DOKUMENTATION

Erzeugt von Doxygen 1.14.0

Kapitel 1

mainpage2

/**

1.1 Peer-to-Peer Chat – Dokumentation

1.1.1 Projektüberblick

Ein dezentraler Chat-Client im lokalen Netzwerk, der Text- und Bildnachrichten ohne zentralen Server ermöglicht. Hauptfunktionen: Peer Discovery, Text- und Bildübertragung, Kommandozeilen-Bedienung (CLI), automatische Nutzererkennung.

1.1.1.1 Gruppenmitglieder

- Aysha Aryubi
- Nhu Ngoc Le
- Artur Gubarkov

1.1.1.2 Abgabedatum / Semester / Kurs

- Abgabedatum: 22. Juni 2025
- Semester: SoSe 2025
- Kurs: Betriebssysteme und Rechnernetze
- Dozent: Prof. Dr.-Ing. Markus Miettinen

1.1.2 Architektur

Das System ist modular aufgebaut und besteht aus folgenden Hauptkomponenten:

- **Interface (CLI):** Stellt die Kommandozeilen-Bedienung bereit. Nutzer gibt hier Befehle wie `/msg`, `/img`, `/who` usw. ein. Das Interface leitet die Befehle an den Messenger weiter und zeigt empfangene Nachrichten und Bilder an.
- **Messenger:** Übernimmt die Netzwerkkommunikation. Sendet und empfängt Nachrichten sowie Bilder über UDP/TCP. Kommuniziert direkt mit Interface, DiscoveryService und nutzt das Protocol-Modul zum Kodieren/Dekodieren der Nachrichten.
- **Protocol:** Stellt Funktionen zum Kodieren und Parsen des SLCP-Protokolls bereit (JOIN, MSG, IMG etc.).

- **DiscoveryService:** Verantwortlich für Peer-Discovery im lokalen Netzwerk mittels UDP-Broadcasts (JOIN, WHO, LEAVE, KNOWNUSERS). Pflegt eine aktuelle Liste bekannter Teilnehmer.
- **Config:** Verwaltet alle Einstellungen (wie Benutzername, Port, Bildverzeichnis) über eine TOML-Datei. Wird von Messenger und DiscoveryService geladen.
- **main.py:** Startet und verbindet alle Komponenten.

Das Zusammenspiel der Komponenten ist im folgenden Architekturdiagramm dargestellt:

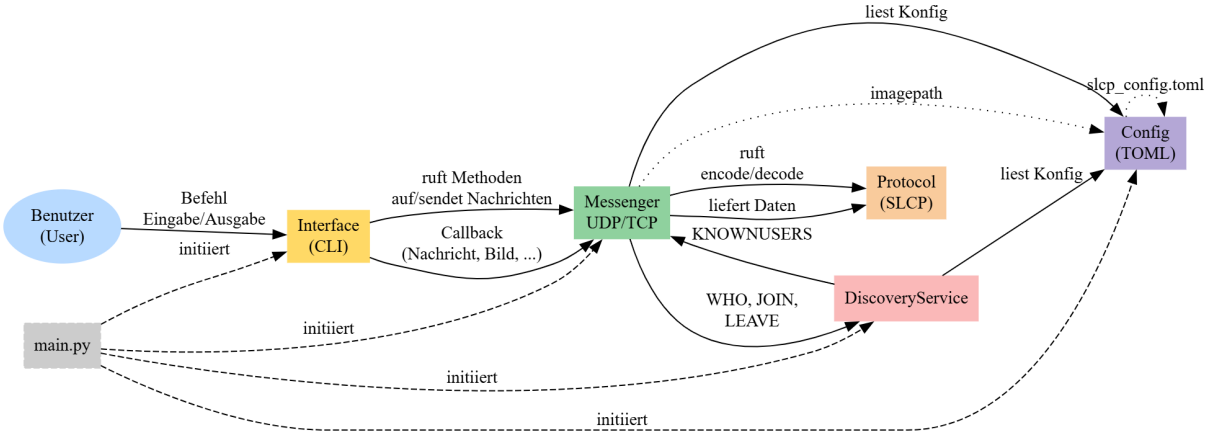


Abbildung 1.1 Architekturdiagramm

1.1.3 Verwendete Technologien & Tools

- **Programmiersprache:** Python 3.10+
- **Libraries:** socket, threading, asyncio, toml, colorama
- **Doxygen:** für technische Dokumentation (HTML, PDF)
- **Git:** für Versionierung (GitHub)
- **Konfigurationsdatei:** TOML-Format (slcp_config.toml)
- **Entwicklungsumgebung:** PyCharm

1.1.4 Protokollübersicht (SLCP)

Textbasierte Kommandos im Simple Local Chat Protocol:

Befehl	Parameter	Beschreibung
JOIN	<Handle> <Port>	Tritt dem Chat bei
LEAVE	<Handle>	Verlässt den Chat
MSG	<Handle> <Text>	Sendet Nachricht an Nutzer
IMG	<Handle> <Size>	Bildübertragung starten
WHO	-	Fragt bekannte Nutzer ab
KNOWNUSERS	<Liste>	Antwort mit bekannten Nutzern

1.1.4.1 Ablauf beim Anmelden im Chat

Im Folgenden wird gezeigt, wie ein Nutzer dem Chat beitritt:

1. **Alice** gibt im Interface den Befehl /join ein.
2. **Das Interface** ruft im Messenger die Funktion zum Beitreten auf.

3. **Der Messenger** veranlasst den Discovery-Service, eine JOIN Alice <Port>-Nachricht per UDP-Broadcast im lokalen Netzwerk zu senden.
4. **Alle Discovery-Services** der anderen Peers empfangen die JOIN-Nachricht und nehmen Alice in ihre Peer-Liste auf.
5. **Alice** ist jetzt als aktiver Teilnehmer im Chat-Netzwerk bekannt.

Der Ablauf ist im folgenden Diagramm dargestellt:

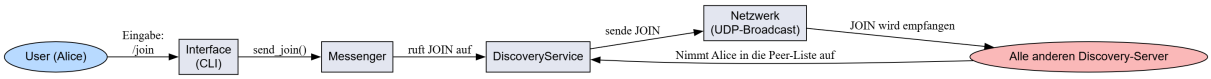


Abbildung 1.2 Ablauf JOIN

1.1.4.2 Ablauf beim Abmelden aus dem Chat

Im Folgenden wird gezeigt, wie ein Nutzer den Chat verlässt:

1. **Alice** gibt im Interface den Befehl /leave ein.
2. **Das Interface** ruft im Messenger die Funktion zum Verlassen des Chats auf.
3. **Der Messenger** veranlasst den Discovery-Service, eine LEAVE Alice-Nachricht per UDP-Broadcast im lokalen Netzwerk zu senden.
4. **Alle Discovery-Services** der anderen Peers empfangen die LEAVE-Nachricht und entfernen Alice aus ihrer Peer-Liste.
5. **Alice** wird aus dem Netzwerk entfernt und erscheint nicht mehr als aktiver Teilnehmer.

Der Ablauf ist im folgenden Diagramm dargestellt:

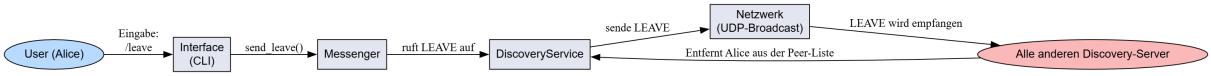


Abbildung 1.3 Ablauf LEAVE

1.1.4.3 Ablauf beim Abfragen und Empfangen der Nutzerliste

Im Folgenden wird gezeigt, wie die Liste aller aktiven Nutzer im Chat abgefragt wird:

1. **Bob** gibt im Interface den Befehl /who ein.
2. **Das Interface** ruft im Messenger die Funktion zum Senden einer WHO-Anfrage auf.
3. **Der Messenger** veranlasst den Discovery-Service, eine WHO-Nachricht per UDP-Broadcast an alle Discovery-Services im lokalen Netzwerk zu senden.
4. **Jeder Discovery-Service** (bei allen aktiven Peers) empfängt die WHO-Anfrage und antwortet per Unicast mit einer eigenen KNOWNUSERS-Nachricht, die alle aktuell bekannten Nutzer (Handle, IP, Port) enthält.
5. **Der Messenger von Bob** empfängt eine oder mehrere KNOWNUSERS-Antworten, konsolidiert die Einträge (entfernt ggf. Duplikate) und aktualisiert seine interne Peer-Liste.
6. **Das Interface bei Bob** zeigt daraufhin die vollständige Liste aller aktuell erreichbaren Nutzer im Terminal an.

Der Ablauf ist im folgenden Diagramm dargestellt:

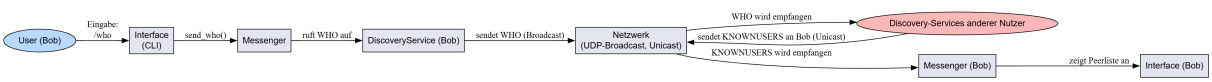


Abbildung 1.4 Ablauf WHO/KNOWNUSERS

1.1.4.4 Ablauf beim Senden einer Textnachricht

Im Folgenden wird gezeigt, wie ein Bild vom Nutzer „Alice“ an „Bob“ gesendet wird:

1. **Alice** gibt im Interface den Befehl /msg Bob Hallo Bob! ein.
2. **Das Interface** ruft im Messenger die Funktion zum Senden einer Nachricht auf.
3. **Der Messenger** ermittelt die Netzwerkadresse von Bob und sendet den SLCP-Befehl MSG Bob "Hallo Bob!" per UDP direkt an Bobs Peer-Adresse.
4. **Bob's Messenger** empfängt die Nachricht, prüft den Befehl und übergibt den Text an das lokale Interface.
5. **Das Interface bei Bob** zeigt die empfangene Nachricht sofort im Terminal an.

Der Ablauf ist im folgenden Diagramm dargestellt:

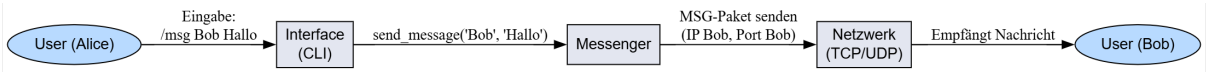


Abbildung 1.5 Ablauf MSG

1.1.4.5 Ablauf beim Senden eines Bildes

Im Folgenden wird gezeigt, wie ein Bild vom Nutzer „Alice“ an „Bob“ gesendet wird:

1. **Alice** gibt im Interface den Befehl `/img Bob pfad/zum/bild.jpg` ein.
2. **Das Interface** ruft im Messenger die Funktion zum Senden eines Bildes auf.
3. **Der Messenger** öffnet eine TCP-Verbindung zum Peer (Bob), sendet den SLCP-Befehl `IMG Bob <Größe>`, danach die Bilddaten als Bytestream.
4. **Bob's Messenger** nimmt die Verbindung entgegen, liest Befehl und Daten und speichert das Bild im vorgegebenen Ordner.
5. **Das Interface bei Bob** zeigt eine Benachrichtigung über das empfangene Bild an.

Der Ablauf ist im folgenden Diagramm dargestellt:

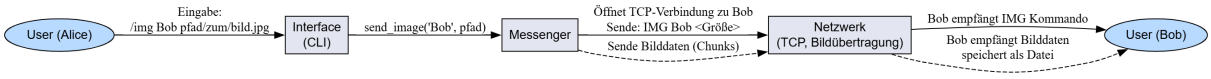


Abbildung 1.6 Ablauf IMG

1.1.4.6 Ausgabe-Beispiel

Im Folgenden einige Screenshots der CLI-Anwendung:

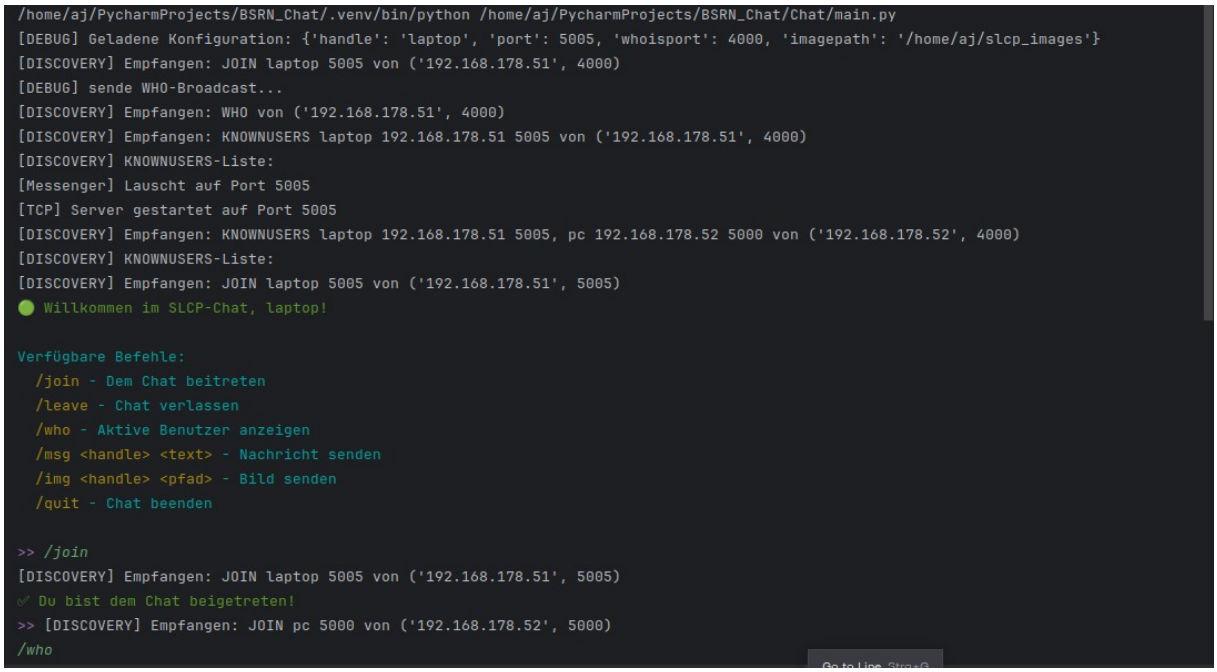


Abbildung 1.7 Ausgabe 1

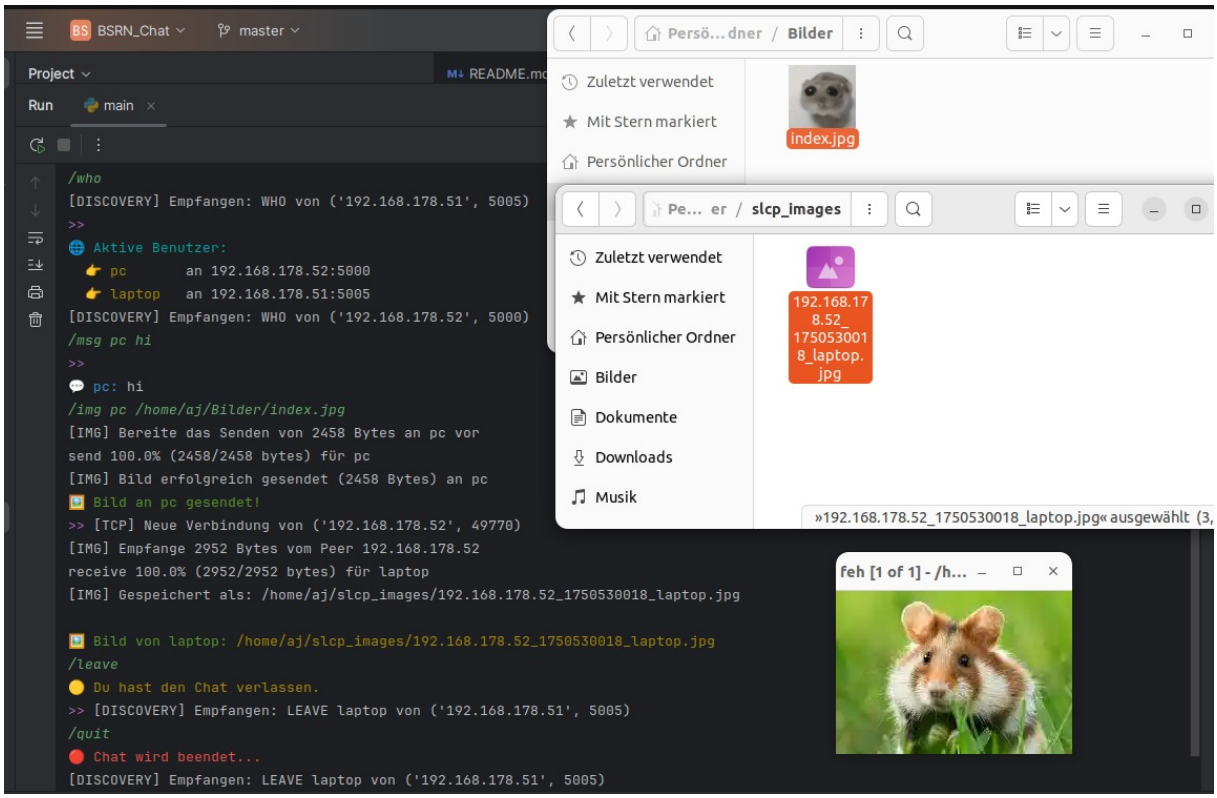


Abbildung 1.8 Ausgabe 2

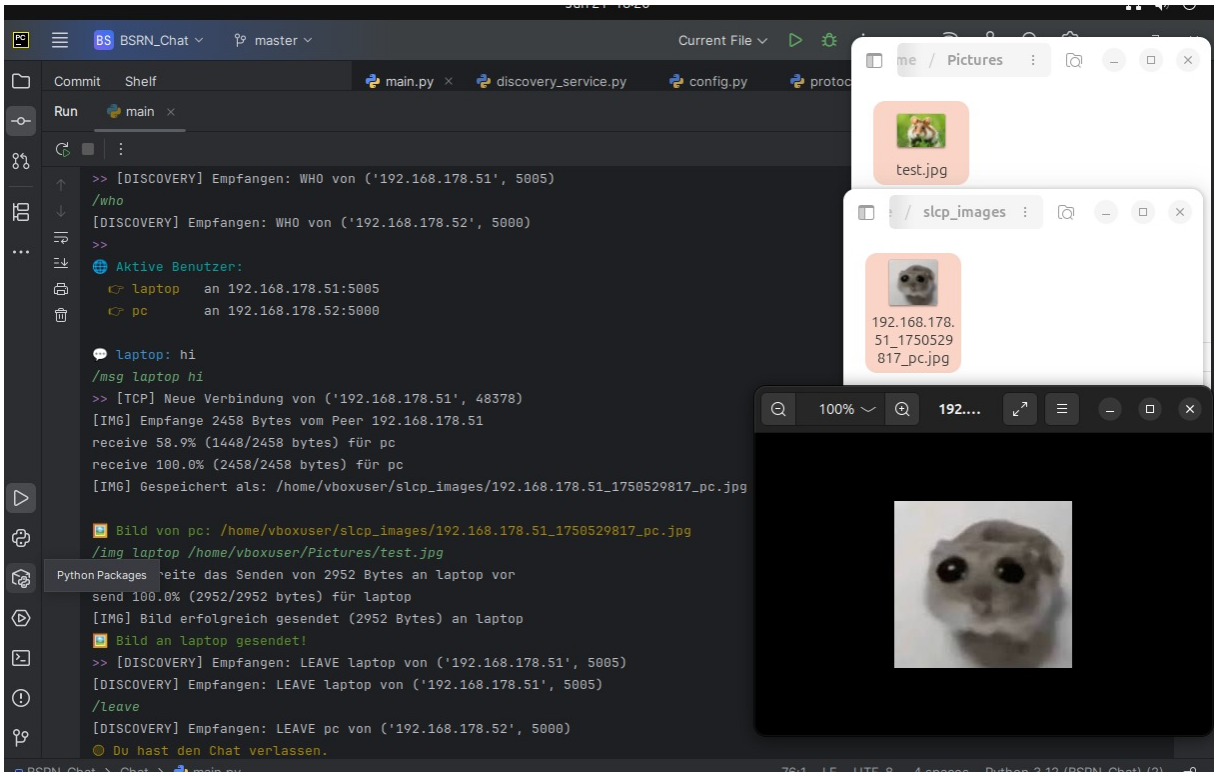


Abbildung 1.9 Ausgabe 3

1.1.5 Teilprobleme und Lösungsansätze (Fehleranalyse)

Im Folgenden werden typische Fehler und ihre Lösungen im Projekt erläutert.
Jedes Beispiel enthält einen Screenshot sowie eine kurze Analyse und die jeweilige Korrektur.

1.1.5.1 IndexError beim Parsen von Benutzerdaten

Beim Parsen der „KNOWNUSERS“-Nachricht wurde die empfangene Zeichenkette mit `info.split()` zerlegt, ohne zu prüfen, ob wirklich drei Felder (`Handle`, `IP`, `Port`) vorliegen. Wenn ein Netzwerkpaket fehlformatiert war (z.B. zu wenige Einträge enthielt), wurde auf ein nicht vorhandenes Element zugegriffen – das führte zu einem IndexError. Dieser Fehler trat insbesondere bei Netzwerkproblemen oder inkompatiblen Peers auf und konnte das ganze Programm zum Absturz bringen.

Chat\discovery\discovery_service.py

```
@@ -106,12 +106,18 @@ class DiscoveryService:
106 106                 user_list = [u.strip() for u in user_str.split(",") if u.strip()]
107 107                 print("[DISCOVERY] KNOWNUSERS-Liste:")
108 108                 with self.peers_lock:
    109 +                   seen = set()
109 110                     for entry in user_list:
110 111                         infos = entry.strip().split()
111 112                         if len(infos) == 3:
112     -                           handle, ip, port = infos
113     -                           self.peers[handle] = (ip, int(port))
114     -                           print(f" - {handle} @ {ip}:{port}")
    113 +                           handle = infos[0].strip()
    114 +                           ip = infos[1].strip()
    115 +                           port = int(infos[2].strip())
    116 +                           key = f"{handle}@{ip}:{port}"
    117 +                           if key not in seen:
    118 +                               self.peers[handle] = (ip, port)
    119 +                               seen.add(key)
    120 +                               print(f" - {handle} @ {ip}:{port}")
115 121
116 122         def send_who(self):
117 123             msg = "WHO\n"
```

Abbildung 1.10 IndexError Fix

Lösung:
Jetzt wird vor dem Zugriff geprüft, ob `len(infos) == 3` gilt. So werden nur gültige Peer-Informationen akzeptiert und verarbeitet, wodurch Abstürze durch fehlerhafte Netzwerkdaten zuverlässig verhindert werden.

1.1.5.2 Relativer Bildpfad und Ordnerprüfung

Ursprünglich wurde als Speicherort für empfangene Bilder ein relativer Pfad (`"./images"`) verwendet. Das war problematisch, da das Verzeichnis je nach aktuellem Arbeitsverzeichnis unterschiedlich interpretiert werden kann und bei fehlender Erstellung Dateioperationen fehlschlugen. Das führte vor allem beim ersten Programmstart zu unerwarteten Fehlern beim Speichern von Bildern.

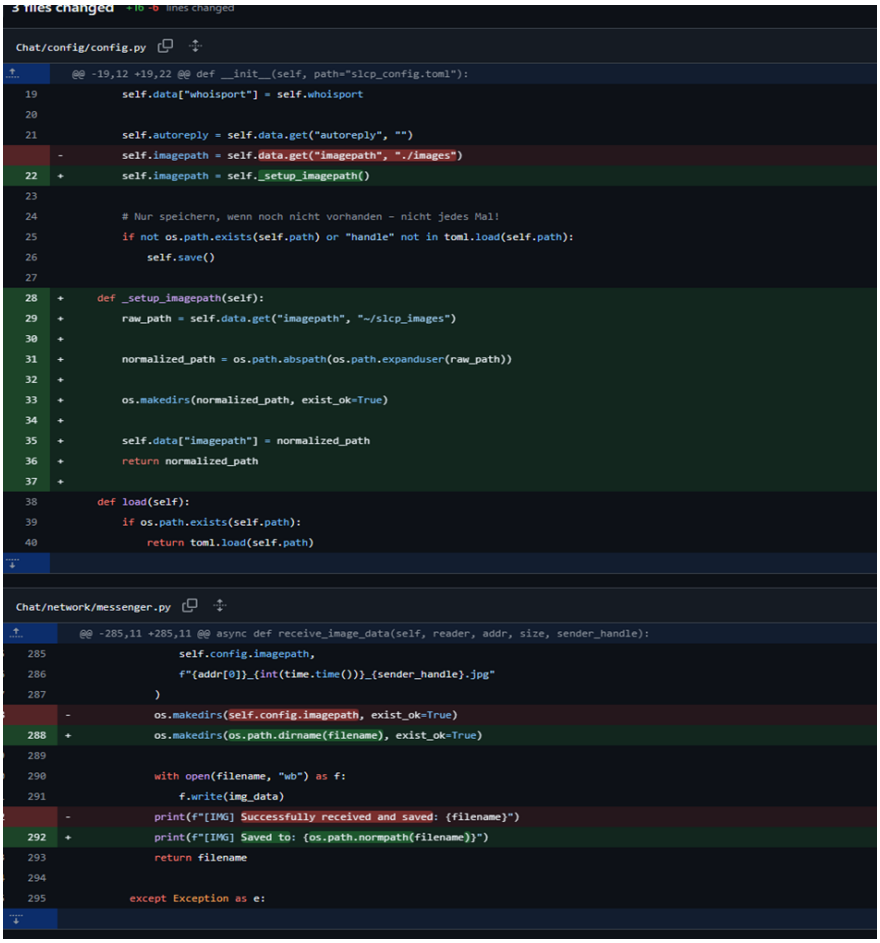


Abbildung 1.11 Bildpfad Fix

Lösung:
Die Methode `_setup_imagepath()` wandelt den Bildpfad jetzt in einen absoluten, benutzerspezifischen Pfad (z.B. `~/slcp_images`) um und legt das Zielverzeichnis automatisch an. Dadurch werden Bilder plattformunabhängig und zuverlässig gespeichert.

1.1.5.3 Konsistente CLI-Befehle und Hilfeanzeige

Die Kommandozeilenschnittstelle (CLI) war zunächst inkonsistent gestaltet. Es gab sowohl den Befehl `/send` als auch `/msg` und die Argumentbeschreibungen waren unklar (z.B. `<message>` statt `<text>`). Das führte dazu, dass neue Benutzer die richtigen Befehle nicht immer sofort fanden und es zu Bedienungsfehlern kam.

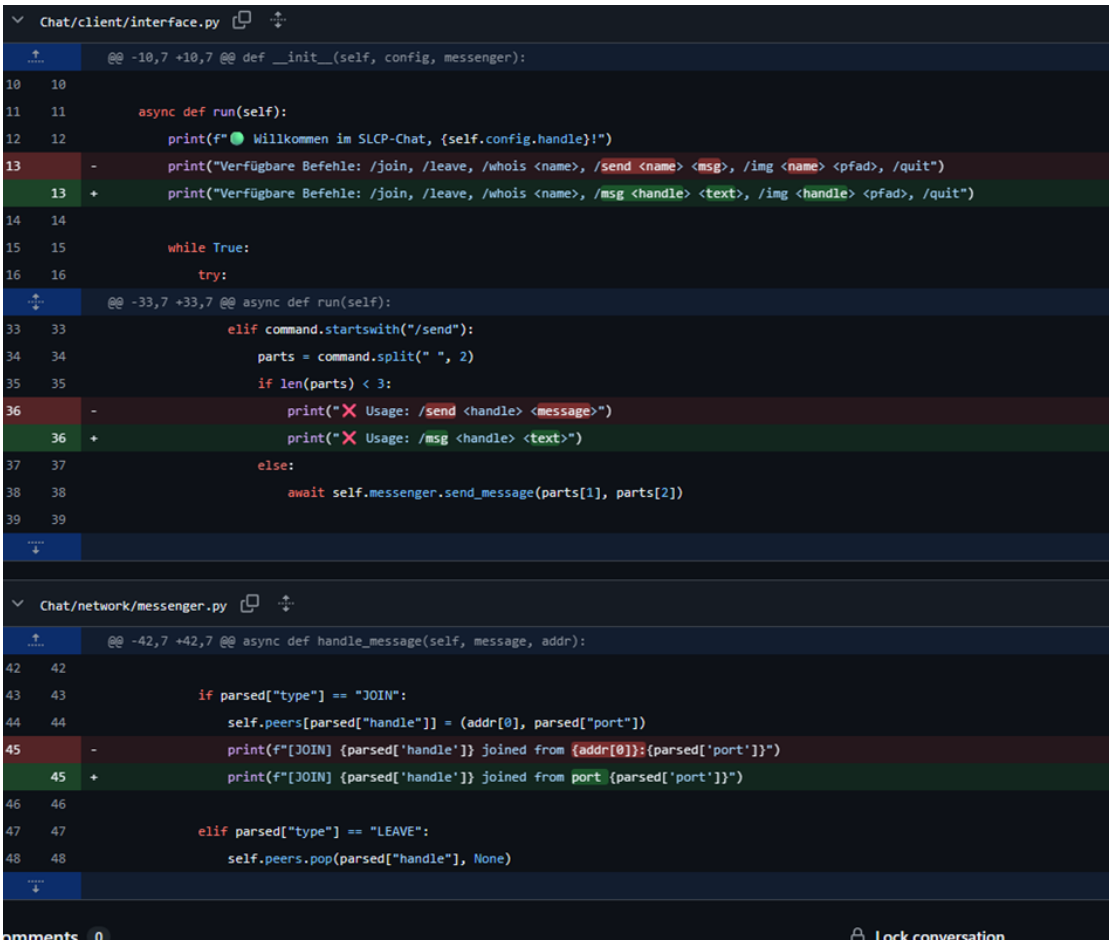
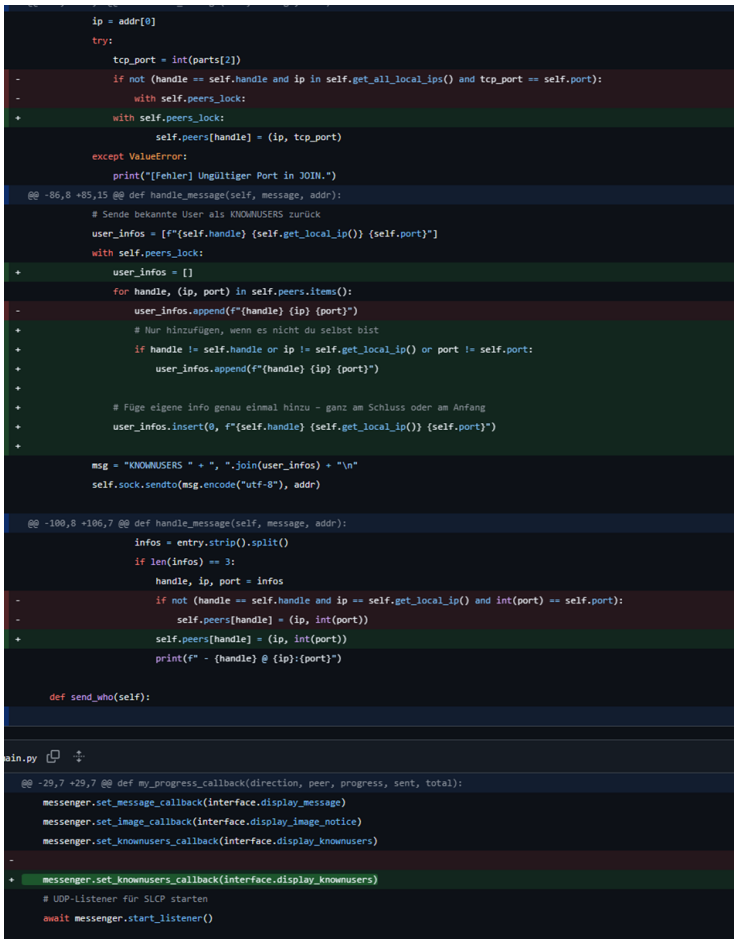


Abbildung 1.12 CLI-Hilfe

Lösung:
Alle Befehle und Hilfetexte wurden vereinheitlicht. Die CLI verwendet jetzt konsistente Benennungen wie `/msg` und eindeutige Argumente, was die Benutzung erleichtert und Missverständnisse minimiert.

1.1.5.4 Flexibler Umgang mit Peer-Daten (JOIN/KNOWNUSERS)

Bei der Verarbeitung von Peer-Informationen (insbesondere bei JOIN/KNOWNUSERS) wurden Einträge starr nach IP und Handle verglichen. Dadurch kam es gelegentlich zu unvollständigen oder doppelten Listen – etwa, wenn ein Nutzer mehrfach im Netzwerk auftauchte oder sich seine IP-Adresse änderte.



```
            ip = addr[0]
            try:
                tcp_port = int(parts[2])
-               if not (handle == self.handle and ip in self.get_all_local_ips() and tcp_port == self.port):
-                   with self.peers_lock:
+               with self.peers_lock:
                        self.peers[handle] = (ip, tcp_port)
            except ValueError:
                print("[Fehler] Ungültiger Port in JOIN.")
@@ -86,8 +85,15 @@ def handle_message(self, message, addr):
            # Sende bekannte User als KNOWNUSERS zurück
            user_infos = [f"{self.handle} {self.get_local_ip()} {self.port}"]
            with self.peers_lock:
+               user_infos = []
                for handle, (ip, port) in self.peers.items():
-                   user_infos.append(f"{handle} {ip} {port}")
+                   # Nur hinzufügen, wenn es nicht du selbst bist
+                   if handle != self.handle or ip != self.get_local_ip() or port != self.port:
+                       user_infos.append(f"{handle} {ip} {port}")
+
+               # Füge eigene info genau einmal hinzu - ganz am Schluss oder am Anfang
+               user_infos.insert(0, f"{self.handle} {self.get_local_ip()} {self.port}")
+
            msg = "KNOWNUSERS " + ", ".join(user_infos) + "\n"
            self.sock.sendto(msg.encode("utf-8"), addr)

@@ -100,8 +106,7 @@ def handle_message(self, message, addr):
                    infos = entry.strip().split()
                    if len(infos) == 3:
                        handle, ip, port = infos
-                       if not (handle == self.handle and ip == self.get_local_ip() and int(port) == self.port):
-                           self.peers[handle] = (ip, int(port))
+                       self.peers[handle] = (ip, int(port))
                        print(f" - {handle} @ {ip}:{port}")

    def send_who(self):

ain.py
@@ -29,7 +29,7 @@ def my_progress_callback(direction, peer, progress, sent, total):
    messenger.set_message_callback(interface.display_message)
    messenger.set_image_callback(interface.display_image_notice)
    messenger.set_knownusers_callback(interface.display_knownusers)
-
+   messenger.set_knownusers_callback(interface.display_knownusers)
    # UDP-Listener für SLCP starten
    await messenger.start_listener()
```

Abbildung 1.13 Peer-Daten Fix

Lösung:
Die neue Logik sammelt alle Peer-Daten und prüft erst nach dem Sammeln, ob Einträge tatsächlich zur eigenen Instanz gehören. Die eigene Information wird gezielt am Ende der Liste eingefügt, doppelte Einträge werden zuverlässig vermieden.

1.1.5.5 Konsolidierung mehrerer KNOWNUSERS-Antworten

Wenn eine WHO-Anfrage gestellt wurde, konnten mehrere „KNOWNUSERS“-Antworten fast gleichzeitig eintreffen. Der ursprüngliche Code verarbeitete jede Antwort einzeln, wodurch Informationen verloren gehen konnten, wenn Nachrichten schnell hintereinander empfangen wurden.

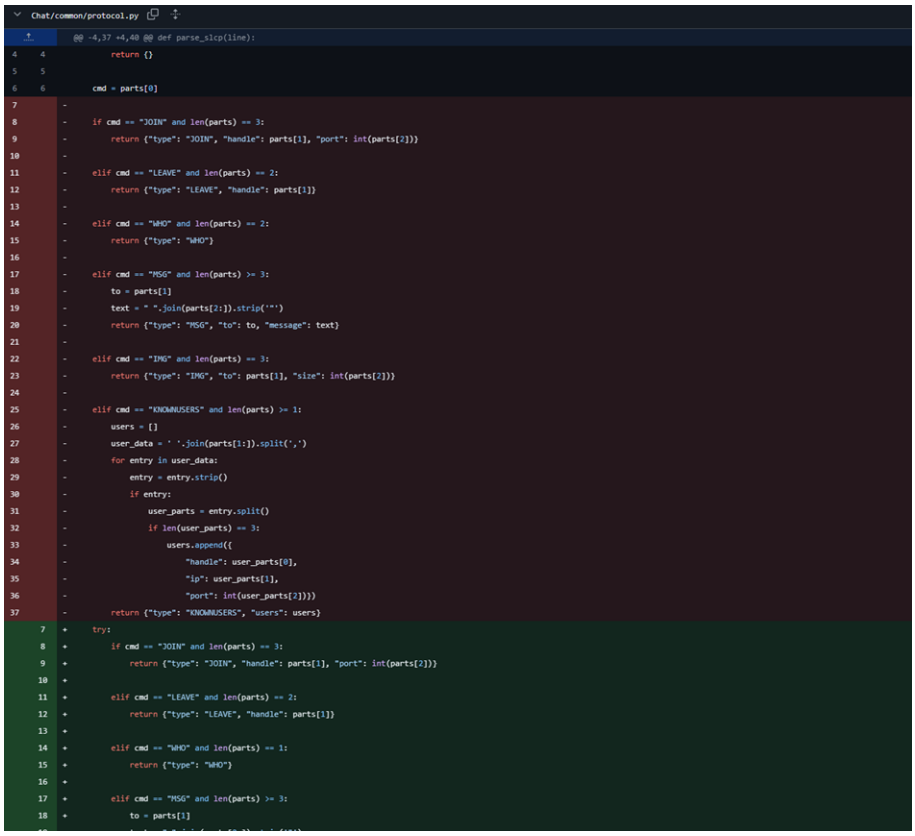
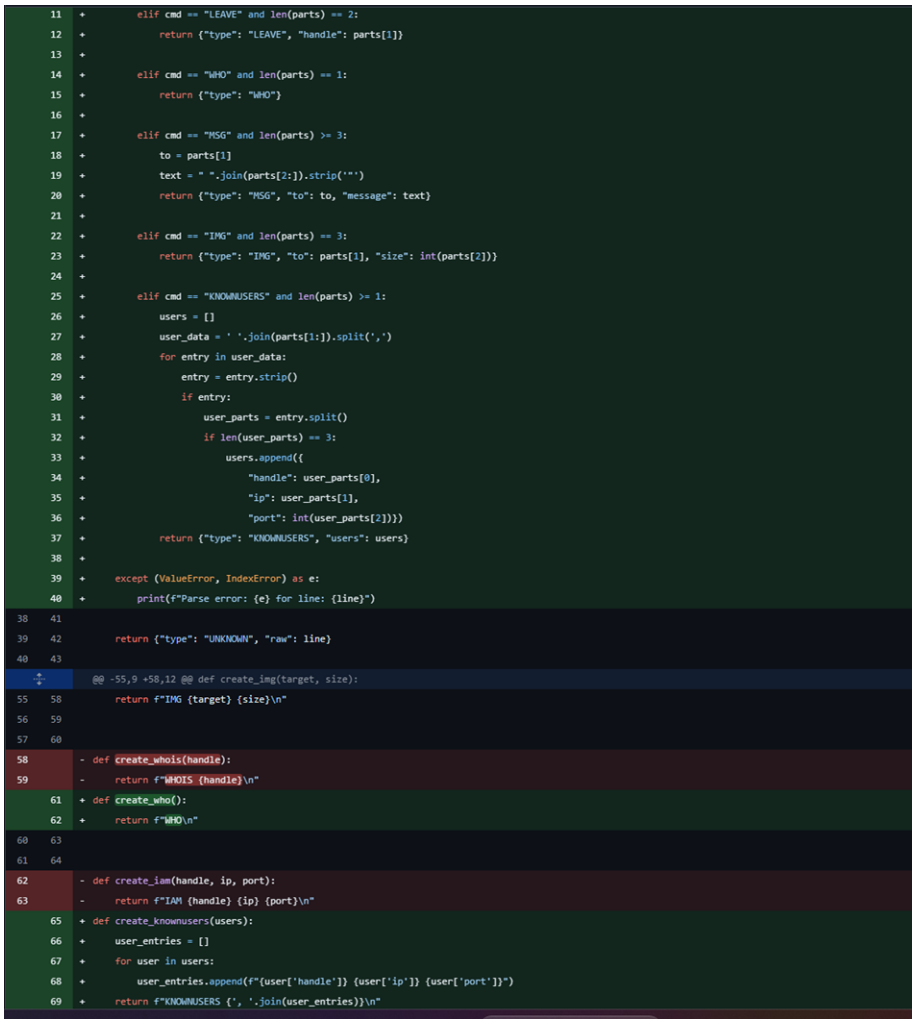


Abbildung 1.14 Knownusers Konsolidierung – vorher



```
+        elif cmd == "LEAVE" and len(parts) == 2:
+            return {"type": "LEAVE", "handle": parts[1]}
+
+        elif cmd == "WHO" and len(parts) == 1:
+            return {"type": "WHO"}
+
+        elif cmd == "MSG" and len(parts) >= 3:
+            to = parts[1]
+            text = " ".join(parts[2:]).strip('"')
+            return {"type": "MSG", "to": to, "message": text}
+
+        elif cmd == "IMG" and len(parts) == 3:
+            return {"type": "IMG", "to": parts[1], "size": int(parts[2])}
+
+        elif cmd == "KNOWNUSERS" and len(parts) >= 1:
+            users = []
+            user_data = ' '.join(parts[1:]).split(',')
+            for entry in user_data:
+                entry = entry.strip()
+                if entry:
+                    user_parts = entry.split()
+                    if len(user_parts) == 3:
+                        users.append({
+                            "handle": user_parts[0],
+                            "ip": user_parts[1],
+                            "port": int(user_parts[2])})
+            return {"type": "KNOWNUSERS", "users": users}
+
+    except (ValueError, IndexError) as e:
+        print(f"Parse error: {e} for line: {line}")
38 41
39 42       return {"type": "UNKNOWN", "raw": line}
40 43
    @@ -55,9 +58,12 @@ def create_img(target, size):
55 58       return f"IMG {target} {size}\n"
56 59
57 60
58      - def create_whois(handle):
59      -     return f"WHOIS {handle}\n"
   61   + def create_who():
   62   +     return f"WHO\n"
60 63
61 64
62      - def create_iam(handle, ip, port):
63      -     return f"IAM {handle} {ip} {port}\n"
   65   + def create_knownusers(users):
   66   +     user_entries = []
   67   +     for user in users:
   68   +         user_entries.append(f"{user['handle']} {user['ip']} {user['port']}")
   69   +     return f"KNOWNUSERS {', '.join(user_entries)}\n"
```

Abbildung 1.15 Knownusers Konsolidierung – nachher

Lösung:
Die neue Funktion `handle_knownusers_response` sammelt alle Antworten für eine kurze Zeit, konsolidiert die erhaltenen Daten und übergibt die vollständige Liste dann gesammelt an das Interface.

1.1.5.6 Verbesserte Anzeige des Absenders

Beim Anzeigen eingehender Nachrichten wurde bisher oft nur die rohe IP-Adresse des Absenders angezeigt, selbst wenn dessen Handle bekannt war. Das erschwerte die Zuordnung und war wenig benutzerfreundlich.

Chat/network/messenger.py

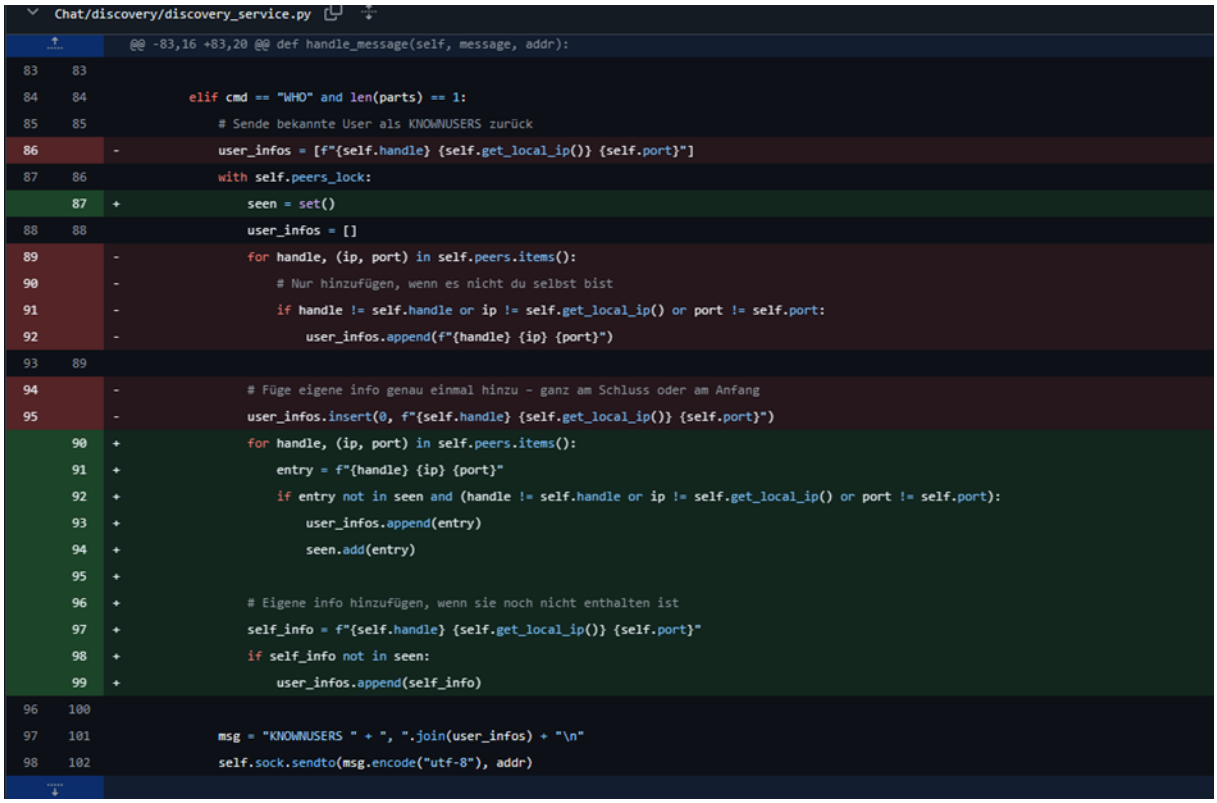
```
@@ -82,12 +82,18 @@ async def handle_message(self, message, addr):
                if parsed["to"] == self.config.handle:
                    msg = parsed["message"]
                    sender = addr[0]
+                   sender_handle = None
+                   for handle, (ip, _) in self.peers.items():
+                       if ip == sender:
+                           sender_handle = handle
+                           break
+                   sender_display = sender_handle if sender_handle else f"Unbekannt ({sender})"
                    if self.message_callback:
-                       await self.message_callback(sender, msg)
+                       await self.message_callback(sender_display, msg)
                    else:
-                       print(f"[{sender}] {msg}")
+                       print(f"\n💬 Nachricht von {sender_display}: {msg}")
                    if self.config.autoreply:
-                       await self.send_message(sender, self.config.autoreply)
+                       await self.send_message(sender_display, self.config.autoreply)

            elif parsed["type"] == "IMG":
                if parsed["to"] == self.config.handle:
@@ -127,9 +133,12 @@ async def send_message(self, handle, message):
        if handle not in self.peers:
            print(f"[Error] No known peer with handle '{handle}'")
            return
-       msg = protocol.create_msg(handle, message)
-       ip, port = self.peers[handle]
-       await self.send_slcp(msg, ip, port)
+       if handle in self.peers:
+           ip, port = self.peers[handle]
+           msg = protocol.create_msg(handle, message)
+           await self.send_slcp(msg, ip, port)
+       else:
+           print(f"[Error] Handle '{handle}' not connected")

    async def send_image(self, handle, filepath):
        if handle not in self.peers:
```

Abbildung 1.16 Absenderanzeige

Lösung:
Jetzt wird – sofern möglich – immer der Benutzername (Handle) angezeigt. Ist dieser nicht bekannt, erscheint stattdessen „Unbekannt“.

1.1.5.7 Doppelte Einträge in WHO/KNOWNUSERS-Liste

Die Peer-Liste enthielt manchmal doppelte Einträge, da bei jedem Empfang der eigenen Information diese erneut angehängt wurde. Das sorgte für Unübersichtlichkeit in der Nutzeranzeige.



Chat/discovery/discovery_service.py

```
@@ -83,16 +83,20 @@ def handle_message(self, message, addr):
83  83
84  84          elif cmd == "WHO" and len(parts) == 1:
85  85              # Sende bekannte User als KNOWNUSERS zurück
86      -           user_infos = [f"{self.handle} {self.get_local_ip()} {self.port}"]
87  86              with self.peers_lock:
    87  +               seen = set()
88  88                  user_infos = []
89      -               for handle, (ip, port) in self.peers.items():
90      -                   # Nur hinzufügen, wenn es nicht du selbst bist
91      -                   if handle != self.handle or ip != self.get_local_ip() or port != self.port:
92      -                       user_infos.append(f"{handle} {ip} {port}")
93  89
94      -               # Füge eigene info genau einmal hinzu – ganz am Schluss oder am Anfang
95      -               user_infos.insert(0, f"{self.handle} {self.get_local_ip()} {self.port}")
    90  +               for handle, (ip, port) in self.peers.items():
    91  +                   entry = f"{handle} {ip} {port}"
    92  +                   if entry not in seen and (handle != self.handle or ip != self.get_local_ip() or port != self.port):
    93  +                       user_infos.append(entry)
    94  +                       seen.add(entry)
    95  +
    96  +               # Eigene info hinzufügen, wenn sie noch nicht enthalten ist
    97  +               self_info = f"{self.handle} {self.get_local_ip()} {self.port}"
    98  +               if self_info not in seen:
    99  +                   user_infos.append(self_info)
96  100
97  101             msg = "KNOWNUSERS " + ", ".join(user_infos) + "\n"
98  102             self.sock.sendto(msg.encode("utf-8"), addr)
```

Abbildung 1.17 Peerlist Duplikate

Lösung:
Mithilfe einer sogenannten „seen“-Menge werden doppelte Einträge beim Sammeln gezielt vermieden.

Weitere Details und vollständige Screenshots siehe:
Screenshots mit Fehleranalyse

1.1.6 Bedienung

- **Start:**
  ```bash python main.py
  ```

Kapitel 2

Verzeichnis der Namensbereiche

2.1 Liste aller Namensbereiche

Liste aller dokumentierten Namensbereiche mit Kurzbeschreibung:

Kapitel 3

Hierarchie-Verzeichnis

3.1 Klassenhierarchie

Die Liste der Ableitungen ist, mit Einschränkungen, alphabetisch sortiert:

Kapitel 4

Klassen-Verzeichnis

4.1 Auflistung der Klassen

Hier folgt die Aufzählung aller Klassen, Strukturen, Varianten und Schnittstellen mit einer Kurzbeschreibung:

Kapitel 5

Datei-Verzeichnis

5.1 Auflistung der Dateien

Hier folgt die Aufzählung aller dokumentierten Dateien mit einer Kurzbeschreibung:

Kapitel 6

Dokumentation der Namensbereiche

6.1 config.config-Namensbereichsreferenz

Klassen

- class Config

6.1.1 Ausführliche Beschreibung

```
@file config.py
@brief Klasse zur Verwaltung und Speicherung der Konfiguration des Chat-Clients (Laden/Speichern von slcp_conf
```

6.2 discovery.discovery_service-Namensbereichsreferenz

Klassen

- class DiscoveryService

Funktionen

- bool is_port_in_use (int port)

Variablen

- int **BROADCAST_PORT** = 4000
- int **BUFFER_SIZE** = 1024
- **service** = DiscoveryService("slcp_config.toml")
- **peers** = service.get_peers()

6.2.1 Ausführliche Beschreibung

```
@file discovery_service.py
@brief Discovery-Service für den P2P-Chat. Verwaltet Peer-Discovery, JOIN/LEAVE/WHO Broadcast, etc.
```

6.2.2 Dokumentation der Funktionen

6.2.2.1 is_port_in_use()

```
bool discovery.discovery_service.is_port_in_use (
            int port)

@brief Prüft, ob ein TCP-Port bereits belegt ist.

@param port TCP-Portnummer, die geprüft werden soll
@return True, wenn der Port belegt ist, sonst False
```

6.3 interface-Namensbereichsreferenz

Klassen

- class Interface

6.3.1 Ausführliche Beschreibung

```
@file interface.py
@brief Kommandozeilen-Benutzeroberfläche (CLI) für den SLCP-Chat-Client.
Stellt Eingabe, Ausgabe und Nutzerinteraktion über Terminal bereit.
```

6.4 main-Namensbereichsreferenz

Funktionen

- main ()

6.4.1 Ausführliche Beschreibung

```
@file main.py
@brief Hauptprogramm des P2P-Chat-Clients. Startet Discovery, Messenger, Interface (CLI).
```

6.4.2 Dokumentation der Funktionen

6.4.2.1 main()

```
main.main ()

@brief Hauptfunktion des SLCP-Clients.

Initialisiert alle Hauptkomponenten des Systems:
- Lädt Konfiguration
- Startet den Discovery-Dienst (UDP-basiert)
- Initialisiert die Messenger-Komponente (SLCP)
- Setzt Callback-Funktionen
- Startet die Benutzeroberfläche (Kommandozeile)
- Öffnet den UDP-Listener (für asynchrone Nachrichtenannahme)

Ablauf:
1. Lade Einstellungen wie Handle und Port aus TOML-Datei.
2. Starte Discovery-Service für JOIN/LEAVE/WHO per Broadcast.
3. Starte SLCP-Messenger für Nachrichten- und Bildversand.
4. Starte CLI für Benutzereingaben.

@return None
```

6.5 network.messenger-Namensbereichsreferenz

Klassen

- class Messenger

6.5.1 Ausführliche Beschreibung

```
@file messenger.py
@brief Messenger-Klasse für das SLCP-Chat-Projekt.
@details
    Diese Klasse implementiert die UDP- und TCP-Kommunikation für den Austausch von Nachrichten und Bildern
    im dezentralen Chat. Sie verwaltet Peers, verarbeitet SLCP-Nachrichten und stellt Methoden zum Senden
    und Empfangen bereit.
```

Kapitel 7

Klassen-Dokumentation

7.1 config.config.Config Klassenreferenz

Öffentliche Methoden

- __init__ (self, path="slcp_config.toml")
- load (self)
- save (self)

Öffentliche Attribute

- **path** = path
- **data** = self.load()
- **handle** = self.data.get("handle")
- **port** = int(self.data.get("port") or input("Port: "))
- **whoisport** = int(self.data.get("whoisport") or input("Whois-Port (z.B. 4000): "))
- **autoreply** = self.data.get("autoreply", "")
- **imagepath** = self._setup_imagepath()

Geschützte Methoden

- _setup_imagepath (self)

7.1.1 Ausführliche Beschreibung

```
@class Config
@brief Verwaltet die Konfiguration des Chat-Clients.

Diese Klasse lädt, speichert und verwaltet Benutzereinstellungen aus einer TOML-Datei.
Falls Werte fehlen (z.B. Benutzername oder Port), werden sie beim ersten Start interaktiv abgefragt.
```

7.1.2 Beschreibung der Konstruktoren und Destruktoren

7.1.2.1 __init__()

```
config.config.Config.__init__ (
              self,
              path = "slcp_config.toml")

@brief Initialisiert eine neue Konfigurationsinstanz.

- Liest Konfiguration aus Datei (sofern vorhanden).
- Fragt fehlende Parameter interaktiv ab.
- Erstellt Standardverzeichnisse bei Bedarf.

@param path Dateipfad zur TOML-Konfigurationsdatei (Standard: slcp_config.toml)
```

7.1.3 Dokumentation der Elementfunktionen

7.1.3.1 _setup_imagepath()

```
config.config.Config._setup_imagepath (
              self)  [protected]
```

```
@brief Bereitet den Ordnerpfad für empfangene Bilder vor.

- Nutzt Standardpfad, wenn nicht angegeben.
- Erstellt das Verzeichnis bei Bedarf.

@return Absoluter Pfad zum Bildspeicherordner (z.B. ~/slcp_images)
```

7.1.3.2 load()

```
config.config.Config.load (
              self)
```

```
@brief Lädt die Konfiguration aus der TOML-Datei.

@return Dictionary mit den geladenen Konfigurationswerten.
```

7.1.3.3 save()

```
config.config.Config.save (
              self)
```

```
@brief Speichert die aktuelle Konfiguration zurück in die TOML-Datei.
```

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Datei:

- Chat/config/config.py

7.2 discovery.discovery_service.DiscoveryService Klassenreferenz

Öffentliche Methoden

- __init__ (self, config_path)
- listen (self)
- handle_message (self, message, addr)
- send_who (self)
- get_local_ip (self)
- send_join (self)
- send_leave (self)
- get_peers (self)
- start (self)
- stop (self)

Öffentliche, statische Methoden

- load_config (path)

Öffentliche Attribute

- dict **peers** = {}
- **peers_lock** = threading.Lock()
- bool **running** = True
- **config** = self.load_config(config_path)
- **handle** = self.config["handle"]
- **port** = int(self.config["port"])
- **whois_port** = self.config.get("whoisport", 0)
- **sock** = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

7.2.1 Ausführliche Beschreibung

```
@class DiscoveryService
@brief Implementiert den Discovery-Dienst für das dezentrale Chat-System.

Verwaltet bekannte Peers, sendet und empfängt UDP Broadcast-Nachrichten.
```

7.2.2 Beschreibung der Konstruktoren und Destruktoren

7.2.2.1 __init__()

```
discovery.discovery_service.DiscoveryService.__init__ (
              self,
              config_path)

@brief Konstruktor für den DiscoveryService.

Lädt Konfiguration, initialisiert Variablen und bindet UDP Socket.

@param config_path Pfad zur TOML-Konfigurationsdatei
```

7.2.3 Dokumentation der Elementfunktionen

7.2.3.1 get_local_ip()

```
discovery.discovery_service.DiscoveryService.get_local_ip (
              self)

@brief Ermittelt die lokale IP-Adresse des Hosts.

@return String mit der lokalen IP-Adresse
```

7.2.3.2 get_peers()

```
discovery.discovery_service.DiscoveryService.get_peers (
              self)

@brief Gibt eine Kopie der aktuellen Peer-Liste zurück.

@return Dictionary mit Peers {handle: (IP, Port)}
```

7.2.3.3 handle_message()

```
discovery.discovery_service.DiscoveryService.handle_message (
              self,
              message,
              addr)

@brief Verarbeitet eingehende UDP-Nachrichten gemäß Protokoll.

@param message Empfangene Nachricht als String
@param addr Adresse des Senders (IP, Port)
```

7.2.3.4 listen()

```
discovery.discovery_service.DiscoveryService.listen (
              self)

@brief Endlosschleife zum Empfangen und Verarbeiten von UDP-Nachrichten.

Läuft in eigenem Thread, verarbeitet JOIN, LEAVE, WHO und KNOWNUSERS Nachrichten.
```

7.2.3.5 load_config()

```
discovery.discovery_service.DiscoveryService.load_config (
              path)  [static]
```

@brief Lädt eine TOML-Konfigurationsdatei.

@param path Pfad zur TOML-Datei
@return Dictionary mit Konfigurationsdaten oder leeres Dict bei Fehler

7.2.3.6 send_join()

```
discovery.discovery_service.DiscoveryService.send_join (
              self)
```

@brief Sendet eine JOIN-Nachricht als Broadcast, um sich anzumelden.

7.2.3.7 send_leave()

```
discovery.discovery_service.DiscoveryService.send_leave (
              self)
```

@brief Sendet eine LEAVE-Nachricht als Broadcast, um sich abzumelden.

7.2.3.8 send_who()

```
discovery.discovery_service.DiscoveryService.send_who (
              self)
```

@brief Sendet eine WHO-Anfrage als UDP-Broadcast, um bekannte Peers abzufragen.

7.2.3.9 start()

```
discovery.discovery_service.DiscoveryService.start (
              self)
```

@brief Startet den Discovery-Service: Listener-Thread, JOIN und WHO senden.

7.2.3.10 stop()

```
discovery.discovery_service.DiscoveryService.stop (
              self)
```

@brief Stoppt den Discovery-Service, sendet LEAVE und schließt das Socket.

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Datei:

- Chat/discovery/discovery_service.py

7.3 interface.Interface Klassenreferenz

Öffentliche Methoden

- __init__ (self, config, messenger)
- run (self)
- display_message (self, sender_display, message)
- display_image_notice (self, sender, filename)
- display_knownusers (self, user_list)

Öffentliche Attribute

- **config** = config
- **messenger** = messenger

7.3.1 Ausführliche Beschreibung

```
@class Interface
@brief CLI-basierte Benutzeroberfläche für den SLCP-Chat-Client.

Diese Klasse stellt die Interaktion des Nutzers mit dem SLCP-Chat über ein Konsoleninterface bereit.
Sie verwaltet alle Benutzereingaben, interpretiert SLCP-Befehle wie /join, /leave, /msg usw.
und leitet sie asynchron an die Messenger-Komponente zur Verarbeitung weiter.

Zusätzlich wird die farbliche Konsolenausgabe mit dem Modul `colorama` unterstützt,
um Statusnachrichten und Befehle übersichtlicher darzustellen.
```

7.3.2 Beschreibung der Konstruktoren und Destruktoren

7.3.2.1 __init__()

```
interface.Interface.__init__ (
              self,
              config,
              messenger)

@brief Konstruktor der Interface-Klasse.

Initialisiert das Interface mit Konfigurationsdaten und der Messenger-Instanz
zur Netzwerkkommunikation.

@param config Ein Konfigurationsobjekt mit Nutzername, Port, Autoreply etc.
@param messenger Eine Messenger-Instanz, die SLCP-Nachrichten verarbeitet und verschickt
```

7.3.3 Dokumentation der Elementfunktionen

7.3.3.1 display_image_notice()

```
interface.Interface.display_image_notice (
              self,
              sender,
              filename)

@brief Zeigt eine Benachrichtigung über ein empfangenes Bild an.

Diese Methode wird als Callback bei erfolgreichem Empfang eines Bildes aufgerufen.

@param sender Handle oder IP des Absenders
@param filename Pfad zur lokal gespeicherten Bilddatei
```

7.3.3.2 display_knownusers()

```
interface.Interface.display_knownusers (
              self,
              user_list)

@brief Gibt alle bekannten/erkannten Nutzer formatiert auf der Konsole aus.

Diese Methode wird nach Empfang einer KNOWNUSERS-Nachricht genutzt, um
alle bekannten Peers in der Benutzeroberfläche anzuzeigen.

@param user_list Liste von Tupeln: (handle, ip, port)
```

7.3.3.3 display_message()

```
interface.Interface.display_message (
              self,
              sender_display,
              message)

@brief Zeigt eine empfangene Textnachricht in der Konsole an.

Wird vom Messenger aufgerufen, wenn eine neue Nachricht vom Netzwerk empfangen wurde.

@param sender_display Der Anzeigename oder die IP-Adresse des Absenders
@param message Die empfangene Textnachricht
```

7.3.3.4 run()

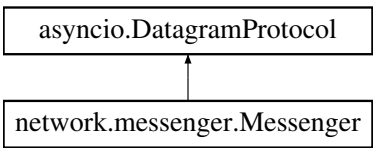
```
interface.Interface.run (
              self)

@brief Startet die Haupt-Eingabeschleife für den Nutzer.

Die Methode zeigt verfügbare Befehle an, liest Eingaben von der Konsole
(z.B. /join, /msg, /img), prüft diese auf Gültigkeit und ruft entsprechende
Messenger-Methoden zur Verarbeitung auf. Sie läuft bis der Befehl /quit ausgeführt wird.
```

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Datei:

- Chat/client/interface.py

7.4 network.messenger.Messenger Klassenreferenz

Klassendiagramm für network.messenger.Messenger:



Öffentliche Methoden

- __init__ (self, config)
- start_listener (self)
- connection_made (self, transport)
- datagram_received (self, data, addr)
- handle_message (self, message, addr)
- send_slcp (self, line, ip, port)
- send_broadcast (self, line)
- send_join (self)
- send_leave (self)
- send_who (self)
- send_message (self, handle, message)
- send_image (self, handle, filepath)
- send_image_data (self, tcp_socket, img_bytes, handle)
- start_tcp_server (self)
- handle_tcp_connection (self, reader, writer)
- receive_image_data (self, reader, addr, size, sender_handle)
- set_progress_callback (self, callback)
- set_message_callback (self, callback)

- set_image_callback (self, callback)
- set_knownusers_callback (self, callback)
- handle_knownusers_response (self, message, addr)
- send_known_to (self, ip, port)
- get_local_ip (self)

Öffentliche Attribute

- **config** = config
- dict **peers** = {}
- **transport** = None
- **message_callback** = None
- **image_callback** = None
- **knownusers_callback** = None
- **progress_callback** = None
- dict **pending_who_responses** = {}
- float **who_timeout** = 2.0

7.4.1 Ausführliche Beschreibung

```
@class Messenger
@brief Messenger für den Versand und Empfang von Chat-Nachrichten und Bildern.
@details
    Verwaltet alle UDP- und TCP-Kommunikationsprozesse, Peer-Liste, sowie das Senden und Empfangen von Nachric
    Stellt zudem verschiedene Callback-Funktionen für die Interaktion mit der Benutzeroberfläche bereit.
```

7.4.2 Beschreibung der Konstruktoren und Destruktoren

7.4.2.1 __init__()

```
network.messenger.Messenger.__init__ (
              self,
              config)
```

```
@brief Konstruktor der Messenger-Klasse
@param config Konfigurationsobjekt mit folgenden Attributen:
    - handle: Benutzername
    - port: Port für TCP/UDP Kommunikation
    - whoisport: Port für WHO-Broadcasts
    - imagepath: Pfad zum Speichern empfangener Bilder
    - autoreply: Automatische Antwort (optional)
```

7.4.3 Dokumentation der Elementfunktionen

7.4.3.1 connection_made()

```
network.messenger.Messenger.connection_made (
              self,
              transport)
```

```
@brief Wird aufgerufen, wenn die UDP-Verbindung erfolgreich hergestellt wurde.
@param transport Das UDP-Transport-Objekt
```

7.4.3.2 datagram_received()

```
network.messenger.Messenger.datagram_received (
              self,
              data,
              addr)
```

```
@brief Wird aufgerufen, wenn eine UDP-Nachricht empfangen wird.
@param data Empfangene Bytes (Nachricht)
@param addr Adresse des Absenders als Tupel (IP, Port)
```

7.4.3.3 get_local_ip()

```
network.messenger.Messenger.get_local_ip (
              self)

@brief Ermittelt die lokale IP-Adresse.
@return Die lokale IP-Adresse als String
```

7.4.3.4 handle_knownusers_response()

```
network.messenger.Messenger.handle_knownusers_response (
              self,
              message,
              addr)

@brief Verarbeitet KNOWNUSERS-Antworten.
@param message Die vollständige KNOWNUSERS-Nachricht
@param addr Absender-Adresse als (ip, port) Tupel
@details Parst die Benutzerliste, aktualisiert die Peer-Informationen
        und ruft den entsprechenden Callback auf. Sammelt Antworten
        für eine kurze Zeit um mehrfache Antworten zu konsolidieren.
```

7.4.3.5 handle_message()

```
network.messenger.Messenger.handle_message (
              self,
              message,
              addr)

@brief Verarbeitet empfangene SLCP-Nachrichten.
@param message Die dekodierte Nachricht als String
@param addr Absender-Adresse als (ip, port) Tupel
@details Parst SLCP-Befehle und führt entsprechende Aktionen aus:
    - JOIN: Neuen Peer registrieren
    - LEAVE: Peer entfernen
    - WHO: Bekannte Benutzer senden
    - KNOWNUSERS: Benutzerliste verarbeiten
    - MSG: Private Nachricht empfangen
    - IMG: Bildübertragung initialisieren
```

7.4.3.6 handle_tcp_connection()

```
network.messenger.Messenger.handle_tcp_connection (
              self,
              reader,
              writer)

@brief Behandelt eingehende TCP-Verbindungen für Bildempfang.
@param reader StreamReader für eingehende Daten
@param writer StreamWriter für ausgehende Daten
@details Liest IMG-Befehle und empfängt Bilddaten, speichert diese
         lokal und ruft Image-Callbacks auf.
```

7.4.3.7 receive_image_data()

```
network.messenger.Messenger.receive_image_data (
              self,
              reader,
              addr,
              size,
              sender_handle)
```

```
@brief Empfängt Bilddaten über TCP und speichert sie.
@param reader StreamReader für die Datenübertragung
@param addr Absender-Adresse als (ip, port) Tupel
@param size Erwartete Dateigröße in Bytes
@param sender_handle Benutzername des Absenders
@return Dateiname der gespeicherten Datei oder None bei Fehler
@details Empfängt Daten in Chunks, zeigt Fortschritt an und speichert
        das Bild mit einem eindeutigen Dateinamen.
```

7.4.3.8 send_broadcast()

```
network.messenger.Messenger.send_broadcast (
              self,
              line)
```

```
@brief Sendet eine SLCP-Broadcast-Nachricht an alle Teilnehmer im lokalen Netzwerk.
@param line Die zu sendende SLCP-Nachricht (String)
```

7.4.3.9 send_image()

```
network.messenger.Messenger.send_image (
              self,
              handle,
              filepath)
```

```
@brief Sendet ein Bild an einen bestimmten Peer via TCP.
@param handle Der Ziel-Benutzername
@param filepath Pfad zur Bilddatei
@return True bei Erfolg, False bei Fehler
@details Überprüft die Datei auf Gültigkeit, öffnet eine TCP-Verbindung
         zum Ziel-Peer und überträgt das Bild in Chunks.
```

7.4.3.10 send_image_data()

```
network.messenger.Messenger.send_image_data (
              self,
              tcp_socket,
              img_bytes,
              handle)
```

```
@brief Sendet die Binärdaten eines Bildes über einen TCP-Socket und ruft Progress-Callbacks auf.
@param tcp_socket Offener TCP-Socket
@param img_bytes Bilddaten als Byte-Array
@param handle Ziel-Handle des Empfängers (für Progress-Callback)
```

7.4.3.11 send_join()

```
network.messenger.Messenger.send_join (
              self)
```

```
@brief Sendet eine JOIN-Nachricht per UDP-Broadcast, um dem Chat beizutreten.
```

7.4.3.12 send_known_to()

```
network.messenger.Messenger.send_known_to (
              self,
              ip,
              port)
```

```
@brief Sendet bekannte Benutzer als Antwort auf WHO-Anfrage.
@param ip Ziel-IP-Adresse
@param port Ziel-Port
@details Erstellt eine KNOWNUSERS-Nachricht mit allen bekannten Peers
        inklusive eigener Informationen und sendet diese direkt an
        den anfragenden Peer.
```

7.4.3.13 send_leave()

```
network.messenger.Messenger.send_leave (
              self)

@brief Sendet eine LEAVE-Nachricht per UDP-Broadcast, um den Chat zu verlassen.
```

7.4.3.14 send_message()

```
network.messenger.Messenger.send_message (
              self,
              handle,
              message)

@brief Sendet eine Textnachricht an einen bestimmten Peer.
@param handle Ziel-Handle (Benutzername) des Empfängers
@param message Die zu sendende Nachricht (String)
```

7.4.3.15 send_slcp()

```
network.messenger.Messenger.send_slcp (
              self,
              line,
              ip,
              port)

@brief Sendet eine SLCP-Nachricht (UDP) an die angegebene Zieladresse.
@param line SLCP-formatierte Nachricht (String)
@param ip Ziel-IP-Adresse
@param port Ziel-Portnummer
```

7.4.3.16 send_who()

```
network.messenger.Messenger.send_who (
              self)

@brief Sendet eine WHO-Nachricht, um die Liste aktiver Teilnehmer zu erfragen.
```

7.4.3.17 set_image_callback()

```
network.messenger.Messenger.set_image_callback (
              self,
              callback)

@brief Setzt den Callback für empfangene Bilder.
@param callback Funktion mit Signatur: (sender_handle, filename)
```

7.4.3.18 set_knownusers_callback()

```
network.messenger.Messenger.set_knownusers_callback (
              self,
              callback)

@brief Setzt den Callback für Benutzerlisten.
@param callback Async-Funktion mit Signatur: (users_list)
    users_list ist eine Liste von (handle, ip, port) Tupeln
```

7.4.3.19 set_message_callback()

```
network.messenger.Messenger.set_message_callback (
              self,
              callback)

@brief Setzt den Callback für empfangene Nachrichten.
@param callback Async-Funktion mit Signatur: (sender_handle, message)
```

7.4.3.20 set_progress_callback()

```
network.messenger.Messenger.set_progress_callback (
              self,
              callback)

@brief Setzt den Callback für Übertragungsfortschritt.
@param callback Funktion mit Signatur: (direction, handle, progress, bytes_transferred, total_bytes)
    - direction: "send" oder "receive"
    - handle: Benutzername des Partners
    - progress: Fortschritt in Prozent (0-100)
    - bytes_transferred: Übertragene Bytes
    - total_bytes: Gesamtanzahl Bytes
```

7.4.3.21 start_listener()

```
network.messenger.Messenger.start_listener (
              self)

@brief Startet den UDP-Listener und den TCP-Server.
@details
    Öffnet den UDP-Socket für Nachrichtenempfang und startet
    parallel den TCP-Server für den Empfang von Bildern.
```

7.4.3.22 start_tcp_server()

```
network.messenger.Messenger.start_tcp_server (
              self)

@brief Startet den TCP-Server zum Empfang von eingehenden Bildübertragungen.
```

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Datei:

- Chat/network/messenger.py

Kapitel 8

Datei-Dokumentation

8.1 Chat/common/protocol.py-Dateireferenz

Enthält Funktionen zum Parsen und Erzeugen von SLCP-Protokollnachrichten.

Funktionen

- common.protocol.parse_slcp (line)
- common.protocol.create_join (handle, port)
- common.protocol.create_leave (handle)
- common.protocol.create_msg (target, text)
- common.protocol.create_img (target, size)
- common.protocol.create_who ()
- common.protocol.create_knownusers (users)

8.1.1 Ausführliche Beschreibung

Enthält Funktionen zum Parsen und Erzeugen von SLCP-Protokollnachrichten.
Diese Datei stellt das Kommunikationsprotokoll bereit, das vom Messenger und Discovery-Service verwendet wird, um Text- und Bildnachrichten sowie Netzwerkanfragen zu senden und zu empfangen.

8.1.2 Dokumentation der Funktionen

8.1.2.1 create_img()

```
common.protocol.create_img (
              target,
              size)

@brief Erstellt eine IMG-Nachricht zum Senden von Bilddaten.

@param target Empfänger-Handle
@param size Größe des Bildes in Bytes
@return SLCP-konforme IMG-Zeile
```

8.1.2.2 create_join()

```
common.protocol.create_join (
              handle,
              port)

@brief Erstellt eine JOIN-Nachricht.

@param handle Benutzername
@param port Portnummer
@return SLCP-konforme JOIN-Zeile
```

8.1.2.3 create_knownusers()

```
common.protocol.create_knownusers (
              users)

@brief Erstellt eine KNOWNUSERS-Nachricht mit Liste aller bekannten Peers.

@param users Liste von Dictionaries mit Schlüsseln 'handle', 'ip', 'port'
@return SLCP-konforme KNOWNUSERS-Zeile
```

8.1.2.4 create_leave()

```
common.protocol.create_leave (
              handle)

@brief Erstellt eine LEAVE-Nachricht.

@param handle Benutzername
@return SLCP-konforme LEAVE-Zeile
```

8.1.2.5 create_msg()

```
common.protocol.create_msg (
              target,
              text)

@brief Erstellt eine MSG-Nachricht zum Senden eines Textes.

@param target Empfänger-Handle
@param text Nachrichtentext
@return SLCP-konforme MSG-Zeile
```

8.1.2.6 create_who()

```
common.protocol.create_who ()

@brief Erstellt eine WHO-Broadcast-Nachricht zur Abfrage aller bekannten Nutzer.

@return SLCP-konforme WHO-Zeile
```

8.1.2.7 parse_slcp()

```
common.protocol.parse_slcp (
              line)

@brief Parst eine SLCP-Zeile (Simple Local Chat Protocol) in ein Dictionary.

Unterstützte Befehle:
- JOIN <handle> <port>
- LEAVE <handle>
- WHO
- MSG <to> <message>
- IMG <to> <size>
- KNOWNUSERS <handle1> <ip1> <port1>, ...

@param line SLCP-Zeile als String
@return Dictionary mit Schlüssel "type" und weiteren Feldern je nach Befehl
```