

# ***P2P-Chatclient – Dokumentation***

## ***Architektur, Komponentenbeschreibung & Benutzeranleitung***

*Ein lokalbasierter Peer-to-Peer-Messenger mit Text- und  
Bildübertragung*

***Gruppe: A3***

***Autoren:***

*Maximilian Herman, Marius Schleiff, Jan Fabiunke,  
Anton List und Simon Göring*

***Datum: 20.06.2025***

# *Inhaltsverzeichnis*

<b>1</b>	<b>Einleitung .....</b>	<b>3</b>
<b>2</b>	<b>Ansatz &amp; Architektur .....</b>	<b>4</b>
2.1	Ablauf „Nachricht senden“ .....	4
2.2	Schlüsselentscheidungen .....	5
<b>3</b>	<b>Problemstellungen .....</b>	<b>6</b>
<b>4</b>	<b>Modul- und Klassenübersicht.....</b>	<b>9</b>
4.1	Client-Klasse .....	9
4.2	Server-Klasse.....	10
4.3	Discovery-Service-Klasse .....	11
4.4	SLCP-Handler-Klasse.....	12
4.5	Kommandozeilen-Interface .....	13
4.6	Grafisches-Interface .....	14
4.7	Hauptmodul.....	15
<b>5</b>	<b>Konfiguration: config.toml.....</b>	<b>16</b>
<b>6</b>	<b>Bedienungsanleitung .....</b>	<b>17</b>
6.1	Vorbereitung.....	17
6.2	Benutzung mit GUI.....	17
6.3	Benutzung mit CLI.....	19

# 1 Einleitung

Diese Dokumentation beschreibt die Architektur, Funktionsweise und Bedienung eines selbst entwickelten **P2P-Chatclients**, der Text- sowie Bildnachrichten ausschließlich innerhalb eines lokalen Netzwerks austauscht und dabei vollständig auf zentrale Server verzichtet.

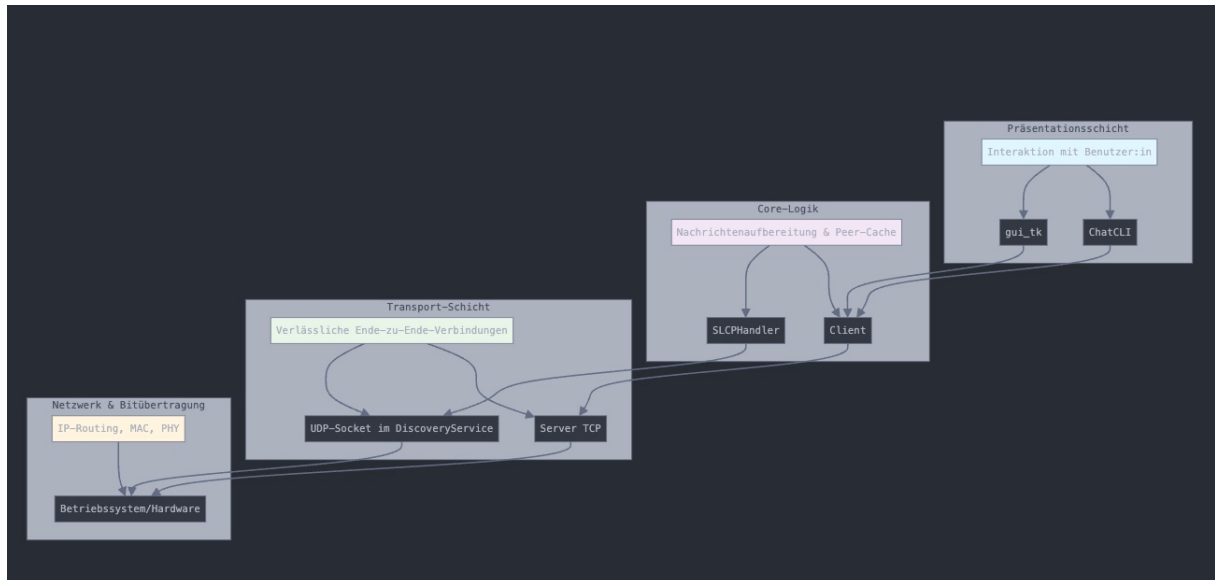
Der Client basiert auf dem **SLCP-Protokoll (Simple Local Chat Protocol)**, welches speziell für die strukturierte Übertragung von Nachrichten in dezentralen Umgebungen entworfen wurde. Die Anwendung ist strikt modular aufgebaut. **Peer-Discovery per UDP-Broadcast, zuverlässiger Nachrichten- und Dateitransfer über TCP, eine klar getrennte Kernlogik sowie zwei Anwendungsschichten (CLI und GUI).**

Ziel dieses Projekts ist es, eine robuste, erweiterbare und leicht nachvollziehbare Kommunikationsplattform für geschlossene Gruppennetzwerke zu schaffen.

Diese Dokumentation richtet sich an Lehrende und Projektmitarbeitende, die Aufbau, Logik und Bedienung des Systems im Detail verstehen wollen und bei Bedarf den Client selbst erweitern möchten. Neben der Architekturbeschreibung enthält sie eine vollständig nachvollziehbare Schritt für Schritt Anleitung zur praktischen Nutzung.

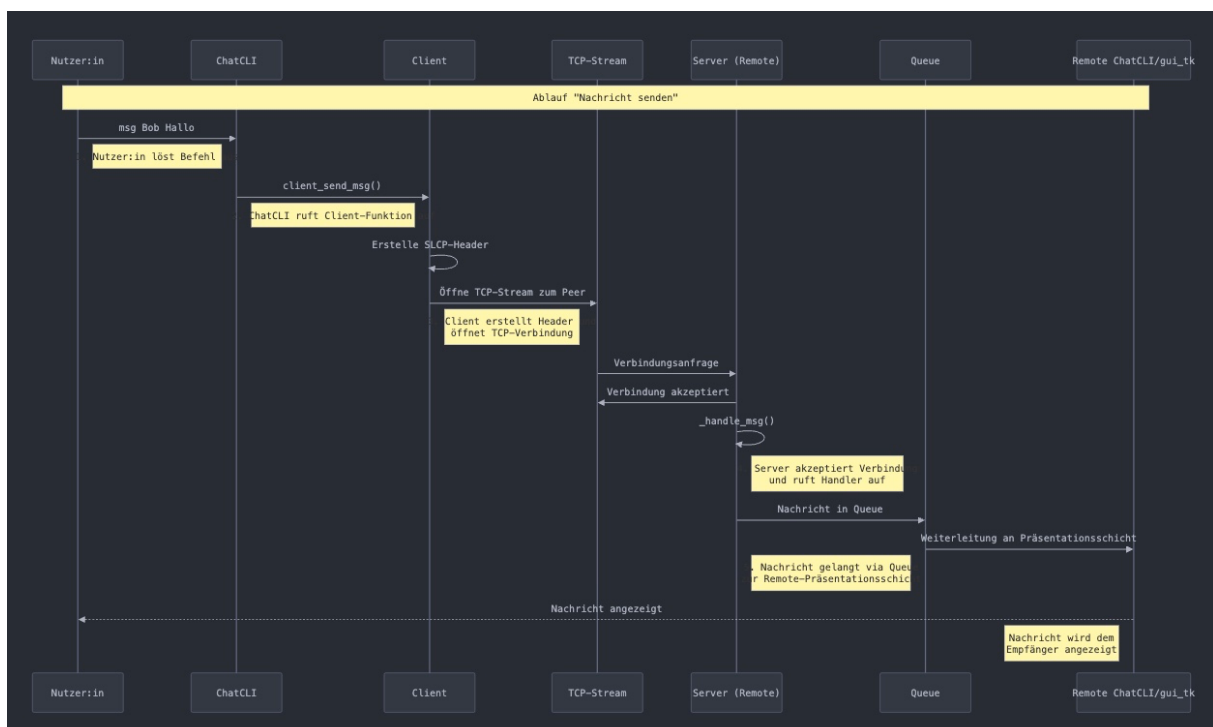
## 2 Ansatz & Architektur

→ Der Messenger folgt einem Layer-Ansatz, welcher Zuständigkeiten klar trennt.



→ Die Layer kommunizieren ausschließlich nach unten (Dependency-Inversion).

### 2.1 Ablauf „Nachricht senden“



## 2.2 Schlüsselentscheidungen

Im folgenden Abschnitt werden die wichtigsten Architektur- und Implementierungsentscheidungen aufgelistet, die den Projektverlauf maßgeblich beeinflusst haben und das Zusammenspiel der Komponenten bestimmen.

Teilproblem	Lösung
<b>Zuverlässigkeit</b>	Nutzdaten ausschließlich über TCP; UDP nur für Discovery.
<b>Netzlast</b>	WHOIS-Broadcast auf 30 s gedrosselt; Peer-TTL 120 s.
<b>Thread-Safety</b>	Gemeinsame <code>multiprocessing.Queue</code> , regelmäßiges Polling mit 250 ms (GUI) bzw. 500 ms (CLI).
<b>Port-Konflikte</b>	Dynamische Port-Suche bei Belegung.
<b>Plattform-Pfadtrennung</b>	<code>pathlib.Path</code> in allen Dateioperationen.
<b>GUI-Framework</b>	Wechsel von PyQt5 zu Tkinter (keine externen Binaries).

## 3 Problemstellungen

### 1. Peer-Discovery ohne Broadcast-Sturm

→ Problem: Periodische WHOIS-Broadcasts können in großen Subnetzen zu Netzlast und Paketverlusten führen.

→ Lösung: Der Discovery-Dienst verschickt WHOIS nur alle 30 Sekunden und ignoriert Duplikate anhand des IP-Ports. Jeder empfangene IAM-Datensatz aktualisiert einer Peer-Cache mit Zeitstempel, sodass veraltete Einträge nach 120 Sekunden verworfen werden. So bleibt die Zahl der Broadcast-Pakete konstant und gering.

### 2. Zuverlässiger Dateientransfer

→ Problem: UDP eignet sich nicht für große Binärblöcke wie Bilder oder verlässliche Textzustellung.

→ Lösung: UDP wird nur für Discovery-Pakete verwendet. Sämtliche Nutzdaten laufen über einen zuverlässigen TCP-Stream. Bei Bildern sendet der Client zuerst einen IMG-Header mit Dateigröße und SHA-256-Hash, anschließend folgt der Binärstream. Der Server verifiziert die Prüfsumme und legt die Datei im definierten `imagepath` ab.

### 3. Thread-Synchronisation und Graceful Shutdown

→ Problem: Server-, Discovery- und GUI/CLI-Threads dürfen sich nicht gegenseitig blockieren und müssen sauber terminieren

→ Lösung: Alle Nebenläufigkeit erfolgt über `threading.thread` mit nicht blockierenden `multiprocessing.Queue`-Objekten. Jeder Thread prüft zyklisch ein `shutdown_event`. Sockets besitzen ein 1-Sekunden Timeout, damit `KeyboardInterrupt` oder GUI-Quit schnell möglich ist.

#### **4. GUI ↔ Core-Kupplung**

→ Problem: Die erste Fassung nutzte einen IPC-Handler-Prozess, Wartung und Debugging erwiesen sich als umständlich.

→ Lösung: IPC-Layer wurde entfernt. Die GUI (Tkinter) legt dieselben Queues an wie die CLI und tauscht Ereignisse direkt mit Client und Server aus. Dadurch entfällt Serialisierung über Pipes und die Codebasis bleibt schlanker.

#### **5. Automatische Abwesenheitsantwort ohne Ping-Pong**

→ Problem: Zwei abwesende Peers können sich wechselseitig mit Auto-Replies überfluten.

→ Lösung: Der Client speichert für jedes Gegenüber eine „autoreply sent“. Während eines Away-Zyklus wird pro Absender exakt eine Auto-Antwort generiert, beim Zurücksetzen (Back-Befehl) werden alle Flaggen zurückgesetzt.

#### **6. Port-Belegungen und Firewall-Hürden**

→ Problem: Standard-Ports können bereits durch andere Dienste blockiert sein.

→ Lösung: Beim Start prüft `main.py`, ob `port` und `whoisport` frei sind. Andernfalls wird der nächstfreie Port gesucht und dem Benutzer in Log-Form angezeigt.

#### **7. Konfigurations-Konsistenz**

→ Problem: Inhomogene Konfigurationen führen zu Verbindungsfehlern, etwa wenn Peers unterschiedliche Broadcast-Adressen nutzen.

→ Lösung: `config.toml` wird strikt validiert (Datentypen, IP-Format, Portbereich). Beim Start meldet der Client Abweichungen in roter Schrift und bricht ab, solange das Problem nicht behoben wurde.

## 8. Plattform-übergreifende Dateipfade

→ Problem: Unterschiedliche Pfad-Separatoren (Windows/MacOS/Linux) verursachen Fehlzugriffe auf `imagepath`.

→ Lösung: `pathlib.Path` wird konsequent genutzt, sodass GUI und CLI alle Pfade im nativen Betriebssystemformat anzeigen können.

## 9. Verbindungsaufbau im Universitäts-WLAN

→ Problem: Das Hochschul-WLAN setzt Client-Isolation und Broadcast-Filter ein. UDP-Broadcasts auf `whoisport` erreichen daher keine Endgeräte, wodurch JOIN/WHO-Pakete verlorengehen und keine Peers gefunden werden.

→ Lösung: ein eigener Wi-Fi-Hotspot ohne Isolation.

## 10. Peer-Discovery nicht möglich im GitHub-Repository

→ Problem: Das Repository ließ sich zwar starten, jedoch fand der Client keine anderen Peers.

→ Lösung: endgültige Ausführung des Clients findet lokal statt und nicht über das GitHub-Repository.

## 11. GUI-Technologiewechsel: PyQt5 → Tkinter

→ Problem: Die GUI war anfangs in PyQt5 implementiert. Die Abhängigkeit erwies jedoch als schwergewichtig für ein lokales Studentenprojekt.

→ Lösung: Das Team portierte die Oberfläche deshalb auf Tkinter (`gui tk.py`). Tkinter ist in jeder Standard Python Distribution enthalten und benötigt keine externen Bibliotheken. Gleichzeitig ließ sich die Event-Verarbeitung einfach an die vorhandenen Queues anbinden. Die Umstellung reduzierte die Installationszeit und beseitigte die letzten Import-Fehler.



## 4 Modul- und Klassenübersicht

### 4.1 Client-Klasse

**Klasse:** client.py

**Verantwortung:**

Erzeugt und versendet SLCP-Nachrichten (JOIN, MSG, IMG).  
Pflegt den lokalen Peer-Cache, verwaltet den Abwesenheitsstatus  
und leitet eingehende Ereignisse an Server- oder GUI/CLI-Queues  
weiter.

**Wichtige Funktionen:**

- **Empfangen und Beantworten:** erkennt eingehende SLCP-Nachrichten und unterscheidet zwischen Text, Bildern und Steuerbefehlen ( `join`, `whois`, `iam`, `msg`, `img` ).
- **Abwesenheitsmodus:** kann bei Aktivierung eine automatische Nachricht zurücksenden.
- **Peer-Verwaltung:** speichert Informationen über bekannte Peers (z.B. IP, Port, Handle).
- **Bildverarbeitung:** verwaltet den Empfang und die Speicherung von Bildern im vorgegebenen Pfad.
- **SLCP-Kopplung:** interagiert mit dem `SLCP-Handler` zur Einhaltung des Protokolls.

## 4.2 Server-Klasse

**Klasse:** server.py

### Verantwortung:

Agiert als TCP-Listener für alle einkommenden Nutzdaten. Die Klasse öffnet genau einen Stream-Socket (`AF_INET / SOCK_STREAM`) auf dem in `config.toml` hinterlegten Port, akzeptiert Verbindungen anderer Peers und leitet deren SLCP-Nachrichten an die Präsentationsschicht weiter. UDP-Broadcasts gehören ausschließlich zum `DiscoveryService` und sind hier nicht enthalten.

### Wichtige Funktionen:

- **Fehlerbehandlung:** der Server reagiert robust auf ungültige Pakete oder unbekannte Absender.
- **Graceful-Shutdown:** bricht bei gesetztem `shutdown_event` sauber ab.
- **server\_loop():** wartet auf neue TCP-Verbindungen und gibt sie an den Handler weiter.
- **Msg-Bedingung:** Textnachrichten in `net_to_interface_queue` einreihen.
- **Img-Bedingung:** TCP-Stream empfangen, Prüfsumme (SHA-256) verifizieren, Datei ablegen.
- **Logging:** schreibt Fehlermeldungen und Verbindungsabbrüche ins Log.

## 4.3 Discovery-Service-Klasse

**Klasse:** discovery\_service.py

**Verantwortung:**

Findet Peers im LAN per WHOIS/IAM-Broadcast, hält eine aktuelle Peer-Liste und sendet Änderungen an Client & GUI.

**Wichtige Funktionen:**

- **Broadcast:** versendet regelmäßig **whois**-Anfragen im Netzwerk
- **Antwortverarbeitung:** erkennt **iam**-Antworten und meldet neue Peers an den Client.
- **Peer-Aktualisierung:** erkennt doppelte oder veränderte Peers anhand von IPs/Handles und aktualisiert sie ggf.
- **Fallback-Unicast:** kann eine direkte Peerverbindung anlegen, falls Broadcast geblockt wird.
- **Timeout-Handling:** entfernt inaktive Peers, bevor sie veraltet angezeigt werden.

## 4.4 SLCP-Handler-Klasse

**Klasse:** slcp\_handler.py

**Verantwortung:**

Baut und zerlegt Nachrichten nach Simple Local Chat Protocol (SLCP). Arbeitet stateless als reine Utility-Klasse.

**Wichtige Funktionen:**

- **Protokollkodierung:** erstellt SLCP-konforme Nachrichten inkl. Header, Typ, Sender und Inhalt.
- **Protokolldekodierung:** analysiert eingehende Nachrichten und extrahiert Meta-Informationen.
- **Fehlertoleranz:** behandelt ungültige oder fehlerhafte SLCP-Nachrichten robust.
- **Unterstützung von Anhängen:** kann mit Bilddaten umgehen und diese Textinhalte unterscheiden.
- **Version-Konstante:** ermöglicht spätere Protokollerweiterungen.

## 4.5 Kommandozeilen-Interface

**Klasse:** CLI.py

**Verantwortung:**

Ein einfaches Interface für die Kommandozeile, über das Nutzer den Client steuern können. Ideal für Debugging-Zwecke.

**Wichtige Funktionen:**

- **Textinterface:** Benutzer geben Befehle direkt ein.
- **Peer-Steuerung:** Join, Away/Back, Nachricht senden, Bilder übertragen über Konsolensteuerung. `do_join()`, `do_msg()`, `do_img()`, `do_msgall()` → Befehle direkt zum Client.
- **Auto-Away-Timer:** setzt Away-Flag nach definierter Inaktivität.
- **Hilfe-/Fehlermeldungen:** bietet Feedback bei falschen Eingaben oder Verbindungsproblemen.

## 4.6 Grafisches-Interface

**Klasse:** gui\_tk.py

### **Verantwortung:**

Stellt ein vollständiges Tkinter realisiertes Chat-Fenster bereit. Es visualisiert alle eingehenden Ereignisse (Text, Bilder, Peer-Status und leitet Benutzereingaben an den Client weiter. Die Klasse dient damit als einzige Schnittstelle zwischen Endnutzer und Kernlogik im GUI-Modus und übernimmt zusätzlich das periodische Polling der Netzwerk-Queues, ohne den TK-Hauptthread zu blockieren.

### **Wichtige Funktionen:**

- **Fenster- und Layout-Erstellung:** initialisiert Hauptfenster, Nachrichten Canvas, Peer Liste, Eingabezeile und Toolbar-Buttons
- **Nachrichtenanzeige:** rendert eingehende Text- und Bildevents aus `net_to_interface_queue` chronologisch im Chat-Canvas.
- **Text senden:** wertet Eingabe-Return oder „Senden“ Button aus und ruft `Client.client_send_msg()` auf.
- **Bildversand:** öffnet einen Datei-Dialog, lädt die gewählte Datei und übergibt sie an `Client.client_send_img()`.
- **Peer-Liste aktualisieren:** übernimmt IAM/WHO Antworten aus `disc_to_interface_queue`, zeigt Handles mit Onlinestatus an.
- **Periodisches Queue-Polling:** nutzt `after(250,...)`, um Netzwerk-Queues Thread sicher auszulesen, ohne dabei das GUI zu blockieren.

## 4.7 Hauptmodul

**Klasse:** main.py

**Verantwortung:**

Haupt-Einstiegspunkt des Programms, lädt `config.toml`, startet `Client`, `Server`, `DiscoveryService`. Verbindet alle Komponenten und entscheidet, ob GUI oder CLI verwendet wird.

**Wichtige Funktionen:**

- **Initialisierung:** liest Konfigurationswerte, erstellt Instanzen von `Client`, `Server`.
- **Thread-Management:** startet `Discovery-Service` und `Server` parallel.
- **Interface-Auswahl:** je nach Kontext wird CLI oder GUI geladen.
- **Fehlerbehandlung:** fängt Startfehler ab und gibt entsprechende Hinweise.
- **Konfig-Validierung:** prüft IP-Formate, Port-Bereiche, Broadcast-Subnetz
- **Port-Prüfung:** sucht nächstbesten Port, falls Standard belegt ist.
- **Signal-Handler:** beendet Unterthreads bei `KeyboardInterrupt`.

## 5 Konfiguration: config.toml

### Funktion:

Zentrale Konfigurationsdatei des P2P-Chatclients. Hier werden alle grundlegenden Laufzeit- und Netzwerkparameter abgelegt, sodass der Programmcode selbst nicht verändert werden muss, wenn sich Umgebungs- oder Benutzereinstellungen ändern. Die nachfolgende Auflistung erläutert die wichtigsten Schlüssel und ihre jeweiligen Aufgaben.

- **handle:** Vorgabewert für den eigenen Benutzernamen. Wird beim Programmstart eingelesen und kann später durch den join-Befehl überschrieben werden.
- **port:** UDP-Port, auf dem Server und Client dieses Rechners Nachrichten austauschen. Alle eingehenden Chatpakete werden hier empfangen.
- **whoisport:** Gemeinsamer Discovery-Port für das gesamte lokale Netzwerk. Alle Peers senden und empfangen hierüber whois-/iam-Broadcasts, um sich gegenseitig zu finden.
- **broadcast:** Broadcast-Adresse des Subnetzes. Darüber werden die Discovery-Nachrichten ausgestrahlt. Muss auf allen Peers identisch eingestellt sein.
- **imagepath:** Verzeichnis, in das empfangene Bilddateien automatisch gespeichert werden. Falls nicht vorhanden, legt der Client den Ordner beim Start an.
- **autoreply:** Text, den der Client im Abwesenheitsmodus (Befehl **away**) als automatische Antwort verschickt.



## 6 Bedienungsanleitung

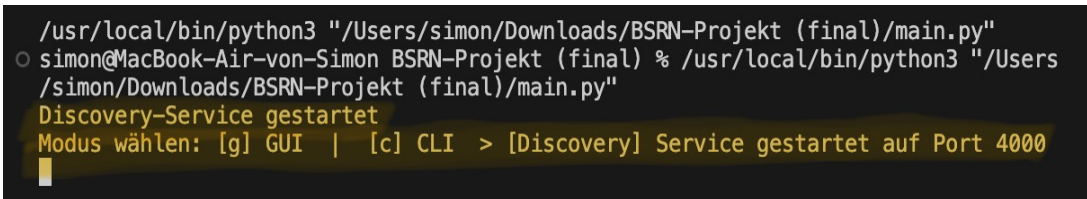
Im Folgenden wird beschrieben, wie das System gestartet und bedient wird und welche Klassen jeweils involviert sind. So erhalten Entwickler und User einen unmittelbaren Bezug zwischen praktischer Bedienung und Code-Architektur.

### 6.1 Vorbereitung

1. Installation von min. Python Version  $\geq 3.8$ , pip install von **Tkinter**, **toml** und **Pillow**.
2. Sicherstellen, dass Parameter wie **handle**, **port**, **whoisport**, **broadcast**, **impagepath** und **autoreply** in **config.toml** enthalten sind.
3. **Handle, Ports, Broadcast** müssen nicht festgelegt werden, wird vom Programm selbst festgelegt.
4. **Handle** wird beim Start des Programms vom Benutzer, selbst festgelegt und in der **config.toml** überschrieben.
5. Ordner aus **impagepath** anlegen
  - Sicherstellen, dass empfangene Bilder gespeichert werden können.

### 6.2 Benutzung mit GUI

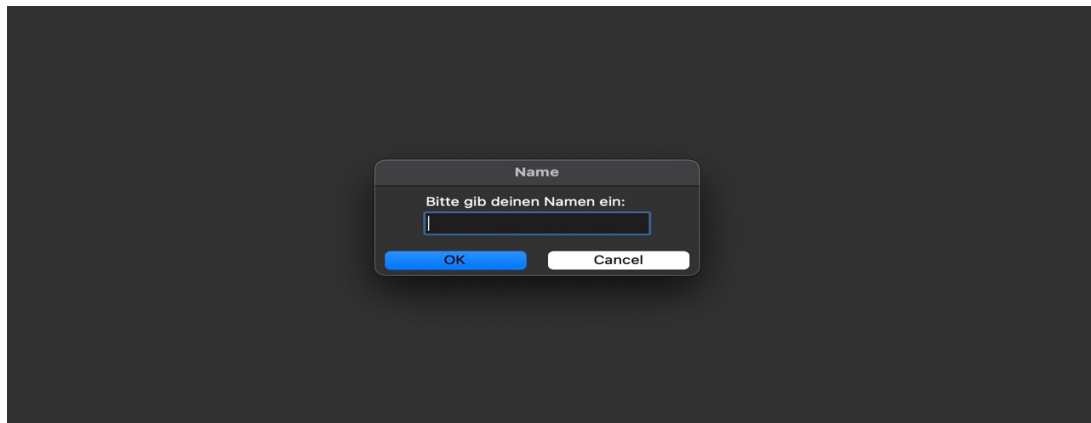
1.  → **main.py** mithilfe des Run-Button in der Entwicklungsumgebung starten.

2. 

```
/usr/local/bin/python3 "/Users/simon/Downloads/BSRN-Projekt (final)/main.py"  
○ simon@MacBook-Air-von-Simon BSRN-Projekt (final) % /usr/local/bin/python3 "/Users/simon/Downloads/BSRN-Projekt (final)/main.py"  
Discovery-Service gestartet  
Modus wählen: [g] GUI | [c] CLI > [Discovery] Service gestartet auf Port 4000
```

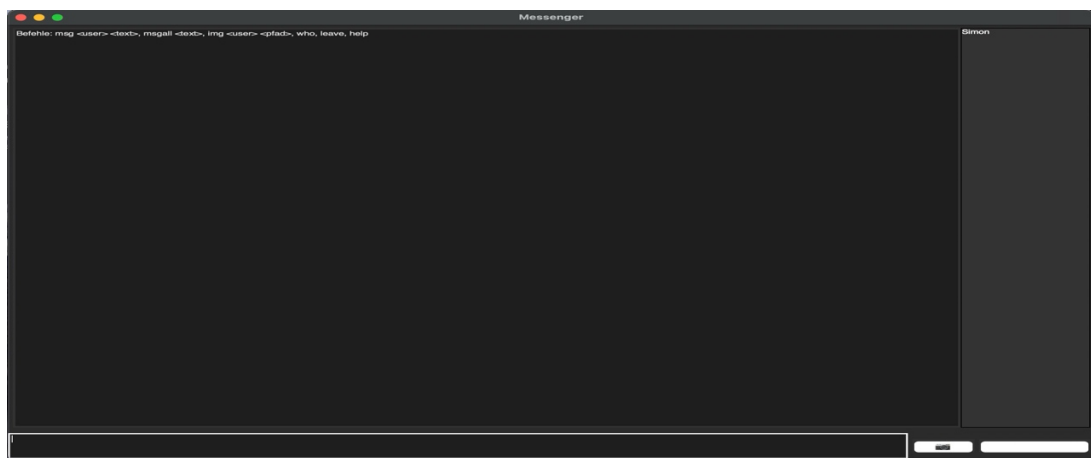
→ [g] für GUI auswählen.

3.



→ Anzeigenamen im Chat angeben.

4.



→ Chat öffnet sich, Befehle können eingegeben werden.

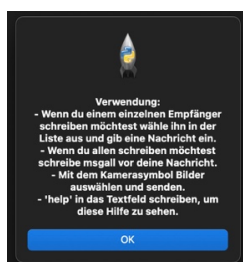
5. → Befehle, welche zur Verfügung stehen: **msgall <text>**, **img <user> <pfad>**, **who**, **help**, Textnachrichten an einen Teilnehmer werden durch Anklicken des Namens und Eingabe des gewünschten Textes realisiert.

6.



→ Eingabezeile für Befehle und eingebauten Foto-Button, worüber durch Anklicken der gewünschten Zielperson im Personenregister direkt Fotos an den Chat-Teilnehmer geschickt werden können.

7.

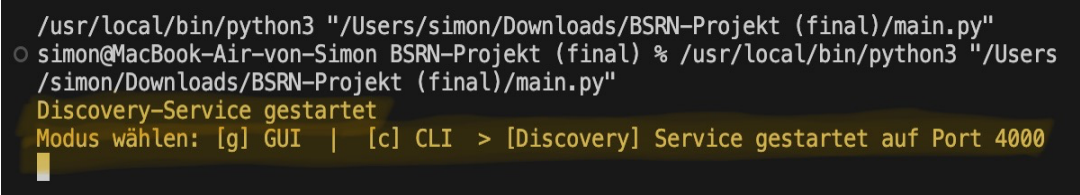


→ durch Eingabe des Befehls „**help**“ öffnet sich dieses Popup-Fenster.

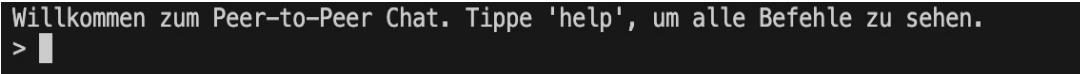
8. → Bei gewünschtem Verlassen des Chat-Rooms, muss der Befehl **leave** eingegeben werden und anschließend kann das Programm geschlossen werden.

## 6.3 Benutzung mit CLI

1.  → **main.py** mithilfe des Run-Buttons in der Entwicklungsumgebung starten.

2. 

→ [c] für CLI auswählen

3. 

→ **'help'** für Befehlsauswahl in das Terminal eingeben.

4. → Befehle, welche zur Verfügung stehen: **msg <user> <text>**, **msgall <text>**, **img <user> <pfad>**, **who**, **leave**, **help**, **exit**, **set\_config <parameter> <wert>**, **show\_config**
5. → Bei gewünschtem Verlassen des Chat-Rooms, muss der Befehl **leave** eingegeben werden.