Version 1 – Projekt ohne IPC – Branch „NO_IPC_Version“ (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 erste Variante des Projekts, die Version ohne richtige IPC.
Die Dokumentation für die IPC-Version finden Sie in einem separaten Dokument.

Aufbau

Der Programmcode lässt sich in 6 Hauptbestandteile aufteilen. Sie sind in der Ordnerstruktur innerhalb des Projekts gut erkennbar. Im Folgenden wird diese Struktur aufgeschlüsselt und erklärt.

1. Main.py
2. Core
3. Cli
4. resources
5. config
6. gui

Teilprobleme

main.py

Dieses Modul bildet den zentralen Einstiegspunkt der Anwendung. Es verwaltet das Initialisieren der Konfigurationsdatei und steuert den Startprozess der Benutzeroberfläche.

initialize_user()

Diese Methode prüft beim Start des Programms, ob eine Konfigurationsdatei (config.toml) existiert. Falls diese nicht vorhanden ist, wird sie automatisch mit Standardwerten erstellt. Dazu gehören ein leerer Benutzername, ein Standard Empfangsport (5001), ein Standardordner für empfangene Bilder und ein Auto-Reply Text für Abwesenheiten. Anschließend wird der Benutzername aus der Datei gelesen. Ist dieser leer oder nicht gesetzt, wird der Benutzer aufgefordert, seinen Namen einzugeben. Der eingegebene Name wird dauerhaft in der Konfigurationsdatei gespeichert. Damit ist eine Wiedererkennung des Nutzers bei zukünftigen Starts gewährleistet.

main()

Diese Methode wird automatisch aufgerufen, wenn das Skript direkt ausgeführt wird. Nach einer Startmeldung ruft sie initialize_user() auf, um sicherzustellen, dass ein gültiger Benutzername existiert. Anschließend wird der Benutzer gefragt, ob er die Anwendung im GUI-Modus (Option 1) oder im CLI-Modus (Option 2) starten möchte. Der GUI-Modus ist aktuell noch nicht implementiert. Wird die CLI gewählt, wird das Modul cli_chat gestartet. Bei einer ungültigen Eingabe wird eine Fehlermeldung angezeigt.

Core

Der nächste große Hauptbestandteil des Projekts ist „core“. Hier sind alle Kernfunktionen des Programms enthalten, die grundlegende Aufgaben im Programm erfüllen. Zum einen ist hier der Discovery/Network-Service zu finden. Zum anderen sind hier auch Dinge wie das SLCP-Protokoll und der image-handler. Mehr dazu gleich.

discovery.py

Dieses Python-Modul stellt einen Discovery-Service für ein Peer-to-Peer-Chat-System bereit. Es erlaubt es Teilnehmern, sich gegenseitig im lokalen Netzwerk zu erkennen, indem sie einfache UDP-Nachrichten austauschen. Ziel ist eine dezentrale Benutzererkennung ohne zentrale Server. Die Kommunikation basiert auf drei Hauptbefehlen: JOIN, LEAVE und WHO.

Nach dem Laden einer Konfigurationsdatei im TOML-Format oder dem Zurückgreifen auf Standardwerte startet ein Hintergrunddienst, der Nachrichten empfängt und verarbeitet.

broadcast(msg)

Diese Methode versendet eine Nachricht als UDP-Broadcast an alle Hosts im lokalen Netzwerk. Sie erstellt einen temporären Socket, aktiviert den Broadcast-Modus (SO_BROADCAST) und sendet die Nachricht an die Adresse 255.255.255.255 und den konfigurierten Port (WHOIS_PORT).

Zweck: Neue Teilnehmer im Netzwerk ankündigen (JOIN), oder ihren Austritt bekannt machen (LEAVE).

send_direct(ip, port, message)

Diese Funktion sendet eine gezielte UDP-Nachricht an einen bestimmten Teilnehmer im Netzwerk, unter Angabe von IP-Adresse und Port.

Zweck: Zum Beispiel die gezielte Antwort auf eine WHO-Anfrage mit der Liste aller bekannten Nutzer.

handle_join(tokens, addr)

Diese Methode verarbeitet eine eingehende JOIN-Nachricht. Erwartet werden genau drei Elemente: das Wort JOIN, ein Handle (Benutzername) und ein Port.

Der Teilnehmer wird anschließend zur globalen Teilnehmerliste participants hinzugefügt, unter Angabe seiner IP-Adresse und Portnummer.

Beispiel: JOIN Alice 5002 → Alice wird als Teilnehmer registriert.

Fehlerbehandlung: Ungültige oder unvollständige Nachrichten (z. B. kein Port) werden ignoriert.

handle_leave(tokens, addr)

Wird eine LEAVE-Nachricht empfangen, löscht diese Methode den entsprechenden Teilnehmer aus der Liste. Die Nachricht enthält den Befehl LEAVE sowie den Handle des Benutzers.

Beispiel: LEAVE Alice → Alice wird aus der Teilnehmerliste entfernt, sofern sie dort eingetragen ist.

handle_who(addr)

Diese Funktion reagiert auf WHO-Anfragen von Teilnehmern, die wissen wollen, wer im Netzwerk aktiv ist.

Zunächst wird überprüft, ob die IP-Adresse des Anfragenden bereits bekannt ist. Falls ja, wird eine Nachricht vom Typ KNOWNUSERS an ihn zurückgesendet, die alle bekannten Teilnehmer (Handle, IP, Port) enthält.

Hinweis: Nur bekannte Teilnehmer (bereits durch JOIN erfasst) erhalten eine Antwort.listen()

Der zentrale Listener für Discovery-Nachrichten. Diese Methode öffnet einen UDP-Socket auf dem konfigurierten Discovery-Port (WHOIS_PORT) und lauscht dauerhaft auf eingehende Nachrichten.

Empfangene Daten werden dekodiert, in Tokens zerlegt und entsprechend ihrem Befehl (JOIN, LEAVE, WHO) an die zugehörige Methode weitergeleitet.

Fehlertoleranz: Netzwerkfehler führen nicht zum Abbruch – alle Fehler werden abgefangen und gemeldet.

start_discovery_service()

Diese Funktion startet den Listener (listen) in einem eigenen Hintergrund-Thread. Der Thread läuft als Daemon, was bedeutet, dass er automatisch beendet wird, sobald das Hauptprogramm endet.

Verwendung: Wird einmal zu Beginn des Chat-Programms aufgerufen, um Discovery dauerhaft im Hintergrund zu aktivieren.

send_join(handle, port)

Wird beim Start des Programms aufgerufen, um sich im Netzwerk anzumelden. Die Methode erstellt eine JOIN-Nachricht mit Benutzername und Port und sendet sie über broadcast() an alle anderen Instanzen im lokalen Netz.

Zweck: Andere Teilnehmer werden dadurch auf den neuen Nutzer aufmerksam und können ihn registrieren.

Globale Variable participants

Dies ist ein zentrales Dictionary, das alle bekannten Teilnehmer speichert.

- Key: Benutzername (Handle)
- Value: Tuple mit IP-Adresse und Port
- Diese Datenstruktur wird von allen Handlern verwendet und laufend aktualisiert (JOIN/LEAVE).

---

image_handler.py

Diese Datei enthält die Funktionalität, Bilder zwischen den Clients auszutauschen. Es können Bilder versendet, empfangen, gespeichert und geöffnet werden.

Die Datei enthält zwei wichtige Funktionen. Einerseits „send_image(image_path, target_ip, target_port)“. Diese Methode überprüft zuerst, ob das gemeinte Bild existiert. Falls das nicht der Fall ist, wird eine Fehlermeldung ausgegeben. Wenn das Bild aber existiert, wird die Bilddatei eingelesen und die Länge sowie der Name herausgelesen. Diese Informationen werden zusammengesetzt und als Header an den Empfänger per UDP versendet. Hierzu wird zuerst ein Socket erstellt.

Im Anschluss wird die Bilddatei in kleinere Chunks unterteilt. Die Größe der Chunks ist durch die Variable „BUFFER_SIZE“ festgelegt und wurde aufgrund der Projektanforderungen fest einprogrammiert. Diese Chunks werden nach und nach über den zuvor erstellten Socket abgesendet.

Die Funktion „receive_image(callback=None, port=6000, save_dir=IMAGEPATH)” ist für das Empfangen von Bildern da. Diese Funktion wird durch einen Daemon zu Beginn des Programms ausgeführt, und hört permanent auf dem IMAGEPORT (Siehe Übergabeparameter in slcp.py). In einer Endlosschleife wird geprüft, ob ankommende Nachrichten mit „IMG“ beginnen. Hierbei handelt es sich dann um den zuvor erklärten Header. Er wird aufgeschlüsselt in „image_name“ und „image_size“.

Anschließend wird das Bild über den Socket in Chunks empfangen und unter dem angegebenen Bildpfad abgelegt. Als Bedingung für die while-Schleife wird hier die Größe des empfangenen Bildes verwendet. Die Größe aus dem Header muss der Größe des empfangenen Bildes exakt entsprechen.

Letztendlich versucht die Methode, das Bild zu öffnen. Hierfür gibt es abhängig vom Betriebssystem verschiedene Wege. Tritt beim Empfang des Bildes ein Fehler auf, wird dieser abgefangen und als Fehlermeldung ausgegeben.

---

slcp.py

Die Klasse SLCPChat implementiert einen einfachen Chat über das Simple Lightweight Chat Protocol (SLCP) mithilfe von UDP. Teilnehmer können über IP und Port Nachrichten im lokalen Netzwerk austauschen.

Die Klasse unterstützt parallelen Empfang über Threads, erlaubt das Senden an mehrere Peers und bietet eine Callback-Funktion zur flexiblen Weiterverarbeitung empfangener Nachrichten.

Konstruktor:

__init__(username, listen_port, peers)

Beim Instanziieren werden die folgenden Parameter benötigt:

- username: Der Anzeigename dieses Clients
- listen_port: Der UDP-Port, auf dem Empfangen wird
- peers: Eine Liste von bekannten Zielteilnehmern im Format (IP, Port)

Zusätzlich wird ein UDP-Socket erstellt und an "0.0.0.0" gebunden, um eingehende Nachrichten von allen Netzwerkschnittstellen zu empfangen.

start()

Startet die Empfangsschleife in einem separaten Hintergrund-Thread. Dadurch kann das Hauptprogramm weiterlaufen, während parallel auf neue Nachrichten gewartet wird. Der Thread ruft die Methode listen_loop() dauerhaft auf.

stop()

Beendet den Chat-Client, indem self.running auf False gesetzt und der Socket geschlossen wird.

Ein try-except-Block verhindert, dass ein Fehler beim Schließen des Sockets das Programm unterbricht.

register_callback(callback)

Erlaubt das Setzen einer Funktion, die auf empfangene Nachrichten reagieren soll. Diese Callback-Funktion muss zwei Parameter erwarten: sender (Name des Absenders) und message (Inhalt der Nachricht).

Beispiel:
def on_message(sender, msg):
print(f"{sender} sagt: {msg}")
chat.register_callback(on_message)
send_message(message)

Diese Methode wird verwendet, um eine Nachricht an alle bekannten Peers zu senden.

1. Nachricht wird als Username:Nachricht formatiert.
2. Nachricht wird in UTF-8 kodiert.
3. Sie wird dann an jede (IP, Port)-Kombination in der Peer-Liste per UDP versendet.

Fehlertoleranz: Fehler beim Senden an einen bestimmten Peer werden protokolliert, aber der Vorgang läuft weiter.

listen_loop()

Diese Methode läuft dauerhaft im Hintergrund und wartet auf eingehende UDP-Nachrichten.

- Der empfangene Byte-Stream wird dekodiert.
- Wenn die Nachricht einen Doppelpunkt enthält, wird sie als Username:Nachricht interpretiert.
- Die Callback-Funktion (falls vorhanden) wird mit den extrahierten Daten aufgerufen.

Fehlerschutz: Sollte während des Empfangs ein Fehler auftreten (z. B. Netzwerkunterbrechung), wird dieser nur gemeldet, wenn self.running noch aktiv ist.

Datenfluss

1. Empfang: Nachrichten kommen über den UDP-Socket rein, werden dekodiert und verarbeitet.
2. Verarbeitung: Die Callback-Funktion erhält sender und message.
3. Versand: Nachrichten werden an alle bekannten Peers über send_message() geschickt.

---

udp_handler.py

Die Datei udp_handler.py stellt eine gekapselte Lösung für das Senden und Empfangen von UDP-Nachrichten im Rahmen des SLCP-Chatprotokolls dar. Sie abstrahiert die zugrunde liegende Netzwerkkommunikation, indem sie alle UDP-bezogenen Aufgaben in einer eigenen Klasse organisiert: UDPHandler.

Diese Klasse wird sowohl im CLI- als auch im GUI-Kontext verwendet und bildet damit eine zentrale Kommunikationsschnittstelle im Programm. Durch den Einsatz eines separaten Empfangsthreads bleibt die Hauptanwendung jederzeit reaktiv.

Klasse: UDPHandler

Konstruktor:

__init__(listen_port, on_receive_callback)

Beim Erstellen einer neuen Instanz von UDPHandler werden zwei Parameter benötigt:

- listen_port: Gibt an, auf welchem Port das Programm eingehende UDP-Nachrichten erwartet.
- on_receive_callback: Eine Callback-Funktion, die automatisch aufgerufen wird, sobald eine Nachricht empfangen wurde.

Zweck:

Durch diese Konfiguration kann UDPHandler universell eingesetzt werden – z. B. für Textnachrichten, Systembefehle oder Bildübertragungen.

start()

Startet einen neuen Daemon-Thread, der im Hintergrund dauerhaft auf eingehende UDP-Nachrichten wartet. Der Empfang erfolgt nicht blockierend, d. h. die Hauptanwendung wird nicht unterbrochen.

Technisch:

Thread startet die interne Methode _receive_loop()
Der Empfangssocket wird in diesem Thread geöffnet und dauerhaft überwacht

stop()

Beendet den Empfangsthread, indem das interne Flag self.running auf False gesetzt wird. Dadurch verlässt die Methode _receive_loop() ihre Endlosschleife.

Hinweis:

Der Thread selbst wird als Daemon ausgeführt – d. h. er wird auch automatisch beendet, wenn das Hauptprogramm endet.

send_message(message, peers)

Ermöglicht das Versenden von UDP-Nachrichten an eine Liste von Zielteilnehmern.

- message: Der zu versendende Text (als String)
- peers: Eine Liste von (IP, Port)-Tupeln

Für jeden Peer wird ein eigener Socket erzeugt, die Nachricht als UTF-8 kodiert und direkt versendet.

Verwendung:
Diese Methode wird typischerweise verwendet, um Nachrichten an mehrere Teilnehmer gleichzeitig zu senden – z. B. beim Senden eines Chat-Posts oder einer systemweiten Benachrichtigung.

_receive_loop()

Dies ist die zentrale Empfangsschleife, die im Hintergrund läuft.

Ablauf:

Öffnet einen UDP-Socket auf dem konfigurierten Port

Wartet mit Timeout auf eingehende Pakete

Dekodiert empfangene Daten (UTF-8)

Übergibt den Textinhalt an die übergebene Callback-Funktion

Fehlertoleranz:

Die Methode arbeitet mit einem Timeout (1 Sekunde), um kontrolliert auf self.running prüfen zu können. Netzwerkfehler wie leere Pakete oder Timeouts führen nicht zum Abbruch.

Fazit

udp_handler.py stellt eine robuste und modulare Lösung für die UDP-Kommunikation im BSRN-Projekt dar. Die Trennung in einen Empfangs- und Sendeprozess über Threads erlaubt eine einfache Parallelisierung der Netzwerkaktivitäten. Durch die Übergabe einer

Callback-Funktion bleibt die Klasse zudem flexibel und erweiterbar – egal ob für Text, Steuerbefehle oder Bildübertragungen.

Wenn du möchtest, kann ich dir diesen Abschnitt direkt so formatieren, dass du ihn in euer Gesamtdokument übernehmen kannst – inkl. Seitenverweis oder Inhaltsverzeichnis-Eintrag. Sag einfach Bescheid!

Cli

Cli_chat.py

In dieser Datei wird alles rund um die Chat-Funktionalität in der Kommandozeile geregelt.

Diese Datei enthält drei wichtige Funktionen. Zuerst die Funktion „leave_chat()“. Hier wird die „leave()“-Funktion von SLCP-Chat ausgeführt. Diese Methode kümmert sich um die Netzwerkkommunikation und sendet die Broadcast-Nachricht an alle Netzwerkteilnehmer. Außerdem versendet die Funktion „leave_chat()“ eine Benachrichtigung in die Konsole, welche den Nutzer über die Aktion benachrichtigt.

Die Funktion wird durch die Bibliothek „atextit“ aufgerufen, welche immer beim Schließen des Programms ausgeführt wird. Es stellt sicher, dass beim (unerwarteten) beenden des Programms immer eine „LEAVE“-Nachricht abgeschickt wird.

Eine weitere Funktion, welche in „cli_chat.py“ realisiert wird, ist „on_message(message, addr)“. Diese Funktion wird permanent ausgeführt, um empfangene Nachrichten nach ihrem Inhalt zu analysieren.

Beginnt die Nachricht mit „KNOWNUSERS“ – ist also eine Antwort auf eine „WHO“-Anfrage, wird der Rest der Nachricht formatiert, in der „users“-Liste gespeichert und mithilfe der „PrettyTable“-Tabelle ausgegeben.

Falls die Nachricht nicht „KNOWNUSERS“ ist, wird die Nachricht einfach in der Konsole per „print()“ ausgegeben.

Anschließend prüft die Methode noch, ob in der Konfigurationsdatei gerade der Abwesenheitsassitent eingeschaltet ist und die „autoreply“-Nachricht abgeschickt werden soll. Wenn dies der Fall ist, wird diese Abwesenheitsnachricht abgeschickt und der Nutzer in der Konsole benachrichtigt.

Die nächste Funktion in „cli_chat.py“ ist „input_loop()“. Diese Funktion wird auch konstant ausgeführt. Sie gibt dem Nutzer immer die Möglichkeit, Befehle in die Konsole einzugeben.

Die Eingabe wird anschließend nach dem ersten Wort analysiert. Je nachdem welcher Befehl hier steht, wird die entsprechende Aktion ausgeführt. Mögliche Befehle sind hier „JOIN“, „LEAVE“, „WHO“, „MSG“, „IMG“, „CONFIG“.

Bei „CONFIG“ gibt es eine ganze Auswahl an Unterbefehlen, welche der Nutzer verwenden kann, um den Inhalt der

Ressources

config.toml

Die Konfigurationsdatei des Chatprogramms ist im TOML-Format verfasst und unterteilt sich in drei Hauptbereiche: [user], [image] und [network]. Jeder Abschnitt definiert Einstellungen, die das Verhalten der Anwendung beeinflussen und lässt sich flexibel anpassen.

[user] – Benutzereinstellungen

handle = "Sumaya“

Definiert den Namen, unter dem der Benutzer im Netzwerk sichtbar ist. Dieser Name wird z. B. beim Versenden von Nachrichten angezeigt.

port = [5001]

Eine Liste von Ports, über die der Benutzer Textnachrichten empfangen kann. In der Regel wird nur der erste Port verwendet.

imageport = 6000

Ein separater UDP-Port speziell für den Empfang von Bilddateien.

inactive = false

Steuert die Erreichbarkeit des Benutzers. Ist der Wert auf true gesetzt, gilt der Benutzer als inaktiv.

autoreply = "Hi! Not here."

Wird automatisch als Antwort gesendet, wenn inactive = true ist. Dient als automatische Abwesenheitsmeldung.

[image] – Bildeinstellungen

imagepath = "received_images/"

Gibt das Verzeichnis an, in dem empfangene Bilder gespeichert werden. Der Pfad ist relativ zum Programmverzeichnis.

[network] – Netzwerkeinstellungen

whoisport = 4000

Definiert den Port, auf dem das Programm Discovery-Nachrichten wie JOIN, LEAVE oder WHO empfängt und sendet. Dieser Port wird für die Verwaltung der Teilnehmer im Netzwerk verwendet.

Config

Config_loader.py

Funktion:

Das Modul config_loader.py stellt zentrale Funktionen zum Laden und Speichern der Anwendungskonfiguration bereit. Es agiert als Bindeglied zwischen der Laufzeitumgebung der Chat-Anwendung und der persistenten Konfigurationsdatei config.toml. Dadurch wird gewährleistet, dass Nutzereinstellungen und systemrelevante Parameter dauerhaft gespeichert und beim Programmstart korrekt geladen werden können.

load_config()

Beschreibung:

Diese Funktion überprüft, ob die Datei config.toml im Verzeichnis resources/ vorhanden ist. Ist dies nicht der Fall, wird eine FileNotFoundError-Exception ausgelöst. Andernfalls wird der Inhalt der Datei eingelesen und als Dictionary zurückgegeben.

Ablauf:

Prüft Existenz der Datei über os.path.exists()
Bei Erfolg: Öffnet und parst die Datei mit toml.load()
Rückgabe eines Dictionary-Objekts mit der kompletten Konfiguration

Zweck:

Stellt sicher, dass alle benötigten Konfigurationswerte zu Beginn der Programmausführung geladen werden. Diese umfassen u. a. den Benutzernamen, Ports, Speicherorte für Bilder und Auto-Reply-Nachrichten.

Fehlertoleranz:

Löst eine FileNotFoundError aus, falls die Datei nicht vorhanden ist. Diese Ausnahme kann vom aufrufenden Modul (z. B. main.py) behandelt werden, um im Fehlerfall automatisch eine Standardkonfiguration zu generieren.

Verwendung:

main.py ruft load_config() auf, um initiale Benutzerdaten zu laden.
GUI- und CLI-Komponenten nutzen die Rückgabewerte indirekt zur Konfiguration des Chatverhaltens.

save_config(config)

Beschreibung:

Diese Funktion speichert eine gegebene Konfiguration zurück in die Datei config.toml. Hierbei wird das übergebene Dictionary als TOML-Datei serialisiert.

Ablauf:

Öffnet resources/config.toml im Schreibmodus
Nutzt toml.dump() zur Serialisierung des Dictionaries in TOML-Syntax
Überschreibt die bestehende Datei vollständig

Zweck:

Ermöglicht das dauerhafte Speichern von Benutzereinstellungen. Beispielsweise wird beim ersten Start der Anwendung ein Nutzername festgelegt, der dann per save_config() persistiert wird. Auch Änderungen an Abwesenheitstexten oder Ports können so gesichert werden.

Fehlertoleranz:

Fehler beim Dateizugriff (z. B. fehlende Schreibrechte) werden nicht direkt im Modul behandelt und müssen extern abgefangen werden, falls notwendig.

Verwendung:

Wird von main.py und ggf. anderen Modulen wie cli_chat.py genutzt, um Konfigurationsänderungen des Benutzers dauerhaft zu sichern.

GUI

Dieses Python-Modul implementiert die grafische Benutzeroberfläche der Anwendung mithilfe von PyQt5. Es ermöglicht es Benutzern, Nachrichten einzugeben, zu senden und eingehende Nachrichten im Fenster anzuzeigen. Die Kommunikation erfolgt über das SLCP-Protokoll. Die zentrale Klasse heißt

---

chat_window.py

__init__(self, username, listen_port, peers)

Diese Methode initialisiert das gesamte Chatfenster. Der Benutzername, der Empfangsport und eine Liste von Peers werden übergeben. Die grafische Oberfläche wird aus einer .ui-Datei geladen. Anschließend wird eine SLCPChat-Instanz erstellt, die für den Nachrichtenaustausch verantwortlich ist. Ereignisse wie das Klicken auf den

Senden-Button oder das Drücken der Enter-Taste werden mit der Methode nachricht_senden() verknüpft. Zudem wird eine Callback-Funktion registriert, die automatisch bei empfangenen Nachrichten ausgelöst wird. nachricht_senden(self) Diese Methode wird aufgerufen, wenn der Benutzer eine Nachricht senden möchte. Zuerst wird geprüft, ob überhaupt Text im Eingabefeld vorhanden ist. Falls nicht, passiert nichts. Andernfalls wird die Nachricht rechtsbündig als eigene Nachricht im Chatfenster angezeigt und dann über das Netzwerk an alle definierten Peers gesendet. Die Nachricht wird dabei durch SLCP verarbeitet. Beispiel: Gibt der Benutzer 'Hallo Sumaya' ein, wird 'Sohal: Hallo Sumaya' an 127.0.0.1:4568 geschickt. Anschließend wird das Eingabefeld geleert.

on_message_received(self, sender, message)

Diese Methode wird automatisch aufgerufen, wenn eine Nachricht vom Netzwerk empfangen wurde. Sie übernimmt den Absendernamen und den Textinhalt. Anschließend wird add_chat_bubble() aufgerufen, um die Nachricht im Fenster anzuzeigen. Die Methode nutzt QMetaObject.invokeMethod(), um sicherzustellen, dass die Anzeige im Haupt- GUI-Thread erfolgt – das ist nötig, weil eingehende Nachrichten in einem Neben-Thread verarbeitet werden.

add_chat_bubble(self, text, align="left", color="#e6f2ff")

Diese Methode erzeugt visuelle Nachrichtenelemente in Form von Blasen (Chatbubbles). Je nach Ausrichtung (align=left für fremde Nachrichten, right für eigene) wird die Blase links oder rechts im Nachrichtenfeld angezeigt. Die Farbe der Blase kann angepasst werden. Die Methode verwendet QLabel-Elemente, die dynamisch zum Layout hinzugefügt werden. Am Ende wird das Scroll-Element automatisch nach unten bewegt, sodass neue Nachrichten sichtbar sind.

closeEvent(self, event)

Diese Methode wird beim Schließen des Chatfensters ausgelöst. Sie sorgt dafür, dass der SLCPChat-Client beendet wird, indem self.chat.leave() aufgerufen wird. Dadurch wird der UDP-Socket geschlossen und das Programm sicher beendet.

---

start_sohal.py

Dieses Modul startet die grafische Benutzeroberfläche für Sohal. Es definiert den Benutzernamen, den Empfangsport und die Zieladresse für die Peer-Kommunikation.

__main__

Hier wird eine Qt-Anwendung mit QApplication gestartet. Anschließend wird ein ChatWindow erstellt, das Sohal als Benutzer identifiziert, Nachrichten über Port 4567 empfängt und Nachrichten an Sumaya über Port 4568 auf localhost sendet. Die Anwendung bleibt so lange aktiv, bis das Fenster manuell geschlossen wird.

start_sumaya.py

Dieses Modul startet die grafische Benutzeroberfläche für Sumaya. Es ist inhaltlich identisch zu Sohals Startdatei, aber mit vertauschten Rollen.

__main__

Hier wird eine Qt-Anwendung mit QApplication gestartet. Anschließend wird ein ChatWindow erstellt, das Sumaya als Benutzer identifiziert, Nachrichten über Port 4568 empfängt und Nachrichten an Sohal über Port 4567 auf localhost sendet. Die Anwendung bleibt so lange aktiv, bis das Fenster manuell geschlossen wird.