Technische Dokumentation: Peer-to-Peer Chatprogramm (“ByteMe”)

Marvin Ehmler (1507605),

Sadik Busch (1530535),

Batuhan Erdogan (1518522),

Muhammed Emin Cicek (1532654)

---

1. Projektüberblick

Das Projekt “ByteMe” wurde im Rahmen der Lehrveranstaltung “Betriebssysteme und Rechnernetze” an der Frankfurt University of Applied Sciences im Sommersemester 2025 entwickelt. Es handelt sich um ein dezentrales Peer-to-Peer-Chatprogramm, das es Nutzern ermöglicht, Textnachrichten und Bilder innerhalb eines lokalen Netzwerks auszutauschen, ohne dabei auf einen zentralen Server angewiesen zu sein.

Ziel der Umsetzung war es, praktische Erfahrungen mit Netzwerksockets, Interprozesskommunikation (IPC), Datei-I/O sowie der Entwicklung und Anwendung eines proprietären Kommunikationsprotokolls (SLCP) zu sammeln. Durch die Arbeit an einem realitätsnahen Szenario wurden zusätzlich wichtige Kompetenzen im Bereich Softwarearchitektur, Modularisierung und Prozesskommunikation vertieft.

Das Chatprogramm setzt vollständig auf Java und gliedert sich in mehrere modular aufgebaute Klassenpakete. Besonderes Augenmerk wurde auf saubere Trennung der Prozesse, Robustheit der Netzwerkkommunikation sowie einfache Erweiterbarkeit gelegt. Die Anwendung unterstützt sowohl Text- als auch Bildnachrichten und kann mit mehreren Clients im selben Netzwerk gleichzeitig betrieben werden.

---

2. Systemarchitektur

Die Architektur basiert auf einer klaren Trennung in drei Hauptprozesse:

- **Benutzeroberfläche (UI / CLI.java):** Diese Komponente ermöglicht die Interaktion mit dem Nutzer über eine Kommandozeilenschnittstelle. Sie nimmt Eingaben entgegen, verarbeitet sie und gibt Rückmeldungen oder empfangene Nachrichten aus.
- **Netzwerk-Kommunikation (NetworkManager.java / Receiver.java):** Dieser Teil verarbeitet ausgehende und eingehende SLCP-Nachrichten über UDP- und TCP-Sockets. Er ist für die direkte Kommunikation mit anderen Clients verantwortlich.
- **Discovery-Dienst (DiscoveryService.java):** Ein separater Dienst, der durch Broadcast-Kommunikation andere Teilnehmer im Netzwerk erkennt und aktiv verwaltet.

Diese Prozesse werden unabhängig voneinander gestartet und kommunizieren über benutzerdefinierte Interprozesskommunikation mittels Named Pipes. Der IPC-Mechanismus basiert auf synchronisierten FIFO-Dateien und stellt sicher, dass Nachrichten eindeutig und in richtiger Reihenfolge zwischen CLI und Netzwerkmodul ausgetauscht werden.

---

3. Modulbeschreibungen

3.1 Benutzeroberfläche (CLI.java)

Die Benutzeroberfläche ist als CLI (Command Line Interface) realisiert. Über ein Menüsystem oder direkte Befehle kann der Nutzer:

- einem Chat beitreten,
- andere Nutzer im Netzwerk abfragen,
- Textnachrichten senden,
- Bilder versenden,
- seine Konfiguration anzeigen und anpassen,
- das Programm beenden.

Die UI kommuniziert mit dem Netzwerkprozess über eine Pipe. Eingaben werden dort serialisiert und an das Netzwerkmodul übermittelt. Eingehende Nachrichten werden asynchron empfangen, formatiert dargestellt und ggf. mit Autoreply beantwortet, falls der Benutzer nicht aktiv ist.

3.2 Netzwerkkommunikation (NetworkManager.java, Receiver.java)

Dieses Modul bildet das Herzstück der SLCP-Kommunikation. Es ist zuständig für:

- Erstellen und Versenden von SLCP-Nachrichten (JOIN, MSG, IMG, LEAVE etc.)
- Empfang und Parsen eingehender Nachrichten über UDP
- Empfang von Bilddaten über TCP gemäß SLCP-Vorgabe
- Verwaltung der Teilnehmerliste zur Direktadressierung

`NetworkManager` kapselt die grundlegende Logik, während `Receiver` als eigenständiger Thread auf eingehende Pakete wartet und diese analysiert. Die Übergabe der Daten an das UI erfolgt ebenfalls über Pipes.

3.3 Discovery-Dienst (DiscoveryService.java)

Der Discovery-Dienst wird unabhängig von den Clients gestartet und läuft dauerhaft im Hintergrund. Seine Aufgaben umfassen:

- Lauschen auf WHO- und JOIN-Broadcasts
- Versenden von KNOWUSERS-Nachrichten bei WHO-Anfragen
- Verwalten einer dynamischen Liste aktiver Teilnehmer

- Entfernen von Nutzern nach Empfang einer LEAVE-Nachricht

Er verhindert durch Lock-Mechanismen Mehrfachstarts auf demselben Host. Seine Ausgabe ist für andere Komponenten transparent, da alle Anfragen über das Netzwerkmodul erfolgen.

---

4. Kommunikation & SLCP-Protokoll

Das Projekt setzt das Simple Local Chat Protocol (SLCP) vollständig um. Dieses Protokoll basiert auf textueller Syntax und definiert eindeutige Befehle zur Kommunikation.

SLCP-Nachrichten folgen diesem allgemeinen Format:

`<Befehl> <Parameter1> <Parameter2> ... <ParameterN>\n`

Alle Nachrichten sind UTF-8-kodiert und verwenden einen UNIX-Zeilenumbruch ().

Unterstützte Befehle:

- `JOIN <Handle> <Port>` – Anmeldung im Netzwerk per Broadcast
- `LEAVE <Handle>` – Abmeldung aus dem Netzwerk
- `WHO` – Anfrage zur Nutzererkennung
- `KNOWUSERS` – Antwort mit IP, Port und Handle aller bekannten Clients
- `MSG <Handle> <Text>` – Textnachricht an bestimmten Nutzer
- `IMG <Handle> <Size>` – Einleitung des Bildversands (nachfolgend TCP-Daten)

Der Bildtransfer erfolgt in zwei Phasen: zunächst die IMG-Kommandozeile, dann die exakte Anzahl an Bytes über TCP. Empfangene Bilder werden gespeichert und im Log dokumentiert.

---

5. Konfigurationsdatei

Die Datei `config.toml` wird automatisch beim ersten Start angelegt oder übergeben. Sie beinhaltet folgende Parameter:

```toml
handle = "marvin"
port = 5000
whoisport = 4000
ipcport = 7000
autoreply = "Bin gerade nicht erreichbar."
imagepath = "./images"
```

- `handle`: Benutzername des Clients
- `port`: UDP-Port zur Netzwerkkommunikation
- `whoisport`: Port für Broadcast-Anfragen
- `ipcport`: Port oder Bezeichner für interne FIFO-Kommunikation

- `autoreply`: Antwort bei Abwesenheit
- `imagepath`: Speicherort empfangener Bilder

Die Klasse `ConfigManager` kapselt Lese- und Schreibzugriffe und prüft beim Start auf Kollisionen oder Fehler.

---

6. Verwendete Technologien

- **Sprache:** Java 17
- **Netzwerk:** UDP und TCP via `DatagramSocket`, `Socket`
- **Prozesskommunikation:** Named Pipes via `Files.newInputStream` / `Files.newOutputStream`
- **Protokoll:** Eigenes Textprotokoll (SLCP)
- **Datenhaltung:** TOML (Konfig-Datei), Dateisystem (Bilder)
- **Build & Projektstruktur:** Maven (`pom.xml`), Source-Ordnerstruktur nach Konvention
- **Start-Skripte:** Shellskripte `start-client.sh`, `discovery.sh`

Alle Klassen enthalten Doxygen-kompatible JavaDoc-Kommentare zur automatisierten Dokumentation.

---

7. Besondere Herausforderungen & Lösungen

- **Prozessentkopplung:** Die Koordination mehrerer unabhängig laufender Prozesse wurde über FIFO-Dateien umgesetzt. Die IPC-Klassen abstrahieren den Zugriff und vermeiden Datenverlust.
- **Fehlertoleranz im Netzwerk:** SLCP-Nachrichten werden gegen leere oder fehlerhafte Eingaben geprüft. Die Netzwerkkomponenten ignorieren unvollständige Daten und geben strukturierte Fehlermeldungen aus.
- **UDP-Broadcast Einschränkungen:** Broadcast-Frames wurden in manchen Netzen gefiltert. Lösung: alternative manuelle Bekanntgabe der Teilnehmer via Konfigurationsdatei.
- **Vermeidung von Mehrfachstarts:** Discovery-Prozess prüft beim Start, ob bereits eine Instanz läuft (PID-/Lock-Datei).
- **Dateiübertragung:** Empfängt exakt die in `IMG` angegebene Byte-Anzahl über TCP, speichert diese und überprüft sie mit Checksummen.

---

8. Nutzungshinweise

Startvoraussetzungen:

Zuerst muss der Discovery-Dienst gestartet werden:

```
./discovery.sh
```

Anschließend kann man einen Nutzer erstellen:

```
./start-client.sh "<handle>" "<port>" "<ipcport>"
```

Wichtig: Die Parameter `port` und `ipcport` müssen für jeden Benutzer eindeutig sein, um Netzwerk- und IPC-Kollisionen zu vermeiden.

Verfügbare Kommandos:

- `join` – Anmelden am Chatnetzwerk (falls nicht automatisch geschehen)
- `msg <handle> <text>` – Textnachricht an bestimmten Nutzer senden
- `img <handle> <pfad>` – Bild senden (nur lokale Dateien)
- `who` – Aktive Nutzer anzeigen lassen
- `leave` – Netzwerk verlassen (senden von LEAVE)
- `beenden` – UI und IPC beenden, Prozesse schließen

Weitere Hinweise:

- Empfangene Bilder werden im Ordner `images` gespeichert
- Logs werden in der Konsole ausgegeben und enthalten Details zu Fehlern, Verbindungen, Protokollereignissen
- Alle Prozesse beenden sich sauber über `beenden`, um Named Pipes freizugeben und Zombie-Prozesse zu vermeiden