Gruppe 4 –

Automatisiertes Multi-Server-Chat-System

Yanan Wang
03773246

Sizhe Cen
03775608

Rui Ying
03754558

Jingsong Lin
03774416

Wenhao Cheng
03776065

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.

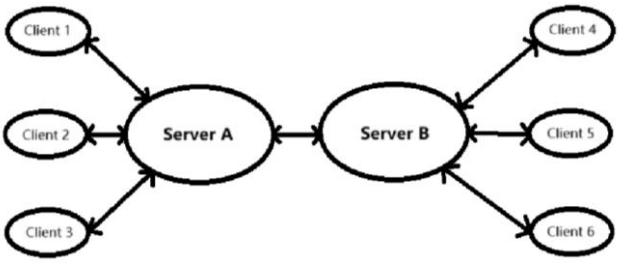
II. Hauptteil

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 basiert auf einem Multi-Threading-Architekturmodell: Für unterschiedliche Aufgaben wie die grafische Benutzeroberfläche (GUI), die Verwaltung von Client-Verbindungen

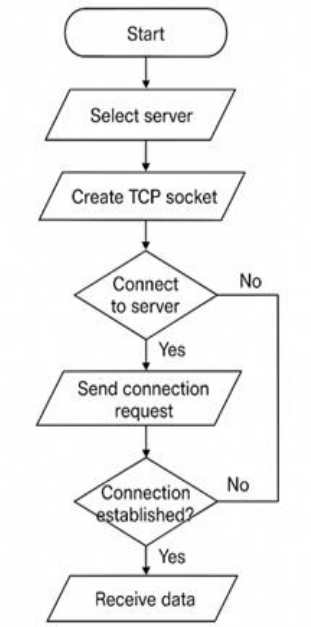
(def handle_tcp_client(self, client_socket, client_addr)), die Kommunikation zwischen Servern, das Heartbeat-System, die Reminder-Warteschlange (Min-Heap) sowie das kontinuierliche Listen auf UDP-Ports

(Thread(target=self.start_udp_listener,daemon=True).start()) werden jeweils eigene Threads eingesetzt.
Durch diese funktionale Trennung wird Blockierung vermieden, sodass die Anwendung auch unter Last reaktionsfähig bleibt.

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 (\n) angehängt.

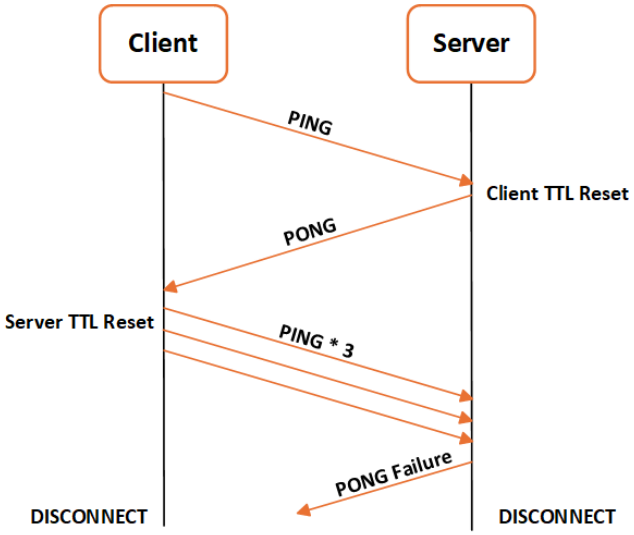
'[PURPOSE] [LENGTH] [PAYLOAD]\n' Am Empfänger werden zusammenhängende oder fragmentierte TCP-Datenströme in einem Puffer gesammelt und anhand des Headers sowie des Terminator-Zeichens wieder in einzelne vollständige Nachrichten getrennt. Dadurch werden sämtliche Nachrichten trotz TCP-„Sticky Packets“ oder Fragmentierung korrekt erkannt und fehlerfrei verarbeitet.(Bitte beziehen Sie sich auf die Python-Datei PackingandUnpacking.py)



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(255.255.255.255 : 9999); 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 (Snowflake-Identifier). 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 (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.

Es wurden zwei Ansätze für die Umsetzung untersucht:

Variante 1 – Einfache Liste Round Robin: Erinnerungen werden in einer gemeinsamen Liste gespeichert und regelmäßig (z. B. jede Sekunde) von einem separaten Thread überprüft. Ist der Zielzeitpunkt erreicht, wird eine REMINDER-Nachricht an den jeweiligen Nutzer gesendet.

Variante 2 – Prioritätsbasierte Ereigniswarteschlange: Alle Erinnerungen werden in einer Min-Heap-Datenstruktur (Priority Queue) gespeichert. Ein dedizierter „Reminder Worker“-Thread überwacht ausschließlich das am frühesten fällige Element:

Liegt der Zielzeitpunkt noch in der Zukunft, schläft der Thread bis zu diesem Zeitpunkt.

Ist die Zeit erreicht, wird der Reminder ausgeführt und aus dem Heap entfernt.
Dieser Ansatz ist äußerst effizient, da der Thread nicht unnötig CPU-Zeit verbraucht.

Aus den oben genannten Gründen haben wir uns Min-Heap-Datenstruktur entschieden. (Figure V,VI) (TABLE I)

GUI

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

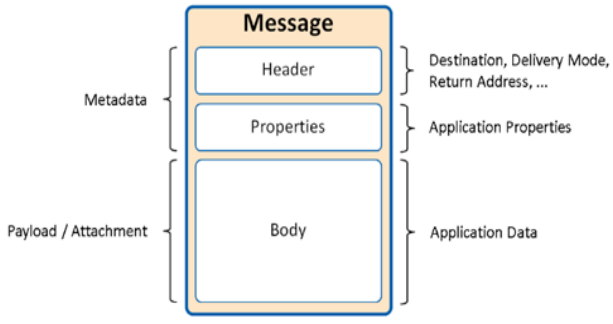
Die ursprüngliche Demo-Version des Clients verfolgte primär funktionale Ziele und verfügte über eine einfache Struktur. Mit dem Fortschreiten der Entwicklung wurde die Benutzeroberfläche visuell und interaktiv verbessert – insbesondere durch Übergangsanimationen, konsistente Farbschemata, Schatten und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

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 IX)

Protokoll



(TABLE II)

Die Entwicklung des Protokolls

Unsere Gruppe verwendet eine (client-server-server-client) Struktur, während Gruppe 2 eine (client-server-client) Struktur nutzt. Das von uns ursprünglich definierte Nachrichtenformat war für sie zu kompliziert, da es eine Weiterleitungslogik zwischen Servern beinhaltete.

Unsere ursprüngliche Logik bestand darin, translation in ChatMessage zu integrieren. Dadurch konnten wir das vorhandene Feld user author in ChatMessage nutzen, um die übersetzte Nachricht an author.serverId zurückzusenden. Der Server des author prüft dann mithilfe der hasField-Funktion, ob translated_text vorhanden ist – in diesem Fall wurde die Nachricht an den anfragenden user zurückgeleitet. Wenn nur original_text vorhanden ist, bedeutet das, dass die Nachricht noch nicht übersetzt wurde und an unseren Übersetzungsserver weitergeleitet werden muss.

Da Gruppe 2 nur mit einem client mit uns verbunden ist und sie diese Logik als zu kompliziert empfanden (was sie eigentlich nicht ist – für die (client-server-server-client) Struktur ist sie zwingend notwendig), habe ich ein alternatives Format vorgeschlagen: TRANSLATE und TRANSLATED. (TABLE III)

Unsere Prinzipien beinhalten, dem anfragenden user möglichst viele Informationen bereitzustellen, z. B. original_text, target_language, translated_text – so kann der user direkt alle Informationen aus der Nachricht extrahieren, anstatt lokal zu speichern. Dieses Format wurde jedoch von Gruppe 2 erneut abgelehnt.

Gruppe 2 verstand nicht, warum wir messageSnowflake verwenden, obwohl das Konzept dem von message und message_ack entspricht: Der client startet nach Anforderung der Übersetzung einen neuen Thread, der auf die Antwort wartet. Um zu prüfen, ob eine Übersetzung empfangen wurde, ist ein eindeutiger messageSnowflake erforderlich.

In realen Szenarien – etwa durch Netzwerkprobleme (wir testeten mit mobilem Hotspot statt mit Switch und Kabel) oder durch geringe Serverleistung (z. B. auf einem Laptop) – kann es vorkommen, dass übersetzte Nachrichten in falscher Reihenfolge zurückgesendet werden. Ohne messageSnowflake könnte die Zuordnung zur Originalnachricht verloren gehen. Das author-Feld hilft außerdem, dem richtigen user die Nachricht zuzustellen, ohne IP-Adressen aus dem Socket lesen zu müssen.

Zitat aus der Diskussion:
„Wenn innerhalb eines sehr kurzen Zeitraums zwei Nachrichten empfangen werden, kann es ohne Snowflake zu Verwirrung kommen."

Dies ist in einem Netzwerk mit vielen clients und Servern kein Edge Case, sondern ein realistisches Szenario.

Dennoch stimmten wir zu, das einfachere Format von Gruppe 2 zu übernehmen – obwohl es weniger stabil und robust ist.

Wir haben also ihr TRANSLATE/TRANSLATED-Format implementiert, behalten aber gleichzeitig unser ursprüngliches Format (Translation in ChatMessage) für client-server-server-client Fälle bei.

Ein Mitglied von Gruppe 2 erklärte, dass ihr client die übersetzte Nachricht nach Empfang selbst weiterleitet. Nach Absprache übernehmen wir daher keine Verantwortung für die Weiterleitung von TRANSLATE und TRANSLATED – wir bieten lediglich Echo + Translate Funktionalität an. (Figure X)

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.

Eigenschaft	**Liste Round Robin**	**Min-Heap**
Zeitkomplexität	O(n) pro Sekunde	O(log n) pro Einfügung
Speicherkomplexität	O(n)	O(n)
CPU-Auslastung	Hoch ständigeAbfrage	Niedrig bedarfsgesteuert
Speichernutzung	Mittel	Mittel
Empfohlenes Szenario	<100 Erinnerungen	>100 Erinnerungen

TABLE I: Algorithmusvergleich

Message Name	Direction	Protocol	Description	Notes

MESSAGE	C → S
	S → S
	S* → C
		TCP	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...
			}
			}	

LANGUAGE			enum Language {
			DE = 0; // German
			EN = 1; // English
			ZH = 2; // Chinese
			TR = 3; // Turkish
			}	

TRANSLATION	C → S
	S → S
	S* → C
		TCP	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;
			}
				For Client-Server-Server-Client Architecture

TRANSLATE	C → S	TCP	message Translate {
			Language target_language = 1;
			string original_text = 2;
			optional string translated_text = 3;
			}
				For Client-Server-Client Architecture

TRANSLATED	S → C	TCP	message Translated {
			Language target_language = 1;
			string original_text = 2;
			optional string translated_text = 3;
			}
				For Client-Server-Client Architecture

SET_REMINDER	C → S	TCP	message SetReminder {
			User user = 1;
			string event = 2; // ex. "Time to go to bed"
			uint32 countdownSeconds = 3; // ex. 60 for 1 minute
			}
				TCP → User set a reminder for himself after countdownSeconds.

REMINDER	S → C	TCP	message Reminder {
			User user = 1;
			string reminderContent = 2;
			}
				TCP → Server reminds user with reminderContent.

TABLE II: Protokoll

TRANSLATE	C → S	TCP	Message Translate{
			enum Language {
			DE = 0; // German
			EN = 1; // English
			ZH = 2; // Chinese}
			uint64 messageSnowflake = 1;
			User author = 2;
			Language target_language = 3;
			string original_text = 4;}

TRANSLATED	S → C	TCP	Message Translate{
			enum Language {
			DE = 0;
			EN = 1;
			ZH = 2; }
			uint64 messageSnowflake = 1;
			User author = 2;
			Language target_language = 3;
			string original_text = 4;}

TABLE III: Protokoll Version 1

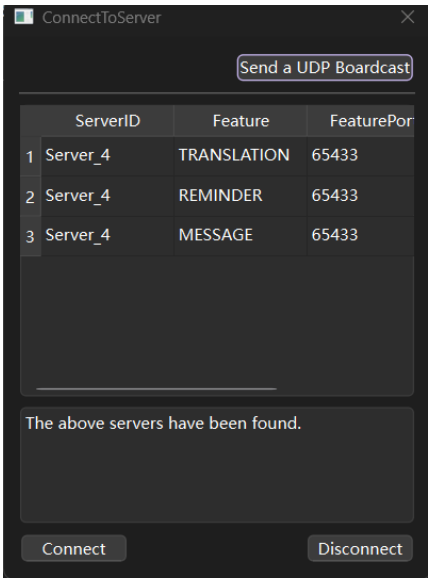


FIGURE I: Die vom Client empfangene Serverliste (ServerID, Feature, Port)

```python
def hanle_udp_boardcast(self):
    while True:
        try:
            # print("[Debug] UDP listening loop, waiting for data...")
            data, clientaddr = self.udp_socket.recvfrom(2048)
            purpose, length, payload = Unpacking(data)
            if purpose == 'DISCOVER_SERVER':
                # Received DISCOVER_SERVER: first reply to clientaddr, then
broadcast to all UDP ports
                print(f"[Debug] Received DISCOVER_SERVER: {clientaddr}")
                tosend = self.Feature()
                # Reply to the sender (temporary port of client or server)
                self.udp_socket.sendto(tosend, clientaddr)
                # Broadcast to all server UDP ports
                for port in self.udp_ports:
                    self.udp_socket.sendto(tosend, (self.BROADCAST_IP, port))
                global_ms.log_signal.emit(f"Replied to {clientaddr} and broadcasted
SERVER_ANNOUNCE to all ports {self.udp_ports}")
            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]
                # print(f"[Debug] Received SERVER_ANNOUNCE: server_id={server_id},
features={features}, from={clientaddr}")
                if server_id == self.server_id:
                    # Ignore self
                    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] Discovered new server:
{server_id} @ {clientaddr[0]} features={features}")

                # Proactively connect to the discovered server
                print(f"[Debug] Discovered server {server_id}, initiating
connection")
                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] hanle_udp_boardcast error: {e}")
            continue
```

FIGURE II: Core Code for UDP in Server

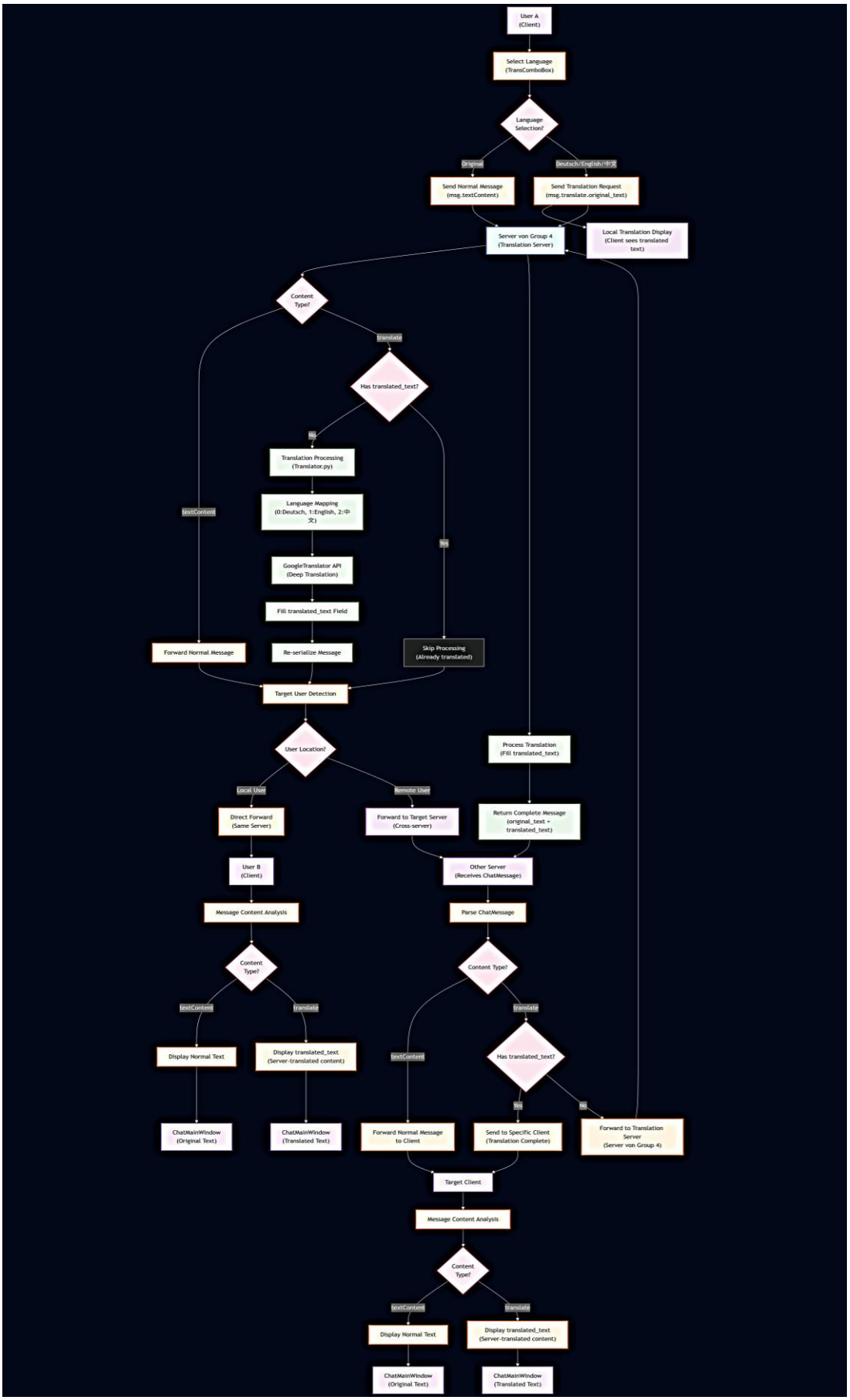


FIGURE III: Translate Verlauf

```python
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

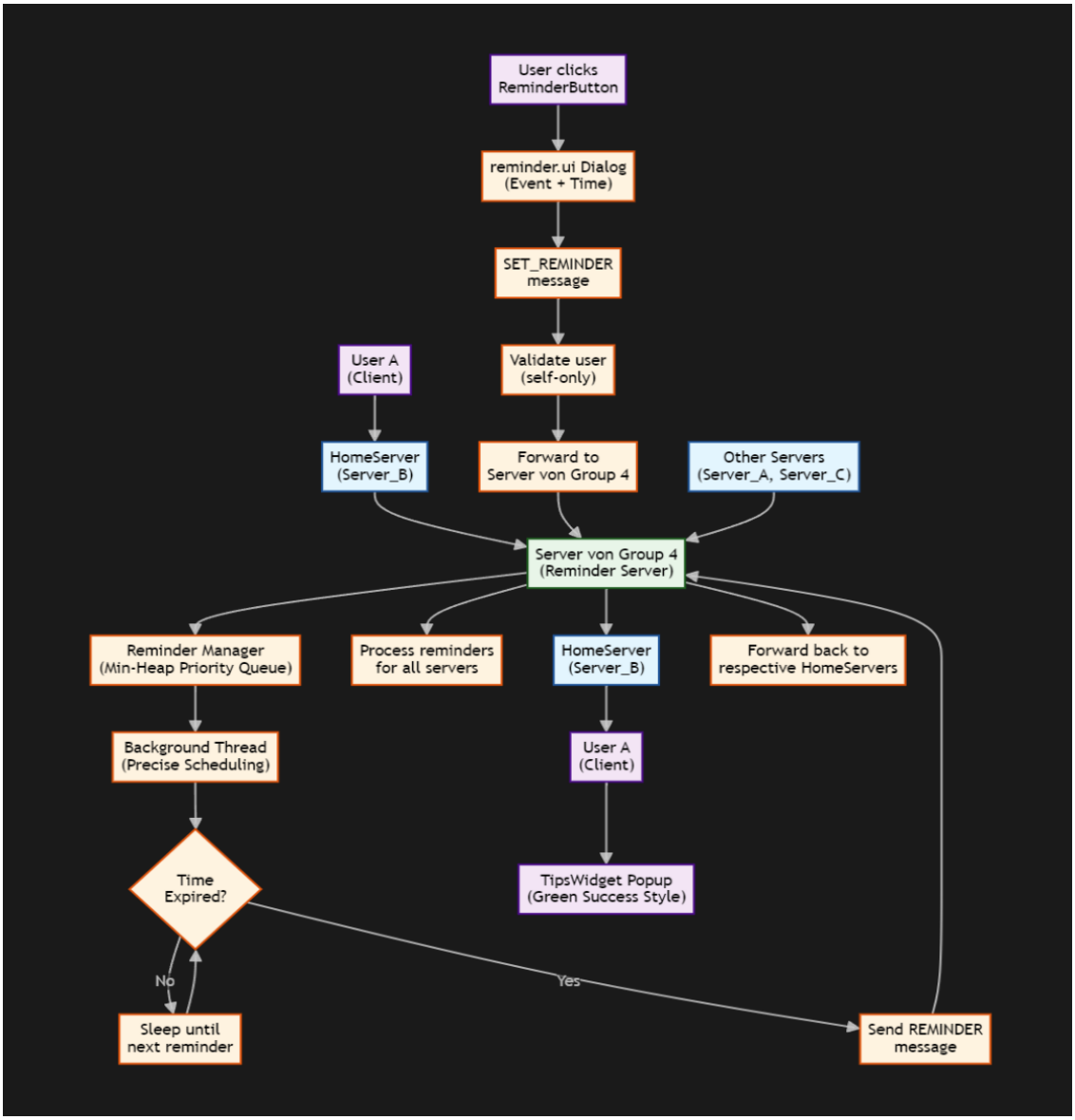
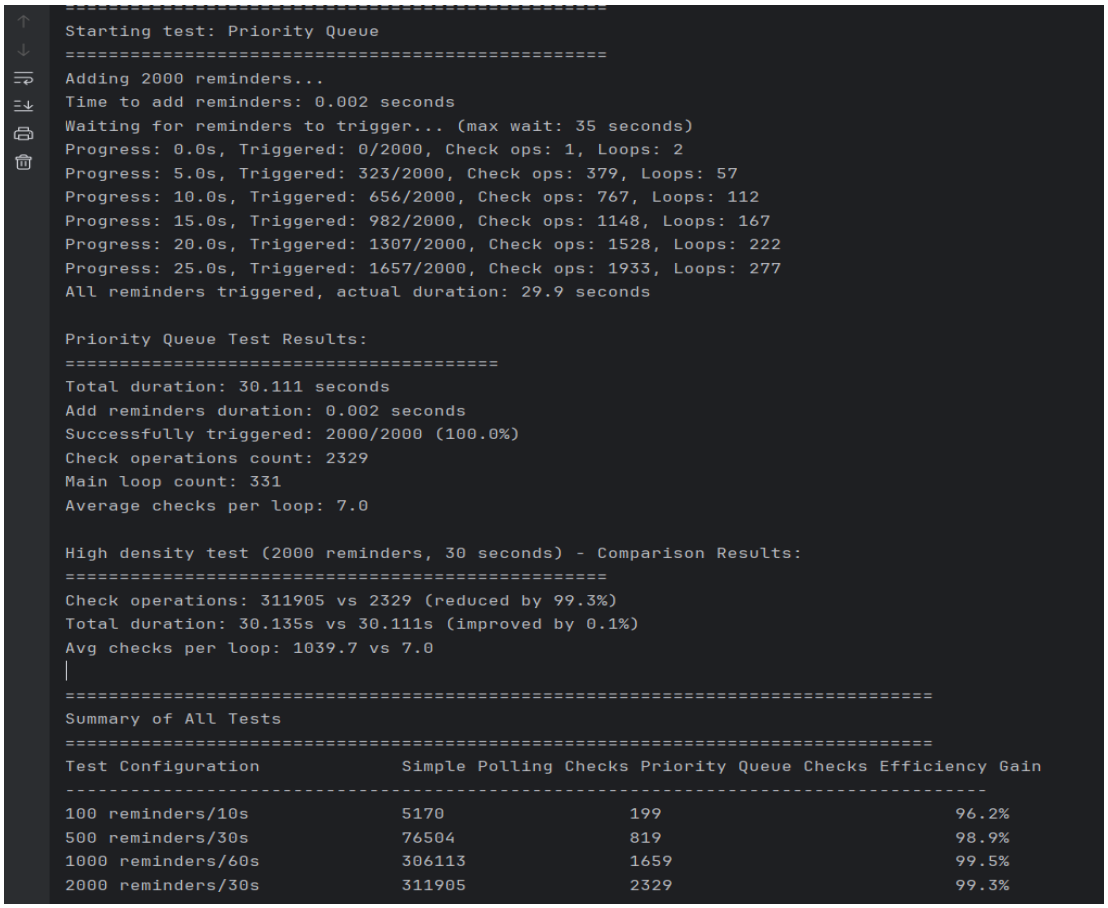


FIGURE V: Reminder Verlauf



```
================================================
Starting test: Priority Queue
================================================
Adding 2000 reminders...
Time to add reminders: 0.002 seconds
Waiting for reminders to trigger... (max wait: 35 seconds)
Progress: 0.0s, Triggered: 0/2000, Check ops: 1, Loops: 2
Progress: 5.0s, Triggered: 323/2000, Check ops: 379, Loops: 57
Progress: 10.0s, Triggered: 656/2000, Check ops: 767, Loops: 112
Progress: 15.0s, Triggered: 982/2000, Check ops: 1148, Loops: 167
Progress: 20.0s, Triggered: 1307/2000, Check ops: 1528, Loops: 222
Progress: 25.0s, Triggered: 1657/2000, Check ops: 1933, Loops: 277
All reminders triggered, actual duration: 29.9 seconds

Priority Queue Test Results:
========================================
Total duration: 30.111 seconds
Add reminders duration: 0.002 seconds
Successfully triggered: 2000/2000 (100.0%)
Check operations count: 2329
Main loop count: 331
Average checks per loop: 7.0

High density test (2000 reminders, 30 seconds) - Comparison Results:
================================================
Check operations: 311905 vs 2329 (reduced by 99.3%)
Total duration: 30.135s vs 30.111s (improved by 0.1%)
Avg checks per loop: 1039.7 vs 7.0

================================================================================
Summary of All Tests
================================================================================
Test Configuration             Simple Polling Checks Priority Queue Checks Efficiency Gain
--------------------------------------------------------------------------------
100 reminders/10s              5170                  199                   96.2%
500 reminders/30s              76504                 819                   98.9%
1000 reminders/60s             306113                1659                  99.5%
2000 reminders/30s             311905                2329                  99.3%
```

FIGURE VI: Reminder Verlauf

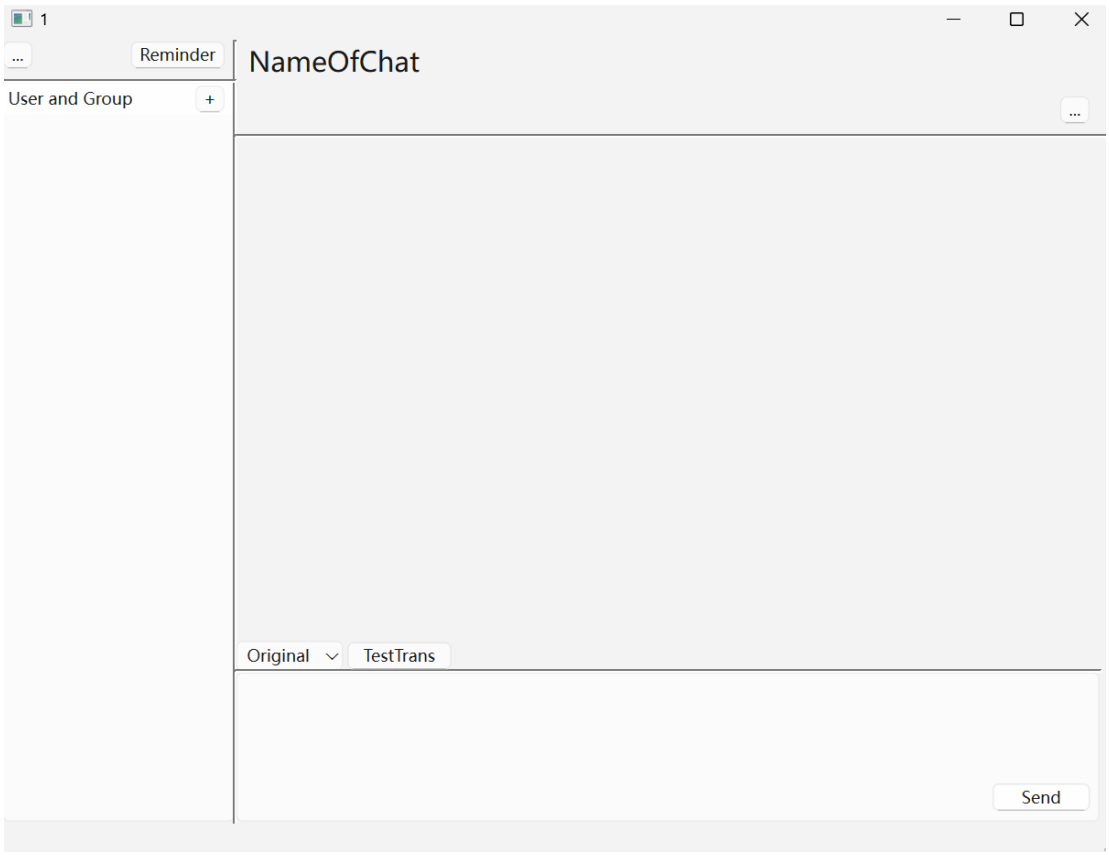


FIGURE VII: Alte Version

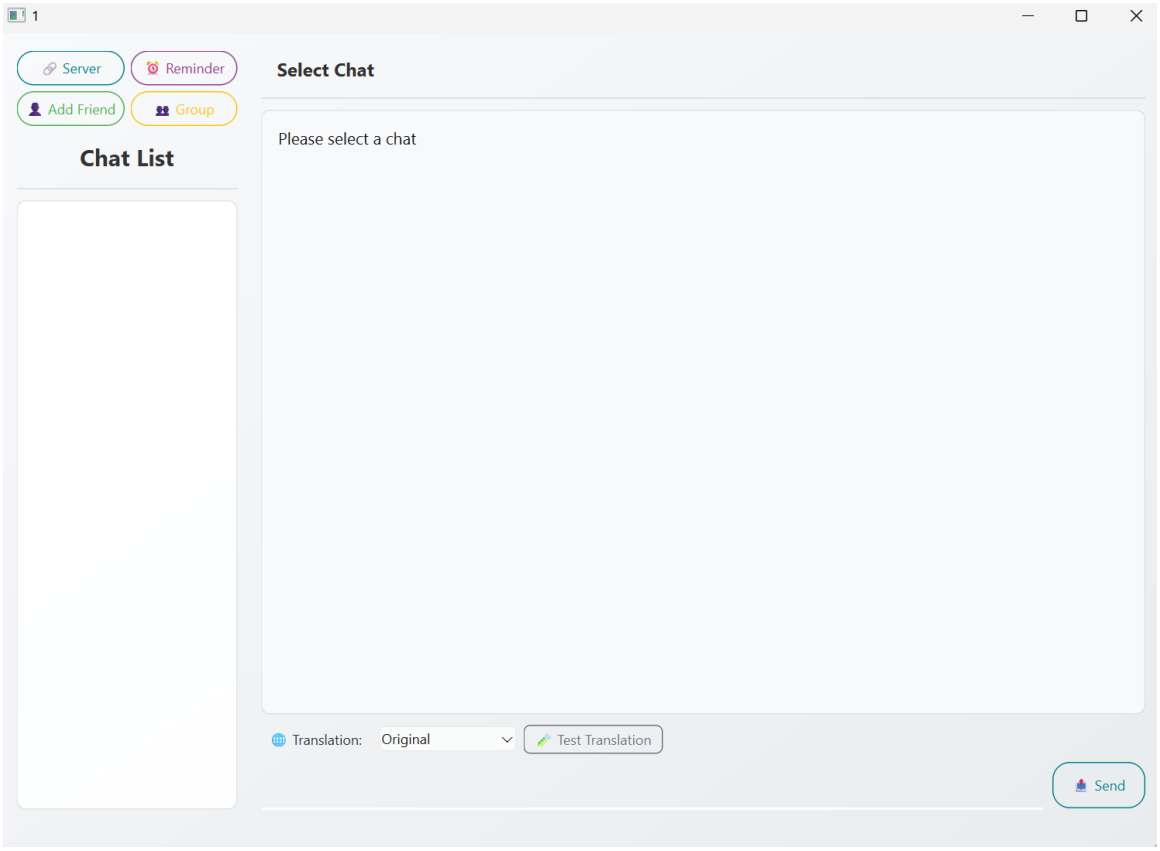


FIGURE VIII: Neue Version

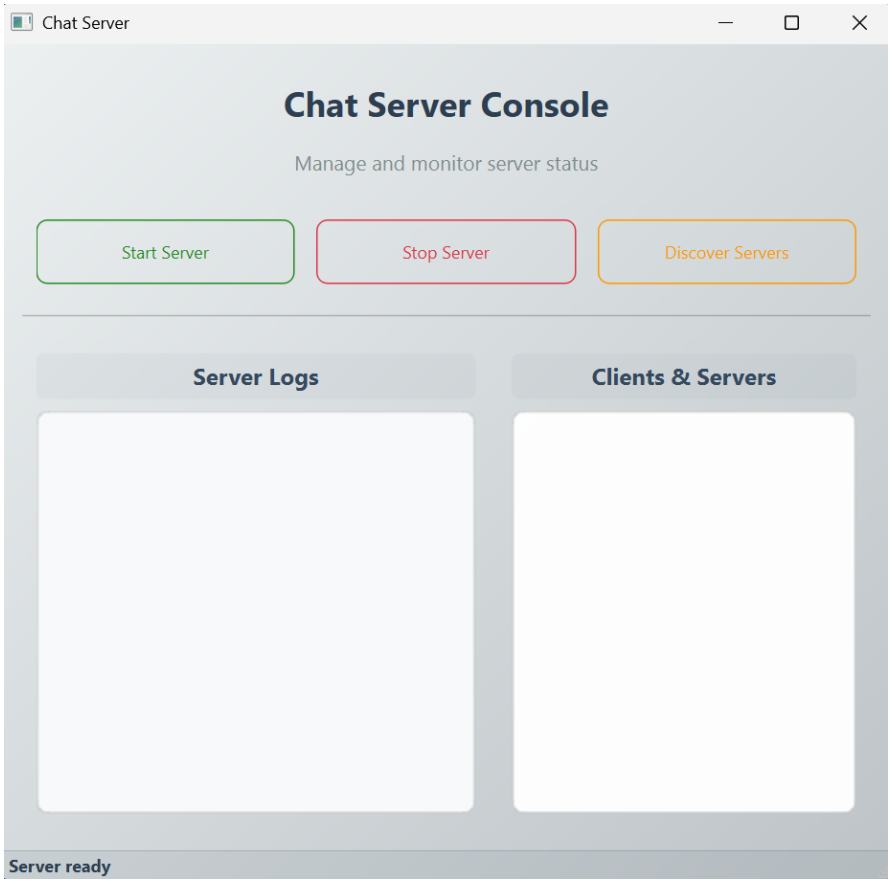


FIGURE IX: Server GUI mit Effekt

FIGURE X: Die Diskussion mit Gruppe 2