

Einleitung

Dieses Dokument beschreibt die technische Umsetzung eines Peer-to-Peer (P2P) Chat-Clients. Die Anwendung ermöglicht es Benutzern, in einem lokalen Netzwerk (LAN) miteinander zu kommunizieren, ohne auf einen zentralen Server angewiesen zu sein. Die Kommunikation erfolgt direkt zwischen den einzelnen Clients.

Das System unterstützt:

- Private Nachrichten zwischen zwei Benutzern.
- Gruppenchats, denen Benutzer beitreten und die sie verlassen können.
- Automatische Erkennung anderer Benutzer im Netzwerk.
- Übertragung von Binärdaten (simulierte Bildübertragung).

Architektur

Die Anwendung folgt einer modularen Architektur, die in drei Hauptkomponenten unterteilt ist:

1. **Benutzeroberfläche (`user_interface.py`):** Verarbeitet die Kommandozeileingaben des Benutzers in einem separaten Thread.
2. **Netzwerk-Handler (`network/network_handler.py`):** Kapselt die gesamte Netzwerklogik. Diese Komponente ist verantwortlich für das Senden, Empfangen und Verarbeiten von Nachrichten.
3. **Konfigurationslader (`utils/config_loader.py`):** Ein Hilfsmodul zum Laden von benutzerspezifischen Einstellungen aus einer TOML-Datei.

Netzwerk-Architektur

Die Kommunikation basiert auf UDP-Sockets. Es werden zwei primäre Kommunikationsmechanismen verwendet:

- **Broadcast (UDP):** Wird für Nachrichten verwendet, die an alle Teilnehmer im Netzwerk gerichtet sind. Dies umfasst die Erkennung von Peers (ALIVE, JOIN), das Verlassen von Gruppen (LEAVE) und öffentliche Gruppennachrichten (MSG).
- **Unicast (UDP):** Wird für die direkte Kommunikation zwischen zwei Clients verwendet. Dies betrifft private Nachrichten (MSG) und die Übertragung von Binärdaten (IMG).

Bedienung

Die Anwendung wird über die Kommandozeile gestartet. Optional kann der Pfad zu einer Konfigurationsdatei als Argument übergeben werden.

```
python main.py [config_datei.toml]
```

Nach dem Start können folgende Befehle im Client eingegeben werden:

- `msg <nutzer> <text>`: Sendet eine private Nachricht an einen bestimmten Nutzer.
- `/img <nutzer> <größe>`: Sendet einen Block zufälliger Binärdaten an einen Nutzer.
- `who`: Zeigt alle bekannten Nutzer in der aktiven Gruppe an.
- `/create <gruppe>`: Erstellt eine neue Gruppe und tritt ihr bei.

- `/join <gruppe>`: Tritt einer bestehenden Gruppe bei.
- `/leave <gruppe>`: Verlässt eine Gruppe.
- `/switch <gruppe>`: Wechselt die aktive Gruppe für das Senden von Nachrichten.
- `/groups`: Listet alle Gruppen auf, denen der Benutzer beigetreten ist.
- `exit`: Beendet die Anwendung.

Einleitung

Bei der Entwicklung des Chats sind wir auf ein paar knifflige Probleme gestoßen. Hier beschreiben wir, was die Herausforderungen waren und wie wir sie gelöst haben.

1. Gleichzeitige Aktionen im Programm

Problem: Das Programm muss mehrere Dinge gleichzeitig tun: auf Benutzereingaben warten, Nachrichten empfangen und senden. Am Anfang kam es hier zu Fehlern, weil manchmal zwei Aktionen zur selben Zeit auf die gleichen Daten zugreifen wollten. Zum Beispiel wollte das Programm eine Nachricht an einen Nutzer senden, der im selben Moment aus der Liste gelöscht wurde. Das führte zu Abstürzen.

Lösung: Wir haben eine einfache Regel eingeführt: Immer, wenn ein Programmteil eine Liste von Benutzern durchgeht, arbeitet er mit einer schnellen Kopie dieser Liste. So können andere Programmteile die originale Liste verändern, ohne dass es zu einem Konflikt kommt.

2. Nutzer, die einfach verschwinden

Problem: Wenn ein Nutzer sein Programmfenster einfach schließt, anstatt den `exit`-Befehl zu nutzen, bekommen die anderen das nicht sofort mit. Der " verschwundene " Nutzer blieb als Geist in der Benutzerliste stehen.

Lösung: Jeder Teilnehmer im Chat sendet alle 15 Sekunden automatisch ein kurzes "Ich bin noch da"-Signal. Wenn von einem Nutzer länger als 35 Sekunden kein solches Signal ankommt, wird er automatisch aus der Liste entfernt. So räumt sich die Liste von selbst auf.

3. Namens-Doppelgänger

Problem: Anfangs haben wir Nutzer nur an ihrem Namen erkannt. Das wurde zum Problem, als zwei Nutzer mit dem gleichen Namen im Chat waren. Das Programm konnte sie nicht mehr auseinanderhalten.

Lösung: Jeder Nutzer wird jetzt durch eine unsichtbare, aber einzigartige Kombination aus seiner IP-Adresse und einer Port-Nummer identifiziert. Der Name, den man sieht, ist nur noch eine Art Spitzname.

4. Die Regeln der Kommunikation

Problem: Unser erster Plan, wie die Programme miteinander reden, war zu simpel. Wir hatten nicht bedacht, dass es mehrere Chat-Gruppen geben soll. Als wir das später einbauen wollten, mussten wir die Kommunikationsregeln komplett überarbeiten.

Lösung: Wir haben die Regeln (das "Protokoll") so angepasst, dass bei fast jeder Nachricht dabeisteht, zu welcher Gruppe sie gehört. Das hat die weitere Entwicklung viel einfacher gemacht.

main Namespace Reference

Functions

<code>show_welcome_banner ()</code>
Zeigt ein Willkommensbanner mit grundlegenden Befehlen an.
<code>main ()</code>
Hauptfunktion, die alles ins Rollen bringt.

Function Documentation

<div>◆ <code>main()</code></div> <div><code>main.main ()</code></div> <div>Hauptfunktion, die alles ins Rollen bringt. Lädt die Konfiguration, startet die Benutzeroberfläche und wartet dann, bis der Benutzer das Programm beendet.</div>
<div>◆ <code>show_welcome_banner()</code></div> <div><code>main.show_welcome_banner ()</code></div> <div>Zeigt ein Willkommensbanner mit grundlegenden Befehlen an.</div>

network.network_handler.NetworkHandler Class Reference

Kümmert sich um die gesamte Netzwerkkommunikation. [More...](#)

Public Member Functions

<code>__init__</code> (<code>self</code> , <code>config</code>)
Initialisiert den NetworkHandler .
<code>discover_users</code> (<code>self</code> , <code>group_name=None</code>)
Zeigt die bekannten Benutzer in einer bestimmten Gruppe an.
<code>announce_presence</code> (<code>self</code> , <code>group_name=None</code>)
Sendet eine JOIN-Nachricht, um die eigene Anwesenheit bekannt zu machen.
<code>send_message</code> (<code>self</code> , <code>handle</code> , <code>text</code>)
Sendet eine private Nachricht an einen bestimmten Benutzer.
<code>send_group_message</code> (<code>self</code> , <code>text</code>)
Sendet eine Nachricht an die aktuell aktive Gruppe.
<code>leave_group</code> (<code>self</code> , <code>group_name</code>)
Verlässt eine bestimmte Gruppe.
<code>join_group</code> (<code>self</code> , <code>group_name</code>)
Tritt einer Gruppe bei und macht sie zur aktiven Gruppe.
<code>shutdown</code> (<code>self</code>)
Führt die Netzwerkkomponenten herunter.
<code>get_local_ip</code> (<code>self</code>)
Ermittelt die lokale IP-Adresse des Hosts.
<code>switch_active_group</code> (<code>self</code> , <code>group_name</code>)
Wechselt die aktive Gruppe.
<code>list_groups</code> (<code>self</code>)
Listet alle Gruppen auf, denen der Benutzer beigetreten ist.
<code>send_image</code> (<code>self</code> , <code>handle</code> , <code>size_str</code>)
Sendet einen Block zufälliger Binärdaten an einen Benutzer.

Public Attributes

<code>config</code> = <code>config</code>
bool <code>running</code> = <code>True</code>
dict <code>image_transfer_info</code> = <code>{}</code>
Speichert Metadaten für erwartete Bildübertragungen.
<code>handle</code> = <code>self.config['user']['handle']</code>
<code>port</code> = <code>self.config['user']['port']</code>

```

broadcast_port = self.config['user']['whoisport']
str broadcast_address = '255.255.255.255'
list groups = ['default']
str active_group = 'default'
dict users_by_group = {'default': {}}
unicast_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
broadcast_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
unicast_listener = Thread(target=self._listen_unicast)
broadcast_listener = Thread(target=self._listen_broadcast)
periodic_thread = Thread(target=self._periodic_tasks)

```

Protected Member Functions

_listen_unicast (self)

Wartet auf eingehende direkte Nachrichten (Unicast) und Bilddaten.

_listen_broadcast (self)

Horcht im Netzwerk nach öffentlichen Broadcast-Nachrichten.

_handle_unicast_message (self, message, addr)

Verarbeitet eine eingehende Unicast-Nachricht.

_handle_broadcast_message (self, message, addr)

Verarbeitet eine eingehende Broadcast-Nachricht.

_send_leave_broadcast (self, group_name)

_periodic_tasks (self)

Führt periodische Aufgaben aus.

_save_image (self, data, sender)

Speichert empfangene Bilddaten in einer Datei.

Detailed Description

Kümmert sich um die gesamte Netzwerkkommunikation.

Diese Klasse ist das Herzstück der P2P-Logik. Sie verwendet Sockets für direkte Nachrichten (Unicast) und öffentliche Nachrichten (Broadcast), um den Chat, die Gruppenverwaltung und die Anwesenheitserkennung zu realisieren.

Constructor & Destructor Documentation

◆ **__init__**()

```
network.network_handler.NetworkHandler.__init__( self,
                                                config )
```

Initialisiert den **NetworkHandler**.

Parameters

config Die geladene Konfiguration der Anwendung.

Member Function Documentation

◆ _handle_broadcast_message()

```
network.network_handler.NetworkHandler._handle_broadcast_message ( self,
                                                                    message,
                                                                    addr )
```

protected

Verarbeitet eine eingehende Broadcast-Nachricht.

Parameters

message Die empfangene Nachricht als String.
addr Die Adresse (IP, Port) des Absenders.

Protokoll-Nachrichten:

- ALIVE <Gruppe> <Handle> <Port>: Periodisches Lebenszeichen.
- JOIN <Gruppe> <Handle> <Port>: Beitritt zu einer Gruppe.
- LEAVE <Gruppe> <Handle>: Verlassen einer Gruppe.
- MSG <Gruppe> <Handle> <Text>: Nachricht an eine Gruppe.

◆ _handle_unicast_message()

```
network.network_handler.NetworkHandler._handle_unicast_message ( self,
                                                                message,
                                                                addr )
```

protected

Verarbeitet eine eingehende Unicast-Nachricht.

Parameters

message Die empfangene Nachricht als String.

addr Die Adresse (IP, Port) des Absenders.

Protokoll-Nachrichten:

- MSG <Absender> <Text>: Eine private Textnachricht.
- MSG-AUTOREPLY <Absender> <Text>: Eine automatische Antwort.
- IMG <Absender> <Größe>: Kündigt eine Bildübertragung an.
- REPLY <Gruppe> <Handle> <Port>: Antwort auf eine JOIN-Anfrage.

◆ _listen_broadcast()

```
network.network_handler.NetworkHandler._listen_broadcast ( self )
```

protected

Horcht im Netzwerk nach öffentlichen Broadcast-Nachrichten.

Hier werden Dinge wie Lebenszeichen (ALIVE) oder öffentliche Gruppennachrichten (GMSG) empfangen.

◆ _listen_unicast()

```
network.network_handler.NetworkHandler._listen_unicast ( self )
```

protected

Wartet auf eingehende direkte Nachrichten (Unicast) und Bilddaten.

◆ _periodic_tasks()

```
network.network_handler.NetworkHandler._periodic_tasks ( self )
```

protected

Führt periodische Aufgaben aus.

Sendet alle 15 Sekunden ALIVE-Nachrichten und entfernt Benutzer, von denen seit über 35 Sekunden nichts mehr gehört wurde.

◆ _save_image()


```
network.network_handler.NetworkHandler._save_image ( self,
                                                    data,
                                                    sender )
```

protected

Speichert empfangene Bilddaten in einer Datei.

Parameters

data Die empfangenen Binärdaten.

sender Der Handle des Absenders.

◆ _send_leave_broadcast()

```
network.network_handler.NetworkHandler._send_leave_broadcast ( self,
                                                                group_name )
```

protected

◆ announce_presence()

```
network.network_handler.NetworkHandler.announce_presence ( self,
                                                            group_name = None )
```

Sendet eine JOIN-Nachricht, um die eigene Anwesenheit bekannt zu machen.

Parameters

group_name Die Gruppe, in der die Anwesenheit bekannt gemacht wird. Wenn None, wird sie in allen beigetretenen Gruppen bekannt gemacht.

◆ discover_users()

```
network.network_handler.NetworkHandler.discover_users ( self,
                                                         group_name = None )
```

Zeigt die bekannten Benutzer in einer bestimmten Gruppe an.

Parameters

group_name Der Name der Gruppe. Wenn None, wird die aktive Gruppe verwendet.

◆ get_local_ip()

```
network.network_handler.NetworkHandler.get_local_ip ( self )
```

Ermittelt die lokale IP-Adresse des Hosts.

Returns

Die lokale IP-Adresse als String.

◆ join_group()

```
network.network_handler.NetworkHandler.join_group ( self,  
                                                    group_name )
```

Tritt einer Gruppe bei und macht sie zur aktiven Gruppe.

Parameters

group_name Der Name der beizutretenden Gruppe.

◆ leave_group()

```
network.network_handler.NetworkHandler.leave_group ( self,  
                                                    group_name )
```

Verlässt eine bestimmte Gruppe.

Parameters

group_name Der Name der zu verlassenden Gruppe.

◆ list_groups()

```
network.network_handler.NetworkHandler.list_groups ( self )
```

Listet alle Gruppen auf, denen der Benutzer beigetreten ist.

◆ send_group_message()

```
network.network_handler.NetworkHandler.send_group_message ( self,  
                                                         text )
```

Sendet eine Nachricht an die aktuell aktive Gruppe.

Parameters

text Der Nachrichtentext.

◆ send_image()

```
network.network_handler.NetworkHandler.send_image ( self,  
                                                    handle,  
                                                    size_str )
```

Sendet einen Block zufälliger Binärdaten an einen Benutzer.

Parameters

handle Der Handle des Empfängers.

size_str Die Größe der zu sendenden Daten als String.

◆ send_message()

```
network.network_handler.NetworkHandler.send_message ( self,  
                                                      handle,  
                                                      text )
```

Sendet eine private Nachricht an einen bestimmten Benutzer.

Parameters

handle Der Handle des Empfängers.

text Der Nachrichtentext.

◆ shutdown()

```
network.network_handler.NetworkHandler.shutdown ( self )
```

Führt die Netzwerkkomponenten herunter.

Sendet eine LEAVE-Nachricht an alle Gruppen und schließt die Sockets.

◆ switch_active_group()

```
network.network_handler.NetworkHandler.switch_active_group ( self,
                                                         group_name )
```

Wechselt die aktive Gruppe.

Parameters

group_name Die Gruppe, die aktiv werden soll.

Member Data Documentation

◆ active_group

```
str network.network_handler.NetworkHandler.active_group = 'default'
```

◆ broadcast_address

```
network.network_handler.NetworkHandler.broadcast_address = '255.255.255.255'
```

◆ broadcast_listener

```
network.network_handler.NetworkHandler.broadcast_listener =
Thread(target=self._listen_broadcast)
```

◆ broadcast_port

```
network.network_handler.NetworkHandler.broadcast_port = self.config['user']['whoisport']
```

◆ broadcast_socket

```
network.network_handler.NetworkHandler.broadcast_socket = socket.socket(socket.AF_INET,
socket.SOCK_DGRAM)
```

◆ config

```
network.network_handler.NetworkHandler.config = config
```

◆ groups

```
list network.network_handler.NetworkHandler.groups = ['default']
```

◆ handle

```
network.network_handler.NetworkHandler.handle = self.config['user']['handle']
```

◆ image_transfer_info

```
dict network.network_handler.NetworkHandler.image_transfer_info = {}
```

Speichert Metadaten für erwartete Bildübertragungen.

Format: {(ip, port): (groesse, handle)}

◆ periodic_thread

```
network.network_handler.NetworkHandler.periodic_thread = Thread(target=self._periodic_tasks)
```

◆ port

```
network.network_handler.NetworkHandler.port = self.config['user']['port']
```

◆ running

```
bool network.network_handler.NetworkHandler.running = True
```

◆ unicast_listener

```
network.network_handler.NetworkHandler.unicast_listener = Thread(target=self._listen_unicast)
```

◆ unicast_socket

```
network.network_handler.NetworkHandler.unicast_socket = socket.socket(socket.AF_INET,  
socket.SOCK_DGRAM)
```

◆ users_by_group

```
dict network.network_handler.NetworkHandler.users_by_group = {'default': {}}
```

The documentation for this class was generated from the following file:

- network/[network_handler.py](#)

user_interface.UserInterface Class Reference

Die Kommandozentrale für den Benutzer. [More...](#)

Public Member Functions

<code>__init__ (self, config)</code>
Initialisiert die Benutzeroberfläche.

Public Attributes

<code>config = config</code>
bool <code>running = True</code>
<code>network = NetworkHandler(self.config)</code>
<code>input_thread = threading.Thread(target=self._input_loop)</code>

Protected Member Functions

<code>_start_input_thread (self)</code>
Startet den Thread für die Benutzereingabe.
<code>_input_loop (self)</code>
Die Hauptschleife, die auf die Eingaben des Benutzers wartet und sie verarbeitet.
<code>_print_help (self)</code>
Zeigt eine Übersicht aller verfügbaren Befehle an.
<code>_shutdown (self)</code>
Führt die Anwendung sauber herunter.

Detailed Description

Die Kommandozentrale für den Benutzer.

Diese Klasse startet einen eigenen Thread, der ständig auf die Eingaben des Benutzers wartet. So wird die Netzwerk-Kommunikation nicht blockiert. Alle Befehle wie 'msg' oder '/join' werden hier erkannt und an den **NetworkHandler** weitergegeben.

Constructor & Destructor Documentation

◆ <code>__init__()</code>

```
user_interface.UserInterface.__init__( self,  
                                       config )
```

Initialisiert die Benutzeroberfläche.

Parameters

config Die geladene Konfiguration der Anwendung.

Member Function Documentation

◆ _input_loop()

```
user_interface.UserInterface._input_loop ( self )
```

protected

Die Hauptschleife, die auf die Eingaben des Benutzers wartet und sie verarbeitet.

Warning

Das Parsen von Benutzernamen mit Leerzeichen in 'msg' ist etwas heikel, besonders wenn ein Benutzername der Anfang eines anderen ist.

◆ _print_help()

```
user_interface.UserInterface._print_help ( self )
```

protected

Zeigt eine Übersicht aller verfügbaren Befehle an.

◆ _shutdown()

```
user_interface.UserInterface._shutdown ( self )
```

protected

Führt die Anwendung sauber herunter.

◆ _start_input_thread()

```
user_interface.UserInterface._start_input_thread ( self )
```

protected

Startet den Thread für die Benutzereingabe.

Member Data Documentation

◆ config

```
user_interface.UserInterface.config = config
```

◆ input_thread

```
user_interface.UserInterface.input_thread = threading.Thread(target=self._input_loop)
```

◆ network

```
user_interface.UserInterface.network = NetworkHandler(self.config)
```

◆ running

```
bool user_interface.UserInterface.running = True
```

The documentation for this class was generated from the following file:

- **user_interface.py**

utils.config_loader Namespace Reference

Functions

load_config (config_filename="config.toml")

Liest eine TOML-Datei ein und gibt die Konfiguration als Dictionary zurück.

Function Documentation

◆ load_config()

```
utils.config_loader.load_config ( config_filename = "config.toml" )
```

Liest eine TOML-Datei ein und gibt die Konfiguration als Dictionary zurück.



Parameters

config_filename Der Name der Konfigurationsdatei.

Returns

Ein Dictionary mit der Konfiguration, oder None, wenn etwas schiefgeht.

File List

✓  network	
^L discovery.py	Implementiert den Discovery-Dienst für die P2P-Chat-Anwendung
^L network_handler.py	Implementiert die Kernlogik der Netzwerkkommunikation
✓  utils	
^L config_loader.py	Hilfsmodul zum Laden der Konfiguration
^L protocol.py	Hilfsfunktionen für das SLCP-Protokoll
^L main.py	Haupt-Einstiegspunkt der P2P-Chat-Anwendung
^L user_interface.py	Benutzeroberfläche für die P2P-Chat-Anwendung