HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

BACHELORARBEIT

Herr
Moritz K. Rehschuh

Entwurf und Evaluation eines performanten Netzwerkmoduls für simultane Multiplayer-Spiele am Beispiel "Scoo-King"

2023

Fakultät **Angewandte Computer- und Biowissenschaften**

BACHELORARBEIT

Entwurf und Evaluation eines performanten Netzwerkmoduls für simultane Multiplayer-Spiele am Beispiel "Scoo-King"

Autor:
Moritz K. Rehschuh

Studiengang:
Medieninformatik und Interaktives Entertainment

Seminargruppe:
MI18w1-B

Erstprüfer:
Prof. Dr.-Ing. Christian Roschke

Zweitprüfer:
M. Sc. Manuel Heinzig

Mittweida, 05 2023

Bibliografische Angaben

Rehschuh, Moritz K.: Entwurf und Evaluation eines performanten Netzwerkmoduls für simultane Multiplayer-Spiele am Beispiel "Scoo-King", 48 Seiten, 7 Abbildungen, Hochschule Mittweida, University of Applied Sciences, Fakultät Angewandte Computer- und Biowissenschaften

Englischer Titel: *Design and evaluation of a high-performance network module for simultaneous multiplayer games using the example of "Scoo-King"*

Bachelorarbeit, 2023

Satz: LaTeX

Kurzbeschreibung

Diese Arbeit beschäftigt sich mit der Entwicklung von performanten Modulen zur Implementierung von Netcode in einem Multiplayer-Spiel. Sie vergleicht dabei zunächst verschiedene Frameworks und dokumentiert den Entscheidungsprozess anhand eines konkreten Spiels. Es werden daraus folgend mehrere Ansätze für den Aufbau eines Netzwerkmoduls aufgezeigt und diese schließlich miteinander auf ihre Performance verglichen.

I. Inhaltsverzeichnis

II. Abbildungsverzeichnis

III. Tabellenverzeichnis

IV. Abkürzungsverzeichnis

CDN	Content-Delivery-Network
HLAPI	High Level Application Programming Interface
LLAPI	Low Level Application Programming Interface
MLAPI	Mid Level Application Programming Interface
MMORPG	Massively Multiplayer Online Role-Playing Game
P2P-Verbindung	Peer-to-Peer Verbindung
RPC	Remote Procedure Call
RTT	Round-Trip-Time
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

V. Danksagung

Ich danke Herrn Heinzig für die tatkräftige Unterstützung, wann immer ich Fragen zu dieser Arbeit hatte.

Ich danke auch Professor Roschke, der kurzfristig bereit war, aufgrund unvorhersehbarer Änderungen, die Rolle des Erstprüfers für meine Arbeit zu übernehmen.

Ich danke Professor Marbach für seine Unterstüzung während der Entwicklung von Scoo-King für die „Beta 2021“ und die Möglichkeit, mit jedem Problem zu ihm zu kommen.

Ich danke meiner Familie für ihre Unterstützung, wann immer ich sie brauchte.

1 Einleitung

Jedes Jahr findet gegen Ende des Wintersemesters an der Hochschule Mittweida für den Studiengang Medieninformatik ein großes Event mit dem Namen „Beta“ statt. Als Höhepunkt dieses Events stellen die Studierenden des jeweilige Abschlussjahrgangs ein Computerspiel vor, welches sie im Voraus selbst entwickelt haben. Für die „Beta 2021“ war dies mein Jahrgang mit dem Spiel Scoo-King.

Scoo-King war dabei ursprünglich als ein lokaler Multiplayer ausgelegt. Die Arbeit an dem Spiel wurde jedoch auch nach dieser ersten offiziellen Veröffentlichung auf freiwilliger Basis fortgeführt und so wurde Scoo-King zu einem Online-Multiplayer-Spiel weiterentwickelt. Über die letzten zwei Jahre war ich sehr aktiv an diesem Prozess beteiligt. Im Laufe dessen stellten sich die Fragen, welche diese Arbeit nun beantworten soll: Wie könnte ein netzwerkfähiges Modul zur Implementierung von Spielmechaniken in einem Spiel wie Scoo-King aufgebaut sein? Und wie schneiden verschiedene Ansätze für ein solches Vorhaben in einem Vergleich ihrer Performance untereinander ab?

Um diese Fragen zu beantworten, wird sich diese Arbeit daher zunächst mit einigen Grundlagen für die Entwicklung von Multiplayer-Spielen auseinandersetzen. Im Anschluss werden verschiedene gängige Multiplayer-Frameworks für Unity vorgestellt und beispielhaft demonstriert, wie der Entscheidungsprozess für ein konkretes Framework, in Abhängigkeit der Anforderungen des Spiels, aussehen kann. Daraufhin werden die von dem gewählten Framework dargebotenen Optionen analysiert und mit deren Hilfe vier verschiedene Ansätze zum Aufbau eines Netzwerkmoduls erarbeitet. Diese Ansätze werden dann testweise implementiert und gemessen, bevor dann die gesammelten Ergebnisse miteinander verglichen werden können. Zu guter Letzt sollen dann mithilfe dieser Daten die oben gestellten Fragen beantwortet werden können.

2 Grundlegender Aufbau von Multiplayer-Spielen

Multiplayer-Spiele haben in den letzten Jahren enorm an Beliebtheit gewonnen. Dabei handelt es sich um Videospiele, die von mehreren Spielern gleichzeitig gespielt werden können. Dies kann entweder online, über ein lokales Netzwerk oder in Form von lokalem Multiplayer mit mehreren Spielern vor demselben Bildschirm passieren. (Armitage, Claypool und Branch, 2006, S. 15–17) Letztere Variante stellt aus Sicht der Software-Architektur einen Sonderfall dar, da die Code-Struktur eines Spiels dieser Art eher einem Singleplayer-Spiel ähnelt als einem der anderen beiden Fälle. Da diese Spiele in Echtzeit gespielt werden, ist eine stabile und schnelle Netzwerkverbindung von entscheidender Bedeutung. Dieses Kapitel soll sich daher mit der grundlegenden Netzwerk-Topologie von Multiplayer-Spielen befassen und einige der wichtigsten Konzepte und Technologien vorstellen.

2.1 Netzwerk-Topologie

Der Begriff Netzwerk-Topologie beschreibt die Anordnung von verschiedenen Geräten und Verbindungen in einem Netzwerk. Er bezieht sich auf die Art und Weise, wie Geräte in einem Netzwerk miteinander verbunden sind. In einem Multiplayer-Spiel ist es wichtig, dass alle Spieler in Echtzeit miteinander kommunizieren können. Dafür gibt es im Wesentlichen zwei Ansätze: zentralisierte und dezentralisierte Netzwerk-Topologien.

Bei einer zentralisierten Topologie wird ein zentraler Server eingesetzt, der als Vermittler zwischen den Spielern fungiert. Das Client-Server-Modell teilt die Verantwortlichkeiten zwischen den Clients und dem Server auf. Der Server empfängt und sendet Daten zwischen den Spielern und ist für die Synchronisierung des Spielzustands verantwortlich. Er nimmt Anfragen an, verarbeitet diese und teilt den Clients die Ergebnisse mit. Der Client auf der anderen Seite stellt in der Regel Anfragen an den Server und visualisiert dann das Ergebnis der abgerufenen Informationen für den Benutzer.

In der Regel werden Client und Server auf zwei separaten Computern ausgeführt und sind über ein Netzwerk miteinander verbunden. In diesem Fall spricht man von einem dedizierten Server, der nichts weiter tut, als mit den Clients zu kommunizieren und deren Anfragen zu bearbeiten. Diese Art von Netzwerk-Topologie wird häufig in großen Multiplayer-Spielen wie MMORPGs (Massively Multiplayer Online Role-Playing Games) verwendet, da sie eine bessere Kontrolle über das Spiel ermöglicht. Des Weiteren ob-

liegt so die Entscheidung über die Leistungsanforderungen an die Hardware des Servers sowie dessen Netzwerkverbindung dem Entwickler und kann bei Bedarf problemlos aufgerüstet werden. Durch den Einsatz von dedizierten Servern entsteht dadurch nahezu unbegrenzte Skalierbarkeit. (Kelly und Kumar, 2022, S. 2–3)

Es ist jedoch auch möglich, dass Client und Server auf demselben Computer ausgeführt werden. In der Regel spricht man dann von einem Host. In gewisser Weise stellt ein solches Host-Client-Modell eine Mischlösung zwischen zentralisierten und dezentralisierten Ansätzen dar. Die fehlende Notwendigkeit, einen extra Computer als Server zu betreiben, stellt einen erheblichen Vorteil eines Host-Client-Modells dar. Dadurch kann man die Betriebskosten eines solchen Systems erheblich senken. Andererseits muss damit der Host die Rechenarbeit des Servers und seines eigenen Clients übernehmen. Dies kann in Abhängigkeit der Hardware des Endnutzers, der für eine Sitzung als Host fungiert, möglicherweise zu Einbußen in der Performance führen. Auch die Internetverbindung des Host ist ein ungewisser Faktor und kann zu Verbindungsproblemen führen, da sie in einer solchen Konfiguration viel stärker beansprucht wird als bei den anderen Clients der Sitzung. (Wang, 2017, S. 9–10)

Bei einer dezentralisierten Topologie hingegen gibt es keinen zentralen Server. Stattdessen kommunizieren die Spieler direkt miteinander und synchronisieren den Spielzustand untereinander. (Boroń, Brzeziński und Kobusińska, 2019) Diese Art von Netzwerk-Topologie wird häufig in kleineren Multiplayer-Spielen wie First-Person-Shootern und Strategiespielen eingesetzt, da sie eine höhere Geschwindigkeit und weniger Latenz bieten. Grund dafür ist die Möglichkeit des direkten Datenaustauschs zwischen zwei Clients, ohne dass dieser zuerst über einen extra Server umgeleitet werden muss. Allerdings muss damit auch jeder Client über die Verbindungsinformationen für alle anderen Clients Bescheid wissen. Dies kann zu Sicherheitsproblemen führen, da ein bösartiger Client so in den Besitz der IP-Adressen aller anderen Clients einer Sitzung kommen kann. Zusätzlich skaliert dieser Ansatz nur begrenzt. Aufgrund der Tatsache, dass jeder Client stetig Änderungen an alle anderen Clients kommunizieren muss, wird die Menge der zu versendenden Daten ab einer gewissen Anzahl verbundener Clients einfach zu groß, als dass die durchschnittliche Internetverbindung eines Endnutzers diese Last bewältigen kann.

2.2 Netzwerk-Protokolle

Um eine stabile und schnelle Netzwerkverbindung für Multiplayer-Spiele zu gewährleisten, werden verschiedene Netzwerk-Protokolle verwendet. Ein Protokoll ist im Wesentlichen eine Reihe von Regeln und Verfahren, die die Kommunikation zwischen verschiedenen Geräten im Netzwerk regeln. Die beiden wichtigsten Netzwerk-Protokolle für Multiplayer-Spiele sind: TCP und UDP. (Sanchez, RJ (Rodrigo), 2020, S.15)

TCP steht für Transmission Control Protocol und ist ein zuverlässiges Protokoll, das eine sequenzielle Übertragung von Datenpaketen gewährleistet. Dabei stellt es sicher, dass die übertragenen Datenpakete vollständig, verlustfrei und in der richtigen Reihenfolge eintreffen. TCP wird daher häufig für die Übertragung von wichtigen Informationen verwendet, die nicht verloren werden dürfen. Zu diesem Zweck werden die ausgehenden Daten einer Anwendung in einzelne TCP-Frames aufgeteilt und einzeln übertragen. Die TCP-Schicht des Zielhosts bestätigt explizit den Empfang eines jeden TCP-Frames mit einer Rückmeldung an den Sender. Dadurch ist das Protokoll in der Lage zu erkennen, wann Verluste aufgetreten sind. Wird der Erhalt eines TCP-Frames nicht bestätigt, weiß der Sender, dass die Daten nicht korrekt angekommen sind und überträgt den TCP-Frame erneut bis zur Bestätigung durch den Empfänger. Dadurch wird sichergestellt, dass die Datenübertragung vollständig und zuverlässig erfolgt. Durch die größere Menge an Informationen im Header des Datenpakets und die ständigen Rückmeldungen für jeden TCP-Frame ist TCP allerdings langsamer bei der Übertragung von Daten.

UDP steht für User Datagram Protocol und ist dahingegen ein unzuverlässiges Protokoll. Dafür ermöglicht es schnelle Übertragung von großen Mengen an Datenpaketen. UDP wird daher häufig für sich ständig wandelnde Informationen, wie beispielsweise Standortdaten in der Spielwelt oder das Streaming von Video-Dateien, verwendet. Bei dieser Art von Informationen ist es in der Regel kein großes Problem, wenn ein einzelnes Datenpaket zwischendurch verloren geht, da die Information von der nächsten Übertragung ohnehin wieder überschrieben wird. UDP erzwingt keine Flusskontrolle bei der Paketübertragung zur Ankunft der Pakete in exakt derselben Reihenfolge, wie sie versendet wurden. Aus diesem Grund bietet es daher keine Verlusterkennung von Paketen oder deren Wiederherstellung. Gleichzeitig ermöglicht das UDP-Protokoll dadurch aber auch hohe Übertragungsgeschwindigkeiten selbst bei großen Datenmengen. (Armitage, Claypool und Branch, 2006, S. 45–47)

2.3 Latenz und Bandbreite

Zwei wichtige Faktoren, die sich entscheidend auf die Netzwerk-Performance eines Multiplayer-Spieles auswirken können, sind Latenz und Bandbreite.

Latenz beschreibt dabei die Zeit, die benötigt wird, um Daten von einem Punkt im Netzwerk zum anderen zu übertragen. Der Endnutzer nimmt Latenz als die zeitliche Verzögerung zwischen einer vom ihm ausgeführten Aktion und der Darstellung des Resultats dieser Aktion auf seinem Client wahr. (Glazer, 2016) Je niedriger die Latenz, desto schneller und reibungsloser wird das Spiel ausgeführt. Die Latenz wird durch die Round-Trip Time (RTT) gemessen. Diese berechnet sich aus der Zeit, die ein gesendetes Paket von einem Punkt im Netzwerk zu einem anderen benötigt, und der Zeit, die das Antwortpaket benötigt, um zum Ausgangspunkt zurückzukehren.

Neben der Zeit zum reinen Senden und Empfangen eines Netzwerkpaketes wird die RTT auch von der Framerate und der benötigten Rechenzeit für die Serialisierung und Deserialisierung der Informationen des Pakets beeinflusst. (Glazer, 2016) Wieviel Latenz für einen Nutzer akzeptabel ist, hängt auch stark von psychologischen Faktoren ab. Dabei hat die Art der Interaktion gemeinhin Einfluss auf die Akzeptanzschwellen für unterschiedliche Anwendungen. (Ahde, 2017, S. 13–15)

Die Bandbreite hingegen bezieht sich auf die Menge an Daten, die gleichzeitig übertragen werden können. Je höher die Bandbreite, desto mehr Daten können in kürzerer Zeit übertragen werden, was zu einer besseren Netzwerk-Performance führt. Die Zahl der gleichzeitigen Nutzer sowie die Komplexität der Informationen, die kommuniziert werden müssen, haben dabei einen großen Einfluss auf die benötigte Bandbreite. Da Bandbreite eine limitierte Ressource ist, hat die Entscheidung, welche Daten in einem Spiel tatsächlich notwendigerweise versendet werden müssen und in welcher Regelmäßigkeit diese kommuniziert werden müssen, einen großen Einfluss auf die Performance eines Multiplayer-Spiels. (Magda El Zarki, o. D.)

Um Latenz und Bandbreite zu optimieren, werden verschiedene Technologien und Ansätze verwendet. Ein Ansatz ist die Verwendung von dedizierten Servern, die speziell für Multiplayer-Spiele konzipiert sind und eine höhere Bandbreite und geringere Latenz bieten können als herkömmliche Server. Eine weitere Technologie ist die Verwendung von Content-Delivery-Networks (CDNs), die den Spielinhalt auf mehrere Server auf der ganzen Welt verteilen, um die Latenz für Spieler auf der ganzen Welt zu minimieren.

3 Vorüberlegungen

3.1 Gängige Unity-Multiplayer-Frameworks

Online-Multiplayer-Spiele sind keineswegs eine Neuerung. Bereits seit den 1980-er Jahren gab es textbasierte Multi-User Dungeons, die es ihren Spielern erlaubten, über das Internet eine gemeinsame Spielwelt zu durchwandern und zu beeinflussen. (Smed, Kaukoranta und Hakonen, 2002, S.13) Während diese Spiele im Vergleich zum heutigen Maßstab noch sehr simpel waren, entwickelten sie sich im Laufe der Zeit in Maßstab und Anspruch bedeutend weiter. (Armitage, Claypool und Branch, 2006, S. 17–28) Mit diesen Entwicklungen bedurfte es jedoch auch technischer Neuerungen.

Bei diesen frühen Vertretern wurde jedes Spiel vollständig autark programmiert, wodurch es notwendig war, alle grundlegenden Funktionen jedes Mal von Grund auf neu zu schreiben. Dies änderte sich mit der Entwicklung von Game-Engines wie *id Softwares* „Quake Engine“, *Valves* „Source Engine“ oder *Epic Games* „Unreal Engine“. Dabei handelt es sich im Wesentlichen um eine Sammlung von Werkzeugen und Funktionen, die es Entwicklern ermöglichen, Grafiken, Audio, Physik, KI, Netzwerkfunktionen und andere Komponenten eines Spiels zu integrieren und zu verwalten, ohne sie von Grund auf neu schreiben zu müssen. Game-Engines ermöglichten es, Spiele schneller und kosteneffektiver zu produzieren sowie eine schnellere und einfachere Portierung von Spielen auf verschiedene Plattformen wie PC, Konsolen und Mobilgeräte zu gewährleisten.

Im Laufe der Zeit entwickelten sich einige sehr berühmte und erfolgreiche Game-Engines. Zwei der bekanntesten Beispiele für öffentlich verfügbare Game-Engines sind Unreal Engine und Unity. (Wang, 2017, S. 4–8) Das in dieser Arbeit thematisierte Spiel ScooKing beispielsweise ist in Unity geschrieben.

Unity wird von Unity Technologies entwickelt. Die Engine bietet Entwicklern auf der ganzen Welt einfache Möglichkeiten, um Spiele und andere interaktive Anwendungen für eine Vielzahl von Plattformen wie PC, Konsolen, mobile Geräte und AR/VR-Headsets zu entwickeln. Die Unity-Engine ist bekannt für ihre Benutzerfreundlichkeit und ihre leistungsfähigen Tools zur Entwicklung von Spielen. Sie bietet eine Vielzahl von Funktionen, die die Entwicklung von Spielen vereinfachen und beschleunigen können. Dazu gehören eine leistungsstarke grafische Benutzeroberfläche, die Möglichkeit, Spiele in Echtzeit zu bearbeiten, ein integriertes Physiksystem, Animationstools und eine große Bibliothek von Assets, die von Entwicklern genutzt werden können.

Unity unterstützt auch mehrere Programmiersprachen wie C#, JavaScript und Boo. Dadurch bietet Unity Entwicklern die Freiheit, die Programmiersprache zu wählen, mit der sie am besten vertraut sind. Es ist möglich, Plugins und Erweiterungen zu erstellen, um die Funktionalität von Unity darüber hinaus zu erweitern.

Ähnlich wie in der allgemeinen Computerspiel-Entwicklung Game-Engines entstanden, um die Programmierung von Spielen einfacher zu machen und immer gleiche Bestandteile in wiederverwendbaren Bibliotheken anzubieten, passierte Vergleichbares auch für die Netzwerk-Komponenten von Online-Spielen. Durch das Angebot verschiedenster Netzwerk-Frameworks bieten sich einfache Möglichkeiten, Daten zu empfangen, Daten zu versenden oder Informationen aus Datenbanken und APIs zu beziehen. (Ahmed et al., 2018, S.8)

Ein Netzwerk-Framework stellt eine Bibliothek von Source-Code dar, die die Implementierung von Netzwerk-Features für ein Online-Multiplayer-Spiel stark vereinfacht. Aufwendige Schritte wie beispielsweise die Implementierung von Netzwerk-Protokollen können hinter einfach zu verwendenden Abstraktionsebenen verborgen werden. Auch die Aufschlüsselung von Informationen in einzelne Datenpakete, die dann über eine Netzwerkverbindung verschickt werden können, lässt sich durch den Einsatz solcher Frameworks automatisieren und vereinfachen. Wichtig zu vermerken ist dabei, dass der Grad der Abstraktion von Framework zu Framework variieren kann. Während eine hohe Abstraktionsebene zwar die Verwendung eines Frameworks erheblich vereinfachen kann, bieten geringere Level an Abstraktion mehr Kontrolle über die tatsächlich ein- und ausgehenden Datenströme und können somit dazu beitragen, hochperformante Multiplayer-Spiele mit geringer Bandbreitennutzung zu schreiben. Zu den gängigsten dieser Frameworks für Unity gehören unter anderem UNet, MLAPI, DarkRift 2, Mirror, Forge Networking Remastered und Photon. In den folgenden Abschnitten werden diese mit ihren Fähigkeiten sowie Vor- und Nachteilen kurz vorgestellt.

3.1.1 UNet

Das Erste, nach dem ein Entwickler vermutlich Ausschau hält zum Thema Netcode in Unity, wäre Unitys offizielle (First-Party) Networking-Solution. Hier trifft er allerdings auf ein Problem. Denn bis vor einigen Jahren war dies Unitys UNet-Framework. In Kombination mit zwei möglichen APIs, der HLAPI für hochrangige Abstraktions-Konzepte und eine einfache Handhabung und der LLAPI für geringe Abstraktion und große Kontrolle über den Datenverkehr, stellte UNet die Standard Networking-Solution für Unity dar. Dennoch hatte UNet viele Probleme in den Bereichen von Performance, Skalierbarkeit und Sicherheit, und so entschied man sich im April 2019 von offizieller Seite, UNet für veraltet zu erklären und jeglichen weiteren Support einzustellen. (Glover, 2019)

3.1.2 MLAPI / Netcode for GameObjects

Mit der Einstellung des Supports für UNet kündigte das Unity-Entwicklerteam gleichzeitig die Entwicklung eines komplett neuen Netzwerk-Frameworks an, welches in Zukunft die Rolle der offiziellen Networking-Solution übernehmen sollte. Zu diesem Zweck erwarb Unity das bereits existierende Netzwerk-Framework MLAPI, kurz für Mid Level Application Programming Interface, und entwicklte dieses unter dem Namen „Netcode for GameObjects“ weiter.

Netcode for GameObjects ist eine Open-Source Netzwerk-Bibliothek, die eine große Bandbreite von moderat einfach zu verwendenden Möglichkeiten bietet, um Netcode für Unity-Spiele zu schreiben. Zu diesem Zweck stellt sie Features wie Remote Procedure Calls (RPCs), NetworkedVariables, Scene Management, Network Messaging und vieles mehr zur Verfügung. Durch die Verwendung einer strikten Abstraktion zwischen Netcode und Transportschicht bietet das Framework eine einfache Lösung, um verschiedene Transports zu verwenden und unterschiedliche Netzwerk-Topologien und Plattformen anzusprechen. (Unity Technologies, 2020, S.3)

Netcode for GameObjects fokussiert sich primär auf die Entwicklung von kleineren, kooperativen P2P-Spielen und bietet gute Anbindungsmöglichkeiten an Unity Relay und Unity Lobby Solutions. (House, 2020)

3.1.3 DarkRift 2

DarkRift 2 ist eine schnelle und hochperformante Low-Level Networking-Solution für Unity. Dabei fokussiert das Framework eine Netzwerktopologie mit dediziertem Server und implementiert eine Multithread Server Runtime, die mit Plug-ins erweitert werden kann. Dieses Plug-in-System kann auch in Kombination mit einem Unity-basierten Server verwendet werden und ist dank Multithreading eine der skalierbarsten Networking-Solutions auf dem Markt. Ursprünglich als kostenlose Version mit reduziertem Umfang oder kostenpflichtige Version mit allen Features und Zugang zum Sourcecode angeboten, ist DarkRift2 seit April 2022 kostenlos als Open-Source Projekt verfügbar. (Kubasova, 2022)

DarkRift 2 bietet Unterstützung für mehrere Netzwerkkanäle, um sowohl TCP als auch UDP-Verbindungen zu erlauben. Des Weiteren bietet das Framework Möglichkeiten für Serialisierung und anpassbare Protokollierung, erfordert aber gehobene C#-Kenntnisse, um es effektiv nutzen zu können. (Unity Technologies, 2020, S.5)

3.1.4 Mirror

Ähnlich wie auch MLAPI ist Mirror eine Weiterentwicklung der HLAPI von UNet mit dem Ziel, das Framework zu verbessern, bestehende Probleme zu beheben und seit der Einstellung der Entwicklung von UNet ersatzweise kontinuierlichen Support zu gewährleisten. Als Open-Source Projekt wurzelt Mirror auf einer großen und aktiven Community und glänzt dadurch mit einer gewaltigen Sammlung von Tutorials, Beispielen, Dokumentation und Hilfe-Foren.

Mirror fokussiert eine Client-Server Architektur, bietet aber die Möglichkeit, diese mit minimalem Aufwand in eine Host-Client Architektur umzuwandeln. Das Framework lässt sich leicht skalieren und ist sowohl für kleinere Spiele mit nur wenigen Spielern, aber auch für die Entwicklung von Massively-Multiplayer-Online-Games (MMORPGs) mit mehreren Hundert Spielern pro Session geeignet. Mirror ermöglicht umfangreiche Anpassungen, um die Engine an das zu entwickelnde Spiel anzupassen, einschließlich des einfachen Austauschs des Transportschicht gegen andere unterstützte Bibliotheken. Gleichzeitig werden dem Entwickler zahlreiche praktische Tools an die Hand gegeben, um auf einfachste Weise den Netcode für das eigene Spiel schreiben zu können. Dazu gehören Remote Procedure Calls (RPCs), synchronisierte Variablen, Scene Management, Custom Network Messages, Interest Management, fertige Netzwerk-Komponenten, Serialisierung und vieles mehr. Die Verwendung von Mirror ist komplett kostenlos. (Sanchez, RJ (Rodrigo), 2020, S. 20–21)

3.1.5 Forge Networking Remastered

Forge Networking Remastered ist ebenfalls eine Open-Source-Netzwerkbibliothek und kann als solche kostenlos verwendet werden. In ihrem Workflow ähnelt sie stark dem von Photon Bolt, welches im folgenden Abschnitt näher erläutert wird. Im Gegensatz zu den meisten anderen hier vorgestellten Frameworks ist Forge Networking nicht speziell an Unity als Game-Engine gebunden ist und auch außerhalb dieses Kontextes verwendbar ist. Das Framework konzentriert sich dabei auf eine Server-Client-Architektur und bietet verschiedene Features zur Entwicklung von Low-Level-Netcode wie Remote Procedure Calls (RPCs), State Rewinding, NAT-Punch-Through-Server, Master-Server und Multithreaded Code. Diese werden in der Regel mittels eines Wizards im Editor implementiert, der dann entsprechenden Boilerplate-Code generiert, welcher dann wiederum vom Entwickler ausgebaut werden kann. Forge Networking Remastered präsentiert qualitativ hochwertige, aber manchmal knappe Dokumentation und versagt in dieser Kategorie, indem es einige veraltete Video-Tutorials und eine schwer lesbare API bereitstellt. (Sanchez, RJ (Rodrigo), 2020, S.21)

3.1.6 Photon

Eine weitere sehr populäre Networking-Solution für Unity ist Photon. Oder besser gesagt die Photon-Produktpalette der Firma Exit Games. Diese Beliebtheit resultiert aus dem Umstand, dass Photon eine Reihe von Networking-Solutions mit unterschiedlichem Umfang und Ansatz darstellt, mit denen ein breites Spektrum an Bedürfnissen abgedeckt werden kann. Des Weiteren werden auch diverse Cloud-Services und Server-Hosting angeboten. Durch diese Angebotsvielfalt, die ausführliche und leicht verständliche Dokumentation und die hohe Qualität der angebotenen Features erfreuen sich Exit Games' Produkte auch im professionellen Bereich großer Beliebtheit. Die Nutzung dieser Dienste zu Testzwecken ist in der Regel kostenfrei, zieht aber nach Veröffentlichung des Spiels eine monatliche Gebühr nach sich, deren Höhe in Abhängingkeit von der Zahl der gleichzeitigen Nutzer variiert. Neben Grundlagen-Lösungen wie Photon Server oder Photon Realtime, mit denen sich eigene Networking-Stacks entwickeln lassen, finden sich hier auch Photon PUN, Photon Bolt und Photon Quantum. (Exit Games, 2023)

Photon PUN
Photon PUN ist im Wesentlichen eine Mesh-Topologie-Lösung oder auch direkte P2P-Lösung, bei der jeder Client die Daten aller anderen Clients synchronisiert. Dabei stellt das Framework ein einfaches Scripting-Interface rund um Remote Procedure Calls und Network Events zur Verfügung. Probleme mit NAT-Traversal löst Photon PUN über einen Relay- (oder auch Master-) Server. Dieser ist ausschließlich für die Weiterleitung von Netzwerk-Nachrichten zuständig und führt selbst keinerlei Synchronisierungslogik aus. Über Plug-ins kann der Relay-Server zwar eingehende Nachrichten überprüfen und somit zur Betrugserkennung verwendet werden, ist aber nicht in der Lage, zur Betrugsprävention aktiv einzugreifen. (Unity Technologies, 2020, S.7)

Photon Bolt
Photon Bolt ist ein führender Service in der Kategorie der schnellen Actionspiele und bietet fortschrittliche Tools, die gut zu diesem Zielspieltyp passen. Zum Feature Set des Frameworks gehören autoritäres Movement, Lag-Kompensation, Hitbox Recording, Client-Prediction, Interpolation und vieles mehr. Anders als Photon PUN verwendet Photon Bolt ein streng autoritäres Server-Client-Konzept. Die Einrichtung mit der Erstellung eines extra Kontos und der Registrierung der Spielinstanz ist etwas komplizierter als bei anderen Einträgen auf dieser Liste. Dafür jedoch ist die eigentliche Spiele-Entwicklung mit Photon Bolt sehr einfach. Innerhalb von Unity lassen sich mittels eigener Editor-Fenster alle Netzwerk-Objekte konfigurieren und Entwickler finden hier die Möglichkeit, entsprechenden Quellcode automatisch generieren zu lassen. Dieser Boilerplate-Code kann im Anschluss entsprechend erweitert und modifiziert werden. (Sanchez, RJ (Rodrigo), 2020, S. 20–21)

Photon Quantum

Während Photon PUN und Photon Bolt recht ähnliche Ansätze verfolgen, ist Photon Quantum im Wesentlichen der Simulationsteil einer Game-Engine, die sich im Kern auf die nötigen Features für Spiele mit einem deterministischen Rollback-System konzentriert. Photon Quantum verwendet ein Entity Component System (ECS) und Multithreading. Diese sind nicht zu verwechseln mit Unitys jüngstem ECS und Job System. Die Verwendung von Photon Quantum erfordert daher zwingend die Verwendung der Framework-eigenen Simulationsbibliotheken für beispielsweise Physik, Fixpunktmathematik und Pathfinding anstelle von Unity-Bibliotheken.

Die Idee von deterministischem Rollback besteht darin, lediglich die Nutzereingaben eines Clients zu senden, und nachdem diese von allen anderen Clients akzeptiert wurden, sie auf jedem Client lokal auszuführen. Jeder Client erstellt dann separate Vorhersagen für andere Spieler und ist in der Lage, diese Vorhersagen wenn nötig zu korrigieren oder gar komplett rückgängig zu machen, falls sie sich als inkorrekt erweisen sollten. Dieser Ansatz bietet einen Kompromiss zwischen der Betrugsresistenz und garantierten Synchronizität eines reinen Lockstep-Algorithmus auf der einen Seite und des geschmeidigeren Spielgefühls konventioneller P2P-Lösungen auf der anderen Seite. Durch die Notwendigkeit, Synchronisation und Simulation für jeden Spieler auf jedem Client auszuführen, ist dieser Ansatz stark von der Ziel-Hardware des Endnutzers abhängig und limitiert die Verwendung des Frameworks auf Spiele mit geringeren Spielerzahlen pro Lobby. (Unity Technologies, 2020, S.9)

	Stability/ support	Ease-of-use	Performance	Scalability	Feature breadth	Cost*	Customers recommend for
MLAPI	★★★★★	★★★★☆	★★★★★	★★★★☆	★★★★★	Free	Most client-server games for up to ~64 players that want a stable breadth of mid-level features
DarkRift 2	★★★★★	★★★☆☆	★★★★★	★★★★★	★★★☆☆	$100 for source	Games with high perf/ scale requirements that want to build on a stable LL layer
Photon PUN	★★★★☆	★★★★★	★★★☆☆	★☆☆☆☆	★★★★☆	$0.30/PCU	Simple and small (2–8 players) mesh-topology games
Photon Quantum 2.0	★★★★☆	★★★★☆	★★★★★	★★☆☆☆	★★★★☆	$1000/mo + $0.50/PCU	Games desiring deterministic roll-back, like MOBA games, for up to 32 players
Mirror	★★★★★	★★★★★	★★★☆☆	★★★★☆	★★★★☆	Free	Stable and proven client-server solution, loved best for its community and ease-of-use

* Note that Photon pricing provides access to the networking libraries and services, whereas other solutions are standalone networking libraries, and the cost of services is separate.

Abbildung 3.1: Gegenüberstellung mehrerer Multiplayer Frameworks. Aus (House, 2020)

.

3.2 Ein Spiel namens Scoo-King und die Wahl des richtigen Frameworks

Scoo-King ist ein Multiplayer-Spiel, das ursprünglich aus dem studentischen Abschlussprojekt des Medieninformatik-Studiengangs 2018 der Hochschule Mittweida hervorging. Die Entwicklung und Konzeptionierung des Spiel fand im Laufe eines halben Jahres inmitten der Covid-19 Pandemie statt und wurde während der „Beta 2021“ erstmals öffentlich vorgestellt. (Hammer, 2021)

Scoo-King ist ein kompetitives Top-Down 2v2-Party-Game, bei dem zwei Teams aus zwei Spielern in einer wilden Essensschlacht gegeneinander antreten. Mithilfe von Zutaten, die sie auf der Map verteilt finden, bereiten sie allerhand merkwürdige Gerichte zu, mit denen dann die gegnerische Küche beschossen wird. Wird die Küche eines Teams vollständig zerstört, endet das Spiel und das Team mit einer noch intakten Küche gewinnt. Um das gegnerische Team an ihrem Vorhaben zu hindern, haben Spieler die Möglichkeit, das gegnerische Team zu rammen, zu betäuben und ihnen ihre Items zu stehlen. Zusätzlich werden durch den Einsatz der verschiedenen Gerichte unterschiedliche Spezialfähigkeiten freigesetzt, die für wildes Chaos während des Spielverlaufs sorgen.

Abbildung 3.2: Logo des Spiels Scoo-King

In seiner ursprünglichen Form ist Scoo-King ein lokales Multiplayer-Spiel, das von vier Spielern vor demselben Computer gespielt wird. (Im Nachhinein betrachtet eine großartige Idee für ein Spiel während einer globalen Pandemie.) Doch damit endete Scoo-Kings Entwicklung nicht. In den nachfolgenden Monaten erklärten sich einige der ursprünglichen Entwickler dazu bereit, in ihrer Freizeit weiter an dem Spiel zu arbeiten und eine schlussendliche Veröffentlichung von Scoo-King auf der Online-Spiele-Plattform „Steam“ anzustreben. Doch der Umfang der ursprünglichen Idee wuchs. Was mit einigen einfachen Bug-Fixes anfing, entwickelte sich schnell zu einem Redesign einzelner Spielmechaniken und der Aufrüstung des Spiels zu einem Online-Multiplayer-Spiel. Letztere Aufgabe fiel mir zu. Seitdem befindet sich Scoo-King in einem umfassenden Rework hin zum Netzwerk-Spiel.

Abbildung 3.3: Screenshot der Scoo-King-Version für die „Beta 2021“

Eine der ersten Fragen, die sich bei der Umsetzung eines solchen Vorhabens stellt, ist die nach dem richtigen Framework für das jeweilige Spiel. Für Scoo-King gab es hier einige entscheidende Vorüberlegungen:

Zuerst einmal wäre da eine ganz pragmatische Kostenüberlegung. Scoo-King ist ein Studentenprojekt und soll nach bisherigem Plan kostenlos veröffentlicht werden. Das heißt, dass das Spiel keinerlei Profite generieren wird. Sollte die Wahl also auf eine Server-Client-Architektur fallen, würde daraus folgend das Problem entstehen, wie die für das Spiel notwendigen Server gehostet werden sollen. Die klassische Variante wäre, dass der Entwickler die zum Spielen notwendigen Server selbst hostet oder bei einem

Cloud-Dienstleister die notwendigen Server mietet. Dabei entstehen laufende Kosten und ohne Einnahmen, die diese Kosten decken könnten, schließt sich diese Möglichkeit daher von selbst aus. Die einzig andere Möglichkeit, dennoch mit einer Server-Client-Architektur zu arbeiten, wäre, die notwendigen Dateien zum Hosten eines Scoo-King-Servers zu veröffentlichen und die Server von Freiwilligen aus der Spielerschaft hosten zu lassen. Dies ist jedoch weder garantiert noch besonders nutzerfreundlich und würde vermutlich die Mehrheit der potentiellen Spieler davon abschrecken, das Spiel zu testen.

Somit bleibt nur die Möglichkeit, dass Scoo-King mit einer Peer-to-Peer-Architektur (P2P-Architektur) arbeitet. Allein dadurch scheiden viele der Frameworks aus. Übrig bleiben Netcode for GameObjects, Mirror und Photon PUN. Da Photon PUN allerdings ab einer gewissen Anzahl gleichzeitiger Spieler ebenfalls Kosten verursacht, gäbe es bereits bei mehr als fünf gleichzeitig geöffneten Lobbies wieder Probleme.

Auch die Art des Spiels ist ein wichtiger Faktor für die Wahl des richtigen Frameworks. Scoo-King ist ein kleines Party-Spiel mit lediglich vier Spielern. Die Netzwerk-Auslastung des Spiels wird also voraussichtlich sehr gering ausfallen und benötigt kaum Skalierbarkeit. Auch die Tatsache, dass die Spieler eines Party-Games aller Wahrscheinlichkeit nach befreundet seien werden, stellt einen Faktor dar, der sich für die Entscheidung nutzen lässt. Denn diese Tatsache verringert das Betrugsrisiko zwischen den Spielern ungemein. Sollte ein Spieler dennoch mit Cheats spielen, hätte dies im Normalfall immense soziale Folgen für ihn innerhalb seiner Freundesgruppe und stellt deshalb kaum einen Reiz dar. Dies heißt nun für die Entwicklung von Scoo-King, dass Sicherheit nicht an erster Stelle stehen muss und dass man beispielsweise der Performance des Spiels zuweilen den Vortritt lassen kann.

Unter diesen Gesichtspunkten sollte die Wahl also auf Netcode for GameObjects oder Mirror fallen. Für die finale Entscheidung war hier schlicht und einfach die Zugänglichkeit des Frameworks für neue Entwickler entscheidend. Zum Zeitpunkt der Entscheidung steckte Netcode for GameObjects immer noch im Entwicklungsprozess. Auch verfügte es damals noch über keine stabile Release-Version. Diese Tatsachen, gepaart mit einer im Vergleich zu Mirror deutlich geringeren Menge an Ressourcen zum Erlernen des Frameworks, führten schlussendlich dazu, dass die Wahl auf Mirror fiel.

3.3 Mirror – Ein Multiplayer-Framework für Unity

Mit der getroffenen Entscheidung für ein konkretes Framework besteht der nächste Schritt nun in der genaueren Betrachtung, wie das gewählte Framework funktioniert. Mirror bietet eine Vielzahl von fertigen Komponenten und Tools zur Implementierung von eigenem Netcode. Diese sollen nun im folgenden Kapitel genauer betrachtet werden.

Mirror arbeitet grundsätzlich mit einer Server-Client-Architektur, bietet jedoch die Möglichkeit, eine Host-Client-Architektur zu verwenden, bei der einer der Clients gleichzeitig als Server für die anderen Clients fungiert. Im Gegensatz zu anderen Frameworks arbeitet Mirror dabei nicht mit unterschiedlichen Builds für Server und Client oder konditioneller Kompilierung. Stattdessen wird derselbe Netcode auf allen Geräten ausgeführt und prüft, wenn nötig zur Laufzeit mittels Statusvariablen, die Rolle der eigenen Spielinstanz. Dadurch soll nicht nur der Entwicklungsprozess von Netzwerk-Spielen mit Mirror vereinfacht werden, sondern auch die Arbeit an einem einzelnen Projektstand ermöglicht werden. (Engelbrecht, 2022, S. 195–199)

3.3.1 Mirrors Netzwerk-Komponenten

Eine der wichtigsten Komponenten in Mirror ist der NetworkManager. Der NetworkManager ist verantwortlich für das Hosten eines Mirror-Servers oder das Herstellen einer Verbindung zu einem Mirror-Server. Darüber hinaus bietet er eine breite Vielzahl von Konfigurationseinstellungen. Zu diesen gehören die maximale Anzahl an Spielern, die gleichzeitig mit dem Server verbunden seien dürfen, die Netzwerk-Adresse des Servers, eine Autostart-Möglichkeit für reine Serverbuilds und die Auswahl der zu verwendenden Transport- und Authentifizierungs-Diensten. (Engelbrecht, 2022, S.68) Denn diese können beliebig ausgetauscht werden, sogar zur Laufzeit. Durch eine strikte Separierung von Netcode und Transportschicht bietet Mirror somit große Flexibilität und Ausfall-Sicherheit. Während Mirror selbst für die High-Level-Netzwerkfunktionalität verantwortlich ist, händelt die Transportschicht die eigentliche Netzwerk-Kommunikation. Diese Abstraktion ist einer der Gründe, warum Mirror so mächtig ist. Sie ermöglicht es, verschiedene Transportschichten basierend auf den Bedürfnissen des Spiels zu verwenden. So erlaubt die Verwendung der Transportschicht FizzySteamworks beispielsweise eine nahtlose Anbindung an die Matchmaking-Services der Spieleplattform Steam. Mit deren Hilfe können Schwierigkeiten wie der Austausch von IP-Adressen und NAT-Hole-Punching komplett umgangen werden, solange man die Steam-ID des anderen Spielers als Netzwerk-Adresse angibt. Sollte Steam dann zur Laufzeit nicht verfügbar sein, kann problemlos eine andere Transportschicht wie Mirrors Standard-KCP-Transportschicht als Redundanz eingewechselt werden, ohne dass dabei Änderungen am Netcode des Spiels nötig wären.

Sobald eine Verbindung zwischen mehreren Spielinstanzen besteht, bietet Mirror bereits eine ganze Reihe an fertigen Komponenten, um die Funktionalität von Unitys Kernkomponenten netzwerkfähig zu machen. So ermöglichen NetworkTransforms die automatische Synchronisation von Position, Rotation und Skalierung eines GameObjects. NetworkRigidbodys sorgen für netzwerktaugliche Kollisionserkennung, NetworkAnimators für synchronisierte Animationen auf allen Clients und so weiter. (vis2k, 2023)

3.3.2 NetworkIdentities

Im Generellen benötigt jedes GameObject, das Netzwerk-Komponenten von Mirror benutzen möchte, eine NetworkIdentity. Diese spezielle Komponente erlaubt es Mirror ein GameObject in einer Szene zu serialisieren und auf verschiedenen Clients zu synchronisieren. In den allermeisten Fällen fügt Unity sogar automatisch eine NetworkIdentity hinzu, wenn ein GameObject eine Netzwerk-Komponente verwenden möchte. Dennoch gibt es hier einige Limitationen, die man als Entwickler im Kopf behalten sollte. So müssen beispielsweise Prefabs, die eine NetworkIdentity besitzen und zur Laufzeit gespawnt werden sollen, vorher im NetworkManager registriert werden. Dies kann sowohl ganz einfach im Unity-Editor getan werden oder aber auch dynamisch während der Laufzeit aus dem Code heraus. Ist ein solches Prefab nicht registriert, bevor es gespawnt wird, wird die Anwendung zu diesem Zeitpunkt einen Fehler werfen und den Spawn-Prozess abbrechen.

Die wichtigste Einschränkung bei der Verwendung von NetworkIdentities besteht jedoch in der Tatsache, dass diese nicht verschachtelt werden dürfen. Es ist daher ratsam, alle Komponenten, die eine NetworkIdentity benötigen, stets auf dem Wurzel-Objekt eines GameObjects anzusiedeln. Sollte dies nicht der Fall sein, wird Mirror ausschließlich mit der ersten NetworkIdentity in einer Hierachie arbeiten und alle weiteren Netzwerk-Komponenten auf Kind-Objekten dieses Wurzel-Objekts deaktivieren. Diese Eigenheit von Mirror kann zuweilen etwas lästig seien, insbesondere wenn man Mechaniken wie das Aufheben und Herumtragen von Items, die selbst eine NetworkIdentity benötigen, implementieren möchte. Die offizielle Dokumentation von Mirror empfiehlt an dieser Stelle, das ursprüngliche Item zu entsorgen und stattdessen eine visuelle Attrappe zu spawnen, die der Spieler dann aufheben kann. Speichert das Item selbst wichtige Zustandsinformationen, müssen diese in der Attrappe zwischengespeichert werden, bis beim Ablegen des Items das Original-Objekt wieder gespawnt werden kann. (vis2k, 2023) Dieser ganze Vorgang ist recht umständlich und aufwendig zu implementieren. Daher sei hier der Tipp gegeben, dass alternativ das Austauschen des Items durch eine Attrappe vermieden werden kann, wenn man auf echtes Parenting verzichtet und stattdessen ein Script des Spielers einfach Position und Rotation des zu tragenden Items konstant anpasst, solange es gehalten wird. Dies mag weniger performant sein, ist jedoch in der Implementierung deutlich schneller umgesetzt.

3.3.3 Autorität

Ein wichtiges Konzept, das Mirror bei der Verwaltung von Netzwerk-Objekten einsetzt, ist das Konzept der Autorität. (Anmerkung: In den neusten Versionen seit Mirror v68.0.0 wurde dieses Konzept in Ownership, also Besitz über ein GameObject, umbenannt.) Dabei wird jeder NetworkIdentity entweder ein Client oder der Server zugewiesen, der die Autorität über dieses Objekt besitzt. Standardmäßig obliegt die Autorität über ein jedes Objekt immer dem Server, mit Ausnahme der individuellen Spieler-Objekte, die ihren jeweiligen Clients gehören. Der Server kann dann Autorität über individuelle Objekte an einzelne Clients vergeben und sie damit für das Objekt verantwortlich machen. Nur ein Client mit der Autorität für ein Objekt ist in der Lage, Änderungen an diesem Objekt vorzunehmen und es zu kontrollieren. Dadurch stellt Mirror sicher, dass es zu jedem Zeitpunkt für jedes Objekt eine klare Entscheidungshoheit gibt und keine zwei Clients gleichzeitig versuchen können, dasselbe Objekt zu manipulieren. (Engelbrecht, 2022, S. 99–100) Als Faust-Regel gilt dabei, dass jedes Objekt, das Teil der Map ist, immer dem Server gehört, dass die Spieler-Objekte ihren jeweiligen Clients gehören und dass für sonstige Items die Autorität mit dem Besitz durch einen Spieler vergeben wird.

3.3.4 NetworkBehaviours

Für die Implementierung von eigenem Netcode ist es am einfachsten, das Script, statt wie üblich in Unity von MonoBehaviour erben zu lassen, von Mirrors NetworkBehaviour zu erben. Dies bietet die gleichen Möglichkeiten wie das Erben von MonoBehaviour plus einige zusätzliche Optionen. Dazu zählen Zugang zur NetworkIdentity des Objekts, Statusvariablen wie „isServer“, „isClient“, „isLocalPlayer“ oder „hasAuthority“ (seit Mirror v68.0.0 „isOwned“) und eine Reihe von Callbacks für diverse Netzwerk-Events. Des Weiteren steht Kind-Klassen von NetworkBehaviour die Verwendung von Remote Procedure Calls (RPCs) und SyncVars zur Verfügung, die womöglich die wichtigsten Tools zur Implementierung von eigenem Netcode mit Mirror darstellen.

3.3.5 Remote Procedure Calls

Remote Procedure Calls (kurz RPCs) sind ein gängiges Werkzeug zur Implementierung von High-Level-Netcode in verschiedensten Frameworks. Zu diesen zählt auch Mirror. Die grundlegende Idee von RPCs ist dabei die Implementierung einer Funktion, die auf einem Computer aufgerufen werden kann, jedoch auf einem anderen Computer ausgeführt wird. RPCs treten in der Regel in einer von zwei Hauptvarianten auf: Server-RPCs oder Client-RPCs. Ein Server-RPC, oder Command, wie diese in Mirror genannt werden, ist eine Funktion, die auf einem Client aufgerufen und dann auf dem Server ausgeführt wird. Gegenteilig dazu sind Client-RPCs Funktionen, die vom Server aufge-

rufen werden können und dann auf den Clients ausgeführt werden. Mirror nimmt des Weiteren eine Unterscheidung vor zwischen Client-RPCs, die auf allen Clients ausgeführt werden, und Target-RPCs, die nur auf ausgewählten Clients ausgeführt werden. Die Deklaration eines RPCs erfolgt in Mirror mittels der jeweiligen Funktions-Attribute: „[Command]“, „[ClientRpc]“ oder „[TargetRpc]“. RPCs können unbegrenzt viele Parameter deklarieren. Allerdings muss jeder Datentyp, der dabei verwendet werden soll, durch Mirror serialisierbar sein. Die Möglichkeit eines Rückgabewertes besteht nicht. (vis2k, 2023)

3.3.6 SyncVars

Ein weiteres wichtiges Werkzeug in Mirror sind SyncVars, kurz für „Synchronized Variables“. Dabei handelt es sich um Variablen, die von Mirror, wenn sie auf dem Server geändert werden, automatisch auf allen Clients synchronisiert werden. Zuweisungen der Variable durch einen Client ist bei SyncVars nicht möglich und hat keinen Effekt. Mirror erlaubt es, jede Klassen-Variable eines NetworkBehaviour mit dem „[SyncVar]“-Attribut zu kennzeichnen. Die einzige Voraussetzung ist, dass der Datentyp der Variable durch Mirror serialisierbar sein muss.

Zusätzlich bietet das Framework die Option, einen Callback zu hinterlegen. Die hier angegebene Funktion wird dann automatisch auf allen Instanzen aufgerufen, sobald sich der Wert der Variable ändert. Wichtig zu wissen ist außerdem, dass die Synchronisierung der Variablen nicht augenblicklich erfolgt, sondern aus Gründen der Performance in regelmäßigen Intervallen. Die Länge dieser Intervalle kann für jede NetworkIdentity im Unity-Editor individuell angepasst werden.

3.3.7 Network Messages

Die letzte wichtige Form der Datenübertragung mit Mirror findet sich in der Verwendung von selbstdefinierten Network Messages. Network Messages sind die freiste Form der Datenübertragung in Mirror. Sie geben dem Entwickler volle Kontrolle über die zu versendenden Informationen sowie auch über Art und Weise, wie auf eingehende Network Messages reagiert werden soll. Weiterhin können Network Messages von überall im Code verwendet werden und benötigen nicht die zugrundeliegende Architektur eines NetworkBehaviours. Dafür erfordert ihre Verwendung jedoch die meiste Vorarbeit. Um eine eigene Network Message zu benutzen, muss zuerst ein Struct mit Variablen für alle zu versendenden Daten definiert werden. Auch hier wieder müssen alle dabei verwendeten Datentypen durch Mirror serialisierbar sein. Außerdem muss das Struct Mirrors „Network Message“-Interface implementieren. Beim Versenden einer Network Message dieses Typs wird dann eine neue Instanz des Structs erzeugt und mit den richtigen Werten versehen. Diese kann dann über Mirrors NetworkClient an den Server oder über Mirrors NetworkServer an alle Clients verschickt werden. Damit Mirror weiß, wie es auf

den Eingang einer Network Message dieses Typs reagieren soll, müssen vor der ersten Verwendung entsprechende Funktionen zum Bearbeiten der Übertragung registriert werden.

3.3.8 Serialisierung

Um einen Datentyp über ein Netzwerk versenden zu können, muss der fragliche Datentyp serialisierbar sein. Dabei reicht es jedoch nicht immer, auf C#'s oder Unitys Serialisierung von Datentypen zu hoffen. Denn auch, wenn bereits eine Serialisierung des Typs existiert, muss Mirror darüber Bescheid wissen. Glücklicherweise ist Mirror bereits von Haus aus in der Lage, den Großteil aller Datentypen, die einem Unity-Entwickler bei seiner Arbeit begegnen könnten, serialisieren zu können. Dazu gehören nicht nur alle elementaren Datentypen, wie Integer, Booleans oder Strings, sondern auch eine Vielzahl von wichtigen Unity-Datentypen, wie beispielsweise Vector3, Quaternion, GameObject oder Transform. Soweit ist das für viele Multiplayer-Frameworks nicht ungewöhnlich. Wodurch Mirror jedoch hervorsticht, ist die Verwendung von Reflection zur Serialisierung von unbekannten Datentypen. Auf diese Weise ist Mirror in der Lage, auch die meisten Daten-Klassen, Structs, Enums und Arrays automatisch zu serialisieren, selbst wenn diese zuvor komplett unbekannt sind. Wenn die Serialisierung für einen speziellen Datentyp dennoch nicht möglich ist oder Mirrors automatische Serialisierung nicht dem gewünschten Verhalten entspricht, können eigene Implementationen hinterlegt werden. Dies kann durch die Definition von Extension Methods für Mirrors NetworkWriter- und NetworkReader-Klasse zur Verarbeitung des gefragten Datentyps erreicht werden. (vis2k, 2023)

4 Entwurf eines modularen, netzwerkfähigen Systems für Scoo-King

Die Entwicklung von Computerspielen bringt häufig sehr komplexe Softwaresysteme hervor. Aufgrund der Vielzahl von Features und der immer präsenten Interaktivität dieser Programme, welche die ständige Möglichkeit von Benutzereingaben birgt, entsteht eine Komplexität, die Entwickler geradezu einlädt, mögliche Bugs zu übersehen. Auch die sich immer wieder wandelnden Anforderungen an einzelne Komponenten der Software durch Wechsel in der Design-Philosophie des Spiels oder Feedback aus der Community sorgen für stetiges Wachstum und Wandel im Quellcode dieser Anwendungen.

Aus diesem Grund werden häufig Wartbarkeit und Ausbaufähigkeit der Software an erster Stelle geschrieben. Optimierungen zugunsten der Performance, die diesem Prozess oft zuwider stehen, werden in der Regel erst vorgenommen, wenn sie auch notwendig sind. Achtung, dies heißt keinesfalls, dass Programmierer beim Entwickeln von Computerspielen nicht auf Performance achten müssen; lediglich, dass andere Aspekte der Software-Architektur zuweilen Vorrang haben.

Einen guten Ansatz für dieses Vorhaben stellt die Modularisierung der Software dar. Einzelne Features der Software werden in separate Systeme aufgeteilt, die mit nur minimaler Interaktion untereinander ihre jeweilige Funktionalität für den Rest der Anwendung bereitstellen. Durch diese Abkapslung entsteht eine klare Trennung zwischen unterschiedlichen Mechaniken des Spiels und im selben Zug wird auch das Testen neuer Systeme vereinfacht.

Bei der Entwicklung eines Multiplayer-Spiels steigt die Komplexität nun weiter durch die Notwendigkeit von Netcode. Anders als bei einem Singleplayer-Spiel muss die Logik eines Systems nicht mehr nur auf einem Computer ausgeführt werden, sondern alle wichtigen Zustandsänderungen müssen auf allen anderen Clients der Sitzung repliziert werden. Doch wie könnte der Aufbau eines solchen modularen Systems aussehen? Recherchiert man zu diesem Thema, findet man zwar viel Literatur über den grundlegenden Aufbau von Netzwerk-Spielen, kaum etwas jedoch zu der Architektur von High-Level Netcode zum Zwecke der tatsächlichen Implementierung von Spielmechaniken. Das folgende Kapitel widmet sich daher vier verschiedenen Ansätzen zur Implementierung eines Netzwerkmoduls zu diesem Zweck:

1. Selbstverwaltung mittels RPCs
Mittels der Nutzung von Remote Procedure Calls kann der Aufruf einer Funktion auf allen Clients einer Sitzung gewährleistet werden. Wird die Ausführung der Logik einer Funktion den einzelnen Clients selbst überlassen, müssen diese lediglich mit den übergebenen Argumenten, die für die korrekte Ausführung der Funktion notwendig sind,

versorgt werden. Diese werden via der RPCs an alle Clients versendet, sodass diese die Logik dann lokal ausführen und daraus entstandene Resultate direkt vor Ort selbst verwalten können.

2. Autoritäre RPCs
Eine alternative Verwendung von Remote Procedure Calls stellt die Errichtung einer absoluten, serverseitigen Autorität dar. Jeder Aufruf eines Clients wird hierbei mittels RPCs an den Server weitergeleitet. Allein der Server ist dann berechtigt, die Funktion entweder wie angefordert auszuführen oder deren Exekution zu verweigern. Die aus der Ausführung resultierenden Zustandsänderungen werden dann an alle Clients gesendet, um sie replizieren zu können. Die Clients bilden dabei lediglich die Änderungen ab, ohne jedoch sie selbst zu berechnen.

3. Bereitstellung von Informationen via SyncVars
Wenn das Modul ohnehin eine Datenstruktur zur Speicherung wichtiger Informationen nutzt, können diese um Konfigurationsdaten zur Ausführung der Logik des Moduls angereichert werden. Die verwendete Datenstruktur kann dann zur SyncVar erklärt werden und zentral für alle Clients vom Server aus verwaltet werden. Dadurch entsteht eine Mischung, bei der der Server einerseits die Anweisungen gibt und die notwendigen Daten zur Verfügung stellt. Andererseits reagieren die Clients auf diese Änderungen und führen die gefragten Funktionen bei sich lokal aus.

4. Selbstverwaltung mittels eigen definierter Network Messages
Wenn keiner der anderen Ansätze anwendbar ist, weil es sich zum Beispiel um ein statisches System handelt, können auch eigene Network Messages definiert und versendet werden. Ähnlich dem ersten Ansatz werden dabei die zur Ausführung notwendigen Daten einer Funktion an alle Clients verschickt und dann vor Ort ausgeführt. Das Modul kümmert sich dann lokal um die Verwaltung der Resultate, ist dabei aber nicht an ein NetworkBehaviour gebunden.

Für eine beispielhafte Implementierung dient hier das Partikelsystem aus Scoo-King. Scoo-King verwendet eine Host-Client-Architektur, bei der einer der Clients zusätzlich den Server für die restlichen Teilnehmer der Sitzung spielt. Da Scoo-King in seiner ursprünglichen Konzeption für die „Beta 2021“ als lokaler Multiplayer erdacht war, soll diese Option weiterhin erhalten bleiben. Weil die Architektur eines lokalen Muliplayer-Spiels jedoch eher der eines Singleplayer-Spiels ähnelt, wäre in großen Teilen des Spiels eine doppelte Implementierung der verschiedenen Systeme notwendig. Dies wäre nicht nur unpraktisch, sondern auch ein Albtraum für die Wartbarkeit des Spiels. Scoo-King umgeht daher dieses Problem, indem es auch eine lokale Sitzung als Netzwerk-Spiel begreift. Jedoch erlaubt das Spiel in diesem Modus nur einen einzigen Client und bietet keine Möglichkeit, von außen beizutreten. Dieser Trick ermöglicht eine durchgängige Architektur als Netzwerk-Spiel und vereinheitlicht das Design seiner Systeme.

Das Partikelsystem des Spiels ist dabei ein exzellentes Beispiel für den Aufbau eines solchen Netzwerkmoduls. Nach außen sichtbar sind nur einige wenige Funktionen, um Partikeleffekte zu erstellen, zu löschen oder zu modifizieren. Dabei verwendet das System einen Namen in Form eines Strings in Kombination mit einem im Unity-Editor vordefinierten Look-Up, um spezifische Partikeleffekte zu adressieren. Effekte sind hierbei als Prefab in Unity vordefinierte und gespeicherte GameObjects, die auf Bedarf in der Szene gespawnt werden können. Mithilfe von optionalen Parametern kann man das genaue Verhalten einer Funktion zusätzlich noch weiter modifizieren.

4.1 Selbstverwaltung mittels RPCs

Eines der am häufigsten zu findenden Features in Netzwerk-Frameworks sind Remote Procedure Calls. Hierbei handelt es sich um Funktionen, die, wenn sie aufgerufen werden, nicht auf der lokalen Maschine sondern auf einem anderen Client ausgeführt werden. In Mirror deklariert man eine Funktion als RPC, indem man diese mittels eines Funktions-Attributs kennzeichnet.

Das Konzept der Selbstverwaltung mittels RPCs setzt auf die lokale Ausführung der eigentlichen Logik einer Funktion auf den jeweiligen Clients. Zu diesem Zweck werden alle betroffenen Clients mittels RPCs mit den notwendigen Aufrufparametern der Funktion versorgt. Interne Zustände werden nicht über das Netzwerk kommuniziert sondern lokal abgebildet. Dieses Verfahren birgt sowohl Vor- als auch Nachteile.

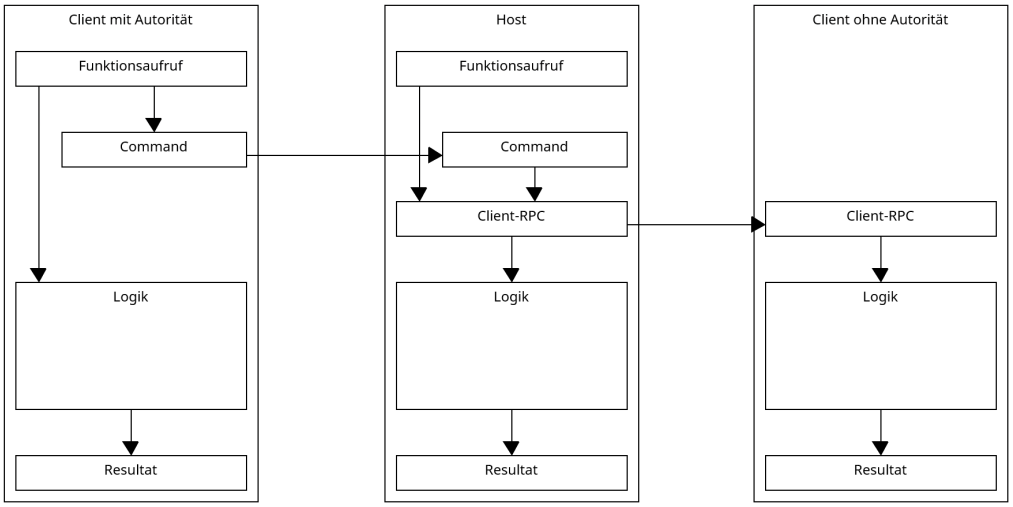

Abbildung 4.1: Aufbau eines selbstverwaltenden Netzwerkmoduls mit RPCs

.

Positiv ist hier insbesondere die schnelle Reaktionszeit zwischen Aufruf der Funktion und ihrer tatsächlichen Ausführung. Ein Client, der die Autorität über eine Modul-Instanz besitzt, kann die angeforderte Logik sofort realisieren, noch bevor er irgendwelche Netzwerk-Aufrufe machen muss. Des Weiteren müssen etwaige von diesem Modul kontrollierte Objekte, wie beispielsweise die von dem Modul kontrollierten Partikeleffekte im unten stehenden Code, selbst nicht über Network Identities verfügen, was die Implementierung und den Einsatz solcher Systeme vereinfacht.

Andererseits bietet das Fehlen einer absoluten Autorität und der Mangel an Kontrollmechanismen zur Überprüfung einer korrekten Ausführung der Logik auf allen Clients auch Sicherheitsrisiken und Potenzial für mögliche Betrugsversuche einzelner Spieler.

Nach außen sind lediglich die Funktionen des Moduls sichtbar, die notwendig sind, um es anzusteuern. Jegliche interne Logik ist entweder „private“ oder „protected“. Wird eine der öffentlichen Funktionen aufgerufen, gibt es aus netzwerktechnischer Sicht drei mögliche Zustände:

1. Die Funktion wurde direkt auf dem Server aufgerufen. In diesem Fall sollte die Funktion immer ausgeführt werden. Auch dann wenn das Modul selbst keiner strengen Server-Autorität unterliegt, hat der Server am Ende des Tages immer das letzte Wort und muss mindestens die Möglichkeit haben, von außen selber Funktionen des Moduls aufzurufen.

2. Die Funktion wurde von einem Client aufgerufen, der die Autorität über das Modul besitzt. Relativ häufig kann es sinnvoll sein, einem Client die Autorität für eine Modul-Instanz zu übergeben. Dies ist besonders der Fall für Systeme, die direkt mit einem Spieler interagieren oder sich anderweitig unter direkter Kontrolle eines Spielers befinden. Auch in diesem Fall sollte der Aufruf der Funktion zugelassen werden und der Server zusätzlich über den Aufruf informiert werden.

3. Die Funktion wurde von einem Client ohne Autorität für das Modul aufgerufen. Während es Fälle geben mag, in denen jeder Client auf ein Modul zugreifen darf, ist dies in den meisten Situationen eher ein Sicherheitsrisiko als ein tatsächlicher Vorteil für die Architektur der Software. In der Regel werden Aufrufe dieser Art einfach abgebrochen und geben falls notwendig einen negativen Rückgabewert zurück.

Mittels Guard-Clauses können alle Fälle, die nicht zu einem Aufruf der Funktion führen sollen, herausgefiltert werden. Auch andere Kriterien, die eventuell zu beachten sind, können an dieser Stelle geprüft werden. Anschließend folgt der eigentliche Funktionsaufruf.

Abhängig von welchem der drei oben genannten Fälle die Funktion aufgerufen wird, muss mit unterschiedlichen Ausführungspfaden reagiert werden. Die eigentliche Logik der Funktion ist aus diesem Grund in eine private Unterfunktion ausgelagert, die hier im Beispiel mit dem Namenszusatz „Internal“ gekennzeichnet wird. Diese Unterfunktion soll auf jedem Client ausgeführt werden. Um dies zu erreichen, sendet der Server einen Client-RPC an alle Clients. Kam der Aufruf ursprünglich vom Server, so kann die Ausführung direkt geschehen. Wenn nicht und der Aufruf kam von einem Client, muss dieser zuerst einen „Command“ oder auch Server-RPC verschicken, so dass der Server dann im nächsten Schritt alle restlichen Clients informieren kann.

Dieser Ablauf allein würde bereits funktionieren, führt aber noch zu einem großen Input-Lag zwischen Aufruf der Funktion und ihrer tatsächlichen Ausführung, da ein Client erst den Server benachrichtigen muss, welcher dann wiederum den Befehl zur Ausführung zurücksendet. Da jede Form der Netzwerkkommunikation immer mit zeitlichen Verzögerungen einher geht, treten hier selbst bei einer guten Internetverbindung mindestens ein paar Millisekunden Verzögerung auf. Da die Logik jedoch ohnehin lokal ausgeführt wird, kann dieser Effekt vermieden werden. Hierzu wird dem originalen Client erlaubt, seinen internen Funktionsaufruf sofort ohne Rückmeldung des Servers zu tätigen.

```
public bool DestroyParticle(string name) {

    if (!isServer && !hasAuthority)
        return false;

    bool result = DestroyParticleInternal(name);

    if (!isServer) Cmd_DestroyParticle(localClientID, name);
    else RPC_DestroyParticle(localClientID, name);

    return result;
}
```

Um zu vermeiden, dass die Funktion ein weiteres Mal ausgeführt wird, wenn der Client-RPC vom Server eingeht, muss beiden RPC-Funktionen ein weiteres zusätzliches Argument mitgegeben werden: Eine Caller-ID, mit der der ursprüngliche Client identifiziert werden kann.

In Mirror tritt an dieser Stelle das Problem auf, dass Mirror keine universal einheitliche Client-ID zur Verfügung stellt, die hier verwendet werden könnte. Da Mirror jedoch davon ausgeht, dass jeder Client ein lokales Spieler-Objekt besitzt, kann stattdessen einfach die Netzwerk-ID des Spieler-Objekts des Clients verwendet werden, um den Client zu identifizieren. Alternativ können natürlich auch die IP-Adresse oder selbst definierte IDs diese Rolle einnehmen.

```
private int localClientID => (NetworkClient.localPlayer) ?
    (int)NetworkClient.localPlayer.netId : -1;
```

Wenn es nun um die Ausführung der Funktion durch einen eingehenden Client-RPC geht, kann einfach geprüft werden, ob die lokale Client-ID und die Caller-ID übereinstimmen, um einen erneuten Aufruf auf dem ursprünglichen Client zu verhindern.

Im hier aufgeführten Beispiel handelt es sich um eine Implementierung dieses Ansatzes für Scoo-Kings Partikelsystem. Dessen wichtigste Funktionen sind das Erstellen und das Löschen von Partikeleffekten. Da während des Erstellens von Partikeleffekten auf eine Vielzahl von optionalen Konfigurationsparametern eingegangen wird, soll für ein besseres Verständnis stattdessen die Funktion zum Löschen von Partikeleffekten hier als Beispiel dienen. Während der Exekution der Funktionslogik bleibt daher nur eine letzte Validitätsprüfung, ob der zu zerstörende Effekt dem System auch bekannt ist, bevor die tatsächliche Löschung und das Entfernen aus der Liste der aktiven Effekte folgen kann.

```
[Command]
private void Cmd_DestroyParticle(int callerID, string name) {
    RPC_DestroyParticle(callerID, name);
}

[ClientRpc]
private void RPC_DestroyParticle(int callerID, string name) {
    if (callerID != localClientID) DestroyParticleInternal(name);
}


private bool DestroyParticleInternal(string name) {

    if (!activeEffects.ContainsKey(name))
        return false;

    GameObject particleEffect = activeEffects[name];
    activeEffects.Remove(name);
    Destroy(particleEffect);

    return true;
}
```

Zusammenfassend werden in dieser Herangehensweise die für die Ausführung einer Funktion notwendigen Informationen mittels RPCs an alle Clients verschickt, damit diese dann vor Ort die Anfrage umsetzen können. Das Ergebnis der Umsetzung wird dabei nicht noch einmal explizit zwischen den Clients und dem Server kommuniziert.
Dadurch können vor allem auf dem Client, von dem die Anfrage ausgeht, eine schnelle Umsetzung und ein sehr geringer Input-Lag garantiert werden. Jedoch erfordert dieser Ansatz auch Vertrauen in die korrekte Umsetzung der Anweisung auf den Clients und bietet somit eine mögliche Anfälligkeit gegenüber Manipulationsversuchen.

4.2 Autoritäre RPCs

Eine andere Einsatzmöglichkeit von RPCs bietet sich in einem autoritäreren Ansatz. Statt den einzelnen Clients die Ausführung der Logik, die sie betrifft, selbst zu überlassen, ist in diesem Szenario ausschließlich der Server für die Exekution von Logik verantwortlich. Clients können unter diesen Voraussetzungen lediglich Anfragen zur Ausführung einer speziellen Server-Funktion stellen, auf die Ausführung selbst haben sie jedoch keinen Einfluss.

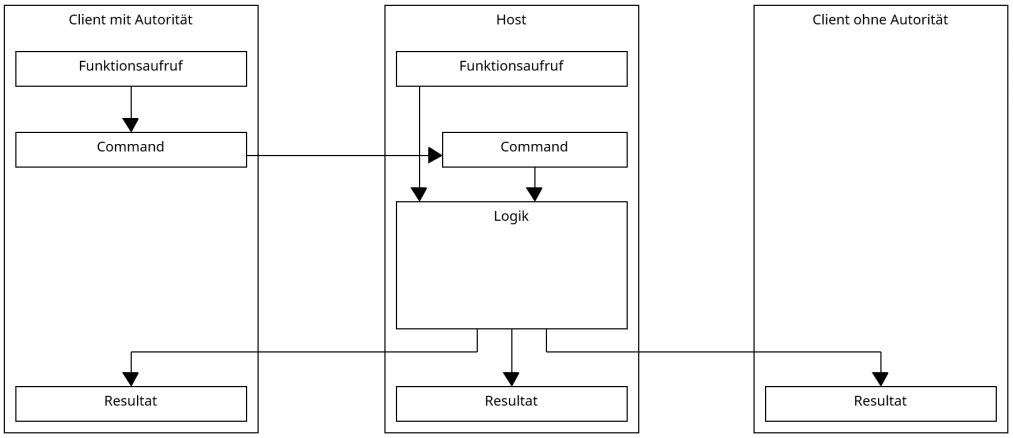

Abbildung 4.2: Aufbau eines autoritären Netzwerkmoduls mit RPCs

.

Dieses Verfahren führt zu längeren Reaktionszeiten zwischen Aufruf und Exekution einer Funktion, da zuerst ein Datentransfer mit der Anfrage zur Ausführung zum Server entsendet werden muss und im Anschluss eine weitere Übertragung mit den Resultaten der Exekution der Funktion vom Server zu den Clients stattfinden muss. Erst nachdem diese beiden Vorgänge abgeschlossen sind, kann der Client die Resultate bei sich lokal replizieren und dem Nutzer ein Ergebnis präsentieren.

Auch die Verteilung von Rechenzeit fällt hier sehr zu Lasten des Servers aus. In einer reinen Server-Client-Architektur kann dies ein Vorteil sein, da die Computer der Spieler weniger Leistung bringen müssen. Dadurch kann das Spiel mit geringeren Systemanforderungen werben oder die freie Rechenzeit beispielsweise in detailliertere Grafik investieren. In einer Host-Client-Architektur läuft der Server allerdings auf dem Computer eines Kunden, zusätzlich zu dessen Client, und kann zu sehr ungleichen Resultaten in der Performance des Spiels zwischen dem Host einer Lobby und den anderen Clients führen.

Der Vorteil in dieser Herangehensweise besteht in der Sicherheit gegenüber Betrugsversuchen. Im Falle, dass ein Client kompromittiert ist und versucht, das Spielgeschehen unerlaubt zu seinem Vorteil zu beeinflussen, kann dieser maximal eine Anfrage an den Server schicken, nicht jedoch selbst den Zustand des Systems verändern. Der Server dann hat die Autorität, die Anfrage zu prüfen und entweder die angeforderte Funktionalität auszuführen oder die Anfrage abzulehnen.

Wird von außen eine Funktion des Moduls aufgerufen, so folgt zunächst wieder eine Überprüfung, ob die Funktion auf dem Server oder einem der Clients aufgerufen wurde. Handelt es sich um einen Client, wird ebenfalls überprüft, ob dieser die Autorität für diese Modul-Instanz besitzt. Solange der Aufruf vom Server selbst kommt oder der Client vorher vom Server Autorität bekommen hat, soll der Aufruf zugelassen werden. Die eigentliche Logik der Funktion ist auch in diesem Ansatz in eine namensgleiche, private Unterfunktion mit dem Namenszusatz „Internal“ ausgelagert. Erfolgt der Aufruf direkt auf dem Server, so kann diese sofort aufgerufen werden. Wenn nicht, ist zuerst ein Command notwendig, der den Aufruf der Unterfunktion auf dem Server dann übernimmt.

```
public bool DestroyParticle(string name) {

    if (!isServer && !hasAuthority)
        return false;

    if (isServer) return DestroyParticleInternal(name);
    else Cmd_DestroyParticle(name);

    return true;
}


[Command]
private void Cmd_DestroyParticle(string name) =>
    DestroyParticleInternal(name);
```

Die Unterfunktion mit der tatsächlichen Logik wird des Weiteren mit einem Server-Attribut gekennzeichnet. Dadurch wird sicher gestellt, dass versehentliche oder auch bösartige Ausführungsversuche auf einem Client, der nicht der Host ist, sofort im Keim unterbunden werden. Je nachdem, für welche Funktionalität dieses Pattern implementiert wird, hat der Server nun die Möglichkeit, verschiedenste Validitätsprüfungen durchzuführen. Im Beispiel von Scoo-Kings Partikelsystem prüft der Server hier beispielsweise, ob ihm der Partikeleffekt, den er löschen soll, auch bekannt ist und ob er auf dem Server eine Instanz besitzt. Erst nachdem dies sichergestellt ist, führt er die Löschung aus und entfernt ihn aus der Liste der aktiven Effekte.

```
[Server]
private bool DestroyParticleInternal(string name) {

    if (!activeEffects.ContainsKey(name))
        return false;

    GameObject particleEffect = activeEffects[name];
    activeEffects.Remove(name);
    NetworkServer.Destroy(particleEffect);

    return true;
}
```

Selbstverständlich müssen die hierbei verursachten Änderungen ebenfalls auf allen Clients repliziert werden. Die Herangehensweisen zum Erreichen dieses Ziels variieren allerdings je nach Aufgabe des Moduls. Einfache Zustandsänderungen können mithilfe von SyncVars kommuniziert werden. Komplexere Logik muss stattdessen entweder auf die Verwendung von anderen bereits netzwerkfähigen Systemen setzen oder selbstständig Änderungen regelmäßig an alle Clients senden. Im Fall von Scoo-Kings Partikelsystem geht es hier explizit um das Spawnen und Despawnen von GameObjects. Daher kann hier recht einfach mit Mirrors bereits existierenden Systemen zum Verwalten von GameObjects gearbeitet werden.

Dies heißt allerdings, dass jedes GameObject, welches von diesem Modul verwaltet wird, eine NetworkIdentity besitzen muss und innerhalb von Mirrors NetworkManager registriert sein muss. Je nachdem, wie viele verschiedene Partikeleffekte das Spiel später verwenden soll, entsteht dadurch eventuell mehr Arbeit bei der Verwendung des Systems in Unitys Editor.

4.3 SyncVars

Bisher haben beide der zwei vorgestellten Ansätze mit Remote Procedure Calls gearbeitet, um Informationen zwischen verschiedenen Geräten auszutauschen. RPCs sind jedoch nicht die einzige Option in Mirror, um dieses Ziel zu erreichen. Eine weitere Möglichkeit bietet sich in der Verwendung von SyncVars. Dies sind Variablen, die sobald sie auf dem Server modifiziert werden, sämtliche Änderungen auf allen Clients korrekt abbilden.

Die Idee hier ist es, alle wichtigen Informationen für den Aufruf einer Funktion an den Server weiterzuleiten. Dieser stellt sie dann mittels einer SyncVar für alle Clients zur Verfügung. Die Clients ihrerseits werden dann bei einer Änderung informiert und sind in der Lage, die angefragte Logik bei sich lokal auszuführen.

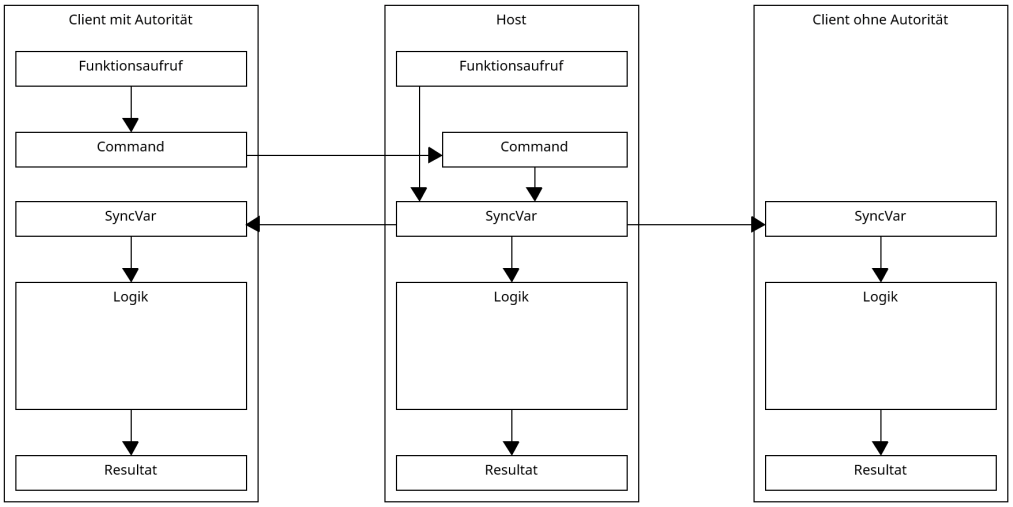

Abbildung 4.3: Aufbau eines Netzwerkmoduls mit SyncVars

.

Dabei behält der Server seine Funktion als autoritäre Kontrollinstanz und kann eine angeforderte Ausführung zu jedem Zeitpunkt blockieren, wenn sie nicht rechtmäßig ist. Gleichzeitig wird jedoch die Last der Rechenarbeit auf allen Clients gleichmäßig verteilt, so dass es beim Host nicht zu Engpässen kommt. Im Beispiel von Scoo-Kings Partikelsystem kann sogar eine in jedem Ansatz notwendige Datenstruktur zum Speichern der aktiven Effekte direkt in die SyncVar mit integriert werden.

Auf der anderen Seite werden SyncVars nur in regelmäßigen Abständen synchronisiert, wodurch es zu verhältnismäßig langen Reaktionszeiten zwischen Aufruf einer Funktion und deren tatsächlicher Ausführung auf dem Client kommen kann. Da die Synchronisierung von SyncVars nicht eventbasiert operiert, sondern mit zyklischen Updates arbeitet, variiert die Ausführungszeit stark. Sollte das betreffende oder ein darauf aufbauendes System also zeitliche Annahmen über die Ausführungszeit machen, kann es hier zu Problemen kommen.

Bei Aufruf einer Funktion dieses Moduls erfolgt auch hier wieder eine kurze Überprüfung der Legitimität desselben. Nachdem dies geklärt ist, ist es notwendig, dass die Daten vom Server aus in die SyncVar geschrieben werden. Erfolgt der Aufruf nicht auf dem Server, ist es daher notwendig, die Daten zuerst per Server-RPC zu versenden. Auf dem Server dann können die nötigen Änderungen am Inhalt der SyncVar vorgenommen werden.

```
public bool DestroyParticle(string name) {

    if (!isServer && !hasAuthority)
        return false;

    if (isServer) return Server_DestroyParticle(name);
    else Cmd_DestroyParticle(name);

    return true;
}


[Command]
private void Cmd_DestroyParticle(string name) =>
    Server_DestroyParticle(name);


[Server]
private bool Server_DestroyParticle(string name) {

    if (!activeEffects.ContainsKey(name))
        return false;

    activeEffects.Remove(name);
    return true;
}
```

Für die SyncVar an sich eignet sich in diesem Beispiel am besten ein Dictionary, welches das Partikelsystem ohnehin verwendet, um seine aktiven Effekte zu speichern. Ein generisches SyncDictionary erfüllt diese Rolle genau und sendet seinen Inhalt an alle Clients. Die im Dictionary verwendete Datenklasse speichert alle für das System wichtigen Konfigurationsdaten sowie eine Referenz auf die lokale Instanz des Partikeleffekts. Letztere wird nicht mit synchronisiert, sondern von jedem Client lokal zugewiesen.

```
private readonly SyncDictionary<string, ParticleConfig> activeEffects
    = new SyncDictionary<string, ParticleConfig>();


private class ParticleConfig {
    public string name;
    public bool destroyAfterFinished;
    public bool useThisTransform;
    public Transform alternativeParentTransform;
    public string useKey;

    public GameObject instance {
        get => effectStorage[instanceID];
        set {
            if (value != null) effectStorage[instanceID] = value;
            else effectStorage.Remove(instanceID);
        }
    }
    public Guid instanceID;
}


private static readonly Dictionary<Guid, GameObject> effectStorage
    = new Dictionary<Guid, GameObject>();
```

Die korrekten Informationen mittels SyncVar auf alle Clients zu synchronisieren, reicht aber natürlich noch nicht, um eine spezielle Aufgabe zu erfüllen. Dazu ist es notwendig, bei einer Änderung der Daten im SyncDictionary auf den Clients darauf zu reagieren. Mithilfe eines Callbacks lässt sich eine Funktion hinterlegen, die Mirror jedes mal aufrufen wird, wenn sich der Inhalt des SyncDictionarys ändert. Im Beispiel heißt diese Funktion „OnActiveEffectsChanged“ und wird während der Initialisierung des Systems dem Callback des SyncDictionarys hinzugefügt.

Die „OnActiveEffectsChanged“-Funktion ruft dann die entsprechende Unterfunktion auf, ausgehend von der Art der auf dem Dictionary ausgeführten Operation. Dort bleibt in diesem Fall nichts weiter zu tun, als die lokale Instanz des Partikeleffekts zu löschen und die dazugehörige Referenz aus dem Dictionary zu entfernen.

```
public override void OnStartClient() {

    activeEffects.Callback += OnActiveEffectsChanged;
}


private void OnActiveEffectsChanged(SyncDictionary<string,
    ParticleConfig>.Operation operation, string key,
    ParticleConfig particleConfig) {

    switch (operation) {

        case SyncIDictionary<string, ParticleConfig>
            .Operation.OP_ADD:
            CreateParticleInternal(key, particleConfig);
            break;

        case SyncIDictionary<string, ParticleConfig>
            .Operation.OP_REMOVE:
            DestroyParticleInternal(key, particleConfig);
            break;
    }
}


private void DestroyParticleInternal(string key,
    ParticleConfig particleConfig) {

    Destroy(particleConfig.instance);
    particleConfig.instance = null;
}
```

4.4 Selbstdefinierte Network Messages

Nicht immer hat man den Luxus, ein System innerhalb eines NetworkBehaviour entwerfen zu können. Sowohl Remote Procedure Calls als auch SyncVars erfordern in Mirror jedoch die Grundlage eines NetworkBehaviour, um zu funktionieren. Für diese Fälle gibt es dennoch eine Möglichkeit, Informationen über ein Netzwerk zu versenden: selbstdefinierte Network Messages.

Die Architektur dieses Ansatzes gleicht stark dem der Selbstverwaltung mithilfe von RPCs. Eingehende Aufrufe können direkt auf dem lokalen Client ausgeführt werden, was zu schnellstmöglichen Reaktionszeiten führt. Gleichzeitig wird der Server über den Aufruf informiert und leitet alle Information zur Ausführung der Funktion an alle Clients weiter. Nur der ursprüngliche Client wird an dieser Stelle übergangen. Er hat den Inhalt der Funktion bereits ausgeführt. Die eigentliche Logik der Funktion wird lokal von jedem Client einzeln exekutiert. Der einzige wirkliche Unterschied besteht darin, wie die Informationen im Netzwerk verschickt werden. Statt RPCs werden Network Messages genutzt.

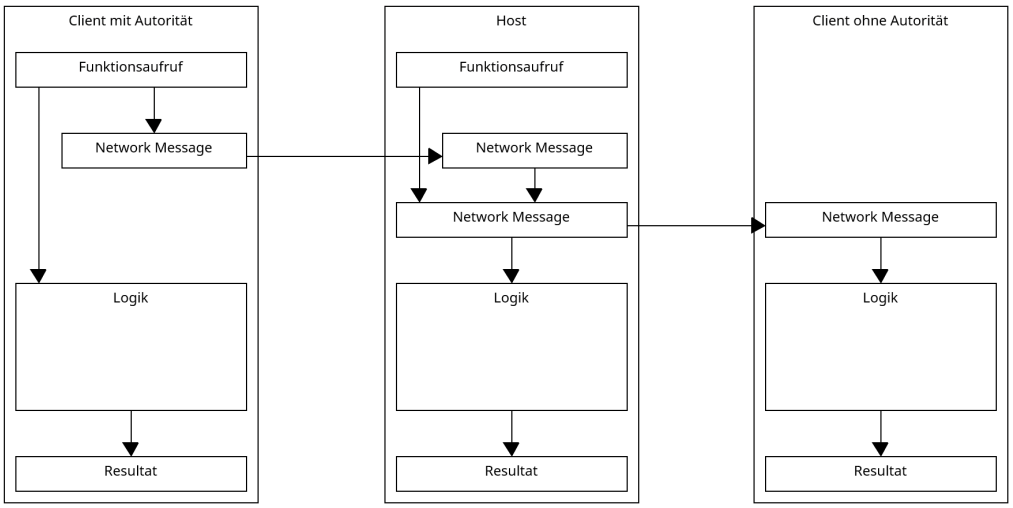

Abbildung 4.4: Aufbau eines selbstverwaltenden Netzwerkmoduls mit Network Messages

.

Aufgrund dieser Ähnlichkeit wirken auch die Vor- und Nachteile dieser Herangehensweise sehr vertraut. Bedingt durch die Tatsache, dass das System nicht erst auf eine Rückmeldung des Servers warten muss, liegt die Verzögerung zwischen dem Aufruf einer Funktion und deren tatsächlicher Ausführung bei einem absoluten Minimum. Im Ge-

genzug dazu besteht allerdings eine erhöhte Anfälligkeit gegenüber Betrugsversuchen. Sollte einer der Clients unzulässige Anfragen senden, hat der Server nur begrenzte Möglichkeiten, das zu kontrollieren und wenn nötig dagegen vorzugehen.

Anders als beim ersten Ansatz aber ist das System nicht auf die Präsenz eines zugrundeliegenden NetworkBehaviour zum Versenden der Daten angewiesen. Selbst statische Systeme lassen sich problemlos mittels Network Messages implementieren. Diese Freiheit hat dennoch ihren Preis. Während RPCs fast gänzlich ohne weitere Vorbereitung nutzbar sind, fällt diese bei Network Messages deutlich umfangreicher aus. Für jeden Anwendungsfall müssen eigene Nachrichten selbst definiert und angemeldet werden. Je nachdem, wie vielen Aufgaben ein solches Modul nachkommen soll, kann dabei einiges an zusätzlichem Code anfallen.

Zum Verwenden von Network Messages müssen diese zunächst definiert und registriert werden. Zu diesem Zweck wird ein Struct definiert, welches das „NetworkMessage“-Interface implementiert. Die einzelnen Variablen innerhalb des Structs repräsentieren dabei die Informationen, die während der Nutzung der Network Message versandt werden.

```
private static bool initialized = false;

private struct P_DestructionMessage : NetworkMessage {
    public string name;
    public int callerID;
    public ParticleComponent_NetMessage particleComponentInstance;
}

private static void Initialize() {

    if (initialized) return;

    if (NetworkServer.active) {
        NetworkServer.RegisterHandler
            <P_DestructionMessage>(OnDestroyParticleServer);
    }
    if (NetworkClient.active) {
        NetworkClient.RegisterHandler
            <P_DestructionMessage>(OnDestroyParticleClient);
    }

    initialized = true;
}
```

Dies allein erlaubt zwar bereits das Versenden der jeweiligen Daten, viel passieren wird jedoch noch nicht. Damit Mirror weiß, wie es mit einer Nachricht dieser Art umgehen soll, muss zuvor eine Funktion registriert werden, die den jeweiligen Nachrichtentyp bearbeiten kann. Zu diesem Zweck besitzt das Modul eine Initialisierungsfunktion, die irgendwann zu Beginn des Programms aufgerufen werden muss. Eine statische Variable kann gleichzeitig Auskunft darüber geben, ob das System bereits initialisiert wurde, wie auch die mehrfache Ausführung der Funktion verhindern.

Nachdem die Vorarbeit aus dem Weg geschafft ist, bleibt die eigentliche Aufgabe des Moduls. Bei Aufruf einer Funktion wird auch hier wieder zunächst die Rechtmäßigkeit der Ausführung geprüft. Im Anschluss erfolgt direkt die Exekution auf dem lokalen Client, wie auch im Ansatz der Selbstverwaltung mittels RPCs, bevor im nächsten Schritt auch die restlichen Clients informiert werden. Der größte Unterschied zu RPCs besteht hier in der Notwendigkeit, die Network Message als Objekt zu instanziieren und mit den korrekten Daten füllen zu müssen, bevor selbige versendet werden kann.

```
public bool DestroyParticle(string name) {

    if (!isServer && !hasAuthority)
        return false;

    bool result = DestroyParticleInternal(name);

    P_DestructionMessage message = new P_DestructionMessage() {
        name = name,
        particleComponentInstance = this,
        callerID = localClientID
    };

    if (!isServer) NetworkClient.Send(message);
    else NetworkServer.SendToReady(message);

    return result;
}
```

Dennoch bleibt der grundsätzliche Aufbau derselbe. Erfolgt der Aufruf direkt vom Server, kann dieser sofort alle Clients verständigen. Erfolgt der Aufruf von einem Client mit Autorität, setzt dieser zuerst den Server in Kenntnis und dieser informiert den Rest der Clients.

```
[Server]
private static void OnDestroyParticleServer(NetworkConnection conn,
    P_DestructionMessage msg) => NetworkServer.SendToReady(msg);

[Client]
private static void OnDestroyParticleClient(P_DestructionMessage msg) {

    if (localClientID == msg.callerID) return;

    msg.particleComponentInstance.DestroyParticleInternal(msg.name);
}
```

Aufgrund der Tatsache, dass die Logik auf dem ursprünglichen Client bereits aufgerufen wurde, wird eine Caller-ID mitgeschickt, sodass die Funktion hier doppelte Ausführungen vermeiden kann. Da außerdem die für die Bearbeitung einer Network Message notwendigen Funktionen stets statisch sein müssen, ist es vermutlich notwendig, einen Verweis auf die konkrete Modulinstanz, die die Übertragung verschickt hat, in der Network Message mit anzugeben. Dies ermöglicht den Aufruf auf der richtigen Instanz und gewährt Zugriff auf nicht-statische Klassenvariablen. Die tatsächliche Kernlogik der Funktion beläuft sich auch in diesem Beispiel wieder auf das Löschen des gefragten Partikeleffekts und das Austragen aus der Liste der aktiven Effekte.

```
private bool DestroyParticleInternal(string name) {

    if (!activeEffects.ContainsKey(name)) return false;

    GameObject particleEffect = activeEffects[name];
    activeEffects.Remove(name);

    Destroy(particleEffect);

    return true;
}
```

5 Auswertung

5.1 Metriken und Methodik zum Vergleichen der verschiedenen Ansätze

Um die im vorherigen Kapitel vorgestellten Ansätze zu vergleichen, gibt es zwei Metriken, die sich dazu besonders gut eignen: Latenz und Bandbreite. Die Latenz vergleicht dabei in gewisser Weise die Geschwindigkeit der Module. Da alle Varianten dieselbe Funktionalität ausführen, bleibt die dafür aufgewendete Zeit nahezu konstant. Unterschiede entstehen allerdings durch die Zeit, die Mirror zum Serialisieren, Versenden und Verarbeiten der Daten für jeden Fall benötigt. Die Bandbreite hingegen gibt Auskunft darüber, wie viele Daten in jeder Variante verschickt werden müssen und wie stark sie dadurch die verwendete Netzwerkverbindung belasten.

Zu diesem Zweck wurden alle vier Ansätze einmal in Unity am Beispiel von Scoo-Kings Partikelsystem implementiert. Der Test erfolgt in einer sonst leeren Szene, um interferierende Hintergrundberechnungen zu vermeiden. Per Knopfdruck wird dann ein automatisierter Test mit fixen Wartezeiten zwischen den einzelnen Aufrufen gestartet. Dabei werden mit jedem System jeweils 17 verschiedene Partikeleffekte aus Scoo-King zunächst erzeugt und nach Ablauf ihrer Lebensdauer wieder zerstört.

Für den Test läuft das Spiel in zwei Instanzen auf unterschiedlichen Computern. Beide Geräte befinden sich im selben Netzwerk und sind über LAN-Kabel miteinander verbunden. Dadurch soll ein realistischer Anwendungsfall mit mehreren Geräten simuliert werden, während gleichzeitig unbekannte Störfaktoren einer Internetverbindung zwischen verschiedenen Netzwerken vermieden werden sollen.

Das Messen der Latenz erfolgt innerhalb der Anwendung. Jedes Modul misst selbstständig die Zeit zwischen Aufruf einer Funktion und deren Ausführung. Die Messung erfolgt stets auf dem Computer, der die Funktion aufruft. Wenn dies nicht der Host ist, müssen die Anfrage an den Server und die Antwort des Servers abgewartet werden, bevor die Funktion ausgeführt werden kann. Gemessen wird hier explizit die Round-Trip Time, also die Zeit, die eine Übertragung braucht, um von Punkt A zu Punkt B und wieder zurück zu Punkt A zu gelangen. Da einige der Module allerdings in der Lage sind, den Inhalt der Funktion sofort umzusetzen, ohne auf eine Antwort des Servers zu warten, bildet die Round-Trip Time allein den Sachverhalt noch nicht vollständig ab. In diesen Fällen wird daher zusätzlich die reale Ausführungs-Verzögerung der Funktion gemessen, da die spätere Antwort des Servers ohnehin ignoriert wird. Die gemessenen Latenzen werden dann in eine Log-Datei zum anschließenden Auswerten geschrieben.

Für die vom Spiel genutzte Bandbreite gestaltet sich das Sammeln von Messwerten etwas schwieriger. Hier ist es notwendig, die Hilfe eines externen Programms, zum Protokollieren des Datenverkehrs eines Netzwerkes, hinzuzuziehen. Eine der beliebtesten Optionen dafür ist das Programm Wireshark. Mit dessen Hilfe wird der Netzwerkverkehr während des Versuchs mitgeschnitten und gespeichert. Dieser lässt sich dann mithilfe der richtigen IP-Adressen beider Geräte, des genutzten Netzwerk-Protokolls und des vom Spiel verwendeten Ports so filtern, dass ausschließlich die Datenpakete der Anwendung übrig bleiben.

Sobald alles vorbereitet ist und beide Computer miteinander verbunden sind, kann der Test beginnen. Auf sowohl dem Client als auch auf dem Host wird nacheinander der automatisierte Versuchsablauf gestartet. Anschließend werden die Protokolldateien gespeichert und können ausgewertet werden.

5.2 Evaluierung der Ergebnisse

Bei Betrachtung der gemessenen Latenzen in Tabelle 5.1 fällt zunächst auf, dass die Zeiten sich bei einer Ausführung auf dem Host kaum unterscheiden. Das ist logisch, da der Host keine netzwerktechnischen Verzögerungen erfährt und die ausgeführte Logik stets dieselbe ist. Sehr viel interessanter dahingegen sind die Latenzen beim Ausführen einer Funktion von einem Client. Betrachtet man hier die durchschnittliche Latenz, gibt es zwei klare Gewinner. Dies sind die beiden Ansätze mit Selbstverwaltung. Sowohl die Umsetzung mithilfe von RPCs als auch die mittels Network Messages liegen mit ihrer Ausführungsverzögerung bei nahezu null. Tatsächlich sind es auch genau diese Ansätze, denen erlaubt wurde, die Funktion sofort auszuführen, ohne eine Reaktion des Servers abzuwarten. Ihre Round-Trip Time ist zwar mit rund 15 ms etwa genauso hoch wie auch die des autoritären RPC-Systems, aber in der Praxis irrelevant, da die Ausführung bereits erfolgt ist.

Den zweiten Platz belegen die autoritären RPCs. Schlusslicht sind bei der Latenzmessung mit einem deutlichen Abstand von rund 76 ms die SyncVars. Dafür ist bei diesen gut die Intervall-basierte Synchronisierung zu erkennen. Zu erkennen sind regelmäßige Abschnitte mit stetig immer niedriger werdenden Latenzen, die dann schlagartig wieder auf ein Maximum springen. Bei diesen Umbrüchen fand jeweils eine Übertragung statt. Die darauf folgenden Aufrufe müssen dann das komplette Intervall abwarten, während die Wartezeit für die folgenden Aufrufe immer geringer wird. Die Latenz schwankt für diesen Ansatz folglich stark.

Ein leicht anderes Bild zeichnet sich, wenn man stattdessen auf die verwendete Bandbreite der einzelnen Ansätze in Tabelle 5.2 schaut. Zunächst einmal lässt sich festhalten: Funktionsaufrufe, die vom Host getätigt wurden, bestehen logischerweise aus nur einer Übertragung, während Aufrufe, die von einem Client getätigt wurden, sich aus einer Anfrage an den Server und einer Antwort desselbigen zusammensetzen. Ebenfalls sollte klar sein, dass bei zwei Übertragungen mehr Daten entstehen als bei einer. Der Fokus soll daher wieder auf den vom Client getätigten Aufrufen liegen, da diese ein detaillierteres Bild vermitteln.

Auch in dieser Metrik liegen die sich selbst verwaltenden Systeme wieder vorn. Diesmal ist jedoch mit durchschnittlich 132 Byte zu 145 Byte ein kleiner Vorsprung der Network Messages gegenüber den RPCs zu erkennen. Bei beiden Ansätzen entspricht die Anzahl der versendeten Pakete etwa der Anzahl der Aufrufe jeder Funktion. Weiterhin lässt sich nach einem etwas verworrenen Start sogar eine direkte Byte-genaue Korrelation zwischen der Größe der einzelnen Pakete und der Länge des versendeten Strings, der den Namen des Partikeleffekts angibt, feststellen.

SyncVars belegen hier den dritten Platz. Auch wenn die versendeten Pakete mit durchschnittlich 186 Byte deutlich größer sind, benötigt dieser Ansatz dafür nur 28 statt 33 Übertragungen. Dies Verhalten stammt aus der Synchronisierung in Intervallen von SyncVars, wodurch normalerweise mehrere Effekte in einem großen Paket (bis zu 515 Bytes) gebündelt werden. Die größere Gesamtdatenmenge im Vergleich zu den ersten beiden Ansätzen, obwohl diese dieselben Informationen verschicken, rührt von der zentralisierten Verwaltung der SyncVars. Während in der Selbstverwaltung nur einmal die Daten zum Erstellen des Effekts gesendet werden müssen und sich die Clients dann selbst um das Löschen des Effekts kümmern, benötigt eine zentrale Verwaltung eine Übertragung zum Erstellen und eine weitere Übertragung zum Löschen des Effekts. Die Differenz rührt also folglich von den zusätzlichen Löschanweisungen des Servers.

Den letzten Platz in Sachen Bandbreite belegt das autoritäre RPC-System. Mit insgesamt 7198 versendeten Bytes in 39 Paketen liegt dieses Modul sogar deutlich hinter den SyncVars. Die große Datenmenge ist einfach erklärt. Neben zusätzlichen Löschbefehlen gegenüber den selbstverwaltenden Systemen ist hauptsächlich die Art der übertragenen Informationen schuld. Da dieser Ansatz Mirrors internes System zum zentralen Verwalten von GameObjects nutzt, werden anders als sonst nicht nur einzelne Konfigurationsdaten, sondern vollständig serialisierte GameObjects versendet. Die Serialisierung eines kompletten GameObjects fällt logischerweise deutlich größer aus als die Serialisierung einer Hand voll primitiver Datentypen. Ein gutes Beispiel dafür, wie wichtig es ist, nur die Informationen zu versenden, die auch wirklich notwendig für die Aufgabe sind.

	Host					
	SelfRPC		AuthRPC	SyncVar	NetMessage	
	Ausf.-Verz.	RTT	RTT	RTT	Ausf.-Verz.	RTT
Effekt 1	2,1770	7,9887	4,3180	1,6094	1,2359	5,1049
Effekt 2	0,1704	0,4004	0,3456	0,1568	0,1863	0,5631
Effekt 3	0,1800	0,4283	0,3849	0,1411	0,1514	0,3838
Effekt 4	0,1574	0,3587	0,3802	0,1352	0,1730	0,4229
Effekt 5	0,1949	0,4574	0,4258	0,1818	0,2024	0,4367
Effekt 6	0,1927	0,3977	0,5529	0,2048	0,1900	0,5193
Effekt 7	0,1326	0,3211	0,4055	0,1442	0,1449	0,4113
Effekt 8	0,1658	0,4083	0,3969	0,1550	0,1520	0,4209
Effekt 9	0,1617	0,4124	0,3902	0,1447	0,1553	0,3693
Effekt 10	0,1103	0,3350	0,3534	0,0841	0,1051	0,3482
Effekt 11	0,0900	0,2875	0,3451	0,0988	0,0939	0,3058
Effekt 12	0,0944	0,3091	0,4491	0,1108	0,1092	0,3870
Effekt 13	0,0906	0,3054	0,3478	0,0868	0,0918	0,3216
Effekt 14	0,3031	0,5323	0,6073	0,2754	0,2819	0,5108
Effekt 15	0,2055	0,5131	0,4133	0,1499	0,1358	0,3633
Effekt 16	0,1608	0,3690	0,4640	0,1830	0,1387	0,3718
Effekt 17	0,1094	0,3288	0,4573	0,1128	0,1072	0,3403
Ø Latenz	0,2763	0,8325	0,6493	0,2338	0,2150	0,6812

	Client					
	SelfRPC		AuthRPC	SyncVar	NetMessage	
	Ausf.-Verz.	RTT	RTT	RTT	Ausf.-Verz.	RTT
Effekt 1	1,5305	36,0847	16,3659	32,3676	0,9932	30,7852
Effekt 2	0,4436	41,4975	15,2949	115,1525	0,4798	16,9405
Effekt 3	0,5010	16,4690	15,5636	98,9126	0,5048	14,4302
Effekt 4	0,4989	13,9955	15,2189	89,1902	0,4915	15,7536
Effekt 5	0,5679	15,6432	15,9338	73,1932	0,5462	15,1966
Effekt 6	0,6363	15,1022	14,9657	57,7855	0,6538	15,1178
Effekt 7	0,4583	14,8891	15,3087	42,4920	0,4875	15,7369
Effekt 8	0,4776	15,0591	15,9530	131,6298	0,4757	15,4651
Effekt 9	0,5666	15,1676	14,7438	107,1579	0,3913	15,1011
Effekt 10	0,3156	15,7189	15,7764	101,6891	0,2770	15,6486
Effekt 11	0,3306	14,8020	15,7565	85,4303	0,3018	16,0313
Effekt 12	0,3201	16,0134	14,6274	69,8400	0,3533	14,6823
Effekt 13	0,2879	14,3255	15,4260	54,1581	0,3537	15,8084
Effekt 14	0,8041	14,9492	15,5755	38,9819	0,8303	15,3962
Effekt 15	0,4012	14,8555	15,1665	22,2836	0,4344	15,0185
Effekt 16	0,4309	15,3924	15,9684	92,2480	0,4171	15,8383
Effekt 17	0,4114	15,3714	15,0633	84,3089	0,3963	14,9198
Ø Latenz	0,5284	17,9610	15,4534	76,2836	0,4934	16,3453

Tabelle 5.1: Gemessene Latenzen in Millisekunden für alle vier Ansätze.
(Ausf.-Verz.: Ausführungs-Verzögerung, RTT: Round-Trip Time)

	Host				Client							
	SelfRPC	AuthRPC	SyncVar	NetMessage	SelfRPC		AuthRPC		SyncVar		NetMessage	
					An	Aw	An	Aw	An	Aw	An	Aw
	120	206	163	112	120	144	144	230	116	187	112	195
	196	206	457	111	220	211	143	230	115	457	111	136
	144	206	441	112	143	143	144	230	140	507	136	135
	139	206	241	111	144	144	143	230	115	199	135	136
	120	206	141	112	143	143	144	230	116	141	136	135
	119	206	151	111	144	144	143	265	115	199	135	136
	120	241	297	112	168	167	144	265	116	199	136	135
	119	206	163	111	133	133	143	230	115	307	135	125
	133	241	397	101	138	138	133	265	129	163	125	130
	114	241	221	106	138	138	138	265	177	515	130	130
	114	241	241	106	138	138	138	265	110	220	130	130
	114	266		106	138	138	138	265	110	149	130	130
	114	241		106	139	139	138	265	111		130	131
	115	241		107	135	135	139	265	107		131	127
	111	241		103	137	137	135	265	133		127	129
	113	241		105	139	139	137	277	135		129	131
	115	253		107			139	123			131	
		123						123				
		123						123				
		123						147				
		123						111				
		135						146				
		111										
		111										
		107										
Anz.	17	25	11	17	32		39		28		33	
Ø	124,71	193,80	264,82	108,18	145,25		184,56		185,82		132,42	
Ges.	2120	4845	2913	1839	4648		7198		5203		4370	

Tabelle 5.2: Gemessener Datenverkehr in Bytes für alle vier Ansätze.
(An.: Anfrage, Aw.: Antwort, Anz.: Anzahl Datenpakete,
Ø: Durchschn. Paketgröße, Ges.: Gesamtdatenmenge)

5.3 Fazit

Abschließend lässt sich festhalten: Netzwerk-Spiele nutzen verschiedene Topologien und Protokolle, um ihre Ziele zu erreichen. Am häufigsten wird dabei mit einer Server-Client-Architektur gearbeitet, bei kleineren Spielen aber mitunter auch mit einer Host-Client-Architektur, bei der einer der Clients zusätzlich die Rolle des Servers mit übernimmt.

Um Netcode in Spielen zu implementieren, gibt es eine Reihe von Frameworks mit verschiedensten Ansätzen, die diese Aufgabe erleichtern. Jedes dieser Frameworks hat Vor- und Nachteile und die Wahl des richtigen Frameworks für ein Spiel hängt stark von dessen Anforderungen ab. Für Scoo-King empfiehlt sich eine Host-Client-Architektur und am besten eignet sich das Framework Mirror.

Mirror stellt eine Reihe von wichtigen Tools zur Verfügung, um einfach Netcode für ein Spiel zu schreiben. Die wichtigsten davon sind Remote Procedure Calls, SyncVars und selbstdefinierte Network Messages. Unter Verwendung eben dieser wurden vier Ansätze zum Aufbau eines Netzwerkmoduls in einem Spiel vorgestellt: zwei selbstverwaltende Systeme mit entweder RPCs oder Network Messages, ein strikt autoritärer Ansatz ebenfalls mittels RPCs und ein gemischter Ansatz unter Zuhilfenahme von SyncVars, bei dem die grundlegenden Aufträge des Systems vom Server verwaltet werden, deren Umsetzung aber den Clients obliegt.

Gemessen wurden für alle vier dieser Ansätze die Latenz und die verwendete Bandbreite des Moduls. Dabei schneiden in beiden Fällen die selbstverwaltenden Systeme besonders gut ab. Wobei Network Messages sogar noch ein winziges bisschen weniger Bandbreite verwenden. Dafür sind selbstverwaltende Module anfälliger gegenüber Manipulation. Strikt oder teilweise autoritäre Systeme sind langsamer, da sie zuerst auf eine Antwort des Servers warten müssen. Besonders die Verwendung von SyncVars führt zu langen Verzögerungen, da Änderungen hier in wenigen großen Datenpaketen gebündelt werden und nicht sofort verschickt. Bei der Verwendung von fertigen generischen Systemen in autoritären Ansätzen können dennoch sehr viel größere Datenmengen entstehen, da hier nicht immer nur die tatsächlich notwendigen Informationen verschickt werden.

Wenn die Sicherheit des Systems es also zulässt, sind selbstverwaltende Systeme mit RPCs oder Network Messages zu präferieren. Dennoch sind RPCs auch in autoritären Ansätzen eine legitime Lösung mit nicht allzu großen Verzögerungen. Dabei muss hier nur aufgepasst werden, dass dazu verwendete Systeme auch nur wirklich relevante Informationen verschicken. SyncVars dahingegen belasten zwar die Bandbreite nicht sehr viel mehr als RPCs oder Network Messages das auch tun, eignen sich aber nicht für zeitkritische Anwendungsfälle.

Originalarbeiten

Ahde, J (2017) Real-time Unity Multiplayer Server Implementation.

Ahmed, I, KS Shabab, MS Hossain, M Akand und S Majumder (2018) Real time online multiplayer gaming framework using kinect sensors.

Armitage, G, M Claypool und P Branch (2006) Networking and Online Games - Understanding and Engineering Multiplayer Internet Games. John Wiley und Sons, Ltd. ISBN: 0-470-01857-7.

Boroń, M, J Brzeziński und A Kobusińska (2019) P2P matchmaking solution for online games.

Engelbrecht, D (2022) Building Multiplayer Games in Unity: Using Mirror Networking. Apress und Imprint Apress. Berkeley, CA. ISBN: 9781484274743.

Glazer, JL (2016) Multiplayer game programming: Architecting networked games. Addison-Wesley. New York. ISBN: 9780134034355.

Kelly, S und K Kumar (2022) Unity networking fundamentals: Creating multiplayer games with Unity. Apress. Berkeley, CA. ISBN: 9781484273586.

Magda El Zarki The Challenges of Networked Games.

Sanchez, RJ (Rodrigo) (2020) Realtime Networking Technologies for Unity.

Smed, J, T Kaukoranta und H Hakonen (2002) A Review on Networking and Multiplayer Computer Games.

Wang, Y (2017) A framework for developing network based games using Unity and Ice.

Webseiten

Exit Games. *Multiplayer Made Simple*. URL: https://www.photonengine.com/ (aufgerufen am 26. 02. 2023).

Glover, D. (2019) *UNet Deprecation FAQ*. URL: https://support.unity.com/hc/en-us/articles/360001252086-UNet-Deprecation-FAQ (aufgerufen am 22. 02. 2023).

Hammer, H. (2021) *Scoo-King – Spieleentwickler präsentierten ihr Abschlussprojekt*. URL: https://www.hs-mittweida.de/news/aktuell/6770/ (aufgerufen am 12. 04. 2023).

House, B. (2020) *How to choose the right netcode for your game*. URL: https://blog.unity.com/technology/choosing-the-right-netcode-for-your-game (aufgerufen am 18. 02. 2023).

Kubasova, N. (2022) *Unordinal acquires DarkRift2 and releases it as an open source initiative*. URL: https://unordinal.com/unordinal-acquires-darkrift2-and-releases-it-as-an-open-source-initiative/ (aufgerufen am 22. 02. 2023).

Unity Technologies. (2020) *Choosing the Right Netcode*. URL: https://create.unity.com/form-netcode-report (aufgerufen am 19. 02. 2023).

vis2k. *Mirror Networking*. URL: https://mirror-networking.gitbook.io/docs/ (aufgerufen am 17. 04. 2023).

Erklärung

Hiermit erkläre ich, dass ich meine Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die Arbeit noch nicht anderweitig für Prüfungszwecke vorgelegt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Mittweida, 04.05.2023