Technische Hochschule Wildau
Technical University of Applied Sciences

Dokumentation des Chat-Anwendungs-Projekts
Gruppe 2

Modul: Parallele und Verteilte Systeme

Gruppenmitglieder: Arne Gutschick, Tim Fuchs, Gourav Gourav, Carlos Slaiwa, Natalie Danner

Seminargruppen: I23, IDP23

Kursverantwortlicher: Peter Bernhardt

Wintersemester 2025/26

Berlin, den 07. Januar 2026

Inhaltsverzeichnis

1. Auflistung der implementierten Erweiterungen 3
2. Design-Entscheidungen 3
2.1 Dateiaustausch 3
2.2 Befehlsverarbeitung 4
2.3 Statistikbefehl 4
2.4 Privatchat (1:1 Messages) 4
2.5 Farbkodierung/Benutzernamen 5
2.6 Asynchrones Nachrichtensenden 5
2.7 Nutzerliste und Status 6
2.8 Heartbeatservice 6
3. Architektur-Diagramm 6
4. Aufteilung der Aufgaben auf die Teammitglieder 7
5. Lessons Learned 7

1. Auflistung der implementierten Erweiterungen

Aufwandskategorie	Titel der Erweiterung	zugeteiltes Mitglied
Leichter Aufwand	5. Befehlsverarbeitung	Natalie Danner
Mittlerer Aufwand	7. Eindeutige Benutzernamen & Farbkodierung	Carlos Slaiwa
Mittlerer Aufwand	8. Private Nachrichten (1:1 Chat)	Gourav Gourav
Mittlerer Aufwand	10. Statistik-Funktion	Tim Fuchs
Mittlerer Aufwand	12. Einfacher Dateiaustausch	Arne Gutschick

Um die Funktionalität der oben genannten Use Cases zu ermöglichen, wurden zusätzlich die Erweiterungen 4 und 6 umgesetzt. Zu bewerten sind jedoch vorwiegend die oberen Erweiterungen.

Aufwandskategorie	Titel der Erweiterung	zugeteiltes Mitglied
Leichter Aufwand	4. Asynchrones Nachrichtensenden	Arne Gutschick
Leichter Aufwand	6. Nutzerliste & Status	Arne Gutschick

2. Design-Entscheidungen

2.1 Dateiaustausch

Der Dateiaustausch erfolgt auf Basis eines *Publish/Subscribe*-Mechanismus. Der Sender-Client verschickt einen *FileSendCommand*, der auf einem konzeptionellen *FileMessage*-Objekt basiert. Der Server empfängt den Command, verarbeitet ihn und schickt selbst ein *BroadcastFileEvent* an alle Clients.
Die Dateien werden als Base64-kodierte Strings zusammen mit Informationen zur Dateigröße, Dateiname und Absender versendet. Die Entscheidung für Base64-Kodierung erlaubt den Transport binärer Inhalte über das bestehende textbasierte Nachrichtensystem ohne Protokollerweiterung.
Zur Vermeidung unnötiger Serveranfragen wird das Größenlimit (< 1 MB) und die Existenz der Datei direkt im Client überprüft.
Um die Reaktivität der Anwendung zu gewährleisten, werden Dateien asynchron gelesen und geschrieben. Empfangene Dateien werden im Ordner Downloads/Chat gespeichert, der bei Nichtvorhandensein automatisch erstellt wird.
Für eine übersichtliche Codestruktur haben wir im Chatserver einen eigenen *FileHandler* implementiert, der die Nachricht abonniert, an den *FileService* zur Weiterverarbeitung leitet und die Datei wieder versendet. Analog dazu gibt es im Chatclient-Backend einen eigenen *FileService*, der die Validität der Datei überprüft und diese verschickt, sowie einen *MessageSubscriber*, der die Datei empfängt und für alle Clients außer dem Absender ein Dialogfenster zum Speichern anzeigt.

2.2 Befehlsverarbeitung

Die Erkennung und Vorverarbeitung von Befehlen erfolgt clientseitig. Eingaben, die mit einem „/“ beginnen, werden vom Client als Befehle interpretiert und bereits lokal auf syntaktische Korrektheit geprüft. Dadurch können fehlerhafte oder unvollständige Befehle frühzeitig abgefangen und unnötige Serveranfragen vermieden werden.
Architektonisch kommen zwei unterschiedliche Kommunikationsmuster zum Einsatz, abhängig von der Art des Befehls und dem gewünschten Antwortverhalten:

- *Publish/Subscribe* wird für Befehle verwendet, deren Ergebnis an mehrere oder alle Clients verteilt werden soll, beispielsweise */msg* oder */sendfile*. Diese Befehle werden als Commands an den Server gesendet und dort als Events an die relevanten Clients weiterveröffentlicht.
- *Request/Response (RPC)* wird für Befehle genutzt, die eine gezielte Antwort ausschließlich an den anfragenden Client erfordern, bspw. */time*. In diesem Fall wird eine synchrone Anfrage an den Server gestellt, der die Antwort direkt an den Ursprungsclient zurücksendet.

Der Befehl */help* wird vollständig lokal im Client verarbeitet, da hierfür keine serverseitigen Daten benötigt werden. Stattdessen wird unmittelbar eine Hilfenachricht mit den verfügbaren Befehlen angezeigt. Der Befehl */time* wird hingegen per *RPC* an den Server delegiert, sodass die Serverzeit als *Single Point of Truth* genutzt werden kann.
Innerhalb des Clients übernimmt die *ChatLogic* die zentrale Steuerung der Befehlsverarbeitung. Sie leitet erkannte Befehle an den jeweils zuständigen Service weiter, welcher die entsprechende Anfrage an den Server initiiert oder im Falle lokal verarbeitbarer Befehle direkt die Antwort erzeugt.

2.3 Statistikbefehl

Für den */stats*-Befehl wurde ein eigenes *RPC*-Protokoll implementiert, das auf den Nachrichten *StatisticsRequest* und *StatisticsResponse* basiert. Dadurch wird sichergestellt, dass statistische Auswertungen gezielt und ausschließlich an den anfragenden Client zurückgegeben werden. Die dafür notwendigen Schnittstellen sind in eigenen *Chatcontracts* definiert, einschließlich eines zusätzlichen *DTO*s zur Abbildung der Top-Chatter-Statistik.
Auf Serverseite werden die Nutzer- und Nachrichtenstatistiken in einem *Concurrent Dictionary* verwaltet. Diese Datenstruktur ermöglicht ein threadsicheres Aktualisieren der Statistiken auch bei parallel eingehenden Nachrichten und stellt somit die Konsistenz der Auswertungen sicher.
Command-Messages werden bewusst nicht als Nutzernachrichten gespeichert, sodass ausschließlich „echte“ Chatnachrichten in die statistische Auswertung einfließen.
Im Chatclient erfolgt das Versenden der Anfrage über den *MessageService*, auf dem Server empfängt ein *StatisticsHandler* die Request, fragt die Statistik vom *StatisticsService* ab und schickt die Response zurück an den Client, der sie im *MessageSubscriber* empfängt.

2.4 Privatchat (1:1 Messages)

Für den Versand privater Nachrichten wurde ein eigenes *Publish/Subscribe*-Protokoll implementiert, das auf den Chatcontracts *SendPrivateMessageCommand* und *PrivateMessageEvent* basiert. Dieses Protokoll ermöglicht den gezielten Austausch von Nachrichten zwischen zwei Clients, ohne andere Teilnehmer des Chats einzubeziehen.

Zur Umsetzung wird ein *Custom Topic Exchange* verwendet, der so konfiguriert ist, dass private Nachrichten ausschließlich an den Sender und den angegebenen Empfänger zugestellt werden. Dadurch wird die Vertraulichkeit privater Kommunikation gewährleistet, während gleichzeitig die bestehende *Pub/Sub*-Infrastruktur wiederverwendet werden kann.
Vor dem Versand einer privaten Nachricht überprüft der Server, ob der angegebene Empfänger in der threadsicheren Nutzerliste registriert ist. Existiert der Empfänger nicht, wird die Nachricht nicht weitergeleitet. Stattdessen sendet der Server ein *ErrorEvent* an den Absender, das diesen über den nicht zulässigen bzw. fehlgeschlagenen Befehl informiert.
Auch hier besteht die Clientlogik aus *MessageService* und *MessageSubscriber*, während im Server ein eigener *PrivateMessageHandler* und *PrivateMessageService* genutzt werden.

2.5 Farbkodierung/Benutzernamen

Die Nutzerverwaltung erfolgt serverseitig über eine threadsichere Nutzerliste, die als *Single Point of Truth* für alle verbundenen Clients dient. Neben der Verwaltung der Nutzernamen übernimmt der Server auch die Farbzuweisung für die Darstellung der Nachrichten. Die Farben werden zyklisch aus einem vordefinierten Pool möglicher Farben vergeben, wodurch eine konsistente und kollisionsfreie Zuordnung sichergestellt wird.
Für den Login-Prozess wurde ein eigenes *RPC*-Protokoll implementiert, das auf *LoginRequest*- und *LoginResponse*-Nachrichten basiert. Dieses Verfahren ermöglicht eine gezielte Rückmeldung ausschließlich an den anfragenden Client.
Der Server überprüft die Validität des Nutzernamens anhand klar definierter Regeln: Nutzernamen dürfen nicht leer sein, keine Leerzeichen oder Sonderzeichen enthalten und müssen innerhalb der aktuellen Sitzung eindeutig sein. Wird eine dieser Bedingungen verletzt, lehnt der Server die Login-Anfrage ab. In diesem Fall wird dem Client ein entsprechender Fehler übermittelt, wodurch das Client-Programm beendet wird. Dadurch wird sichergestellt, dass nur gültige und eindeutig identifizierbare Nutzer am Chat teilnehmen.
Im Client erfolgt das Versenden der Login-Anfrage über einen *LoginService*, während im Server ein *UserHandler* und *UserService* für Login und Logout zuständig sind.

2.6 Asynchrones Nachrichtensenden

Um ein zuverlässiges und benutzerfreundliches Versenden von Dateien über das Clientterminal zu ermöglichen, wurde eine leichtgewichtige *Terminal-GUI* in die Anwendung integriert. Diese abstrahiert die reine Konsoleneingabe und bietet Funktionen wie automatisches Scrollen, Textwrapping sowie ein separates Eingabefeld, das das asynchrone Senden und Empfangen von Nachrichten unterstützt. Dadurch bleibt die Benutzeroberfläche auch während laufender Dateiübertragungen reaktiv.
Zusätzlich ermöglicht die *Terminal-GUI* das Öffnen eines Dialogfensters zum Speichern empfangener Dateien. Dies verbessert die Benutzererfahrung und stellt sicher, dass empfangene Dateien gezielt und unter dem ursprünglichen Dateinamen abgelegt werden können.
Die asynchrone Kommunikation mit dem Server wird durch die Nutzung der *EasyNetQ*-Methoden *PublishAsync()* und *RequestAsync()* realisiert. Diese erlauben ein nicht-blockierendes Versenden von Nachrichten sowohl im *Publish/Subscribe*-Kontext als auch im *RPC*-basierten *Request/Response*-Verfahren und tragen somit zur Reaktivität der Anwendung bei.

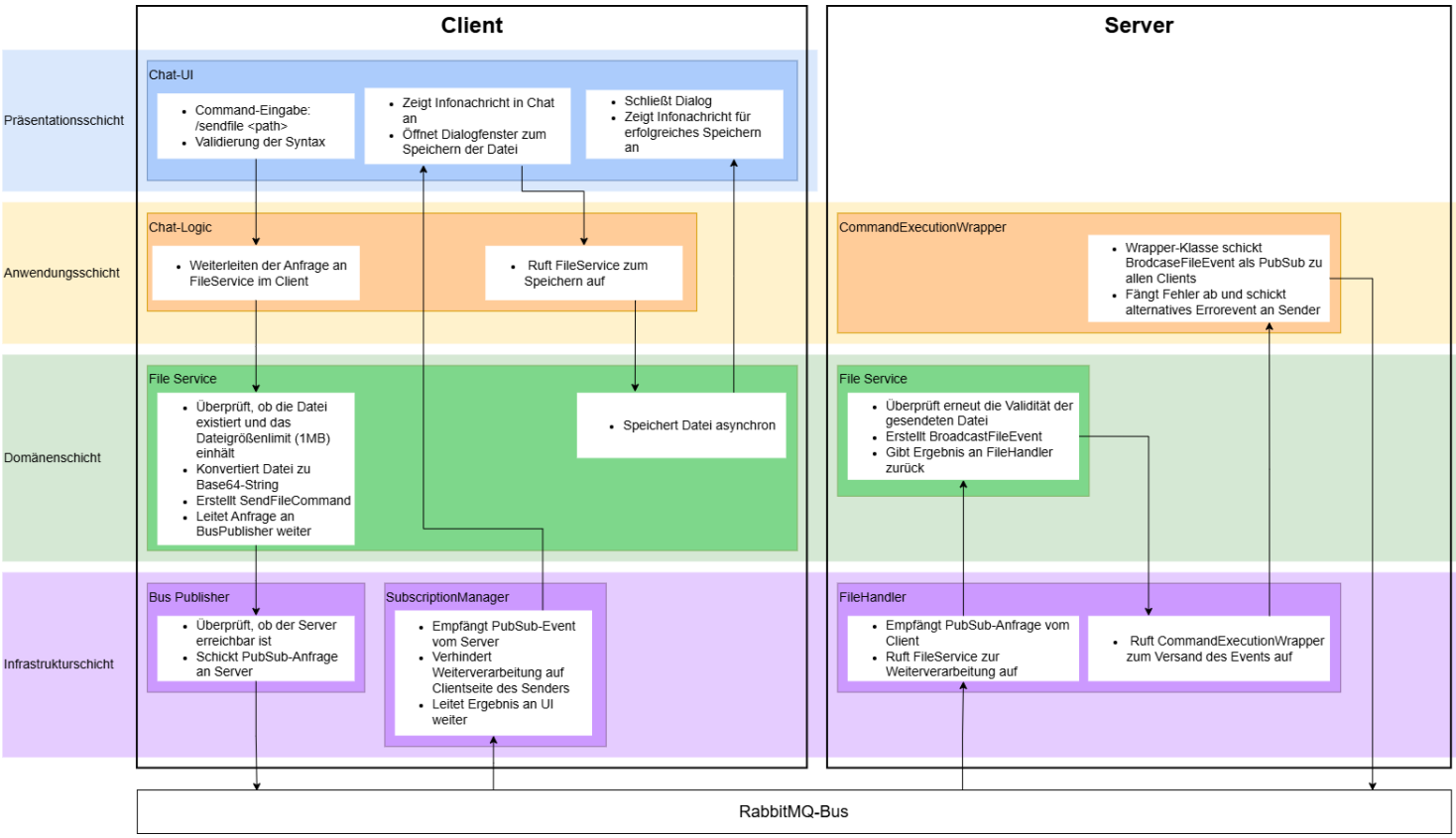
2.7 Nutzerliste und Status

Zur Unterstützung der Farbzuweisung sowie der statistischen Auswertungen werden eingeloggte Nutzer serverseitig in einem threadsicheren *ConcurrentDictionary* verwaltet. Diese Datenstruktur ermöglicht konsistente Lese- und Schreibzugriffe auch bei parallel eingehenden Ereignissen und dient als zentrale Grundlage für nutzerbezogene Funktionalitäten.
Beim Beitritt eines neuen Clients zum Chat werden alle anderen Clients über eine *Publish/Subscribe*-Benachrichtigung informiert. Dadurch bleibt der Nutzerstatus für alle Teilnehmer aktuell, ohne dass zusätzliche Abfragen erforderlich sind.
Ergänzend wurde der Befehl */users* implementiert, der über eine *RPC*-Anfrage eine Liste aller aktuell eingeloggten Nutzer vom Server abruft. Dieser Befehl wird automatisch beim Betreten des Chats ausgelöst, sodass der Client unmittelbar nach dem Login über den aktuellen Nutzerbestand informiert ist.

2.8 Heartbeatservice

Zur Erhöhung der Zuverlässigkeit und Erreichbarkeit der Chat-Anwendung wurde ein *Heartbeat*-Service implementiert. Dieser überprüft kontinuierlich die Konnektivität zwischen Clients und Server. Sowohl der Server als auch die Clients senden in regelmäßigen Intervallen *Heartbeat*-Nachrichten an die jeweils andere Seite, um ihre Erreichbarkeit zu signalisieren.
Fällt ein *Heartbeat* aus, bspw. durch einen Ausfall von *RabbitMQ* oder einen Client-/Server-Crash , erkennt der *Heartbeat*-Service diesen Zustand. Infolgedessen werden zukünftige Anfragen vom betroffenen Client oder Server sofort abgefangen.

3. Architektur-Diagramm



4. Aufteilung der Aufgaben auf die Teammitglieder

Hinweis: Die Aufteilung der Erweiterungen ist unter "Auflistung der implementierten Erweiterungen" zu finden

zugeteiltes Mitglied	Erweiterung	weitere übernommene Aufgaben
Natalie Danner	Befehlsverarbeitung	Erstellung der Dokumentation
Carlos Slaiwa	Eindeutige Benutzernamen & Farbkodierung	Input Validation für Methoden
Tim Fuchs	Statistik-Funktion	Erstellung der Konfigurationsdatei
Arne Gutschick	Einfacher Dateiaustausch	Error Handling, UI Design, Implementierung des Heartbeat-Services auf Client- und Serverseite
Gourav Gourav	Private Nachrichten	Code-Kommentare, Method Summaries

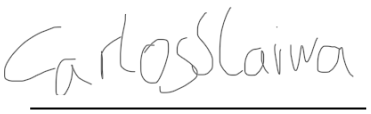
5. Lessons Learned

Während der Entwicklung unserer Chat-Anwendung haben wir mehrere wichtige Erkenntnisse gewonnen. Das Kapseln von Server- und *RabbitMQ*-Ausfällen im Error Handling erwies sich als aufwändiger als die Implementierung der eigentlichen Logik. Es hat sich gezeigt, dass robuste Fehlerbehandlung frühzeitig eingeplant werden muss, da sie einen erheblichen Einfluss auf die Gesamtarchitektur hat.
Die Aufteilung in *Services*, *Handler* und *Infrastructure-Components* war komplex und teilweise umständlich, da wir zunächst die einzelnen Erweiterungen in einer einzelnen Datei gesammelt hatten. Für zukünftige Projekte wäre es sinnvoll, das Design vor der Umsetzung der Use Cases gründlich zu durchdenken, um spätere Anpassungen zu minimieren.
Das Terminal-basierte UI brachte Herausforderungen mit sich. Hilfreiche Tools wie *Copilot* erzeugten häufig falsche Syntax, und die offizielle Dokumentation war teilweise schwer verständlich. Vermutlich wäre es einfacher gewesen, ein richtiges Frontend außerhalb des Terminals zu realisieren.
Besonders positiv hervorzuheben ist, dass die Syntax und Codestruktur für die einzelnen Use Cases durch *EasyNetQ* sehr konsistent war. Dies erleichterte die Implementierung erheblich und machte das Design der Kommunikationslogik insgesamt relativ unkompliziert.
Darüber hinaus hat die Arbeit mit *GitHub* sehr gut funktioniert. *Branching*, *Pull Requests* und Code-Reviews haben die Zusammenarbeit erleichtert und dafür gesorgt, dass Konflikte früh erkannt und gelöst werden konnten. Die Aufgabenverteilung im Team war klar strukturiert: Jedes Teammitglied war für die konsistente und vollständige Implementierung einer bestimmten Erweiterung verantwortlich. Zusätzliche Aufgaben wurden nach Fertigstellen der Grundfunktionalität gemeinsam zugeteilt und es wurde darauf geachtet, dass die parallele Bearbeitung gut möglich ist. Dadurch konnte effizient gearbeitet werden und jeder wusste, für welchen Teil er die Verantwortung trägt.

Anhang

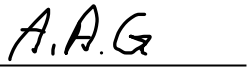
Selbstständigkeitserklärung

Hiermit erklären wir, dass wir wörtlich oder sinngemäß aus anderen Werken – dazu gehören auch Internetquellen – übernommene Inhalte als solche kenntlich gemacht und die entsprechenden Quellen angegeben habe. Wir willigen ein, dass unsere Arbeit mittels einer Software auf Plagiate überprüft werden kann. Uns ist bekannt, dass es sich bei der Abgabe eines Plagiats um ein schweres akademisches Fehlverhalten handelt und dass Täuschungen nach der für uns gültigen Studien- und Prüfungsordnung geahndet werden. Die vorliegende Arbeit haben wir selbstständig und ohne jede unerlaubte Hilfe konzipiert und angefertigt und keine anderen als die erlaubten und im Moodle-Kurs angegebenen Quellen und Hilfsmittel benutzt, sowie eigene Textbausteine keiner anderen Person zur Verfügung gestellt. Uns ist bewusst, dass wir Autoren der vorliegenden Arbeit sind und volle Verantwortung für den Text tragen. Zusätzlich versichern wir, dass wir IT-gestützte oder auf künstlicher Intelligenz (KI) basierende Schreibwerkzeuge nur in Absprache mit der Betreuungsperson verwendet haben. Dabei stand unsere eigene geistige Leistung im Vordergrund, und wir haben jederzeit den Prozess steuernd gearbeitet. Sofern die zuständigen Prüfenden bis zum Zeitpunkt der Ausgabe der Aufgabenstellung konkrete KI-gestützte Schreibwerkzeuge ausdrücklich als nicht anzeigepflichtig benennen, müssen diese nicht aufgeführt werden. Wir willigen ein, dass unsere Arbeit mittels einer Software auf KI-Textbausteine überprüft werden kann.



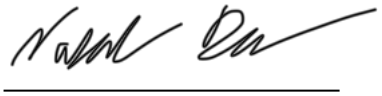
Carlos Slaiwa

Tim Fuchs



Arne Gutschick

Gourav Gourav



Natalie Danner