Netcode-Mechaniken in Echtzeitanwendungen

Alexander Seitz

14.05.2025

Inhaltsverzeichnis

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.

2 Auswahl der Spiel-Engine

In dieser Hinsicht ist es wichtig, Auswahlkriterien verschiedener Spiel-Engines in Betracht zu ziehen. Die wichtigsten Kriterien waren:

- **Lernkurve**: Wie zugänglich ist die Engine für Entwickler mit unterschiedlichen Erfahrungsstufen?
- **Community und Ressourcen**: Verfügbarkeit von Dokumentationen, Tutorials und aktiver Community.
- **Netcode-Fähigkeiten**: Wie gut lässt sich Multiplayer-Logik implementieren?
- **Performance**: Optimierungsmöglichkeiten für verschiedene Plattformen, insbesondere im Multiplayer-Bereich.
- **Kosten und Lizenzmodell**: Wie gut zugänglich sind die einzelnen Produkte?
- **Programmiersprache**: Gibt es hier noch hohen zusätzlichen Lernaufwand?
- **Marktrelevanz**: Wie etabliert ist die Engine in der Spielebranche?

Ein direkter Vergleich der bekanntesten Engines zeigt folgende Eigenschaften:

Tabelle 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.

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 [4].

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.

2.1 Eigenimplementierung und Auswahl des Netcode-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 [5]. 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.

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.

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.

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 Netzwerk-Synchronisation solcher Effekte gilt als nicht trivial und ist auch in professionellen Produktionen fehleranfällig.

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)

[1] Platzhalter: *Name des Assets einfügen*

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 (aktueller Stand)

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.

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

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 eigenes Tool entwickelt 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.
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.

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.

5 Literaturverzeichnis

Literatur

[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] 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.

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