

Gruppe 4 –

Automatisiertes Multi-Server-Chat-System

Yanan Wang
03773246

Sizhe Cen
03775608

Rui Ying
03754558

Jingsong Lin
03774416

Wenhao Cheng
03776055

Zewen Yang
03779726

Keywords—Translate und Reminder

I. EINLEITUNG

In diesem Projekt wurde ein robustes, verteiltes Multi-Server-Chat-System entwickelt, das zur Nachrichtenserialisierung auf Protocol Buffers (Protobuf) basiert. Der Fokus lag auf der Umsetzung einer zuverlässigen Kommunikationslösung über TCP und UDP mit Funktionen für Nachrichtenbestätigung (ACK), kontinuierliche Verbindungsüberprüfung (Heartbeat), eine Erinnerungsfunktion sowie automatische Übersetzung von Nachrichten. Dabei wurde besonderer Wert auf Plattformunabhängigkeit, Leistungsfähigkeit unter hoher Last und eine skalierbare Architektur gelegt. Diese Einführung skizziert das Projektziel und den Kontext, während die folgenden Kapitel das System, seine Implementierung und Testergebnisse im Detail beschreiben.

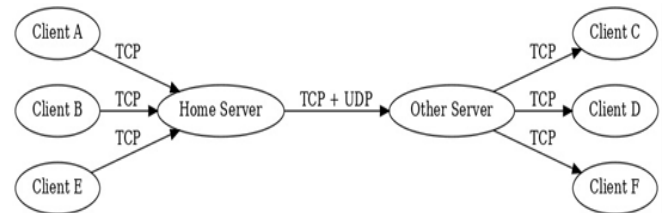
II. HAUPTTEIL

A. Beschreibung/Szenario

Bei der entwickelten Anwendung handelt es sich um ein verteiltes Chat-System, in dem mehrere Server simultan zusammenarbeiten. Jeder Client verbindet sich mit einem bestimmten Home-Server. Dieser Home-Server übernimmt die Weiterleitung der vom Client gesendeten Nachrichten an andere Server, falls die Zielnutzer sich auf anderen Serverinstanzen befinden. Auf diese Weise können Nutzer nahtlos miteinander chatten, auch wenn sie mit unterschiedlichen Servern verbunden sind. Die Server synchronisieren dafür untereinander wichtige Informationen wie Empfangsbestätigungen (ACKs) für Nachrichten und Nutzerstatus (online/offline), um einen konsistenten Systemzustand zu gewährleisten. Das Szenario zielt darauf ab, eine verteilte Chat-Infrastruktur zu schaffen, die lastverteiltes Messaging, automatische Übersetzung von Nachrichten in verschiedene Sprachen und Erinnerungen an Ereignisse bietet. Die Anwender profitieren von einer dynamischen Servererkennung (Clients finden verfügbare Server automatisch), hoher Ausfallsicherheit durch Mehrserver-Betrieb und nützlichen Zusatzfunktionen, die den Chat-Alltag erleichtern..

B. Implementierung

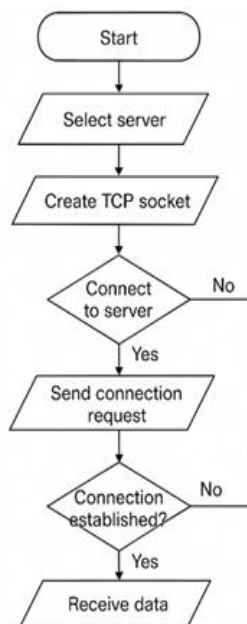
Programmstruktur



Die Programmstruktur des Chat-Systems ist modular und auf Verteilbarkeit sowie Fehlertoleranz ausgelegt. Die technische Basis bildet eine plattformunabhängige Implementierung in Python, welche umfangreiche Bibliotheken für Netzwerkkommunikation und Protokollfunktionen nutzt (z.B. socket, threading, protobuf). Das System verwendet ein multithreadiges Architekturmodell: Separate Threads kümmern sich um verschiedene Aufgaben wie Client-Verbindungen (TCP), Server-zu-Server-Kommunikation, Heartbeat-Überwachung und UDP-Broadcast. Durch diese Aufteilung werden Blockierungen vermieden und die Anwendung bleibt selbst unter Last responsiv.

Protobuf dient als leichtgewichtiges Nachrichtenformat: Alle Nachrichtentypen sind kompakt definiert, was eine schnelle Serialisierung und klare Typisierung erlaubt. Die Kombination aus TCP (für zuverlässige Punkt-zu-Punkt-Verbindungen) und UDP (für Broadcast-Discovery) bildet das Rückgrat der Kommunikation. Auf diesem Fundament wurden mehrere funktionale Module implementiert:

TCP-Kommunikation und Paketbehandlung: Um eine zuverlässige Nachrichtenübertragung zu gewährleisten, wurde ein eigener Paketrahmen definiert. Jeder Nachricht wird ein Header mit Typ und Länge des Payloads vorangestellt und ein spezielles Endzeichen angehängt. Am Empfänger werden zusammenhängende oder fragmentierte TCP-Datenströme in einem Puffer gesammelt und anhand des Headers sowie des Terminator-Zeichens wieder in einzelne vollständige Nachrichten getrennt. Dadurch werden sämtliche Nachrichten trotz TCP-„Sticky Packets“ oder Fragmentierung korrekt erkannt und fehlerfrei verarbeitet.

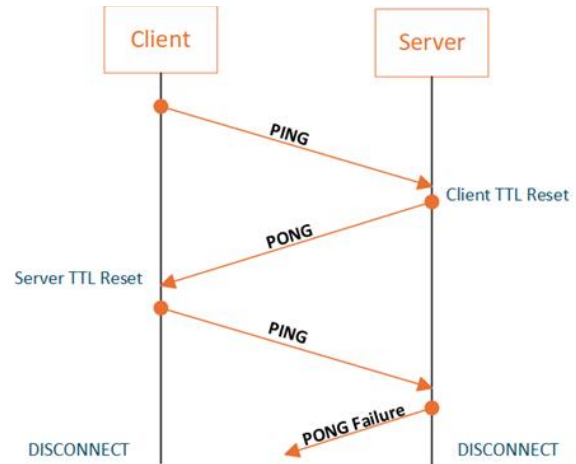


UDP-Servererkennung: Jeder Client kann verfügbare Server im lokalen Netzwerk automatisch entdecken. Dazu sendet der Client beim Start einen UDP-Broadcast an einen bekannten Port; alle Server, die an diesem Port lauschen, antworten mit Informationen wie ihrem Namen, der Adresse/Port für TCP-Verbindungen und den angebotenen Funktionen (z.B. ob Übersetzung oder Reminder unterstützt werden). Der Client erhält so eine Liste von Servern und kann einen Server auswählen, ohne manuell IP-Adressen konfigurieren zu müssen. Dieses Modul erhöht die Benutzerfreundlichkeit und ermöglicht eine dynamische Konfiguration des Systems in unbekannten Netzwerken. (Figure I, II)

Nachrichten-ACK-Mechanismus: Um die Zustellung jeder Nachricht über mehrere Server hinweg sicherzustellen, wurde ein expliziter Bestätigungsmechanismus implementiert. Jede versendete Chat-Nachricht erhält eine eindeutige Nachrichten-ID (z.B. ein Snowflake-Identifizier). Empfangende Server und Clients schicken für jede erhaltene Nachricht eine ACK-Bestätigung zurück. Der ursprüngliche Server markiert die Nachricht erst dann als zugestellt, wenn das ACK vom Ziel eingetroffen ist, und leitet ein DELIVERED-ACK an den Absender-Client weiter. In Gruppenchats wird ein optimiertes Verfahren genutzt, um nicht von jeder einzelnen Empfangsbestätigung abhängig zu sein und die Performance hoch zu halten. Dieser Ansatz erlaubt es dem Absender, den Status seiner Nachricht (zugestellt/nicht zugestellt) jederzeit präzise nachzuvollziehen und stellt sicher, dass keine Nachricht unbemerkt verloren geht oder doppelt verarbeitet wird.

Heartbeat-Überwachung: Ein kontinuierlicher Heartbeat-Mechanismus überwacht die Verbindungstreue aller Teilnehmer. Zwei parallel laufende Threads – einer für Client-Verbindungen und einer für Server-zu-Server-Verbindungen – senden in regelmäßigen Abständen Ping-Nachrichten an alle bekannten Partner. Jeder aktive Client oder Server antwortet mit einem entsprechenden Pong. Bleibt eine Antwort aus (z.B. weil die Verbindung unterbrochen wurde oder der Partner abgestürzt ist), wird diese Verbindung nach einem Timeout automatisch als inaktiv betrachtet, geschlossen und aus den internen Verbindungslisten entfernt. Dieser Mechanismus verhindert „Zombie-Clients“ bzw. Geistereinträge (Nutzer,

die als online angezeigt werden, aber nicht mehr verbunden sind) und trägt wesentlich zur Stabilität des Systems bei.



Asynchrone Anfrageverarbeitung: In einem Mehrserver-System treten Fälle auf, in denen Anfragen zeitverzögert oder in Teilstücken beantwortet werden (etwa bei serverübergreifenden Suchanfragen oder Mehrfach-Abfragen). Dafür implementiert das System eindeutige Anfrage-Handles, die typischerweise auf Zeitstempeln basieren. Wenn ein Client eine Anfrage stellt (z.B. Suche nach einem Benutzer oder eine Übersetzungsanfrage, die mehrere Server betrifft), erhält diese Nachricht einen solchen eindeutigen Handle. Bei Weiterleitungen zwischen Servern bleibt der Handle unverändert an der Nachricht hängen. Jede Antwort auf die Anfrage – egal von welchem Server sie kommt – enthält denselben Handle. Der Client verwaltet ausstehende Anfragen in Wartelisten (z.B. `pending_search` für Suchanfragen, `pending_acks` für unbestätigte Nachrichten) und kann so eingehende Antworten dem richtigen ursprünglichen Request zuordnen. Eingehende Antworten ohne bekannten Handle werden ignoriert oder protokolliert. Dieser Mechanismus erlaubt parallele, nicht-blockierende Abläufe und stellt sicher, dass auch bei komplexen Abläufen mit Verzögerungen keine Antwort falsch zugeordnet wird.

Übersetzungsmodul: Das Chat-System bietet die Möglichkeit, Nachrichten automatisch zu übersetzen, um mehrsprachige Unterhaltungen zu erleichtern. Die Übersetzungslogik ist direkt in die Nachrichtenverarbeitung integriert. Wenn ein Nutzer die Übersetzungsfunktion aktiviert, sendet sein Client eine spezielle Chat-Nachricht mit dem Originaltext und der gewünschten Zielsprache. Der Server erkennt zunächst die Sprache des Originaltexts (mittels `langdetect`-Bibliothek) und vergleicht sie mit der Zielsprache. Ist eine Übersetzung nötig, wird der Inhalt im Hintergrund automatisch mit Googletrans in die Zielsprache übersetzt. Die Antwort an den Nutzer enthält sowohl den ursprünglichen Originaltext als auch den übersetzten Text, beide sauber getrennt innerhalb derselben Nachrichtenstruktur. Durch diese Erweiterung entstehen kaum zusätzliche Verzögerungen, und die Kommunikation wird für Nutzer verschiedener Sprachen wesentlich erleichtert. (Figure III, IV)

Erinnerungsmodul: Nutzer können Erinnerungen setzen, die sie nach einer festgelegten Zeitspanne an ein Ereignis oder eine Notiz erinnern. Die Implementierung dieses Moduls erfolgt über einen eigenständigen Reminder-Thread, der zeitgesteuerte Nachrichten verwaltet. Jedes gesetzte Ereignis

(mitgegeben als Text und Countdown in Sekunden) wird in einer prioritätsbasierten Warteschlange (Min-Heap) gespeichert. Der Reminder-Thread schläft jeweils bis zum frühesten Fälligkeitszeitpunkt und schickt dann automatisch eine REMINDER-Nachricht mit dem betreffenden Ereignistext an den entsprechenden Nutzer. Dieses Verfahren – implementiert mit Python-Standardbibliotheken (heapq, threading) und durch Locks abgesichert – ist sehr ressourcenschonend, da es im Ruhezustand keine CPU-Zeit benötigt. Auch bei vielen aktiven Timern werden Erinnerungen präzise und termingerecht ausgeführt, was die Anwendung um ein nützliches Benachrichtigungstool erweitert. (Figure V)

GUI

Die grafische Benutzeroberfläche des Chat-Clients wurde mit PySide6(Qt für Python) realisiert.

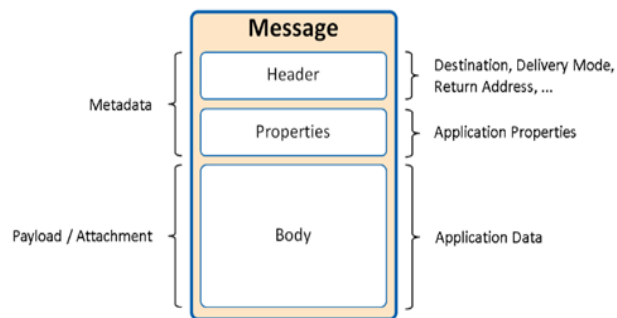
Die ursprüngliche Demo-Version des Clients verfolgte primär funktionale Ziele und verfügte über eine einfache Struktur. Mit dem Fortschreiten der Entwicklung wurde die Benutzeroberfläche visuell und interaktiv verbessert – insbesondere durch Übergangsanimationen, konsistente Farbschemata, Schatten und angepasste Schriftgrößen. Grundlage hierfür bildeten die Apple Human Interface Guidelines, die Klarheit, Konsistenz und Nutzerfeedback betonen. (Figure VI, VII)

Ein zentrales Problem war die Stilkohärenz: Änderungen einzelner UI-Elemente, wie z. B. Buttonfarben oder Scrollleisten, führten häufig zu visuellen Inkonsistenzen. Die Lösung bestand in einer systematischen Gestaltung des gesamten visuellen Systems, wobei alle Elemente auf einheitliche Nutzererfahrung und emotionale Wirkung ausgerichtet wurden.

Für die Server-UI wurde ein Ampelschema zur Visualisierung von Statusinformationen verwendet. Bei Buttons wie „Discover Servers“ zeigte sich, dass unzureichender Farbkontrast (z. B. blasses Gelb auf hellem Hintergrund) die Lesbarkeit stark beeinträchtigen kann. Daher erfolgten umfangreiche Farbtests zur Optimierung der Sichtbarkeit unter verschiedenen Bildschirmbedingungen.

Schriftgröße und -gewicht wurden gezielt angepasst, um die Lesbarkeit und Interaktionsgeschwindigkeit zu verbessern. Zudem wurden animierte Übergangseffekte eingeführt, um interaktive Elemente besser hervorzuheben. Die finale UI entstand durch iteratives Feintuning von CSS-Parametern, gestützt durch Designrichtlinien und praktische Tests. (Figure VIII)

Protokoll



(TABLE I)

III. SCHLUSS

Im Rahmen dieses Projekts wurde ein verteiltes Multi-Server-Chat-System realisiert, das auf TCP/UDP-Kommunikation sowie Protocol Buffers basiert. Ziel war die Entwicklung eines stabilen, skalierbaren Nachrichtensystems mit modernen Zusatzfunktionen wie automatischer Übersetzung, Erinnerungen, Zustellbestätigung (ACK) und Heartbeat-Überwachung.

Die Systemarchitektur ist modular aufgebaut und nutzt eine mehrthreadige Python-Implementierung. Durch eigene Nachrichtenrahmen wurde das bekannte TCP-Problem der „Sticky Packets“ gelöst, während ein UDP-basiertes Broadcast-Modul die automatische Servererkennung ermöglicht.

Ein zentraler Bestandteil ist der strukturierte ACK-Mechanismus zur Zustellverfolgung über mehrere Server hinweg. Parallel sorgt ein zweistufiges Heartbeat-System für die Erkennung inaktiver Verbindungen und bewahrt die Konsistenz des Online-Status.

Zusatzmodule wie die Übersetzungsfunktion (basierend auf langdetect und googletrans) und das Reminder-Modul (realisiert mit Min-Heap und Timer-Thread) erhöhen die Benutzerfreundlichkeit deutlich. Die Client-Oberfläche, entwickelt mit PySide6, trennt Logik und Darstellung strikt und erlaubt eine reaktive, erweiterbare Bedienung.

Insgesamt entstand ein robustes, fehlertolerantes System, das auch bei Netzwerkausfällen, Fragmentierung und Mehrserverbetrieb zuverlässig arbeitet und die Grundlage für weitere Entwicklungsschritte bietet.

Message Name	Direction	Protocol	Description	Notes
MESSAGE	C → S S → S S* → C	TCP	<pre> message ChatMessage { uint64 messageSnowflake = 1; User author = 2; oneof recipient { User user = 3; //etc.... }; oneof content { string textContent = 11; Document document = 12; LiveLocation live_location = 22; Translation translation = 44; // stickers , etc... } } </pre>	
LANGUAGE			<pre> enum Language { DE = 0; // German EN = 1; // English ZH = 2; // Chinese TR = 3; // Turkish } </pre>	
TRANSLATION	C → S S → S S* → C	TCP	<pre> message Translation { Language target_language = 1; // The original text to be translated (client → other server → Our server). string original_text = 2; // The translated text (Our server → other server → client). optional string translated_text = 3; } </pre>	For Client-Server-Server-Client Architecture
TRANSLATE	C → S	TCP	<pre> message Translate { Language target_language = 1; string original_text = 2; optional string translated_text = 3; } </pre>	For Client-Server-Client Architecture
TRANSLATED	S → C	TCP	<pre> message Translated { Language target_language = 1; string original_text = 2; optional string translated_text = 3; } </pre>	For Client-Server-Client Architecture
SET_REMINDER	C → S	TCP	<pre> message SetReminder { User user = 1; string event = 2; // ex. "Time to go to bed" uint32 countdownSeconds = 3; // ex. 60 for 1 minute } </pre>	TCP → User set a reminder for himself after countdownSeconds.
REMINDER	S → C	TCP	<pre> message Reminder { User user = 1; string reminderContent = 2; } </pre>	TCP → Server reminds user with reminderContent.

TABLE I: Protokoll

```

UDP/Client

def discover_servers(self):
    def send_discover():
        try:
            msg = Packing('DISCOVER_SERVER', b'')
            for port in self.udp_ports:
                udp = socket(AF_INET, SOCK_DGRAM)
                udp.setsockopt(SOL_SOCKET, SO_BROADCAST, 1)
                udp.settimeout(2)
                udp.sendto(msg, (self.BROADCAST_IP, port))
                udp.close()
            global_ms.log_signal.emit(f'[Server] DISCOVER_SERVER wurde an alle Ports {self.udp_ports} gesendet')
        except Exception as e:
            global_ms.log_signal.emit(f'[Server] Senden von DISCOVER_SERVER fehlgeschlagen: {e}')
    Thread(target=send_discover, daemon=True).start()

```

FIGURE I: Core code for UDP in Client

```

UDP/Server

def handle_udp_broadcast(self):
    while True:
        try:
            data, clientaddr = self.udp_socket.recvfrom(2048)
            purpose, length, payload = Unpacking(data)
            if purpose == 'DISCOVER_SERVER':
                print(f'[Debug] DISCOVER_SERVER erhalten von: {clientaddr}')
                tosend = self.Feature()
                # Antwort an den Client (temporärer Port)
                self.udp_socket.sendto(tosend, clientaddr)
                # An alle Server-Ports weiterleiten
                for port in self.udp_ports:
                    self.udp_socket.sendto(tosend, (self.BROADCAST_IP, port))
                global_ms.log_signal.emit(f"Antwort an {clientaddr} gesendet und SERVER_ANNOUNCE an alle Ports {self.udp_ports} gesendet")
            elif purpose == 'SERVER_ANNOUNCE':
                announce = Message_pb2.ServerAnnounce()
                announce.ParseFromString(payload)
                server_id = announce.serverId
                features = [(f.featureName, f.port) for f in announce.feature]
                if server_id == self.server_id:
                    # Ignoriere eigene Ankündigung
                    continue
                with self.server_list_lock:
                    self.server_list[server_id] = {
                        'ip': clientaddr[0],
                        'features': features,
                        'port': clientaddr[1],
                        'last_announce': time.time(),
                        'socket': None,
                    }
                global_ms.log_signal.emit(f"[Server] Neuer Server entdeckt: {server_id} @ {clientaddr[0]} features={features}")
                print(f"[Debug] Server {server_id} entdeckt, versuche Verbindung")
                Thread(target=self.connect_to_server, args=(server_id, clientaddr[0], features), daemon=True).start()
            except Exception as e:
                global_ms.log_signal.emit(f"[Server] Fehler in handle_udp_broadcast: {e}")
                continue

```

FIGURE II: Core code for UDP in Server

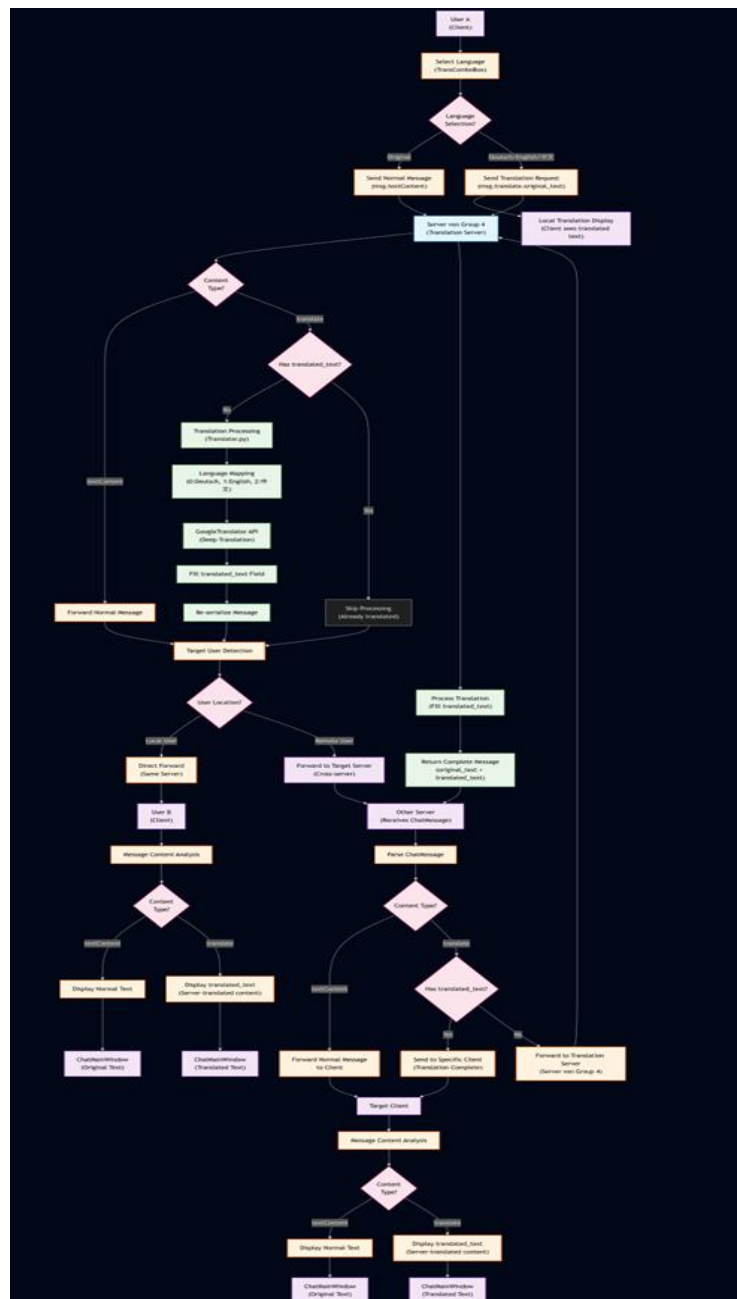


FIGURE III: Translate Verlauf

```

def translator(text: str, language: str):
    language_map = {
        'Deutsch': 'de',
        'English': 'en',
        '中文': 'zh-CN',
        'Original': None # Keine Übersetzung
    }
    try:
        target_lang = language_map.get(language)
        if target_lang is None:
            return text # Originaltext beibehalten
        detected_lang = detect(text)
        if detected_lang == target_lang:
            return text # Keine Übersetzung nötig
        return GoogleTranslator(source='auto', target=target_lang).translate(text)
    except:
        return text # Fehler -> Rückgabe Original
  
```

FIGURE IV: Core Code Translate

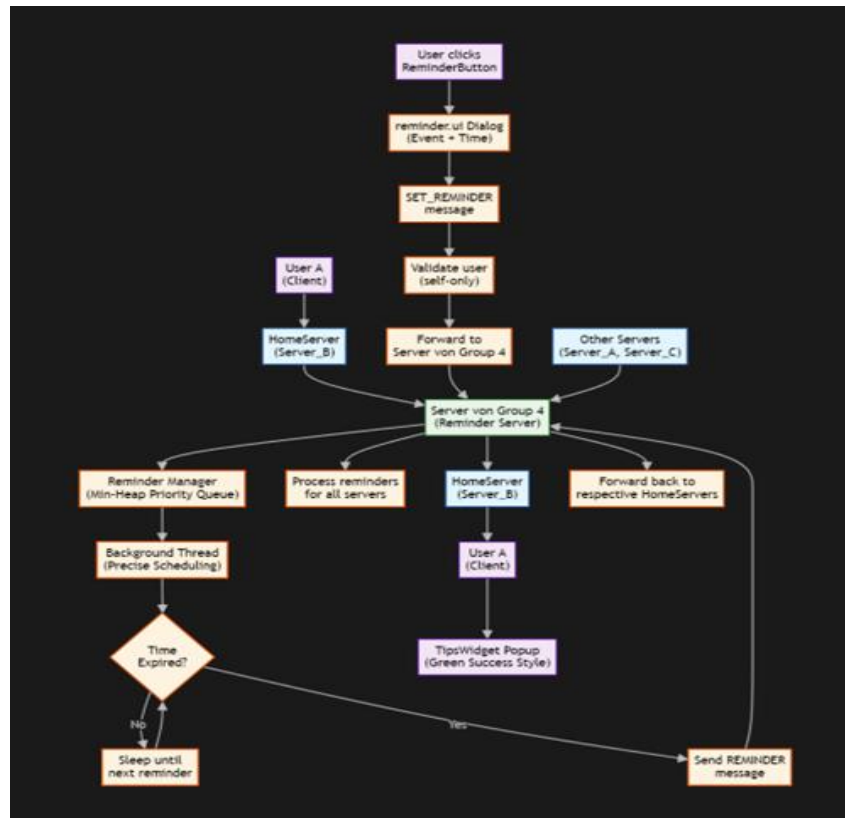


FIGURE V: Reminder Verlauf

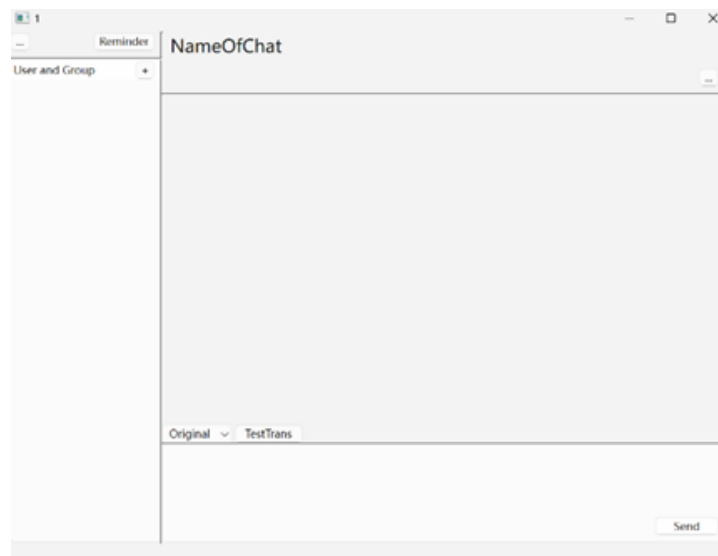


FIGURE VI: Alte Version

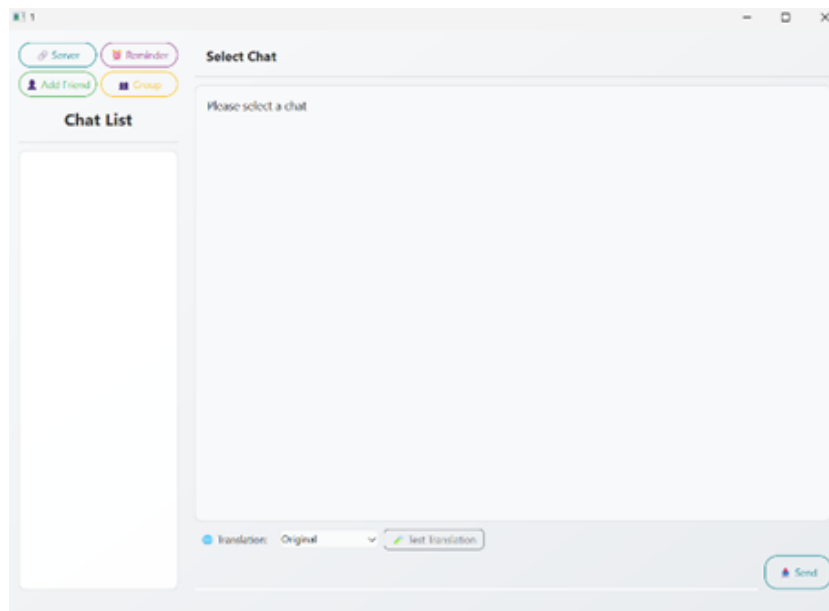


FIGURE VII: Neue Version

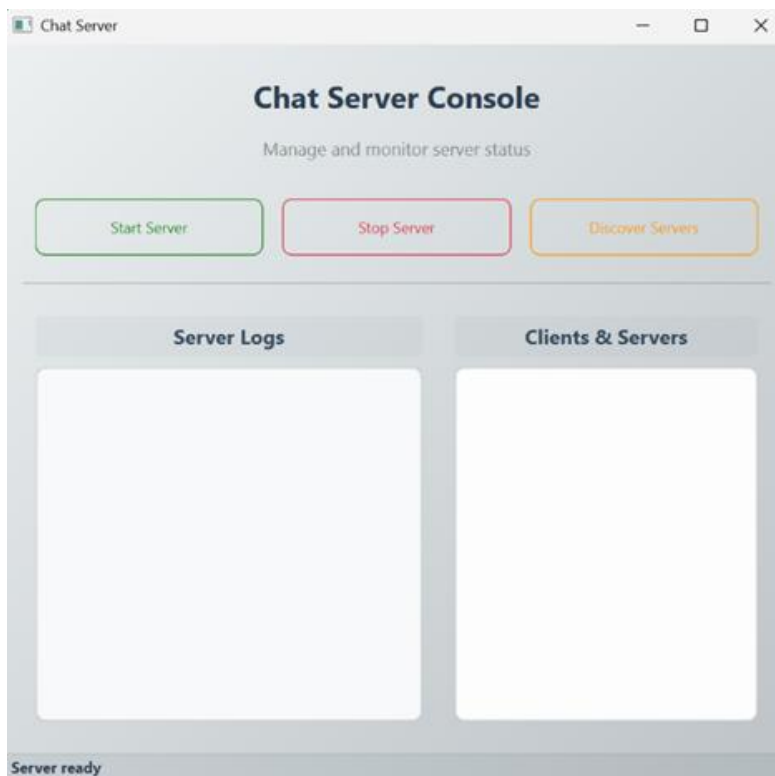


FIGURE VIII: Server GUI mit Effekt