GamyTim Dokumentation

Verteilte-Systeme-Projekt

des Studiengangs

Allgemeine Informatik

an der Dualen Hochschule Baden-Württemberg, Campus Heidenheim

Diese Dokumentation entstand im Rahmen des Verteilte-Systeme-Projekts an der Dualen Hochschule Baden-Württemberg. Sie beschreibt die Konzeption und Umsetzung von GamyTim, einem verteilten System zur sichergestellten Vergnügung, dessen Ziel das hochverfügbare und sichere leaderboard-climben auf der GamyTim-Seite ist.

Bearbeitungszeitraum	10 Wochen
Kurs	INF2023AI
Gruppenmitglieder	Aaron Reiber, Moritz Flaig

Inhaltsverzeichnis

1 Einleitung

Im Rahmen des Moduls „Verteilte Systeme“ haben wir gemeinsam ein Netzwerkspiel auf Basis des Klassikers „Achtung, Kurve!“ entwickelt. Ziel des Projekts war es, die Prinzipien verteilter Systeme praktisch umzusetzen und die Interaktion mehrerer Clients über einen zentralen Server zu ermöglichen.

1.1 Projektüberblick

Die Umsetzung erfolgte in Godot als Game Engine, ergänzt durch verschiedene Backend-Komponenten zur Verwaltung von Spielern, Lobbys und Spielzuständen. Dabei kamen unter anderem eine User-Datenbank, ein Redis-Server für schnelle Synchronisation, sowie ein Masterserver und ein Loadbalancer für die Organisation und Skalierung der Spielinstanzen zum Einsatz.

Die Architektur wurde so entworfen, dass sie eine saubere Trennung zwischen Frontend, Spiel-Logik und Backend-Systemen ermöglicht. Eine genauere Beschreibung der einzelnen Komponenten und deren Zusammenspiel erfolgt in den folgenden Kapiteln.

1.2 Ziele und Motivation

Dabei stand nicht nur der Spielspaß im Vordergrund, sondern insbesondere das praktische Verständnis von Konzepten wie: Lastverteilung durch einen Loadbalancer, Zentrale Verwaltung von Lobbys und Spielzuständen über einen Masterserver, Persistente Datenspeicherung in Datenbanken, Skalierbarkeit durch die Nutzung mehrerer Game-Server und Redis für schnelle Synchronisation.

Die Motivation war also, ein praxisnahes Beispiel zu schaffen, bei dem die theoretischen Inhalte aus der Vorlesung in einem realen Projekt erlebbar werden. Durch die Umsetzung in Godot konnte zusätzlich die Anbindung einer Game Engine an ein verteiltes Backend erprobt werden.

2 Architektur

In diesem Kapitel wird die Architektur des Projekts beschrieben.

2.1 Überblick

Die Gesamtarchitektur gliedert sich in mehrere Teilsysteme, die zusammenarbeiten, um ein stabiles Mehrspieler-Erlebnis zu gewährleisten. Das Spiel selbst läuft in Godot-Instanzen (Game-Server), die durch zentrale Backend-Komponenten verwaltet werden. Die Kommunikation erfolgt über klar definierte Schnittstellen zwischen Frontend, Loadbalancer, Masterserver, Datenbanken und Game-Servern. Ein schematischer Überblick ist in Abbildung 2.1 dargestellt.

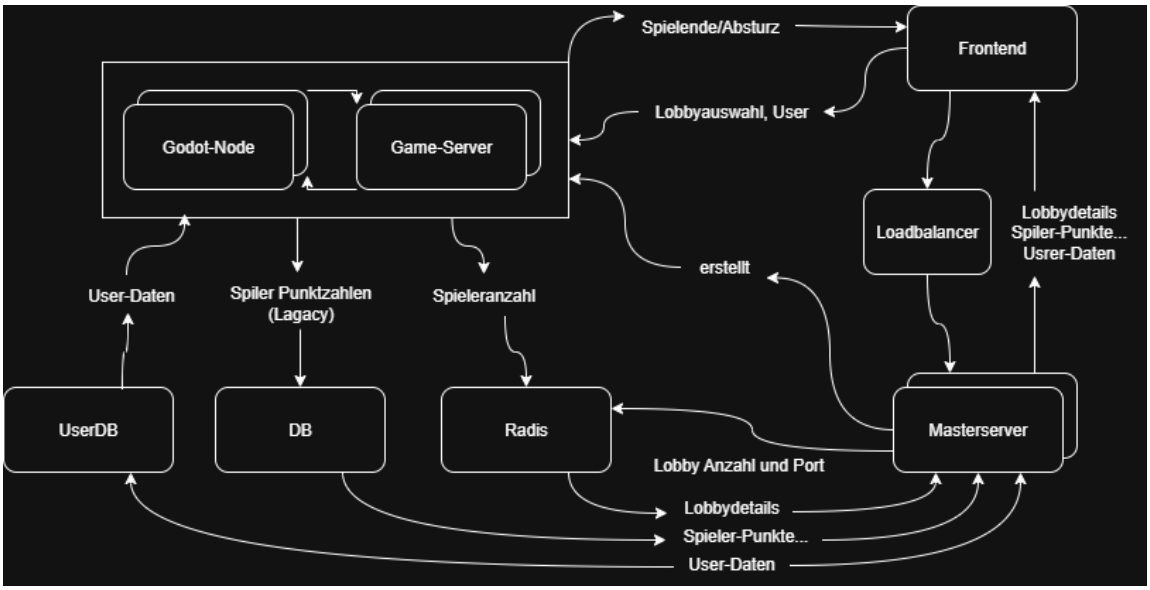

Abbildung 2.1: Architekturübersicht des Projekts

Bei der Planung der Architektur wurde stets darauf geachtet für den Benutzer die bestmögliche Erfahrung zu schaffen. Hierfür wurde sich für WebSockets entschieden,

um es dem Spieler zu ermöglichebn einfach im Browser zu spielen, ohne zusätzliche Software installieren zu müssen. WebSockets bieten eine durchgehende Verbindung, bei der Client und Server jederzeit Daten senden können, was bei Echtzeit-Anwendungen wie Spielen die Verzögerung deutlich reduziert. Vgl. hierzu auch: (Alex Booker: *WebSockets vs HTTP: Which to choose for your project in 2024*, Ably, 13 min read https://ably.com/topic/websockets-vs-http (letzter Zugriff: 17.09.2025))

2.2 Frontend

Das Frontend stellt die Schnittstelle zum Spieler dar. Hier können Lobbys ausgewählt, Benutzer eingeloggt und Spieldetails angezeigt werden. Nach Spielende oder im Falle eines Absturzes liefert das Frontend entsprechendes Feedback an den Benutzer. Dieses wurde in Angular umgesetzt und kommuniziert über HTTP und WebSockets mit dem Backend. Angular wurde ausgewählt, da es eine moderne, komponentenbasierte Struktur bietet und die Entwicklung von Single-Page-Applications (SPAs) erleichtert. Außerdem war Angular bereits aus vorherigen Projekten bekannt, was die Einarbeitungszeit verkürzte. Vgl. hierzu auch: Sparkout Tech Solutions: *Angular: Modern Web Application Development*, Medium, 28. Oktober 2023. URL: https://sparkouttechsolutions.medium.com/angular-modern-web-application-development-ac2b3acdc878 (letzter Zugriff: 17.09.2025)

2.3 Loadbalancer

Der Loadbalancer verteilt eingehende Anfragen auf die verfügbaren Game-Server. Er sorgt dafür, dass neue Lobbys erstellt werden können und die Last gleichmäßig über mehrere Server verteilt wird. Dadurch wird die Skalierbarkeit des Systems gewährleistet. Dieser wurde mit Nginx realisiert, da es eine bewährte Lösung für Lastverteilung darstellt. Es können beim Start der Container beliebig viele Masterserver-Instanzen angegeben werden. Dies geschieht unter dem Docker-Compose-Flag –scale masterserver=Anzahl.

2.4 Masterserver

Der Masterserver (in Express) übernimmt die zentrale Verwaltung aller Lobbys und handhabt Informationen zu:

- Anzahl und Ports der Lobbys,
- Spielern und deren Punkteständen,
- Metadaten wie Spielstatus und Benutzerinformationen.

Diese Daten werden sowohl an das Frontend als auch an die Game-Server weitergegeben.

2.5 Game-Server und Godot

Die eigentliche Spiellogik läuft in den Godot nodes, welche dann als HTML5 exportiert werden. Das exportierte Spiel wird dann über die Game-Server dynamisch für den Client bereitgestellt. Jeder Game-Server repräsentiert eine Lobby und übernimmt die Synchronisation der Spielzustände zwischen den Spielern. Die Game-Server kommunizieren kontinuierlich über einen Redis-Cache mit dem Masterserver, um Daten wie Spieleranzahlen und Statusmeldungen zu übermitteln. Die Punktestände sowie Userdaten werden in einer seperaten relationalen Datenbank gespeichert.

2.6 Datenbanken

Zur Verwaltung werden zwei Datenbanken eingesetzt:

- **UserDB**: Speicherung von Benutzerinformationen wie Username und gehashtem Passwort in einer PostgreSQL-Datenbank.
- **LeaderboardDB**: Speicherung von Spielerpunktzahlen und gespielten Spielen in einer PostgreSQL-Datenbank.
- **Redis**: Speicherung von Lobbydetails und temporären Statusdaten für schnelle Zugriffe und Synchronisation zwischen Masterserver und Game-Servern.

2.7 Zusammenspiel

Das Zusammenspiel der Komponenten erfolgt in folgenden Schritten:

1. Ein Spieler wählt im Frontend eine Lobby aus oder erstellt eine neue.
2. Der Loadbalancer leitet die Anfrage an den Masterserver weiter.
3. Der Masterserver erstellt eine Lobby auf einem Game-Server und verwaltet alle relevanten Metadaten.
4. Während des Spiels werden kontinuierlich Statusdaten (z. B. Punktestände, Spieleranzahlen) zwischen Game-Server, Redis und Masterserver ausgetauscht.
5. Nach Spielende oder einem Absturz werden die Daten an das Frontend zurückgemeldet und gespeichert.

3 Reflektion

3.1 Herausforderungen und Lösungen

Synchronisation der Kollisionen

Problem: Bei zwei offenen Tabs waren Crashes nicht synchron (bei einem Spieler „knapp vorbei“, beim anderen „voll drin“).
Ursache: Jeder Client prüfte lokal Kollisionen/Out-of-Bounds und hatte leicht unterschiedliche Arena-Bounds, FPS, RNG-Sequenzen.
Lösung: Nur der Host prüft Kollisionen auf seinem Spiel und beendet die Runde, bei einer Kollision.
Clients bekommen das Ergebnis vom Host mitgeteilt und beenden ihre Runde, egal ob sie lokal eine Kollision hatten oder nicht.
Anmerkung: Kann zwar zu Asynchronität führen, ist im Rahmen des Projekts aber vertretbar (Selbst große Spiele haben diese Synchronisationsprobleme).

Spieleranzahl-Übertragung

Problem: Spieleranzahl wurde nicht im Frontend oder nur beim festgelegten "DevServer angezeigt.
Ursache: Der GameServer wusste nicht auf welchem Port er läuft und konnte somit nicht die Spieleranzahl an den Masterserver übertragen. Somit hat er die Spielerzahl immer nur für den festgelegten Port des "DevServers übertragen.
Lösung: Der wichtigste Aspekt war die Differenzierung zwischen Internal- und Public-Port. Intern kann immer auf dem "DevPort (standard) kommuniziert werden der

Public-Port musste jedoch vom Masterserver identifiziert und für die spielerzahl festegelegt werden.

Username-Übergabe an Godot

Problem: Der Username wird im Frontend erstellt und in der Datenbank gespeichert, jedoch nicht direkt an Godot übergeben.
Ursache: Godot hat keinen direkten Zugriff auf die Datenbank und fragt bei der Authentifizierung mit dem Token nur den Login-Service ab.
Lösung: Der Username wird in der URL (so wie der Auth-Token) als Query-Parameter übergeben und von Godot ausgelesen.
Anmerkung: Dabei war das ganze vorerst nicht trivial. Das Auslesen und Weiterverarbeiten von Query-Parametern in der Engine musste zunächst manuell implementiert werden.

3.2 Mögliche Alternativen

Synchronisation der Kollisionen

Anstatt ein großes Spielfeld mit äuf Pixeln"basierter Kollisionsprüfung zu verwenden, könnte man das Spielfeld in ein Raster aufteilen und die Positionen der Spieler auf die Rasterzellen beschränken. -> Räumliche Indexierung (z. B. Grid/Quadtree) für die Trails und Köpfe der einzelnen Spieler.

Spawnpositionen

Derzeit werden die Spawnpositionen der Spieler zufällig gewählt, was zu unfairen Startbedingungen führen kann (z. B. wenn Spieler zu nah beieinander spawnen oder sehr nah am Rand). Dies war jedoch eine bewusste Designentscheidung, um das Spiel dynamisch und spannend zu gestalten. Als Alternative wären auch vordefinierte Spawnpunkte denkbar, die strategisch über das Spielfeld verteilt sind, welche zu einem faireren Start führen würden, auf Kosten der Spannung.

3.3 Lessons Learned

Login mit PostgreSQL und JWT

Die Implementierung eines des Logins mit PostgreSQL und JWT war mit erstaunlich wenig Einarbeitungszeit verbunden und hat sich als sehr effektiv erwiesen. Die Gruppenmitglieder hatten zuvor hiermit noch nicht gearbeitet, sind aber positiv überrascht und werden diese Erkenntnis in zukünftigen Projekten anwenden.

Datencache mit Redis

Auch hiermit hatten die Gruppenmitglieder zuvor noch nicht gearbeitet und waren positiv überrascht, wie einfach und effektiv die Implementierung eines Datencaches mit Redis war.

Godot Engine

Die Godot Engine war für alle Gruppenmitglieder neu und die Einarbeitung hat verhältnismäsig viel Zeit (im Vergleich zu anderen neu-erlernten Technologien) in Anspruch genommen. Auch die Arbeit nach der Einarbeitung war teilweise etwas umständlich und ging eher schleppend voran. Die Komplexität und Größe des Codes und Architektur in Godot wurde zu Beginn stark unterschätzt.

4 UI (optionale Erläuterung)

4.1 Frontend-Lobby

Die *Lobby* bildet den Einstiegspunkt für die Spieler und ist der zentrale Bestandteil des Frontends. Sie wurde mit dem Framework **Angular** umgesetzt und mit **Bootstrap** für ein modernes, responsives Design gestaltet. Die in Abbildung 4.1 dargestellte Oberfläche ermöglicht den Login, die Serverauswahl sowie die Anzeige von Spielstatistiken.

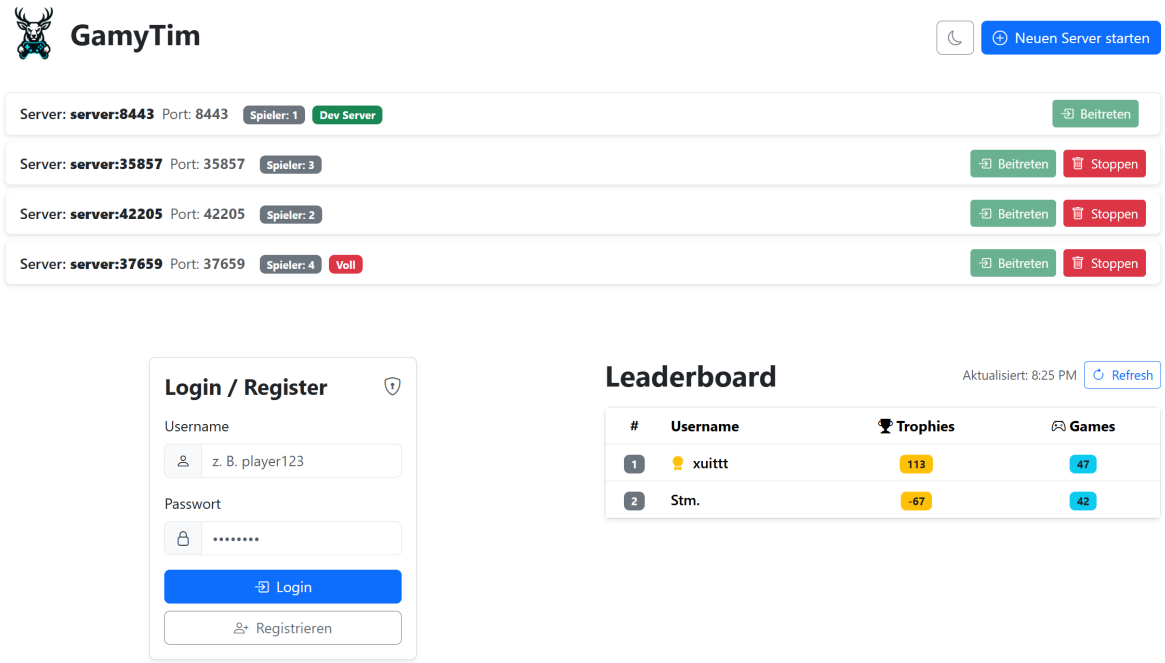

Abbildung 4.1: Frontend-Lobby des Projekts

4.2 Funktionen der Lobby

Die Frontend-Lobby umfasst mehrere Kernfunktionen, die jeweils mit Backend-Komponenten verknüpft sind:

4.2.1 Login / Registrierung

Über das Login-Formular können sich Benutzer mit einem bestehenden Account anmelden oder einen neuen Benutzer registrieren. Die Eingaben werden an die **UserDB** weitergeleitet, wo Validierung und Speicherung der Daten erfolgen. Nach erfolgreicher Anmeldung stehen dem Benutzer alle Funktionen der Lobby zur Verfügung.

4.2.2 Token basierte Authentifizierung

* Im Projekt wird die Authentifizierung über JWT umgesetzt. Benutzer registrieren sich oder loggen sich ein, Passwörter werden mit bcrypt gehasht gespeichert. Nach erfolgreichem Login erhält der Client ein Token, das bei allen Anfragen im Header mitgeschickt und vom Backend validiert wird. Dadurch wird die Integrität der Benutzer-, Spiel- und Trophäendaten gewährleistet. Um sich im GodotSpiel zu authentifizieren, wird das Token im URL-Parameter `token` übergeben, wodurch sicher gestellt wird, dass in Godot immer der Name sowie die Authorität des Spieler bekannt sind. Vgl. hierzu auch: Fettke, Peter; Vogel-Heuser, Birgit (Hrsg.): *Digitale Authentifizierung und Autorisierung*, Springer Vieweg, 2021. Jones, M. et al.: *JSON Web Token (JWT)*, IETF RFC 7519,May 2015. `https://datatracker.ietf.org/doc/html/rfc7519` (lezter Zugriff: 17.09.2025)

4.2.3 Serverübersicht

Im oberen Bereich wird die aktuelle Serververbindung angezeigt. Dabei sind `Servername`, `Port` sowie die Anzahl der verbundenen Spieler sichtbar. Die Daten stammen vom **Masterserver**, der kontinuierlich den Status aller verfügbaren Lobbys aktualisiert.

4.2.4 Serververwaltung

Benutzer können neue Serverinstanzen über den Button „Neuen Server starten“ anlegen und ggf. bestehende Server über „Stoppen“ wieder beenden. Diese Anfrage wird an den **Loadbalancer** übermittelt, welcher die Instanz erstellt und die notwendigen Metadaten (z. B. Port, Lobby-ID) an den Masterserver weitergibt.

4.2.5 Lobbybeitritt

Mit der Funktion „Beitreten“ können Spieler einer vorhandenen Lobby beitreten. Hierbei wird die Verbindung zu einem **Game-Server** hergestellt, der die eigentliche Spiellogik ausführt.

4.2.6 Leaderboard

Das Leaderboard zeigt die globalen Punktestände und Spielstatistiken aller Spieler an. Die Daten werden aus der Datenbank geladen und im Frontend aktualisiert. Dargestellt werden unter anderem:

- Benutzername,
- Anzahl der Trophäen,
- Anzahl gespielter Spiele.
- Leader

4.2.7 Datenintegritäts-Sicherung

* Im Backend wird über eine Validierung der einkommenden Trophäen- und Gamedaten die Integrität der Daten im Rahmen des Projektes angegangen.

4.3 Godot-Spiel

Das Spiel folgt dem klassischen Prinzip von *Achtung, Kurve!*: Jeder Spieler steuert eine farbige Linie, die sich kontinuierlich über das Spielfeld bewegt. Ziel ist es, möglichst lange zu überleben, indem man nicht mit den Linien anderer Spieler oder den Spielfeldrändern kollidiert. Die Steuerung erfolgt ausschließlich über Richtungsänderungen (links/rechts).

4.4 Spieloberfläche

Die Spieloberfläche zeigt die farbigen Linien der aktiven Spieler sowie deren aktuelle Positionen (Abbildung 4.2). Am oberen Rand werden die Punktestände der Spieler angezeigt, welche bei Kollisionen entsprechend angepasst werden.

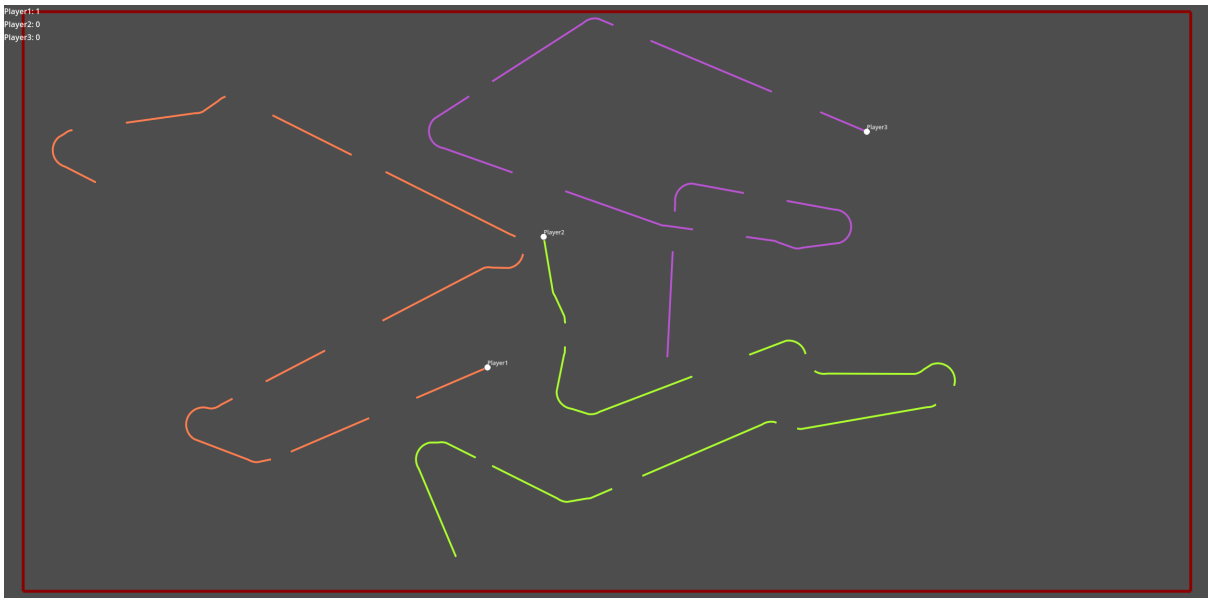

Abbildung 4.2: Spielfeld in Godot mit drei aktiven Spielern

4.5 Lobby- und Ready-System

Vor Spielbeginn wird eine Lobby-Ansicht geladen, in der sich alle Spieler sammeln. Hierbei übernimmt der erste Spieler die Rolle des **Hosts**. Alle weiteren Spieler treten über die Lobby bei. Jeder Spieler muss seinen Status auf *Ready* setzen, bevor das Spiel starten kann. Die Synchronisation erfolgt über den Game-Server, der den Status aller Spieler an die Lobby zurückmeldet (Abbildung 4.3).

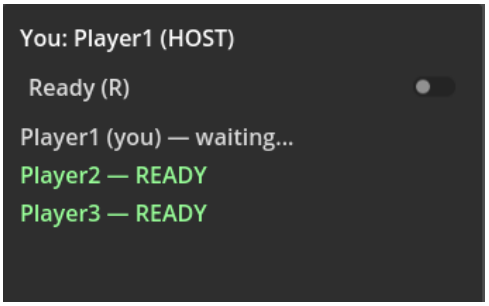

Abbildung 4.3: Ready-System: alle Spieler müssen bereit sein, bevor das Spiel startet

Vgl. hierzu auch: Godot Engine Documentation: *High-level multiplayer* und *Networking fundamentals*, `https://docs.godotengine.org/en/stable/tutorials/networking/high_level_multiplayer.html` (lezter Zugriff: 17.09.2025).

4.6 Synchronisation und Kommunikation

Die Game-Server sind für die Synchronisation der Spiellogik verantwortlich. Dabei werden folgende Informationen zwischen den Instanzen ausgetauscht:

- Spielerpositionen und Bewegungsrichtungen,
- Statusänderungen (z. B. Ready/Not Ready, Spielstart, Spielende),
- Kollisionsereignisse und daraus resultierende Punktestände.

Der zuvor bei Lobby-Beitritt festgelegte Host übernimmt die Rolle des **Master-Clients** und koordiniert die Spielabläufe. D.h. wenn eine Kollision durch abweichende Spielabläufe auf Clientseite fälschlicherweise stattfinden, wird diese nicht erkannt. Nur Kollisionsereignisse, die der Master-Client erkennt, werden an alle Clients weitergegeben und somit erkannt. Dies führt zu einem geregelten Spielablauf. Ein solches

Verhalten (d.h. dass nur der Master-Client Kollisionsereignisse erkennt und propagiert) entspricht bekannten Entwurfsmustern in Multiplayer-Spielen, in denen eine autoritative Instanz (Server oder Host) nötig ist, um Inkonsistenzen durch etwaige Abweichungen in Client Simulationen zu vermeiden (vgl. z. B. Liljekvist, Detecting Synchronisation Problems in Networked Lockstep Games, KTH 2016; oder Artikel zu Netzwerk-Synchronisation in Multiplayer Games (lezter Zugriff: 17.09.2025)).

Literaturverzeichnis