Hochschule Karlsruhe
University of Applied Science

Fakultät für Informatik und Wirtschaftsinformatik
Wirtschaftsinformatik

BACHELORTHESIS

Netcode in Unity

Von	Alexander Seitz
Matrikelnr.	77642
Arbeitsplatz	Hochschule Karlsruhe
Erstbetreuer	Prof. Dr. Udo Müller
Zweitbetreuer	Prof. Dr. Jan Stöß
Abgabetermin	31.08.2025

Karslruhe, DATUM

Netcode in Unity

Alexander Seitz

June 27, 2025

Contents

Chapter 1

Einleitung

Moderne Echtzeitanwendungen, insbesondere im Bereich der Computerspiele, stellen hohe Anforderungen an die zugrunde liegende Netzwerkarchitektur. Die Synchronisation von Spielzuständen, die Minimierung von Latenzzeiten sowie ein robuster Umgang mit Paketverlust und Jitter sind entscheidend für eine positive Nutzererfahrung. Multiplayer-Systeme, wie sie etwa mit Unity entwickelt werden, müssen deshalb nicht nur funktional, sondern auch leistungsfähig und fehlertolerant sein.

Die vorliegende Arbeit beschäftigt sich mit den theoretischen und technischen Grundlagen der Netzwerkkommunikation in Mehrspielerumgebungen. Anhand eines prototypischen Projekts in Unity werden exemplarisch zentrale Mechanismen wie Reconciliation, Interpolation sowie die Kommunikation im Client-Server-Modell umgesetzt und analysiert.

Im Folgenden werden zunächst die Motivation und Zielsetzung der Arbeit erläutert. Anschließend wird der strukturelle Aufbau der Arbeit vorgestellt.

1.1 Motivation

In den meisten modernen Spielen ist oft ein Teil des Spiels als Mehrspieler ein kooperativer Teil des Spiels integriert. Diese Bereiche reichen von strategischen FPS, Survival-Spielen bis zu Rennspielen. In all diesen Spielen wird von den Spielern ein hoher Wert auf Genauigkeit und Funktionalität des Netcode-Systems verlangt. Diese Integration bringt oft sehr viele Herausforderungen, in manchen Fällen ist auch eine komplette Eigenimplementierung auf Grund der Komplexität von der Basis des Spiels nicht abzudenken. Viele Entwickler vor allem Indie-Entwickler, stellen wenn überhaupt einen Mehrspieler erst einmal

hinten an, da eine ausgereifte Integration vor allem für Unerfahrene eine große Herausforderung werden kann.

Im Bereich der AAA-Titel jedoch, kommt es immer häufiger zu Kritikäußerungen von Medien als auch Spielern. In vielen Fällen wird sich auf die kostensparenden Ansätze der Entwickler beziehungsweise der Publisher der Spiele bezogen, die sich für billigere und somit leistungsärmere Alternativen wie bspw. Server mit niedrigeren Tickraten entscheiden. In den letzten Jahren kam jedoch eine der kuriosesten Neuerungen in den Mainstream der strategischen FPS, nämlich sollte das Counter Strike 2 – der direkte Nachfolger von CS:GO – auf Grundlage eines Subtick-Systems implementiert werden, anstatt das altbewährte klassische Ticksystem zu verbessern und modernisieren. Das System existierte schon vor der Integration des beliebten FPS, dennoch wurde es nie konsequent oder erfolgreich integriert. Subtick ist im Grunde ein eigenes Ticksystem zwischen Ticks, demnach ist jeder Input mit einem konkreten Timestamp versehen und soll rein theoretisch, zu genaueren und direkterem Spielgefühl führen. Beim Klassischen System werden Eingaben zwischen Ticks an den nächsten Tick delegiert, was Synchronisation zwischen Clients oft erschwert bzw. oft zu uneinheitlichen Erfahrungen fürht.

Demnach liegt die Frage nahe, ob es sich wirklich um ein Problem beim Entwickler handelt oder ein Problem, von maßloser Komplexität, die von so vielen Variablen abhängt, dass sie in keinem Fall zu perfektionieren ist. (Das könnte man auf jeden Fall ändern, war nur das erste was mir eingefallen ist).

1.2 Zielsetzung

In dieser Arbeit sollen Netcode und darauf aufbauende Netcode Mechanismen auf theoretischer und praktischer Grundlage beleuchtet werden und in einem prototypischen Spiel integriert werden. Da die reine Umsetzung keinen großen akademischen Mehrwert bietet, sollen Strategien entworfen werden, die zur Visualisierung und damit hoffentlich der Verbesserung des Verständnisses der Mechaniken dienen. Es sollen des weiteren verschiedene Szenarien untersucht werden, die auch von Idealbedingungen abweichen können.

Chapter 2

Auswahl der Engine

Um die bestmöglichste und vor allem effizienteste Entwicklung des Spiels und der geplanten Netcode Mechanismen zu integrieren, ist also eine durchdachte Wahl der Spiel Engine extrem wichtig. Da vor allem Zeit ein Engpass in einem ambitionierten Projekt, wäre es fatal, wenn auf Grund der Wahl der Engine kein Fortschritt gemacht werden kann. Die wichtigsten Kriterien waren:

2.1 Lernkurve

Eine der wichtigeren Aspekte bezieht sich auf die Lernkurve der Game Engine für den Entwickler. Wenn also eine Engine mit so viel Lernaufwand in Verbindung steht um überhaupt eine triviale Szene zum Laufen zu bekommen, dann ist diese wohlmöglich nicht die richtige Engine. Wenn jedoch gewisse Vorkenntnisse bestehen im Bereich Spieleentwicklung, dann wird ein

2.2 Community und Ressourcen

Die Zugänglichkeit von Dokumentation, Foren und Tutorials ist in diesem Kontext extrem wertvoll, da vor allem die Kombination daraus ein tiefes Verständnis einbringen kann. Wenn also die Dokumentation nicht verständlich sein sollte oder es nicht genug Anwendungsbeispiele im Internet geben sollte, kann es den Lernaufwand zusätzlich erhöhen oder zu Missverständnissen kommen.

2.3 Netcode-Fähigkeiten

Wenn also "out of the box" Features gut integriert und dokumentiert sind, die die Weiterentwicklung geplanter Netcode-Mechaniken erleichtern, ist dies der Idealzustand. Dennoch ist für diese Arbeit eine Abgrenzung zu machen, da die Eigenimplementierung im Mittelpunkt steht. Wenn also Mechanismen von der Engine bereits zum Großteil gestellt sind und der Quellcode hiervon nicht anpassbar ist, wirkt sich dies eher negativ auf die Bewertung aus.

2.4 Performance

Wie gut das Spiel im Einsatz von mehreren Clients läuft und wie viele Einbuße in diesem Fall der Multiplayer-Aspekt mit sich bringt. Sollte dies jedoch garnicht mehr möglich sein, dass ein Editor und ein oder mehrere Clients verbunden sind - was im Grunde das Laufen von drei Spielinstanzen darstellt - nicht funktionieren sollte, weil keine Ressourcen mehr auf dem Computer übrig bleiben, ist dies wohl keine geeignete Engine oder man muss optimieren. Allerdings kann eine graphische Grundvoraussetzung zum Problem werden. Wenn also die Engine durch die Natur der Renderpipeline zu sehr die GPU belastet, kann das schon das Prototyping beeinflussen, da in diesem Use Case eine hohe Framerate bevorzug ist.

2.5 Kosten und Lizenzmodell

Viele der bekannten Engines sind nicht Open Source sondern erwarten i.d.R. eine jährliche Zahlung oder eine Art Pacht . In manchen Fällen ist eine jährliche Hochschullizenz möglich, dennoch fällt das somit immernoch unter die Kategrie der Zahlungspflicht. Wenn eine Engine somit einen hohen monetären Einsatz erfordert ohne der Möglichkeit auch nur auf eine bspw. Testversion zuzugreifen, müssen die anderen Aspekte sehr viel gutmachen.

2.6 Programmiersprache

In vielen Fällen steht für den Entwickler ein hoher Lernaufwand mit einer noch zu erlernenden Programmiersprache, bei der neue Aspekte in den Mittelpunkt gerückt werden, die u.U. noch nicht zuvor behandelt wurden. In den meisten Fällen sind dies eher hardwarenähere Programmiersprachen wie bspw. C++. Hierbei müssen neue Konzepte wie der Umgang mit Memory erlernt und gemeistert werden, was im Umgang mit neueren Programmiersprachen wie C# schon bereits gestellt ist. Im Grunde stellt das nicht nur ein Risiko dar, sondern ist auf der anderen Seite auch eine Chance zu lernen, wie man richtig Spiele entwickelt und vor allem auch Quellcode optimieren kann. Nicht ohne Grund bauen die meisten Spiele-Engines auf C++ auf und werden in der Zukunft auch eher nicht ersetzt.

2.7 Marktrelevanz

Die Marktrelevanz ist wie in Programmiersprache schon angesprochen, stark abhängig von der in der Engine genutzten Programmiersprache. Abgesehen davon gibt es unterschiedliche Einsatzbereiche verschiedener Engines und somit lässt sich die Marktrelevanz nicht nach absoluten Zahlen der Nutzter und oder bestimmter Spiele klären. Dennoch muss man die Relevanz der Engine im Auge behalten, wenn es also kaum Spiele auf dem Markt gibt, die mit einer bestimmten Engine entwickelt wurden, dann ist die potenziell investierte Zeit in einer beliebteren Engine wahrscheinlich angebrachter.

Ein direkter Vergleich der bekanntesten Engines zeigt folgende Eigenschaften:

TABLE 2.1: Vergleich populärer Spiel-Engines

Kriterium	**Unity**	**Unreal Engine**	**Godot**
Programmiersprache	C#	C++ / Blueprints	GDScript / C#
Lernkurve	Mittel	Hoch	Niedrig
Community	Sehr groß	Groß	Wächst schnell
Netcode	NGO, FishNet u.a.	Eigenes Replication-System	Drittanbieter, rudimentär
Performance	Gut (abhängig von Optimierung)	Sehr hoch	Mittel
Lizenz / Kosten	Kostenlos (eingeschränkt), Runtime Fee seit 2024 [1]	Royalty-basiert [2]	Open Source [3]
Verbreitung in Industrie	Sehr hoch	Hoch	Gering

Basierend auf den genannten Faktoren fiel die Wahl auf **Unity**, da diese Engine eine ausgewogene Kombination aus Zugänglichkeit, Flexibilität und Netcode-Erweiterbarkeit bietet. Besonders die Integration von C# als Programmiersprache, die umfangreiche Dokumentation sowie die breite Community-Unterstützung waren ausschlaggebend.

Unreal Engine hingegen spielt vor allem in der Entwicklung moderner, grafisch anspruchsvoller Spiele eine zentrale Rolle. Während Unity und Godot insbesondere im Indie-Bereich weit verbreitet sind, wird Unreal Engine regelmäßig für sogenannte AAA-Titel eingesetzt – etwa bei *Fortnite*, *Valorant* oder der *Gears of War*-Reihe. In Addition kommt noch eines der am meist antizipiertesten Spiele überhaupt dazu nämlich The Witcher 4 [4].

Ein besonderes Beispiel für die Weiterentwicklung der Engine durch Eigengebrauch ist das Prinzip des „Dogfooding“. Epic Games nutzt die Unreal Engine intern intensiv zur Entwicklung eigener Spiele wie *Fortnite*, wodurch neue Features – etwa das Partikelsystem *Niagara* – direkt unter realen Produktionsbedingungen erprobt und optimiert werden [5].

Insbesondere mit der Einführung von Unreal Engine 5 (UE5) hat sich die Engine als Standardlösung für kleinere bis mittlere Studios etabliert, die auf fotorealistische Darstellung und moderne Grafiktechnologien wie *Lumen* oder *Path Tracing* setzen. Unity kann in diesen Bereichen zwar nicht vollständig mithalten, bietet jedoch im Kontext dieser Arbeit – etwa bei der prototypischen Umsetzung netzwerkbasierter Mechaniken – eine effizientere und zugänglichere Plattform. Beispiele wie *BattleBit Remastered* (2023) jedoch zeigen, dass auch technisch reduzierte, aber netzwerkseitig ausgereifte Multiplayer-Titel mit Unity erfolgreich realisiert werden können.

2.8 Eigenimplementierung und Auswahl des Frameworks

Im Bereich Netcode existieren für Unity verschiedene Frameworks. Zunächst wurde Unitys offizielles **Netcode for GameObjects (NGO)** in Betracht gezogen. In der praktischen Erprobung zeigten sich jedoch Schwächen im zugrunde liegenden *Tickmodell*. NGO nutzt ein festes Zeitraster zur Synchronisation zwischen Server und Clients, wobei alle Netzwerkaktionen strikt an die sogenannte *NetworkTick*-Rate gebunden sind.

Dieses Modell ist zwar grundsätzlich stabil, führt jedoch bei Spielen mit hoher Eingabefrequenz oder schnellen Bewegungsabläufen zu Einschränkungen: Eingaben, die zwischen zwei Ticks liegen, werden verzögert verarbeitet, was sich in Form von spürbarer Latenz

oder schwankender Reaktionsgeschwindigkeit äußern kann. Besonders bei deterministisch kritischen Anwendungen erschwert dies eine präzise Umsetzung von Mechaniken wie Client-Side Prediction oder Server Reconciliation.

Als Alternative wurde das Framework **FishNet** evaluiert, das in der offiziellen Dokumentation ausdrücklich für Anwendungsfälle mit hohen Anforderungen an Präzision und deterministische Logik empfohlen wird [6]. FishNet nutzt ein eigenes Tick-basiertes Netzwerkmodell und bietet unter anderem:

- Vollständige Kontrolle über Netzwerklogik
- Support für Host-Modus, Dedicated Server und Peer-to-Peer
- Integration mit Prediction, Interpolation und Authentifizierung

Trotz dieser technischen Vorteile wurde für diese Arbeit letztlich **NGO verwendet**. Hauptgrund war zum einen der geringere Integrationsaufwand und die direkte Unterstützung durch Unity. Eine vollständige Migration auf FishNet hätte umfangreiche Anpassungen erfordert, die im gegebenen Zeitrahmen nicht realisierbar gewesen wären.

Zum anderen liegt der Schwerpunkt dieser Arbeit explizit auf dem **Verständnis und der eigenständigen Implementierung** zentraler Netcode-Mechaniken wie *Client-Side Prediction*, *Server Reconciliation* und *Interpolation*. Statt auf bereits vollständig implementierte Framework-Funktionalitäten zurückzugreifen, wird ein eigenes, leichtgewichtiges System entwickelt, um die Funktionsweise dieser Konzepte praktisch und nachvollziehbar umzusetzen.

NGO wird hierbei vor allem als *Basisschicht* für die Netzwerkkommunikation genutzt, während zentrale Mechanismen unabhängig davon realisiert werden. Dieser Ansatz erlaubt eine tiefere Auseinandersetzung mit den zugrunde liegenden Prinzipien und Herausforderungen im Netcode-Design.

Chapter 3

Beschreibung des Unity Prototyps

Der entwickelte Prototyp bildet eine einfache First-Person-Shooter-Sandbox ab und dient als Testumgebung für die Untersuchung netzwerkbezogener Mechaniken wie Interpolation, Client-Side Prediction und Lag Compensation. Der Prototyp wird auch zum Abschluss der Arbeit bewusst schlicht gehalten, da der Fokus auf der Netzwerkschicht liegt – nicht auf grafischen oder animationsbasierten Aspekten. Texturen, Beleuchtung und Animationen wurden daher nur in minimalem Umfang berücksichtigt. Im Verlauf der Arbeit existieren prinzipiell zwei Versionen, bei denen es sich zum einen um die Grundstruktur des Spiels gekümmert werden musste, was demnach den Offline-Prototypen darstellt. Diese Version besteht aus fundamentalen aber auch sehr schlicht gehaltenen Bestandteilen für ein FPS-Spiel. Diese bestehen aus einem Character Controller mit dem man springen und laufen kann und einer Waffe in First-Person-Ansicht, die über eine Schuss- und Nachlademechanik verfügt. Ein Schadensmodell, das als Metadaten den einzelnen Waffen zugewiesen werden kann und das alles auf einer minimalistischen Prototyp-Map. Auf diesem Prototypen anknüpfend, sollen und wurden besagte Netcode-Mechaniken inklusive Synchronisation integriert werden.

3.1 Szenario und Spiellogik

Die Umgebung erinnert an Aim-Trainingsszenarien aus Spielen wie *Valorant* (Range) oder eigenständigen Tools wie *Aim Lab*. Der Spieler wird durch einen FPS-Controller dargestellt, der als `Prefab` implementiert ist. In der Szene bewegen sich ein oder mehrere Zielobjekte (Targets), die sich zufällig und unvorhersehbar über das Spielfeld bewegen. Diese Targets können wiederholt erscheinen (respawnable) und dienen im späteren Verlauf zur Untersuchung netzwerkbedingter Bewegungsartefakte. Zusätzlich können sich mehrere Clients auf den Server verbinden und sollen synchronisiert werden. Hierbei geht es darum Inputlatenz auch bei hohem Ping minimal zu halten und ein flüssiges und direktes Spielgefühl zu erlauben auch bei hoher Latenz.

3.2 Waffenmechanik und Spielgefühl

Für das Waffenmodell wurde ein Asset aus dem Unity Asset Store verwendet[1], da die Eigenmodellierung in Blender aus zeitlichen Gründen verworfen wurde. Dennoch wurden grundlegende FPS-typische Elemente integriert:

- Rückstoß (Recoil)
- Bewegungsbasiertes Waffenbobbing und Sway
- Mündungsfeuer (Muzzle Flash)
- Einschusslöcher (Decals)

Diese Features verbessern nicht nur das Spielgefühl, sondern sind auch eine wichtige Grundlage für die spätere visuelle Analyse der Netzwerkmechaniken. Die Netzwerksynchronisation solcher Effekte gilt als nicht trivial und ist auch in professionellen Produktionen fehleranfällig und wird demnach ebenfalls ausgelassen. Spielerobjekte werden in Unity meistens über ein eigenes Network-Transform synchronisiert, welches in der Regel hauptsächlich nur Für die Position des jeweiligen Transforms zuständig ist. Wenn also Einschusslöcher synchronisiert werden müssen oder Animationen bzgl. der Waffen muss dies auf anderer Ebene entstehen was auch zu Kollisionen mit dem bereits verwendetem Network Transform.

[1] Platzhalter: *Name des Assets einfügen*

3.3 Technische Herausforderung: Raycasting

Ein zentrales technisches Problem ergab sich bei der Definition des Ursprungs der Schusslinie (Raycast). Zwei gängige Ansätze wurden gegenübergestellt:

1. Raycast vom Lauf der Waffe (Mündung)
2. Raycast aus dem Zentrum der Kamera (Fadenkreuz)

Für den Prototyp wurde Variante 2 gewählt, da diese Methode eine konsistentere Treffergenauigkeit gewährleistet und im Hinblick auf Debugging und Netzwerkanalyse einfacher zu handhaben ist.

3.4 Netzwerkarchitektur des Prototypen

Der Prototyp basiert auf dem *Netcode for GameObjects*-Framework von Unity und nutzt ein Server-Client-Modell. Clients werden über den Unity-Build gestartet und mit einem lokalen Server verbunden. Netzwerkkomponenten wie Zustandsübertragung und Objektregistrierung erfolgen aktuell noch über Unitys `NetworkTransform`-Komponente.

Diese Komponente erlaubt eine schnelle Prototypenerstellung, übernimmt jedoch automatisch Funktionen wie Interpolation und Zustandssynchronisation. Für die Zwecke dieser Arbeit ist dies ungeeignet, da keine gezielte Kontrolle oder Analyse der einzelnen Netzwerkmechanismen möglich ist.

3.4.1 Ausblick:

In den kommenden Entwicklungsschritten soll die `NetworkTransform`-Komponente teilweise durch eine eigene Implementierung ersetzt werden, um die Funktionsweise und Auswirkungen von Interpolation, Prediction, Reconciliation und Lag Compensation explizit untersuchen und visualisieren zu können.

Chapter 4

Analysewerkzeuge für Netcode und Debugging

Zur Untersuchung netzwerkbasierter Spielmechaniken wie Client-Side Prediction, Lag Compensation oder Interpolation ist der Einsatz geeigneter Analysewerkzeuge essenziell. Ziel ist es, sowohl das Verhalten dieser Mechaniken unter verschiedenen Bedingungen zu verstehen, als auch deren Einfluss auf das Spielgefühl sichtbar und messbar zu machen.

Im Rahmen dieser Arbeit soll ein System verwendet werden, bestehend aus einem Debug-Overlay und einer integrierten Ingame-Konsole. Dieses Werkzeug ermöglicht es, netzwerkrelevante Parameter – wie beispielsweise Interpolationszeiten, künstliche Latenz oder Glättungsalgorithmen – zur Laufzeit zu verändern. Alternativ wird dies nach herkömmlichem Ablauf über Netzwerkvariablen im Unity-Editor
So lassen sich spezifische Szenarien gezielt nachstellen und deren Auswirkungen visuell nachvollziehen.

Neben dieser Eigenentwicklung existieren auch externe Lösungen, wie etwa der Unity Profiler mit Netcode-Unterstützung oder Werkzeuge von Drittanbietern wie Photon Fusion Analyzer. Diese bieten detaillierte technische Einblicke, sind jedoch oft nicht für eine interaktive Analyse innerhalb des Spiels ausgelegt.

Für die Konsole, die auch in Spielen wie Counter Strike oder anderen Spielen existiert um ohne Benutzeroberfläche Einstellungen zu ändern, wird hier aus dem Unity Asset Store bezogen. [**<empty citation>**] Mit diesem Asset lassen sich Konsolen-Kommandos im Quellcode registrieren, die den Spielfluss manipulieren können. Außerdem ist auch das

Auslesen der Debugging-Konsole möglich wenn man im Build ist, welcher keine integrierte Konsole besitzt wie der normale Editor.

Das entwickelte Debug-System hingegen erlaubt eine tiefere Integration in den Entwicklungsprozess: Es zeigt Zustände wie die aktuelle Netzwerkverzögerung, Tick-Synchronisation oder Prediction-Fehler direkt im Spiel an. Ergänzend dazu erlaubt die Konsole das Aktivieren und Deaktivieren einzelner Netcode-Komponenten oder das dynamische Nachjustieren von Parametern – ohne das Spiel neu starten zu müssen.

Durch diese Flexibilität ist das System besonders geeignet für die iterative Entwicklung und Bewertung von Netcode-Strategien und stellt daher das zentrale Analysewerkzeug dieser Arbeit dar.

Chapter 5

Client-side Prediction & Reconciliation

Im folgenden Abschnitt werden die im Projekt umgesetzten und verwendeten Mechaniken beschrieben und anhand des Quellcode beschrieben. Hierbei liegt der Fokus darauf, ein durchdachtes System zu verwenden, das den Kern der Problematik einfach im Spiel erkenntlich machen kann.

5.1 Client-side Prediction beim Bewegen und Schießen

In einem serverautoritativen Netzwerkmodell ist es essenziell, ein faires und sicheres Spielumfeld zu gewährleisten. Gleichzeitig darf die Spielbarkeit aus Sicht des Spielers nicht unter einer hohen Latenz leiden. Besonders bei reaktionsintensiven Genres wie First-Person-Shootern oder Rennspielen ist eine direkte Rückmeldung auf Eingaben entscheidend für das Spielerlebnis.

Ein vollständiger Verzicht auf Client-Authorität würde zwar maximale Sicherheit bieten, jedoch zu spürbarer Verzögerung bei der Bewegung oder Aktion führen. Ziel ist daher ein Kompromiss: Der Server behält die Entscheidungsgewalt, während der Client Eingaben sofort lokal umsetzt und visuelles Feedback liefert.

In der vorliegenden Implementierung wird dies durch ein Überschreiben der Unity-eigenen `NetworkTransform`-Komponente realisiert. Die Autoritätslogik wird über eine einfache Enumeration gesteuert:

```
using Unity.Netcode.Components;
using UnityEngine;

namespace Player {
    public enum AuthorityMode {
        Server,
        Client
    }

    [DisallowMultipleComponent]
    public class ClientNetworkTransform : NetworkTransform {
        public AuthorityMode authorityMode =
                Player.AuthorityMode.Client;

        protected override bool OnIsServerAuthoritative() =>
                authorityMode == Player.AuthorityMode.Server;
    }
}
```

Listing 5.1: Überschreiben der NetworkTransform-Komponente mit AuthorityMode

Die Methode `OnIsServerAuthoritative()` gibt in der Regel `false` zurück, sodass der Client die Kontrolle über Bewegung und Eingabe erhält, ohne dass ein Roundtrip zum Server erforderlich ist. Dadurch wird die Latenzwahrnehmung beim Spieler erheblich reduziert.

5.2 Reconciliation – Zustandskorrektur bei Abweichungen

Trotz lokaler Vorhersage durch den Client bleibt der Server die Referenzinstanz für den „wahren“ Spielzustand. Sollte es zu größeren Abweichungen zwischen Client- und Serverposition kommen, wird ein Reconciliation-Prozess ausgelöst. Dabei wird der Client zurück auf den zuletzt vom Server bestätigten Zustand gesetzt und die eigenen Eingaben ab diesem Zeitpunkt erneut simuliert.

In Spielen mit präzisem Bewegungsfeedback (z.B. FPS) ist die Toleranzschwelle für Positionsfehler oft sehr gering. In diesem Projekt wurde ein Grenzwert: (`reconciliationThreshold`) definiert, ab dem eine Zustandskorrektur erfolgt. Dieser Wert kann abhängig von variabler Netzwerklatenz angepasst werden.

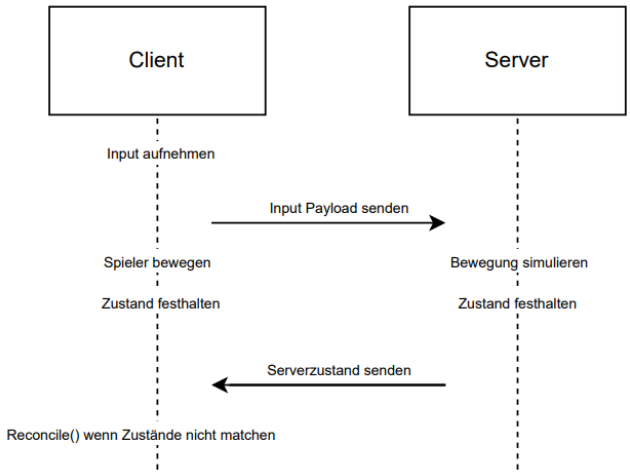

FIGURE 5.1: Sequenzdiagramm Reconciliation

5.3 Testmechanismus zur Validierung

Zur gezielten Auslösung der Reconciliation wurde ein einfacher Testmechanismus implementiert: Ein Client-Cheat teleportiert den Spieler um 20 Einheiten nach vorne – weit über die Fehlertoleranz hinaus. Der Server erkennt die Abweichung und triggert die Korrektur, wodurch der Spieler sichtbar in den korrekten Zustand „zurückspringt“.

Zur Visualisierung wurden farblich unterschiedliche Proxies (3D-Würfel) über dem Spielerobjekt angezeigt, die die vom Client bzw. Server berechneten Zustände darstellen.

5.4 Datenstrukturen und Ticksystem

Als Datenstruktur wurde für das Ticksystem auf zirkuläre Buffer gesetzt. Im Netcode-Bereich ist dies defacto Standard <Quelle maybe> um die mit den einzelnen Ticks verbundenen Informationen oder genauer Positionen, Rotationen und Geschwindigkeiten umzugehen und Overheads zu negieren. Demnach eignen sich bspw. Arrays oder Dictionaries deutlich schlechter, da diese die gespeicherten Informationen nicht verwerfen und somit unnöitge Daten halten, die für bspw. die Reconciliation außer Reichweite liegen und somit die Performance beeinträchtigen. In Kombination mit State und Input Payloads die jeweils als Structs implementiert wurden, sollen die Positionen zum passenden Tick verglichen werden können. Die Eigenimplementierung des Circular Buffer ist somit generisch und basiert auf den Input oder State Payloads. Mit der Methode Add(), die ein generisches Item also in diesem Fall das State oder Input Payload und einen Index als Parameter nimmt, kann somit der Buffer befüllt werden. Das Befüllen folgt den algorithmischen Grundlagen eines solchen Buffers und das geschieht bis zu der statischen Buffergröße und fängt dann wieder von null an zu indezieren. Somit werden in diesem Fall intern also bis 1024 Ticks verwaltet, was bei einer Standard-Tickrate von 60Hz ungefähr 17 Sekunden wären, die man zurückspulen könnte. Eine solche Buffergröße ist im Grunde schon relativ großzügig, sollte man jedoch auf eine Tickbasis aufsteigen von etwa 128Hz, ist es schon etwas sinnvoller. (vielleicht noch erwähnen warum genau...)

Im Folgenden sieht man noch den Aufbau des Circular Buffers und die Strukturen die essentiell sind für den Datentransfer:

```
namespace Player
{
    public class CircularBuffer<T>
    {
        private T[] buffer;
        private int bufferSize;

        public CircularBuffer(int bufferSize)
        {
            this.bufferSize = bufferSize;
            buffer = new T[bufferSize];
        }

        public void Add(T item, int index) =>
            buffer[index % bufferSize] = item;
        public T Get(int index) => buffer[index % bufferSize];
        public void Clear() => buffer = new T[bufferSize];
    }
}
```

LISTING 5.2: Methode `CircularBuffer()`

Auf der anderen Seite muss dies durch die angesprochenen Strukturen gespeichert werden:

```
public struct InputPayload : INetworkSerializable
{
    public int tick;
    public Vector3 inputVector;
    public bool forceTeleport;
    public Vector3 position;

    public void NetworkSerialize<T>(BufferSerializer<T> serializer)
        where T : IReaderWriter
    {
        serializer.SerializeValue(ref tick);
        serializer.SerializeValue(ref inputVector);
        serializer.SerializeValue(ref forceTeleport);
        serializer.SerializeValue(ref position);
    }
}
```

LISTING 5.3: Methode `Payload`

In der InputPayload werden die Eingabedaten des Spielers verwaltet und serialisiert, was über die vom zu implementierenden Interface INetworkSerializeable gestellten Methode NetworkSerialize() getan wird.

5.5 Kernmethoden der Reconciliation-Logik

Im Grunde lässt sich die Logik auf einige Methoden runterbrechen, dies sich alle im PlayerMovement Skript befinden. Diese Methoden sind jedoch verschachtelt in einem komplexeren Ablauf dieses Skriptes, reichen aber fürs Verständnis aus.

1. `ShouldReconcile()`
   Diese Methode prüft, ob neue Serverdaten vorliegen und ob diese sich vom zuletzt verarbeiteten Zustand unterscheiden:

```
bool ShouldReconcile()
{
    bool isNewServerState = !_lastServerState.Equals(default);
    bool isLastStateUndefinedOrDifferent =
         _lastProcessedState.Equals(default)
         || !_lastProcessedState.Equals(_lastServerState);
         return isNewServerState &&
                isLastStateUndefinedOrDifferent;
}
```

LISTING 5.4: Methode `ShouldReconcile()`

2. `ReconcileState(StatePayload rewindState)`
   Hier wird der Client-Zustand auf den vom Server bestätigten Zustand zurückgesetzt. Anschließend werden alle seitdem aufgezeichneten Eingaben erneut angewendet:

```
void ReconcileState(StatePayload rewindState)
{
    controller.enabled = false;
    transform.position = rewindState.position;
    transform.rotation = rewindState.rotation;
    _velocity = rewindState.velocity;
    controller.enabled = true;

    if (rewindState.Equals(_lastServerState)) return;

    _clientStateBuffer.Add(rewindState, rewindState.tick);
    int tickToReplay = _lastServerState.tick + 1;

    while (tickToReplay < _timer.CurrentTick)
    {
        int bufferIndex = tickToReplay % k_bufferSize;
        StatePayload statePayload =
        ProcessMovement(_clientInputBuffer.Get(bufferIndex));
        _clientStateBuffer.Add(statePayload, bufferIndex);
        tickToReplay++;
    }
}
```

LISTING 5.5: Methode `ReconcileState(StatePayload)`

3. `HandleServerReconciliation()`
   Diese Methode koordiniert den gesamten Reconciliation-Prozess und vergleicht die Zustände beider Seiten:

```
void HandleServerReconciliation()
{
    if (_lastServerState.tick <= 1) return;
    if (!ShouldReconcile()) return;

    int bufferIndex = _lastServerState.tick % k_bufferSize;
    if (bufferIndex - 1 < 0) return;

    StatePayload rewindState = IsHost
        ? _serverStateBuffer.Get(bufferIndex - 1)
        : _lastServerState;

    float positionError = Vector3.Distance(
        rewindState.position,
        _clientStateBuffer.Get(bufferIndex).position
    );

    if (positionError > reconciliationThreshold)
    {
        Debug.Break();
        ReconcileState(rewindState);
    }

    _lastProcessedState = _lastServerState;
}
```

LISTING 5.6: Methode `HandleServerReconciliation()`

Bibliography

[1] *Unity Runtime Fee – Unity*. Zugriff am 21. Mai 2025. URL: https://unity.com/pricing-updates.

[2] *Unreal Engine Licensing - Unreal Engine*. Zugriff am 21. Mai 2025. URL: https://www.unrealengine.com/en-US/license.

[3] *Godot Engine – Lizenzmodell*. Zugriff am 21. Mai 2025. URL: https://godotengine.org/license.

[4] CD Project Red. *The Witcher 4 — Unreal Engine 5 Tech Demo*. Letzter Zugriff: 25.06.2025. 2025. URL: https://www.thewitcher.com/us/en/news/51521/the-witcher-4-unreal-engine-5-tech-demo.

[5] Scott Kennedy. *How Epic is integrating Niagara into Fortnite*. https://www.unrealengine.com/de/tech-blog/how-epic-is-integrating-niagara-into-fortnite?. Letzter Zugriff: 21.05.2025. 2019.

[6] *FishNet Documentation*. Zugriff am 21. Mai 2025. URL: https://fish-networking.gitbook.io/docs/.