TuiTalk

Projektarbeit

für die Vorlesung

Verteilte Systeme

des Studiengangs Informatik

an der Dualen Hochschule Baden-Württemberg Heidenheim

von

Behr Tobias, Schönig Marc, Seidl Anian

September 2025

Bearbeitungszeitraum 8 Wochen

Erklärung

Wir versichern hiermit, dass wir unsere Projektarbeit mit dem Thema: *TuiTalk* selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben. Wir versichern zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Heidenheim, September 2025

Behr Tobias, Schönig Marc, Seidl Anian

Abstract

Inhaltsverzeichnis

Abkürzungsverzeichnis

WASM	Web Assembly
CLI	Command Line Interface
WS	WebSocket

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Einleitung

Kommunikation erfolgt heutzutage über eine Vielzahl von Plattformen. Bekannte Anwendungen wie WhatsApp, Discord oder Signal bieten bereits umfangreiche Möglichkeiten zum Austausch von Nachrichten. Das Ziel dieses Projekts ist die Entwicklung einer einheitlichen Lösung, die sowohl über das Terminal als auch über den Browser genutzt werden kann. Hierfür wird ein Backend implementiert, das durch eine Terminal-Anwendung und eine Weboberfläche ergänzt wird. Ein wesentlicher Unterschied zu bestehenden Lösungen besteht darin, dass die Nutzung ohne vorherige Registrierung möglich ist.

2 Architektur

2.1 Anforderungen

Im Folgenden werden anhand der aufgelisteten internen und externen Stakeholder die funktionalen und nicht-funktionalen Anforderungen aufgeführt.

Interne Stakeholder

- Entwickler
- Betreuer
- Systemadministrator

Externe Stakeholder

- Nutzer

2.1.1 Funktionale Anforderungen

- Client Kommunikation: Als Nutzer und Entwickler möchte ich mich über WebSockets mit dem System verbinden können, wobei ich zwischen einem CLI- und einem WASM-Client entscheiden möchte, damit ich meine präferierte Umgebung verwenden kann. Ich möchte Nachrichten in Echtzeit senden und empfangen, Räumen beitreten und verlassen, sowie meinen Anzeigenamen ändern können. Darüber hinaus soll es möglich sein, vergangene Nachrichten laden zu können, damit ich den gesamten Chatverlauf einsehen kann.
- Loadbalancer: Als Nutzer und Systemadministrator möchte ich, dass ein Loadbalancer die WebSocket Verbindungen gleichmäßig auf die vorhandenen Backends verteilt, damit eine optimale Performance erreicht werden kann. Wenn ein Backend ausfällt, sollen die bestehenden Verbindungen nahtlos vom Loadbalancer zu anderen Backends umgeleitet werden, sodass ich beim Benutzen keine Unterbrechung bemerke.

- Backend Services: Als Entwickler möchte ich, dass die Client Verbindungen vom Backend verwaltet werden und Nutzer spezifische Buffer verwendet werden, um Nachrichten effizient und zuverlässig verarbeiten zu können. Nachrichten sollen über ein Publish/Subscribe-System verteilt werden, damit alle Backendinstanzen synchron sind und in einer Datenbank gespeichert werden, um Chatverläufe abrufen zu können. Das System soll Pub/Sub-Mechanismen als Cluster unterstützen, damit es Skalierbar und Ausfallsicher ist.

2.1.2 Nichtfunktionale Anforderungen

- Hohe Verfügbarkeit: Als Nutzer, Systemadministrator und Betreuer möchte ich, dass das System weiterhin funktioniert, auch wenn ein Backend ausfällt, ohne das der Nutzer eine Unterbrechung mitbekommt. Nach dem Ausfall eines Backends soll die Verbindung automatish von einem anderen Backend übernommen werden ohne das Nachrichten verloren gehen.
- Zuverlässigkeit: Als Nutzer und Systemadministrator möchte ich, dass Nachrichten zugestellt werden, solange die Datenbank und mindestens ein Backend verfügbar sind, so dass Nachrichten nicht verloren gehen, auch wenn Teile des Systems ausfallen. Die Konsistenz der Daten soll durch eine Datenbank sichergestellt werden, sodass keine Nachrichten des Chatverlaufs fehlen und dieser damit eindeutig und zuverlässig ist.
- Skalierbarkeit: Als Entwickler und Betreuer möchte ich, dass das System horizontal skalierbar ist, damit es auch unter hoher Last performant bleibt. Es sollen mehrere verteilte Pub/Sub-Instanzen und Backends verwendet werden.

2.2 Architekturentscheidungen

2.2.1 Rust

Als Programmiersprache wird Rust genutzt. Da Rust so vielseitig einsetzbar ist wird diese für die Clients als auch für das Backend verwendet. Dabei wird im Backend die Rust-Library *tokio* genutzt, um Asynchrone Aufgaben zu verwalten. Im Web Assembly (WASM)-Client wird die Rust-Library *wasm-bindgen* genutzt, um WebAssembly zu nutzen. Im Command Line Interface (CLI)-Client wird die Rust-Library *ratatui* genutzt, um die Kommandozeile als User Interface zu nutzen. Die Programmiersprache wurde aus bestehendem Eigeninteresse in diesem Projekt verwendet.

2.2.2 Redis

Redis ist eine in-memory Datenbank, die eine einfache und schnell zu verwendende, leistungsfähige, skalierbare und zuverlässige Datenbank ist. Allerdings ist der Hauptnutzen für das System in dieser Arbeit die Publish und Subscribe Funktionalität. Das Setup eines Redis Containers ist sehr einfach und benötigt wenig Konfiguration, dadurch eignet sich Redis als Kommunikationsschnittstelle unter den Backendinstanzen.

2.2.3 Loadbalancer

Als Loadbalancer wird *Nginx* genutzt. Nginx bietet nicht nur Stabilität und bewährte Performance, sondern ermöglicht auch flexible Strategien zur Lastverteilung. In diesem Projekt wird die Strategie *Least Connections* verwendet. Dabei werden neue Verbindungen an den Server mit den aktuell wenigsten aktiven Verbindungen weitergeleitet. Dies führt zu einer gleichmäßigeren Auslastung der Backend-Instanzen, insbesondere in Szenarien mit langen Verbindungen, wie sie bei WebSocket-Kommunikation üblich sind.

2.2.4 PostgreSQL-Datenbank

Für die persistente Datenspeicherung wird eine *PostgreSQL*-Datenbank eingesetzt. PostgreSQL wurde gewählt, da sie ein stabiles, weit verbreitetes und leistungsfähiges relationales Datenbanksystem ist. Die Datenbank PostgreSQL wurde auch ausgewählt, da die Entwickler dieses Projekts bereits mit PostgreSQL gearbeitet haben. Die Trennung zwischen Redis (für flüchtige, schnelle Daten) und PostgreSQL (für persistente Daten) ermöglicht eine klare Architektur, bei der beide Systeme ihre jeweiligen Stärken optimal ausspielen können. Jede andere relationale SQL-Datenbank wie MySQL oder MariaDB hätte im Rahmen dieses Projekts auch ausgereicht.

2.3 Systemkomponenten

Das System besteht aus fünf Komponenten: *Client*, *Backend*, *Redis-Cluster*, *Loadbalancer* und *Datenbank*.

Client

Es gibt zwei verschiedene Arten von Clients:

- *CLI*

- *WASM*

Beide Clients verbinden sich mit dem Loadbalancer, dieser leitet auf eine Backendkomponente weiter, um Nachrichten zu senden und zu empfangen.

Backend

Die Backendkomponente ist ein in Rust geschriebener Websocket-Server. Diese verwaltet die Nachrichten und kommuniziert über das Redis-Cluster mit den anderen Backendinstanzen. Zu dem persistiert diese die Nachrichten und Informationen über den Nutzer in einer Datenbank. Es werden zur Ausfallsicherheit drei Instanzen aufgesetzt.

Redis-Cluster

Das Redis-Cluster ist die Kernkomponente für die Synchronisation der Nachrichten und stellt die Kommunikation unter den Backendinstanzen her. Dafür wird die Publish und Subscribe Funktion von Redis genutzt. Jeder Chatraum wird als Channel in Redis abgebildet. Jede Nachricht des Clients wird auf den jeweiligen Channel in Redis gepublished. Um eine Nachricht zu erhalten erstellt das Backend für jeden Client einen Subscriber auf dem entsprechenden Channel. Das Redis-Cluster besteht aus sechs Redis-Nodes, drei davon sind Master-Nodes und die restlichen drei sind zu den Master-Nodes die jeweiligen Replicas.

Loadbalancer

Der Loadbalancer ist eine Komponente, die die Verbindungen zwischen den Clients und den Backends verteilt. Als Loadbalancer wird Nginx genutzt, da dieser einer der renomiertesten in dieser Technologie ist. Als Strategie zur Aufteilung auf die Backendinstanzen wird nach dem Verfahren der wenigsten Verbindungen vorgegangen.

Datenbank

Die Datenbank ist eine PostgreSQL-Datenbank, die die Nachrichten und Informationen der Nutzer speichert. Die Backendinstanzen verbinden sich mit der Datenbank, um die Nachrichten zu laden und zu speichern. Zusätzliche werden die Nutzerinformationen und in welchem Chatraum diese sich befinden gespeichert. Die Datenbank ist keine Systemkritischen Komponente und sorgt im Falle eines Ausfalls nur für eine Funktionseinschränkung.

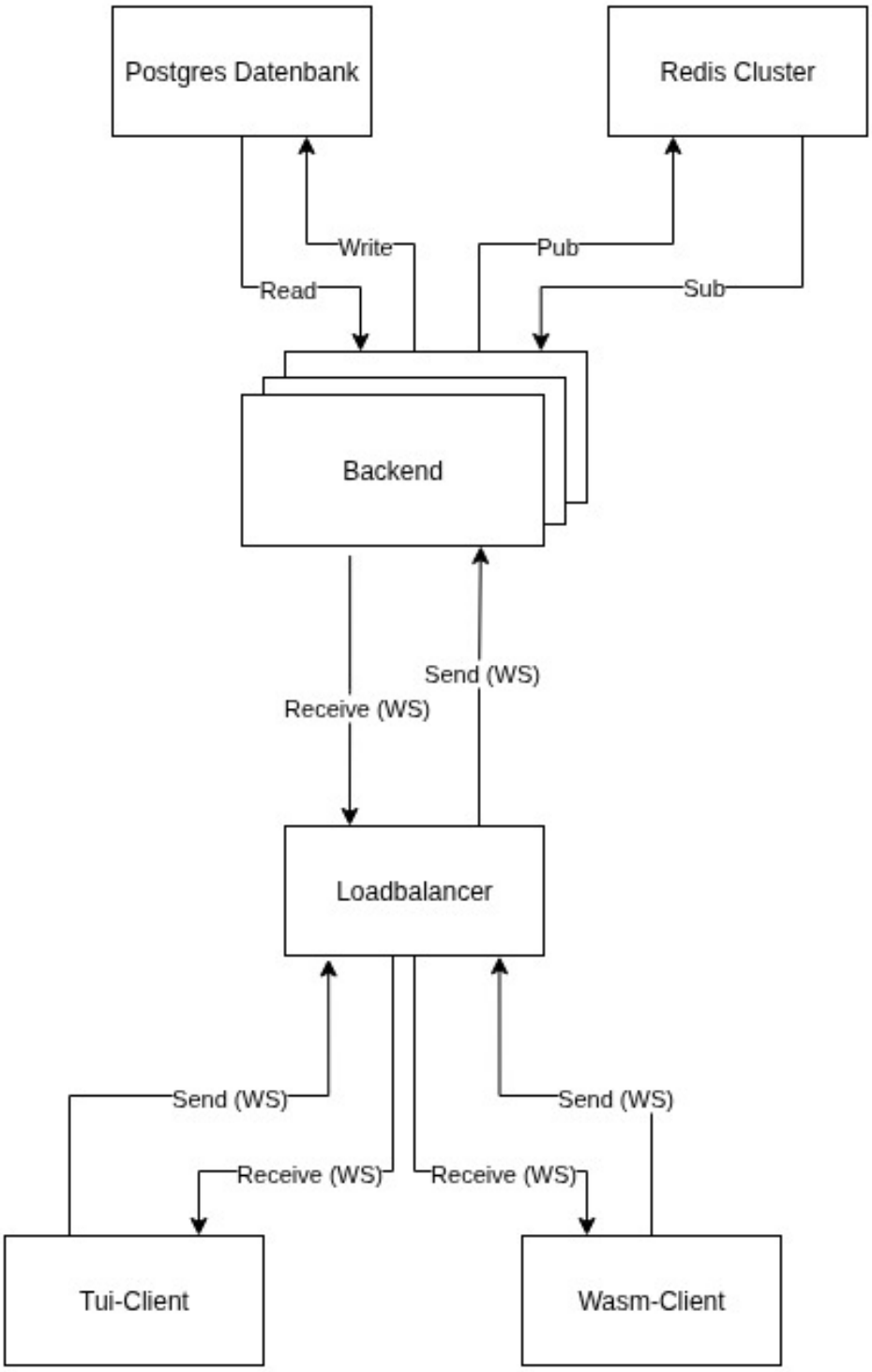

Abbildung 2.1: Architekturdiagramm

3 Umsetzung

Für die Kommunikation zwischen den Clients und dem Backend wurde ein einheitliches Protokoll definiert. Dieses beinhaltet Nachrichten, welche von den Clients zum Backend kommen, Nachrichten welche vom Backend zurück an die Clients gehen und Nachrichten, welche in beide Richtungen geschickt werden. Durch dieses Protokoll ist die Erweiterung für andere Nachrichtentypen ohne großen Aufwand möglich.

```
pub enum TalkProtocol {
    // Client -> Server Commands
    JoinRoom { room_id: i32, uuid: Uuid, username: String, unixtime: u64},
    LeaveRoom { room_id: i32, uuid: Uuid, username: String, unixtime: u64},
    ChangeName {uuid: Uuid, username: String, old_username: String, unixtime: u64},
    Fetch { room_id: i32, limit: i64, fetch_before: u64},
    LocalError { message: String },

    // Server -> Client Events
    UserJoined { uuid: Uuid, username: String, room_id: i32, unixtime: u64 },
    UserLeft { uuid: Uuid, username: String, room_id: i32, unixtime: u64  },
    UsernameChanged {uuid: Uuid, username: String, old_username: String, unixtime: u64
        },
    History { text: Vec<TalkProtocol> },
    Error { code: String, message: String },


    // Server <-> Client
    PostMessage { message: TalkMessage },
}
```

Listing 3.1: Kommunikationsprotokoll

3.1 Clients

Dieses Protokoll befindet sich neben dem Verbindungsaufbau in einer geteilten Bibliothek. Da die WebSocket (WS) Verbindung für die unterschiedlichen Clients auf unterschiedlichen Bibliotheken beruht mussten diese seperat implementiert werden. Um die Verbindungen trotzdem gesammelt haben wurden diese auch in die geteilte Bibliothek eingebunden.

3.1.1 Funktionalitäten

Beide Clients bieten neben den Grundfunktionen wie Senden und Empfangen von Nachrichten, die folgenden Funktionen:

Funktion	**Beschreibung**
/help	Zeigt Informationen zu den verfügbaren Befehlen
/clear	Löscht den Chatverlauf
/name <String>	Setzt den Nutzernamen
/room <Integer>	Wechselt den Raum
/fetch <Integer>	Holt ausgehend von der ersten Nachricht im Chatverlauf die vorherigen Nachrichten

Tabelle 3.1: Funktionen der Clients

3.2 Backend

4 Reflektion

4.1 Rust

Da vor diesem Projekt fast keine Vorkenntnisse zur Programmierung in Rust bestanden, war der Einstieg für dieses Projekt sehr aufwendig. Aufgrund dieser Einstiegshürde befinden sich in diesem Projekt Altlasten, welche in Zukunft für die weiterarbeit an diesem Projekt überarbeitet werden müssen, allerdings den Rahmen dieses Projekts gesprengt hätten.

4.2 Talkprotokoll

Initial war das Talkprotokoll auf einen Nachrichtentyp begrenzt, welcher wiefolgt aussah:

```
1 pub struct TalkMessage {
2     pub uuid: Uuid,
3     pub username: String,
4     pub text: String,
5     pub room_id: i32,
6     pub unixtime: u64
7 }
```

Listing 4.1: Altes Kommunikationsprotokoll

Dieses wurde für unter anderem für die Anzeige von Errors und Informationen wie Namensänderungen genutzt durch setzen des Namens zu Error oder Info und ähnlichem. Durch die Überarbeitung des Protokolls wurde es einfacher weitere Nachrichtentypen zu implementieren, wie zum Beispiel die Implementierung des Fetch Befehls.

4.3 Wasm-Client

Aufgrund der bereits genannten Einstiegshürde und da zu Beginn des Projekts mit dem Wasm-Client begonnen wurde und es initial nicht so funktioniert hat wie erwartet, hat sich diese Einstellung generell auf WebAssembly übertragen. Dies hat zu einer großen Bevorzugung des Tui-Client geführt, weshalb der Wasm-Client vernachlässigt wurde und nicht sein volles Potenzial entfalten konnte.

Anhang