RACOOSH

Ein P2P-Multiplayerspiel

DHBW Heidenheim

Allgemeine Informatik

Labor Verteilte Systeme (T3INF4223.2)

Sommersemester 2023

Dozenten

Marius Erlen, B.Sc.

Benjamin Salchow, M.Sc.

Studierende

Sandesh Dulal (7540129)

Philipp Erath (5633275)

Lasse Herzog (8099843)

Abgabetermin: 29.09.2023

Inhaltsverzeichnis

1 Allgemeines

1.1 Spielidee

Dieses Dokument beschäftigt sich mit Racoosh – einem Computerspiel, das von Studierenden des Studiengangs Allgemeine Informatik an der DHBW Heidenheim im Rahmen des Labors Verteilte Systeme entwickelt wurde. Racoosh ist Online-Multiplayerspiel, das neben einer Server-Client auch über eine Peer-to-Peer-Architektur verfügt und zur Ausführung in einem modernen Webbrowser gedacht ist. Racoosh verfügt neben dem eigentlichen Spielmechanismus über eine Spielelobby, in der sich Spieler zusammenschließen können, um gegeneinander anzutreten. Darüber hinaus verfügt die Implementierung über eine einfache Benutzerverwaltung (Registrierung und Anmeldung) sowie über eine Bestenliste der vergangenen Spiele.

Das Spielprinzip von Racoosh ist einfach. Mit bis zu drei Mitspielern kann ein Spiel bestritten werden. Jeder Spieler steuert hierbei einen Waschbären und versucht, andere Spieler abzuwerfen und somit auszuschalten. Gleichzeitig möchte man selbst nicht von den anderen Spielern getroffen werden. Auf der Spielkarte befinden sich neben Hindernissen, hinter denen man sich verstecken kann, auch Nahrung, um die eigene Gesundheit aufzufüllen. Treffer anderer Spieler vermindern die eigene Gesundheit. Bei leerer Gesundheit hat man das Spiel verloren. Das Spiel endet, sobald nur noch ein Spieler übrig ist. Dieser ist der Gewinner.

1.2 Verwendete Technologien

Da es sich bei Racoosh um ein Browserspiel handelt, hat sich das Team dazu entschieden, die Implementierung mit JavaScript-Technologien und -Frameworks umzusetzen. Im Folgenden wird in Bullet-Points auf den Technologiestack eingegangen und kurz beschrieben, welche warum an welcher Stelle eingesetzt wurden.

Backend

TypeScript: Sowohl im Frontend als auch im Backend wurde TypeScript als JavaScript-Superset verwendet, um Typsicherheit zu gewährleisten.

Express: User- und Gameservice sowie der Peerserver wurden in Form von Webservern über das Node.js-Framework Express umgesetzt.

RESTful APIs: User- und Gameservice verfügen über RESTful APIs, die als Schnittstellen zur Kommunikation mit dem Frontend dienen.

Websockets über Socket.IO: Der Gameservice verfügt darüber hinaus über eine Websocket-Schnittstelle, die mit Hilfe des Socket.IO-Frameworks umgesetzt wurde. Als integriertes Fallback bietet dieses Framework HTTP Long Polling an, damit höchste Verfügbarkeit gewährleistet wird.

MongoDB: Sowohl der User- als auch der Gameservice haben jeweils Zugriff auf eine eigene MongoDB-Instanz, um Datenpersistenz zu gewährleisten. Da die Daten des Projekts in einer natürlichen Aggregatsform (User und Spiele) vorkommen, fiel die Entscheidung auf eine dokumentenbasierte Datenbank.

Redis: Auch hier verfügt sowohl der User- als auch der Gameservice über eine eigene Instanz der In-Memory-Datenbank Redis. Im Userservice dient sie der JWT-Revocation und im Gameservice der Skalierung von Socket.IO-Instanzen[1].

Frontend

WebRTC über PeerJS: Die verschiedenen Client-Instanzen tauschen ihre Spieldaten per P2P-Netzwerk aus. Zur Implementierung wurde die JavaScript-Library PeerJS eingesetzt, welche die WebRTC-Technologie verwendet.

Three.js: Diese JS-Bibliothek wird im Client eingesetzt und dient der Darstellung des eigentlichen Spiels mit 3D-Computergrafik. Sie ist kompatibel mit vielen Browsern und verwendet WebGL zur Darstellung.

Svelte: Diese Technologie wurde verwendet, um die Benutzeroberflächen der Nutzerverwaltung und des Lobbymechanismus umzusetzen. Svelte gilt als leichtgewichtig und bietet schnelle Entwicklungserfolge.

nginx: Das Projekt verfügt über zwei nginx-Instanzen. Eine Instanz dient als Reverse Proxy und leitet einkommende Anfragen weiter. Eine zweite Instanz dient als Webserver und stellt das Frontend und den Spielclient in Form einer Webanwendung zur Verfügung.

---

[1] Vgl.: https://socket.io/docs/v3/using-multiple-nodes/

2 Architektur

2.1 Allgemein

Der Container Frontend beinhaltet die clientseitige Webanwendung und bündelt sowohl die Benutzeroberfläche der Spielelobby als auch den Spieleclient. Diese Instanz stellt die zentrale Spielsteuerung jedes Spielers dar und kann als Client angesehen werden.

Das Backend ist über einzelne Services verwirklicht. Es beinhaltet einen User- und einen Gameservice sowie einen sogenannten Peerserver[2]. Diese Struktur spiegelt sich auch in den Docker-Containern wider.

Racoosh ist sowohl über eine P2P-Architektur als auch über eine Server-Client-Architektur verwirklicht. Über die Server-Client-Struktur werden mehrere Aufgaben übernommen:

- Nutzer registrieren und anmelden: Hierfür kommuniziert das Frontend mit dem Userservice über RESTful APIs.
- Spiele anlegen, beitreten, verlassen und durch Verlust/Gewinn beenden: Hierfür kommuniziert das Frontend mit dem Gameservice über RESTful APIs bzw. Websockets.

Über die Server-Client-Struktur wird somit der Spielerahmen verwirklicht. Der Austausch der eigentlichen Spieldaten wurde über eine P2P-Architektur mit Hilfe von WebRTC umgesetzt. Hierbei kommunizieren die einzelnen Clients direkt miteinander. Zur Initiierung dieser Verbindung ist der Peerserver zuständig. Er dient als Signallingserver[3].

Das Team entschied sich hauptsächlich aus Interesse für diese Architektur, da noch kein Mitglied Erfahrungen mit P2P-Technologien gesammelt hatte. Der Vorteil einer solchen Struktur liegt in der Unabhängigkeit von einem zentralen Server beim Austausch der Spieledaten. Wie am Backend zu erkennen ist, ist für viele weitere Funktionen dennoch ein zentraler Server nötig. Nachteile einer solchen Architektur sind, dass keine zentrale Instanz den momentanen Spielzustand kennt und mehrere – zum Teil konkurrierende – Wahrheiten über das Spiel existieren. Im letzten Kapitel zur Reflektion wird nochmals näher auf diesen Punkt eingegangen.

---

[2] https://github.com/peers/peerjs-server

[3] https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Signaling_and_video_calling

2.2 Systemkomponenten

Racoosh besteht aus mehreren Schlüsselkomponenten, die ein reibungsloses Benutzererlebnis und Gameplay sicherstellen. Die Systemkomponenten sind in der nachfolgenden Komponentendiagramm dargestellt, das einen Überblick darüber bietet, wie diese Komponenten miteinander interagieren. Dieses Komponentendiagramm spiegelt auch die Struktur der Docker-Container wider und kann damit auch als Deployment-Diagramm aufgefasst werden. Die Grüns hinterlegten Container sind skalierbar.

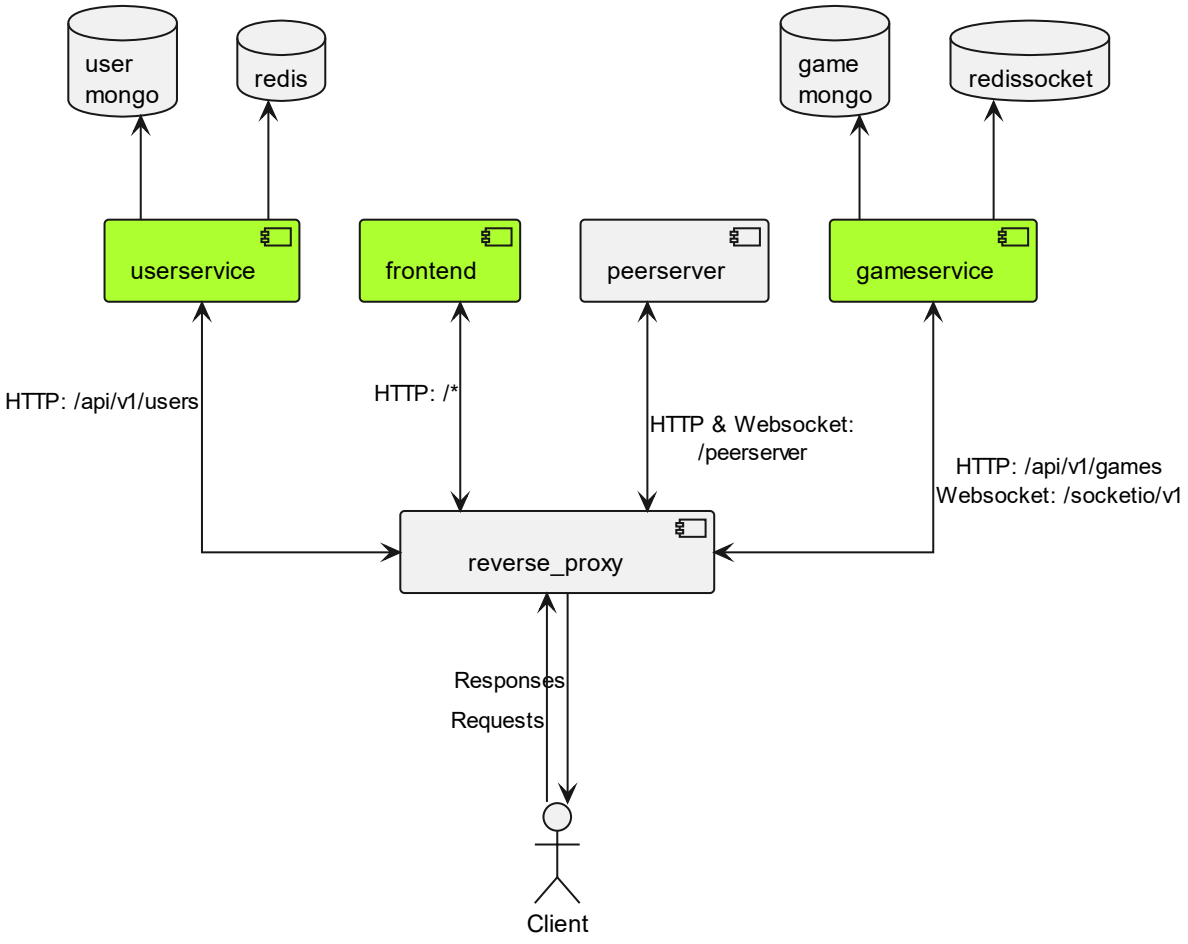

Abbildung 1: Komponentendiagramm

Reverse_Proxy: Nginx fungiert als Reverse-Proxy. Es verarbeitet die eingehenden Client-Anfragen und leitet sie zu den entsprechenden Backend-Diensten weiter. Es ist auch für die Lastverteilung und die Verwaltung des HTTP- und WebSocket-Datenverkehrs verantwortlich.

Frontend: Der Frontend-Container repräsentiert die Benutzeroberfläche und den Spieleclient von Racoosh. Dies ist ebenfalls ein Nginx-Server, der als Webserver fungiert und die Client-Anwendung an die Benutzer ausliefert. Er kommuniziert mit dem Reverse-Proxy, um Daten abzurufen und WebSocket-Verbindungen herzustellen. Die Frontend-Implementierung ist ein kooperativer Ansatz von zwei Frameworks, nämlich Svelte und Three.js. Das Svelte-Framework wurde verwendet, um alle Webseiten zu erstellen und die zugehörige Logik zu implementieren. Three.js kommt ins Spiel, wenn mehr als ein Benutzer das Spiel startet, und kümmert sich um verschiedene 3D-Elemente wie Waschbären, Steine usw.

Peerserver: Der PeerJS-Server spielt eine entscheidende Rolle bei der Ermöglichung der Peer-to-Peer-Verbindungen zwischen den Spielern, was das grundlegende Konzept dieser Anwendung ist. Die Implementierung einer Peer-to-Peer-Verbindung ist ein komplexes Thema, weshalb PeerJs verwendet wurde. PeerJs bietet die Schnittstelle zur einfachen Erstellung von Peer-to-Peer-Verbindungen zwischen den Clients. Die Logik, wie die Implementierung von STUN-Servern und TURN-Servern hinter der Schnittstelle, bleibt unberührt. PeerJs bietet seinen eigenen TURN-Server an, verwendet jedoch öffentlich verfügbare STUN-Server. Der Client-Code selbst handhabt das Thema, wann eine Peer-to-Peer-Verbindung hergestellt werden soll.

Userservice: Der Userservice ist ein Express-Server, der die REST-Endpunkte bereitstellt, die es den Spielern ermöglichen, Aktionen wie die Registrierung und das Anmelden durchzuführen. Diese Komponente kommuniziert mit der usermongo-Datenbank, um Benutzerprofile zu speichern und abzurufen. Außerdem übernimmt sie die Authentifizierung mithilfe von JWT (JSON Web Tokens), um die Identität der Benutzer zu überprüfen und die Sicherheit und Integrität des Spiels aufrechtzuerhalten. Sie verwaltet auch die JWT-Revocation, bei der die Redis-Datenbank verwendet wird, um Tokens von nicht authentifizierten Benutzern zu speichern. Dadurch wird sichergestellt, dass nur gültige Tokens Zugriff auf das Spiel erhalten.

Usermongo: Die usermongo ist eine dedizierte Datenbank, die verwendet wird, um die datenbezogenen Benutzerinformationen zu speichern. Diese Daten umfassen in der Regel Details wie Benutzername und den gehashten Wert des Passworts. Die in dieser Datenbank gespeicherten Passwort-Hashes durchlaufen sowohl Salting als Peppering. Die dedizierte Benutzerdatenbank wurde unter Berücksichtigung der Skalierbarkeit konzipiert und bietet somit eine Grundlage für die wachsende Anzahl von Benutzern und das Wachstum dieses Projekts selbst.

Redis: Wie bereits beschrieben, ist die Implementierung von Redis für die JWT-Revocation nützlich. Die Speicherung von Daten in Form von Schlüssel-Wert-Paaren ist der Hauptaspekt, der

für die Auswahl dieser Datenbank spricht. Diese Schlüssel-Wert-Paare haben auch eine Verfallszeit, was bedeutet, dass JWT-Daten im System gespeichert werden, bis sie abgelaufen sind. Es wurde auch aufgrund seiner schnellen Lese- und Schreiboperationen implementiert.

Gameservice: Der Gameservice ist wie der Userservice auch ein Express-Server. Er ist sowohl über RESTful APIs als auch über Websocket-Verbindungen vom Frontend aus erreichbar. Er wird dazu verwendet, Spiele zu erstellen, ihnen beizutreten bzw. zu verlassen und sie zu starten. Über eine permanente Websocket-Verbindung mit jedem Client über den gesamten Spieleverlauf hinweg, werden hierüber auch das Spielende sowie Verbindungsabbrüche von Clients gehandelt. Seine Schnittstellen sind ebenfalls über das JWT aus dem Userservice geschützt.

Gamemongo: Diese Komponente ist eine MongoDB-Instanz, in welcher die einzelnen Spieleobjekte abgespeichert werden.

Redissocket: Bei dieser Komponente handelt es sich um eine Redis-Instanz. Um den Gameservice, der auch als Socket.IO-Server fungiert, skalierbar zu machen, ist es nötig, eine solche Redis-Instanz einzubinden. Hierdurch können Nachrichten zwischen den Socket.IO-Instanzen ausgetauscht werden, um ein Broadcasting an mehrere Clients sicherzustellen[4]. Dies ist etwa bei Verbindungsabbruch eines Spielers nötig.

[4] Vgl.: https://socket.io/docs/v3/using-multiple-nodes/

2.3 Anforderungen

In diesem Abschnitt wird eine umfassende Darstellung sowohl der funktionalen als auch der nicht-funktionalen Anforderungen vorgenommen. Diese Anforderungen spielten eine entscheidende Rolle bei der Gestaltung des Projekts. Zunächst einige der funktionalen Anforderungen mit einer kurzen Beschreibung:

Anforderungen	**Beschreibung**
Benutzerverwaltung	Benutzer sollen sich registrieren und anmelden können.
Lobbyverwaltung	Spieler sollen ein Spiel erstellen können, das anderen Spielern in einer Lobby dargestellt wird. Hierüber sollen andere Spieler diesem Spiel beitreten können. Spieler sollen auch in der Lage sein, ein beigetretenes Spiel wieder zu verlassen.
Spiellogik und -Steuerung	Das Spiel soll die Spiellogik und Steuerung korrekt implementieren und die Gewinnbedingungen ausführen können.
Persistenz von Spielerdaten	Userdaten und Spieldaten müssen dauerhaft gespeichert werden.
Peer-to-Peer-Spielverbindungen	Die Anwendung soll Peer-to-Peer-Verbindungen zwischen Spielern herstellen und verwalten können, um die Spieledaten auszutauschen.
3D-Modelle und Animationen	Spieler sollen einen Waschbären steuern können, der über Animationen verfügt und sich auf einer Map befindet, welche ebenfalls mit 3D-Modellen ausgestattet ist.

Eine Liste der nichtfunktionalen Anforderungen:

NF-Anforderung	**Beschreibung**
Verlässlichkeit (Reliablity)	Die Anwendung soll die korrekten Ergebnisse liefern (Nachrichten, Dateien).
Skalierbarkeit (Scalability)	Das Produkt verfügt über eine skalierbare Architektur.
Nutzerfreundlichkeit (Usability)	Mit der Webseite kann man schnell klarkommen und sie einfach bedienen.

Vertraulichkeit (Confidentiality)	Nur der Nutzer selbst soll auf seine User-Daten zugreifen können.
Verfügbarkeit (Availability)	Die Anwendung soll eine möglichst hohe Verfügbarkeit haben.
Kompatibilität (Compatibility)	Die Anwendung ist unabhängig vom Betriebssystem aufrufbar.
Leistung (Performance)	Die Anwendung kann auch mit vermehrten Anfragen umgehen.
Wartbarkeit (Maintainability)	Der Zeitaufwand für Updates und das Deployment ist möglichst gering.

3 Umsetzung

3.1 Implementierung

In diesem Abschnitt wird die Implementierung der zentralen Systemkomponenten näher beschrieben und Alternativen diskutiert. Es umfasst die Implementierung folgender Komponenten: Gameservice, Userservice und den Client.

3.1.1 Gameservice

Der Gameservice wurde als Express-Server implementiert und verwendet Mongoose, um sowohl den Datentyp zu modellieren als auch auf die MongoDB zuzugreifen. Die Komponente ist im Repository im gameservice-Ordner zu finden. Die grundlegende Struktur dieser Komponente ist wie folgt.

Der Nutzer greift über RESTful APIs auf die Endpunkte zu, welche im routes-Ordner definiert sind. Diese rufen die entsprechenden Controller-Funktionen auf, welche im controllers-Ordner zu finden sind. Die Controllerfunktionen wiederum greifen auf entsprechende Servicefunktionen zu, welche sich im services-Ordner befinden. Die Idee dahinter ist, dass nur in den Servicefunktionen Werte- und Schemavalidierungen stattfindet und dort dann entsprechende Exceptions geworfen werden. Lediglich in den Servicefunktionen findet der Datenbankzugriff statt. Die Controllerfunktionen wiederum sollen möglichst schlank gehalten sein und behandeln hauptsächlich das Erstellen der entsprechenden HTTP-Response. Je nachdem, ob die Servicefunktion eine Exception wirft oder nicht, wird eine entsprechende Antwort erstellt. Eine solche Struktur ermöglicht auch eine bessere Testbarkeit, da in den Servicefunktionen alle Grenzfälle behandelt werden.

Gleichzeitig übernimmt der Gameservice auch die Kommunikation mit dem Frontend über Websockets mit Hilfe der Socket.IO-Library. Diese Kommunikation ist notwendig, da sich bei dynamischen Situationen Änderungen ergeben, die der Server dem Client mitteilen muss und er auch direkt Manipulationen an der Datenbank vorzunehmen hat. Etwa wenn ein anderer Spieler in der Lobby dem Spiel beitritt bzw. diese verlässt oder wenn ein Spieler einen Verbindungsabbruch hat und nicht weiter am laufenden Spiel teilnehmen kann. Die Socket.IO-Serverinstanz wurde in socketios.ts im root-Ordner definiert. Die Clientinstanz wurde über ein Singleton-Pattern im Frontend als socketService.ts implementiert.

Abbildung 2 zeigt das Zusammenspiel von Aufrufen der API-Endpunkte und der Kommunikation über Websockets. In einigen Fällen muss der Socket-Server auch direkt Manipulationen an der Datenbank vornehmen. Die Fälle sind folgende:

- Der Host verlässt ein Spiel, das noch nicht gestartet wurde: Der Datenbankeintrag muss gelöscht werden,
- Ein Spieler verlässt ein Spiel, das noch nicht gestartet wurde: Der Datenbankeintrag muss aktualisiert werden,
- Ein Spieler verlässt ein Spiel, das bereits gestartet wurde: Der Datenbankeintrag muss aktualisiert werden.

An dieser Stelle hat sich die Aufteilung zwischen einer Controller- und einer Serviceschicht als hilfreich erwiesen. Anstatt direkt Datenbankmanipulationen im Socket.IO-Server vorzunehmen, mussten lediglich die entsprechenden Servicefunktionen aufgerufen werden.

Generell hat sich die Verwendung von Socket.IO als sehr hilfreich erwiesen, da durch die permanente Websocket-Verbindung zwischen Gameservice und Client schnell und einfach Verbindungsabbrüche detektieren lassen, über welche die restlichen Clients informiert werden können. So kann die Anwendung nun robust mit Verbindungsabbrüchen von Mitspielern vor und während des Spiels umgehen. Über den Server werden auch Gewinne und Verluste an die Mitspieler mitgeteilt.

Im Nachhinein hätte eine alternative Implementierung dieser Funktionalität auch in einer strikteren Trennung von Socket.IO-Serverinstanz und der Logik für die RESTful-API des Gameservices bestehen können. Zwei getrennte Services hätten in einer saubereren und besser skalierbaren Lösung resultiert. Das Ziel der loseren Kopplung wäre besser erreicht worden. Gleichzeitig wäre es in diesem Fall aber auch denkbar gewesen, einen dritten Dienst, der allein die Datenbankmanipulationen vornimmt, zu erstellen und bereitzustellen. Diese Trennung hätte allerdings in einem deutlich erhöhten Entwicklungsaufwand resultiert.

Eine Alternative von Websockets hätte auch in der Verwendung von Message-Oriented-Middleware wie etwa RabbitMQ bestanden. Hierüber ist es für den Server ebenfalls möglich, Nachrichten an die Clients zu versenden und sie über Spielereignisse zu informieren.

Abbildung 2: Sequenzdiagramm der Spielelobby vom Erstellen bis zum Start eines Spiels.

3.1.2 Userservice

Der Userservice wurde als Express-Server implementiert und verwendet Mongoose, um das Datenmodell zu definieren und auf die MongoDB zuzugreifen. Darüber hinaus wurde das Paket "redis" verwendet, um auf die Redis-Datenbank zuzugreifen. Diese Komponente befindet sich im Repository im Ordner "userservice". Die grundlegende Struktur dieser Komponente ist wie folgt:

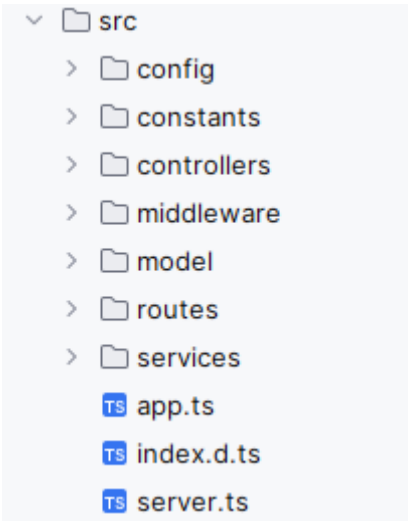

Abbildung 3: Ausschnitt aus dem Ordner "userservice"

Der Client greift über RESTful APIs auf die Endpunkte zu, die im Ordner "routes" definiert sind. Diese Endpunkte rufen die entsprechenden Controller-Funktionen auf, die sich im Ordner "controllers" befinden. Die Controller-Funktionen wiederum greifen auf entsprechende Service-Funktionen zu, die sich im Ordner "services" befinden. Die Idee dahinter ist, dass in den Service-Funktionen Wert- und Schemaüberprüfungen sowie Ausnahmenbehandlungen stattfinden, falls erforderlich. Der eigentliche Datenbankzugriff erfolgt ausschließlich in den Service-Funktionen. Die Controller-Funktionen sind darauf ausgerichtet, die HTTP-Response zu erstellen und möglichst schlank zu halten. Abhängig davon, ob eine Service-Funktion eine Ausnahme auslöst oder nicht, wird die entsprechende Antwort generiert. Diese Struktur erleichtert auch die Testbarkeit, da alle kritischen Überprüfungen in den Service-Funktionen durchgeführt werden.

Die JWT-Validierung wurde als Middleware implementiert, um sicherzustellen, dass eingehende Anfragen ordnungsgemäß authentifiziert werden. Diese Middleware wurde nur für bestimmte Routen gesetzt, die vor dem Erreichen des Controllers überprüft werden müssen. Benutzer können auf die Login- und SignIn-Endpunkte zugreifen, ohne im Besitz des JWT-Tokens zu sein, aber andere Routen sind durch diese Middleware geschützt. Wenn die eingehende Anfrage mit

einem gültigen JWT gesendet wird, leitet der Benutzerdienst sie an den entsprechenden Controller weiter. Andernfalls wird eine Ausnahme ausgelöst und an den Client zurückgesendet.

Abschließend wäre eine Alternative zu dieser Implementierung die Verwendung des Nest.js-Frameworks anstelle des Express.js-Frameworks gewesen. Nest.js bietet viele hilfreiche Schnittstellen zur Validierung und Serialisierung des Request-Objekts, was in Express manuell erfolgt. Die Erstellung von OpenAPI-Dokumentationen wäre mit diesem Framework einfacher. Die Verwendung von Decorators macht den Code außerdem intuitiver und reduziert die Notwendigkeit für repetitive Konfigurationen. Trotz dieser Vorteile wurde Express aufgrund der schnellen Implementierung und des Mangels an Wissen über Nest.js verwendet.

3.1.3 Client

Der Client wurde mit Typescript, PeerJS und Three.js implementiert. Das GameClient Objekt ist zuständig für die Verwaltung des gesamten Clients. Dazu gehören: Die Kommunikation mit anderen Peers, die Handhabung von Benutzereingaben, das Rendern der Three.js Szene, die Manipulation des Spielzustandes.

Jeder GameClient verwaltet einen PeerClient. Der PeerClient ist ein Wrapper für den von PeerJS bereitgestellten Client zur Kommunikation via WebRTC. Die Hauptaufgabe des Wrappers ist es, asynchrone Methoden für die Nutzung des PeerClients zur Verfügung zu stellen. Bei Start des Spiels wird dem GameClient eine Liste mit den Ids der Mitspieler übermittelt. Daraufhin baut der PeerClient des GameClients zu jedem der Mitspieler eine WebRTC-Verbindung auf. Hieraus entsteht eine Full Mesh-Topologie. Bei Spielende werden diese Verbindungen wieder geschlossen. Die Synchronisierung des Spielzustandes erfolgt über Snapshots. In einem festen Intervall sendet der GameClient den Zustand des eigenen Spielers und dessen Geschosse an die anderen Peers. Diese passen bei Erhalt den Zustand des Spielers im lokalen GameState an.

Das Rendern der Szene erfolgt in einer eigenen Renderer Klasse. Diese verwaltet alle zum Rendern benötigten Three.js Objekte und stellt Methoden zur Manipulation der Szene bereit. Sie enthält außerdem die Game Loop. In der Game Loop wird die Szene gerendert, der neue Spielzustand berechnet und synchronisiert. Alle drei Aufgaben sind unabhängig voneinander und die Häufigkeit, mit der jede dieser Aufgaben passiert kann daher angepasst werden.

Die Manipulation des Spielzustandes übernimmt eine primitive Physik-Engine. Diese wrappt die Three.js Szene und durchquert für einen neuen Zyklus den Szenengraph, um den Zustand jedes Objekts, welches das Interface Updateable implementiert zu verändern. Dazu wird beim Durchqueren des Szenengraphs auch Kollisionsdetektion durchgeführt.

4 Reflektion

Eine große Herausforderung war es den Zustand über alle Knoten zu synchronisieren. Dazu wurden zwei verschiedene Prototypen mit zwei in der Spieleindustrie häufig verwendeten Protokollen implementiert: “Deterministic Lockstep” und “Snapshot Interpolation”. “Deterministic Lockstep” erwies sich als zu komplex in der Implementierung, sodass der Prototyp nie in einen lauffähigen Zustand war. Die Synchronisierung wurde also mit “Snapshot Interpolation” umgesetzt. Die Methode wurde allerdings auch nur zu einem geringen Grad umgesetzt. So fehlt die Interpolation zwischen zwei Snapshots in der Implementierung. Das Thema war sehr komplex und konnte daher im Rahmen des Projekts nur knapp betrachtet werden. Ein Learning ist daher, für relevante Praxisprojekte existierende GameEngines zu verwenden, da diese bereits Mechanismen zur Synchronisation des Spielzustandes beinhalten. Für das Projekt war es jedoch sehr interessant sich einmal näher mit diesem Thema auseinanderzusetzen.

Eine Anforderung an das Projekt war es unfaires Verhalten (cheaten) zu verhindern. Dazu wurden für mehrere Szenarien Lösungsstrategien entworfen. Um zu verhindern, dass ein Peer gefälschte Snapshots sendet, sollten die Snapshots anderer Spieler validiert werden. Außerdem sollte verhindert werden, dass ein Spieler Snapshots mutwillig zurückhält, um diese dann dicht hintereinander zu senden (lag machine). Dies könnte durch Mehrheitsentscheidung bei Treffern gelöst werden. Ein Spieler hätte in diesem Fall einen Nachteil seine eigenen Snapshots zurückzuhalten, da die Bewegung seines Avatars mit Interpolation sehr vorhersehbar wäre. Ein Spieler könnte außerdem seine eigenen Pakete zurückhalten um “in die Zukunft zu sehen” (look ahead cheat). Dieser Cheat ist allerding irrelevant, da man beim Spielprinzip von Racoosh dadurch keinen relevanten Vorteil gewinnt. Von all diesen Abwehrstrategien wurde aus Zeitgründen keine tatsächlich implementiert.

Viele der Probleme, die während des Projekts auftraten, wären in einer Client-Server Architektur deutlich weniger Komplex gewesen. Außerdem war das Abdecken von Grenzszenarien, welche durch die P2P-Architektur auftraten, zeitaufwendig und die Grenzfälle waren als solche nicht immer auf Anhieb erkennbar. Oft war es schwierig diese Grenzfälle zu antizipieren und erst nach einer längeren Implementierung traten neue Probleme auf. Im Rahmen des Projektes war es jedoch sehr spannend einen hybriden Ansatz aus P2P und Client-Server zu verwenden.