikation erfolgt über das Netzwerk



IPC-Mechanismus

- CLI ←→ Network: Über Queues werden Benutzerbefehle und Rückmeldungen ausgetauscht.
- CLI ←→ Discovery: Über Queues werden WHO-Anfragen oder Statusaktualisierungen koordiniert.

Startlogik

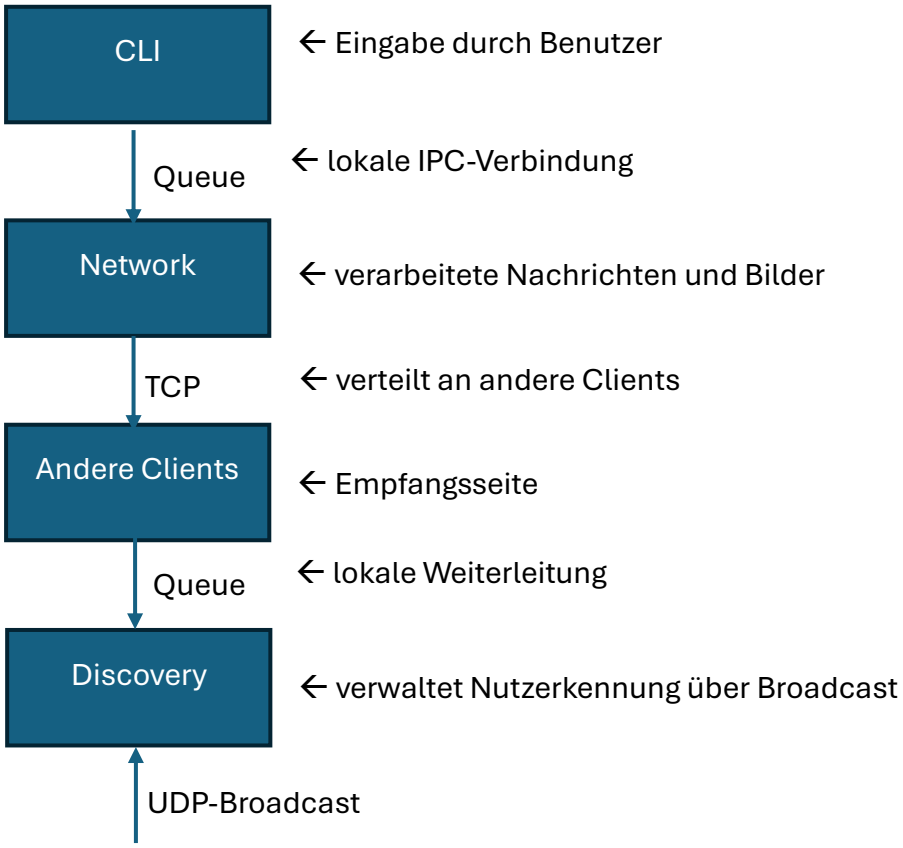
- Beim Start prüft das Hauptprogramm, ob bereits ein Discovery-Prozess läuft. Falls nicht, wird dieser gestartet.
- Jeder Benutzer startet also mindestens CLI und Network. Der Discovery-Dienst läuft nur einmal pro Rechner.

Diese Architektur erfüllt alle Projektanforderungen: klare Trennung, saubere Kommunikation, nur ein Discovery-Dienst, einfache Erweiterbarkeit. Verwaltet Teilnehmer im Netzwerk mithilfe von Broadcast (UDP) |

Die Prozesse werden über Pipes oder Message Queues miteinander verbunden. Die Architektur ermöglicht, dass bei Mehrfachstart auf demselben Rechner nur ein Discovery-Dienst aktiv ist.

Kommunikationsfluss

Dieses Diagramm zeigt die Architektur und den Datenfluss des Chat-Programms zwischen den drei Hauptkomponenten:



Modulbeschreibung

Datei: main.py

load_config()

Diese Funktion prüft, ob eine Datei unter config/config.toml existiert. Falls ja, wird sie geöffnet und der Inhalt mit der toml-Bibliothek eingelesen und als Dictionary zurückgegeben. Falls nicht, wird ein leeres Dictionary erzeugt. Diese Konfiguration enthält z. B. Handle, Port oder Speicherpfade.

save_config(cfg)

Diese Funktion speichert das übergebene Dictionary cfg wieder in der Datei config.toml. Es wird dafür die Bibliothek toml verwendet, um das Dictionary korrekt zu serialisieren. Die Datei wird im Schreibmodus ("w") geöffnet, wodurch ältere Inhalte überschrieben werden.

prompt_missing_config(cfg)

Diese Funktion prüft, ob bestimmte Schlüsseleinträge (handle, port, whoisport) im Dictionary cfg fehlen. Wenn ja, werden diese über die Standard-Eingabe vom Nutzer abgefragt (z. B. input("Wie heißt du?")) und in das Dictionary geschrieben. Die Eingabe von Port erfolgt als int, der Rest als String.

is_discovery_running()

Diese Funktion prüft, ob die Datei discovery.lock im Projektverzeichnis existiert. Sie dient als einfacher Mechanismus (Lockfile), um zu erkennen, ob ein Discovery-Prozess bereits läuft. Rückgabe ist True, wenn Datei existiert, sonst False.

start_discovery(handle, whois_port)

Diese Funktion startet einen neuen Subprozess für den Discovery-Dienst, aber nur, wenn noch keiner aktiv ist (Lockfile-Check). Der Prozess erhält handle und whois_port als Argumente. Danach wird sofort die Lockfile erstellt, um Doppelausführung zu verhindern.

main()

Dies ist die Hauptfunktion des Programms. Sie lädt zuerst die Konfiguration (oder fragt fehlende Angaben ab), prüft, ob Discovery läuft und startet die drei Prozesse: CLI, Network und optional Discovery. Alle Prozesse laufen parallel (via Process(...).start()). Ein try/except-Block sorgt dafür, dass beim Abbruch (z. B. STRG+C) alle Prozesse beendet und die Lockfile gelöscht werden.

Datei: core/cli.py

__init__(self, username, to_net, to_disc, from_net, from_disc, config)

Diese Methode initialisiert die CLI-Klasse und nimmt alle erforderlichen Kommunikationsschnittstellen entgegen:

- username: Handle des aktuellen Benutzers.
- to_net: Queue für Nachrichten, die von der CLI an die Network-Komponente gehen.
- to_disc: Queue für Nachrichten an den Discovery-Prozess (z. B. WHO-Anfragen).
- from_net: Eingehende Nachrichten, die vom Network-Prozess an die CLI geliefert werden (z. B. empfangene Nachrichten).
- from_disc: Antworten vom Discovery-Prozess (z. B. KNOWNUSERS).
- config: Konfigurationsobjekt (aus der TOML-Datei geladen).
  Alle Parameter werden gespeichert, um innerhalb der Klasse verwendet zu werden.

run(self)

Diese Methode startet den CLI-Hauptprozess:

- Zuerst wird ein separater Thread für _network_listener() gestartet, um eingehende Nachrichten im Hintergrund zu verarbeiten.
- Danach beginnt eine Eingabeschleife mit input(), die Benutzerkommandos entgegennimmt.
- Je nach eingegebenem Befehl wird der Text analysiert und in strukturierte Kommandos zerlegt (MSG, IMG, WHO, LEAVE etc.).
- Diese Kommandos werden dann über die Queues an Network oder Discovery gesendet.
- Eingabefehler (z. B. unvollständige Parameter) werden durch kurze Rückmeldungen an den Benutzer abgefangen.

_network_listener(self)

Ein Hintergrundthread, der kontinuierlich auf neue Nachrichten in from_net prüft:

- Solange Nachrichten ankommen, werden diese direkt mit print() in der Konsole angezeigt.
- Wird häufig verwendet, um Nachrichten, Bilder oder Fehlermeldungen auszugeben.
- Da der Hauptthread mit Benutzereingabe blockiert ist, läuft dieser Listener unabhängig parallel.

Datei: core/discovery.py

__init__(self, in_queue, out_queue, imagepath)

Diese Methode initialisiert die Discovery-Klasse:

- in_queue: Eine Queue, über die die CLI-Befehle wie WHO, JOIN, LEAVE ankommen.
- out_queue: Queue, in die Antworten oder Ereignisse (wie neue Benutzer) zurück an die CLI gesendet werden.
- imagepath: Speicherort für empfangene Bilddateien, wird an andere Komponenten weitergegeben.
  Diese Parameter werden in Instanzvariablen gespeichert. Der Discovery-Prozess benötigt sie, um Nachrichten weiterleiten und beantworten zu können.

run(self)

Diese Methode ist die zentrale Ausführungseinheit des Discovery-Prozesses:

- Öffnet einen UDP-Socket, der auf eingehende SLCP-Nachrichten wartet (JOIN, WHO, LEAVE).

- Läuft in einer Schleife, um Nachrichten kontinuierlich zu empfangen und zu verarbeiten.
- Nachrichten vom Socket werden gelesen, dekodiert und analysiert.
- Ein JOIN-Ereignis wird registriert und ggf. per KNOWNUSERS beantwortet.
- WHO-Anfragen von anderen Clients werden erkannt und führen zu Antworten über UDP-Unicast.
- LEAVE-Nachrichten werden erkannt und führen zur Löschung des zugehörigen Clients aus der internen Liste.
- Gleichzeitig wird geprüft, ob neue Befehle aus in_queue anliegen und dann entsprechend verarbeitet.
- Die gesammelten Informationen über bekannte Nutzer werden über out_queue an die CLI übergeben, damit diese eine Nutzerauswahl anzeigen kann.

Datei: core/network.py

__init__(self, username, port, in_q, out_q, to_disc, from_disc, config)

Diese Methode initialisiert die Netzwerkkomponente und bereitet alle Ressourcen für Nachrichtenübertragung und Discovery-Kommunikation vor:

- username: Eigener Benutzername (z. B. "Alice")
- port: Der Port, über den TCP-Kommunikation (für Nachrichten und Bilder) stattfindet
- in_q: Eingehende Befehle von der CLI (z. B. MSG, IMG)
- out_q: Rückmeldungen an die CLI (z. B. Fehler, Bestätigungen)
- to_disc: Queue zur Kommunikation mit dem Discovery-Dienst
- from_disc: Empfangs-Queue für Antworten vom Discovery-Prozess
- config: Konfigurationsparameter wie autoreply, whoisport, imagepath

Zusätzlich wird ein UDP-Socket eingerichtet:

- self.udp: Wird auf dem whoisport geöffnet und für SLCP-Broadcast-Nachrichten verwendet
- Der Socket wird nicht-blockierend gesetzt, sodass er in der Hauptschleife nicht stört

start_tcp_image_server(self)

Diese Methode startet einen Hintergrundthread, der einen TCP-Server auf dem Port self.port + 100 öffnet:

- Dieser TCP-Server wartet auf Verbindungen von anderen Clients, die Bilddaten senden möchten
- Für jede eingehende Verbindung wird die Methode handler() genutzt, um die Bilddaten zu verarbeiten

- Dadurch ist die Netzwerk-Komponente in der Lage, Bilder asynchron zu empfangen und zu speichern

run(self)

Diese Methode stellt die zentrale Verarbeitungseinheit dar. Sie läuft in einer Endlosschleife und übernimmt:

1. Empfang von Benutzerbefehlen aus in_q
    a. MSG <Handle> <Text>: Nachricht an bekannten Benutzer senden (via TCP)
    b. IMG <Handle> <Pfad>: Bilddatei öffnen, in Bytes konvertieren und an Ziel senden
2. Empfang von Discovery-Antworten aus from_disc (z. B. KNOWNUSERS)
    a. Aktualisierung lokaler Peer-Liste
3. Überwachung des UDP-Sockets
    a. Für eingehende WHO-Broadcasts kann auf Wunsch geantwortet werden

Alle Antworten, Fehler und Statusmeldungen werden über out_q an die CLI geschickt.

handler()

Dies ist die Methode, die vom TCP-Server beim Empfang eines Bildes verwendet wird:

- Öffnet einen TCP-Socket und akzeptiert eingehende Verbindungen
- Liest die Rohdaten (Bytes) des Bildes, basierend auf der in der IMG-Nachricht angekündigten Größe
- Übergibt die Daten an image_handler.save_and_open_image(), um das Bild zu speichern
- Die Methode läuft typischerweise als Thread im Hintergrund, getrennt vom Hauptprozess

---

Datei: core/image_handler.py

save_and_open_image(sender, binary_data, imagepath)

Speichert ein empfangenes Bild unter Angabe des Absenders im angegebenen Verzeichnis. Optional wird das Bild direkt geöffnet.

load_image_as_bytes(path)

Liest eine Bilddatei vom angegebenen Pfad ein und gibt deren Byte-Inhalt zurück.

chunk_image_data(b64_string, max_size)

Zerteilt einen base64-kodierten String in Blöcke der Größe max_size, um sie über das Netzwerk senden zu können.

Konfigurationsdatei (config.toml)

Handle	= Benutzername
Port	= Eigener Port für TCP-Kommunikation
Whoisport	= UDP-Port für Broadcasts
Autoreply	= Automatische Antwort bei Abwesenheit
Imagepath	= Pfad zum Speichern empfangener Bilder

Beispielablauf

In diesem erweiterten Szenario interagieren zwei Benutzer – Ani und Luca – im selben lokalen Netzwerk:

1. Ani startet das Programm, gibt als Handle "Ani" ein, wählt Port 5001 und WHOIS-Port 4000. Sie sendet automatisch:
2. JOIN Ani 5001

→ Die Discovery-Komponente registriert Ani und antwortet neuen WHO-Anfragen mit ihrer Adresse.

3. Luca startet das Programm, gibt "Luca" ein, wählt Port 5002. Auch er sendet:
4. JOIN Luca 5002

→ Discovery fügt auch Luca hinzu.

5. Luca möchte wissen, wer online ist, und gibt ein:
6. WHO

→ Die Discovery-Komponente antwortet mit:

KNOWNUSERS Ani 192.168.0.10 5001, Luca 192.168.0.11 5002

→ Die CLI zeigt die Teilnehmerliste an.

7. Ani sendet eine Textnachricht an Luca:
8. MSG Luca Hallo Luca!

→ Die Nachricht wird über TCP direkt an Lucas IP und Port übertragen. Luca sieht sofort:

[Ani] Hallo Luca!

9. Luca antwortet:
10. MSG Ani Hey Ani, alles klar?
11. Ani sendet ein Bild:
12. IMG Luca cat.jpg

→ Ani wählt das Bild cat.jpg. Die Datei wird gelesen, die Größe berechnet und folgendes gesendet:

IMG Luca 15432

→ Danach folgen genau 15432 Bytes als Bilddaten. Luca speichert das Bild und öffnet es automatisch mit dem Standardprogramm.

13. Luca geht offline und sendet:
14. LEAVE Luca

→ Discovery entfernt Luca aus der Liste. Ani erhält optional eine Benachrichtigung.

15. Ani gibt erneut WHO ein und sieht nun:
16. KNOWNUSERS Ani 192.168.0.10 5001

IPC- und TCP-Datenfluss

Interprozesskommunikation (IPC)

Für die Kommunikation zwischen den lokalen Prozessen (CLI, Network, Discovery) wird Interprozesskommunikation verwendet. Konkret werden unter Windows Named Pipes und unter Linux/macOS Unix Domain Sockets genutzt.

- **CLI → Network**: Benutzerbefehle (z. B. MSG, IMG) werden über eine Queue an Network gesendet
- **CLI → Discovery**: Anfragen wie WHO oder LEAVE gehen über IPC ebenfalls an Discovery
- **Network → CLI**: Statusnachrichten, Fehler, eingehende Nachrichten werden zurück zur CLI geschickt
- **Discovery → CLI**: Reaktionen wie KNOWNUSERS werden zurückgegeben

Diese Kommunikation ist vollständig lokal und synchronisiert die drei Teilprozesse.

Netzwerkkommunikation (TCP/UDP)

Nachrichtenversand (TCP)

- TCP wird verwendet, um Bilder (IMG) gezielt an andere Clients zu senden
- Die Network-Komponente öffnet dazu eine Verbindung zum Zielport des Empfängers
- Der Empfänger besitzt einen eigenen TCP-Server, der Bilder lauscht (standardmäßig port + 100)

Discovery (UDP-Broadcast)

- Der Discovery-Prozess verwendet UDP-Broadcast zur Netzwerkerkennung
- JOIN-, LEAVE- und WHO-Nachrichten werden per Broadcast an alle im Netzwerk gesendet (Port whoisport)

- Die Antwort KNOWNUSERS wird per UDP-Unicast zurück an den anfragenden Client gesendet
- Auch Nachrichten (MSG) zwischen den Nutzern werden aufgrund ihrer Unkompliziertheit und Schnelligkeit per UDP-Unicast umgesetzt.

Gesamtübersicht

Diese Architektur erlaubt es, Bilder gezielt über TCP zu verschicken, und Nachrichten auch gezielt über UDP-Unicasts zu versenden, während Discovery automatisch alle Clients im Netzwerk auf UDP-Broadcast-Basis erkennt.
Kommunikation erfolgt über Named Pipes (Windows) oder Unix Domain Sockets (Linux/macOS).
Eine Registry-Datei sorgt dafür, dass pro Rechner nur ein Discovery-Prozess läuft.

Herausforderungen und Probleme

Unicast vs. Broadcast

Problem:

Zu Projektbeginn erfolgte der Versand aller Nachrichten im Netzwerk über einen Broadcast. Das hatte zur Folge, dass jede einzelne Nachricht an alle Netzwerkteilnehmer gesendet wurde, unabhängig davon, ob sie für sie bestimmt war oder nicht. Durch den Umstand, dass deshalb jeder Client jede Nachricht empfängt und verarbeiten muss, kam es zu überflüssigem Datenverkehr und möglichen Performance-Problemen.

Lösung:

Wir haben den Code so modifiziert, dass bestimmte Nachrichten, vor allem direkte Textnachrichten (MSG), ausschließlich per Unicast an den jeweiligen Ziel-Client gesendet werden, Dies garantiert, dass nur der wirklich angesprochene Empfänger die Nachricht bekommt. Außerdem reduziert es den Datenverkehr und steigert die Effizienz des Systems.

Implementierung der Autoreply-Funktion

Problem

Wir wollten sicherstellen, dass die Einstellung der Konfigurationsdatei, ob der Nutzer gerade aktiv ist, oder eine automatische Antwort verschicken werden soll, direkt über die Konsole geändert werden kann.

Das Problem hierbei war, dass der Nutzer in den Projektordner in die Konfigurationsdatei gehen musste, um seine Inaktivität einzuschalten und auszuschalten.

Dies ist nicht benutzerfreundlich.

Lösung

Um dieses Problem zu lösen, haben wir eine „CONFIG inactive ON/OFF“-Funktion implementiert. Diese Funktion ermöglicht es, den Inaktivitätsstatus direkt während des laufenden Betriebs zu ändern. Ist der Inaktivitätsmodus aktiviert, wird automatisch eine vordefinierte Antwort an den Absender der Nachricht gesendet. Dadurch bleibt der Nutzer auch in seiner Abwesenheit kommunikativ präsent und informiert mögliche Chatpartner über seine Abwesenheit.

Implementierung der grafischen Benutzeroberfläche

Problem

Bei der Implementierung der grafischen Benutzeroberfläche gab es das Problem, dass mit der Library „PyQT5“ keine UDP-Kommunikation umgesetzt werden konnte.

Bedeutet, dass über die grafische Oberfläche nur zwischen zwei Clients auf dem gleichen Rechner kommuniziert werden kann.

Auch gab es hier Probleme beim Versenden und Empfangen von Textnachrichten und Bildern über die grafische Benutzeroberfläche.

Zeitweise wurden Bilder zurück an den Sender verschickt, und Nachrichten an sowohl den Empfänger als auch den Sender übermittelt

Lösung

Durch Nutzung einer andere Library „tkinter“ war es deutlich leichter, einer Darstellung der UDP-Chats einzurichten.

Auch konnten wir durch weiteres Probieren und Verbessern die anderen Probleme in Verbindung mit dem GUI lösen.

Die grafische Benutzeroberfläche sieht durch die andere Library zwar anders aus, erfüllt funktional aber besser ihren Zweck.