Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Sven Allers

Lastverteilung in einer clusterbasierten verteilten virtuellen Umgebung

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Sven Allers

Lastverteilung in einer clusterbasierten verteilten virtuellen Umgebung

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Martin Hübner
Zweitgutachter: Lutz Behnke

Eingereicht am: 12. September 2015

Sven Allers

Thema der Arbeit
Lastverteilung in einer clusterbasierten verteilten virtuellen Umgebung

Stichworte
Verteilte virtuelle Umgebungen, Lastverteilung, Load Balancing, Verteilte Systeme, Erlang

Kurzzusammenfassung
Diese Arbeit befasst sich mit der Lastverteilung in verteilten virtuellen Umgebungen. Dazu wurden zwei Verfahren zur Lastverteilung im Kontext des studentischen Projektes Timadorus entwickelt. Diese sollen zusammenhängende Spielergruppen bilden, die jeweils einem Spielserver zugewiesen werden. Das eine Verfahren, versucht die Kantengewichte eines imaginären Graphen zu maximieren, um so die Abstände zwischen den Spielserver möglichst hoch zu halten. Das andere Verfahren baut eine Heat Map auf, die beschreibt wie stark welche Region, durch die einzelnen Spielserver belegt ist und versucht darüber einen geeigneten Spielserver für neue Spieler zu finden. Außerdem wird das, auch in dieser Arbeit entwickelte, Load Balancing System vorgestellt, in dem die Verfahren zum Einsatz kommen. Über Experimente wird zudem die Leistungsfähigkeit der entwickelten Verfahren gezeigt.
Sven Allers

Title of the paper
Load Distribution in a Cluster Based Distributed Virtual Environment

Keywords
Distributed Virtual Environments, Load Distribution, Load Balancing, Distributed Systems, Erlang

Abstract
This work covers the load distribution of distributed virtual environments. For this two load distribution techniques were developed in the context of the student project Timadorus. These should allocate continuous groups of players to the game servers. One of the techniques tries to maximize the weights of the edges of an imaginary graph, to uphold the distances between the game servers. The other technique creates a Heat Map which describes how strong game servers are represented at parts of the game world and tries to find the most appropriate game server for new player with this. In addition, the load balancing system applying the techniques, which were developed in this work, will be introduced. Furthermore, experiments will show the performance of the developed techniques.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1. Einleitung

1.1. Motivation

Mithilfe von verteilten virtuellen Umgebungen (Distributed Virtual Environment (DVE)) können Menschen auf der ganzen Welt virtuell miteinander interagieren. Hierbei handelt es sich um 3D-Welten, welche über mehrere verteilte Rechner genutzt werden können. Dazu zählen u.a. Multiplayer Online Games [DL14]. Die Beliebtheit solcher Multiplayer Online Games ist in den letzten Jahren deutlich gestiegen, während Lineage Ende des Jahres 2000 mit ca. 2 Mio. Abonnenten eines der größten Online Spiele auf dem Markt war, konnte World of Warcraft im November des Jahres 2013 ca. 7,6 Mio. Abonnenten vorweisen [MMO14].

Damit tausende Spieler parallel gleichzeitig spielen können, ist es notwendig, die Last effizient auf die einzelnen Spielserver aufzuteilen. Eine beliebte Möglichkeit ist eine geographische Partitionierung der Spielwelt. Hierbei wird jede Partition von einem Spielserver verwaltet. Dies kann statisch sowie dynamisch geschehen. Statische Methoden sind zwar einfach zu implementieren, neigen jedoch dazu, dass einzelne Server überfüllt sind, wohingegen andere kaum genutzt werden [LC10]. Dadurch können Spieler auf solchen überlasteten Servern eine deutlich höhere Verzögerung haben, als solche auf weniger stark ausgelasteten Spielserver. In Spielen, wie zum Beispiel World of Warcraft oder Second Life, ist es deshalb gar nicht erst möglich, dass Spieler aus verschiedenen Partitionen miteinander interagieren, um die damit einhergehenden Synchronisationsprobleme zu vermeiden [DL14].

Einen anderen Ansatz bieten die dynamischen Partitionierungsmethoden. Diese ordnen den zu verwaltenden Bereich den Spielservern in Abhängigkeit von der Auslastung bestimmter Bereiche zu und streben eine Balance zwischen der Auslastung der einzelnen Spielserver und den Kommunikationskosten der Spielserver untereinander an [CWD+05]. Neben der Partitionierung kann auch Sharding eingesetzt werden, um die Last zu kontrollieren. Hierbei wird die Spielwelt auf mehreren Spielservern repliziert und die Spielerzahl innerhalb der einzelnen Spielwelten begrenzt. So werden beispielsweise bei World of Warcraft weitere Shards in einem

Server Cluster gestartet, wenn die Auslastung aller Spielserver sehr hoch ist. Ein Nachteil dieser Methode ist, dass sich dadurch immer nur eine Teilmenge der Spieler innerhalb eines Shards befindet und diese nicht miteinander interagieren können, solange diese sich in verschiedenen Shards aufhalten [MO12].

Man sieht, es gibt eine Reihe von Möglichkeiten, um die Last von Spielservern innerhalb von DVEs zu verwalten. Die folgende Arbeit wird sich mit diesem Thema im Kontext des studentischen Projektes Timadorus befassen. Ziel dieses Projektes ist es, ein Online-Rollenspiel zu schaffen, in dem sämtliche Spieler ein gemeinsames Spielerlebnis haben können. Das bedeutet, dass diese sich jederzeit in einer gemeinsamen Spielwelt befinden. Eines der bekanntesten Spiele dieser Art ist Eve Online mit ca. 500.000 Abonnenten (Stand: Februar 2013), Tendenz steigend [MMO14]. Damit solch ein Spiel trotz der hohen Spielerzahlen von bis zu 65.000 Spielern zeitgleich [MMO14] leistungsfähig bleibt, ist es unumgänglich, ein effizientes Load Balancing Verfahren zur Verfügung zu haben [Eve13]. Das Timadorus Projekt weist neben den üblichen Problemen einer gerechten Lastverteilung noch einige Besonderheiten, wie eine nicht serverbasierte Verwaltung einzelner Regionen bzw. Objekte, auf.

1.2. Problemstellung

Da Timadorus ein Online-Rollenspiel werden soll, bei dem sich potentiell tausende Spieler zeitgleich auf der Spielwelt befinden, ist es nötig, dass diese auch von mehreren Spielservern verwaltet werden kann. Daraus folgt, dass Spieler, welche dem Spiel beitreten, auf die Spielserver verteilt werden müssen. Ein Verfahren zum Verteilen der Spieler ist jedoch bis jetzt noch nicht vorhanden und muss daher entwickelt werden. Neben den üblichen Problemen, wie eine gerechte Verteilung der Ressourcen, kommen in Timadorus noch einige Besonderheiten hinzu. So sind die von den Spielservern verwalteten Objekte nicht über eine dem Spielserver zugeordnete Region definiert, sondern ergeben sich aus dem Interessengebiet (Area of Interest) der einzelnen Avatare, die sich auf dem Spielserver befinden. Daraus ergeben sich Probleme, wie ein sich stetig ändernder Verwaltungsbereich der einzelnen Spielserver bei Bewegung der Spieler oder der Möglichkeit, dass mehrere Spielserver zeitgleich dasselbe Objekt in Verwaltung haben können. Die Kontrolle, ob bestimmte Veränderungen erfolgreich waren und wie der Zustand bestimmter Objekte ist, wird dabei von dem Storagesystem QuP im Hintergrund übernommen [BGL14].

Dadurch kommen neben klassischen Performancekriterien, wie Auslastung von Speicher

und CPU, auch die Lokalität der Spieler auf der Spielwelt hinzu. So ist es bei einer initialen Zuordnung nicht möglich zu betrachten, welcher Spielserver für welchen Bereich der Spielwelt zuständig ist. Somit muss eine Möglichkeit gefunden werden, dass neue Spieler einem Spielserver zugewiesen werden, auf dem sich möglichst viele Spieler mit überlappenden Interessengebieten befinden.

1.3. Zielsetzung

Ziel der Arbeit ist es eine Lösung zu entwickeln und umzusetzen, mit der es möglich ist, die Clients auf die einzelnen Spielserver aufzuteilen. Dabei soll durch das Load Balancing die Effizienz des Gesamtsystems gesteigert werden. Dafür sollen die genauen Effizienzkriterien innerhalb der Arbeit ermittelt werden. Auch soll das bestehende Projekt betrachtet werden und auf dieser Grundlage mehrere Verfahren sowie ein System zur Lastverteilung entwickelt und implementiert werden. In diesem Zusammenhang sollen eine API entstehen, mithilfe dessen die benötigten Lastinformationen ausgetauscht werden können, sowie Spieler die Möglichkeit haben, sich einem Spielserver zuordnen lassen zu können. Zum Abschluss soll die Leistungsfähigkeit der entwickelten Verfahren experimentell ermittelt werden.

Innerhalb dieser Arbeit soll nur die initiale Zuweisung der Clients auf die Spielserver betrachtet werden. Eine dynamische Migration der Clients zwischen den Servern ist nicht vorgesehen und kann Bestandteil zukünftiger Arbeiten sein.

1.4. Struktur der Arbeit

Die Arbeit ist in acht Kapitel unterteilt. Im Folgenden soll die Gliederung näher Beschrieben werden.

Zunächst wird im Kapitel „Timadorus“ das Timadorus Projekt vorgestellt werden, innerhalb dessen die Arbeit realisiert wurde. Hier werden neben einer allgemeinen Einführung, die Elemente des bisher vorliegenden Systems vorgestellt, sowie alle Relevanten Konzepte und Begriffe beschrieben.

Im folgenden Kapitel „Bisherige Load Balancing Verfahren“ werden bereits vorliegende Load Balancing Konzepte betrachtet. Dazu werden Load Balancing Verfaren in anderen Online-Spielen sowie andere wissenschaftliche Arbeiten betrachtet. Zudem wird am Ende betrachtet,

ob die Verfahren für Timadorus geeignet sind.

Anschließend werden im Kapitel „Anforderungen“ die Anforderungen an das zu entwickelnde System sowie den zu entwickelnden Verfahren beschrieben. Das beinhaltet neben funktionalen, auch Anforderungen an die Verfahren und die Architektur des Systems.

Im nächsten Kapitel „Die Verfahren“ werden die beiden in der Arbeit entwickelten Verfahren beschrieben. Das ist zum einen ein graphbasiertes Verfahren und zum anderen ein Verfahren, das versucht mithilfe von Heat Maps einen passenden Spielserver zu finden. Dazu werden zunächst die Ziele der Verfahren beschrieben und daraus eine allgemeine Kostenfunktion erarbeitet. Als nächstes wird das entwickelte graphbasierte Verfahren beschrieben. Das beinhaltet die Verwaltung der Spielserver sowie das Ermitteln eines Spielservers für einen Spieler. Zudem wird die theoretische Laufzeit des Verfahrens ermittelt. Darauf folgend wird das Heat Map Verfahren beschrieben. Auch hier werden die Verwaltung der Spielwelt, das Ermitteln eines Spielservers sowie die Laufzeit des Verfahrens betrachtet. Zusätzlich wird eine Optimierung durch Parallelisierung, sowie deren Laufzeit beschrieben.

Darauf folgt das Kapitel „Implementierung“. Hier wird das System beschrieben, in dem die Verfahren eingesetzt werden. Neben Architektur und Schnittstellen, wird auch auf weitergehende Probleme eingegangen. Wie das Konsistenzproblem, das entsteht, wenn zu starke Veränderungen in der Spielwelt auftreten, während ein Spielserver ermittelt wird oder wie damit verfahren wird, wenn ein Spielserver keine Daten mehr übermittelt. Zudem werden zum tieferen Einblick in das Gesamtsystem die Umsetzung bestimmter Anwendungsfälle erläutert. Zum Abschluss wird beschrieben, wie die Verfahren umgesetzt wurden. Das beinhaltet die Integration in den Load Balancing Server und den, in den Spielservern integrierten, Monitor sowie die eingesetzten Datenstrukturen.

Im Kapitel „Experimente“ werden zunächst die auf den Verfahren ausgeführten Experimente beschrieben. Um die Leistungsfähigkeit der Verfahren zu ermitteln, wird betrachtet, wie effizient diese die Spieler gruppieren und wie die Auswirkungen der Verfahren auf die Speicherbelastung der Spielserver ist. Zudem wurde ein Lasttest durchgeführt, um zu ermitteln, wie effizient die Verfahren auch bei hohen Spielerzahlen sind.

Im letzten Kapitel „Abschluss“ gibt es zunächst einen Rückblick über die Arbeit. Anschlie-

ßend wird ein Fazit der Arbeit gezogen und zum Abschluss wird vorgestellt, wie mögliche weiterführenden Arbeiten aussehen könnten.

2. Timadorus

Da die Load Balancing Verfahren innerhalb des Timadorus Projektes entwickelt wurden, soll im Folgenden Kapitel dieses Projekt näher dargestellt werden. Dazu werden im ersten Abschnitt allgemeine Informationen zum Projekt vorgestellt. Da die Darstellung von Positionen und Größen eine zentrale Rolle in den entwickelten Verfahren spielt, wird zunächst auf diese eingegangen. Anschließend wird die Architektur des Systems näher betrachtet und die einzelne Bestandteile genauer beschrieben. Zum Abschluss wird auf die Kommunikation der einzelnen Module des Timadorus Projektes eingegangen und diese näher dargestellt.

Bei der Betrachtung des Systems im folgenden Kapitel handelt es sich um den Stand vor der Entstehung dieser Arbeit. Diese dient als Grundlage der entwickelten Verfahren sowie des entwickelten Systems.

2.1. Allgemein

Timadorus ist ein Projekt an der HAW Hamburg, welches von Studenten entwickelt wird. Ziel ist es, ein Online-Rollenspiel zu entwickeln, in dem potentiell tausende von Spielern gleichzeitig spielen können. Dabei soll die Technologie für Online-Rollenspiele voran gebracht werden und neueste Technologien eingesetzt werden. Die besondere Herausforderung liegt darin, ein ausfallsicheres System zu schaffen, welches in der Lage ist, rund um die Uhr lauffähig zu sein. Für das zu entwickelnde Spiel gilt, dass die komplette Spiellogik serverbasiert ist und die Clients somit nur der Darstellung der Spielwelt dienen. Zudem soll Sharding vollkommen vermieden werden. So können sich sämtliche Spieler innerhalb einer Spielwelt bewegen und ein gemeinsames Spielerlebnis haben, da sie nicht auf verschiedene Instanzen der Spielwelt aufgeteilt werden. Auch soll es keine nachladende Zonen geben, damit der Spielfluss nicht unterbrochen wird [Timb].

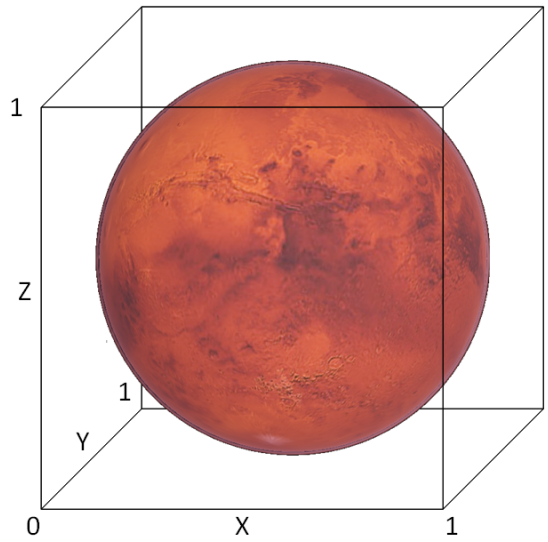


Abbildung 2.1.: Planet als Spielwelt in einem euklidschen Raum mit normierten Achsen

2.2. Darstellung von Positionen und Größen

Die Spielwelt in Timadorus wird in einem euklidschen Raum beschrieben, wobei die maximale Länge einer jeden Achse normiert wurde. Die Werte für X, Y und Z können somit maximal Eins betragen. Dadurch bestehen die Elemente der Vektoren, welche Größe und Position der Objekte der Spielwelt beschreiben, aus Werten zwischen Null und Eins. Eine wichtige Haupteigenschaft der Spielwelt ist zudem, dass es weder Oben noch Unten gibt. So könnte sich in dieser Matrix auch ein einzelner Planet befinden, auf dem sich die Spieler bewegen (siehe Abb. 2.1). Spieler, die sich auf der Südhalbkugel befinden, haben dort eine kleinere Z-Koordinate, obwohl diese sich nicht zwingend auf einer niedrigeren Höhe auf dem Planeten befinden müssten.

Im Folgenden werden die Strukturen zur Beschreibung von Position, sowie Größe von Objekten der Spielwelt, näher beschrieben. Damit soll aufgezeigt werden, wie dieses Konzept in Timadorus umgesetzt wurde. Außerdem wird ein Verständnis dieser Strukturen benötigt, um die folgende Arbeit nachvollziehen zu können.

Position

Die Position der Spieler wird in Form eines Tupels, bestehend aus den vier Elementen Koordinate, Bewegung, Orientierung sowie Rotation beschrieben. Im Detail sieht eine Position somit wie folgt aus:

$$\begin{aligned} Position &= \{Koordinate, Bewegung, Orientierung, Rotation\} \\ Koordinate &= \{X, Y, Z\} \\ Bewegung &= \{BewegungX, BewegungY, BewegungZ\} \\ Orientierung &= \{OrientierungW, OrientierungX, OrientierungY, OrientierungZ\} \\ Rotation &= \{RotationW, RotationX, RotationY, RotationZ\} \end{aligned} \tag{2.1}$$

Im Folgenden werden die Elemente der Position noch einmal näher Beschrieben:

Koordinate Die Koordinate beschreibt die Position des Objekts im euklidschen Raum in Form eines dreidimensionalen Vektors.

Bewegung Beschreibt die aktuelle Bewegungsrichtung und -geschwindigkeit eines Objekts. Dies wird vor allem zur Neuberechnung von Positionen benötigt. Auch dieser Wert wird durch einen dreidimensionalen Vektoren beschrieben, bei dem X, Y und Z maximal den Wert Eins haben dürfen.

Orientierung Beschreibt die Orientierung des Objektes mithilfe eines Quaternions.

Rotation Beschreibt die aktuelle Rotationsbewegung des Objektes in Form eines Quaternions. Dies wird hauptsächlich zur Neuberechnung von Positionen benötigt.

Bounding Box

Alle physischen Objekte der Spielwelt haben eine bestimmte Größe. Diese Größe wird über eine sogenannte Bounding Box beschrieben. Die Bounding Box in Timadorus ist wie folgt beschrieben:

$$BoundingBox = \{groesseX, groesseY, groesseZ\} \tag{2.2}$$

Es handelt sich um einen dreidimensionalen Vektor, dessen Elemente die Größe des Objektes auf einer Achse beschreiben. Um den Bereich zu ermitteln, der durch die Bounding Box abgedeckt wird, wird zusätzlich die Position benötigt. Dieser kann nun ermittelt werden, indem die Bounding Box um die Position gezogen wird.

View Box

Jeder Spieler kann in Timadorus nur einen bestimmten Bereich einsehen. Dieser sichtbare Bereich wird in Timadorus über die View Box eingegrenzt. Alle Objekte außerhalb der View Box sind somit für den Spieler nicht sichtbar. Technisch gesehen handelt es sich bei der View Box um eine spezielle Art der Bounding Box und beschreibt Höhe, Breite und Tiefe des Sichtbereiches. Deshalb ist diese in Timadorus auch gleich definiert:

$$ViewBox = \{groesseX, groesseY, groesseZ\} \tag{2.3}$$

Auch der abgedeckte Bereich kann, wie bei der Bounding Box, ermittelt werden, indem die View Box um die Position gezogen wird.

Innerhalb der bisherigen Timadorus Systeme stellt die View Box eine reine Beschreibung der Größe dieser Box dar. In den zukünftigen Kapiteln soll der Begriff View Box jedoch, zur einfacheren Kommunikation, allgemeiner gefasst werden. Demnach soll die View Box, nicht nur die Größe der Box, sondern auch die Position beschreiben. Das bedeutet, dass die View Box in folgenden Kapiteln aus zwei Elementen besteht. Dies ist zum einen die Position der View Box, sowie der bisherige Tupel einer View Box, zur Beschreibung des abgedeckten Bereiches. Diese ist demnach wie folgt aufgebaut:

$$LBViewBox = \{Position, ViewBox\} \tag{2.4}$$

2.3. Architektur

Timadorus beruht auf einer 3 Tier Architektur, die von außen betrachtet jedoch wie eine klassische Client/Server Architektur wirkt. So kann von einer Aufteilung von Spielclients, Spielservern sowie dem QuP zur Datenhaltung gesprochen werden. Parallel dazu gibt es noch einen Authentifizierungsserver, über den die Anmeldung an das System sowie die Zuweisung zu einem Spielserver stattfindet. Mit Ausnahme zur Anmeldung an das System, bei dem die Spielclients eine Anfrage an den Authentifizierungsserver senden, kommunizieren die Spielclients ausschließlich mit dem ihm zugeordnetem Spielserver. Dabei ist jeder Client genau einem Spielserver zugeordnet. Hinter den Spielservern steht das QuP-System zur Verwaltung der Daten. Da es sich beim QuP um ein verteiltes Storagesystem handelt, gibt es von den QuP-Servern mehrere Instanzen. Eine feste Zuordnung zwischen den Spielservern und den QuP-Servern gibt es in diesem Fall nicht. So kann jeder Spielserver potentiell jeden QuP-Server

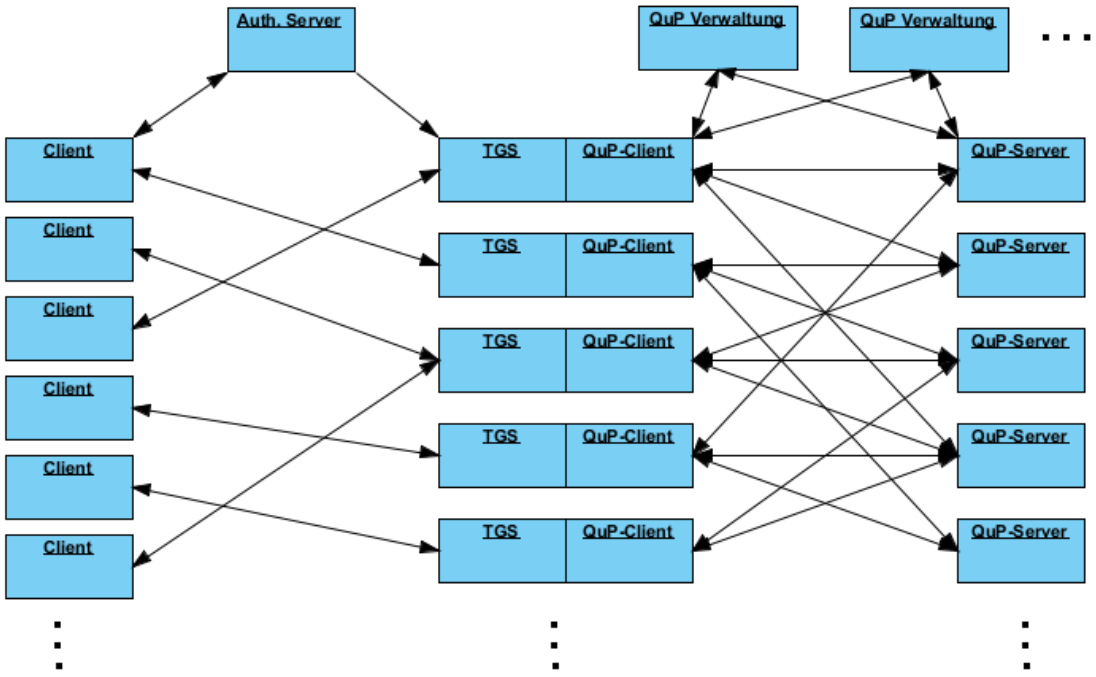


Abbildung 2.2.: Aufbau Timadorus-System (TGS = Timadorus Game Server)

kontaktieren. Die Spielserver haben dazu einen speziellen QuP-Client integriert, der über ein Hashing-Verfahren die zu verwendenden QuP-Server für die jeweiligen Anfrage ermittelt [BGL14]. Zur Verwaltung der QuP-Server und QuP-Clients besitzt das QuP-System zudem mehrere Kontrollinstanzen, bei denen sich diese registrieren, wenn sie das System betreten bzw. abmelden, wenn sie dieses verlassen.

2.3.1. Spielclients

Die Spielclients in Timadorus sollen rein der Darstellung der 3D-Welt dienen. Dazu wird eine Lösung in Java entwickelt. Da die Spielclients möglichst wenig Logik verwalten sollen, werden sämtliche spiellogischen Anfragen an den Spielserver weitergeleitet und die clientseitige Darstellung erst durchgeführt, wenn von Serverseite eine Antwort mit dem Resultat kommt. Spielclients sind jederzeit mit nur einem der Spielserver verbunden. Diesen erhalten sie zum Zeitpunkt der Anmeldung von dem Authentifizierungsserver, zusammen mit einem Token zur Verifikation am Spielserver. Zur Kommunikation mit dem Server wurde das sogenannte Timadorus Message Protokoll entwickelt in dem sämtliche Anfragen mit den entsprechenden Antworten definiert wurden [Timc].

2.3.2. Spielserver

Die Spielserver verwalten die Spieler sowie die Logik des Spiels. Sollten sich Objekte verändern, benachrichtigt dieser das QuP-System. Zudem abonniert der Spielserver beim QuP-System alle Objekte, die für die bei ihm angemeldeten Clients von Interesse sind. Sollte sich nun ein Objekt auf einem anderen Server verändern, so wird der Spielserver über das QuP-System darüber informiert und leitet diese Nachricht an die entsprechenden Clients mittels des Timadorus Message Protokolls weiter [Timc].

2.3.3. QuP

Beim QuP handelt es sich um ein verteiltes System zur Datenhaltung, welches stark auf DVEs spezialisiert ist. So gibt es die Möglichkeit, sich als Beobachter bestimmter Regionen der Spielwelt zu registrieren oder einzelne Objekte zu abonnieren und benachrichtigt zu werden, sobald sich diese verändern. Um dies zu realisieren, wurde ein Octree implementiert, um effizient Objekte den einzelnen Regionen zuordnen zu können. Das System basiert auf einer Client/-Server Architektur und besteht somit aus zwei Hauptkomponenten. Einmal dem QuP-Client und den QuP-Servern. Bei den Servern handelt es sich um ein verteiltes Serversystem, in dem jeder Server eine Teilmenge der Objekte abspeichert. Über ein Hashing-Verfahren wird ermittelt, auf welchen Servern, welche Objekte abgelegt werden und wo diese zu finden sind. Um die Ausfallsicherheit zu erhöhen, wird der Zustand der Objekte zusätzlich über mehrere Server repliziert. Gültige Lese- und Schreibzugriffe werden innerhalb des Systems über ein Quorum bestimmt [BGL14]. Damit den QuP-Clients auch alle QuP-Server bekannt sind, gibt es zusätzlich sogenannte Kontaktknoten zur Verwaltung der Spielserver. Bei denen melden sich die Clients nach dem Start an und erhalten eine Liste aller registrierten Server. Dazu registrieren sich die Server nach Start bei den Knoten. Sollte ein neuer Server hinzukommen, werden alle registrierten Clients über diesen direkt informiert.

Innerhalb von Timadorus befindet sich das QuP-System direkt hinter den Spieleservern und verwaltet den Zustand der Spielwelt. Dazu sind die zuvor genannten QuP-Clients in die Spielserver integriert worden. Dadurch brauchen sich diese nicht mehr um die Haltung des Weltzustandes kümmern und können sich vollkommen auf die Spiellogik begrenzen.

Datenhaltung im QuP-Client

Um die Kommunikation zwischen dem QuP-Client und dem QuP-Server so gering wie möglich zu halten, speichert der QuP-Client sämtliche Objekte, die von den angebundenen Spielclients

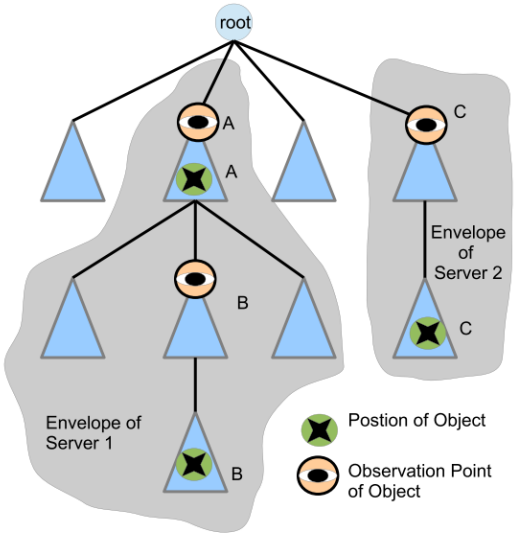


Abbildung 2.3.: Baum mit Observern und Objekten. Darstellung als Quadtree. Aus [BGL14].

benötigt werden In-Memory. Dazu macht sich das System die Annahme zunutze, dass ein Großteil der Informationen innerhalb einer DVE statisch sind und sich somit selten verändern. Um Veränderungen mitzubekommen, die auf anderen Spielservern stattgefunden haben, abonniert der QuP-Client zudem beim QuP-Server alle Objekte die benötigt werden. Sollten nun Veränderungen beim QuP-Server eingehen, so wird der QuP-Client direkt darüber benachrichtigt. Das QuP-System bietet dabei nicht nur die Möglichkeit einzelne Objekte zu abonnieren, sondern auch, mit sogenannten Observern, bestimmte Bereiche der Spielwelt zu abonnieren. Dadurch abonniert dieser sämtliche Objekte innerhalb dieses Bereiches. Zudem wird er benachrichtigt, wenn sich neue Objekte in diesem Bereich aufhalten sollten bzw. diesen verlassen [BGL14]. So kann der Client direkt darauf reagieren und die entsprechenden Objekte abonnieren bzw. die Abonnements kündigen.

Der Octree

Um die Spielwelt zu verwalten, wird diese vom QuP-System in einen sogenannten Octree eingeteilt. Hierbei handelt es sich um einen speziellen Baum, der die Spielwelt darstellt. Dabei stellt die Wurzel die gesamte Spielwelt dar. Eine Ebene tiefer stellt die Spielwelt in acht gleichgroße Abschnitte dar, wobei jeder Knoten einen dieser Abschnitte repräsentiert. Die Kinder eines jeden Knotens unterteilen dessen Bereich wiederum in acht gleich große Teile. Jede Ebene stellt somit eine Verfeinerung des Spielwelt dar. Im QuP-System wird dies bis hin

zu einer fest definierten Tiefe gemacht. Jeder dieser Knoten kann nun Objekte und Observer der entsprechenden Position enthalten. Dabei beinhaltet immer der kleinstmögliche Knoten die Objekte bzw. Observer [BGL14].

Damit das QuP-System Objekte einem bestimmtem Knoten zuordnen kann, berechnet es aus dessen Position und Bounding Box einen Positionscode, der den entsprechenden kleinstmöglichen Knoten darstellt. Bei Observern wird äquivalent vorgegangen, nur mit dem Unterschied, dass hier statt der Bounding Box, die View Box des Objektes verwendet wird.

Observer

Observer beschreiben eine besondere Art von Objekten, die innerhalb eines Octrees abgelegt werden. Diese ermöglichen es einen bestimmten Bereich der Spielwelt zu beobachten. Während es sich bei den meisten Objekten innerhalb eines Octrees um konkrete Elemente einer Spielwelt handelt, so handelt es sich bei Observern um Objekte, die den registrierten Beobachtungsbereich beschreiben. Wenn ein QuP-Cliet einen solchen registriert, so wird dieser auf dem, aus Position und View Box, berechneten Knoten abgelegt und der komplette Unterbaum, einschließlich dem Knoten auf dem der Observer liegt, vom QuP-Client abonniert. So können die für Spieler benötigten Bereiche über die Position und die View Box der Avatare berechnet werden.

2.3.4. Authentifizierungsserver

Die Aufgabe des Authentifizierungsserver ist das komplette Anmeldeverfahren des Systems abzuwickeln. Dazu melden sich die Spielclients, bevor sie sich mit den Spielserver verbinden, bei dem Authentifizierungsserver an. Dieser überprüft daraufhin, die Anmeldedaten und weist bei erfolgreicher Überprüfung, dem jeweiligen Spielclient einen Spielserver zu. Sollten sich ein Spielclient erfolgreich angemeldet haben, so wird neben dem Spielserver auch ein Token an den Spielclient sowie dem Spielserver übermittelt, welcher den Spielclient dazu berechtigt, sich an dem Spielserver anzumelden.

2.4. Kommunikation

Die komplette Spiellogik wird auf Seite des Spielservers berechnet, dies bewirkt einen hohen Grad an Kommunikation zwischen dem Spielclient und dem Spielserver. Zudem übermitteln die Spielserver jegliche Veränderungen an das QuP-System, welches wiederum die Änderungen an alle Abonnenten der geänderten Objekte übermittelt. Deshalb soll im Folgenden näher auf die Kommunikation im System eingegangen werden. Der Fokus soll hierbei vor allem

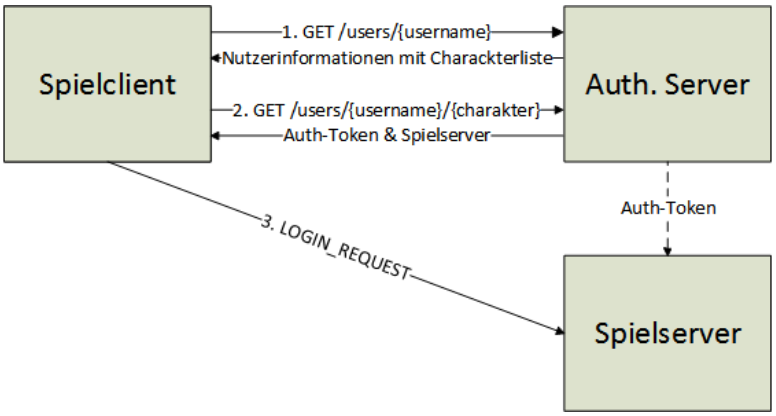


Abbildung 2.4.: Authentifizierungsvorgang

auf dem Austausch von Daten liegen, welche auch einen Einfluss auf das Load Balancing haben.

Im Allgemeinen kann Timadorus als ein Publish-/Subscribe-System betrachtet werden. So abonnieren Spieler sämtliche Bereiche, die von Interesse für sie sind und werden über Veränderungen in diesen Bereichen informiert. Das Abonnieren bzw. Kündigen von Abonnements wird hierbei jedoch nicht von den Spielclients direkt gemacht, sondern wird von dem Spielserver verwaltet. Dieser abonniert beim QuP-System, die von den Spielern benötigten Objekte und leitet Veränderungen direkt an die entsprechenden Clients weiter. Spielclients abonnieren Objekte bzw. kündigen Abonnements für Objekte somit durch bestimmte Aktionen implizit. So registriert der Spielserver, wenn sich ein Spieler bewegt und aktualisiert daraufhin die Abonnements für die Objekte des Sichtfeldes. Das Verwalten wird vom Spielserver über den integrierten QuP-Client geregelt. Dort registriert der Spielserver für jeden Spieler einen Observer.

2.4.1. Anmeldeverfahren

Um sich bei einem Spielserver anmelden zu können, muss der Spielclient zunächst den Authentifizierungsserver kontaktieren. Das Anmeldeverfahren dort läuft über HTTP Basic Access Authentication unter Angabe des Benutzernamens sowie des Passwortes. Über eine REST-Schnittstelle können, die restlichen benötigten Daten, wie der Spielcharackter oder der zugeordnete Spielserver, bezogen werden.

Diese ist wie folgt definiert:

GET /users/{username}

Liefert allgemeine Informationen zu dem Benutzer mit dem übergebenem Benutzernamen

Parameter

username Der Benutzername des Benutzers

Rückgabe

Ein JSON-Objekt mit folgenden Attributen:

name Der Benutzername des Benutzers

entities Eine Liste aller Charaktere des Benutzers

GET /users/{username}/{charakter}

Ermittelt den passenden Spielserver, sowie den Auth-Token

Parameter

username Der Benutzername des Benutzers

charakter Name des gewählten Charakters

Rückgabe

Ein JSON-Objekt mit folgenden Attributen:

name Der Name des Charakters

gameServer Der Spielserver mit dem sich der Client verbinden soll

authToken Ein Token, um den Spieler beim Spielserver zu verifizieren

sessionKey (optional) Ein Schlüssel zur symmetrischen AES-Verschlüsselung für Nachrichten zwischen Spielclient und Spielserver. Ist nur vorhanden, wenn der Authentifizierungsserver entsprechend konfiguriert ist.

Über diese kann sich der Spieler in drei Schritten Anmelden (siehe auch Abb. 2.4):

1. Anfrage an den Authentifizierungsserver per GET /users/{username}, um die Charaktere des Spielers zu erhalten.
2. Anfrage an den Authentifizierungsserver per GET /users/{username}/{charakter}, um den Auth-Token und den Spielserver zu bekommen.

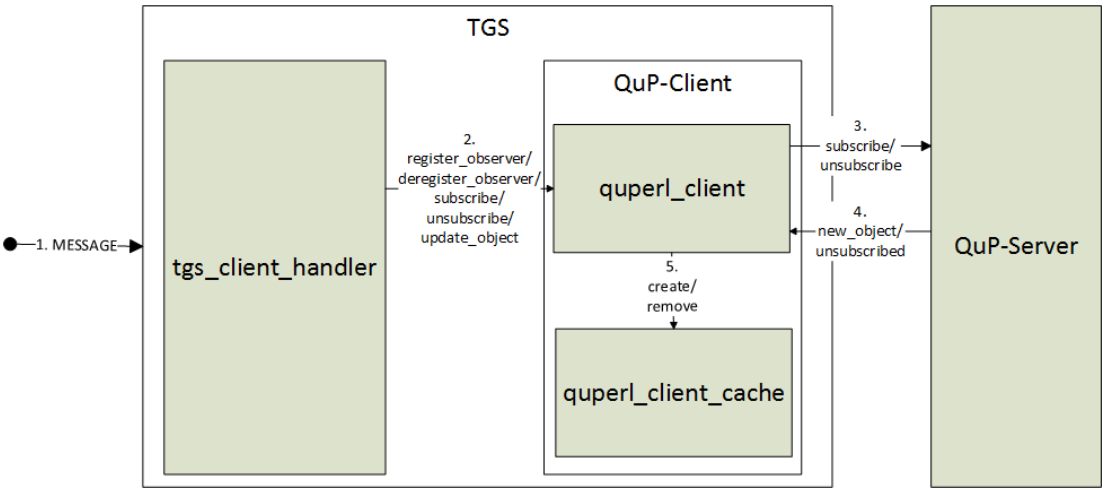


Abbildung 2.5.: Nachrichtenaustausch in Timadorus

3. LOGIN_REQUEST mit dem Auth-Token an den Spielserver

Der Spielclient erhält somit nach der zweiten Anfrage den Spielserver, mit dem er sich verbinden soll. Außerdem erhält der Spielclient einen Auth-Token, mit dem er sich beim Spielserver verifizieren kann. Dieser Token wird auch an den zugewiesenen Spielserver gesendet und setzt sich aus Benutzername, Charaktername, Spielserver und diversen weiteren Werten zusammen, die vom Authentifizierungsserver verschlüsselt wurden. Es handelt sich um eine symmetrische Verschlüsselung, dessen Schlüssel nur dem Authentifizierungsserver sowie dem Spielserver bekannt ist. So kann der Spielserver die Werte entschlüsseln und den Spieler verifizieren [Tima].

2.4.2. Nachrichtenaustausch

Der folgende Abschnitt soll einen Überblick über die Bedeutung des Nachrichtenaustausches innerhalb des Timadorus Systems für den Lastausgleich geben. Um die Bedeutung der Nachrichtenflüsse zu verstehen, muss deren Auswirkungen betrachtet werden. Abb. 2.5 soll dazu einen Überblick verschaffen, wie der Informationsaustausch innerhalb des Serversystems, nach Eintreffen einer Nachricht stattfindet. Zur besseren Darstellung wurde die Grafik stark vereinfacht und vereinzelt Komponenten ausgelassen. So gibt es z.B. innerhalb des QuP-Clients auch noch einen Subscription Manager, der das Abonnieren von Objekten verwaltet und somit dafür sorgt, dass nicht für jedes Objekt eine neue Anfrage an den Server gesendet werden muss. Auch sind zwischen den Komponenten nur Nachrichten aufgezählt, die für eine Veränderung der Last, im Sinne von mehr oder weniger genutztem Speicher, sorgen.

Nachdem eine Nachricht bei dem Spielserver eingetroffen ist (1), wird diese bearbeitet. Sollte sich daraus ergeben, dass sich der bestehende Datensatz auf dem Spielserver verändern muss, so wird der QuP-Client darüber informiert (2). Dies kann bedeuten, dass neue Bereiche der Spielwelt beobachtet werden, bzw. welche nicht mehr beobachtet werden (register_observer/deregister_observer), dass einzelne Objekte abonniert werden bzw. das Abonnement einzelner Objekte gekündigt wird (subscribe/unsubscribe) oder dass sich konkrete Objekte der Spielwelt verändern (update_object). Unter der Veränderung von Objekten fällt auch der Wechsel von Positionen der Spieler. Infolgedessen würde der Observer des Spielers von der alten Position deregistriert und ein neuer, auf der neue Position, registriert werden. Stellt der QuP-Client nach diesen Benachrichtigen fest, dass neue Objekte bezogen oder zu bisher vorliegende Objekten nicht mehr benötigt werden, so benachrichtigt dieser den QuP-Server (3). Ist die Anfrage erfolgreich verlaufen, antwortet der QuP-Server entsprechend mit *new_object* oder *unsubscribed*(4). In der Folge wird das QuP-Modul zur Verwaltung des Cache angesprochen (5). Dieser schreibt oder entfernt nun die entsprechenden Daten in oder aus den internen Speicher.

Timadorus Message Protokoll

Wie schon in Abschnitt 2.3.1 erwähnt, kommunizieren die Spielclients mit den Spielservern über das Timadorus Message Protokoll (TMP) [Timc]. Da jede Aktion des Spielers über das Protokoll dem Server mitgeteilt wird, sollen im Folgenden nur die Nachrichten betrachtet werden, die Einfluss auf die Lastverteilung nehmen können. Hiervon sind somit sämtliche Nachrichten ausgeschlossen, die vom Spielserver an den Spielclient gesendet werden.

LOGIN_REQUEST

Hierbei handelt es sich um die erste Anfrage, die ein jeder Spielclient an den Spielserver sendet, um dem Spiel beizutreten. Infolge der Anfrage wird ein Observer aus Spielerposition und Sichtbereich des Spielers ermittelt und dieser beim QuP-System registriert.

LEAVING

Diese Nachricht wird vom Spielclient gesendet, wenn dieser das Spiel verlassen möchte. Während des Abmeldeprozesses wird unter anderem der Observer des Spielers deregistriert.

MOVE

Um sich zu bewegen, muss der Spielclient diese Nachricht an den Spielserver senden. Die Bewegung bewirkt eine Veränderung des Sichtbereiches und somit des Observers des Spielers. Deshalb wird der Observer der alten Position, deregistriert und ein neuer für die neue Position registriert.

CONSUME

Über diese Nachricht wird mitgeteilt, dass der Spieler ein Objekt aufnehmen möchte. Da somit ein Objekt aus der Spielwelt verschwindet, kann dies zur Folge haben, dass auf vereinzelten Servern dieses Objekt aus dem Speicher genommen werden muss. Die Auswirkungen sind pro CONSUME-Nachricht jedoch als gering zu betrachten, da es sich nur um einzelne Objekte handelt, die verschwinden können.

KICK

Diese Nachricht wird von einem Spielleiter gesendet, wenn ein Spieler aus dem Spiel entfernt werden soll. Da hier ein Spieler die Spielwelt verlassen muss, sind die Auswirkungen die selben wie bei *LEAVING*.

BAN

Diese Nachricht ist vergleichbar mit *KICK*. Der Unterschied besteht darin, dass der Spieler für eine bestimmte Zeit komplett vom Spiel gesperrt ist und somit noch eine Zeit angegeben werden muss, für die ein Spieler nicht mehr am Spiel teilnehmen darf. Folglich muss auch hier ein Spieler die Spielwelt verlassen. Somit sind auch hier die Folgen die gleichen wie bei *LEAVING*.

3. Bisherige Verfahren zur Lastverteilung

Dieses Kapitel soll einen Überblick über gängige Methoden der Lastverteilung in DVEs geben. Dazu werden zunächst Verfahren zur Lastverteilung in populären Online-Rollenspielen betrachtet. Anschließend werden andere Wissenschaftliche Arbeiten, im Bereich Lastverteilung in DVEs, betrachtet. Zum Abschluss werden die Verfahren zusammengefasst und ihre Tauglichkeit, in Bezug auf Timadorus, bewertet.

3.1. Lastverteilung in Online-Spielen

Im Folgenden sollen die Verfahren zur Lastverteilung in bereits bestehenden Online-Spielen betrachtet werden. Stellvertretend werden in diesem Abschnitt Eve Online, World of Warcraft sowie Second Life betrachtet.

3.1.1. Eve Online

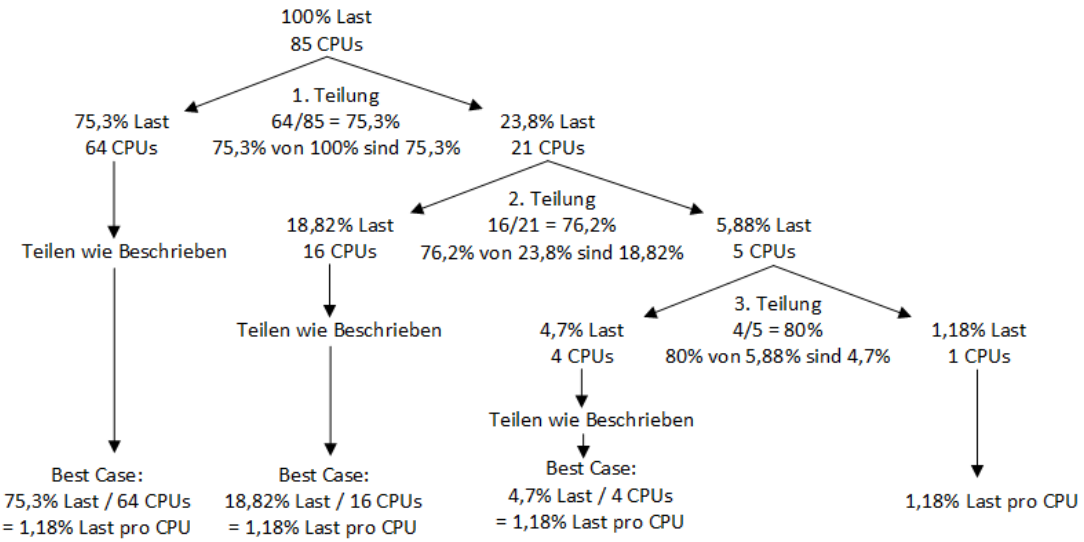


Abbildung 3.1.: Aufteilung der Spielwelt in Eve Online. Nach [Eve13].

Bei Eve Online befinden sich sämtliche Spieler innerhalb einer Spielwelt. Da so die Server nicht durch Sharding entlastet werden können, ist es nötig, dass Teile der Spielwelt dynamisch den Servern zugeordnet werden. Dazu wird die Spielwelt in Eve Online in Systeme eingeteilt. Zum Load Balancing wird bei Eve Online ein sogenannter Static Cluster Premapper eingesetzt. Dieser erstellt einen Fingerabdruck des Lastverhaltens der Systeme und teilt diese, basierend auf eine Schätzung, den einzelnen CPUs zu. Beim zuweisen wird darauf geachtet, dass einer CPU immer zusammenhängende Systeme zugewiesen werden. Dazu wird die Spielwelt in mehrere Teile geteilt. Das Schema lautet wie folgt:

1. Teile die Spielwelt in zwei Teile mit gleich starker Auslastung
2. Teile das stärker ausgelastete Teil der beiden nun bestehenden Teile in zwei Teile mit gleich starker Auslastung
3. Teile das am stärksten ausgelastete Teil der drei nun bestehenden Teile in zwei Teile mit gleich starker Auslastung
4. Wiederhole dies bis gleich viele Teile wie CPUs vorliegen

Dies sorgt allerdings nur für eine gleichmäßige Aufteilung, wenn die Anzahl der CPUs einer Zweierpotenz entspricht. Deshalb wird zunächst versucht die Spielwelt so aufzuteilen, dass ein Teil entsteht, dessen Anzahl an CPUs, auf die diese aufgeteilt wird, einer Zweierpotenz entspricht. Abb. 3.1 soll das illustrieren. Dazu wird zunächst aus der Anzahl der vorliegenden CPUs die nächstkleinere Zweierpotenz ermittelt und berechnet, wie viel Prozent der Gesamtanzahl der CPUs das entspricht. Entsprechend wird die Last nun aufgeteilt. Der Teil, der aus der nächstkleineren Zweierpotenz ermittelt wurde, kann nun genauso aufgeteilt werden wie zuvor beschrieben. Mit dem bestehenden Rest wird genauso Verfahren wie zuvor. Es wird der prozentuale Anteil der nächstkleineren Zweierpotenz, der noch übrigen CPUs, ermittelt und die Spielweltlast entsprechend aufgeteilt. Dies geschieht nun so lange, bis keine weitere Unterteilung mehr möglich ist [Eve13].

3.1.2. World of Warcraft

Um die Last zu verteilen wird in World of Warcraft mit Shards gearbeitet (in World of Warcraft auch Realms genannt). Von diesen gibt es über 700 Stück, aufgeteilt auf mehreren Datenzentren weltweit [LC10]. Dadurch gibt es viele unabhängige Kopien der Spielwelt. Da die Spielerzahl in jedem Shard begrenzt ist, gibt es die Möglichkeit, dass zusätzliche Servercluster mit weiteren Shards gestartet werden, sobald die bestehenden überlastet sind [MO12]. Zusätzlich wird die

Spielwelt innerhalb der Realms, in mehrere Zonen unterteilt, die jeweils von eigenen Servern verwaltet werden [AIB09]. Um Synchronisierungsprobleme zu vermeiden, können Spieler verschiedener Zonen, jedoch nicht miteinander interagieren [DL14].

3.1.3. Second Life

Bei Second Life befinden sich ähnlich wie bei Eve Online alle Spieler in einer Spielwelt. Dennoch wird dort auf eine statische Partitionierung der Spielwelt gesetzt. Jeder Server verwaltet ein Gebiet von 256 x 256 Meter [MO12, RO03]. Da sämtliche Gebiete in einem Gitter angeordnet sind, kann jeder Server bis zu vier Nachbarn haben. Sollte sich die Spielwelt nun vergrößern, so wird ein zusätzlicher Spielserver hinzugefügt, der für eine weitere Fläche zuständig ist [RO03]. Ähnlich wie bei World of Warcraft können allerdings auch hier Spieler aus verschiedenen Partitionen nicht miteinander interagieren [DL14].

3.2. Bisherige Arbeiten

Eine Vielzahl von wissenschaftlichen Arbeiten sind zum Thema Lastverteilung in DVEs bereits entstanden. In den meisten Arbeiten wird eine Partitionierung der Spielwelt vorgeschlagen, bei der jeder Server einen Teil der Spielwelt verwaltet. Ng et al. schlagen dazu vor, dass jeder Server eine Region verwaltet [NSLL02]. Jede dieser Regionen ist dazu in weitere kleinere Zellen unterteilt, die Spielserver bei hoher Last an andere Spielserver abtreten können. Die Lastinformationen der Spielserver werden dort in regelmäßigen Abständen an einen zentralen Server, dem sogenannten Load Collector, übermittelt. Ist ein Spielserver nun überlastet, so erkundigt sich dieser beim Load Collector nach den Lastinformationen seiner Nachbarserver und überträgt einen Teil der verwalteten Zellen an den Server mit der geringsten Last [NSLL02]. Da dadurch keine globalen Informationen betrachtet werden, kann die Last ungleichmäßig verschoben werden, sodass kaum beanspruchte Server nicht beachtet werden [Lau10]. Deshalb entwickelte Lau dies zu einem hybriden Verfahren weiter [Lau10]. In diesem übermittelt der Load Collector neben Informationen der direkten Nachbarserver, auch Lastinformationen der anderen Spielserver. Aus diesen wird versucht ein Cluster aus mehreren Servern mit geringer Last zu ermitteln. Nachdem nun potentielle Nachbarserver zur Lastübernahme ermittelt wurden, wird ein Teil der Last an den Server übertragen, welcher sich am nächsten zu diesem Cluster befindet [Lau10]. Lau konnte in seiner Arbeit nachweisen, dass dieses Verfahren zu deutlich weniger überlasteten Servern führt, als das von Ng e al. [Lau10].

Lui und Chan lösen die Lastverteilung über lineare Optimierung. Dazu betrachten sie ne-

ben der Last der Server auch die Kommunikation der Spielserver untereinander [LC02]. Ziel des Verfahrens ist es, die Last möglichst gleichmäßig über alle Server zu verteilen und dabei die Interserverkommunikation zu minimieren. Zunächst wird die über Rekursive Bisektion nach einer ausgeglicheneren Partitionierung der Spielwelt gesucht. Im nächsten Schritt wird ein Graph über alle vorliegenden Spieler aufgebaut und eine Kante zwischen zwei Spielern gezogen, wenn der andere sich in deren Interessengebiet befindet. Darüber soll eine bessere Lastverteilung erreicht werden, indem auf dem Graphen nach Spielern gesucht wird, die Überschneidungen zu Spielern auf anderen Servern aufweisen und überprüft wird, ob sich die Kosten verringern, wenn diese auf den anderen Server wechseln. In einem letzten Schritt wird die Kommunikation zwischen den Server optimiert, indem Spieler mit einem starken Kommunikationsaufkommen zu anderen Spielservern, auf diese wechseln. Dies wird so lange wiederholt, bis die Kosten unter einen bestimmten Schwellwert fallen [LC02]. Damit die Rechenzeit möglichst gering ist, wurde zudem eine parallelisierte Variante entwickelt [LC02]. Morillo, Fernández und Orduña greifen in ihrem Verfahren auf die Kostenfunktion von Lui und Chan zurück, da auch aus ihrer Sicht das Ziel einer guten Lastverteilung sein sollte, dass Last gleichmäßig verteilt und die Interserverkommunikation minimiert wird [MFOn03]. Um eine bessere Aufteilung der Spielwelt zu ermitteln, wird auch hier ein heuristisches Suchverfahren verwendet. In diesem Fall wird mithilfe eines Ameisenalgorithmusses versucht, die Aufteilung zu optimieren. Dazu werden sämtliche Avatare betrachtet, deren Interessengebiete sich mit anderen Spielservern überschneiden. Jede Ameise versucht nun mithilfe einer Bewertungsfunktion, die Spieler neu aufzuteilen. Die neue Aufteilung wird daraufhin mithilfe die Kostenfunktion von Lui und Chan evaluiert. Stellen sich die Kosten nun als besser heraus, so wird diese neue Konfiguration übernommen [MFOn03]. In Experimenten stellte sich heraus, dass ihr Verfahren bei kleinen Spielwelten (13 Avatare und drei Spielserver) schlechtere Werte hat als das von Lui und Chan erzielt. Bei großen Spielwelten (2500 Avatare und acht Spielserver) konnten jedoch bessere Werte erzielt werden. So war das Verfahren deutlich schneller und konnte, außer bei gleichmäßiger Verteilung der Spieler, stets bessere Kostenwerte erzeugen [MFOn03].

Bei Chen et al. wird gar nicht versucht, die Last gleichmäßig über alle Spielserver zu verteilen. Stattdessen gibt es ein fest definiertes Quality of Service (QoS), bestehend aus CPU-Auslastung und Netzwerkauslastung, die jeder Server erfüllen muss. Wird der Schwellwert für einen der beiden Werte überschritten, so wird das Load Balancing ausgelöst [CWD+05]. Die Spielwelt ist dazu in Regionen unterteilt, wovon jeder Spielserver mehrere verwalten kann. Sollte dieser nun eine Überlastung feststellen, so versucht dieser Last an Nachbarserver abzugeben. Dazu wird zunächst nach Nachbarn geschaut, bei denen das QoS noch eingehalten wird. An diese

werden daraufhin Regionen abgetreten. Sollte kein geeigneter Nachbarserver gefunden werden, so gibt es auch die Möglichkeit, dass Regionen an entfernte Server abgetreten werden. Dies kann zu einer Zerstückelung der verwalteten Gebiete und somit zu einer übermäßigen Interserverkommunikation führen. Sollte dies festgestellt werden, so wird versucht, Regionen erneut zusammenzufassen [CWD+05].

Da eine statische Partitionierung in vielen Online-Spielen der Standard ist, jedoch der Dynamik eines Online-Spieles nicht gerecht wird, versuchen De Vleeschauwer et al. das Konzept mit Mikrozellen weiter zu entwickeln [DVV+05]. Dazu wird die Spielwelt in viele kleine, sogenannte Mikrozellen, unterteilt. Wovon jeder Spielserver mehrere verwaltet. Jeder dieser Zellen ist für sich eigenständig und muss daher mit den anderen Zellen kommunizieren, wenn es Überschneidungen bei den Interessengebieten der Spieler gibt. Zur Aufteilung der Mikrozellen haben De Vleeschauwer et al. mehrere Algorithmen vorgeschlagen. So kann sich die Verteilung nur nach der Last richten, was allerdings zu einer starken Zerstückelung der Spielwelt führt, oder es kann versucht werden, zusammenhängende Cluster zu bilden. Andere Vorschläge sind die Mikrozellenaufteilung über Simulated Annealing oder ganzzahlige lineare Programmierung zu optimieren. Während Simulated Annealing meist gute Ergebnisse liefert, kann über ganzzahlige lineare Programmierung sogar immer eine optimale Lösung gefunden werden, bei der die berechneten Kosten minimal sind. Die Rechenkosten sind bei ganzzahliger linearer Programmierung jedoch viel zu hoch, als dass dies eine einsetzbare Lösung darstellt [DVV+05]. Zudem hat Deng festgestellt, dass die vorgeschlagenen Algorithmen allgemein sehr teuer sind [DL12].

Kazem, Ahmed und Shirmohammadi sind dagegen der Auffassung, dass Load Balancing Algorithmen für Online-Spiele möglichst simpel sein sollten, damit sie nicht zu viel Rechenleistung aufbrauchen. Deshalb setzen sie auf eine bereits vor Programmstart festgelegte Aufteilung, bei der die Spielwelt in Hexagone aufgeteilt wird. Jeder Spielserver verwaltet ein Hexagon. Sollte ein Spielserver überlastet sein, so werden fest definierte Randgebiete zur Verwaltung an die Nachbarserver abgetreten. Sollte die Last eines der zuvor überlasteten Server nun unter einen bestimmten Schwellwert fallen, so übernimmt dieser wieder die Verwaltung für seine Randgebiete [KAS07]. Deng, Lau und Rynson setzen bei ihrem Verfahren auf ein Wärmediffusionsverfahren [DL12]. Ziel ist es, die Last auf alle Server gleichmäßig zu verteilen. Die Spielwelt wird dazu in mehrere Partitionen unterteilt, für die jeweils ein Server zuständig ist. Jeder dieser Partitionen ist wiederum in kleinere Zellen unterteilt. Die Last der Server wird nun mithilfe der Wärmeleitungsgleichnung umverteilt. Dabei wird die Last einer Partition

über Wärme dargestellt. Über ein Glättungsverfahren wird die Hitze von warm nach kalt geleitet. Dazu wird zunächst global der Fluss berechnet und anschließend, auf lokaler Ebene, die Last umverteilt, indem die Zellen mit hoher Last, die sich am Rand einer Partition befinden, auf den Nachbarserver geschoben werden [DL12]. In einer späteren Arbeit haben Deng, Lau und Rynson dies zu einem rein lokal arbeitendem Verfahren weiterentwickelt, indem zur Umverteilung nur die Lastinformationen der benachbarten Server benötigt werden [DL14]. Dort wurde auch gezeigt, dass das Verfahren effizienter in der Vermeidung überlasteter Server ist, als die Verfahren von Ng et al. [NSLL02], Chen et al. [CWD+05] und das Verfahren von Lau [Lau10].

Ein anderes Verfahren von Morillo et al. verfolgt einen vollkommen anderen Ansatz. In diesem übernehmen die Server nicht die Verwaltung von Regionen, sondern sind an die ihnen zugewiesenen Avatare gebunden [MOnFD03]. Morillo et al. gehen davon aus, dass es nur wichtig ist, die Last der Server unter 100 % zu halten. Deswegen startet der Prozess zur Lastverteilung, wenn die Last eines Servers auf 99 % gestiegen ist. Die Spieler auf dem Spielserver werden dann nach dem sogenannten Präsenzfaktor sortiert. Dieses stellt die Anzahl der Interessengebiete, in denen der Spieler auftaucht, dar. Anschließend werden diese Spieler auf den am schwächsten ausgelasteten Server geschoben, bis die Last um 10 % verringert wurde [MOnFD03]. In Experimenten konnte gezeigt werden, dass so tatsächlich eine Überlastung von Servern besser vermieden werden kann, als in dem Verfahren von Ng et al. [MOnFD03]. Eine Aussage über die Interserverkommunikation wurde in der Arbeit jedoch nicht getroffen. Lu, Parkin und Morgan schlagen ein ähnliches Verfahren vor. Auch bei ihnen findet keine Regionalisierung statt, da sich aus ihrer Sicht Interserverkommunikation dadurch nicht signifikant genug verringern lässt. Stattdessen wollen sie das Load Balancing für Online-Spiele vereinfachen. Deswegen befindet sich bei ihrer Architektur ein einfacher NAT-Server zwischen den Clients sowie den Servern. Dieser kann mit einem Standard Load Balancing Verfahren, z. B. Round Robin, ausgestattet werden [LPM06].

3.3. Bewertung der bisherigen Verfahren

Eine Vielzahl von Verfahren zur Lastverteilung in virtuellen Umgebungen sind bereits entwickelt worden. Ein Großteil der Verfahren teilt die Spielwelt auf die Spielserver auf. Während bei kommerziellen Spielen die statische Aufteilung, aufgrund der einfachen Umsetzbarkeit sehr beliebt ist [RO03, MO12, AIB09], konzentrieren sich wissenschaftliche in erster Linie auf eine dynamische Partitionierung der Spielwelt, bei denen es auch möglich ist, bei hoher Last

einen Teil der Spielwelt an andere Spielserver abzugeben.

Da innerhalb von Timadorus jedoch die Spielserver keine regionale Zuordnung haben, sondern die Bereiche verwalten, die sich im Interessengebiet der Spieler auf den Servern befinden, kommen diese Lösungsansätze nicht in Frage. Es gibt allerdings auch Arbeiten, welche sich mit dieser Form der Zuordnung schon auseinandergesetzt haben [MOnFD03, LPM06]. Diese schlagen jedoch einfache Load Balancing Verfahren, wie Round Robin, für diese Art der Spielverwaltung vor. In früheren Arbeiten wurde jedoch auch gezeigt, dass durch eine gute Gruppierung der Spieler die Interserverkommunikation deutlich gesenkt werden kann [LC02, NSLL02]. Deshalb sollen auch die Verfahren in dieser Arbeit, Spieler nach Möglichkeit, regional gruppieren. Bei Timadorus gibt es zwar keine direkte Kommunikation zwischen den Spielservern, jedoch würde auch hier die Kommunikation deutlich reduziert werden, da zum einen die Update-Nachrichten des QuP-Systems an weniger Spielserver gesendet werden müssten und zum anderen das Risiko von Kollision beim Manipulieren von Spielobjekten, deutlich gesenkt wäre. Zeitgleich könnte die Stärke nicht regionalisierender Algorithmen genutzt werden, die sehr gut Überlastungen verhindern können [MOnFD03] und bei starker Last, Spieler auf beliebige Server geschoben werden.

Ein solches Verfahren ist jedoch bis heute noch nicht entwickelt worden. Deshalb beschäftigt sich diese Arbeit mit der Entwicklung eines solchen Verfahrens, bei dem auch auf Techniken bereits vorhandener Verfahren zurückgegriffen wird.

4. Anforderungen

Aus dem in Kapitel 2 beschriebenen Timadorus System, ergeben sich ganz spezielle Anforderungen an den, zu entwickelnden, Load Balancer. Diese sollen in diesem Kapitel dargestellt werden. Dazu werden zunächst die funktionalen Anforderungen beschrieben. Anschließend werden die nichtfunktionalen Anforderungen beschrieben. Diese beinhaltet neben den allgemeinen Anforderungen, auch die Anforderungen an die Architektur. Dazu werden gängige Modelle betrachtet und auf ihre Machbarkeit analysiert.

4.1. Funktionale Anforderungen

Zunächst sollen die funktionalen Anforderungen an den Load Balancer beschrieben werden, darunter fallen vor allem Funktionalitäten, die der Load Balancer für andere im System befindlichen Komponenten bereitstellt.

Spielserver müssen sich an- und abmelden Sobald ein neuer Spielserver im System ist, muss dieser beim Load Balancer, als zur Verfügung stehender Spielserver, aufgenommen werden. Genauso muss ein Spielserver aus der Liste, der zur Verfügung stehenden Spielserver entfernt werden, sobald dieser sich beendet. Dies soll möglichst automatisiert geschehen.

Der Load Balancer muss einen Spielserver für Spieler auf Anfrage ermitteln können Da der Authentifizierungsserver den Spielserver an den Spielclient übergibt, muss für diesen eine Schnittstelle bereitgestellt werden, über die der Authentifizierungsserver nach einen Spielserver anfragen kann.

Spielserver müssen ihren Zustand an den Load Balancer übermitteln Um ein dynamisches Load Balancing zu ermöglichen, benötigt der Load Balancer Informationen über den aktuellen Zustand der Spielwelt sowie den Zuständen der Spielserver. Dazu müssen diese an den Load Balancer übertragen werden. Dies soll nach Möglichkeit automatisiert geschehen.

4.2. Nichtfunktionale Anforderungen

Neben den funktionalen Anforderungen gibt es auch welche, die sich aus dem bereits vorliegenden Systemen bzw. aus den Anforderungen des Timadorus Projektes ergeben [Timb]. Diese sollen im Folgenden erörtert werden.

Effizienz Spieler erwarten von Online-Spielen, dass diese schnell reagieren und die Latenzzeiten gering sind [MO12]. Deshalb muss der Einfluss der Lastverteilung auf die Spielserver möglichst gering bleiben. Da der Load Balancer als separater Server realisiert wird, betrifft dies vor allem die Übertragung der Spielserverzustände.

Eine Effizienz im Sinne von Dauer der Zuweisung spielt bei unserem Verfahren eine untergeordnete Rolle, da es sich um die initiale Zuweisung handelt und die Dauer somit keinen Einfluss auf den Spielfluss hat. Diese muss einzig in einem für den Spieler akzeptablen Zeitrahmen bleiben. Da in dem Anmeldeverfahren auch andere Komponenten verwickelt sind, soll die maximal akzeptierte Dauer bei einer Sekunde liegen.

Gruppierung der Spieler Mit Gruppierung der Spieler ist gemeint, dass Spieler, die sich nah beieinander befinden, nach Möglichkeit einem Spielserver zugewiesen werden. Dies hat den Effekt, dass es weniger Überschneidungen zwischen den View Boxen der verschiedenen Spielserver gibt. Dadurch werden seltener Objekte von mehreren Spielservern benötigt. Dies soll dafür sorgen, dass weniger Speicher im Gesamtsystem verbraucht wird. Außerdem kann dadurch die Rate der Kollisionen auf dem QuP-System verringert werden, da weniger Spielserver dieselben Objekte verwalten und somit auch seltener zeitgleich manipulieren.

Kein Sharding Eine beliebte Möglichkeit, um Last zu verteilen, ist das sogenannte Sharding. Hierbei wird die Spielwelt mehrfach repliziert, sodass mehrere Kopien auf verschiedenen Spielservern vorliegen. Dies macht zwar die Lastverteilung sehr simpel, da bei hoher Last einfach ein zusätzlicher Server gestartet werden muss, hat allerdings den Nachteil, dass Spieler die miteinander interagieren wollen, sich auch auf demselben Server befinden müssen [MO12, RO03]. Tun sie dies nicht, so ist ein gemeinsames Spielerlebnis nicht möglich. Dies widerspricht jedoch den Anforderungen von Timadorus. Dort soll eine große Spielwelt entwickelt werden, in der alle Spieler ein gemeinsames Spielerlebnis haben können.

Integrierbarkeit verschiedener Verfahren Da in dieser Arbeit mehrere Verfahren entwickelt werden, ist es notwendig, diese einfach zu integrieren. Dazu soll das System in

der Lage sein, dass zwischen diesen Verfahren gewechselt werden kann, ohne den Programmcode anzupassen.

4.2.1. Architektur

Die Auswahl eines Architekturmodells ist ein zentraler Bestandteil zur Realisierung eines Load Balancers. Deshalb sollen zunächst verschiedene Architekturmodelle betrachtet und anschließend bewertet werden, inwiefern diese für das Timadorus System infrage kommen.

Clientbasiert

Bei clientbasierten Methoden entscheidet der Client darüber, welcher Server ausgewählt wird. Dazu muss diesem jeder Zeit bekannt sein, welche Server ihm zur Verfügung stehen. Einer der bekanntesten Implementierungen dieses Ansatzes stellt der Netscape Browser dar. Sobald eine Verbindung zu *netscape.com* aufgebaut wird, ermittelt dieser eine Zufallszahl zwischen eins und der Anzahl der vorhandenen Server und verbindet den Nutzer mit dem zur Zahl gehörenden Server [CCY99].

Beurteilung Durch einen clientbasierten Ansatz könnte das Serversystem entlastet werden, da die Zuweisungsentscheidung auf Seiten der Spielclients stattfinden würde. Jedoch müssten die Clients dazu den Status eines jeden Spielservers einsammeln, damit eine Entscheidung getroffen werden kann, die in Abhängigkeit von der Belegung der einzelnen Server steht. Dazu müssten diese alle Spielserver des Systems kennen. Da Spielserver dynamisch zum System hinzustoßen können und nicht statisch verankert sind, müsste zusätzlich ein System entstehen, bei dem sich die Spielserver anmelden und die Clients nach den aktuell vorhandenen Spielservern fragen können. Außerdem müsste eine Möglichkeit gefunden werden, dass eine koordinierte Zuweisung über die Spielclients hinweg stattfinden könnte. Dies wäre nötig, da es bei zeitgleicher Zuweisung dazu kommen könnte, dass Spieler mit großer Distanz dem selben Spielserver zugewiesen werden und diese dadurch nicht mehr gruppiert werden. Neben den kommunikativen Problemen, müsste auch die Architektur des Systems stark angepasst werden, da bisher der Client den Spielserver über den Authentifizierungsserver übermittelt bekommt und eine clientseitige Zuweisung nicht vorgesehen ist. In der Folge müsste auch das Sicherheitskonzept überarbeitet werden, da der Auth-Token in der aktuellen Form nicht auf Seiten des Authentifizierungsservers generiert werden könnte. Denn dieser beinhaltet den zugewiesenen Spielserver in verschlüsselter Form und müsste auch an diesen gesendet werden. Zudem gäbe es keine Möglichkeit, die Zuweisung zentral zu kontrollieren.

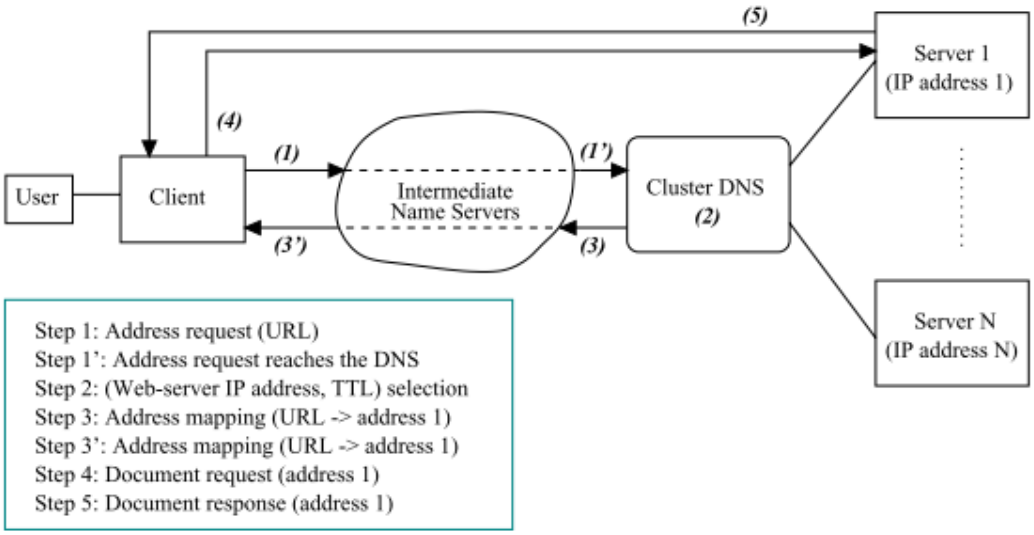


Abbildung 4.1.: DNS basierte Lastverteilung. Aus [CCY99].

Vorteile

- Entlastung des Serversystems

Nachteile

- Koordination der Clients schwierig
- Architektur müsste stark angepasst werden
- Keinerlei zentrale Kontrolle über Zuweisung

Fazit Ein clientbasierter Ansatz stellt für Timadorus keine geeignete Lösung dar. Die Entlastung des Serversysems ist im Vergleich kein gravierender Vorteil. So wäre eine Anpassung der Architektur höchst aufwendig und der Athentifizierungsvorgang müsste komplett überarbeitet werden. Außerdem ist eine geeignete Zuweisungsstrategie, die über die Clients hinweg gut funktioniert, nur schwer zu realisieren.

DNS-Basiert

Bei DNS-basierter Lastverteilung (Abb. 4.1) gibt es einen zentralen DNS-Server. Der Client fragt diesen mit einem symbolischen Namen an und erhält vom DNS-Server eine gültige IP-Addresse mit der sich der Client verbindet. Während des Übersetzungsvorganges, von symbolischem Namen hin zu einer gültigen IP-Addresse, hat der DNS-Server die Möglichkeit, zwischen

den einzelnen Servern des Clusters zu wählen. Dabei können auf Seiten des DNS-Servers die verschiedensten Zuweisungsalgorithmen verwenden werden. Jedoch gibt es keine Möglichkeit der Kontrolle, der einzelnen Anfragen an den Server [CCY99]. Bei vielen großen Webseiten wie Ebay [eBa15] oder Google [NSHW11, S. 962] kommt das sogenannte DNS-Round Robin zum Einsatz. Bei diesem werden mehrere Server für einen Namen gelistet und diese im Wechsel ausgewählt [TS08, S. 602].

Beurteilung Eine klassische Namensauflösung muss zur Lastverteilung im Timadorus System nicht stattfinden, da hier kein Name zu einer IP-Addresse aufgelöst werden muss, sondern die Spielerdaten als Grundlage der Suche dienen. Eine Anpassung des Konzeptes ist jedoch relativ simpel, da in diesem Fall die Namensauflösung wegfällt und nur eine Serversuche auf Basis der Spielerdaten gemacht werden muss. Die Integration eines DNS-basierten Ansatzes in das Timadorus-System wäre mit nur geringen Anpassungen möglich. Dieser könnte als eigenständiger Server umgesetzt und integriert werden. So könnte der Authentifizierungsserver bei diesem nach einem geeignetem Spielserver fragen und die Antwort an den Spielclient weiter geben. Hinzu kommt, dass eine direkte Verbindung zwischen dem Client sowie dem Spielserver bestehen kann und der Load Balancer nicht dazwischen hängt. Da eine sehr hohe Verkehrslast zwischen den Spielclients und dem Spielserver besteht, würde dies verhindern, dass der Load Balancer zum Flaschenhals für eintreffende Nachrichten wird. Jedoch gäbe es keine Kontrolle über die eintreffenden Nachrichten. Dieser Ansatz bietet allerdings die Möglichkeit auch komplexere Algorithmen umzusetzen. So könnten zustandsorientierte Algorithmen implementiert werden, bei denen die Spielserver regelmäßig den Load Balancer über ihren Zustand informieren.

Vorteile

- Einfach zu integrieren
- Direkte Verbindung von Spielclients und Spielservern
- Auch komplexe Algorithmen möglich

Nachteile

- Timadorus benötigt keine Namensauflösung
- Keine Kontrolle über eintreffende Nachrichten

Fazit Für Timadorus scheint die grundlegende Technik durchaus geeignet zu sein. Zwar muss keine Namesauflösung stattfinden, allerdings gibt es eine zentrale Instanz, die leicht in das

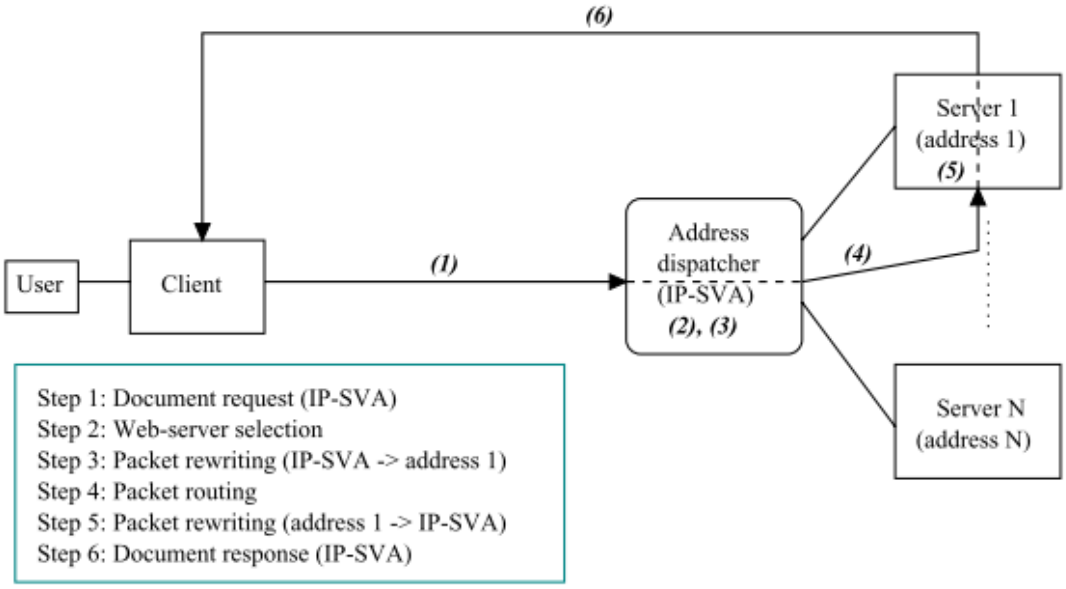


Abbildung 4.2.: Dispatcher basierte Lastverteilung mit einfachem Umschreiben des IP-Haeders. Aus [CCY99].

System zu integrieren ist und die direkt vom Authentifizierungsserver angesprochen werden kann. Zudem verhindert die direkte Verbindung von Spielclient und Spielserver, dass der Load Balancer zum Flaschenhals werden kann. Eine Einsicht oder Manipulation der einzelnen Nachrichten ist für den Load Balancer nicht zwingend nötig. Somit ist dies kein Nachteil für das System.

Dispatcherbasiert

Bei einer dispatcherbasierten Lastverteilung (Abb. 4.2), wird diese über einen Dispatcher geregelt, welcher sich zwischen den Clients und den Servern befindet. Dieser nimmt die Anfrage entgegen und verteilt diese auf die einzelnen Server. Dadurch erhält der Load Balancer volle Kontrolle über sämtliche Anfragen an das System. Ein Dispatchersystem kann in mehreren Varianten umgesetzt werden. So kann die Nachricht einfach zu den Server weitergeleitet werden oder der IP-Header manipuliert und die Zieladresse durch den richtigen Server ersetzt werden. Das manipulieren kann auf zwei Arten geschehen. Zum einen mit einem einfachen Umschreiben des Headers, bei dem nur die Zieladresse geändert wird und zum anderen durch doppeltes Umschreiben, bei dem auch die Quelladresse geändert wird. Dadurch wird die Antwort nicht direkt an den Client zurück gesendet, sondern geht zunächst über den Dispatcher an den

Client zurück. Dadurch hat der Dispatcher nicht nur Kontrolle über eintreffende Nachrichten, sondern auch über deren Antworten. Da ein Dispatcher zwischen Client und den Servern liegt, werden in der Regel einfache Scheduling Verfahren, wie Round Robin oder Least Loaded verwendet [CCY99].

Beurteilung Durch die Umsetzung eines dispatcherbasierten Ansatzes, wäre eine direkte Verbindung von Spielclients sowie Spielserver nicht möglich. Dadurch kann sich der Dispatcher relativ schnell als Flaschenhals erweisen, da aufgrund der Komplexität zur Ermittlung eines geeigneten Spielservers nicht auf einfache Algorithmen zurückgegriffen werden kann, um alle Zuweisungskriterien zu erfüllen. So muss unter anderem berücksichtigt werden, welche Gebiete von den einzelnen Spielservern belegt worden sind, um einen Spielserver zu finden, der möglichst wenig Objekte nachladen muss. Auch müsste ähnlich wie bei einem clientbasiertem Ansatz die Architektur deutlich überarbeitet werden. So wäre es obsolet, wenn der Authentifizierungsserver einen zugewiesenen Spielserver zurück liefert, da der Dispatcher für die Zuweisung zuständig sein soll. Dadurch müsste auch hier das Sicherheitskonzept komplett überarbeitet werden, da der Authentifizierungsserver, ohne dass ihm der zugewiesene Spielserver bekannt ist, den Auth-Token in der jetzigen Form nicht generieren und auch an keinen Spielserver senden kann.

Vorteile

- Einfach zu implementieren
- Vollständige Kontrolle über eintreffende Nachrichten

Nachteile

- Starke Veränderung der Architektur nötig
- Dispatcher kann schnell zum Flaschenhals werden

Fazit Ein dispatcherbasiertes Verfahren stellt für Timadorus keine geeignete Lösung dar. So müsste die Architektur stark verändert werden, da es keinen fest zugewiesenen Spielserver gäbe. Hinzu käme, dass sich der Dispatcher mit einer hohen Wahrscheinlichkeit als Flaschenhals herausstellen kann, da ein sehr hohes Kommunikationsaufkommen zwischen Spielclients sowie Spielservern vorliegt. Zudem ist eine Kontrolle der Nachrichten durch den Load Balancer ist in Timadorus nicht von Nöten.

Serverbasiert

Serverbasierte Methoden basieren auf den DNS-basierten Methoden, diese versuchen jedoch, die Zuweisung zu dezentralisieren. So können die einzelnen Server, die vorgegebene Zuweisung manipulieren und die Clients einem anderen Servern zuweisen. Dies kann aus Sicht des Clients transparent sowie nicht transparent geschehen. Bei transparenten Methoden, wird die Nachricht vom Server direkt an einen anderen Server weitergeleitet, welcher die Antwort an den Client sendet. Somit bekommt der Client nicht mit, wenn seine Anfrage an einen anderen Server weitergeleitet wurde. Bei nicht transparenten Systemen wird dem Client in der Regel mitgeteilt, dass dieser einen anderen Server verwenden soll, welcher sich daraufhin mit diesem verbindet. Ein Beispiel eines solchen Verfahrens ist das HTTP-Redirect [CCY99].

Beurteilung Ein serverbasierter Ansatz könnte gut als Erweiterung zu einem DNS-basierten Ansatz dienen. Durch das dynamische weiterleiten an einen neuen Spielserver, wäre auch eine bessere Skalierbarkeit des Systems gesichert. So könnten Spielserver von sich aus feststellen, dass die Last auf ihnen nicht mehr optimal ist bzw. dass die regionale Zuordnung nicht mehr gut genug gewährleistet ist und die Spieler entsprechend einem neuen Spielserver zugewiesen werden. Dazu müsste es zusätzlich eine Möglichkeit geben, dass die Spielserver untereinander kommunizieren, damit diese beurteilen können, welche anderen Spielserver geeignet wären. Dies würde jedoch die Migration von Clients bedeuten und würde den Inhalt dieser Arbeit übersteigen. Diese soll sich zunächst auf das initiale Zuordnen konzentrieren.

Vorteile

- Bessere Skalierbarkeit
- Erweitert das DNS-basierte Verfahren

Nachteile

- Keine zentrale Kontrolle
- Übersteigt den Inhalt der Arbeit

Fazit Eine serverbasierte Lösung stellt ein interessantes Konzept dar, welches ergänzend zu einer DNS-basierten Lösung arbeiten könnte. Da es bei serverbasierten Lösungen vor allem darum geht, Clients auf andere Server zu migrieren, übersteigt dies jedoch den Inhalt dieser Arbeit.

Zusammenfassung

Ein geeignetes Verfahren zum Load Balancing zu finden, ist zentral für die Entwicklung eines solchen Systems. Dispatcher- sowie clientbasierte Ansätze scheinen nicht die Anforderungen an das Timadorus-System zu erfüllen, da die aktuelle Architektur sowie das aktuelle Sicherheitskonzept nicht darauf ausgelegt sind. Hinzu kommt, dass die Komplexität der Zuweisung eine clientbasierte Lösung sehr aufwendig macht. Für dispatcherbasierte Methoden kommt hinzu, dass die Komplexität der benötigten Zuweisungsalgorithmen, diesen zu einem Flaschenhals machen würde. Daraus folgend wird eine direkte Verbindung von Spielclient und Spielserver benötigt. Dies erfüllen DNS-basierte, sowie serverbasierte Methoden. Bei einem DNS-basiertem Verfahren könnte ein zentraler Server umgesetzt werden, über dem der Authentifizierungsserver einen geeigneten Spielserver ermitteln lässt. Da keine klassische Namensauflösung benötigt wird, müsste dies zu einer auf Spielerdaten basierenden Serversuche abgewandelt werden, was mit nur geringem Aufwand möglich ist. Ein serverbasierter Ansatz könnte dieses Verfahren ergänzen, indem es bei hoher Last einzelner Server bzw. bei schlechter Verteilung der Spieler eine Migration dieser auf andere Spielserver provoziert. Da dies jedoch den Inhalt dieser Arbeit übersteigt, wäre dies ein Ansatz, der in zukünftigen Arbeiten verfolgt werden könnte.

5. Die Verfahren

Dieses Kapitel behandelt die in dieser Arbeit entwickelten Verfahren. Im ersten Abschnitt wird auf die Ziele der Algorithmen eingegangen und eine allgemeine Kostenfunktion daraus ermittelt. Der zweite sowie der dritte Abschnitt beschreiben die Verfahren. Da wäre zum einen ein graphbasierter Ansatz, der versucht, die Kantensummen zwischen Spielservern, in einem imaginären Graphen, zu maximieren und zum anderen ein Heat Map basiertes Verfahren, welches die Spielwelt in mehrere Felder unterteilt, um die aktuelle Belegung von Spielserver und Spielwelt zu ermitteln. Bei beiden Verfahren wird die Serververwaltung sowie die Ermittlung eines Spielservers für einen Spieler betrachtet. Außerdem wird für beide Verfahren das Laufzeitverhalten ermittelt.

5.1. Ziele der Verfahren

Zum entwickeln geeigneter Verfahren muss definiert werden, welche Ziele diese genau verfolgen sollen. Die folgenden Verfahren sollen vor allem folgende Eigenschaften haben:

1. **Gruppieren der Spieler** Das Verfahren muss in der Lage sein, die Spieler regional zu gruppieren. Das bedeutet, je näher Spieler beieinander sind, desto höher muss die Wahrscheinlichkeit sein, dass diese auch auf dem selben Spielserver landen. Dies hat verschiedene Gründe. Zum einen werden durch das regionale Gruppieren Spielclients zusammengelegt, die mit hoher Wahrscheinlichkeit ein überlappendes Interessengebiet haben und somit gleiche Daten benötigen, zum anderen wird dadurch die Gesamtspeicherauslastung des Systems verringert. Denn durch das Gruppieren benötigen weniger Spielserver die selben Objekte und somit müssen weniger in den Speicher von mehreren Spielservern geladen werden. Hinzu kommt, dass der Kommunikationsaufwand im Gesamtsystem verringert werden kann, da das QuP-System weniger Spielserver benachrichtigen muss, wenn Objekte verändert wurden. Zusätzlich schmälert dies auch das Risiko von Kollisionen, wenn mehrere Spielserver ein Objekt manipulieren.

2. **CPU-Auslastung** Neben dem Gruppieren der Spieler, muss auch die Auslastung der Spielserver beachtet werden. Ein Faktor der dazu betrachtet werden soll, ist die Auslastung

der CPU. Denn durch eine höhere Auslastung der CPU, kann es passieren, dass die Rechenzeit für Eingaben einzelner Spieler länger dauert und somit auch die Latenz steigt. Deshalb soll es möglich sein, dass einzelne Spieler nicht mit anderen Spielern zusammen auf einen Spielserver gelegt werden, obwohl dies aus Sicht der regionalen Gruppierung die bessere Lösung gewesen wäre.

3. **Speicherauslastung** Ein weiterer Faktor, der die Serverauslastung betrifft, ist die Auslastung des Speichers. Auch dieser soll betrachtet werden. Auch wenn nicht zwingend eine gleichmäßige Auslastung über die Server angestrebt werden soll, so soll zumindest drauf geachtet werden, dass der Speicher eines Spielserver nicht überfüllt wird.

4. **Zuweisungsgeschwindigkeit** Da sich potentiell sehr viele Spieler in der Spielwelt aufhalten können, ist es nötig, dass auch bei einer hohen Spielerzahl das Auffinden eines Spielservers in einer akzeptablen Zeit möglich ist. In diesem Fall heißt hohe Spielerzahl 100.000 Spieler.

Da es sich bei diese Art von Problemen um ein NP-vollständiges [CWD+05], bei dem Gruppierungsproblem sogar um ein NP-schweres Problem [Fal94] handelt, ist das Ziel der Algorithmen nicht das Auffinden einer optimalen Lösung, sondern in erster Linie eine aus unserer Sicht akzeptable Lösung.

Damit die ersten drei genannten Eigenschaften eingehalten werden können ist es notwendig, dass auf dynamische Zuweisungsalgorithmen gesetzt wird, da diese im Gegensatz zu statischen Zuweisungsalgorithmen auf Zustandsinformationen zurückgreifen [SKS92].

5.1.1. Kostenfunktion

Aus den genannten Eigenschaften kann nun eine allgemeine Kostenfunktion abgeleitet werden:

$$kosten(Gruppierungsfaktor) = g0 * -Gruppierungsfaktor + g1 * CpuAuslastung - g2 * Restspeicher \quad (5.1)$$

Diese beinhaltet drei unserer genannten Eigenschaften: Gruppierung, CPU-Auslastung sowie Speicherauslastung. Die Speicherauslastung wird in diesem Fall jedoch nicht als prozentualer Wert betrachtet, sondern wie viel Speicher auf dem System noch vorhanden ist. Dadurch kann provoziert werden, dass Server mit einem sehr großen Speicher auch eher Zuweisungen bekommen, da dort noch am meisten Platz vorhanden ist. Bei $g0$, $g1$ und $g2$ handelt es sich

um Gewichtungsfaktoren, diese sollen es ermöglichen, dass die Algorithmen den Bedürfnissen des Spiels angepasst werden können. Der Grupierungsfaktor ist als variable gelistet, da dieser als einziger abhängig von den Zuweisungsalgorithmen ist. Daraus folgt, das die Aufgabe der Algorithmen darin besteht, einen geeignete Bewertung der Gruppierung der einzelnen Spieler zu finden.

5.2. Graphbasiertes Verfahren

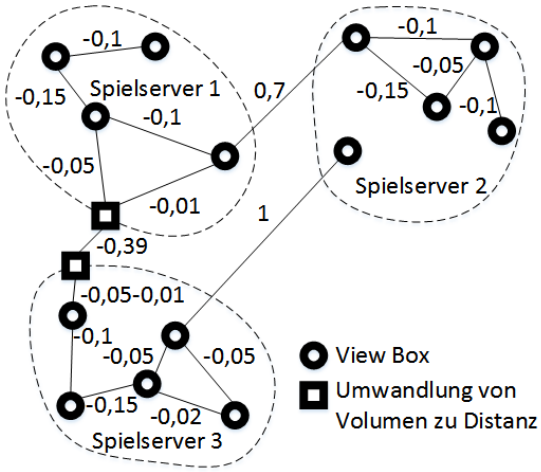


Abbildung 5.1.: Graph mit View Boxen der Spieler

Der graphbasierte Ansatz betrachtet die View Boxen der einzelnen Spieler, um eine möglichst starke Überschneidung der View Box des neuen Spielers mit den View Boxen der Spieler eines Spielservers zu finden. Der Server auf dem die View Box des neuen Spielers die stärksten Überschneidungen, mit den View Boxen der auf ihm befindlichen Spieler, aufweist hat den besten Gruppierungsfaktor. Dazu versucht das Verfahren, auf einem imaginärem Graphen, die Gewichte der Kanten zwischen den Spielservern zu maximieren.

Als Grundlage dieses Verfahrens dient das Layering Partioning aus [LC02]. Auch hier wird versucht Spieler mit überschneidenden Interessengebieten über Kanten zu verbinden, um Gruppen von Spielern zu ermitteln, die nah beieinander liegen und somit einen starken Kommunikationsaufwand untereinander haben. Wie in Abschnitt 5.1 dargestellt soll auch hier die Kommunikation reduziert werden. Zusätzlich soll das Verfahren den genutzten Speicher zu verringern. Das entwickelte Verfahren ist jedoch stark auf seinen Anwendungsfall angepasst und optimiert. So wird zu keiner Zeit der Graph tatsächlich aufgebaut. Vielmehr dient der

Graph als Orientierung dessen, was der Algorithmus erreichen soll. Dies ist möglich, da bei Anmeldung eines neuen Spielers nur ein einzelner Knoten hinzugefügt werden muss und somit zu keinem Zeitpunkt der vollständige Graph betrachtet werden muss. Zudem kann dadurch Rechenzeit gespart werden, da der Aufbau eines Graphen relativ teuer ist. Abschnitt 5.2.1 wird die genauen Gründe im Detail darstellen. Dennoch wird die Bezeichnung graphbasiert verwendet, da dieser die Grundlage des Verfahrens bildet. Bevor nun das Verfahren selber beschrieben wird, soll zunächst auf die Idee hinter dem Graphen eingegangen werden.

Der Grundgedanke des Verfahrens ist, dass aus den vorliegenden View Boxen ein Graph erzeugt werden kann mit folgenden Eigenschaften:

1. Jeder Knoten steht für eine View Box im System.
2. Knoten werden durch eine Kante verbunden, wenn deren View Boxen sich überschneiden.
3. Kanten werden mit dem Abstand der View Boxen, die sie verbinden, als Wert annotiert. Wobei Überschneidungen einen negativen Wert haben. Dieser entspricht dann dem Volumen des Quaders, das durch die Überlappenden View Boxen entsteht.
4. Werden die Knoten nach dem Spielserver gruppiert auf denen diese sich befinden. So gibt es zwischen jeder Servergruppe genau eine Kante.

Betrachtet man die ersten beiden Eigenschaften so ist schnell ersichtlich, dass sich daraus noch nicht zwingend die vierte Eigenschaft ergibt. Außerdem kann es zwischen Spielservern mehrere Überschneidungen geben. Deshalb werden zusätzlich folgende Schritte unternommen:

1. Sollte die Gruppe eines Servers nicht mit der Gruppe eines anderen Servers verbunden sein, so wird der Knoten genommen, der sich am nächsten zu dieser befindet und mit dem Knoten der anderen Servergruppe verbunden, welcher sich am nächsten zur eigenen Servergruppe befindet. Sollten sich mehrere Knoten gleich nah aufhalten, so kann ein beliebiger gewählt werden. Dies wird so lange gemacht, bis zwischen allen Servergruppen Kanten vorliegen.
2. Überschneidungen zwischen zwei Servergruppen, werden zu einer Kante zusammengefasst, bei der die Volumina, mithilfe der Gleichung 5.3, zu einer Distanz umgeformt wurden.

Ein so entstandener Graph kann in Abb. 5.1 gesehen werden. Wenn nun e_i das Gewicht einer Kante zwischen zwei Servergruppen darstellt und P die Anzahl der Kanten zwischen den

Servergruppen ist, so soll der Abstand der Kanten zwischen den Spielservern maximiert werden: $Maximize \sum_{i=0}^{P} e_i$. Da jedoch auch die Auslastung der Spielserver berücksichtigt werden muss, kann es sein, dass unter bestimmten Bedingungen die Anforderung nicht erfüllt wird und zugunsten der Auslastung ein anderer Spielserver gewählt wird.

5.2.1. Warum muss der Graph nicht aufgebaut werden?

In diesem Abschnitt soll nochmal näher darauf eingegangen werden, warum der Graph nicht aufgebaut werden muss. Wie bereits erwähnt, basiert das Verfahren auf der Idee, dass ein Graph, mit den View Boxen als Knoten aufgebaut wird, bei dem die Kanten den Abstand zwischen den View Boxen repräsentieren. Nachdem die View Boxen nach den Servern gruppiert wurden auf denen sie sich befinden, wird versucht die Kantensumme zwischen den Spielservern zu maximieren. Dennoch wird durchgehend kein Graph aufgebaut.

Dies ist möglich, da bei jedem neuen Avatar nur ein einzelner Knoten hinzugefügt wird. Dadurch ist nicht der Gesamtzustand des Graphen entscheidend, sondern wie sich dieser verändert. Dies wird vom Verfahren betrachtet. Dazu wird für jeden Server angeschaut, wie groß der Abstand des Avatars zu den restlichen Servern wäre, wenn dieser dem Server zugeordnet wäre. Dadurch kann für jede Zuordnung der neue potentielle Abstandswert ermittelt werden, der bestehen würde, wenn der Avatarknoten der nächste zu den anderen Spielservern wäre oder Überschneidungen mit diesen hat. Dadurch wird versucht für den Avatar den Spielserver zu finden, von dem aus der Abstand zu den restlichen Spielservern am größten ist. Ist der Avatar nun nicht der nächste Knoten zu den anderen Spielservern oder hat keine Überschneidungen mit den anderen, so hat das zwar keine Auswirkung auf den Gesamtgraphen, jedoch wird so vermieden, dass der Abstandswert eines Servers zu einem anderen Server kleiner wird. Sollte der neue Avatar allerdings der nächste zu einem Spielserver sein oder gar Überschneidungen mit einem anderen haben, so wurde dieser einem Spielserver zugeordnet, bei dem die Verringerung der Abstände zwischen den Server minimal ist.

Der Aufbau des gesamten Graphen ist daher bei der Zuordnung eines einzelnen neuen Avatars nicht zwingend nötig. Zudem spart dies Rechenzeit, da der Aufbau eines solchen Graphen Rechenaufwändig ist.

5.2.2. Serververwaltung

Bevor eine Berechnung stattfinden kann, muss zunächst die Zuordnung der View Boxen zu den Servern festgehalten werden. Dies geschieht in einer zentralen Instanz. Hier werden neben der Zuordnung auch die minimalen und maximalen Ausmaße eines jeden Servers auf jeder Achse festgehalten. Diese Daten werden benötigt um eine Kostenschätzung für bisher noch nicht belegte Spielserver machen zu können. Da jederzeit View Boxen auch von den Spielservern entfernt werden können und es sich dabei auch um die View Box handeln kann, die gerade einen maximalen oder minimalen Wert stellt, müssen die Minimal- und Maximalwerte einer jeden View Box festgehalten werden können.

In diesem Verfahren kann dazu eine beliebige sortierte Datenstruktur für jede Achse verwendet werden. Dadurch, dass die Werte in dieser sortiert sind, kann direkt auf den kleinsten sowie den größten Wert einer jeden Achse zugegriffen werden. Wird nun eine neue View Box auf einem Server registriert, so wird der Mininmal- und der Maximalwert einer jeden Achse in die, für die Achse zuständige, Struktur eingetragen. Eine mögliche Datenstruktur die dafür geeignet wäre, wäre z.B. ein AA-Baum, da dieser eine logarithmische Laufzeit in Aktualisierung und Suche aufweist. Für jeden Server werden somit vier Attribute festgehalten: Die vorhandenen View Boxen, die Minimal- und Maximalwerte der X-Achse, die Minimal- und Maximalwerte der Y-Achse sowie die Minimal- und Maximalwerte der Z-Achse.

Kommt nun z.B. eine View Box hinzu, die auf der X-Achse von 0,2 bis 0,3, auf der Y-Achse von 0,7 bis 0,8 und auf der Z-Achse von 0,1 bis 0,2 reicht, so werden in der Struktur für die X-Achse die Werte 0,2 und 0,3 eingetragen, in der Struktur für die Y-Achse die Werte 0,7 und 0,8 und in der Struktur für die Z-Achse die Werte 0,1 und 0,2.

Veränderungsrate

Um zu ermitteln wie stark sich das abgedeckte Gebiet durch einen Server verändert hat, wird die Veränderungsrate berechnet. Um die Berechnung möglichst simpel zu halten, wird ein Schätzwert ermittelt. Dies geschieht über die Betrachtung der Minimal- und Maximalwerte pro Achse. Gibt es einen neuen Minimal- oder Maximalwert, so wird betrachtet, wie stark sich die Box verändert hat, die mit den Minimal- und Maximalwerten gebildet werden kann. Dazu

wird das Volumen der alten mit der neuen Box verglichen und die prozentuale Veränderung berechnet:

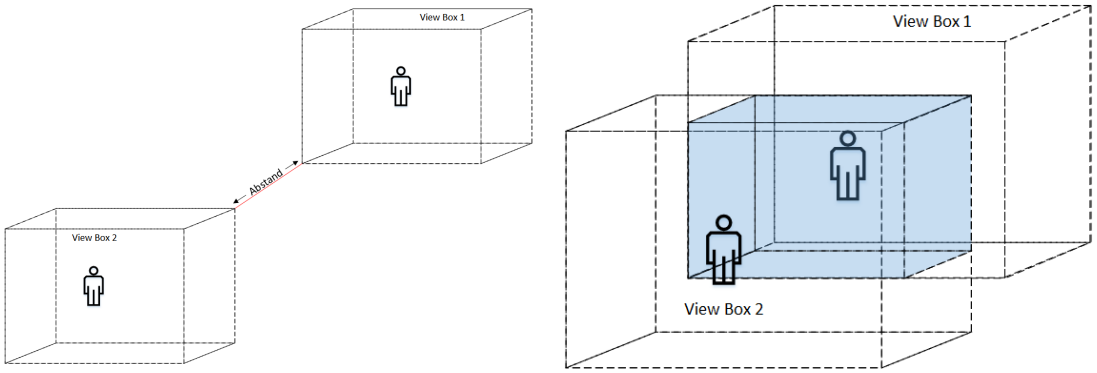
$$\begin{aligned} Volumen &= (maxX - minX) * (maxY - minY) * (maxZ - minZ) \\ Aenderungsrate &= \left| \frac{NeuesVolumen}{AltesVolumen} - 1 \right| \end{aligned} \tag{5.2}$$

5.2.3. Serverberechnung

Das Auffinden eines passenden Spielservers findet über ein Brute Force Verfahren statt. Obwohl Brute Force Techniken allgemein nicht sehr effizient sind, stellt es in diesem Fall eine passende Lösung dar. Da der folgende Algorithmus bei n Spielern im schlimmsten Fall ein Laufzeitverhalten von $O(n)$ aufweist, wobei die Laufzeit, wenn m die Anzahl der Spielserver ist, niemals $n + 2 * m$ übersteigen kann. Eine genauere Betrachtung der Laufzeit erfolgt in Abschnitt 5.2.4. Zum anderen ist es so sehr einfach möglich, den aus Sicht des Algorithmus bestmöglichen Server aufzufinden.

Zur Berechnung werden zunächst die Abstände der einzelnen Spielserver zum neuen Spieler berechnet. Diese werden anschließend als Gruppierungsfaktor zum berechnen der Kosten genutzt.

Ermitteln der Abstände



(a) (b)

Abbildung 5.2.: View Boxen mit Abstand (a). Sollten sich View Boxen überschneiden so wird das Volumen genommen (blauer Bereich) (b).

Zunächst wird der Abstand der View Box des neuen Spielers zu den Spielservern berechnet. Der Abstand zwischen zwei View Boxen ist über die Länge des kürzest möglichen Vektors, der zwischen ihnen gebildet werden kann, definiert. Sollten sich die beiden View Boxen überlappen, so wird der Quader betrachtet, der durch das Überschneiden der beiden View Boxen entsteht. Der Abstand entspricht dann dem negativen Volumen dieses Quaders.

Um den Abstand zu einem Spielserver zu ermitteln, wird über die ihm zugewiesenen View Boxen iteriert und der Abstand zu allen View Boxen aufsummiert, mit denen die View Box des neuen Spielers Überschneidungen aufweist. Um den Serverabstand nun zu ermitteln, wird aus dem so ermittelten Gesamtvolumen, mit dem es Überschneidungen, gibt die Diagonale des Würfels berechnet, der aus dem Volumen gebildet werden kann:

$$Serverabstand = \sqrt{3} * \sqrt[3]{Gesamtvolumen} \tag{5.3}$$

Daraus ergibt sich ein negativer Abstand, da für das Gesamtvolumen negative Volumina aufsummiert wurden. Auf diese Weise kann der gemeinsam abgedeckte Bereich berücksichtigt werden und dennoch bleiben die Werte vergleichbar mit den ermittelten Abständen, wenn es keine Überschneidungen gab, weil es sich so bei allen Werten um Distanzen handelt. Sollte es nämlich keine Überschneidung geben, so ist der geringste ermittelte Abstand, mit einer View Box des Servers, der Serverabstand. Die so berechneten Serverabstände werden für jeden einzelnen Spielserver festgehalten. Zudem wird die Summe sämtlicher Serverabstände aufsummiert. Dies wird für die spätere Kostenberechnung benötigt. Neben den Abständen wird zusätzlich die durchschnittliche Achsenlänge, der durch die Spielserver abgedeckten Bereiche, festgehalten. Dazu wird der Durchschnittswert aus Höhe, Breite und Tiefe für jeden Spielserver ermittelt. Höhe, Breite und Tiefe werden dabei mithilfe der festgehaltenen Minimal- und Maximalwerte einer jeden Achse berechnet. Dies dient als Gradmesser für die Größe, der durch die Server abgedeckten Bereiche. Die so ermittelten Werte werden aufsummiert und dienen später als Referenzwert, zur Ermittlung eines Standardabstands für Spielserver, auf denen sich noch keine Spieler befinden. Dadurch werden zwar keine genauen Werte erhalten, allerdings ist diese Heuristik gut genug für unseren Anwendungsfall, wie die abschließenden Experimente zeigen (Kapitel 7).

Sollte sich noch kein Spieler auf einem Server befinden, so wird dieser in einer Liste für noch nicht belegte Spielserver festgehalten und in einem späteren Schritt ein Referenzwert auf Basis, der bei den anderen Spielserver ermittelten Werte, ermittelt. Dies wird im folgenden

Abschnitt vorgestellt. Ein Pseudocodebeispiel, zur Ermittlung der Abstände, kann in Anhang A Algorithmus 1 gesehen werden.

Sonderfall: Noch kein Spieler auf dem Server

Nachdem der Abstand zu den bereits mit Spielern belegten Spielservern berechnet wurde, wird ein Standardabstand für bisher noch nicht belegte Spielserver ermittelt. In diesem Fall gibt es noch keine registrierten View Boxen und somit nichts womit die Entfernung zur View Box des Avatars berechnet werden könnte. Deshalb wird ein Standardwert, unter Zuhilfenahme der bereits ermittelten Werte, berechnet. Dieser Wert soll den Abstand darstellen, ab dem es Sinn macht einen Spieler einem noch nicht belegten Server zuzuweisen, weil er sich in einer gewissen Distanz zu den anderen Spielern aufhält.

Der Abstand wird wie folgt ermittelt:

$$
\begin{aligned}
DurchschnittsGroesse &= \frac{SummeDurchschnittlicheServerGroesse}{AnzBelegteSpielserver} \\
OptimaleDurchschnittsServergroesse &= \frac{1}{\sqrt[3]{anzahlSpielserver}} \\
AbstandLeereServer &= \frac{OptimaleDurchschnittsServergroesse - DurchschnittsGroesse}{2}
\end{aligned}
\tag{5.4}
$$

$SummeDurchschnittlicheServerGroesse$ stellt dabei die Summe, der während der Abstandsberechnung ermittelten durchschnittliche Achsenlänge pro Server, dar. $AnzBelegteSpielserver$ ist die Anzahl der Spielserver, auf denen sich Spieler befinden.

Mittels der Berechnung soll ein Abstand gefunden werden bei dem sich zwei Spieler mit einer großen Wahrscheinlichkeit auf einem anderen Spielserver befinden. Um dies möglichst einfach zu halten, wird von einer gleichmäßigen Verteilung, der Spielserver auf die Spielwelt, ausgegangen. Dazu wird berechnet, wie lang der abgedeckte Bereich eines Spielservers pro Achse wäre, wenn die ganze Spielwelt auf alle Spielserver verteilt wäre und alle einen gleichgroßen Bereich verwalten würden, der sich nicht mit dem Bereich eines anderen Spielservers überschneidet ($OptimaleDurchschnittsServergroesse$). Als nächstes wird von der optimalen Länge, die ermittelte durchschnittliche Länge der belegten Server pro Achse ($DurchschnittsGroesse$) abgezogen. Der so ermittelte Wert stellt den Abstand der Spielserver dar, wenn jeder den ermittelten durchschnittlichen Bereich verwalten und sich diese genau in der Mitte der optimalen Bereichs befinden würde. Da es jedoch wünschenswert wäre, dass der Spielserver ausgewählt

wird zu dem der optimale Bereich gehört, muss dieser Wert noch halbiert werden. So erhält man den Abstand vom Rand des durchschnittlichen Bereichs zum Rand des optimalen Bereichs. Dieser soll als Standardabstand für die leeren Server dienen. Sollte sich auf noch keinem Server ein Spieler befinden, so wird Null als Standardwert genommen. Ein Pseudocodebeispiel, zur Berechnung der Abstände für Leere Server, kann in Anhang A Algorithmus 1 gesehen werden.

Berechnung der Kosten

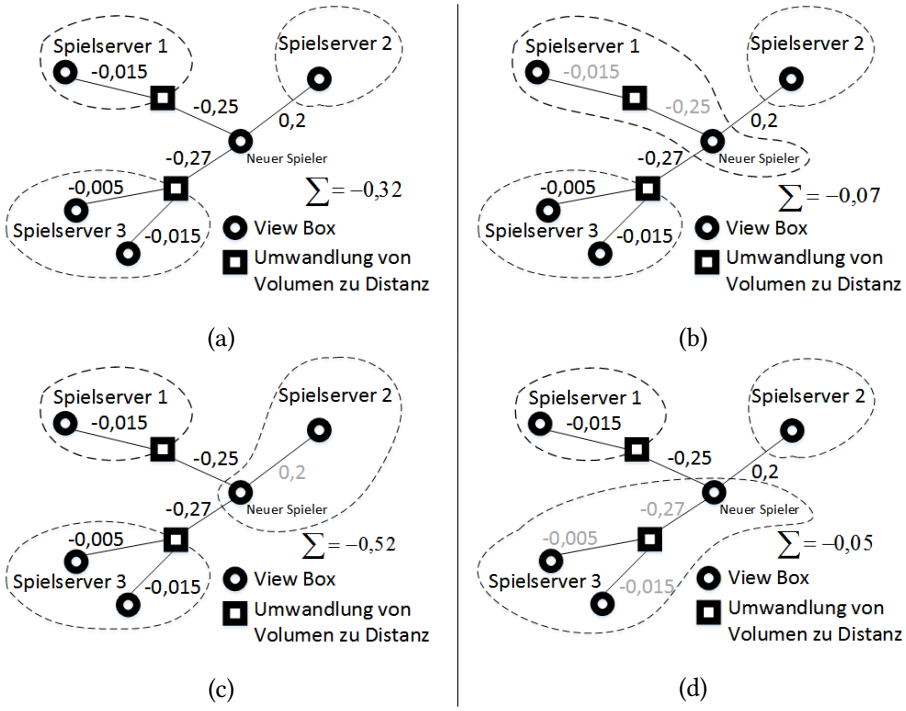


Abbildung 5.3.: Knoten für den neuen Spieler mit Kanten zu den sich am nächsten befindenden View Boxen der Spielserver. (a): Ohne Zuordnung mit Summe der Entfernung zu allen Spielservern; (b): Spielserver 1 zugeordnet mit Summe der Entfernung zu den anderen Spielservern; (c): Spielserver 2 zugeordnet mit Summe der Entfernung zu den anderen Spielservern; (d): Spielserver 3 zugeordnet mit Summe der Entfernung zu den anderen Spielservern.

Stellt man sich den zu Beginn des Abschnitts 5.2 beschriebenen Graphen vor, so wird nun nach dem Spielserver gesucht, mithilfe dessen die Summe der Kanten, zwischen den Spielservern maximiert werden. Dazu wird betrachtet, um wie viel sich die Summe der Kantenverbindungen zwischen den Spielservern maximal erhöht, wenn der Spieler einem bestimmten Spielserver zugewiesen wird. Dazu wird über alle Spielserver iteriert und von der Gesamtsum-

me aller Entfernungen die Entfernung zu dem jeweiligen Spielserver abgezogen. So erhält man den Abstand der View Box des neuen Spielers zu den restlichen Spielservern, wenn dieser dem Spielserver zugewiesen ist. Abb. 5.3 soll diesen Vorgang verbildlichen. Der so ermittelte Wert stellt den Gruppierungsfaktor dar. Mithilfe der in Abschnitt 5.1.1 ermittelten Kostenfunktion, werden nun die Kosten für den Spielserver berechnet. Aus den Spielservern wird dann der mit den geringsten Kosten als am besten geeigneter Server ausgewählt.

5.2.4. Laufzeit

Zum Abschluss soll die Laufzeit der Serverberechnung betrachtet werden. Im Folgenden wird die Anzahl der Spieler mit n, die Anzahl der Spielserver mit m sowie die Anzahl der nicht mit Spielern belegten Spielserver mit m_{leer} beschrieben. n_i beschreibt die Anzahl der Spieler auf Spielserver i.

Ermitteln der Abstände

Betrachten wir zunächst das Ermitteln der Abstände. Dieses kann in zwei Schritte untergliedert werden. Zum einen ins Ermitteln der Abstände, der bereits mit Spielern belegten Spielservern und zum anderen in das Ermitteln, der Abstände der noch nicht belegten Spielserver. Zunächst wird über jeden Spielserver iteriert, sowie innerhalb der Iteration über jede View Box, die von dem Spielserver verwaltet wird. Daraus ergeben sich $\sum_{i=0}^{m} n_i$ Durchläufe. Da jeder Spieler nur einem Spielserver zugeordnet sein kann und somit über jeden Spieler iteriert wird, kann auch von n Durchläufen gesprochen werden. Da einige Spielserver auch ohne Belegung sein können und diese dann in eine Liste aufgenommen werden, kann dies zu $LaufzeitErw = \sum_{i=0}^{m} \begin{cases} n_i & \text{wenn } n_i \neq 0 \\ 1 & \text{sonst} \end{cases}$ erweitert werden. Daraus ergeben sich $n + m_{leer}$ Durchläufe. Nachdem die Abstände der bereits durch Spieler belegten Spielserver berechnet wurden, wird, nachdem der Abstand für die leeren Server berechnet wurde, über diese iteriert, um ihnen den ermittelten Wert zuzuordnen. Dadurch hat die gesamte Berechnung der Abstände nun eine Laufzeit von $n + 2 * m_{leer}$. Da sich die Spieler auf mindestens einem Spielserver befinden müssen, ist im schlimmsten Fall $m_{leer} = m - 1$ wenn $n > 0$. Daraus resultieren $n + 2 * (m - 1)$ Durchläufe. Da m jedoch bei großen n eine vernachlässigbar kleine Zahl darstellt, kann von $O(n)$ gesprochen werden.

Kostenberechnung

Als nächstes wird die Kostenberechnung betrachtet. Die Berechnung der Kosten besteht nur aus einer Iteration innerhalb derer aus den vorher ermittelten Werten die Kosten berechnet werden. Da die Schleife über alle Spielserver iteriert, gibt es in jedem Fall m Durchläufe. Eine Abhängigkeit zu n gibt es hier nicht mehr. Somit kann von $O(m)$ gesprochen werden.

Gesamtlaufzeit

Die Berechnung der Abstände weist ein Laufzeitverhalten von $O(n)$ auf. Die Berechnung der Kosten sogar nur von $O(m)$. Da die beiden Berechnungen sequentiell ausgeführt werden, entsteht eine Laufzeit von $n + m$. m ist jedoch bei großen n sehr klein und kann daher vernachlässigt werden. Daraus ergibt sich eine Laufzeit von $O(n)$. Diese steigt somit linear in Abhängigkeit zur Spielerzahl.

5.3. Einteilung über Heat Map Matching

Beim Heat Map Matching wird die Spielwelt in mehrere gleichgroße Felder unterteilt. Sollte ein Spielserver einen bestimmten Bereich verwalten, so wird dies in die jeweiligen Felder eingetragen. Daraus ergibt sich für jeden Spielserver eine Karte auf der zu sehen ist, wie stark welches Feld belegt ist. Je stärker ein Feld belegt ist umso mehr kann von „Hitze“ gesprochen werden. Abb. 5.4b bis Abb. 5.4d sollen dies in zweidimensionaler Form illustrieren.

Der Grundgedanke ist mit dem der graphbasierten Methode vergleichbar. Es wird nach einer möglichst großen Überschneidung mit der View Box des neuen Spielers gesucht und sollte dies nicht erfolgreich sein, nach dem Spielserver dessen View Boxen, sich möglichst nah an der View Box des neuen Spielers befinden. Das Vorgehen dieses zu ermitteln ist jedoch ein ganz anderes.

Zum Verwalten der Spielerpositionen wird ein dreidimensionales Raster aufgebaut, welches die Spielwelt repräsentiert. Die einzelnen Felder speichern nur die Anzahl der in ihnen befindlichen Spieler. Der Vorteil ist eine konstante Zugriffszeit auf die einzelnen Felder, unabhängig von der Anzahl der Spieler. Jedoch kann die Präzision bei einer zu geringen Anzahl an Feldern leiden, auf der anderen Seite nimmt allerdings auch die Effizienz mit einer steigenden Anzahl an Feldern ab, da zum Auswerten über mehr Felder iteriert werden muss.

[ ]	[{1,2}]	[{1,1}]	[{1,2}]	[ ]	[ ]	[ ]	[ ]
[ ]	[{1,3}]	[{1,4}]	[{1,3}]	[{1,2}]	[{2,1}]	[ ]	[ ]
[{1,3}]	[{1,4}]	[{1,5}]	[{1,3}]	[{1,1}, {2,1}]	[{2,2}]	[{2,2}]	[{2,1}]
[{1,2}, {3,2}]	[{1,3}]	[{1,3}]	[{1,2}, {2,1}]	[{2,3}]	[{2,4}]	[{2,4}]	[{2,3}]
[{3,3}]	[{1,2}, {3,2}]	[{2,2} {3,1}]	[{2,3}, {3,1}]	[{2,5}]	[{2,6}]	[{2,5}]	[{2,2}]
[{3,4}]	[{3,4}]	[{3,3}]	[{2,2}, {3,2}]	[{2,4}]	[{2,4}]	[{2,3}]	[{2,2}]
[{3,5}]	[{3,6}]	[{3,4}]	[{3,2}]	[{2,3}]	[{2,2}]	[{2,2}]	[{2,2}]
[{3,3}]	[{3,5}]	[{3,3}]	[{3,2}]	[{3,1}]	[ ]	[ ]	[{2,1}]

(a) Zusammengefasste Heat Map

0	2	1	2	0	0	0	0
0	3	4	3	2	0	0	0
3	4	5	3	1	0	0	0
2	3	3	2	0	0	0	0
0	2	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

(b) Spielserver 1

0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0
0	0	0	0	1	2	2	1
0	0	0	1	3	4	4	3
0	0	2	3	5	6	5	2
0	0	0	2	4	4	3	2
0	0	0	0	3	2	2	2
0	0	0	0	0	0	0	1

(c) Spielserver 2

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0
3	2	1	1	0	0	0	0
4	4	3	2	0	0	0	0
5	6	4	2	0	0	0	0
3	5	3	2	1	0	0	0

(d) Spielserver 3

Abbildung 5.4.: Heat Maps der Spielwelt. Zur vereinfachten Darstellung zweidimensional. (a) stellt die zusammengefasste Heat Map dar. Die Tabelle wird in Eckigen Klammern dargestellt, sowie die enthaltenen Wertpaare in geschweiften ($Tabelle = [\{Spielserver, AnzSpieler\}]$). (b) - (c) stellen die umgerechneten Heat Maps der einzelnen Spielserver dar.

5.3.1. Serververwaltung

Die Serververwaltung findet über eine zentrale Komponente statt. Treffen Nachrichten mit Zustandsveränderungen der Spielserver ein so wird auch die Heat Map entsprechend angepasst. Damit Speicherplatz gespart werden kann, werden sämtliche Daten innerhalb einer einzelnen Matrix gespeichert. Zum besseren Verständnis soll zunächst die Heat Map im Detail erklärt werden.

Die Heat Map

Die Heat Map dient zur Darstellung der Spielwelt. Dazu wird sie in mehrere gleichgroße Felder eingeteilt. Da die Einteilung der Spielwelt in alle drei Dimensionen geschehen muss handelt es sich bei diesen um Würfel. In jedem Feld wird abgelegt, wie stark dieser Bereich von einzelnen Spielservern belegt ist. Damit nicht für jeden Server eine eigene Heat Map verwaltet werden muss, existiert für jedes Feld eine Tabelle in der aufgeführt ist, mit wie vielen View Boxen die einzelnen Spielserver in diesem Bereich vertreten sind. Sollte ein Server gar nicht vorhanden sein, so ist dieser auch nicht Bestandteil der Tabelle. Das bedeutet auch, dass ein Spielserver aus der Tabelle entfernt werden muss, wenn dieser, aufgrund von Veränderungen, der auf ihm befindlichen View Boxen, den Bereich nicht mehr abdeckt. Abb. 5.4a zeigt wie eine solche Heat Map in einem zweidimensionalem Raum aussehen kann. Die Größe einer Heat Map wird über die Präzision definiert. Diese beschreibt wie viele Felder in jeder Dimension existieren. Ist die Präzision zum Beispiel 10, so besteht die Heat Map aus $10 * 10 * 10 = 1000$ Feldern.

Manipulieren der Heat Map

Nachdem über die Struktur der Heat Map gesprochen wurde, soll nun die Verwaltung dargestellt werden. Dazu soll zunächst das Verfahren zum ermitteln der Position näher gebracht werden.

Wenn Änderungen ankommen, ist von der View Box die Position sowie die umhüllende

Bounding Box bekannt. Zur einfacheren Darstellung sollen diese als Vektoren dargestellt werden:

$$\begin{aligned} Position &= \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \\ BoundingBox &= \begin{pmatrix} LaengeX \\ LaengeY \\ LaengeZ \end{pmatrix} \end{aligned} \tag{5.5}$$

Nun muss zunächst der Beginn sowie das Ende der View Box berechnet werden:

$$\begin{aligned} Startpunkt &= Position - \frac{BoundingBox}{2} \\ Endpunkt &= Position + \frac{BoundingBox}{2} \end{aligned} \tag{5.6}$$

Die Bounding Box muss halbiert werden, da diese, die gesamte Höhe, Breite und Tiefe beinhalten und nur der Abstand zur Position, welche sich immer im Zentrum der Box befindet benötigt wird. Nachdem Start- sowie Endpunkt auf jeder Achse ermittelt wurden, können nun die Indizes der Felder berechnet werden, in denen sich diese Punkte befinden:

$$\begin{aligned} Schrittgroesse &= \frac{1}{Praezision} \\ indizes(Punkt) &= \left\lfloor \frac{Punkt}{Schrittgroesse} \right\rfloor \end{aligned} \tag{5.7}$$

Die Schrittgröße steht für die Länge eines Feldes der Heat Map und wird benötigt um die Position zu ermitteln. Nachdem $Startindizes = indizes(Startpunkt)$ und $Endindizes = indizes(Endpunkt)$ ermittelt wurden, können die Werte in den Feldern angepasst werden. Dazu wird über sämtliche Felder iteriert, die sich in dem Quader befinden, dass durch die Diagonale von $Startindizes$ nach $Endindizes$ aufgespannt wird. Da bei diesem Verfahren auch Felder als belegt eingetragen werden, welche von der View Box des Spielers nur teilweise belegt werden, kann in diesem Fall von einer überschätzenden Heuristik gesprochen werden. Eine Pseudocodebeispiel kann in Anhang B Algorithmus 4 gesehen werden.

Neben der Stärke der Belegung einzelner Felder, werden auch die Anzahl der belegten Felder sowie eine Zuordnung der Server zu den genutzten Feldern festgehalten. Die Anzahl der belegten Felder wird benötigt um später einen Standardwert für leere Spielserver ermitteln zu können. Außerdem wird über diesen Wert die Veränderungsrate bestimmt. Die Zuordnung der

Spielserver zu den belegten Feldern wird benötigt, falls ein Spielserver nicht mehr erreichbar ist und dieser deshalb entfernt werden muss. So kann über dessen Felder iteriert werden und diese aus der Heat Map ausgetragen werden.

Veränderungsrate

Beim Heat Map Matching wird die Veränderungsrate über die Anzahl der belegten Felder ermittelt. Dazu wird die neue Anzahl mit der vor der Veränderung verglichen und die prozentuale Änderung berechnet:

$$Aenderungsrate = \left| \frac{AnzFelderVorher}{AnzFelderNachher} - 1 \right| \tag{5.8}$$

5.3.2. Serverberechnung

Zur Berechnung eines Servers muss die Heat Map ausgewertet werden. Dies geschieht in drei Schritten. Zunächst wird nach Überschneidungen mit dem Bereich der View Box geschaut. Sollte es noch nicht belegte Spielserver geben, so werden als nächstes für diese die Kosten berechnet. Gibt es anschließend noch Spielserver, die nicht betrachtet wurden, so wird abschließend versucht auch für diese die Kosten zu ermitteln.

Überschneidungen mit der View Box

Zunächst wird nach Überschneidungen, der View Box des neuen Spielers, mit den View Boxen auf den Spielservern geschaut. Dazu wird über die, für den neuen Spieler ermittelten, Felder iteriert und die stärke der Überschneidung mit anderen Spielservern ermittelt. Dafür werden zunächst die Start- sowie Endindizes der View Box des neuen Spielers berechnet und über alle Felder, die von der View Box abgedeckt werden, iteriert. Für jeden Server gibt es einen Akkumulator, der beschreibt, wie stark die Überschneidung ist. Dieser wird mit Null initialisiert. Wird ein Feld von einem oder mehreren Servern bereits genutzt, so wird dieser für den entsprechenden Server um die Anzahl an View Boxen erhöht, die auf dem Server für das Feld registriert sind. Damit dieser Wert unabhängig von der eingestellten Präzision ist, wird die Anzahl der View Boxen mit der Schrittgröße multipliziert:

$$Accu \leftarrow Accu + AnzViewBoxen * Schrittgroesse \tag{5.9}$$

Der so ermittelte Wert stellt den Gruppierungsfaktor dar. Je größer dieser ist desto stärker ist die Überschneidung mit der View Box des neuen Spielers. Mithilfe der Kostenfunktion

aus Abschnitt 5.1.1 können die Kosten der Spielserver berechnet werden. Der Server mit den geringsten Kosten ist der aktuell am besten geeignete Server.

Leere Spielserver

Sollten noch nicht belegte Spielserver vorhanden sein, so werden als nächstes deren Kosten berechnet. Hierbei soll ein Abstand ermittelt werden, ab dem es sinnvoll scheint, dass ein Spieler einem anderem Spielserver zugewiesen wird. Dazu wird zunächst eine potentielle ideale Anzahl an Felder pro Server berechnet, wie sie bei gleichmäßiger Aufteilung der Spielwelt auf die Server vorläge:

$$IdealAnzFelder = g0 * \frac{Praezision^3}{AnzServer} + g1 * \frac{BelegteFelder}{AnzServer} \tag{5.10}$$

Diese Berechnung besteht aus zwei Bestandteilen. Zum einen die Aufteilung der gesamten Spielwelt auf die Spielserver und zum anderen die Aufteilung der aktuell genutzten Felder auf die Spielserver. Wobei mit genutzten Feldern nicht nur die Felder, der sich bereits auf Spielservern befindlichen View Boxen gemeint sind, sondern auch die des neuen Spielers. Die beiden Bestandteile können über $g0$ und $g1$ gewichtet werden. Da es sich um eine prozentuale Gewichtung handelt, muss für beide Werte $0 \leq g \leq 1$, wobei $g0 + g1 = 1$, gelten. Aus dieser ermittelten Idealfläche soll ein Abstandswert ermittelt werden. Dazu wird die durchschnittliche Anzahl an Feldern, der Spielserver, von der Idealfläche abgezogen. Dadurch erhält man den durchschnittlichen Abstand ab dem bei Nutzung der Idealfläche im Schnitt der nächste Spielserver liegt. Da sich die Flächen auf einen dreidimensionalen Raum beziehen, jedoch nur der Abstand auf einer Achse von Interesse ist, wird zuvor von beiden Werten die dritte Wurzel gezogen. Damit daraus der Abstand in eine Richtung auf der Achse wird, wird dieser Wert halbiert. Anschließend wird der Wert noch mit der Schrittgröße multipliziert, damit dieser unabhängig von der gewählten Präzision ist:

$$Abstandswert = \frac{\sqrt[3]{IdealAnzFelder} - \sqrt[3]{DurchschnittAnzFelder}}{2} * Schrittgroesse \tag{5.11}$$

Der so ermittelte Wert ist der Abstand ab dem es Sinn macht einen anderen Server zu belegen. Da ein Abstand jedoch schlecht für die Gruppierung von Spielern ist, ist dies der negative Gruppierungsfaktor ($Gruppierungsfaktor = -Abstandswert$). Mit der Kostenfunktion aus Abschnitt 5.1.1 können nun die Kosten für die leeren Spielserver ermittelt werden.

Restliche Spielserver

Sollten einige Spielserver noch nicht betrachtet worden sein, so haben diese trotz bereits registrierter View Boxen, keinerlei Überschneidungen mit dem neuen Spieler. Dennoch müssen auch diese betrachtet werden, da es durchaus sein kann, dass es keine Überschneidungen mit anderen Spielservern gab oder dass die Spielserver mit Überschneidungen stark ausgelastet sind. In solchen Fällen können Spielserver ohne Überschneidungen als besser geeignet betrachtet werden. Dazu soll von dem Bereich der View Box aus immer eine Reihe weiter nach außen gegangen werden und betrachtet werden, ob in dieser Reihe belegte Felder vorliegen. Der Abstand wird dabei für jede Reihe um eins erhöht. Da auch hier der Wert unabhängig von der Präzision sein soll wird auch hier der Abstand mit der Schrittgröße multipliziert. Ähnlich wie bei dem Ermitteln der Abstände der leeren Spielserver, erhält man so den negativen Gruppierungsfaktor ($Gruppierungsfaktor = -Abstand$). Bevor eine Reihe weiter außen geschaut wird, werden die Kosten für die restlichen Spielserver bei diesem Abstand berechnet, auch wenn auf diesen keine View Box in diesem Abstand registriert ist. Dadurch sollen Server vorzeitig herausgefiltert werden, deren Kosten bei dieser Distanz bereits höher sind als die aktuell geringsten Kosten. Sollte in einer Reihe eine View Box registriert sein von einem Spielserver, dessen Kosten noch nicht erfasst sind, so werden dessen Kosten mit den aktuell geringsten Kosten verglichen. Sind diese niedriger, so wird dieser Server als aktuell bester Server übernommen. Die Suche wird abgebrochen, wenn für keinen Server mehr ein besseres Ergebnis, als das aktuell Beste möglich ist. Dies ist der Fall wenn alle noch zu überprüfenden Server herausgefiltert wurden, weil sie a) keine besseres Ergebnis als das aktuell geringste Erzielen konnten oder b) deren Kosten bereits erfasst wurden. Ein Pseudocodebeispiel kann in Anhang B Algorithmus 7 gesehen werden.

Optimierung über Parallelisierung

Durch Parallelisierung kann der Vorgang optimiert werden, da so auf mehrere Felder zeitgleich zugegriffen werden kann. Dies ist sowohl bei der Suche nach Überschneidungen mit der View Box des neuen Spielers sowie bei der Suche außerhalb des Bereiches der View Box des neuen Spielers möglich. Der folgende Vorschlag basiert auf dem Devide and Conquer Prinzip. Dazu werden die zu durchsuchenden Bereiche immer weiter halbiert, bis diese eine gewisse Größe unterschreiten.

Suche nach Überschneidungen mit der View Box

Zunächst soll die Suche nach Überschneidungen betrachtet werden. Statt über jedes Feld hintereinander zu iterieren, kann der Bereich in immer feinere Teilbereiche unterteilt werden.

Dazu wird der zu untersuchende Bereich solange halbiert, bis ein definierter Grenzwert unterschritten wird. Dieser Beschreibt die Anzahl an Felder pro Achse, ab der eine entstandene Box nicht weiter unterteilt werden soll, d.h. sobald die Länge in X, Y und Z-Richtung kleiner ist wird nicht weiter halbiert. Zum Unterteilen einer bestehenden Box wird immer die längste noch bestehende Seite genommen und diese halbiert. Die daraus resultierenden Hälften können in der Folge parallel abgearbeitet werden und die Ergebnisse zusammengetragen werden, wenn beide Seiten fertig sind. Ist der Grenzwert erreicht, so wird über die noch bestehenden Felder, wie bisher, iteriert. Ein Pseudocodebeispiel kann in Anhang B Algorithmus 8 gesehen werden.

Überprüfen der restlichen Spielserver außerhalb des Bereiches der View Box

Als nächstes soll die Suche der Felder außerhalb des Bereiches der View Box des neuen Spielers betrachtet werden. Da von dem Bereich der View Box immer weiter um eine Reihe nach außen gegangen wird, bildet der zu untersuchende Bereich bei jedem Schritt den Rand einer Box, die um den zuletzt untersuchten Bereich liegt. Von dieser wird zunächst eine jede Seite in einen eigenen Prozess aufgeteilt. Jeder dieser Seiten wird nun weiter unterteilt. Wie bei der Suche nach Überschneidungen wird auch hier die längste Seite halbiert, bis Anzahl der Felder in alle Richtungen kleiner als der Grenzwert ist. Ist der Grenzwert unterschritten, so werden die Felder, wie bisher, untersucht. Eine mögliche Einteilung könnte wie in Anhang B Algorithmus 9 aussehen.

5.3.3. Laufzeit

Die Laufzeit der Serverberechnung ist nur geringfügig von der Anzahl der Spieler abhängig, da direkt auf die benötigten Felder zugegriffen werden kann. Diese ist vor allem von der Präzision p abhängig. Im schlimmsten Fall werden sämtliche Felder abgesucht, da nun die Anzahl der Felder p^3 entspricht, wären dies p^3 abfragen. Im schlimmsten Fall wäre zudem auf jedem Feld jeder Spielserver vertreten. Wenn n die Anzahl der Spielserver ist, so kann daraus $O(p^3 * n)$ gefolgert werden. Die genaue Herleitung wird im Folgenden genauer betrachtet. Die Berechnung kann in drei Schritte unterteilt werden: Der Betrachtung der Überlappungen mit dem Bereich der View Box, die Berechnung der Kosten für noch nicht belegte Spielserver sowie der Berechnung der Kosten für Spielserver, welche keine Überlappung mit dem Bereich der View Box aufweisen. Im Folgenden steht n für die Anzahl der Server sowie n_{belegt} für die Anzahl der belegten Server und n_{leer} für die Anzahl der bisher noch nicht mit Spielern belegten Server. Außerdem wird die Anzahl der Felder der Spielwelt als $m = p^3$ definiert. Wobei $m_{viewBox}$ die Anzahl der Felder im Bereich der View Box des neuen Spielers und m_{rest} die Anzahl der restlichen Felder ist.

Berechnung der Überschneidungen

Zunächst wird über sämtliche, von der View Box des neuen Spielers belegten Felder iteriert. In jedem Feld wird zudem über die Spielserver mit registrierten View Boxen innerhalb des Feldes, iteriert. Im schlimmsten Fall sind bei jedem Spielfeld sämtliche Spielserver mit View Boxen registriert. Daraus ergibt sich eine Laufzeit von $m_{viewBox} * n_{belegt}$.

Berechnen des Abstandes für leere Spielserver

Sollten noch nicht belegte Spielserver vorhanden sein, so werden im nächsten Schritt die Kosten für die noch nicht belegten Spielserver berechnet. Da nach der Berechnung des Standardwertes über sämtliche bisher noch nicht belegten Spielserver iteriert wird, ergibt sich daraus eine Laufzeit von n_{leer}.

Berechnung der Kosten für die restlichen Spielserver

Sind noch nicht alle Spielserver für die Berechnung berücksichtigt worden, so werden als nächstes die Felder außerhalb des Bereiches der View Box betrachtet. Auf den ersten Blick könnte auch hier gesagt werden, dass im schlimmsten Fall auf sämtlichen Feldern sämtliche belegte Spielserver betrachtet werden müssen. Daraus würde eine Laufzeit von $m_{rest} * n_{belegt}$ ergeben. Jedoch ist dies aufgrund der Eigenschaft, dass der Algorithmus abbricht sobald sämtliche Spielserver betrachtet wurden, falsch. Daher muss immer mindestens ein Spielserver vorhanden sein der noch nicht betrachtet wurde. Daraus folgt, dass die Laufzeit im schlimmsten Fall $m_{rest} * (n_{belegt} - 1)$ beträgt.

Gesamtlaufzeit

Wenn nun die betrachteten Laufzeiten aufaddiert werden ergibt sich eine Gesamtlaufzeit von $m_{viewBox} * n_{belegt} + n_{leer} + m_{rest} * (n_{belegt} - 1)$. Damit jedoch die Betrachtung des Bereiches außerhalb dem der View Box relevant wird darf auch beim Überprüfen von Überschneidungen im Bereich der View Box mindesten ein belegter Spielserver nicht betrachtet worden sein, da sonst dieser nicht mehr betrachtet wird. Daraus folgt eine Gesamtlaufzeit von $m_{viewBox} * (n_{belegt} - 1) + n_{leer} + m_{rest} * (n_{belegt} - 1)$. Da die Anzahl der Server in der Regel wesentlich kleiner ist als die Anzahl der Felder, ist n_{leer} für die Laufzeit irrelevant und es folgt $m_{viewBox} * (n_{belegt} - 1) + m_{rest} * (n_{belegt} - 1) = (m_{viewBox} + m_{rest}) * (n_{belegt} - 1) = m * (n_{belegt} - 1)$. Im schlimmsten Fall sind nun $n_{belegt} = n$. Daraus folgt eine Gesamtlaufzeit von $m * (n - 1)$. Ist nun $m_{viewBox} = m$, so ist die Betrachtung des Bereiches außerhalb des Bereiches der View Box irrelevant. Daher werden auch alle Felder betrachtet, wenn sämtliche

Server auf sämtlichen Feldern registriert sind. Dadurch ist es nicht nötig dass die Anzahl der Server pro Feld $n-1$ ist, sondern kann auch n sein. Daraus ergäbe sich eine Gesamtlaufzeit von $O(m*n) = O(p^3*n)$.

Da dieser Wert jedoch nur bei extrem großen View Boxen und einer sehr schlechten Verteilung auftritt, kann im Mittel von einer deutlich besseren Laufzeit ausgegangen werden. Auch das Überprüfen der Felder außerhalb des Bereiches der View Box beinhaltet Abbruchbedingungen für den Fall, das Spielserver keine besseren Kosten mehr erreichen können, sodass nicht zwingend bis in den Bereich anderer Spielserver gegangen werden muss und die Berechnung auch deutlich vor diesem beendet werden kann. Hinzu kommt, dass das Verfahren dazu neigt einzelne Regionen der Spielwelt bevorzugt einem Spielserver zuzuweisen, somit ist in den meisten Feldern nur ein einziger Spielserver eingetragen. Dadurch ist die Laufzeit im Durchschnitt wesentlich näher an p^3 als an p^3*n.

Parallelisiertes Verfahren

Durch die Parallelisierung des Verfahrens kann auch der schlechteste anzunehmende Fall deutlich verbessert werden. So wird bei der Betrachtung der Überschneidungen mit den Bereichen der View Box bei jedem Schritt die Menge halbiert und parallelisiert. Dadurch kann sequentiell ein Verhalten von $O(\log(n_{ViewBox}*m))$ erreicht werden.

Auch bei der Betrachtung der Bereiche außerhalb der View Box kann eine deutliche Verbesserung erreicht werden. Hier wird jedoch weiterhin sequentiell nach außen gegangen, sodass ein komplett logarithmisches Verhalten nicht möglich ist. Allerdings werden nach initialem sechsteln jede Seite immer weiter halbiert bis der Grenzwert unterschritten ist. Im schlimmsten Fall muss die Gesamte Spielwelt betrachtet werden. Am äußersten Rand wäre dann jede zu betrachtende Fläche p^2 groß. Sollte tatsächlich alles betrachtet werden müssen, so ist die Seitenlänge im ersten Schritt Eins und wächst mit jedem Schritt nach außen um Zwei. Wenn davon ausgegangen wird, dass der Startpunkt in der Mitte der Spielwelt liegt, ergibt sich daraus eine mittlere Fläche von $\frac{p^2}{\lceil 0{,}5*p \rceil}$. Daraus ergeben sich sequentiell $\frac{\frac{p^2}{\lceil 0{,}5*p \rceil}}{6}$ Feldabfragen. Werden nun die Spielserver mit einbezogen, so sind dies $\frac{\frac{p^2}{\lceil 0{,}5*p \rceil}*(m-1)}{6}$ Abfragen. Wie zuvor gilt auch hier $m-1$, weil der Algorithmus sonst vorzeitig abbricht. Da in diesem Szenario bis zu $\lceil 0,5*p \rceil$ Schritte nach außen gegangen werden kann, ergibt sich ein Laufzeitverhalten von $O(\lceil 0,5*p \rceil * \log(\frac{\frac{p^2}{\lceil 0{,}5*p \rceil}*(m-1)}{6}))$.

6. Implementierung

Zur Umsetzung der vorgestellten Verfahren wurde ein System entwickelt in dem diese eingebettet wurden, worüber dieses Kapitel einen Überblick verschaffen wird. Dazu gibt es zunächst einen allgemeinen Überblick über das System. Darauf folgend wird die Architektur des entwickelten Systems vorgestellt. Als nächstes werden die Schnittstellen vorgestellt, über die mit dem Load Balancer kommuniziert werden kann. Im nächsten Abschnitt wird auf das Konsistenzproblemproblem eingegangen, das besteht wenn sich Daten, während der Berechnung eines Spielservers, verändern. Zum besseren Verständnis des Systems werden daraufhin, Anwendungsfälle aufgezeigt und wie diese umgesetzt wurden. Zum Abschluss wird noch im Detail auf die Implementierung der Verfahren eingegangen.

6.1. Allgemein

Der Load Balancer wurde in Erlang umgesetzt. Dies Ermöglicht eine leichte Integration unseres Systems in das Gesamtsystem, da ein Großteil der schon bestehen Serverlandschaft von Timadorus, wie der Spielserver und das QuP-System, in Erlang umgesetzt wurden. Mit OTP bringt Erlang nämlich von Haus aus ein Framework zur Nachrichtenübermittlung mit, welches genutzt werden kann um schnell und effizient Schnittstellen zwischen den Systemen zu realisieren [Erl].

6.1.1. Integration in das Gesamtsystem

Innerhalb des Gesamtsystems bekommt der Load Balancer Anfragen nach Spielservern für Spieler von dem Authentifizierungsserver und sendet Positionsanfragen an den QuP. Zusätzlich erhält der Load Balancer regelmäßig Statusinformationen von den Spielservern (Abb. 6.1). Im Folgenden soll auf diese Verbindungen zwischen den Komponenten im System näher eingegangen und dargestellt werden, wozu diese benötigt werden.

Authentifizierungsserver Über den Authentifizierungsserver meldet sich der Spieler an. Da dieser dem Client mitteilt, mit welchem Spielserver sich dieser Verbinden soll, muss der

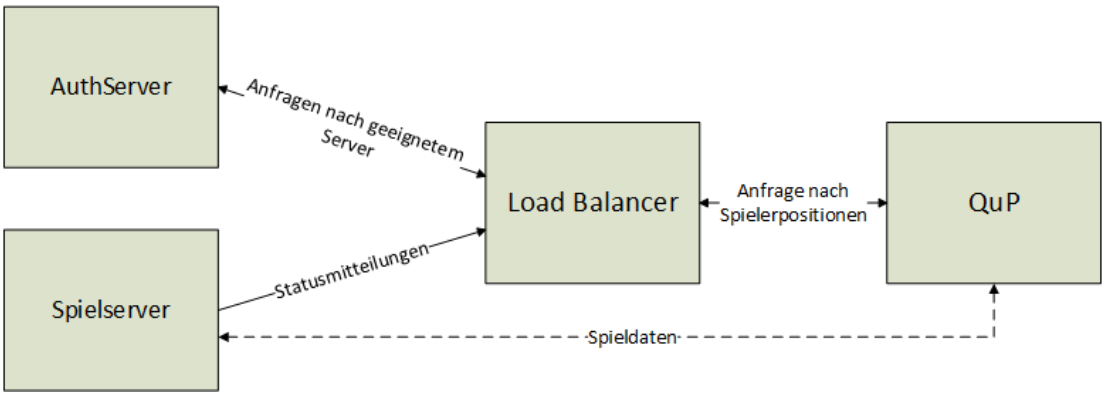


Abbildung 6.1.: Kommunikation des Load Balancers mit anderen Systemen

Authentifizierungsserver beim Load Balancing Server anfragen, welcher Spielserver für den entsprechenden Spieler geeignet ist.

Spielserver Damit der Load Balancing Server auch geeignete Zuweisungen für die Clients tätigen kann, benötigt der Load Balancing Server den aktuellen Status der einzelnen Spielserver und muss informiert werden, wenn neue Spielserver starten bzw. heruntergefahren werden. Der Spielserver sendet regelmäßig Statusinformationen an den Load Balancer. Zusätzlich bietet der Load Balancer Schnittstellen über die sich Spielserver beim Load Balancer an- bzw. abmelden können. Außerdem überwacht der Load Balancer, ob sich ein Spielserver noch im System befindet. Somit kann festgestellt werden, wenn ein Spielserver abstürzen sollte.

QuP Da der Authentifizierungsserver nur die ID des Avatars kennt, den der Spieler ausgewählt hat, aber nicht dessen genaue Position und View Box, kann diese nicht vom Authentifizierungsserver an den Load Balancer übermittelt werden. Um allerdings eine geeignete Entscheidung, zur Zuordnung des Spielers auf einen Spielserver, treffen zu können, wird die Position und die View Box des Avatars vom Load Balancer benötigt. Dies geschieht indem der Load Balancer beim QuP-System nach den Attributen des Spielers anfragt. Mit diesen Daten kann der Load Balancer nach einem Spielserver suchen und dem Authentifizierungsserver mitteilen.

6.2. Architektur

Wie in Abb. 6.2 zu sehen ist, besteht das System aus drei Hauptkomponenten, wobei nur zwei von ihnen sich auf dem Load Balancing Server befinden. Das sind der Message Handler

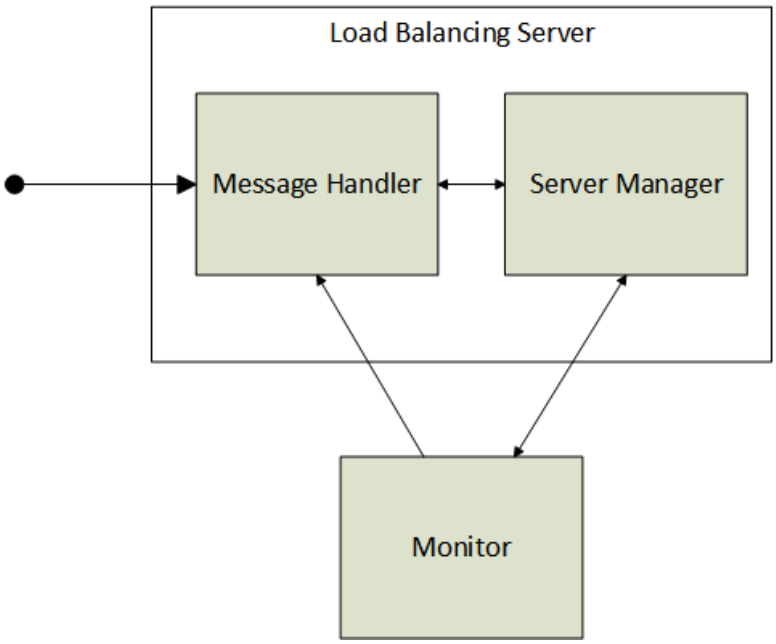


Abbildung 6.2.: Architektur des Load Balaning Systems

und der Server Manager auf Serverseite, sowie der Monitor als Client. Der Aufbau entspricht somit einer Client/Server Architektur. Der Message Handler stellt Schnittstellen zur Verfügung über die der Load Balancer auch durch externe Systeme ansprechbar ist. Diese wird von dem Authentifizierungsserver zum Ermitteln eines geeigneten Spielservers genutzt, sowie vom Monitor zum an- und abmelden der Spielserver.

Da das System in Erlang umgesetzt wurde, basiert die komplette Architektur auf Aktoren. Dadurch bildet jede Komponente einen eigenen Prozess. Dies hat den Vorteil, dass innerhalb einer Komponente keine parallele Bearbeitung stattfinden kann. Dadurch können keine Konsistenzprobleme entstehen, weil zwei Prozesse etwas zeitgleich bearbeiten. Stattdessen wird Message Passing eingesetzt. Dadurch landet jede Anfrage in einer Warteschlange und wird erst bearbeitet, wenn die Vorherige abgearbeitet ist [HÍ13, S. 136 ff.].

6.2.1. Message Handler

Der Message Handler hat die Aufgabe als Schnittstelle für den Load Balancing Server zu dienen. Hier treffen sämtliche Nachrichten vom Authentifizierungsserver sowie vom Monitor ein. Diese werden vorverarbeitet und an den Server Manager weitergeleitet. Intern besteht der Message Handler aus zwei Bestandteilen. Da wäre zum einen der Handler zur Kommunikation

mit dem Authentifizierungsserver und zum anderen der zum verarbeiten der Nachrichten vom Monitor.

Da auch der Monitor in Erlang umgesetzt wurde, wurde auch die entsprechende Schnittstelle mittels Erlang/OTP umgesetzt. Der Authentifizierungsserver basiert auf Java. Dennoch wurde auch dessen Schnittstelle in Erlang/OTP realisiert, da es mittels der JInterface-API [Jin] möglich ist auch über Java Erlang/OTP Schnittstellen anzusprechen. So konnten schnell und effizient die Schnittstellen im Load Balancer umgesetzt werden.

6.2.2. Server Manager

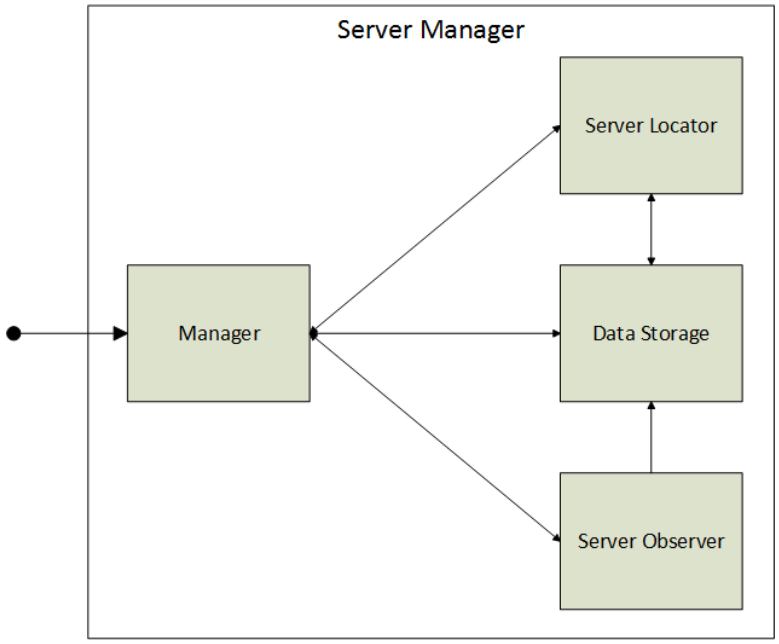


Abbildung 6.3.: Komponenten des Server Managers

Der Server Manager ist der Kern des Load Balancers. Seine Aufgabe ist das Verwalten der Spieler und Server sowie das Ermitteln eines passenden Spielservers, wenn sich ein Spieler anmeldet. Dieser besteht aus vier Unterkomponenten, dem Manager, dem Server Locator, dem Data Storage sowie dem Server Observer. Im Folgenden werden die einzelnen Komponenten genauer betrachtet.

Manager

Der Manager nimmt alle Anfragen von außen entgegen und verteilt diese an die entsprechenden Unterkomponenten. Gleichzeitig ist dieser für die Verwaltung der Server Observer zuständig. Sollte sich ein neuer Spielserver beim Load Balancer anmelden, so wird automatisch ein Server Obserer gestartet, der dafür Zuständig ist, die Zustandsübermittlungen durch den Spielserver entgegen zunehmen. Sollte sich ein Spielserver vom Load Balancer abmelden oder festgestellt werden, dass sich dieser nicht mehr meldet, so entfernt der Manager diesen entsprechend.

Server Locator

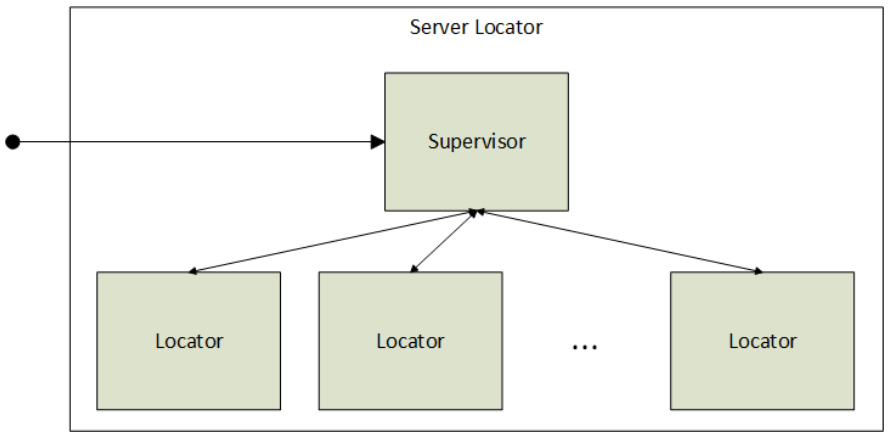


Abbildung 6.4.: Server Locator

Der Server Locator ist für das Ermitteln eines passenden Spielservers, für die Spielclients, zuständig. Dazu arbeitet dieser mit einer Supervisor Strategie und besteht aus zwei Unterkomponenten. So gibt es den Supervisor, welcher für das Starten sowie das Verwalten von Locator-Aktoren zuständig ist, sowie die Locator-Aktoren, welche die Ermittlung von Spielservern für die Spielclients übernehmen. Trifft eine Anfrage zur Berechnung ein, so nimmt der Supervisor die entgegen und startet eine neuen Locator-Aktor. Jeder Locator-Aktor ist somit für einen einzelnen Spielclient zuständig. Dies ermöglicht es uns mehrere Serverberechnungen parallel laufen zu lassen. Zu Beginn der Berechnung erhält der Locator den aktuellen Zustand des Systems. Diese Daten sind statisch und können nicht durch andere Komponenten verändert werden. Dadurch bekommen die Locator Veränderungen im System nicht mit. Das hat den Vorteil, dass diese während der kompletten Berechnung auf einen konsistenten Datenstand zurückgreift. Es gibt jedoch die Möglichkeit die Locator über Veränderungen zu informieren, da es in bestimmten Situationen, z.B. bei starken Veränderungen oder wenn ein Spielserver nicht

mehr erreichbar ist, sinnvoll ist diese zu informieren. Sollte das der Fall sein wird zunächst der Supervisor benachrichtigt, welcher dies an die Locator weiter reicht. Hat ein Locator erfolgreich einen Spielserver ermittelt, so wird dieser beim Data Storage von ihm reserviert. War dies erfolgreich, informiert der Locator den Supervisor, welcher den Locator-Aktor beendet.

Data Storage

Der Data Storage dient dem Verwalten der Zustände der Spielserver, sowie der Spielwelt. Sollten sich neue Spielserver registrieren oder nicht mehr im System sein, so wird der Data Storage benachrichtigt und trägt die entsprechenden in der Liste der Spielserver ein oder aus. Verändern sich die registrierten View Boxen auf einem Spielserver, so nimmt dies der Data Storage auf und trägt das bei sich ein. Sollte sich der Zustand eines Spielservers signifikant verändern, so informiert der Data Storage den Server Locator. Wie stark sich der Zustand eines Spielservers verändert hat, ergibt sich aus der Veränderungrate, die beim Eintragen von Veränderungen berechnet wird. Der Server Locator wird benachrichtigt, wenn dieser Wert einen in der Konfiguration definierten Wert überschreitet.

Server Observer

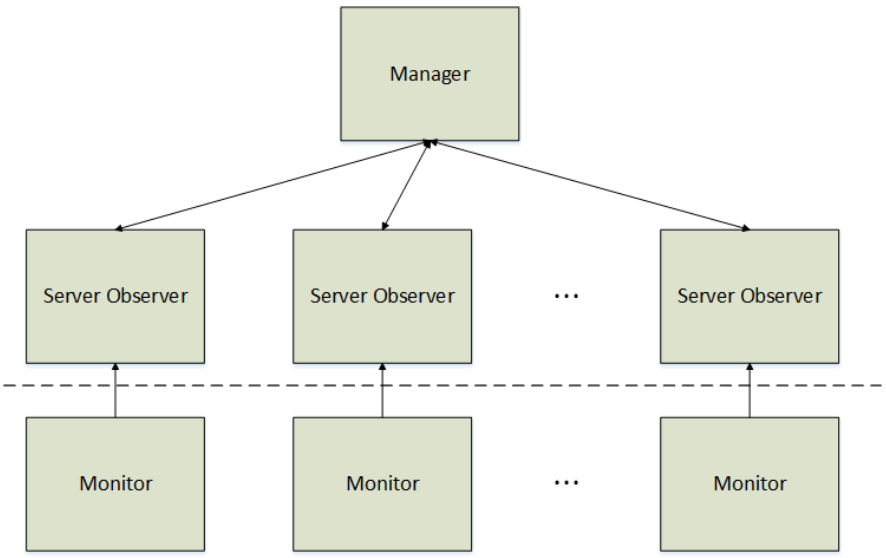


Abbildung 6.5.: Server Observer unterhalb des Managers

Die Server Observer überwachen die Zustände der Spielserver. Für jeden registrierten Spielserver wird dazu ein eigener, ihm zugeordneter, Server Observer gestartet. Der, auf dem

Spielserver befindliche, Monitor sendet daraufhin regelmäßig neue Statusinformationen an den Server Observer. Werden neue Statusnachrichten erhalten, so wird der Data Storage über den neuen Zustand des Spielservers informiert. Zeitgleich überwacht der Server Observer, ob auch regelmäßig Nachrichten eintreffen. Sollte dies für eine gewisse Zeitspanne nicht geschehen, so kann davon ausgegangen werden, dass der Spielserver nicht mehr erreichbar ist. In diesem Fall benachrichtigt der Server Observer den Manager, dass der Spielserver vermutlich nicht mehr läuft, damit dieser zunächst gesperrt wird und keine Zuweisung mehr stattfinden kann. Der Server Observer versucht nun zunächst aktiv den Spielserver zu erreichen. Sollte der Spielserver darauf reagieren oder wieder Nachrichten an der Server Observer senden, so wird der Manager benachrichtigt das der Spielserver wieder freigegeben werden kann. Sollte sich der Spielserver, nach einer in der Konfiguration festgelegten Anzahl von Nachrichten nicht beim Server Observer melden, so wird der Manager benachrichtigt und der Spielserver endgültig entfernt.

6.2.3. Monitor

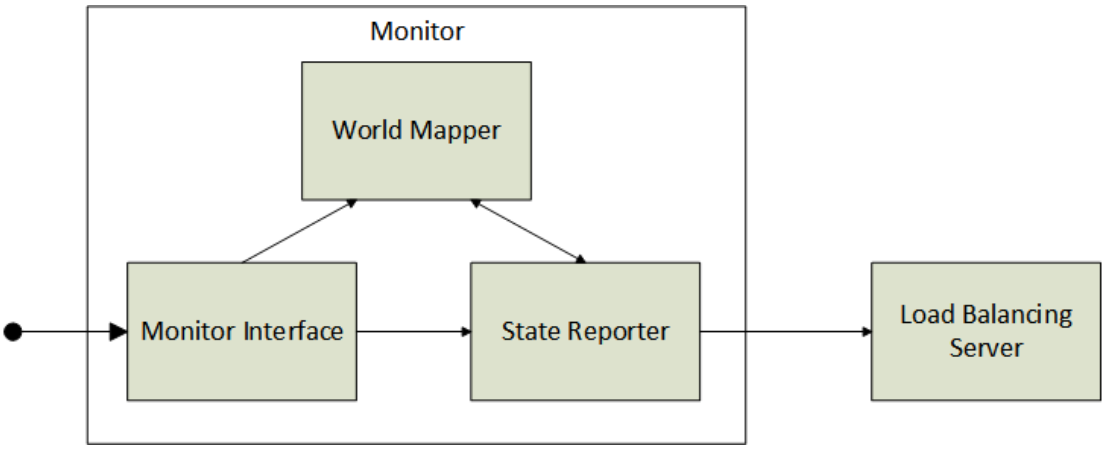


Abbildung 6.6.: Monitor

Der Monitor wird in den Spielserver integriert und soll dessen Zustand überwachen. Beim Start des Monitors meldet dieser den Spielserver automatisch beim Load Balancing Server an und erhält die ID des Spielservers sowie den Server Observer, an den die Statusnachrichten gesendet werden sollen. Sobald Veränderungen an den registrierten View Boxen stattfinden teilt der Spielserver dies dem Monitor mit. Dieser sendet die Veränderungen sowie die aktuelle Auslastung des Systems an den Load Balancing Server. Der Monitor ist in drei Komponenten

unterteilt. Dem Monitor Interface, dem World Mapper sowie dem State Reporter. Im Folgenden werden diese näher vorgestellt.

Monitor Interface

Das Monitor Interface nimmt sämtliche Nachrichten über Veränderungen auf dem Spielserver entgegen und leitet diese an den World Mapper sowie dem State Reporter weiter. Zudem meldet das Monitor Interface den Spielserver bei Start an dem Load Balancing Server an. Sobald sich der Spielserver beendet übernimmt dieser auch die Abmeldung vom Load Balancing Server.

World Mapper

Der World Mapper speichert die aktuell registrierten View Boxen des Spielservers und berechnet die Veränderungsrate, wenn eine Veränderung bei den registrierten View Boxen gibt. Sollte die Rate über den konfigurierten Höchstwert liegen, so teilt der World Mapper dies dem State Reporter mit, damit dieser eine vorzeitige Statusnachricht an den Load Balancing Server sendet. Die Veränderungsrate wird auf Grundlage eines Basiszustands berechnet. Dieser wird immer dann gesetzt wenn der State Reporter eine Statusnachricht an den Spielserver sendet. Dazu wird der aktuelle Zustand des World Mapper zum Basiszustand. Bei allen nun folgenden Veränderung wird der neue Zustand mit dem neuen Basiszustand verglichen und daraus die Veränderungsrate berechnet. Die genaue Berechnung ist vom jeweiligen Verfahren abhängig.

State Reporter

Der State Reporter benachrichtigt den Load Balancer in regelmäßigen Abständen über den Zustand des Spielservers. Dies beinhaltet die aktuelle CPU-Auslastung, den noch freien Arbeitsspeicher, in Bytes, sowie sämtliche Veränderungen, die seit der letzten Nachricht bei den View Boxen auf dem Spielserver stattgefunden haben. Neue Statusnachrichten werden gesendet, wenn es Veränderungen bei den View Boxen auf dem Spielserver gab. Allerdings ist ein zeitlicher Mindestabstand zwischen Benachrichtigungen konfigurierbar, um das Netzwerk nicht mit übermäßig vielen Nachrichten zu überlasten. Dieser kann jedoch ausgehebelt werden, sollte der World Mapper feststellen, dass die stärke der Veränderungen seit der letzten Nachricht über dem konfigurierten Höchstwert liegt. Auch Spieler die sich neu auf dem Spielserver anmelden, werden direkt an den Load Balancer übermittelt, damit die dort vorliegende Reservierung vorher nicht ausläuft. Neben einem Mindestabstand, ist auch ein maximaler Abstand konfigurierbar, falls es für einen längeren Zeitraum keine Veränderungen gibt. Wird dieser maximale Abstand erreicht so sendet der Monitor automatisch ein Heartbeat-Signal

an den Load Balancing Server, auch wenn seit der letzten Nachricht keine Veränderungen stattgefunden haben. So wird vermieden, dass dieser fehlerhaft annimmt, dass der Spielserver abgestürzt sei.

6.2.4. Integration in den Authentifizierungsserver

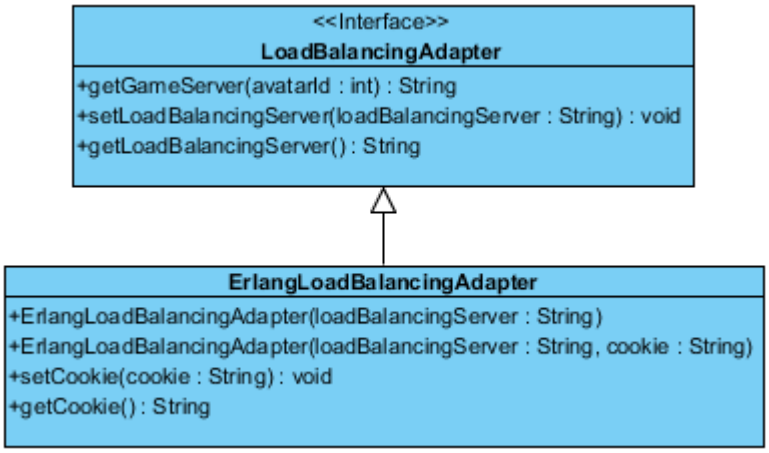


Abbildung 6.7.: Load Balancing Adapter

Da der Authentifizierungsserver in Java umgesetzt wurde, ist eine native Kommunikation mit dem Load Balancing Server nicht möglich. Deshalb wurde die Kommunikation mithilfe der JInterface-API [Jin] umgesetzt. Damit die Kopplung an die JInterface-API geringer ist, gibt es das LoadBalancingAdapter-Interface. Klassen die von diesem Interface erben, implementieren die konkrete Kommunikation mit dem Load Balancing Server. Dies ermöglicht einen leichteren Austausch der Kommunikationsform zwischen dem Authentifizierungsserver, sowie dem Load Balancing Server. Der ErlangLoadBalancingAdapter implementiert die Kommunikation über Erlang/OTP, mittels der JInterface-API.

In Abb. 6.7 kann das dazugehörige Klassendiagramm gesehen werden. Über getGameServer() fragt der Authentifizierungsserver beim Load Balancer nach einem geeigneten Spielserver für einen Spieler an. Dazu übergibt dieser die ID des Avatars, für den ein Spielserver benötigt wird. Als Rückgabewert erhält der Authentifizierungsserver den ermittelten Spielserver. Da sich die Server bei Erlang über Cookies verifizieren bietet der ErlangLoadBalancingAdapter noch zusätzlich Methoden zum Setzten dieser an.

6.3. Schnittstellen

Damit die einzelnen Bestandteile des Systems miteinander kommunizieren können, werden Schnittstellen benötigt, über die der Load Balancer angesprochen werden kann. In diesem Abschnitt sollen diese beschrieben werden. Im Folgenden sind allen synchronen Nachrichten die Bezeichnung „sync“ und allen asynchronen Nachrichten die Bezeichnung „async“ vorangestellt. Sollte keine Rückgabe angegeben sein, so handelt es sich um eine asynchrone Nachricht oder es wird nur *ok* zurückgegeben, als Bestätigung der erfolgreichen Verarbeitung.

6.3.1. An- und Abmelden von Spielservern

Wenn neue Spielserver zum System stoßen, müssen sich diese bei dem Load Balancing Server registrieren. Genauso müssen sie sich abmelden, wenn sie herunterfahren. Dies geschieht über das lb_server_registration_service-Modul. Dieses ist Bestandteil des Message Handlers und dient der An- und Abmeldung von Spielservern. Zur eindeutigen Identifikation der Spielserver, erhalten diese beim Anmelden eine ID vom Load Balancer. Das An- und Abmelden geschieht über folgende Nachrichten:

sync - {hello, Spielserveradresse}

Mit dieser Nachricht meldet sich der Spielserver beim Load Balancing Server an.

Parameter

Spielserveradresse Die Adresse des Spielservers, unter der dieser von außen zu erreichen ist.

Rückgabe

Die ID, die dem Spielserver zugewiesen wurde.

sync - {good_bye, SpielserverID}

Mit dieser Nachricht meldet sich der Spielserver beim Load Balancing Server ab.

Parameter

SpielserverID Die ID des Spielservers.

sync - {address_changed, SpielserverID, AlteAdresse, NeueAdresse}

Mit dieser Nachricht meldet sich der Spielserver beim Load Balancing Server, um mitzuteilen, dass sich die Adresse über die der Spielserver erreichbar ist, verändert hat.

Parameter

SpielserverID Die ID des Spielservers.

AlteAdresse Die vorherige Adresse des Spielservers, unter der dieser von außen zu erreichen war.

NeueAdresse Die neue Adresse des Spielservers, unter der dieser von außen zu erreichen ist.

6.3.2. Serverberechnung

Zur Berechnung eines Spielservers, für einen Avatar, fragt der Authentifizierungsserver beim Load Balancer nach diesem an. Das geschieht über das lb_query_service-Modul, welches Bestandteil des Message Handlers ist. Dort gibt es zwei Möglichkeiten nach einem Spielserver zu fragen. Zum einen find_game_server als Standardanfrage sowie find_new_game_server falls ein bereits angemeldeter Spieler einen neuen Spielserver benötigt. Neben diesen beiden musste auch der Client des QuP-Systems (quperl_client) um get_attributes erweitert werden. Im Folgenden sollen diese Schnittstellen näher erläutert werden:

lb_query_service

sync - {find_game_server, AvatarID}

Startet die Suche nach einem Spielserver für einen Avatar.

Parameter

AvatarID ID des Avatars, zu dem der Spielserver gesucht wird.

Rückgabe

Die Adresse des Spielservers, für den Avatar.

sync - {find_new_game_server, AvatarID, AlteSpielserverAdresse, Grund}

Startet die Suche nach einem neuen Server für den Spieler, wenn dieser bereits auf einem anderen Server angemeldet war. Dies kann z.B. der Fall sein wenn der Server nicht mehr erreichbar ist.

Parameter

AvatarID ID des Avatars zu dem der neue Spielserver gesucht wird.

AlteSpielserverAdresse Die Adresse des Spielservers, mit dem der Spieler zuvor verbunden war.

Grund Der Grund, warum ein neuer Spielserver benötigt wird. Z.B. connection_timeout, wenn die Verbindung abgebrochen ist.

Rückgabe

Die Adresse des neuen Spielservers.

quperl_client

async - {get_attributes, ObjectID, Attribute}

Fragt nach Attributen von einem Objekt. Sollte sich dieses nicht im Cache aufhalten, so wird beim QuP-Server angefragt. Im Gegensatz zu anderen Anfragen beim QuP-Server, werden auf diesem Wege erhaltene Attribute nicht in den Cache geladen. Um die ermittelten Attribute zu erhalten, muss ein Event Handler beim QuP-Client registriert werden. Sollten die Attribute aufgefunden werden, so wird ein „attributes_found"-Event ausgelöst. Dies enthält eine Property Liste mit den Attributen als Schlüssel.

Parameter

ObjectID Die ID zu dem Objekt, dessen Attribute gesucht werden.

Attribute Eine Liste der Attribute nach denen gesucht wird.

6.3.3. Zustandsübermittlung

Damit der Load Balancing Server auch während des Betriebs über den Zustand der einzelnen Spielserver informiert ist, müssen diese in regelmäßigen Abständen Informationen über ihre Auslastung sowie der aktuell registrierten View Boxen, an den Load Balancer senden. Im Folgenden werden die Nachrichten beschrieben mit denen die Statusinformationen an den Load Balancer gesendet werden. All diese Nachrichten werden vom State Reporter des Monitors an den zugehörigen Server Observer (Modul: lb_server_observer) gesendet.

async - {here_i_am, CPUAuslastung, FreierSpeicher, Name, Knoten}

Erste Nachricht des State Reporter an den Server Observer, um ihm seinen Namen sowie den Knoten auf dem er sich befindet mitzuteilen. Diese Daten werden vom Server Observer benötigt, wenn sich der State Reporter nicht mehr meldet, damit dieser aktiv anfragen kann.

Parameter

CPUAuslastung Die Auslastung des Prozessors in Prozent.

FreierSpeicher Der freie Speicher des Spielservers in Byte.

Name Der Name mit dem der State Reporter auf dem Spielserver registriert ist

Node Der Knoten auf dem sich der Spielserver befindet.

async - {heartbeat, CPUAuslastung, FreierSpeicher}

Wird übertragen, wenn die maximale Wartezeit für Veränderungen auf dem Spielserver überschritten wurde und sich in der Zeit auch keine Veränderung in den benötigten Spielbereichen der Spielwelt ergeben hat. Damit signalisiert der Spielserver dem Load Balancing Server, dass dieser immernoch erreichbar ist.

Parameter

CPUAuslastung Die Auslastung des Prozessors in Prozent.

FreierSpeicher Der freie Speicher des Spielservers in Byte.

async - {area_changes, Aenderungen, CPUAuslastung, FreierSpeicher}

Benachrichtigt den Load Balancer über alle Veränderungen, die seit der letzten Statusnachricht stattgefunden haben.

Parameter

Aenderungen Eine Liste sämtlicher Veränderungen der View Boxen die auf dem Spielserver stattgefunden haben. Jede Veränderung wird in Form eines zweier Tupels der Form *{ChangeType, ViewBox}* repräsentiert. Der *ChangeType* definiert, ob die View Box hinzugekommen ist oder entfernt wurde. Ist dieser *new*, so ist die View Box hinzugekommen, ist dieser *removed*, so wurde diese entfernt.

CPUAuslastung Die Auslastung des Prozessors in Prozent.

FreierSpeicher Der freie Speicher des Spielservers in Byte.

async - {new_avatar, AvatarID, ViewBox, Aenderungen, CPUAuslastung, FreierSpeicher}

Meldet dem Load Balancer, dass sich ein Spieler beim Spielserver angemeldet hat.

Parameter

AvatarID ID des Avatars, der sich beim Spielserver angemeldet hat.

ViewBox Die View Box des Avatars, der sich beim Spielserver angemeldet hat.

Changes Alle Veränderungen seit der letzten Statusnachricht (siehe auch *area_changes*).

CPUAuslastung Die aktuelle CPU-Auslastung in Prozent.

FreierSpeicher Der freie Speicher des Spielservers in Byte.

6.3.4. Kontaktverlust zu Spielserver

Sollte der Spielserver keine neuen Statusinformationen senden, so versucht der Server Observer zuerst aktiv diesen zu erreichen. Sollte dies der Fall sein sendet der Server Observer eine „where_are_you“-Nachricht an den State Reporter. Sollte diese bei ihm ankommen, so antwortet dieser mit einer „i_am_here“-Nachricht:

State Reporter

async - where_are_you

Fragt den State Reporter ob dieser noch erreichbar ist. Sollte dieser eine solche Nachricht erhalten, so antwortet der State Reporter umgehend mit einer „i_am_here“-Nachricht.

Server Observer

async - i_am_here

Bestätigt dem Server Observer, dass der Spielserver noch erreichbar ist. Nach Eintreffen der Nachricht, wird der Spielserver direkt wieder freigegeben, sodass ihm wieder Spieler zugewiesen werden können.

6.4. Konsistenzproblem

Da bei der Serversuche nach einer Lösung gesucht wird, die eine ausreichende Annäherung an eine optimale Lösung ist und nicht zwingend der optimalen Lösung entsprechen muss, wird während der Berechnung nicht jede Veränderung der View Boxen auf den Servern den Locator-Aktoren mitgeteilt. Stattdessen arbeiten diese mit dem Stand, der vorlag, als die Berechnung gestartet wurde. Das hat den Vorteil, dass der Algorithmus durchgehend mit konsistenten Daten arbeitet. Zumal davon ausgegangen werden kann, dass die Veränderungen während der Berechnung so gering sind, dass sich der Algorithmus nicht zu weit vom Optimum entfernt. Es gibt jedoch Sonderfälle in denen die Veränderung eines Spielservers so stark ist, als dass die Locator-Aktoren informiert werden müssen. Dies kann der Fall sein wenn sich die View Boxen auf einem Spielserver sehr stark verändert haben oder ein Spielserver entfernt wurde. Daher

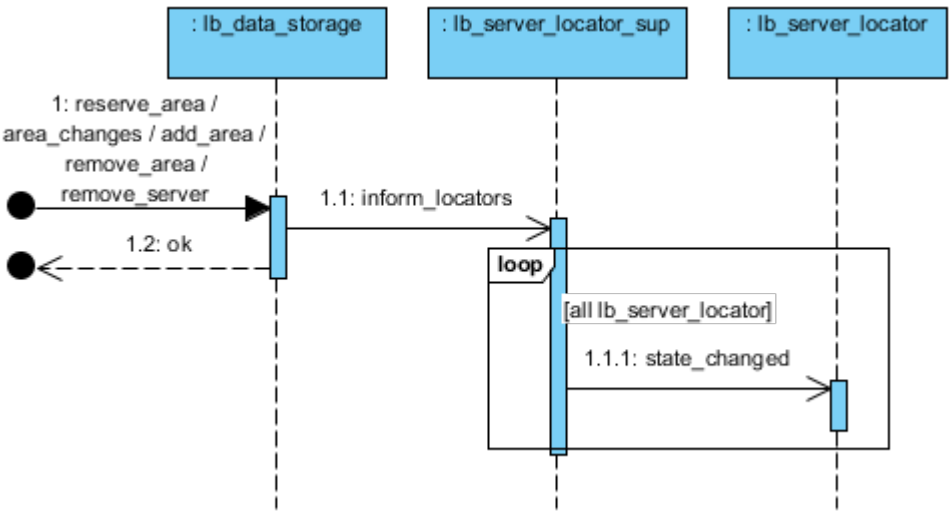


Abbildung 6.8.: Verfahren bei starker Veränderung eines Spielservers

gibt es die Möglichkeit, dass die Locator-Aktoren über den neuen Zustand benachrichtigt werden können.

Damit Locator-Aktoren mit veralteten Informationen nicht die Möglichkeit haben Spieler einem Spielserver zuzuweisen, gibt es zu allen Zustandsinformationen Zeitstempel. Gibt es nun eine starke Veränderung für einen Spielserver, so wird ein aktueller Zeitstempel ermittelt. Alle folgenden Zuweisungen auf diesen Server müssen einen Zeitstempel vorweisen, der neuer oder gleich diesem Zeitstempel ist. Sollte dieser älter sein, so wird die Zuweisung abgelehnt. Damit dieses Szenario nach Möglichkeit nicht Zustande kommt, werden die Locator-Aktoren bei starken Veränderungen direkt darüber Informiert. Dazu informiert der Data Storage den Server Locator Supervisor, welcher die Benachrichtigung an alle aktiven Locator-Aktoren weiterleitet (siehe auch Abb. 6.8). Für ein besseres Verständnis soll im Folgenden die Mechanik anhand eine Beispiels verdeutlicht werden.

Zuweisung auf Spielserver mit geringer Spielerzahl

Sollte ein Spielserver eine sehr geringe Spielerzahl aufweisen, so ist die Wahrscheinlichkeit höher, dass eine Veränderung der View Boxen signifikant ist. Sollte z.B. ein Spieler einem Spielserver zugewiesen werden, auf dem sich noch kein Spieler befindet, so kann es passieren, dass eine parallel laufende Berechnung diesem Spielserver einen Spieler zuweist der sich auf der anderen Seite der Spielwelt befindet und die Lösung sich so extrem weit vom Optimum entfernt. Dies ist zum Beispiel kurz nach Start des Gesamtsystem ein realistisches Szenario, da

zu diesem Zeitpunkt noch kein Spielserver belegt ist. Im groben kann gesagt werden, dass die Veränderung auf den Spielservern bei Zuweisung eines Spielers um so größer ist je weniger Spieler sich zu dem Zeitpunkt auf dem Spielserver befinden. Bei dem zuvor genannten Beispiel würde nun der Spielclient die Zuweisung bekommen, dessen Reservierung zuerst beim Data Storage eintrifft. Aufgrund der starken Veränderung (in diesem Fall um $\infty\%$) würde für den Spielserver ein neuer Zeitstempel gesetzt werden, der mindestens benötigt wird, um eine Zuweisung auf diesen zu machen. Durch die Benachrichtigung über den Supervisor kann auch der andere Locator vorzeitig eine erneute Berechnung starten, bevor er versucht den selben Spielserver für seinen Spieler beim Data Storage zu reservieren. Sollte die Nachricht nicht vor dem Versuch eintreffen, so wird die Reservierung vom Data Storage aufgrund des veralteten Zeitstempels abgelehnt und der Locator würde auf die Benachrichtigung warten, um eine erneute Berechnung zu starten.

6.5. Realisierung bestimmter Anwendungsfälle

Um ein besseres Verständnis davon zu vermitteln, wie die einzelnen Komponenten des Gesamtsystems miteinander arbeiten, soll in diesem Abschnitt die Umsetzung einiger Anwendungsfälle dargestellt werden.

6.5.1. Authentifizierung mit Berechnung

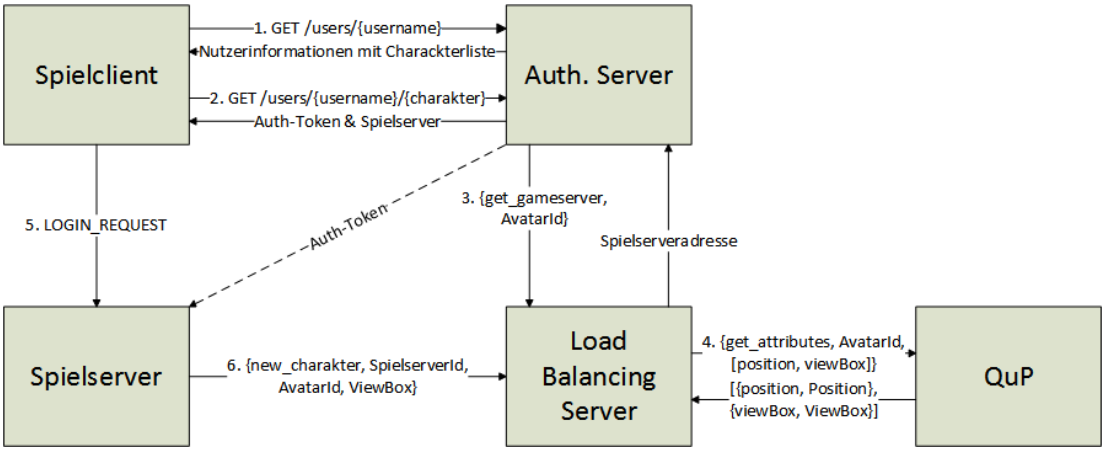


Abbildung 6.9.: Authentifizierungsvorgang

Damit der Load Balancer in das Anmeldeverfahren eingebunden wird, musste das bisherige Verfahren (siehe Abschnitt 2.4.1) erweitert werden. Abb. 6.9 stellt den neuen erweiterten Ablauf

zum Anmelden in das System dar. Bei Vergleich mit dem vorherigen Verfahren aus Abb. 2.4 ist sichtbar, dass die Schritte 3, 4 sowie 6 neu hinzugekommen sind.

Zunächst besorgt sich der Spieler, wie bisher, seine Spielerdaten und wählt daraufhin seinen Avatar aus (1. und 2.). Statt das der Authentifizierungsserver wie bisher, alle Daten selber ermittelt und daraufhin den Auth-Token zurück gibt, leitet dieser eine Anfrage an den Load Balancing Server, damit der einen geeigneten Spielserver ermittelt (3.). Da weder der Authentifizierungsserver noch der Load Balancing Server die Position bzw. die View Boxen von einzelnen Spielcharakteren kennen, muss dies über den QuP ermittelt werden. Deshalb fragt der Load Balancing Server vorm Berechnen des Servers beim QuP nach diesen Werten des Spielers (4.). Sobald der Load Balancing Server diese Daten erhalten hat, kann ein passender Spielserver ermittelt werden. Da zwischen dem Übermitteln des Spielservers an den Spielclient und dem tatsächlichen Verbinden des Spielclients auf dem Spielserver eine zeitliche Differenz liegt, reserviert der Load Balancer die View Box intern für den Spielserver. Reservierte View Boxen werden bereits als Bestandteil, der auf dem Spielserver befindlichen View Boxen, betrachtet. Dadurch kann vermieden werden, dass es fehlerhafte Zuweisungen gibt, die mit der View Box des Spielers auf dem Spielserver hätten nicht stattfinden dürfen. Sollte die Reservierung nicht innerhalb einer bestimmten Zeit von dem Spielserver bestätigt werden, so wird die View Box wieder aus der Liste der View Boxen auf dem Spielserver entfernt. Der ermittelte Spielserver wird nach der Berechnung an den Authentifizierungsserver zurückgegeben, sodass dieser daraus den Auth-Token bilden kann. Mit den ermittelten Daten kann der Client sich nun wie bisher an dem Spielserver anmelden (5.). Sollte die Anmeldung erfolgreich sein, so meldet der Spielserver beim Load Balancing Server, dass sich der Spieler bei ihm angemeldet hat (6.) und bestätigt damit gleichzeitig die Reservierung.

Berechnung

Dieser Abschnitt wird einen Überblick darüber geben, wie die Berechnung eines Spielservers im Detail abläuft. Der Verlauf einer erfolgreichen Anfrage wird auch im Sequenzdiagramm in Abb. 6.10 dargestellt.

Die Berechnung eines Spielservers wird vom Authentifizierungsserver angestoßen. Diese wird vom lb_query_service entgegen genommen. Damit der lb_query_service nicht blockiert, die Anfrage aber aus Sicht des Authentifizierungsserver wie eine synchrone Anfrage erscheint, antwortet der Aktor nicht direkt, sondern speichert den Sender zwischen. Als nächstes wird die Anfrage an den Server Manager weitergeleitet, der diese an den Server Locator Supervisor

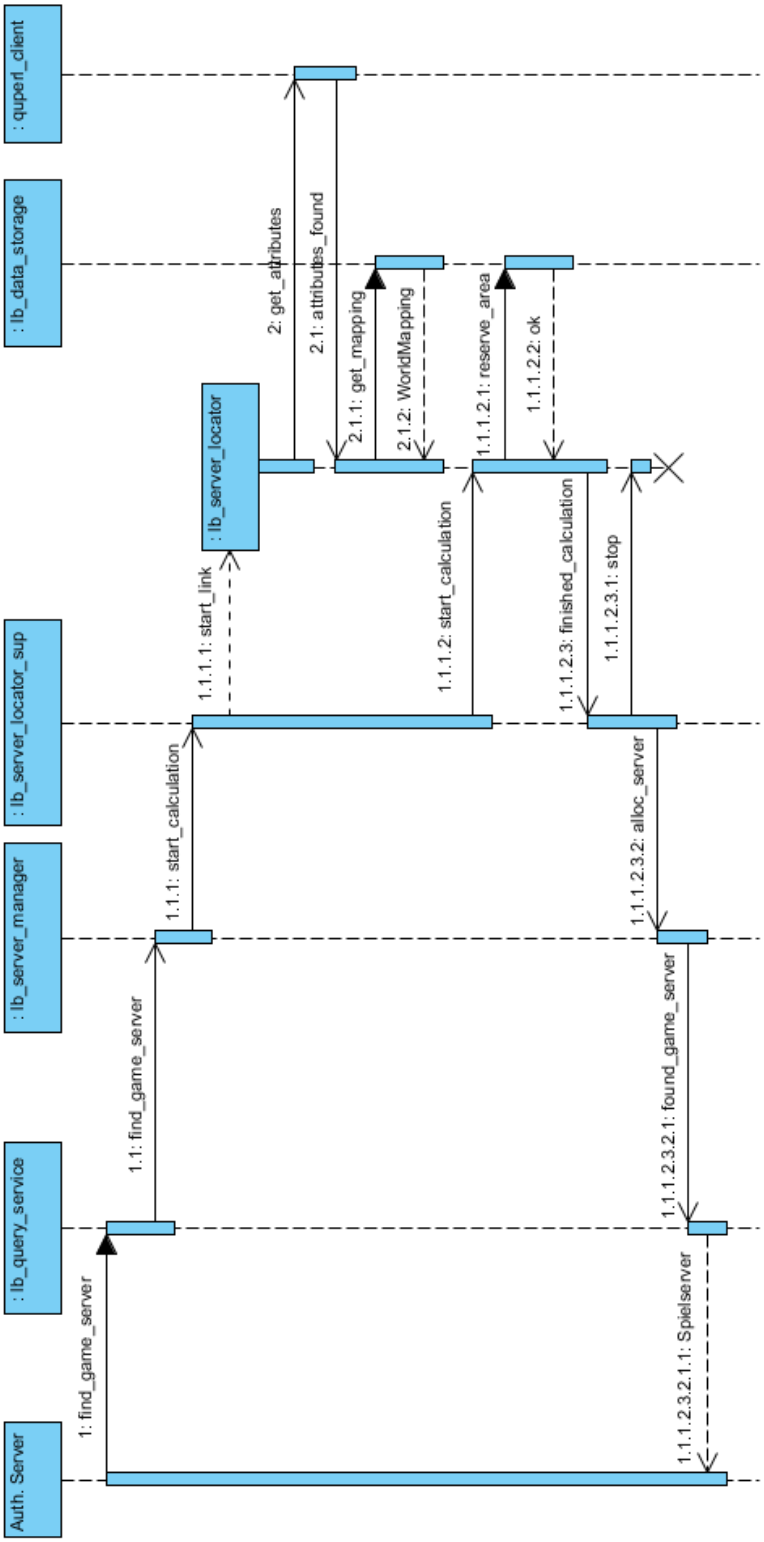


Abbildung 6.10.: Berechnung eines Spielservers als Sequenzdiagramm.

weiterleitet. Dies veranlasst den Supervisor einen neuen Locator-Aktor zu starten. Da dieser zum Berechnen eines Spielservers die View Box sowie die Position des Spielers benötigt, fragt der Locator-Aktor während des Starts beim QuP-System nach diesen Attributen des neuen Avatars. Sobald dieser die aktuelle Position sowie die View Box des Avatars mitgeteilt hat, erkundigt sich der Locator-Aktor, beim Data Storage, nach den aktuellen Zuständen der Spielserver sowie der Spielwelt.

Nachdem er den Locator-Aktor gestartet hat stößt der Supervisor die Berechnung mittels einer „start_calculation"-Nachricht an. Da es jedoch sein kann, dass zu diesem Zeitpunkt die Position sowie die View Box von dem QuP noch nicht gesendet wurde, kann die Berechnung auch verzögert stattfinden. Sollte dies der Fall sein, so merkt sich der Locator-Aktor das die Nachricht gesendet wurde und startet die Berechnung sobald die Informationen vom QuP geliefert werden. Wenn die Berechnung fertig ist versucht der Locator-Aktor den ermittelten Spielserver, für den Spieler, zu reservieren. Sollte dies nicht erfolgreich sein, so startet die Berechnung erneut. Nach erfolgreicher Reservierung, meldet der Locator-Aktor dem Supervisor, dass die Berechnung erfolgreich war. Dieser beendet daraufhin den Locator-Aktor und informiert den Server Manager über den ermittelten Spielserver. Der Server Manager informiert dann den lb_query_service, welcher die Anfrage heraussucht und dem Authentifizierungsserver antwortet.

6.5.2. Server fällt aus

Sollte ein Spielserver für eine bestimmte Zeit keine Statusnachricht senden, so geht der Load Balancer davon aus, dass dieser ausgefallen ist. Sollte dies passieren, sperrt der Server Observer den Spielserver zunächst beim Data Storage und versucht diesen aktiv selber zu erreichen. Sollte nun ein Locator-Aktor versuchen eine View Box auf diesen zu reservieren, so lehnt der Data Storage dies mit der Begründung „heartbeat_timeout" ab. Der Server Observer des Spielservers sendet in der Zwischenzeit eine in der Konfiguration festgelegte Anzahl an „wher_are_you"-Nachrichten in auch in der Konfiguration festgelegten Abständen an den Spielserver. Empfängt der Spielserver so eine Nachricht, antwortet dieser umgehend mit einer „i_am_here"-Nachricht. Sollten nach der maximalen Anzahl an „wher_are_you"-Nachrichten keine Meldung vom Spielserver eintreffen, so informiert der Server Observer den Server Manager, dass der Spielserver ausgefallen ist und beendet sich selbst. Der Server Manager entfernt nun den Spielserver aus all seinen Listen und löscht ihn aus dem Data Storage.

Sollte ein Spieler nach einem neuen Spielserver anfragen, weil mit dem bisherigen keine

Verbindung mehr hergestellt werden kann, so wird der Observer des bisherigen Spielservers darüber benachrichtigt. Wenn auch dieser bereits die Verbindung verloren hat, dann geht dieser direkt davon aus, dass der Spielserver nicht mehr erreichbar ist und bricht die aktiven Versuche den Spielserver zu finden ab.

6.5.3. Neuer Spielerver meldet sich an

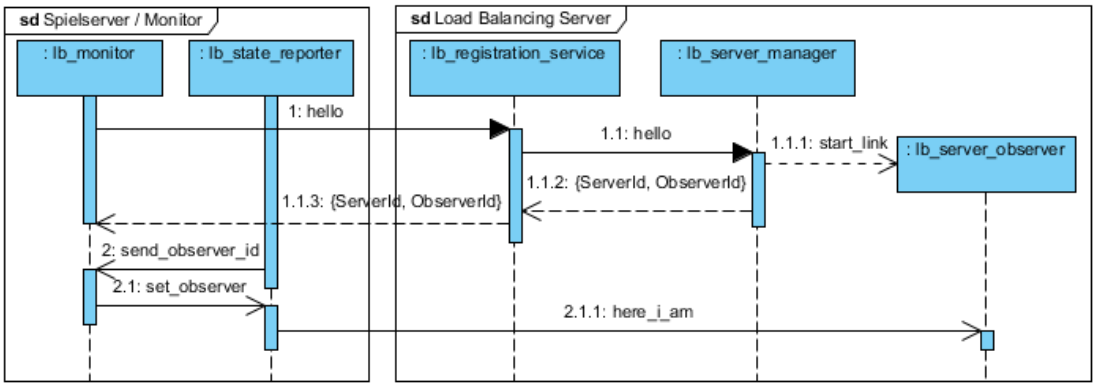


Abbildung 6.11.: Anmeldung des Spielservers als Sequenzdiagramm.

Wenn der Spielserver startet, meldet sich dieser automatisch beim Load Balancing Server an. Dazu sendet das Monitor Interface (Modul: lb_monitor) beim Starten eine „hello“-Nachricht an den lb_registration_service auf dem Load Balancing Server. Dieser reicht das an den Server Manager weiter, welcher einen Server Observer für den Spielserver startet und eine ID für diesen generiert. Diese werden beide als Rückgabewert an das Monitor Interface zurück gegeben. Der State Reporter des Monitors sendet beim Start eine Nachricht an das Monitor Interface, um ihm zu signalisieren, dass er bereit ist und die ID vom Observer erhalten möchte. Sobald der Aktor diese Nachricht entgegen nimmt, sendet dieser dem State Reporter die ID des Observers zu. Dies kann er direkt machen, denn zu diesem Zeitpunkt hat der Monitor schon sicher einen Observer erhalten, da die Registrierung am Load Balancer synchron während der Initialisierung geschah. Nachdem der State Reporter den Observer erhalten hat, sendet er an diesen eine „here_i_am“-Nachricht um zu signalisieren, dass er bereit ist. Außerdem überträgt er auf diesem Weg schon die ersten Lastinformationen sowie den Namen unter dem der State Reporter registriert ist, damit der Server Observer ggf. eine aktive Suche starten kann.

6.5.4. Spielserver meldet sich ab

Sollte der Spielserver sich beenden, so meldet sich dieser automatisch beim Load Balancing Server ab. Dazu sendet der Monitor eine "´good_bye“-Nachricht an den Load Balancing Server. In der Folge wird der Observer beendet und der Spielserver aus allen Listen im Server Manager sowie aus dem Data Storage entfernt.

6.5.5. Übermitteln des Serverzustands

Der Zustand des Spielservers wird über den State Reporter verwaltet. Dazu trägt dieser in einer Liste sämtliche Veränderungen der View Boxen ein, die seit dem letzten Übermitteln der Veränderungen stattgefunden haben. Zudem misst der State Reporter in regelmäßigen Abständen die CPU-Auslastung. Der übermittelte Wert entspricht dem Durchschnittswert der letzten x Messungen. Wobei x ein konfigurierbarer Wert ist, genau wie die Abstände in denen gemessen werden soll. Dies soll verhindern, dass Aufgrund von kurzfristigen Schwankungen ein ungenauer Wert an den Load Balancer übermittelt wird. Damit das Netzwerk nicht zu stark überlastet wird, kann ein Mindestabstand zwischen den Statusinformationen konfiguriert werden. Sollte es bis zu diesem Zeitpunkt noch keine Veränderungen gegeben haben, so wird bei der nächsten Veränderung eine Statusnachricht gesendet. Gab es jedoch bis zum konfigurierten maximalen Abstand keine Veränderung, so wird ein Heartbeat-Signal zusammen mit der aktuellen CPU-Auslastung sowie dem noch vorhandenen Arbeitsspeicher in Bytes gesendet. Der Mindestabstand kann jedoch unterschritten werden, wenn sich ein Spieler neu anmeldet oder eine Veränderung auf dem Spielserver zu stark war.

6.6. Umsetzung der Verfahren

Damit zwischen den Verfahren gewechselt werden kann, ohne dass Programmcode angepasst werden muss, kann in einer Konfigurationsdatei eingetragen werden, welches verwendet werden soll. Zum Programmstart liest der Load Balancer die Konfigurationsdatei ein und sucht die benötigten Erlangmodule heraus. Um einen einheitlichen Zugriff auf die Funktionalitäten der Verfahren zu haben werden Erlangs Behaviours eingesetzt. Diese Erlauben es zu definieren, welche Funktionen bestimmte Module implementiert haben sollen.

In jedem Fall, in denen eine für das Verfahren spezifische Implementierung benötigt wird, gibt es einen, von dem Verfahren unabhängigen, Aktor, der die Anfragen entgegen nimmt. Diesem wurde zum Start das Modul übergeben, welches das Verhalten des Verfahrens implementiert.

Kommt nun eine Anfrage, so ruft der Aktor auf dem implementierenden Modul, die entsprechende Funktion auf. Damit nur das Verfahren und nicht sämtliche Module angegeben werden müssen, gibt es für jeden Fall in denen ein Modul benötigt wird, eine Namenskonvention, die implementierende Module einhalten müssen. Diese setzt sich immer aus dem Namen des Verfahrens sowie einem fallspezifischen zweiten Teil zusammen. Zum Start des Load Balancers werden für jedes benötigte Modul, diese beiden Teile zusammengesetzt und überprüft ob das Modul vorliegt.

6.6.1. Load Balancing Server

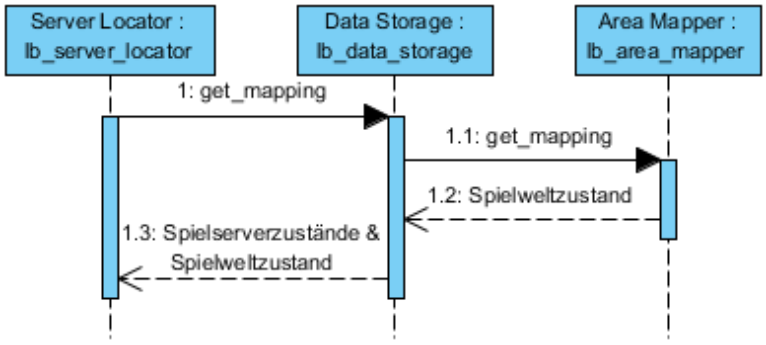


Abbildung 6.12.: Beziehen des Spielweltzustandes.

Im Load Balancing Server werden in zwei Fällen verfahrensspezifische Implementationen benötigt. Zum einen zum Verwalten des Spielweltzustands (Area Mapper) und zum anderen zum Ermitteln eines geeigneten Servers (Server Locator). Da der Server Locator zur Berechnung den Zustand der Spielwelt benötigt und deren Verwaltung vom Vefahren abhängig ist, wird dieser vom Data Storage über den Area Mapper bezogen (siehe Abb. 6.12). So kann für jedes Verfahren eine eigene unabhängige Struktur verwendet werden. Im Folgenden werden die beiden Aktoren vorgestellt, welche die Funktionen auf den implementierenden Modulen aufrufen.

Area Mapper

Der Area Mapper ist für die Zuordnung der Spielwelt und View Boxen zu den Spielservern zuständig und befindet sich hinter dem Data Storage. Alle Anfragen die an dem Data Storage eintreffen und die View Boxen auf den Spielservern betreffen, werden an den Area Mapper-Aktor weitergeleitet. Dies geschieht aus Sicht des restlichen Systems vollkommen transparent. Alle implementierenden Module müssen das lb_area_mapper-Behaviour umgesetzt haben.

Die Namenskonvention für Implementierungen des Behaviours ist der Verfahrensname + „_area_mapper“.

Server Locator

Die Server Locator sind für die Ermittlung des Spielservers zuständig. Bei diesen handelt es sich um die Locator-Aktoren aus der Server Locator-Komponente. Die implementierenden Module müssen das lb_server_locator-Behaviour umsetzen. Sollten nun eine Anfrage beim Locator-Aktor eintreffen, so werden diese gegebenenfalls von diesem vorverarbeitet und anschließend die entsprechende Funktion auf dem Modul aufgerufen. So kümmert sich der Aktor bereits darum, dass die Spielerdaten vom QuP bezogen werden und besorgt vor Start der Berechnung, den aktuellen Zustand der Spielserver sowie der Spielwelt vom Data Storage. Alle Module, die das Verhalten eines Server Locators implementieren, müssen per Namenskonventionen den Verfahrensnamen + „_server_locator“ als Modulnamen haben.

6.6.2. Monitor

Da auch der Monitor die Belegung der Spielwelt des Spielservers verwaltet, wird an dieser Stelle ein für das genutzte Verfahren spezifisches Verhalten benötigt. Im Folgenden soll der zuständige Aktor vorgestellt werden.

World Mapper

Der World Mapper speichert ab, welche View Boxen auf dem Spielserver im Einsatz sind. Zudem berechnet dieser auch die Veränderungsrate bei jeder Veränderung. Da diese Berechnung vom genutzten Verfahren abhängig ist, wird auch hier auf Module zurückgegriffen, welche deren Verhalten implementieren. Diese müssen das lb_world_mapper-Behaviour umsetzten. Bei sämtlichen Anfragen, die nun beim World Mapper eintreffen und die View Boxen der Spielwelt betreffen, werden die Funktionen des entsprechenden Moduls aufgerufen.

6.6.3. Genutzte Datenstrukturen

Da die Leistung der Verfahren auch abhängig von den genutzten Datenstrukturen ist, wird im folgenden Abschnitt beschrieben, welche Datenstrukturen eingesetzt werden. Dazu wird auf die genutzten Strukturen auf Serverseite sowie auf Seiten des Monitors eingegangen.

Das graphbasierte Verfahren

Damit die Berechnung stattfinden kann, müssen zu sämtlichen Spielservern die registrierten View Boxen abgespeichert werden. Zudem müssen die Minimal- sowie Maximalwerte einer jeden Achse für die einzelnen Spielserver abgespeichert werden, damit die Veränderungsrate berechnet werden kann. Während bei den registrierten View Boxen, jegliche Datenstrukturen eingesetzt werden können, bei denen über dem Spielserver auf die View Boxen zugegriffen werden kann, müssen die Minimal- und Maximalwerte in einer sortierten Datenstruktur abgelegt werden. Da auch die Minimal- und Maximalwerte abhängig von den Spielservern sind, muss auch auf die, über die Spielserver, zugegriffen werden können.

Load Balancing Server

Auf dem Load Balancing Server werden die Daten zu jedem Spielserver in einem Tupel aus vier Elementen gehalten. Das sind zum einen drei Tabellen mit den jeweiligen Minimal- und Maximalwerten einer jeden Achse sowie einer Map mit den registrierten View Boxen als Schlüssel und deren Anzahl als Wert. Zugegriffen werden kann auf das Tupel über eine Map mit dem zugehörigen Spielserver als Schlüssel. Damit effizient auf die größten sowie kleinsten Elemente der Tabellen mit den Maximal- und Minimalwerten einer jeden Achse zugegriffen werden kann, wurden diese als „ordered_set“ ETS-Tabellen umgesetzt. Dadurch werden die Daten sortiert abgelegt und es besteht somit die Möglichkeit effizient an die größten und kleinsten Werte zu kommen. Durch die Realisierung der registrierten View Boxen mithilfe einer Map beträgt die Zugriffszeit $O(log(n))$. Wäre dies mittels einer ETS-Tabelle realisiert worden, so läge eine Zugriffszeit von $O(1)$ vor. Dies hätte jedoch den Nachteil, dass die Daten hätten jedes mal kopiert werden müssen, wenn diese für die Berechnung zur Verfügung gestellt worden wären, da eine ETS-Tabelle nicht persistent und somit veränderbar ist. Sonst hätte während der Berechnung nicht auf einen konsistenten Datensatz zugegriffen werden können, da jede Veränderung auch dort sichtbar gewesen wäre. Im Gegensatz dazu handelt es sich bei der von Erlang zur Verfügung gestellten Map um eine persistente Datenstruktur. Zudem verringert dies auch nicht die Laufzeit des Verfahrens, welches mit $O(n)$ eine schlechtere Laufzeit als die Map aufweist.

Sollten die Daten nun für die Berechnung eines Spielservers benötigt werden, so wird aus den vorliegenden Daten eine Map, mit den Servern als Schlüssel und für die Berechnung angepasste Daten als Wert, generiert. Dazu werden die Werte der Server von einem Tupel aus vier Elementen in ein Tupel aus zwei Elementen umgewandelt. Dieses besteht zum einen aus der Map der vorhandenen View Boxen und zum anderen einem weiteren Tupel, bestehend

aus den Minimal- und Maximalwerten einer jeden Achse. Zum generieren dieses Tupels wird aus jeder Tabelle, mit den Minimal- und Maximalwerten einer Achse, der kleinste sowie der größte Wert genommen und als Element zum Tupel hinzugefügt.

Monitor

Da der Monitor nur die Daten des Spielservers benötigt auf dem er eingesetzt wird, wird kein Mapping von den Spielservern zu ihren Daten benötigt. Dadurch ist es möglich, die Daten statt in einem Tupel, in mehreren Attributen zu halten. Wie auf dem Server werden auch hier die Minimal- sowie Maximalwerte einer jeden Achse sowie die registrieren View Boxen gespeichert. Damit schnell auf die Minima sowie die Maxima einer jeden Achse zugegriffen werden kann, sind diese auch hier in einer „ordered_set“ ETS-Tabelle abgelegt. Die registrierten View Boxen werden in einer regulären ETS-Tabelle abgespeichert. Da diese Datenstruktur veränderbar ist und Hashing einsetzt, kann in konstanter Zeit auf die Elemente zugegriffen werden. Diese kann in diesem Fall eingesetzt werden, da hier kein anderer Aktor auf die Daten zugreift. Somit kann eine veränderbare Datenstruktur eingesetzt werden ohne dass Seiteneffekte zu befürchten sind.

Die Heat Map

Intuitiv könnte die Heat Map mithilfe eines dreidimensionalen Arrays umgesetzt werden. Da jedoch davon ausgegangen werden kann, dass sich für gewöhnlich an vielen Orten der Spielwelt keine Spieler befinden, wird mit einem Array jedoch unnötig Speicherplatz verbraucht, was die mögliche Präzision deutlich beschränkt. Werden die Felder der Heat Map als Matrix betrachtet, so kann auch von einer dünn besetzten Matrix gesprochen werden. Um eine solche umzusetzen und dennoch eine konstante Zugriffszeit zu erreichen, kann dies mithilfe von Hashmaps umgesetzt werden. In Erlang gibt es zwei Möglichkeiten. Zum einen über ETS-Tabellen und zum anderen mittels den in Erlang integrierten Maps. In dem entwickelten System ist dies mittels der Maps geschehen. Im Folgenden soll nun zunächst erörtert werden, warum Maps anstatt ETS-Tabellen eingesetzt werden. Anschließend wird die Datenhaltung im Load Balancing Server sowie dem Monitor erläutert.

Map vs. ETS-Tabelle

Die Heat Map kann mithilfe der ETS-Tabelle oder der integrierten Map umgesetzt werden. Während ETS-Tabellen zwar eine konstante Zugriffszeit erlauben, müssen die Daten vor der Serverberechnung jedoch jedes mal komplett kopiert werden, damit der Locator-Aktor durchgehend auf konsistente Daten zugreifen kann, die sich, während der Berechnung nicht

verändern. Bei der Implementierung der Map in Erlang handelt es sich um Hash Array Mapped Tries [Bag01, Erl15]. Diese Erreichen bei einer destruktiven Implementierung eine konstante Zugriffszeit, jedoch ist die Datenstruktur in Erlang als persistente Datenstruktur umgesetzt worden. Dadurch reduziert sich die Zugriffszeit auf $O(log(n))$. Es besteht jedoch kein Kopieraufwand, wenn die Daten berechnet werden müssen.

Um eine geeignete Lösung zu ermitteln, wurde ausgewertet, wie sich die Datenstrukturen bei großen Datenmengen verhalten und wie groß der Kopieraufwand der ETS-Tabelle ist. Dazu wurde jeweils eine Map sowie eine ETS-Tabelle mit 4.000.000 Elementen generiert. Wobei jedes Element als Schlüssel eine aufsteigende Zahl von Eins bis 4.000.000 sowie als Wert den Wert Null erhalten hat. Anschließend wurden 1.000 verschiedene Elemente verändert und haben den Wert Eins erhalten. Zum Abschluss wurden noch 1.000 Elemente gelesen. Ermittelt wurde die totale Zeit, die benötigt wurde, um die Datenstrukturen zu generieren, die Durchschnittszeit zum Verändern der Daten sowie die Durchschnittszeit zum Lesen der Daten. Bei der ETS-Tabelle wurde zusätzlich gemessen wie groß der Zeitaufwand ist um die Daten der generierten Tabelle in eine Liste zu kopieren. Getestet wurde auf einer virtuellen Maschine mit einem 32-Bit Linux Ubuntu 14.04.1 LTS Betriebssystem, mit 512 MB Arbeitsspeicher und einem Intel i5-3230M Prozessor mit 2,6 GHz. Die genutzte Erlang Version war Erlang/OTP 18. Tabelle 6.1 zeigt die Ergebnisse des Experiments. Es ist zu sehen, dass das Erzeugen der

Tabelle 6.1.: Vergleich von Maps und ETS-Tabellen (alle Werte in μs)

	Erzeugen	**Verändern**	**Lesen**	**Kopieren**
Map	7832460	19	12	-
ETS-Tabelle	4792850	16	15	2266430

Datenstrukturen bei einer Map deutlich länger dauert (ca. 1,6 mal so lange). Schaut man sich die Zahlen allerdings genauer an, so wird man feststellen, dass sämtliche Elemente in der Map dennoch sehr schnell eingefügt wurden. Geht man von einem logarithmisch gleichmäßigen Wachstum aus, so kann die Zeit die benötigt wurde um das letzte Elements einzufügen, wie folgt ermittelt werden:

$$\begin{aligned} c * \log_2 4000000 &= x \\ c * \sum_{i=1}^{4000000} \log_2 i &= 7832460 \\ \rightarrow c &\approx 0,0955697 \\ \rightarrow x &\approx 2,0959934 \end{aligned} \tag{6.1}$$

Danach hat das Einfügen des letzten Elements nur ca. $2\mu s$ gedauert, was somit dennoch sehr schnell war. Beim Manipulieren sowie dem Lesen der Daten sind nur minimale Unterschiede festzustellen, so ist die ETS-Tabelle beim Manipulieren im Schnitt um $3\mu s$ schneller, während die Map im Schnitt $3\mu s$ schneller gelesen hat. Das Kopieren der Werte aus der ETS-Tabelle dauerte jedoch über 2s. Dies ist deutlich zu lang, wenn auch eine große Anzahl von Spielern effizient einem Spielserver zugeordnet werden soll. Denn während des Kopiervorgangs, darf niemand die Tabelle verändern, da die erstellte Liste sonst inkonsistent ist.

Aufgrund dessen, dass die Messwerte für die Maps, außer beim Erstellen, vergleichbar mit denen der ETS-Tabelle waren und das der Kopieraufwand für die Tabelle zu hoch ist, wird beim Load Balancing Server zur Implementierung eine Map eingesetzt.

Load Balancing Server

Auf dem Load Balancing Server werden neben der Heat Map auch Präzision, Schrittgröße sowie die Anzahl der genutzten Felder gespeichert. Diese Daten werden benötigt, damit die Veränderungsrate berechnet werden kann. Zudem wird in einer ETS-Tabelle abgelegt, auf welchem Server, welche Felder genutzt werden. Dies wird benötigt, damit beim Entfernen eines Spielservers, schnell ermittelt werden kann, welche Felder davon betroffen sind. Benötigt nun der Locator-Aktor das Mapping zum Berechnen eines Spielservers, so wird ein Tupel bestehend aus der Heat Map, einer Liste aller Spielserver, einer Liste der leeren Spielserver, der Anzahl der genutzten Felder, der Präzision sowie der Schrittgröße generiert.

Monitor

Viele Daten, die im Server gehalten werden, werden auch im Monitor gehalten. So werden auch hier Präzision, Schrittgröße und die Anzahl der genutzten Felder gehalten. Da hier nur der World Mapper auf die genutzten Felder zugreift, werden diese auf dem Monitor in einer ETS-Tabelle verwaltet. Zusätzlich gibt es zum Ermitteln der Veränderungsrate, jeweils eine ETS-Tabelle mit sämtlichen Feldern der Heat Map, die seit erstellen des letzten Basiszustands hinzugekommen sind oder entfernt wurden.

7. Experimente

Mithilfe von Experimenten wurde die Qualität der Verfahren ermittelt. Dies geschah in drei Kategorien: der Qualität der Gruppierung, der Speicherauslastung sowie der benötigten Rechenzeit in Form eines Lasttests. Dazu wird zunächst die Testumgebung vorgestellt und anschließend die einzelnen Testfälle ausgewertet. Zum Abschluss wird ein Fazit der Ergebnisse gezogen.

7.1. Die Testumgebung

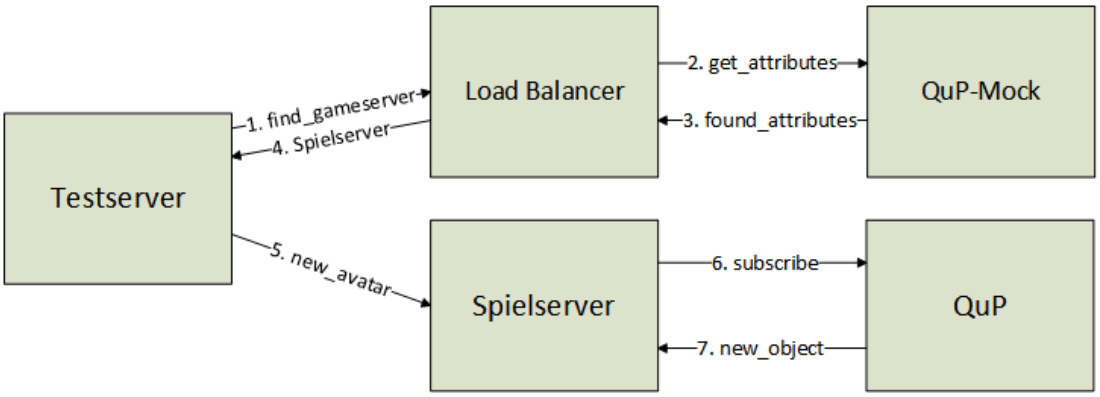


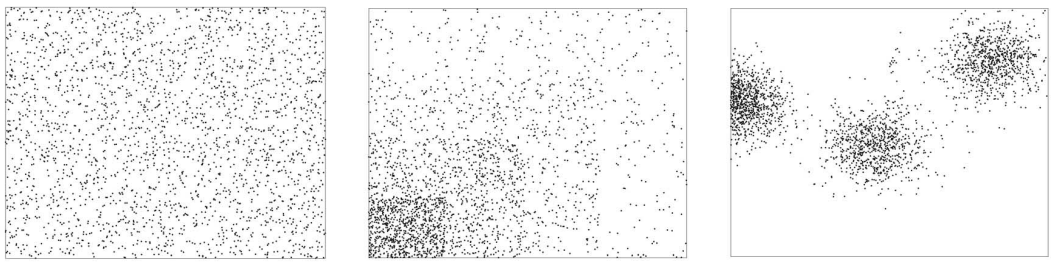
Abbildung 7.1.: Ablauf der Tests

Zum Testen standen drei Testserver mit gleicher Konfiguration zur Verfügung. Diese waren ausgestattet mit einem Linux Ubuntu 15.04 Betriebssystem, mit 2048 MB Arbeitsspeicher sowie einem Intel Xeon E5-2609 Prozessor mit 2,4 GHz. Die genutzte Erlang Version war Erlang/OTP 18.

Bei jedem Test wurde ein Load Balancing Server, ein QuP-Mock, mehrere Spielserverinstanzen sowie mehrere QuP-Instanzen gestartet. Diese wurden auf die einzelnen Server verteilt. Durchgeführt und kontrolliert wurden die Tests durch einen Testserver. Dieser fragt, für jeden Avatar, beim Load Balancer nach einem Spielserver und weist diesen dem ermittelten Spielserver zu. Der Testserver protokolliert die Zeit, die benötigt wurde, um einen Spielserver zu

ermitteln sowie den ermittelten Spielserver zu jedem Spieler. Nach jeder Zuweisung holt sich der Testserver außerdem die aktuellen Lastinformationen der Spielserver. Diese beinhalten auch, wie viele Objekte die Spielserver bereits vom QuP bezogen haben. Statt über das reguläre QuP-System bezieht der Load Balancer seine Informationen über einen QuP-Mock, dadurch kann die benötigte Zeit der Verfahren unabhängig von der Auslastung des QuP-Systems gemessen werden. Dazu werden vor dem Test, sämtliche Avatare nicht nur im QuP-System, sondern auch auf dem QuP-Mock abgespeichert. Neben dem Qup-Mock, wurde auch ein vereinfachter Spielserver implementiert. Dies verringert die Fehleranfälligkeit, die durch die Komplexität eines vollständigen Spielservers gegeben ist. Dieser registrierr bei Zuweisung eines neuen Avatars einen neuen Observer beim QuP und überprüft in regelmäßigen Abständen, wie viele Objekte vom QuP-System bezogen wurden.

Alle Tests werden in drei Szenarien durchgeführt. Dazu werden die Spieler einmal gleichmäßig



(a) (b) (c)

Abbildung 7.2.: Die verschiedenen Testszenarien: gleichmäßig verteilt (a), gedrängt (b) und gruppiert (c). Aus [LC02].

über die Spielwelt, gedrängt um einen Punkt sowie gruppiert um mehrere Punkte verteilt. Dies ist ein gängiges Vorgehen und ermöglicht einen guten Vergleich verschiedener Spielsituationen [LC02, MFOn03, MOnFD03, NSLL02]. Abb. 7.2 soll diese verschiedenen Szenarien in einem zweidimensionalen Raum visualisieren. Sämtliche Objekte wurden gleichmäßig in der Spielwelt verteilt. Alle Szenarien werden jeweils mit dem graphbasierten und dem Heat Map Verfahren getestet. Um ein Referenzwert zu erhalten, werden alle Tests zudem mit dem Round Robin Verfahren, als einfachste denkbare Lösung zum Load Balancing, durchgeführt. Damit das Verhalten möglichst realistisch ist, wird für die Spieler in zufälliger Reihenfolge nach einem Spielserver gesucht. Da die Vergleichbarkeit der Verfahren gewährt werden muss, wird diese abgespeichert und für jedes Verfahren die selbe Reihenfolge angewendet.

Für die Tests gilt, dass sich bei dem gedrängten Szenario 60% der Spieler gemeinsam in einem

Quader befinden, der sich über eine Länge von 0,3 in Höhe, Breite und Tiefe erstreckt. Die restlichen Spieler sind gleichmäßig auf die Spielwelt verteilt. Beim gruppierten Szenario befinden sich 75% der Spieler gleichmäßig verteilt in 10 Gruppen, welche wiederum gleichmäßig über die Spielwelt verteilt wurden. Auch hier wurden die restlichen Spieler gleichmäßig über die Spielwelt verteilt. Die Bounding Box sämtlicher Objekte sowie Avatare haben die Länge 0,002 in Höhe, Breite und Tiefe. Die View Box sämtlicher Avatare hat die Länge 0,05 in Höhe, Breite und Tiefe.

Die Gewichtungsfaktoren für die Kostenfunktionen sind bei dem graphbasierten Verfahren fünf für den Gruppierungsfaktor, drei für die CPU-Auslastung und 0,000000001 für den vorhandenen Speicher. Beim Heat Map Verfahren sind die Werte fünf für den Gruppierungsfaktor, zwei für die CPU-Auslastung und 0,000000001 für den vorhandenen Speicher. Zur Ermittlung des Gruppierungsfaktors für leere Spielserver sind beim Heat Map Verfahren die Gewichtungsfaktoren zur Verteilung der gesamten Spielwelt auf alle Spielserver, sowie der Verteilung der bisher genutzten Spielwelt, jeweils 0,5. Die Präzision des Heat Map Verfahrens ist 100.

Die genaue Anzahl der Avatare und Objeke in der Spielwelt sowie die Verteilung der Komponenten auf die Spielserver ist von dem Test abhängig. Im Folgenden werden diese erläutert.

7.1.1. Gruppierung der Spieler und Speicherauslastung

Die Gruppierung sowie die Speicherauslastung werden gemeinsam, in einem Testdurchlauf, getestet. Dazu werden für jeden Testfall 40.000 Objekte und 10.000 Spieler generiert, die auf der Spielwelt verteilt werden.

Die Komponenten sind wie folgt verteilt: Der Testserver wird auf Server 1 gestartet, zusammen mit zwei QuP-Servern. Auf Server 2 werden drei Spielserverinstanzen gestartet und auf Server 3 läuft der Load Balancer, zusammen mit dem QuP-Mock. Der Load Balancer und der QuP-Mock wurden hierbei bewusst auf einem separaten Server gestartet, damit diese beim Ermitteln möglichst nicht von den anderen Systemkomponenten beeinflusst werden.

7.1.2. Lasttest

Beim Lasttest werden 100.000 Avatare über die Spielwelt verteilt. Da nur überprüft wird, wie lange die Berechnung eines Spielservers dauert, werden keine zusätzlichen Objekte erzeugt. Auch wird kein QuP-Server gestartet. Stattdessen werden die Avatare nur auf dem QuP-Mock

gespeichert. Da kein QuP-Server vorhanden ist, findet nach der Zuweisung eines Spielers auf den Spielserver auch keine Anfrage an den QuP statt. Insgesamt werden bei den Lasttests sieben Spielserver gestartet.

Die Komponenten sind wie folgt verteilt: Der Testserver wird auf Server 1 gestartet, zusammen mit drei Spielservern. Auf Server 2 werden vier weitere Spielserver gestartet. Wie beim Test zur Ermittlung der Gruppierung und Speicherauslastung, werden auch beim Lasttest der Load Balancer sowie der QuP-Mock separat auf Server 3 gestartet, damit deren Verhalten nicht negativ durch die Spielserver beeinflusst wird.

7.2. Gruppierung der Spieler

Beim Test zur Gruppierung der Spieler soll überprüft werden, ob die Spieler durch die Verfahren auch in gemeinsame Gruppen eingeteilt werden. D.h. dass Spieler, die sich nah beieinander befinden, mit hoher Wahrscheinlichkeit auch dem selben Spielserver zugewiesen werden, sodass eine zusammenhängende Gruppe gebildet werden kann. Dies soll verhindern, dass sich die Zuständigkeiten von Spielservern zu stark überschneiden. Dadurch wird die Wahrscheinlichkeit von Kollisionen beim Manipulieren von Objekten verringert.

7.2.1. Metriken

Zum Messen dieses Verhaltens wurde kein klassisches Maß verwendet. Stattdessen, wurde Statistik darüber geführt, wie die Avatare auf die Spielserver verteilt wurden. Aus diesen Statistiken ist eine 3D-Grafik entstanden. Für jeden Avatar ist ein Punkt an dessen Position eingezeichnet, der abhängig vom zugewiesenen Spielserver eingefärbt wurde. Diese Grafik wurde optisch ausgewertet.

7.2.2. Bei gleichmäßiger Verteilung

Wie zu erwarten, ist bei einer gleichmäßigen Verteilung, bei einem Round Robin Verfahren, keinerlei Struktur zu erkennen (Abb. 7.3). Stattdessen ist die Verteilung der Spieler auf der Spielwelt eher zufällig. Das kommt durch die zufällige Reihenfolge, in der für die Spieler ein Spielserver ermittelt wird. Dabei werden diese den Spielservern einfach abwechselnd zugewiesen.

Beim graphbasierten Verfahren ist eine deutliche Gruppierung der Spieler zu sehen (Abb. 7.4).

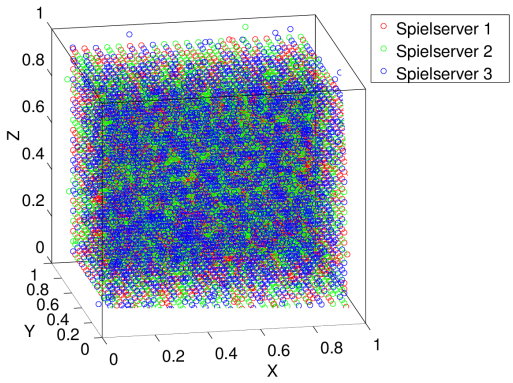


Abbildung 7.3.: Aufteilung der Spieler beim Round Robin Verfahren bei gleichmäßiger Verteilung

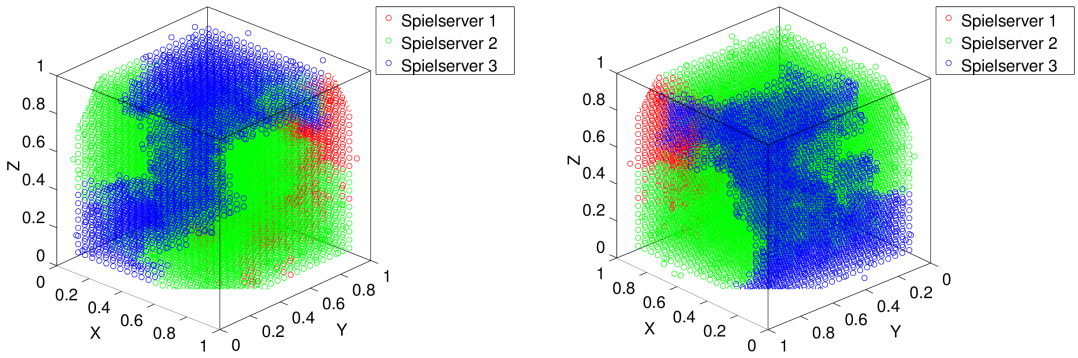


Abbildung 7.4.: Aufteilung der Spieler beim graphbasierten Verfahren bei gleichmäßiger Verteilung

So sind alle Spielserver in einer zusammenhängenden Gruppe. Insgesamt gibt es nur vereinzelt Ausreißer, wie z.B. unterhalb des Bereiches von Spielserver 1 (rot). Der größte Bereich wird von Spielserver 2 (grün) eingenommen, gefolgt von Spielserver 3 (blau). Der Bereich von Spielserver 1 ist deutlich am kleinsten.

Auch beim Heat Map Verfahren ist eine deutliche Gruppierung der Spieler erkennbar (Abb. 7.5). In diesem Fall sind die Bereiche der verschiedenen Spielserver ähnlich groß. Der Bereich von Spielserver 1 (rot) ist hier nicht zusammenhängend. So hat Spielserver 1 zwei relativ nah beieinander liegende Bereiche, zwischen denen ein schmaler Bereich liegt, der vor allem Spielserver 2 zugeordnet ist. In diesem wurden allerdings auch vereinzelt Spieler dem Spielserver 1 zugeordnet. Dies muss jedoch nicht aufgrund einer negativen Eigenschaft des Verfahrens

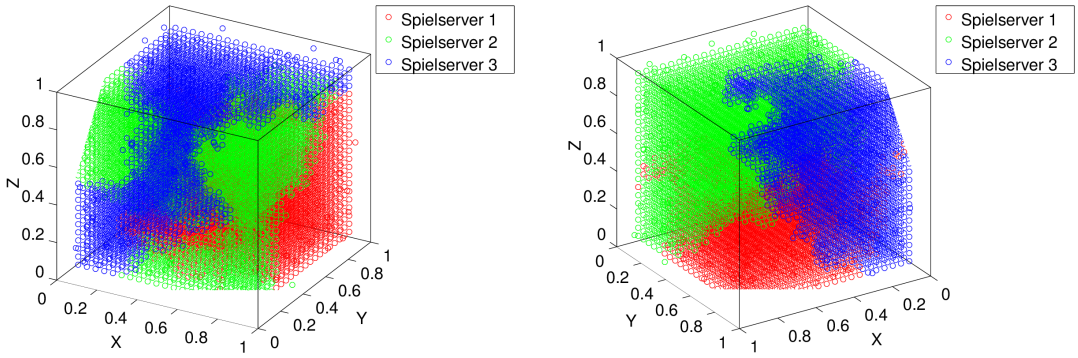


Abbildung 7.5.: Aufteilung der Spieler beim Heat Map Verfahren bei gleichmäßiger Verteilung

zustande gekommen sein, sondern kann auch damit zusammenhängen, in welcher Reihenfolge die Spieler bearbeitet wurden. Auch hier sind nur wenige Ausreißer zu sehen. Die einzigen richtigen Ausreißer stellen die zwischen den beiden Bereichen von Spielserver 1 dar.

Insgesamt lässt sich feststellen, dass bei einer gleichmäßigen Verteilung, alle entwickelten Verfahren in der Lage sind, die Spieler zu gruppieren. Dabei gelingt es dem Heat Map Verfahren besser eine gleichmäßige Aufteilung der Spielwelt zu erreichen, wohingegen beim graphbasierten Verfahren Spielserver 1 deutlich weniger Spieler zugewiesen bekommen hat.

7.2.3. Bei gedrängter Verteilung

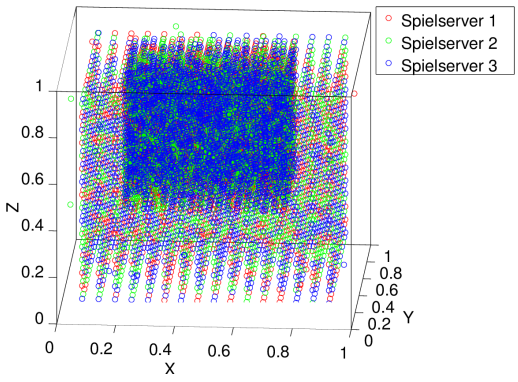


Abbildung 7.6.: Aufteilung der Spieler beim Round Robin Verfahren bei gedrängter Verteilung

Auch bei einer gedrängten Verteilung ist bei einem Round Robin Verfahren keine Struktur, in der Verteilung der Spieler, zu erkennen (Abb. 7.6). Auch in diesem Fall kommt das dadurch, dass die Spieler abwechselnd über die Spielserver verteilt werden und dies in einer zufälligen Reihenfolge geschieht.

Beim graphbasierten Verfahren ist wiederum eine Gruppierung der Spieler erkennbar (Abb.

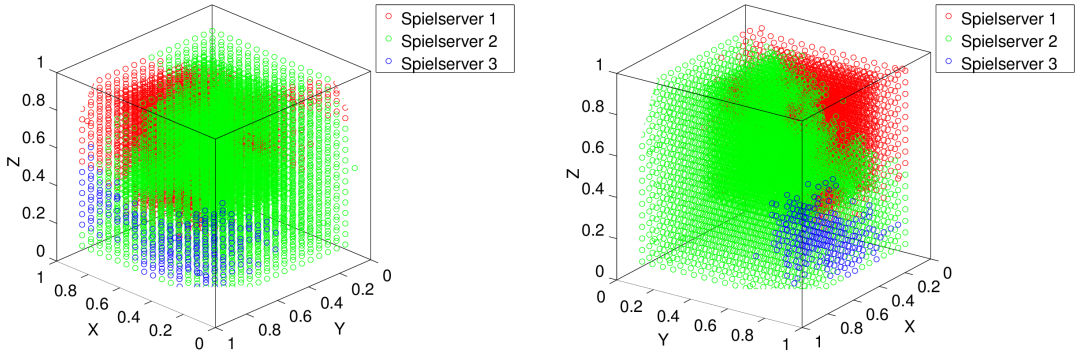


Abbildung 7.7.: Aufteilung der Spieler beim graphbasierten Verfahren bei gedrängter Verteilung

7.7). Auffällig ist, dass der Bereich von Spielserver 2 (grün) deutlich größer ist als die anderen beiden Bereiche. Der Bereich von Spielserver 1 (rot) sowie Spielserver 3 (blau) sind etwa gleich groß. Bei einer gedrängten Verteilung ist die Quote der Ausreißer deutlich höher als bei einer gleichmäßigen Verteilung. So hat Spielserver 1 neben einem größeren zusammenhängenden Bereich, einen kleineren mit Streuung im Bereich X = 0,1, Y = 0,1 und Z = 0,9. Auch ist der Bereich der Überschneidung zwischen den Spielservern jeweils größer als bei einer gleichmäßigen Verteilung.

Auch beim Heat Map Verfahren ist eine Grupierung der Spieler deutlich zu erkennen (Abb. 7.8). Auch wenn der Bereich von Spielserver 2 (grün) auf den Abbildungen leicht größer wirkt, sind die Bereiche von Spielserver 2 sowie Spielserver 1 (rot) etwa gleich groß. Der Bereich von Spielserver 3 (blau) ist deutlich am kleinsten. Die sichtbaren Bereiche von Spielserver 1 sind hinter dem gedrängten Bereich miteinander verbunden. Zwar sind keine deutlichen Ausreißer vorhanden, allerdings ist der Überschneidungsbereich, in denen die Spielserver ineinander übergehen, deutlich größer als bei der gleichmäßigen Verteilung.

Auch bei einer gedrängten Verteilung sind die entwickelten Verfahren in der Lage, die Spieler

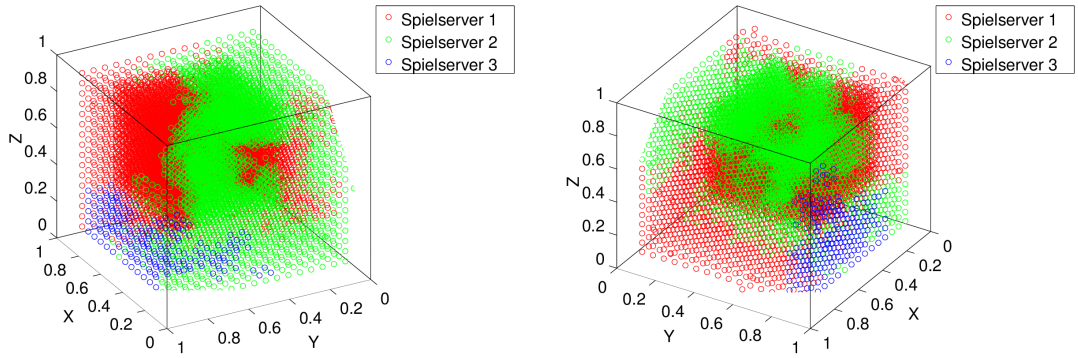


Abbildung 7.8.: Aufteilung der Spieler beim Heat Map Verfahren bei gedrängter Verteilung

zu gruppieren. Im Vergleich zu einer gleichmäßigen Verteilung, steigt jedoch die Menge der Ausreißer und die Übergänge zwischen den Servern sind meist größer. Auffällig ist, dass dies vor allem außerhalb des gedrängten Bereichs der Fall ist. Daraus lässt sich schließen, dass dies auch mit der dünneren Belegung in diesen Bereichen zusammen hängt. Zudem neigen alle Verfahren bei einer gedrängten Verteilung dazu, Spielserver, die den gedrängten Bereich dominieren, stärker zu belegen als die anderen. In diesem Szenario kann keiner der Verfahren heraus stechen. Stattdessen schneiden beide Verfahren bei der Menge der Ausreißer sowie der gleichmäßigen Aufteilung auf die Spielserver qualitativ gleichwertig ab.

7.2.4. Bei gruppierter Verteilung

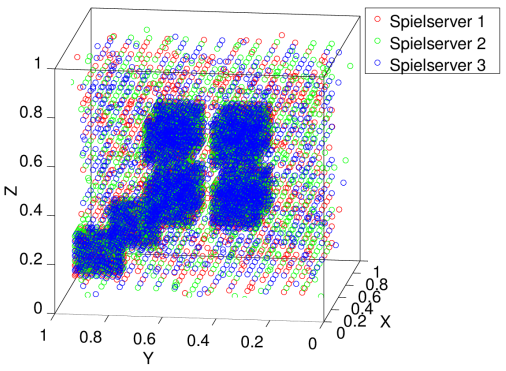


Abbildung 7.9.: Aufteilung der Spieler beim Round Robin Verfahren bei gruppierten Spielern

Auch bei gruppierten Spielern findet bei einem Round Robin Verfahren keine Gruppierung der Spieler statt (Abb. 7.9). Das liegt auch in diesem Fall daran, dass die zufällig eintreffenden Spieler hintereinander abwechselnd den Spielservern zugewiesen werden.

Beim graphbasierten Verfahren ist wieder eine klare Gruppierung zu erkennen (Abb. 7.10). So

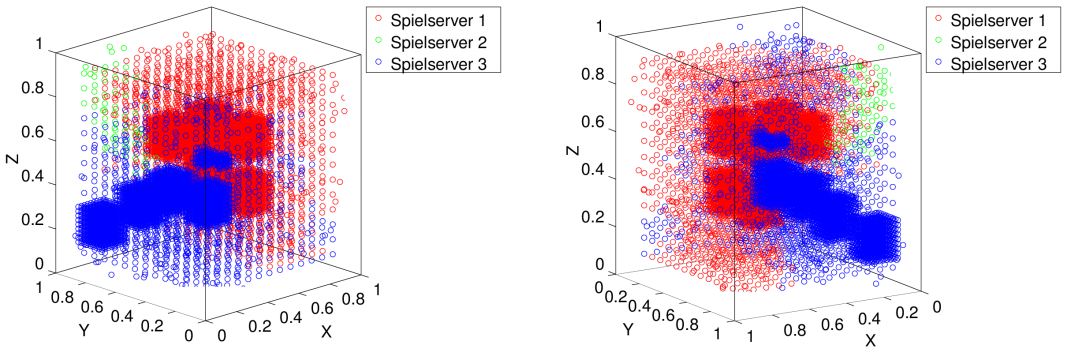


Abbildung 7.10.: Aufteilung der Spieler beim graphbasierten Verfahren bei gruppierten Spielern

werden vier der Gruppen vollständig von Spielserver 3 abgedeckt (blau) und fünf Gruppen vollständig von Spielserver 1 (rot). Eine Gruppe wird von Spielserver 1 sowie Spielserver 3 verwaltet, wobei Spielserver 1 dort dominiert. Ansonsten sind die Bereiche von Spielserver 1 und Spielserver 3 in etwa gleich groß. Nur ein verhältnismäßig kleiner Bereich wurde Spielserver 2 (grün) zugeordnet. Die Quote der Ausreißer liegt etwa bei dem, der gedrängt verteilten Spieler. Genauso ist der Übergang zwischen den Spielservern etwa gleich groß.

Auch beim Heat Map Verfahren ist eine Gruppierung deutlich zu erkennen (Abb. 7.11). Sieben Gruppen werden komplett von Spielserver 2 (grün) verwaltet während die anderen drei vollständig von Spielerver 1 (rot) verwaltet werden. Spielserver 2 ist hier deutlich am dominantesten, gefolgt von Spielserver 1, der etwa $\frac{1}{4}$ der Spielwelt verwaltet. Spielserver 3 ist deutlich am kleinsten. Auch hier ist die Quote der Ausreißer sowie der Übergang zwischen den Spielservern in etwa, mit denen der gedrängten Verteilung, vergleichbar.

Auch bei einer gruppierten Verteilung der Spieler sind all unsere Verfahren in der Lage, die Spieler zu gruppieren. Jedoch neigen die Verfahren bei einer gruppierten Verteilung dazu, einem Spielserver deutlich weniger Spieler zuzuordnen. Dabei werden vor allem diejenigen bevorzugt, die bereits einige Gruppen verwalten.

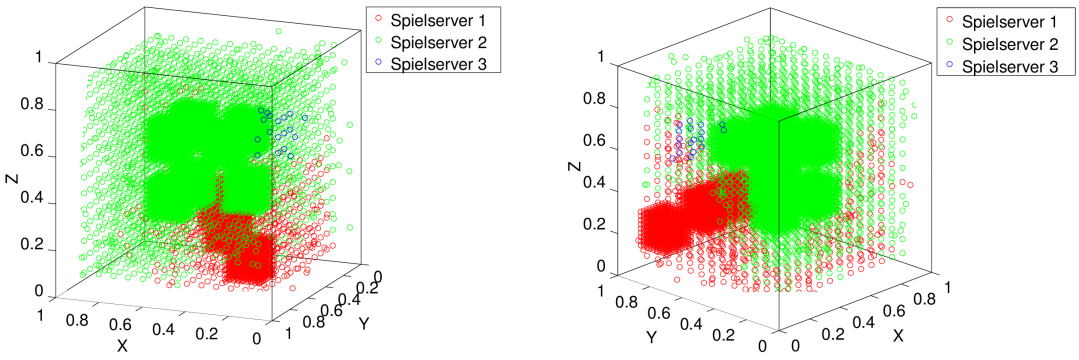


Abbildung 7.11.: Aufteilung der Spieler beim Heat Map Verfahren bei gruppierten Spielern

7.2.5. Bewertung

Alle entwickelten Verfahren sind in der Lage, die Spieler zu gruppieren. Dies gilt bei einer gleichmäßigen, gedrängten sowie gruppierten Verteilung. Dabei leidet die Gruppierung, wenn die Spieler nicht mehr gleichmäßig verteilt sind. Auch neigen alle Verfahren dazu, dass einzelnen Spielservern deutlich weniger Spieler zugewiesen werden. Dies gilt vor allem, wenn die Spieler gedrängt oder gruppiert verteilt sind. Qualitativ schneidet das Heat Map Verfahren bei einer gleichmäßigen Verteilung ein wenig besser ab, wenn es um die gleichmäßige Aufteilung der Spieler geht. Bei allen anderen Szenarien erzielen alle entwickelten Verfahren vergleichbare Ergebnisse.

7.3. Speicherauslastung

Da durch die Gruppierung weniger Überschneidungen, zwischen den View Boxen der Spielservern, vorliegen, kann davon ausgegangen werden, dass weniger Objekte auf mehrere Spielserver gleichzeitig geladen werden müssen. Der Test der Speicherauslastung soll überprüfen, ob diese positive Auswirkung auf die Speicherauslastung auch tatsächlich stattfindet. Zudem werden die Speicherauslastungen der einzelnen Spielserver betrachtet.

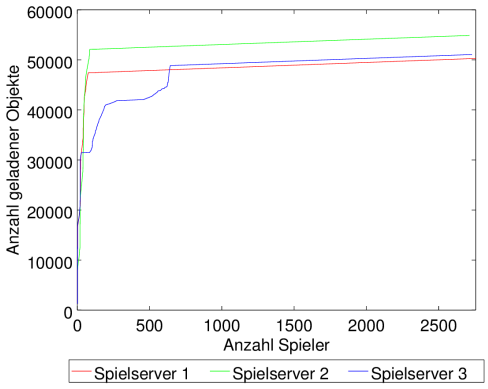
7.3.1. Metriken

Zum Messen der Speicherauslastung, wurden die Anzahl der im Speicher befindlichen Objekte genommen. Dazu wurden die einzelnen Verfahren betrachtet und diese untereinander verglichen. Beim Betrachten der einzelnen Verfahren, wurde verglichen, wie viele Objekte sich auf jedem Server, pro angemeldetem Spieler, befinden. Zum Vergleichen der Verfahren unterein-

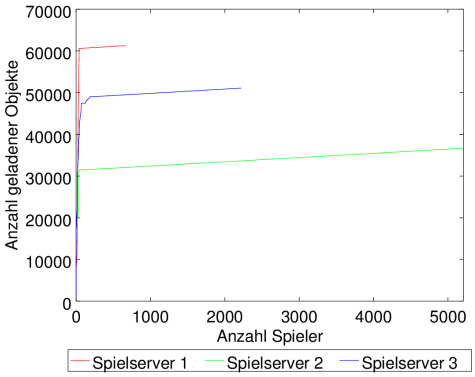
ander, wurde betrachtet, wie viele Objekte bei wie vielen angemeldeten Spielern insgesamt geladen wurden. Dazu wurden die auf jedem Server befindlichen Objekte aufsummiert. Dadurch wurden Objekte mehrfach gezählt, wenn diese sich auf mehreren Spielservern befanden.

Während der Messung wurde die Anzahl der Spieler inkrementiert, wenn die Daten eines Spieler erfolgreich vom QuP-System bezogen wurden. Da nicht alle Spielerdaten vor Ablauf der Tests bezogen werden konnten, ist die Summe der Spieler in den folgenden Diagrammen und Tabellen kleiner als 10.000.

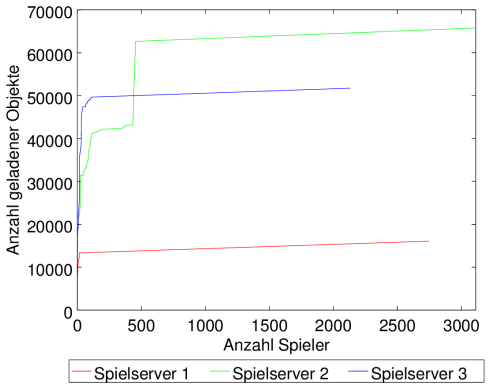
7.3.2. Bei gleichmäßiger Verteilung



(a) Round Robin



(b) Graphbasiertes Verfahren



(c) Heat Map

Abbildung 7.12.: Anzahl Objekte pro Spieler bei gleichmäßiger Verteilung

Wird das Round Robin Verfahren betrachtet, so haben alle Spielserver eine ähnliche Anzahl an Objekten in ihrem Speicher (Abb. 7.12a). Diese liegen, nachdem alle Spieler ihren Spielser-

Tabelle 7.1.: Anzahl geladener Objekte bei gleichmäßiger Verteilung

	Round Robin		Graphbasiert		Heat Map	
	Spieler	**Objekte**	**Spieler**	**Objekte**	**Spieler**	**Objekte**
Spielserver 1	2756	50282	665	61269	2745	16071
Spielserver 2	2712	54887	5207	36666	3111	65775
Spielserver 3	2729	51070	2216	51080	2127	51731
$\sum$	8197	156239	8088	149015	7983	133577

vern zugeordnet wurden, zwischen 50.282 bei Spielserver 1 und 54.887 bei Spielserver 2 (Tabelle 7.1). Da beim Round Robin die Spieler abwechselnd über die Spielserver verteilt werden, ist auch die Anzahl der Spieler bei allen Spielservern gegen Ende ähnlich. Bei allen in der Arbeit entwickelten Verfahren ist die Anzahl der Objekte pro Spielserver sehr unterschiedlich, genau wie die Anzahl der Spieler. Beim graphbasierten Verfahren sticht der Spielserver 1 heraus, welcher trotz der geringsten Anzahl an Spielern, die meisten Obekte im Speicher hat (Abb. 7.12b und Tabelle 7.1). Zudem hat der Spielserver mit den meisten Spielern (Spielserver 2), die geringste Anzahl an Objekten im Speicher. Bei der Heat Map sind ähnliche Verhalten zu erkennen (Abb. 7.12c). So hat Spielserver 1 die zweitmeisten Spieler, hat jedoch die wenigsten Objekte im Speicher (16.071). Sogar Spielserver 3, welcher die wenigsten Spieler verwaltet, hat mit 51.731 Objekten mehr als drei mal so viele Objekte im Speicher (Tabelle 7.1). Werden die Graphen betrachtet, so fällt auf, dass bei allen Verfahren ein Großteil der Objekte bereits mit unter 500 Spielern pro Server geladen worden (Abb. 7.12). Ab 1.000 Spieler gibt es bei keinem der Verfahren mehr einen signifikanten Anstieg der Anzahl an geladenen Objekte.

Das auch bei Spielservern mit geringeren Spielerzahlen deutlich mehr Objekte geladen werden, als bei Spielservern mit höheren Spielerzahlen, lässt sich auf den Octree des QuP-Systems zurückführen. Durch die Struktur kann es vorkommen, dass bei gleich großen View Boxen, unterschiedlich große Bereiche geladen werden. Das hängt damit zusammen, dass der niedrigste Knoten im Baum gewählt wird, der die gesamte View Box des Spielers abdeckt. Ist die View Box nun größer als der Bereich eines Knotens, so wird der Knoten gewählt, der oberhalb aller Knoten liegt, in denen die View Box hinein ragt. Da dieser auch oberhalb von Knoten liegen kann, in denen die View Box nicht hinein ragt, werden dadurch auch Objekte geladen, die außerhalb der Felder liegen, in denen die View Box liegt. Befindet sich der Spieler an einer unvorteilhaften Position, kann der Bereich sehr groß werden.

Werden alle Verfahren miteinander verglichen (Abb. 7.13 und Tabelle 7.1), so wurde beim

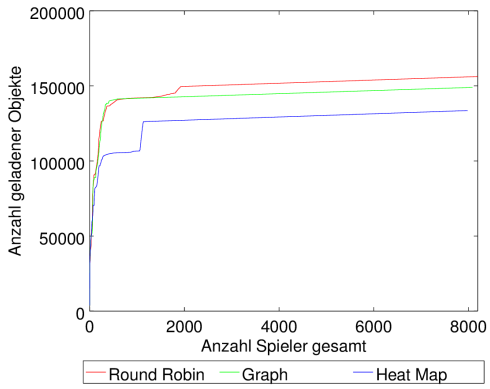


Abbildung 7.13.: Vergleich der Speicherauslastung, der einzelnen Verfahren, bei gleichmäßiger Verteilung

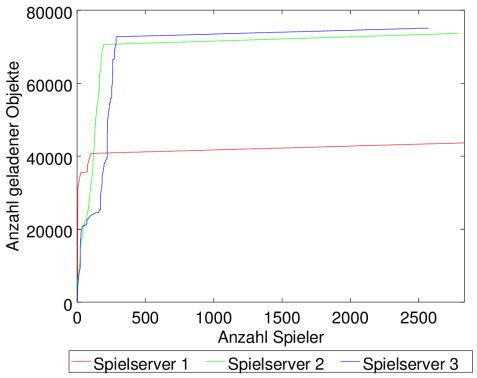
Round Robin Verfahren, mit 156.239 Objekten, die meisten Objekte in den Speicher geladen. Das graphbasierten Verfahren liegt mit 149.015 geladenen Objekten dahinter. Die wenigsten Objekte wurden beim Heat Map Verfahren geladen (133.577). Die entwickelten Verfahren haben somit in diesem Fall tatsächlich weniger Objekte in den Speicher geladen, wenngleich die Zahl nur geringfügig geringer ist.

7.3.3. Bei gedrängter Verteilung

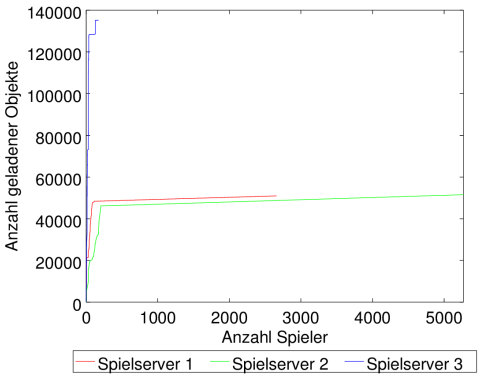
Tabelle 7.2.: Anzahl geladener Objekte bei gedrängter Verteilung

	Round Robin		**Graphbasiert**		**Heat Map**	
	Spieler	**Objekte**	**Spieler**	**Objekte**	**Spieler**	**Objekte**
Spielserver 1	2834	43644	2658	50957	5323	45711
Spielserver 2	2785	73646	5269	51528	3484	54459
Spielserver 3	2569	75102	169	135234	147	79569
$\sum$	8188	192392	8096	237719	8954	179739

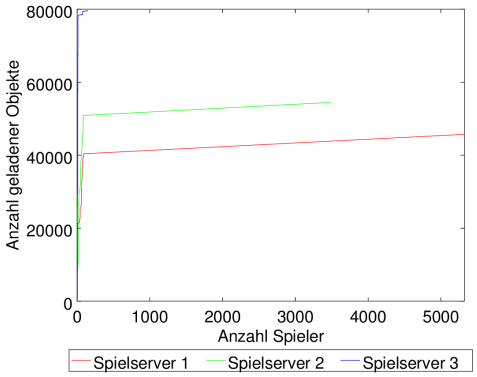
Bei einer gedrängten Verteilung der Spieler, lassen sich, mit Ausnahme vom Round Robin Verfahren, ähnliche Verhalten, wie bei einer gleichmäßigen Verteilung der Spieler feststellen. So haben beim Round Robin Verfahren, im Vergleich zur gleichmäßigen Verteilung, nicht alle Spielserver eine ähnliche Anzahl von Objekten im Speicher, da in diesem Fall Spielserver 1, mit 43.644, deutlich weniger Objekte im Speicher hat als Spielserver 2 (73.646) und Spielserver 3 (75.102) (Tabelle 7.2). Bei den entwickelten Verfahren gibt es, ähnlich wie bei der gleichmäßigen Verteilung, jeweils einen Spielserver mit sehr wenigen Spielern, der jedoch sehr viele Objekte



(a) Round Robin



(b) Graphbasiertes Verfahren



(c) Heat Map

Abbildung 7.14.: Anzahl Objekte pro Spieler bei gedrängter Verteilung

im Speicher hat. So hat beim graphbasieren Verfahren, Spielserver 3 lediglich 169 Spieler, hat aber mit 135.234 mehr als das doppelte an Objekten geladen als Spielserver 1 (2.658 Spieler und 50.957 Objekte) und Spielserver 2 (5.269 Spieler und 51.528 Objekte) (Abb. 7.14b und Tabelle 7.2). Ähnliches gilt auch für das Heat Map Verfahren. Hier hat Spielserver 3, mit 147, die deutlich geringste Anzahl an Spielern, jedoch mit 79.569 auch die deutlich meisten Objekte im Speicher. Im Gegensatz dazu, verwaltet Spielserver 1 mehr als die Hälfte aller Spieler (5.323) und hat die wenigsten Objekte geladen (45.711) (Abb. 7.14c und Tabelle 7.2). Wie bei der gleichmäßigen Verteilung der Spieler, lässt sich auch in diesem Fall, das Verhalten über die Struktur des Octrees erklären. Durch diese ist es möglich, dass für eine View Box, der Bereich der geladenen Objekte, signifikant größer ist als bei einer anderen View Box, obwohl diese gleich groß sind. Auch bei einer gedrängten Verteilung lässt sich feststellen, dass bei allen Verfahren ein Großteil der Objekte mit bereits weniger als 500 Spielern pro Server geladen worden (Abb. 7.14). Im

Gegensatz zur gleichmäßigen Verteilung, weist hier schon ab 500 Spielern keiner der Verfahren mehr einen signifikanten Anstieg an geladenen Objekten auf.

Werden die einzelnen Verfahren miteinander verglichen, so fällt auf, dass das graphbasierte

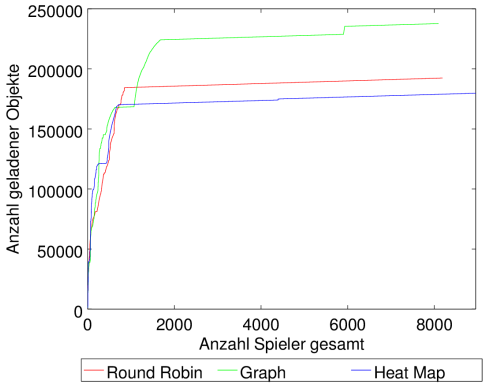
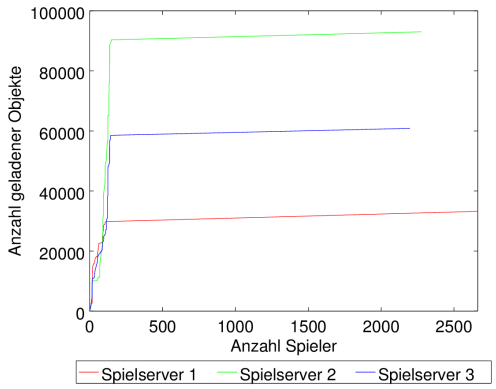


Abbildung 7.15.: Vergleich der Speicherauslastung, der einzelnen Verfahren, bei gedrängter Verteilung

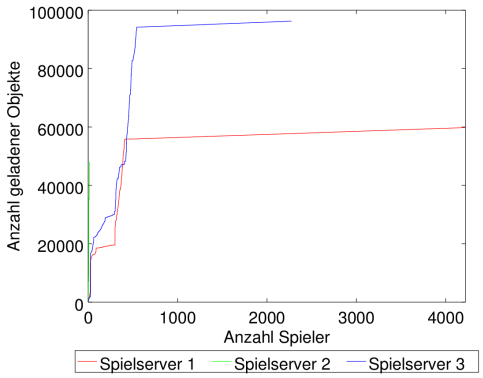
Verfahren mehr Objekte geladen hat als das Round Robin Verfahren (Abb. 7.15 und Tabelle 7.2). Dies ist überraschend, da durch die Gruppierung der Spieler die Anzahl der Überschneidung von View Boxen, die sich auf verschiedenen Spielservern befinden, verringert wurde. Wie jedoch zuvor gezeigt wurde, hängt der durch die View Boxen abgedeckte Bereich nicht zwingend mit der Größe des Bereiches zusammen, der geladen wird. Somit kann auch dies über das Verhalten des Octrees erklärt werden. Ist nämlich die Lage der Grenze zwischen zwei Spielservern unglücklich, so kann es aufgrund der Auswahl des niedrigeren Knotens passieren, dass ein Spielserver auch große Teile der Bereiche laden muss, in denen die Spieler einem anderen Spielserver zugewiesen sind. In diesem Fall hat das graphbasierte Verfahren 237.719 Objekte geladen, während beim Round Robin Verfahren 192.392 Objekte geladen wurden. Nur das Heat Map Verfahren hat, mit 179.739 Objekten, weniger Objekte als das Round Robin Verfahren geladen.

7.3.4. Bei gruppierter Verteilung

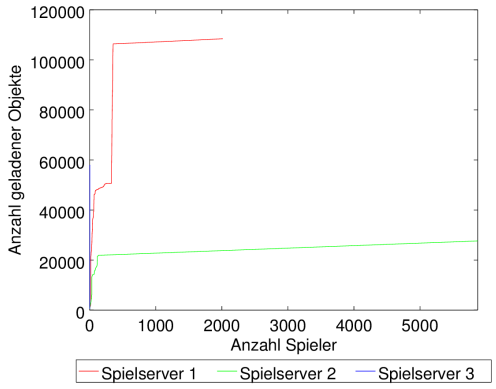
Sind die Spieler in Gruppen angeordnet, so ist die Menge der geladenen Objekte beim Round Robin Verfahren bei keinen der Spielserver mehr ähnlich (Abb. 7.16a). Spielserver 2 hat hier mit 93.012 geladenen Objekten, die meisten Objekte im Speicher, Spielserver 3 mit 60.862 Objekten die zweitmeisten sowie Spielserver 1 mit 33.227 Objekten die wenigsten (Tabelle 7.3). Bei den



(a) Round Robin



(b) Graphbasiertes Verfahren



(c) Heat Map

Abbildung 7.16.: Anzahl Objekte pro Spieler bei gruppierten Spielern

entwickelten Verfahren fällt auf, dass bei allen ein Spielserver dabei ist, dem nur sehr wenige Spieler zugeordnet sind. So sind beim graphbasierten Verfahren Spielserver 2 nur 16 Spieler und beim Heat Map Verfahren Spielserver 3 nur 4 Spieler zugeordnet (Abb. 7.16b und 7.16c sowie Tabelle 7.3). Beim graphbasierten Verfahren hat Spielserver 2 damit zwar auch die wenigsten Objekte geladen (47.925), dies ist jedoch im Verhältnis zur Spielerzahl nur geringfügig weniger als bei den anderen Spielservern. So hat Spielserver 1 mit 59.796 die zweitmeisten Objekte geladen und verwaltet 4.219 Spieler. Beim Heat Map Verfahren hat Spielserver 3 57.905 Objekte geladen. Spielserver 2 verwaltet mit 5.872 Spielern, die meisten Spieler und hat mit lediglich 27.641, die wenigsten Objekte geladen. Die meisten hat Spielserver 1, mit 108.429 Objekten, geladen (Tabelle 7.3). Ähnlich, wie bei der gleichmäßigen sowie der gedrängten Verteilung sind auch hier die meisten Objekte bereits mit 500 Spielern pro Server geladen worden (Abb. 7.16). Ab 1.000 Spielern ist auch hier bei keinem der Verfahren mehr ein signifikanter Anstieg

Tabelle 7.3.: Anzahl geladener Objekte bei gruppierten Spielern

	Round Robin		Graphbasiert		Heat Map	
	Spieler	**Objekte**	**Spieler**	**Objekte**	**Spieler**	**Objekte**
Spielserver 1	2662	33227	4219	59796	2015	108429
Spielserver 2	2276	93012	16	47925	5872	27641
Spielserver 3	2196	60862	2271	96256	4	57905
$\sum$	7134	187101	6506	203977	7891	193975

an geladenen Objekten zu verzeichnen.

Werden die Verfahren miteinander verglichen, so haben hier beide entwickelten Verfah-

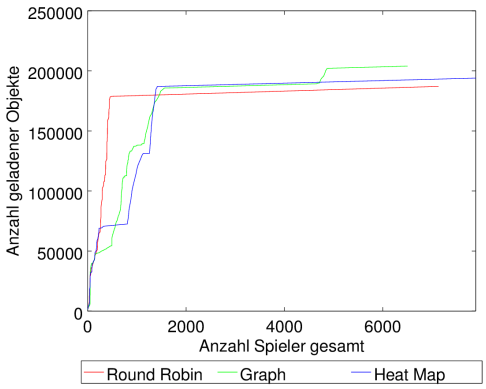


Abbildung 7.17.: Vergleich der Speicherauslastung, der einzelnen Verfahren, bei gruppierten Spielern

ren mehr Objekte geladen als das Round Robin Verfahren (Abb. 7.17 und Tabelle 7.3). So hat das graphbasierte Verfahren 203.977 und das Heat Map Verfahren 193.975 Objekte geladen. Beim Round Robin Verfahren wurden 187.101 Objekte geladen.

7.3.5. Bewertung

Die Gruppierung der Spieler hat keinerlei Einfluss auf die Anzahl der geladenen Objekte. Stattdessen lässt sich festhalten, dass einige Spielserver auch mit sehr wenigen Spielern, eine hohe Anzahl an geladenen Objekten erzielt haben. So gab es Spielserver, die mit weniger als 1.000 Spielern, ein vielfaches an Objekten geladen hatten als wie Spielserver mit über 5.000 Spielern. Dies verwundert auf den ersten Blick, da bei weniger Spielern ein geringerer Bereich durch View Boxen abgedeckt wird. Auch dass, durch die Gruppierung keinen positiven Effekt

auf die Speicherauslastung gibt, verwundert zunächst. Schließlich gibt es durch die Gruppierung der Spieler weniger Überschneidungen zwischen den View Boxen, der verschiedenen Spielserver. Somit könnte davon ausgegangen werden, dass weniger Objekte doppelt geladen werden müssen. Die Experimente haben jedoch gezeigt, dass dies nicht zutrifft. Dies lässt sich über die Struktur des Octrees innerhalb des QuP-Systems erklären. Durch diese kann der Bereich, aus denen Objekte geladen werden, um ein vielfaches größer sein als die View Box, da versucht wird, einen Knoten zu finden, der alle Bereiche der View Box abdeckt. Dadurch kann, bei einer unglücklichen Position, ein Knoten ermittelt werden, der deutlich größere Teile der Spielwelt abdeckt als die View Box. Somit kann es auch vorkommen, dass Spielserver Objekte aus Bereichen laden, in denen die Spieler durch einen anderen Spielserver verwaltet werden. Im schlimmsten Fall kann dies dazu führen, dass sämtliche Objekte der Spielwelt bezogen werden, wenn der Spieler sich genau in der Mitte der Spielwelt aufhält.

7.4. Lasttests

Da die Verfahren auch bei großen Umgebungen eingesetzt werden sollen, musste auch überprüft werden, wie sich die Verfahren bei großen Spielerzahlen verhalten. Dies wurde mithilfe von Lasttests ermittelt. Dazu wurde neben den absoluten Zahlen auch das Verhalten der Verfahren betrachtet. Aus dem theoretisch ermitteltem Laufzeitverhalten ist zu erwarten, dass das graphbasierte Verfahren eine linear steigende Laufzeit aufweist. Für das Heat Map Verfahren wurde eine lineare Abhängigkeit zu der Anzahl der Felder ermittelt. Da die Anzahl der Felder jedoch konstant ist, kann von einer konstanten Laufzeit gesprochen werden. Bei der Implementierung ist eine Map verwendet wurden. Hierbei handelt es sich um eine, von Erlang bereitgestellte, persistente Datenstruktur. Zudem werden nur Felder abgelegt, die auch belegt sind. Deshalb müssen dynamisch Felder hinzugefügt und entfernt werden. Aufgrund dessen kann mit einer logarithmischen Laufzeit in Abhängigkeit zur Anzahl der belegten Felder gerechnet werden. D.h. dass die Laufzeit zunächst logarithmisch steigt, bis sämtliche Felder der Spielwelt belegt sind. Ist dies der Fall, bleibt die Laufzeit konstant. Beim Round Robin Verfahren, als Vergleichsverfahren, kann mit einer konstanten Laufzeit gerechnet werden, die deutlich unterhalb der anderen Verfahren liegen, da der Rechenaufwand nur sehr gering ist.

7.4.1. Metriken

Um zu überprüfen, wie sich die Verfahren bei hoher Last verhalten, wurde für jede Zuweisung die benötigte Zeit in μs gemessen. Diese wurde in Verhältnis zu den angemeldeten Spielern gesetzt. Daraus lässt sich ablesen, wie sich die Verfahren bei steigender Spielerzahl Verhalten.

7.4.2. Bei gleichmäßiger Verteilung

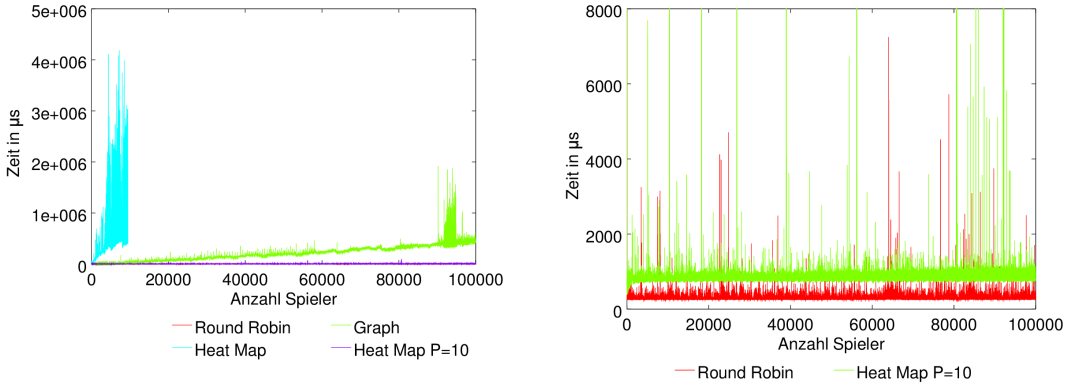


Abbildung 7.18.: Dauer der Ermittlung eines Spielservers bei gleichmäßiger Verteilung

Bei einer gleichmäßigen Verteilung verhalten sich die Verfahren wie angenommen. So weist das Round Robin Verfahren eine Zuweisungszeit auf, die konstant um die 300 μs liegt. Die Laufzeit des graphbasierten Verfahrens steigt mit zunehmender Spielerzahl gleichmäßig an. So dauert die Zuweisung zu Beginn um die 500 μs und steigt gegen Ende auf zwischen 400.000 und 500.000 μs. Auffällig ist, dass die Ausreißerquote zwischen 90.000 und 100.000 Spieler kurzfristig deutlich steigt, mit Ausschlägen bis zu 1923336 μs. Das Heat Map Verfahren zeigt wie erwartet zunächst eine logarithmische Steigung. Auffällig ist, dass die Rate der Ausschläge durchgehend sehr hoch ist. So liegt die Zuweisungszeit beim Heat Map Verfahren, bei um die 500.000 μs, alle sieben bis acht Spieler steigt dieser Wert jedoch auf über eine Sekunde, in Extremfällen sogar über zwei oder gar drei Sekunden. Da die Heat Map sehr viel Speicherplatz benötigt und dieser auf dem Testsystem ausgegangen ist, war das Verfahren bei einer Präzision von 100 nicht in der Lage alle 100.000 Spieler zuzuweisen, sondern brach bei 9.490 Spielern ab. Deshalb wurde der Test auch mit einer kleineren Heat Map durchgeführt. Diese hatt eine Präzision von 10. Es ist deutlich zu sehen, dass nach kurzzeitig logarithmischen Anstieg, die Laufzeit des Verfahrens konstant bleibt. Diese ist mit zwischen 750 und 800 μs deutlich kleiner als bei einer Präzision von 100 oder das graphbasierte Verfahren. Selbst die wenigen Ausreißer sind mit maximal 14.398 μs um ein vielfaches geringer.

7.4.3. Bei gedrängter Verteilung

Die Laufzeiten bei einer gedrängten Verteilung sind vergleichbar mit denen der gleichmäßigen Verteilung. Wie zu erwarten ist die benötigte Zeit, zur Ermittlung beim Round Robin Verfahren, wie zuvor durchgehend bei ca. 300 μs. Auch die Zeit des graphbasierten Verfahrens steigt

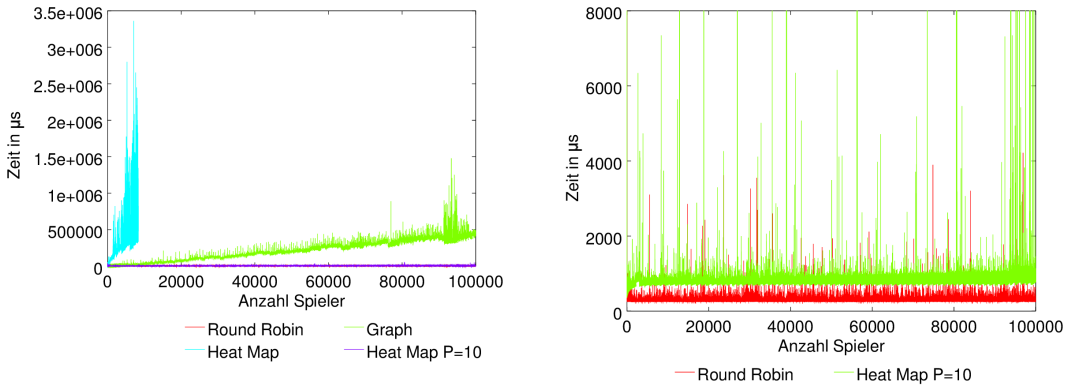


Abbildung 7.19.: Dauer der Ermittlung eines Spielservers bei gedrängter Verteilung

weiterhin linear von wie zuvor ca. 500 μs auf zwischen 400.000 und 500.000 μs. Auch in diesem Fall steigt die Quote der Ausreißer zwischen 90.000 und 100.000 Spielern zwischenzeitlich deutlich. Das Verhalten des Heat Map Verfahrens ist auch mit dem der gleichmäßigen Verteilung vergleichbar. Auch hier steigt die Laufzeit zunächst logarithmisch, bis das Verfahren bei 8.390 Spielern vorzeitig abbricht. Wird das Experiment mit einer Präzision von 10 durchgeführt, so ist auch hier zu sehen, dass das Heat Map Verfahren nach einem anfänglich logarithmischem Verhalten zu einer konstanten Zuweisungszeit neigt, die bei um die 800 μs liegt.

7.4.4. Bei gruppierter Verteilung

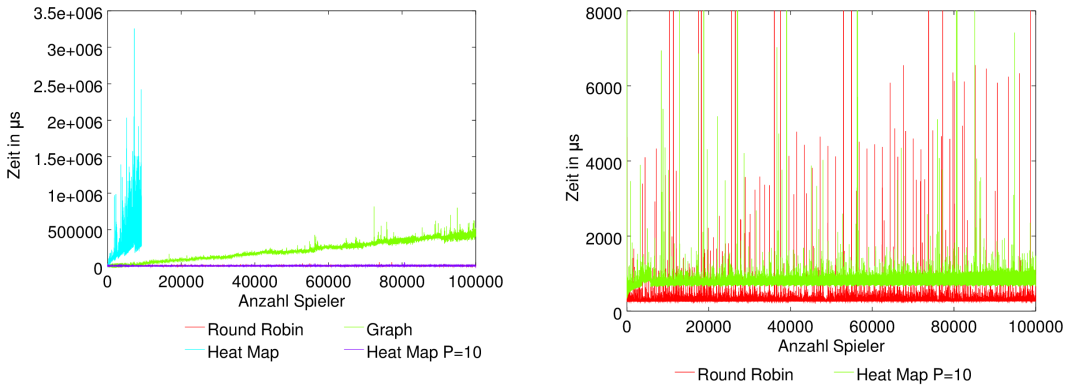


Abbildung 7.20.: Dauer der Ermittlung eines Spielservers bei gruppierten Spielern

Auch bei einer Gruppierung der Spieler, sind die Ergebnisse ähnlich zu den vorherigen Versuchen. So ist auch in diesem Fall die Zuweisungszeit des Round Robin Verfahrens konstant

bei ca. 300 μs. Auch das graphbasierte Verfahren verhält sich ähnlich wie zuvor. Die Zeit steigt auch hier linear von ca. 500 μs auf zwischen 400.000 und 500.000 μs. In diesem treten allerdings keine größeren Ausreißer, in dem Bereich 90.000 bis 100.000 Spieler, auf. Auch das Heat Map Verfahren weist ein ähnliches Verhalten auf und bricht auch in diesem Fall vorzeitig nach 9.230 Spielern ab. Bei einer Präzision von 10, lässt sich jedoch auch hier deutlich sehen, dass nach einer anfänglich logarithmischen Steigung, die Laufzeit konstant bei um die 700 μs bleibt.

7.4.5. Bewertung

Alle entwickelten Verfahren weisen das erwartete Verhalten auf. Das Zeitverhalten ist dabei bei allen Verfahren unabhängig von der Verteilung der Spieler. So konnte nachgewiesen werden, dass das graphbasierte Verfahren auch bei einer hohen Spielerzahl in der Lage ist, effizient zu arbeiten. Das Heat Map Verfahren weist auf unserem Testsystem bei einer Präzision von 100 sehr starke Ausreißer auf, die auch mehrere Sekunden betragen können. Dies macht das Verfahren in dieser Konfiguration praktisch nicht nutzbar. Hinzu kommt, dass die Heat Map mit dieser Konfiguration sehr speicherintensiv ist. So lassen sich, auf dem eingesetzten Testsystem, keine 100.000 Spieler auf die Spielserver zuweisen, weil der Speicher zuvor überfüllt ist. Wird die Präzision jedoch auf 10 gesenkt, kann mit der Heat Map deutlich die beste Zeit erzielt werden, die nur knapp das doppelte eines Round Robin Verfahrens beträgt. Zudem bleibt die Zeit konstant, sobald sämtliche Felder belegt sind. Die Effizienz der Heat Map ist somit stark von Anzahl der Felder, insbesondere der belegten, abhängig. Durch eine geringere Anzahl an Feldern, kann die Effizienz somit enorm gesteigert werden.

7.5. Fazit

Alle entwickelten Verfahren eignen sich für das Load Balancing von Timadorus. So war es mit allen Verfahren möglich, die Spieler zu gruppieren. Hierbei zeigte sich jedoch bei allen Verfahren dieselbe Schwachstelle. Sie neigen alle dazu, dass einzelnen Spielservern deutlich weniger Spieler zugewiesen werden, vor allem, wenn die Verteilung der Spieler nicht gleichmäßig ist. Die Analyse der Speicherauslastung hat gezeigt, dass durch die Gruppierung, die Belastung des Speichers der Spielserver nicht verringert werden konnte. Dies hängt vor allem mit der Struktur des Octrees zusammen. Dennoch kann die Gruppierung als Vorteil betrachtet werden. Denn auch wenn mehr Objekte in den Speicher geladen werden, beschränkt sich der Zuständigkeitsbereich der Spielserver auf den Inhalt der View Boxen. Somit können durch die geringere Anzahl an Überschneidungen der View Boxen, die Fälle verringert werden, in denen Objekte durch mehrere Spielserver manipuliert werden. Dadurch verringert sich

auch das Risiko von Kollisionen beim Abspeichern auf dem QuP-System. Durch den Lasttest konnte zudem gezeigt werden, dass die Verfahren in der Lage sind, auch große Spielermengen zu bewältigen. Während die Laufzeit des graphbasierten Verfahrens linear wuchs, weist das Heat Map Verfahren eine logarithmische Laufzeit auf, die nach Belegung aller Felder konstant bleibt. Das Heat Map Verfahren hat jedoch bei einer hohen Präzision den Nachteil, dass der Speicherbedarf sehr hoch ist. Zudem steigt die Laufzeit zunächst stark an, sodass bei einer Präzision von 100 die Laufzeit, bei allen Szenarien, bereits bei ca. 9.000 Spielern bei 400.000 bis 500.000 μs liegt. Durch eine geringere Präzision lässt sich jedoch auch die Laufzeit deutlich verringern. Zudem kann damit auch der Speicher entlastet werden. So wurde bei allen Szenarien mit einer Präzision von 10 eine konstante Laufzeit von 700 bis 800 μs erreicht. Da 1.000 Felder für die meisten Spiele ausreichend sein sollte, kann dieses Verfahren als leistungsfähig betrachtet werden.

8. Abschluss

Zum Abschluss soll noch einmal betrachtet werden, was alles während dieser Arbeit gemacht wurde. Außerdem wird ein Fazit gezogen, welches die Ergebnisse der Arbeit zusammenfasst. Zum Schluss dieses Kapitels gibt es noch einen Ausblick darauf, wie mögliche weiterführende Arbeiten aussehen könnten. Das beinhaltet Elemente, die in dieser Arbeit nicht betrachtet werden konnten sowie mögliche Weiterentwicklungen auf Basis der in dieser Arbeit entwickelten Verfahren und Implementierung.

8.1. Zusammenfassung

In dieser Arbeit wurden zwei Verfahren entwickelt, welche dem initialen Load Balancing in einer verteilten virtuellen Umgebung dienen. Dabei handelt es sich um ein graphbasiertes Verfahren sowie ein Verfahren basierend auf einer Heat Map. Das graphbasierte Verfahren beruht auf einen gedachten Graphen, bei dem versucht wird, die Abstände zwischen den Spielservern zu maximieren. Da bei jeder Berechnung nur ein einzelner Spieler hinzugefügt wird, konnte dieses Verfahren so weit vereinfacht werden, dass der Graph nicht aufgebaut werden muss, sondern rein konzeptioneller Natur ist. Beim Heat Map Verfahren wird die Spielwelt in Felder unterteilt. Zur Berechnung eines Spielservers wird analysiert, wie viele Spieler, eines Spielservers, sich in den einzelnen Feldern befinden. Dazu werden zunächst die Felder des neuen Spielers ermittelt und daraufhin nach möglichst starken Überschneidungen mit Spielservern gesucht oder, falls es keine Überschneidungen geben sollte, nach Spielservern mit Feldern in der Nähe des Spielers.

Damit die Verfahren auch nutzbar sind, wurde ein Load Balancing System entwickelt und implementiert. Zunächst wurde die Architektur entwickelt, dabei wurde das System in zwei Hauptkomponenten getrennt. Dem Load Balancing Server, als zentrale Serverinstanz sowie dem Monitor, als Client, der in die Spielserver integriert wird, um Statusinformationen an den Load Balancing Server zu senden. Der Server wurde wiederum in vier Komponenten gegliedert. Dem Message Handler als Schnittstelle, um Anfragen an den Load Balancer stellen zu können, dem Data Storage, zum Verwalten der Serverzustände sowie dem Zustand der Spielwelt, dem

Server Locator zum ermitteln eines Spielservers und den Server Observern, welche die Zustände der Spielserver beobachten. Der Monitor hat die Aufgabe, den Zustand der Spielserver auf Seiten des Spielservers zu überwachen und an den Load Balancing Server zu übermitteln. Damit die einzelnen Systemkomponenten miteinander kommunizieren können, werden zusätzlich Schnittstellen bereitgestellt. Neben der Architektur und den Schnittstellen, wurden auch weitergehende Problemen behandelt. So werden Zeitstempel und Veränderungsraten eingesetzt, um zu starke Inkonsistenzen zwischen den von der Berechnung genutzten und den aktuell abgespeicherten Spielwelt- und Spielserverzuständen zu vermeiden. Außerdem wurde ein Vorgehen entwickelt, wie vorzugehen ist, wenn ein Spielserver keine Informationen mehr sendet. All das wurde in Erlang umgesetzt. Bei der Realisierung, wurde auch die Fragestellung behandelt, welche Datenstrukturen aus Erlang zum Implementieren unserer Verfahren am geeignetsten sind. Da bei der Heat Map mehrere infrage kommen, wurden dort mittels Tests, verschiedene Datenstrukturen verglichen und mithilfe der Ergebnisse eine ausgewählt.

Nachdem das System implementiert wurde, wurde die Leistungsfähigkeit der entwickelten Verfahren evaluiert. Dazu wurden mehrere Tests entwickelt und das Verhalten der Algorithmen bei verschiedenen Szenarien verglichen. Getestet wurden die Qualität der Gruppierung, der Verfahren, ob es Vorteile in Bezug auf die Speicherauslastung gibt sowie die Laufzeit in Abhängigkeit zu der Spielerzahl mittels eines Lasttests. Über die ermittelten Ergebnisse konnte analysiert werden, inwiefern die entwickelten Verfahren geeignet sind.

8.2. Fazit

Im Rahmen dieser Arbeit wurden zwei Load Balancing Verfahren sowie ein Load Balancing System im Kontext des Timadorus Projektes entwickelt und implementiert. Das System konnte vollständig in Erlang realisiert werden. Der Load Balancing Server ist in der Lage die Zustände der Spielwelt zu verwalten und Spielserver für neue Spieler zu ermitteln. Dazu wurden Schnittstellen realisiert über die Zustandsinformationen übermittelt werden können und nach Spielserver für Spieler angefragt werden kann. Spielserver melden sich durch das Einbinden des Monitors, automatisiert beim Load Balancing Server an, wenn sie starten. Durch den Monitor, senden diese zudem automatisiert Statusnachrichten an den Load Balancing Server. Auch konnte die Ermittlung eines Spielservers in den Authentifizierungsprozess integriert werden. Außerdem konnten beide entwickelten Verfahren implementiert und in das System integriert werden. Eine parallelisierte Fassung des Heat Map Verfahren wurde jedoch nicht realisiert.

Mittels der Experimente konnte die Leistungsfähigkeit der entwickelten Verfahren gezeigt werden. Zudem wurde damit auch die Funktionsfähigkeit des Systems nachgewiesen. So wurde gezeigt, dass die entwickelten Verfahren in der Lage sind, die Spieler geografisch auf die Spielserver zu gruppieren. Beide entwickelten Verfahren neigen jedoch dazu, dass einige Spielserver mit nur wenigen Spielern belegt werden. Obwohl die Gruppierung der Spieler erfolgreich war, konnte die Auslastung der Speicher der Spielserver nicht verringert werden. Dies hängt vor allem mit der Struktur des Octrees zusammen. Dennoch kann die Gruppierung als erfolgreich betrachtet werden, da somit die Wahrscheinlichkeit von Kollisionen bei der Manipulation von Objekten der Spielwelt, verringert werden konnte. Bei den Lasttests waren beide Verfahren in der Lage bei hohen Spielerzahlen bis 100.000 in unter 0,5 Sekunden einen Spielserver zu ermitteln. Beim graphbasierte Verfahren stieg dabei die Laufzeit linear. Beim Heat Map Verfahren stieg die Laufzeit zunächst logarithmisch, bis sämtliche Felder belegt waren. Ab diesem Zeitpunkt blieb die Laufzeit konstant. Die Effizienz des Heat Map Verfahrens ist jedoch stark von der Präzision abhängig, so dauert eine Zuweisung bei einer Präzision von 100 zwischen 0,4 und 0,5 Sekunden bei ca. 9.000 Spielern, während sie bei einer Präzision von 10 lediglich zwischen 700 und 800 μs bei 100.000 Spielern benötigt werden. Zudem neigt das Heat Map Verfahren bei einer großen Präzision, zu einer starken Speicherauslastung, sodass der Lasttest bei einer Präzision von 100 auf der vorhandenen Testumgebung vorzeitig abgebrochen ist. Einen Großteil der Ziele konnten die Verfahren somit erfüllen. Einzig die Speicherauslastung der Spielserver konnte nicht verringert werden.

Die Berücksichtigung der Auslastung der Spielserver konnte im Rahmen dieser Arbeit nicht getestet werden, da sich mehrere Spielserver immer zusammen auf einem physischen Server befanden und somit immer dieselben Lastwerte ermittelt wurden.

8.3. Ausblick

Innerhalb dieser Arbeit konnten noch nicht alle Aspekte eines Load Balancing Systems berücksichtigt werden, so mangelt es noch an einer Migration der Spieler, wenn einzelne Spielserver überlastet sein sollten oder die Aufteilung nicht mehr ideal ist, weil die Spieler sich bewegt haben. Dazu muss nicht nur die Umverteilung betrachtet werden, sondern auch eine Bewertungsfunktion entwickelt werden, über die beurteilt werden kann, ob eine Umverteilung stattfinden muss. Denkbar wäre auch ein automatisiertes Starten von Spielservern, wenn das gesamte System überlastet ist bzw. ein automatisiertes beenden von Spielservern die Last im

Gesamtsystem gering ist.

Um alle Anforderungen des Timadorus Projektes zu erfüllen, mangelt es auch noch an Ausfallsicherheit [Timb], da es zur Zeit noch kein Konzept gibt, wie mit dem Ausfall des Load Balancing Server umzugehen ist. Eine Möglichkeit wäre es, die Verfahren verteilbar zu machen, sodass die Daten über verschiedene Server repliziert werden. Dadurch wäre auch die Skalierbarkeit des Systems erhöht.

In einem nächsten Schritt können auch die Algorithmen verbessert werden, so könnte auch die Blickrichtung der Spieler mit einbezogen werden, um zu ermitteln, welche Bereiche voraussichtlich zukünftig im Interessengebiet des Spielers stehen. Beim graphbasierten Verfahren wäre ein nächster Schritt, das Laufzeitverhalten auf logarithmisch zu verringern. Eine Idee wäre es, den Graphen doch aufzubauen und mit einer Suche über diesen zu arbeiten. Dies bringt allerdings auch weitere Probleme mit sich. So muss das Einfügen und Entfernen von Knoten sehr effizient sein, da sich sonst bei einer hohen Frequenz an Statusnachrichten, Nachrichten aufstauen würden.

Anhang A.

Pseudocode zum graphbasierten Verfahren

Algorithmus 1 Berechnen der Abstände

```
1:  AbstandSpielserver ← Map()
2:  LeereSpielserver ← []
3:  SummeDurchschnittlicheServerGroesse ← 0
4:  AnzBelegterSpielserver ← 0
5:  GesamtAbstand ← 0
6:  for all Spielserver in AlleSpielserver do
7:      SummeAbstand ← Infinity          ▷ Summe der Abstände von View Boxen mit
    Überschneidungen. Sollte es keine Überschneidung geben der geringste Abstand
8:      Ueberschneidungen ← false
9:      if groesse(Spielserver.ViewBoxen) > 0 then
10:         for all ViewBox in Spielserver.ViewBoxen do
11:             Abstand ← ABSTAND_BERECHNEN(NeueViewBox, ViewBox)
12:             if Abstand < 0 then
13:                 if Ueberschneidungen then
14:                     SummeAbstand ← SummeAbstand + Abstand
15:                 else
16:                     Ueberschneidungen ← true
17:                     SummeAbstand ← Abstand
18:                 end if
19:             else if Abstand < SummeAbstand ∧ ¬Ueberschneidungen then
20:                 SummeAbstand ← Abstand
21:             end if
22:         end for
```

Algorithmus 1 Berechnen der Abstände (Forstsetzung)

```
23:         if Ueberschneidungen then
24:             SummeAbstand ← √3 * ∛SummeAbstand ▷ Mache aus dem Volumen eine Distanz
25:         end if
26:         SummeDurchschnittlicheServerGroesse ← SummeDurchschnittlicheServerGroesse+
            DURCHSCHNITTLICHE_GROESSE(Spielserver.MaxX, Spielserver.MinX,
            Spielserver.MaxY, Spielserver.MinY, Spielserver.MaxZ, Spielserver.MinZ)
27:         AnzBelegterSpielserver ← AnzBelegterSpielserver + 1
28:         AbstandSpielserver[Spielserver] ← SummeAbstand
29:         GesamtAbstand ← GesamtAbstand + SummeAbstand
30:     else                                  ▷ Auf dem Server befindet sich noch kein Spieler
31:         LeereSpielServer.push(Spielserver)
32:     end if
33: end for
34: if groesse(LeereSpielServer) > 0 then
35:     {AbstandSpielserver, GesamtAbstand} ←
        ABSTAND_LEERE_SPIELSERVER(LeereSpielServer, AbstandSpielserver,
        SummeDurchschnittlicheServerGroesse, GesamtAbstand,
        groesse(AlleSpielserver), AnzBelegterSpielserver)
36: end if
37: return {AbstandSpielserver, GesamtAbstand}
```

Algorithmus 2 Berechnung des Abstandes für leere Spielserver

```
1:  function ABSTAND_LEERE_SPIELSERVER(LeereSpielserver, AbstandSpielserver,
    SummeDurchschnittlicheServerGroesse, GesamtAbstand, AnzahlSpielserver,
    AnzBelegterSpielserver)
2:      DurchschnittsGroesse ← SummeDurchschnittlicheServerGroesse / AnzBelegterSpielserver
3:      OptimaleDurchschnittsServergroesse ← 1 / ∛AnzahlSpielserver
4:      AbstandLeereServer ← (OptimaleDurchschnittsServergroesse − DurchschnittsGroesse) / 2
5:      for all Spielserver in LeereSpielserver do
6:          AbstandSpielserver[Spielserver] ← AbstandLeereServer
7:          GesamtAbstand ← GesamtAbstand + AbstandLeereServer
8:      end for
9:      return {AbstandSpielserver, GesamtAbstand}
10: end function
```

Algorithmus 3 Berechnung der Kosten für die Spielserver

```
1: GeringsteKosten ← Infinity
2: BesterSpielserver ← Null
3: for all Spielserver in AlleSpielserver do
4:     DistanzZuAnderenServern ←
   GesamtAbstand − AbstandSpielserver[spielserver]
5:     Kosten ← kosten(DistanzZuAnderenServern)
6:     if Kosten < GeringsteKosten then
7:         GeringsteKosten ← Kosten
8:         BesterSpielserver ← Spielserver
9:     end if
10: end for
11: return BesterSpielserver;
```

Anhang B.

Pseudocode zum Heat Map Matching

Algorithmus 4 Hinzufügen und Entfernen einzelner Einträge aus der Heat Map unter der Annahme, dass der Standardwert in $ServerMap$ Null ist und dass Einträge aus $ServerMap$ automatisch entfernt werden, wenn sie Null sind

```
1:  X ← xStart
2:  while X ≤ xEnd do
3:      Y ← yStart
4:      while Y ≤ yEndS do
5:          Z ← zStart
6:          while Z ≤ zEnd do
7:              ServerMap ← HeatMap[X][Y][Z]
8:              if Veraenderung = neu then
9:                  if serverMap.isEmpty then        ▷ Feld wird noch von keinem anderem Server genutzt
10:                     AnzBelegteFelder ← AnzBelegteFelder + 1
11:                 end if
12:                 NeuerWert ← ServerMap[Spielserver] + 1
13:                 ServerMap[Spielserver] ← NeuerWert
14:                 if NeuerWert = 1 then
15:                     BelegteFelder[Spielserver].push(X, Y, Z)
16:                 end if
17:             else if Veraenderung = entfernt then
18:                 NeuerWert ← ServerMap[Spielserver] − 1
19:                 ServerMap[Spielserver] ← NeuerWert
20:                 if groesse(ServerMap) = 0 then ▷ Feld wird noch von keinem anderem Server mehr genutzt
21:                     AnzBelegteFelder ← AnzBelegteFelder − 1
22:                 end if
```

Algorithmus 4 Hinzufügen und Entfernen einzelner Einträge aus der Heat Map (Forstsetzung)

```
23:                 if NeuerWert = 0 then
24:                     BelegteFelder[Spielserver].delete(X, Y, Z)
25:                 end if
26:             end if
27:             Z ← Z + 1
28:         end while
29:         Y ← Y + 1
30:     end while
31:     X ← X + 1
32: end while
```

Algorithmus 5 Suchen von Überschneidungen unter der Annahme, dass der Standardwert in $ServerMap$ Null ist

```
1:  function SUCHE_ÜBERSCHNEIDUNGEN(xStart, xEnd, yStart, yEnd, zStart, zEnd)
2:      Ueberschneidungen ← Map()
3:      NeueFelder ← 0
4:      X ← xStart
5:      while X ≤ xEnd do
6:          Y ← yStart
7:          while Y ≤ yEnd do
8:              Z ← zStart
9:              while Z ≤ zEnd do
10:                 ServerMap ← HeatMap[X][Y][Z]
11:                 if groesse(ServerVap) = 0 then        ▷ Feld wird noch von niemandem belegt
12:                     NeueFelder ← NeueFelder + 1
13:                 end if
14:                 for all Spielserver in ServerMap do
15:                     Ueberschneidungen[Spielserver] ← Ueberschneidungen[Spielserver] + ServerMap[SpielServer] ∗ Schrittgroesse
16:                 end for
17:                 Z ← Z + 1
18:             end while
19:             Y ← Y + 1
20:         end while
21:         X ← X + 1
22:     end while
23:     return {Ueberschneidungen, NeueFelder}
24: end function
```

Algorithmus 6 Ermitteln eines Standardwerts für leere Server

if $groesse(LeereSpielserver) > 0$ **then**
  $AnzBelegteFelderNeu \leftarrow AnzBelegteFelder + NeueFelder$
  $IdealAnzFelder \leftarrow g0 * \frac{Praezision^3}{AnzServer} + g1 * \frac{\frac{BelegteFelder}{AnzServer}}{AnzBelegteFelder}$
  $Durchschnittsflaeche \leftarrow \frac{}{groesse(AlleSpielserver) - groesse(LeereSpielserver)}$
  $AbstandswertLeereServer \leftarrow \frac{\sqrt[3]{IdealAnzFelder} - \sqrt[3]{DurchschnittAnzFelder}}{2} * Schrittgroessee$
  for all $Spielserver$ in $LeereSpielserver$ **do**
    $Kosten \leftarrow$ KOSTENFUNKTION$(-abstandswertLeereServer)$
    if $Kosten < GeringsteKosten$ **then**
      $BesterSpielserver \leftarrow Spielserver$
      $GeringsteKosten \leftarrow Kosten$
    end if
  end for
end if

Algorithmus 7 Suche nach den nächsten Spielservern außerhalb der View Box

1: $Abstand \leftarrow 1$
2: $Abstandswert \leftarrow Abstand * Schritgroesse$
3: **for all** $Spielserver$ in $RestlicheSpielserver$ **do**
4: **if** $kosten(-Abstandswert) \geq GeringsteKosten$ **then**
5: $RestlicheSpielserver.delete(Spielserver)$
6: **end if**
7: **end for**
8: **while** $groesse(RestlicheSpielserver) > 0$ **do**
9: Verringere jeweils $xStart$, $yStart$ und $zStart$ um Eins, wenn diese größer als Null sind
10: Erhöhe jeweils $xEnd$, $yEnd$ und $zEnd$ um Eins, wenn diese kleiner als $Praezision$ sind
11: $Y \leftarrow yStart$
12: **while** $Y \leq yEnd$ **do**
13: $Z \leftarrow zStart$
14: **while** $Z \leq zEnd$ **do**
15: $\{RestlicheSpielserver, GeringsteKosten, BesterSpielserver\} \leftarrow berechneKosten(HeatMap[xStart][Y][Z], RestlicheSpielserver, GeringsteKosten, BesterSpielserver)$
16: $\{RestlicheSpielserver, GeringsteKosten, BesterSpielserver\} \leftarrow berechneKosten(HeatMap[xEnd][Y][Z], RestlicheSpielserver, GeringsteKosten, BesterSpielserver)$ $Z \leftarrow Z + 1$
17: **end while**$Y \leftarrow Y + 1$
18: **end while**
19: $X \leftarrow xStart + 1$
20: **while** $X \leq xEnd - 1$ **do**

Algorithmus 7 Suche nach den nächsten Spielservern außerhalb der View Box (Forstsetzung)

21: $Y \leftarrow yStart$
22: **while** $Y \leq yEnd$ **do**
23: $\{RestlicheSpielserver, GeringsteKosten, BesterSpielserver\} \leftarrow berechneKosten(HeatMap[X][Y][zStart], RestlicheSpielserver, GeringsteKosten, BesterSpielserver)$
24: $\{RestlicheSpielserver, 1GeringsteKosten, BesterSpielserver\} \leftarrow berechneKosten(HeatMap[X][Y][zEnd], RestlicheSpielserver, GeringsteKosten, BesterSpielserver)$
25: $Y \leftarrow Y + 1$
26: **end while**
27: $Z \leftarrow zStart + 1$
28: **while** $Z \leq zEnd - 1$ **do**
29: $\{RestlicheSpielserver, GeringsteKosten, BesterSpielserver\} \leftarrow berechneKosten(HeatMap[X][yStart][Z], RestlicheSpielserver, GeringsteKosten, BesterSpielserver)$
30: $\{RestlicheSpielserver, GeringsteKosten, BesterSpielserver\} \leftarrow berechneKosten(HeatMap[X][yStart][Z], RestlicheSpielserver, GeringsteKosten, BesterSpielserver)$
31: $Z \leftarrow Z + 1$
32: **end while**
33: $X \leftarrow X + 1$
34: **end while**
35: $Abstand \leftarrow Abstand + 1$
36: $Abstandswert \leftarrow Abstand * Schrittgroesse$
37: **for all** $Spielserver$ in $RestlicheSpielserver$ **do**
38: **if** $kosten(-Abstandswert) \geq GeringsteKosten$ **then**
39: $RestlicheSpielserver.delete(Spielserver)$
40: **end if**
41: **end for**
42: **end while**

B.1. Parallelisierte Algorithmen

Algorithmus 8 Parallelisierte Suche von Überschneidungen im Bereich der View Box

1: **function** SUCHE_ÜBERSCHNEIDUNGEN_REC($StartX$, $EndX$, $StartY$, $EndY$, $StartZ$, $EndZ$, N)
2: $LaengeX \leftarrow EndX - StartX$
3: $LaengeY \leftarrow EndY - StartY$
4: $LaengeZ \leftarrow EndZ - StartZ$
5: **if** $LaengeX \leq N \wedge LaengeY \leq N \wedge LaengeZ \leq N$ **then**
6: **return** SUCHE_ÜBERSCHNEIDUNGEN($StartX$, $EndX$, $StartY$, $EndY$, $StartZ$, $EndZ$)
7: **end if**
8: **if** $LaengeZ > LaengeY$ **then**
9: $\{LinksStartZ, linksEndZ, rechtsStartZ, rechtsEndZ\} \leftarrow halbieren(StartZ, EndZ)$
10: $\{UeberschneidungenLinks, NeueFelderLinks\} \leftarrow$ SUCHE_ÜBERSCHNEIDUNGEN_REC($StartX$, $EndX$, $StartY$, $EndY$, $LinksStartZ$, $LinksEndZ$, N)
11: $\{UeberschneidungenRechts, NeueFelderRechts\} \leftarrow$ SUCHE_ÜBERSCHNEIDUNGEN_REC($StartX$, $EndX$, $StartY$, $EndY$, $RechtsStartZ$, $RechtsEndZ$, N)
12: **else if** $LaengeY > LaengeX$ **then**
13: $\{LinksStartY, LinksEndY, RechtsStartY, RechtsEndY\} \leftarrow halbieren(startY, endY)$
14: $\{UeberschneidungenLinks, NeueFelderLinks\} \leftarrow$ SUCHE_ÜBERSCHNEIDUNGEN_REC($StartX$, $EndX$, $LinksStartY$, $LinksEndY$, $StartZ$, $EndZ$, N)
15: $\{UeberschneidungenRechts, NeueFelderRechts\} \leftarrow$ SUCHE_ÜBERSCHNEIDUNGEN_REC($StartX$, $EndX$, $RechtsStartY$, $RechtsEndY$, $StartZ$, $EndZ$, N)
16: **else** ▷ Alle gleich lang
17: $\{LinksStartX, LinksEndX, RechtsStartX, RechtsEndX\} \leftarrow halbieren(startX, endX$
18: $\{UeberschneidungenLinks, NeueFelderLinks\} \leftarrow$ SUCHE_ÜBERSCHNEIDUNGEN_REC($LinksStartX$, $LinksEndX$, $StartY$, $EndY$, $StartZ$, $EndZ$, N)
19: $\{UeberschneidungenRechts, NeueFelderRechts\} \leftarrow$ SUCHE_ÜBERSCHNEIDUNGEN_REC($RechtsStartX$, $RechtsEndX$, $StartY$, $EndY$, $StartZ$, $EndZ$, N)

Algorithmus 8 Parallelisierte Suche von Überschneidungen im Bereich der View Box (Forstsetzung)

end if
return $\{UeberschneidungenLinks.concat(UeberschneidungenRechts), NeueFelderLinks + NeueFelderRechts\}$
end function

Algorithmus 9 Parallelisiertes Absuchen der Seiten nach Überschneidungen

function SUCHE_SEITEN($StartX$, $EndX$, $StartY$, $EndY$, $StartZ$, $EndZ$, N)
  $\{Ueberschneidungen1, NeueFelder1\} \leftarrow$ SUCHE_ÜBERSCHNEIDUNGEN_REC($StartX$, $StartX$, $StartY$, $EndY$, $StartZ$, $EndZ$, N)
  $\{Ueberschneidungen2, NeueFelder2\} \leftarrow$ SUCHE_ÜBERSCHNEIDUNGEN_REC($EndX$, $EndX$, $StartY$, $EndY$, $StartZ$, $EndZ$, N)
  $\{ueberschneidungen3, neueFelder3\} \leftarrow$ SUCHE_ÜBERSCHNEIDUNGEN_REC($StartX + 1$, $EndX - 1$, $StartY$, $StartY$, $StartZ$, $EndZ$, N)
  $\{Ueberschneidungen4, NeueFelder4\} \leftarrow$ SUCHE_ÜBERSCHNEIDUNGEN_REC($StartX + 1$, $EndX - 1$, $EndY$, $EndY$, $StartZ$, $EndZ$, N)
  $\{Ueberschneidungen5, NeueFelder5\} \leftarrow$ SUCHE_ÜBERSCHNEIDUNGEN_REC($StartX + 1$, $EndX - 1$, $StartY + 1$, $EndY - 1$, $StartZ$, $StartZ$, N)
  $\{Ueberschneidungen6, NeueFelder6\} \leftarrow$ SUCHE_ÜBERSCHNEIDUNGEN_REC($StartX + 1$, $EndX - 1$, $StartY + 1$, $EndY - 1$, $EndZ$, $EndZ$, N)
  return *concat*($Ueberschneidungen1$, $Ueberschneidungen2$, $Ueberschneidungen3$, $Ueberschneidungen4$, $Ueberschneidungen5$, $Ueberschneidungen6$)
end function

Literaturverzeichnis

[AIB09] Ali, Af Ahmad F. ; Ismail, As Abdul Samad A. ; Bade, Abdullah: An Overview of Networking Infrastructure for Massively Multiplayer Online Games. In: *Comp.Utm.My* (2009), Nr. October 2008, 619–628. http://comp.utm.my/pars/files/2013/04/An-Overview-of-Networking-Infrastructure-for-Massively-Multiplayer-Online-Games.pdf

[Bag01] Bagwell, Phil: Ideal hash trees. In: *Es Grands Champs* (2001). http://infoscience.epfl.ch/record/64398/files/idealhashtrees.pdf$\delimiter"026E30F$nhttp://infoscience.epfl.ch/record/64398

[BGL14] Behnke, Lutz ; Grecos, Christos ; Luck, Kai V.: QuP : Graceful Degradation in State Propagation for DVEs. (2014)

[CCY99] Cardellini, Valeria ; Colajanni, Michele ; Yu, Philip S.: Dynamic load balancing on web-server systems. In: *IEEE Internet Computing* 3 (1999), Nr. 3, S. 28–39. http://dx.doi.org/10.1109/4236.769420. – DOI 10.1109/4236.769420. – ISBN 1089–7801

[CWD+05] Chen, Jin ; Wu, Baohua ; Delap, Margaret ; Knutsson, Björn ; Lu, Honghui ; Amza, Cristiana: Locality aware dynamic load management for massively multiplayer games. In: *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '05* (2005), 289. http://dx.doi.org/10.1145/1065944.1065982. – DOI 10.1145/1065944.1065982. ISBN 1595930809

[DL12] Deng, Yunhua ; Lau, Rynson W H.: On delay adjustment for dynamic load balancing in distributed virtual environments. In: *IEEE Transactions on Visualization and Computer Graphics* 18 (2012), Nr. 4, S. 529–537. http://dx.doi.org/10.1109/TVCG.2012.52. – DOI 10.1109/TVCG.2012.52. – ISSN 10772626

[DL14] Deng, Yunhua ; Lau, Rynson W. H.: Dynamic load balancing in distributed virtual environments using heat diffusion. In: *ACM Transactions on Multimedia Computing, Communications, and Applications* 10 (2014), Februar, Nr. 2, 1–19. http://dx.doi.org/10.1145/2499906. – DOI 10.1145/2499906. – ISSN 15516857

[DVV+05] De Vleeschauwer, Bart ; Van Den Bossche, Bruno ; Verdickt, Tom ; De Turck, Filip ; Dhoedt, Bart ; Demeester, Piet: Dynamic microcell assignment for massively multiplayer online gaming. In: *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games - NetGames '05* (2005), 1–7. http://dx.doi.org/10.1145/1103599.1103611. – DOI 10.1145/1103599.1103611. ISBN 1595931562

[eBa15] *Netzwerklast sicher verteilen - Load Balancer bei der Arbeit | eBay.* http://www.ebay.de/gds/Netzwerklast-sicher-verteilen-Load-Balancer-bei-der-Arbeit-/10000000177880852/g.html. Version: 2015, Abruf: 2015-08-04

[Erl] *Erlang/OTP 18.* http://www.erlang.org/doc/, Abruf: 2015-08-19

[Erl15] *Erlang Programming Language Erlang/OTP 18.0 has been released.* http://www.erlang.org/news/88. Version: 2015, Abruf: 2015-08-26

[Eve13] *Building a Balanced Universe - EVE Community.* http://community.eveonline.com/news/dev-blogs/building-a-balanced-universe/. Version: 2013, Abruf: 2015-08-07

[Fal94] Falkenauer, Emanuel: A New Representation and Operators for Genetic Algorithms Applied to Grouping Problems. In: *Evolutionary Computation* 2 (1994), Juni, Nr. 2, 123–144. http://dx.doi.org/10.1162/evco.1994.2.2.123. – DOI 10.1162/evco.1994.2.2.123. – ISSN 1063–6560

[Hí13] Hébert, Fred: *Learn You Some Erlang for Great Good!: A Beginner's Guide.* No Starch Press, 2013. – ISBN 978–1–59327–435–1

[Jin] *Erlang – Jinterface Reference Manual.* http://www.erlang.org/doc/apps/jinterface/, Abruf: 2015-03-13

[KAS07] Kazem, Ihab ; Ahmed, Dewan T. ; Shirmohammadi, Shervin: A visibility-driven approach to managing interest in distributed simulations with dyna-

mic load balancing. In: *Proceedings - IEEE International Symposium on Distributed Simulation and Real-Time Applications, DS-RT* (2007), S. 31–38. http://dx.doi.org/10.1109/DS-RT.2007.11. – DOI 10.1109/DS–RT.2007.11. – ISBN 0769530117

[Lau10] Lau, Rynson W.: Hybrid load balancing for online games. In: *Proceedings of the international conference on Multimedia - MM '10* (2010), 1231. http://dx.doi.org/10.1145/1873951.1874194. – DOI 10.1145/1873951.1874194. ISBN 9781605589336

[LC02] Lui, J C S. ; Chan, M F.: An efficient partitioning algorithm for distributed virtual environment systems. In: *IEEE Transactions on Parallel and Distributed Systems* 13 (2002), Nr. 3, 193–211. http://dx.doi.org/10.1109/71.993202. – DOI 10.1109/71.993202. – ISSN 10459219

[LC10] Lee, Yeng T. ; Chen, Kuan T.: Is server consolidation beneficial to MMORPG? A case study of world of warcraft. In: *Proceedings - 2010 IEEE 3rd International Conference on Cloud Computing, CLOUD 2010* (2010), S. 435–442. http://dx.doi.org/10.1109/CLOUD.2010.57. – DOI 10.1109/CLOUD.2010.57. ISBN 9780769541303

[LPM06] Lu, Fengyun ; Parkin, Simon ; Morgan, Graham: Load balancing for massively multiplayer online games. In: *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games - NetGames '06* (2006), 1. http://dx.doi.org/10.1145/1230040.1230064. – DOI 10.1145/1230040.1230064. ISBN 1595935894

[MFOn03] Morillo, P. ; Fernández, M. ; Orduña, J.M.: An ACS-based partitioning method for distributed virtual environment systems. In: *Proceedings International Parallel and Distributed Processing Symposium* 00 (2003), Nr. C. http://dx.doi.org/10.1109/IPDPS.2003.1213283. – DOI 10.1109/IPDPS.2003.1213283. – ISBN 0–7695–1926–1

[MMO14] *MMOData4.1.0.xlsx*. http://users.telenet.be/mmodata/Charts/MMOData4.1.0.xlsx. Version: 2014, Abruf: 2015-04-15

[MO12] Ma, Minhua ; Oikonomou, Andreas: Network architectures and data management for massively multiplayer online games. In: *Evolving Developments in Grid and Cloud Computing: Advancing Research* (2012), S. 144 – 155

[MOnFD03] Morillo, P ; Orduña, J M. ; Fernández, M ; Duato, José: An Adaptive Load Balancing Technique For Distributed Virtual Environment Systems. In: *Proceedings of the 15th IASTED international PDCS-03* (2003), S. 256–261

[NSHW11] Nemeth, Evi ; Snyder, Garth ; Hein, R. T. ; Whaley, Ben: *Unix and Linux System Administration Handbook*. 4. Pearson Education, 2011. – ISBN 978–0–13–148005–6

[NSLL02] Ng, Beatrice ; Si, Antonio ; Lau, Rynson W. ; Li, Frederick W.: A multi-server architecture for distributed virtual walkthrough. In: *Proceedings of the ACM symposium on Virtual reality software and technology - VRST '02* (2002), 163. http://dx.doi.org/10.1145/585767.585768. – DOI 10.1145/585767.585768. ISBN 1581135300

[RO03] Rosedale, Philip ; Ondrejka, Cory: *Enabling Player-Created Online Worlds with Grid Computing and Streaming*. http://www.gamasutra.com/resource_guide/20030916/rosedale_pfv.htm. Version: 2003, Abruf: 2015-08-10

[SKS92] Shivaratri, NG ; Krueger, Phillip ; Singhal, Mukesh: Load distributing for locally distributed systems. In: *Computer* (1992), 33–44. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=179115

[Tima] *Authentifikations Server - Timadorus*. https://wiki.informatik.haw-hamburg.de/secure-wiki/timadorus/index.php/Authentifikations_Server, Abruf: 2015-03-12

[Timb] *Projekt | Timadorus*. http://timadorus-web2.informatik.haw-hamburg.de/de/project/, Abruf: 2015-03-14

[Timc] *Timadorus Message Protokoll - Timadorus*. https://wiki.informatik.haw-hamburg.de/secure-wiki/timadorus/index.php/Timadorus_Message_Protokoll, Abruf: 2015-03-12

[TS08] Tanenbaum, Andrew S. ; Steen, Maarten V.: *Verteilte Systeme - Prinzipien und Paradigmen*. 2. München : Pearson Studium, 2008. – ISBN 978–3–8273–7293–2

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 12. September 2015 Sven Allers