[redacted]

Netcode – Grundlagen des Networkings in Multiplayer-Videospielen

Praxisarbeit
zur Erlangung des akademischen Grades eines
Bachelor of Science (B. Sc.)
in der Studienrichtung Informatik

Eingereicht von: Simon Nikolaidis
[redacted]

Betreuer: [redacted]

[redacted]

Inhaltsverzeichnis

Abkürzungsverzeichnis

DDOS – Distributed Denial-of-Service

DGS – Dedicated Game Server

GGPO – Good Game, Peace Out

ICMP – Internet Control Message Protocol

IP – Internet Protocol

P2P – Peer-to-Peer

TCP – Transmission Control Protocol

UDP – User Datagram Protocol

WAN – Wide Area Network

1 Einleitung

1.1 Motivation

Schaut man sich Videospiele von vor 15 Jahre an und vergleicht diese mit Heutigen, erkennt man schnell, dass die Entwicklung von Videospielen in den letzten Jahren beeindruckend ist. Für viele gehören sie als Freizeitaktivität zum Alltag und manche haben ihr Hobby damit zum Beruf gemacht. Oft verbringen Menschen viel Zeit beim Spielen von Videospielen und dabei gehört es nicht selten dazu, die Erlebnisse in der digitalen Welt mit anderen Spielern zu teilen. Viele Spiele haben deshalb einen Multiplayer-Modus, welcher es mehreren Spielern auf der ganzen Welt ermöglichen soll, sich miteinander zu vernetzen und gemeinsam oder gegeneinander zu spielen. Das Verlangen nach immer schnelleren und realistischeren Multiplayer-Videospielen bringt jedoch auch große Herausforderungen mit sich, für deren Bewältigung neue Technologien notwendig sind. Die Grundlagen für das Networking von Multiplayer-Spielen, sowie die auftretenden Probleme dabei und Möglichkeiten diese zu beseitigen sollen in dieser Arbeit erläutert werden.

1.2 Was ist Netzcode?

Der Begriff Netzcode leitet sich vom englischen Begriff „netcode“ ab und ist eine Abkürzung für „network-code“ also den Netzwerk-Programmcode.[1] Im Allgemeinen umfasst Netzcode das gesamte Networking in Online-Spielen und die dabei entstehenden Probleme und Herausforderungen, welche es für ein angenehmes Spielerlebnis in Echtzeit Multiplayer-Spielen zu bewältigen gilt.[2]

Niemand möchte noch hinter einer Deckung vom Gegner getroffen werden oder beim Laufen zu einem Ziel ständig zurückgesetzt werden. Das würde genauso viel Spaß machen wie auf einer Gitarre zu spielen, auf der die Töne erst eine Sekunde nach dem Saitenanschlag zu hören sind.

2 Netzwerkarten für Videospiele

Es ist wichtig zu verstehen, welche Netzwerktopologien für Videospiele eingesetzt werden, um später nachvollziehen zu können, wieso bestimmte Probleme auftreten. Im Wesentlichen unterscheiden sich diese in der Art, wie Computer miteinander kommunizieren und vernetzt sind.

2.1 Peer-to-Peer

Bei einer reinen Peer-to-Peer (P2P) Topologie verbindet sich jeder Spieler mit jedem anderen Spieler und tauscht Informationen über den Spielstand und jedes Event, welches das Spielgeschehen beeinflusst, mit ihnen aus. Die einzelnen Clients kommunizieren also alle direkt

miteinander. Dabei gibt es keinen einzelnen „Host“ oder Server, stattdessen akzeptiert jeder Spieler Event- und Spielupdates von jedem anderen Client.[3]
Am häufigsten zum Einsatz kommt dieses Konzept für 1 gegen 1 Kampf- oder Sportspiele, denn durch die direkte Verbindung der Clients entsteht keine zusätzliche Verzögerung durch die Kommunikation über einen Server.[4] Dadurch bekommt keiner der beiden Spieler einen Vorteil durch eine bessere Verbindung. Ein Ziel ist also eine schnellere Kommunikation durch die Verkürzung der Verbindungswege. Da kein Server vorhanden ist, entscheiden die Clients selbst darüber, wie sie die erhaltenen Updates der anderen Spieler in ihr Spielgeschehen einbringen. Dadurch kann es jedoch zu asynchronen Spielständen kommen, wenn zwei Clients unterschiedlich über die Einbringung von Aktualisierungen entscheiden.
Ein großes Problem beim P2P ist die mangelnde Sicherheit. Ohne einen autoritären Server kann nur schwierig verhindert werden, dass ein Client unerwünschte Änderungen am Spielstand vornimmt. Außerdem kann ein Spieler die WAN IP-Adressen seiner Mitspieler sehen, was die Gefahr von beispielsweise DDOS-Attacken erhöht.[4]

2.2 Client-hosted

Hierbei handelt es sich um eine Client-Server Architektur, wobei der Computer oder die Konsole eines Spielers genutzt wird, um den Hauptspielstand bereitzustellen und zu verwalten. Dadurch wird der Spieler auch gleichzeitig zum Server, also Host für die aktuelle Spielsitzung.[3]
Auch dieses Modell bringt einige Probleme mit sich, denn der Client-Host hat oft einen Vorteil gegenüber den anderen Spielern, da er keine Verzögerung bei seiner Verbindung zum Server hat. Außerdem muss der Computer sowie die Internetverbindung des Spielers leistungsstark genug sein, um ein flüssiges Spielerlebnis zu ermöglichen. Dies kann nicht immer mit absoluter Sicherheit gewährleistet werden. Wenn der PC beispielsweise per W-LAN mit dem Internet verbunden ist, kann das schnell zu Problemen führen.[4] Ein weiteres Problem ist die sogenannte Host-Migration. Verlässt der aktuelle Host die Sitzung, muss ein neuer gefunden werden und dadurch wird das Spiel für eine kurze Zeit pausiert.

2.3 Authoritative dedicated Game-Server

Auch diese Topologie ist ein Client-Server-Modell, wobei sich alle Spieler mit einem von einem Unternehmen oder Cloud-Service bereitgestellten „dedicated game server“ (DGS) verbinden. Die Clients senden ihre Updates an den Server, welcher diese validiert, in den aktuellen Spielstand einbindet und anschließend den resultierenden Spielstatus an alle Teilnehmer sendet.[3]
Wichtig ist hierbei natürlich den Spielereingaben nicht blind zu vertrauen. Befindet sich ein Spieler beispielsweise auf der Position (10, 10) und aktualisiert seine Position innerhalb einer Sekunde

auf (20, 10), wobei er möglicherweise durch eine Wand geht oder sich schneller bewegt als möglich, darf der Server das entweder gar nicht erst akzeptieren oder den Spieler gegebenenfalls auf die ursprüngliche Position zurücksetzen. Eine andere Lösung wäre, dass der Spieler dem Server seine Eingabe mitteilt, also dass er beispielsweise ein Feld nach rechts gehen möchte. Da der Server weiß, dass sich der Spieler bei (10, 10) befindet, aktualisiert er diese Position und teilt allen Clients, auch dem sich bewegenden Spieler, die neue Position (11, 10) mit.[5]
Dieses Modell wird am häufigsten für anspruchsvolle Echtzeit Muliplayer-Spiele verwendet. Da der Server immer den aktuellen Hauptspielstand bereitstellt und verwaltet, haben die Entwickler die größte Kontrolle über das Spielgeschehen und viele Fehler und Sicherheitsschwierigkeiten können einfacher gelöst werden.
Hierbei ist zu beachten, dass alle Spieler weltweit Zugang zu einem Server mit einer niedrigen Latenzzeit bekommen, also ausreichend Server, verteilt auf der ganzen Welt, gehostet werden.

Wenn sich eine Spieleentwicklungsfirma dazu entscheidet eine P2P- oder Client-hosted-Topologie für ein 2+ Spieler Multiplayer-Spiel einzusetzen, dann geht es dabei immer um Kostenminimierung und niemals um die Optimierung des Spielerlebnisses, denn dafür ist ein Netzwerk mit DGS's am besten geeignet.[4]

3 Transport Protokolle – TCP vs. UDP

Nachdem dargestellt wurde, wie die Spieler in den meisten Fällen miteinander vernetzt sind, ist es ebenfalls wichtig zu verstehen, wie die Spieldaten unter den Clients ausgetauscht werden. Dafür gibt es zwei relevante Transport Protokolle auf der Transportschicht. Das Transmission Control Protocol (TCP) und das User Datagram Protocol (UDP).[6]
Sowohl TCP als auch UDP basieren auf dem Internet Protokoll (IP). IP ermöglicht es Datenpakete von einem Ort zum anderen zu versenden. Dabei wird jedoch nicht garantiert, dass das Paket an seinem Ziel angekommen ist, dass die Daten nicht beschädigt und verändert wurden oder dass die Pakete in der richtigen Reihenfolge ankommen.[6]
TCP erweitert IP um viele Funktionalitäten, wohingegen UDP sehr nah an IP aufgebaut ist. Zusätzliche Unterschiede sind zum Beispiel, dass UDP im Gegensatz zu TCP verbindungslos ist. Es muss also keine Verbindung für den vollständigen Datenaustausch hergestellt, aufrechterhalten und geschlossen werden. Bei TCP, welches verbindungsorientiert ist, muss vor der Datenübertragung eine Verbindung hergestellt und danach wieder geschlossen werden.[7] Bei UDP wird einfach ein Datenpaket an ein Ziel, also eine IP-Adresse und einen dazugehörigen Port versendet, welches von Computer zu Computer weitergegeben wird, bis es irgendwann am Ziel ankommt oder auf dem Weg verloren geht. Bei TCP hingegen wird eine Übermittlung von Daten

sowie die Einhaltung der Reihenfolge der Pakte garantiert. Es gibt also bei UDP auch keine Retransmission. TCP besitzt deutlich mehr Funktionen und ist komplexer als UDP, dadurch ist es jedoch auch langsamer beim Datentransfer. UDP ist deutlich einfacher und damit effizienter und schneller als TCP.[7]

Bei vielen online Multiplayer-Spielen ist die Latenzzeit sehr wichtig, weshalb hierbei in den meisten Fällen UDP eingesetzt und damit der Verlust einzelner Pakete für ein ruckelfreies Spielerlebnis in Kauf genommen wird. TCP kann teilweise für Runden basierte Strategiespiele eingesetzt werden, jedoch ist die Verwendung von TCP für Videospiele in den meisten Fällen suboptimal.[6] Denn beim Einsatz von TCP würde zum Beispiel, wenn ein Paket verloren geht, das Spiel für alle Spieler kurz pausieren, da der Server auf das verlorene Packet wartet, welches zu dem Zeitpunkt der Ankunft sowieso schon wieder veraltet ist. Deshalb braucht man bei Videospielen keinen zuverlässigen und geordneten Datenstrom, sondern es geht darum die Daten so schnell wie möglich vom Client zum Server zu senden ohne auf verloren gegangene Daten zu warten.[8]

Auch TCP und UDP gleichzeitig zu verwenden ist keine gute Idee, da beide Protokoll auf IP basieren und das dazu führen könnte, dass die von den einzelnen Protokollen gesendeten Pakete sich gegenseitig beeinflussen.[8]

Bei Online-Spielen, wo eine geringe Latenzzeit nötig ist, ist es also die beste Lösung ausschließlich UDP zu verwenden und wenn benötigt, die fehlenden Funktionen in einem individuell entwickelten auf UDP basierenden Transport Protokoll zu implementieren. Dadurch kann man viele Funktionen von TCP angepasst auf den Echtzeit-Datentransfer nutzen und gleichzeitig von der Schnelligkeit von UDP profitieren.

4 Was Netzcode so schwierig macht

4.1 Latenz

Latenz, Latenzzeit oder Verzögerung ist die Zeit, welche ein Signal benötigt, um von seiner Quelle zum Ziel zu kommen. Diese Verzögerung muss bei der Entwicklung des Netzcodes für Echtzeit-Anwendungen unbedingt mit beachtet werden, denn man kann keinesfalls davon ausgehen, dass ein Signal unverzüglich von A nach B gelangen kann.

Bei Online-Spielen begegnet man - vor allem bei der Serverauswahl - oftmals dem Begriff „Ping“. Wenn ein Computer einen Server anpingt, dann sendet dieser ein Internet Control Message Protocol (ICMP) – „Echo Request“ an den Spielserver. Dieser antwortet anschließend mit einem ICMP – „Echo Reply“.[9] Die Zeit zwischen dem Senden der Anfrage und dem Erhalten der Antwort ist der Ping des Computers zum Spielserver. Wenn der Ping also 40ms beträgt, so

benötigen die Daten ca. 20ms um vom Client zum Server zu gelangen. Zu beachten ist hierbei jedoch noch, dass es geringe Unterschiede zwischen der Latzenzzeit beim Senden von ICMP-Requests und den tatsächlichen Spieldaten geben kann. Das bedeutet der angezeigte Ping ist das „Best-Case-Szenario“ und während des Spielens können auch bei einem akzeptablen Ping noch einige Probleme auftreten.

Durch einen hohen Ping kann es beispielsweise zu sogenannten Lags kommen.[10] Also zu einer überdurchschnittlich langen Latenz- bzw. Verzögerungszeit, wodurch das Spielgeschehen als langsam und nicht mehr flüssig wahrgenommen wird. Aus diesem Grund sollten die Latenzzeit sowie der Ping stets möglichst gering sein.

4.1.1 Probleme durch zu hohe Latenz

Zwei Beispiele, warum eine hohe Latenzzeit problematisch in Echtzeit-Onlinespielen ist:

Zu einem bestimmten Zeitpunkt schießt ein Spieler auf einen anderen Spieler, welcher sich genau in seinem Fadenkreuz befindet. Dieses Schuss-Event benötigt jedoch einen Moment bis es an den anderen Spieler gesendet wurde. Zu diesem Zeitpunkt hat sich der Spielstand des anderen Spielers vielleicht bereits so verändert, dass dieser nicht mehr getroffen werden würde, da er eventuell bereits hinter einer Deckung ist. Beachtet man diese Latenzzeit nicht, so kommt es zu einem asynchronen Spielstand der beiden Spieler. Einen wo der andere Spieler getroffen wurde und einen wo er nicht getroffen wurde.[11]

Das gleiche trifft zu, wenn ein Spieler hinter einem Hindernis hervorschaut. Das Event, dass der Spieler hinter der Deckung hervorkommt, muss erst an alle anderen Spieler gesendet werden, was einen kurzen Moment dauern kann. In diesem Fall hätte der Spieler, welcher sich aus der Deckung bewegt einen Vorteil, weil er die anderen Spieler eher sehen kann, bevor sie ihn sehen können. Umso größer diese Verzögerung ist, desto gravierender werden auch diese auftretenden Probleme.

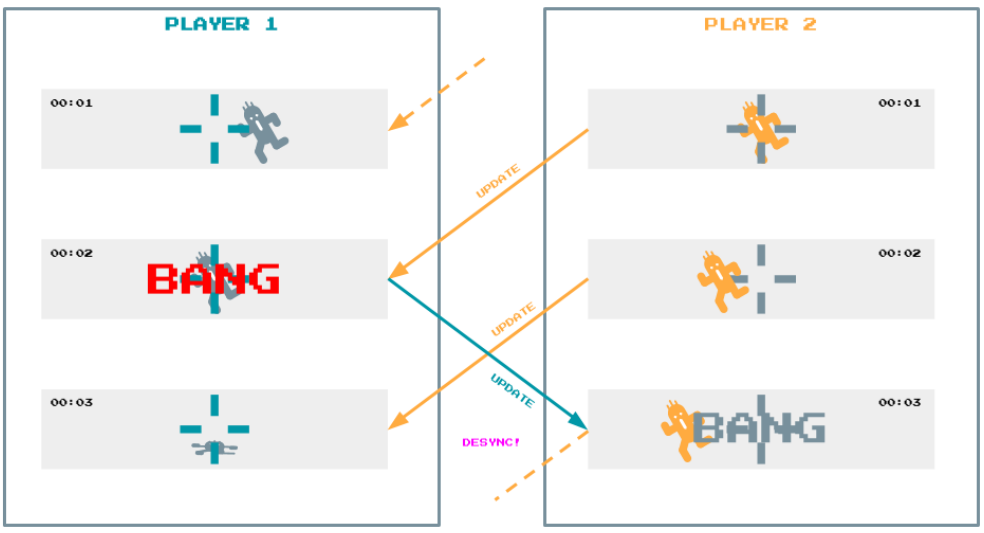

Abbildung 1: Verzögertes Schuss-Event führt zu unterschiedlichen Spielständen [11]

4.1.2 Ursachen für hohe Latenz – Physik und Routing

Nichts ist schneller als das Licht und somit kann sich auch kein Signal schneller als mit Lichtgeschwindigkeit fortbewegen. Betrachtet man die Entfernung eines Spielers in Moskau zu einem Server in Melbourne mit rund 14.430 km Luftlinie, so benötigt ein Signal hier bereits 48,1ms für eine Strecke bei optimalen Bedingungen und auf direktem Weg. Dieser Wert wird jedoch nicht erreicht, denn zur Berechnung dieser Verzögerung wurde die Lichtgeschwindigkeit im Vakuum verwendet. Die Geschwindigkeit des Lichtes in einem Glasfaserkabel ist jedoch um etwa 30% langsamer, wodurch die Latenzzeit nun bereits 72ms betragen würde.[11] Hinzu kommt, dass man nicht einmal davon ausgehen kann, dass sich zwischen dem Computer im Heimnetzwerk und dem Server ausschließlich Glasfaserkabel befinden. Diese zuvor ermittelte Latenz sagt jedoch noch gar nichts aus, denn die größte Verzögerung entsteht durch den Weg, welchen das Signal in Wirklichkeit zurücklegen muss. Dieser ist nämlich deutlich größer als die direkte Luftlinie.

Diese Wegfindung des Signals über mehrere Netzwerke, Pfade und Hops bezeichnet man als Routing. Dabei muss das Datenpaket an mehreren Stellen verarbeitet werden und von dem jeweiligen Router darüber entschieden werden, über welche Leitung beziehungsweise zu welchem Netzwerkknoten es seine Reise fortsetzt.[12]

Umso öfter dieser Vorgang des Lesens, Verarbeitens und Weiterleitens notwendig ist, desto mehr Zeit wird beansprucht und desto größer wird die Verzögerung. Dadurch kann es vorkommen, dass ein Server, welcher geographisch näher am Computer liegt nicht die optimalste Latenzzeit bietet, da die Route zu diesem Server länger und mit einer größeren Verzögerung verbunden ist und es kürzere Routen zu anderen geographisch weiter entfernteren Servern gibt. Das kann vorkommen, wenn ein Router eine schlechte Route auswählt oder eine alternative Route wählen muss, weil die eigentlich Bessere überlastet oder nicht verfügbar ist.[12]

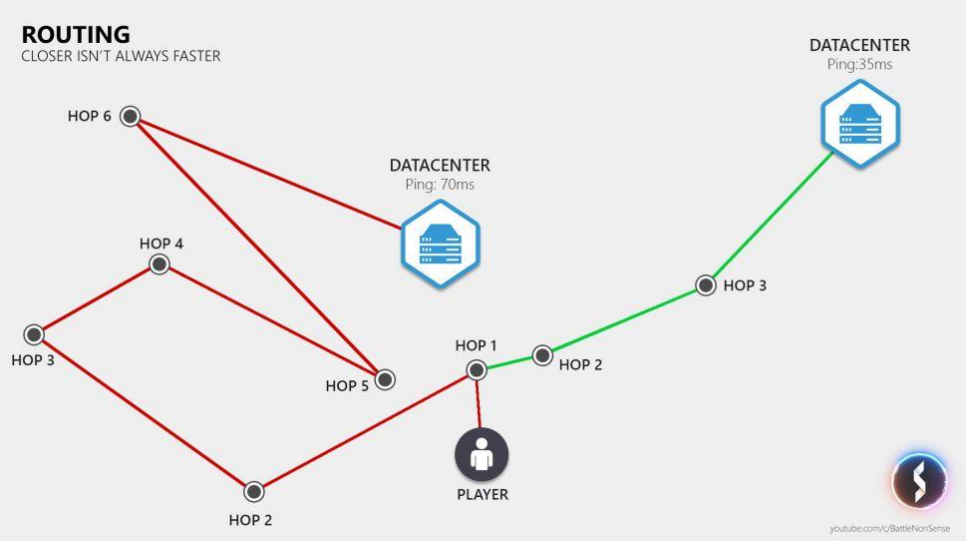

Abbildung 2: Routing - geographisch nahe Server bedeuten nicht gleich eine gute Route [4]

Für einen erfolgreichen Datentransfer benötigt es außerdem eventuell mehr als bloß den Weg vom Computer zum Server. Wenn beispielsweise das TCP-Protokoll verwendet wird, so braucht es noch eine Bestätigung des Servers, dass das Datenpaket angekommen ist. Der Weg muss also doppelt zurückgelegt werden, hin und zurück. Dadurch wird die bereits zu große Verzögerung nochmal verdoppelt. Wenn das Paket unterwegs verloren geht, wird eine Retransmission durchgeführt und der Weg beginnt von vorn. Wenn das Datenpaket irgendwann beim Server ankommt, ist es bereits so veraltet, dass es vermutlich nicht mehr eingearbeitet wird und alles umsonst war.[11] Auch hieran zeigt sich nochmal, warum TCP als Protokoll für Anwendungen, welche auf einen möglichst schnellen Datentransfer angewiesen sind nicht geeignet ist.

4.2 Paketverlust und Jitter

Der angesprochene Paketverlust kann außerdem zusätzliche Probleme verursachen, wenn nämlich für das Spielgeschehen kritische Updates dadurch verlorengehen.

Wenn der Server die Information, dass eine Tür geöffnet wurde an alle Spieler sendet und ein Spieler diese Information durch den Verlust eines Datenpaketes nicht erhält, so bleibt die Tür bei ihm geschlossen und er hat einen asynchronen Spielstand.

Paketverlust kann beispielsweise durch fehlerhafte Datenpakete, Routingfehler, Überlastung des Netzwerkes, einen kurzfristigen Verbindungsabbruch oder Interferenzen bei kabellosen Verbindungen auftreten.[4]

Mindestens genauso schlimm wie eine langsame Latenz ist eine stark variierende Latenz von aufeinanderfolgenden Datenpaketen. Das bezeichnet man als Jitter, was es Clients und Server schwer macht, sich auf ankommende Updates zeitlich einzustellen. Wechselt die Latenz in kurzen Intervallen, so kann dies sogar die Paketreihenfolge verändern und ältere Datenpakete können vor Neueren ankommen. Jitter kann beispielsweise durch eine Netzüberlastung, schwache Hardware oder kabellose Verbindungen ausgelöst werden.[13]

4.3 Update-Rate des Servers

Neben der Zeit wie lange die Daten brauchen um zum Server beziehungsweise zurück zu gelangen, ist ebenfalls wichtig, wie oft also in welchen Zeitabständen die Datenpakete gesendet und empfangen werden. Wie oft ein Server Aktualisierungen pro Sekunde an die Clients sendet, wird als Update-Rate bezeichnet.

Wenn ein Spiel beispielsweise Aktualisierungen mit einer Frequenz von 30Hz sendet und empfängt, vergeht mehr Zeit zwischen den Aktualisierungen als wenn es sie mit 60Hz senden und empfangen würde. Es entsteht also eine größere Verzögerung durch eine langsamere Update-Rate.[12]

Anhand der in den Datenpaketen enthaltenen Informationen wird das Spielgeschehen auf dem Bildschirm dargestellt. Das bedeutet, je mehr Informationen pro Sekunde gesendet werden, desto flüssiger ist das Spielgeschehen.

Ein Problem, welches bei einer zu niedrigen Aktualisierungsrate auftreten kann sind beispielsweise „super-bullets“. Wenn ein Server eine Aktualisierungsrate von 10Hz also 10 Aktualisierungen pro Sekunde hat, vergehen 100ms zwischen jeder Aktualisierung. Bei einer Waffe welche unter 600 Schüsse pro Minute abfeuern kann funktioniert das, denn zwischen jedem Schuss liegen mehr als 100ms. Schießt man jetzt jedoch mit einer Waffe welche 800 Schüsse pro Minute abgeben kann, so kann es dazu kommen, dass 2 Schüsse beziehungsweise Treffer mit einer Aktualisierung gesendet werden. Dann wirkt es so, als wäre es in Wirklichkeit ein Treffer gewesen und die Waffe würde deutlich mehr Schaden anrichten als normalerweise möglich wäre. Ursprünglich waren es jedoch 2 Treffer, welche mit einer Aktualisierung gesendet wurden.[4]

Eine höhere Aktualisierungsrate bietet also ein besseres Spielerlebnis und eine geringe Verzögerung. Außerdem werden Probleme verringert, welche durch einen geringen Paketverlust verursacht werden, da das Fehlen eines verlorengegangenen Paketes unter den vielen anderen nicht mehr so auffällt.

4.4 Tick-Rate des Servers

Die Tickrate bezeichnet bei Spielservern die Frequenz, mit welcher der Server die Spielsimulation neu berechnet und hängt mit der Aktualisierungsrate des Servers zusammen, beschreibt jedoch etwas anderes.[14]

Eine Tickrate von 60Hz ermöglicht es nämlich einem Server 60 Aktualisierungen pro Sekunde an die Spieler zu senden und verursacht dementsprechend weniger Verzögerung als beispielsweise eine Tickrate von 30Hz.[12]

Bei einer höheren Tickrate benötigt ein Server jedoch auch zunehmend eine höhere Rechenleistung, da er das Spielgeschehen viel öfter und vor allem schneller simulieren muss. Zudem steigt die Netzlast durch die erhöhte Kommunikation mit den Clients.

Am wichtigsten ist jedoch, dass ein Server die Verarbeitung der Simulation schnellstmöglich durchführt und vor allem innerhalb des Zeitrahmens schafft. Bei einer Tickrate von 60Hz hat der Server nämlich nur 16,66ms um die erhaltenen Daten zu verarbeiten, die Simulationen durchzuführen und die Ergebnisse an die Clients zu senden. Schafft es der Server nicht rechtzeitig die Verarbeitung abzuschließen und die Spieler mit neuen Informationen zu versorgen, treten starke Fehler und Probleme im Spiel auf.[4]

Ein Beispiel für ein möglicherweise auftretendes Problem wäre „Rubberbanding“. Dies bezeichnet das plötzliche unerwünschte Zurückspringen in Form einer unnatürlich schnellen Bewegung eines

Spielers zu einem vorherigen Standpunkt, weil der Server die Bewegung und damit aktualisierte Position des Spielers nicht rechtzeitig verarbeitet hat.[15]
Die Simulation und Darstellung des Spieles auf Seite der Clients erfolgt grundsätzlich zwischen zwei solchen Server-Ticks.

5 Technologien zur Minimierung der Probleme

Man kann jedoch nicht die Gesetze der Physik brechen und die Elektronen und Photonen, welche zur Kommunikation zwischen Server und Client genutzt werden ins unermessliche beschleunigen. Also wie kann man diese Probleme wenigstens so minimieren, dass sie kaum noch auffallen?

Wenn mehrere Spieler auf der ganzen Welt zusammenspielen, haben diese eine unterschiedlich gute Verbindung zum Server. Damit es zu keinem Ungleichgewicht kommt und einige von ihnen dadurch unfaire Vorteile oder Nachteile erhalten, muss die unterschiedliche Latenz kompensiert werden. Es gibt einige triviale Methoden um mit diesen Problemen umzugehen, welche prinzipiell jedes Onlinespiel integrieren sollte.
Dazu gehört beispielsweise eine Obergrenze für Ping und Paketverlust. Wenn Spieler einen bestimmten Wert überschreiten, sollte ihre Verbindung zum Server getrennt werden und sie sollten erst wieder mit einer stabileren Verbindung mitspielen können, da sie sonst das Spielerlebnis vieler anderer Spieler negativ beeinflussen könnten. Wenn man diese Obergrenze des Pings beispielsweise auf 200ms setzt, sollte man dabei beachten, dass es im schlimmsten Fall dazu kommen kann, dass 2 Spieler mit einem Ping von 200ms aufeinandertreffen, was in den meisten Fällen schon viel zu hoch ist und damit zu einer unangenehmen Spielerfahrung führen wird.
Das zeigt, dass es bereits für Spieler zu Problemen kommen kann, wenn die festgelegte Obergrenze noch nicht überschritten ist. Deshalb sollten Spieler mit einer schwächeren Verbindung zum Server darüber informiert werden, dass möglicherweise auftretende Probleme nicht mit dem Spiel, sondern ihrer Verbindung zusammenhängen. Dazu könnte beispielsweise ein auffälliges Symbol genutzt werden, welches beim Spieler am Bildschirmrand angezeigt wird.
Zusätzlich sollten die Spieler stets, auch bei der Auswahl des Servers ihren Ping sehen können, um über ihre Verbindung informiert zu sein und auftretende Probleme während des Spielgeschehens wahrzunehmen und möglicherweise zu beheben. Dazu könnte es ebenfalls behilflich sein, dem Spieler weitere Informationen über Paketverlust, Round Trip Time usw. zur Verfügung zu stellen.
Wichtig ist natürlich dabei auch, dass Spieler auf der ganzen Welt Zugang zu einem Server mit einer niedrigen Latenzzeit haben.
Auch Einstellungen mit der die Frequenz an ausgehender Datenkommunikation zu einem

gewissen Grad reduziert werden kann, könnte Spielern mit einer schwachen Internetverbindung behilflich sein.

5.1 Lag Compensation

Wie bereits erwähnt ist es nicht möglich all diese auftretenden Probleme vollständig zu beseitigen. Es gibt jedoch Technologien, mit denen auftretende Verzögerungen bis zu einem gewissen Maß vor den Spielern verborgen werden können, sodass der Spielfluss nicht erkennbar beeinträchtigt wird. Diese Maßnahmen zur Verminderung der Auswirkungen bezeichnet man als „Lag Compensation“.

Eine Möglichkeit wäre beispielsweise durch Kompressionstechniken die Größe der zu versendenden Daten zu verringern oder die Menge an Daten zu verdichten und zu reduzieren. Dadurch wird die Verbindung durch das Verringern der benötigten Bandbreite weniger belastet.[13]

Eine ebenfalls oft eingesetzte Technik zur Verringerung der Menge an zu versendenden Daten ist das sogenannte Interessenmanagement. Dabei werden die Updates eines Clients nur an die Spieler versandt, welche auch von diesen beeinflusst werden. Damit ist gemeint, dass Informationen über Aktualisierungen zum Beispiel nur an Spieler im selben Gebiet oder der Sichtweite des Spielers versendet werden. Alle anderen, welche nicht davon betroffen sind, erhalten diese Informationen gar nicht. Dies wird vor allem bei Online-Spielen mit einer großen Spielwelt und vielen Objekten verwendet.[13]

Außerdem kann auch die Frequenz der Übertragung reduziert und der Inhalt mehrerer Nachrichten in einer versendet werden. Das bezeichnet man als Aggregation, wodurch Protokollheader und damit auch Bandbreite eingespart wird.[13]

Ohne Lag Compensation könnte man einen Spieler in einem Echtzeit-Shooter gar nicht treffen, denn wenn man auf einen anderen Spieler zielt und schießt, schießt man aufgrund der Latenz eigentlich dahin, wo sich der Spieler vor einigen Millisekunden befand. [4]

Das kommt wie folgt zustande: Bewegt sich ein Spieler beispielsweise nach Vorn, so wird diese Bewegung erst beim Spieler ausgeführt und währenddessen an den Server gesendet. Nachdem die durch die Latenzzeit des Spielers verzögerten Daten beim Server ankommen, arbeitet dieser die Bewegung in den Hauptspielstand ein. Dadurch befindet sich der Spieler bei sich bereits schon weiter vorne als im Spielstand des Servers. Nach dem Verarbeiten und Validieren der Bewegung muss diese nun noch vom Server an die anderen Clients gesendet werden, wodurch sie nochmals um die jeweilige Latenzzeit der anderen Clients verzögert wird.

Die Lösung dafür ist, dass der Client alle Informationen über das Schussevent mit dem exakten Zeitpunkt des Ausführens an den Server sendet. Der Server rekonstruiert anschließend anhand der

übermittelten Zeitstempel die Welt zu dem Zeitpunkt in der Vergangenheit und weiß dadurch genau, wo der schießende Spieler hingezielt hat und dass sich der andere Spieler in der Vergangenheit auf dieser Position befand, welche für den Schießenden zu dem Zeitpunkt die Gegenwart war.[16]

Dadurch entsteht ein weiteres Problem, welches jedoch akzeptiert werden muss. Manchmal fühlt es sich dadurch so an, als würde man hinter einer Deckung getroffen werden. Das hängt damit zusammen, dass der schießende Spieler keinen Treffer auf der gegenwärtigen Position des getroffenen beziehungsweise verdeckten Spielers landet, sondern auf der Position, wo sich der Charakter des Spielers vor einigen Millisekunden befand. Dieses Problem wird umso deutlicher, je höher die Latenzzeit der beiden Spieler und damit die Zeit zwischen Schuss und Treffer ist. Dieses Prinzip bezeichnet man als „Favor the Shooter“ und sollte ab einem gewissen Ping bei Spielern verhindert werden, da es sich sonst für den getroffenen Spieler sehr frustrierend anfühlen kann.[17]

5.2 Interpolation

Eine Technologie, welche eingesetzt wird um bei einer reduzierten Übertragungsrate trotzdem noch ein flüssiges Bild zu erhalten nennt sich Interpolation. Denn würde man bei einer Update-Rate von 20Hz nur die Objekte an den vom Server erhaltenen Positionen darstellen, so würden Bewegungen und Animationen abgehackt und ruckelig aussehen.[18]

Bei der Interpolation werden Zwischenpakete zwischen 2 Server-Ticks berechnet. Das bedeutet, wenn der Client ein Datenpaket mit den aktuellen Informationen vom Server bekommt, wird dieses nicht direkt dargestellt. Stattdessen wird auf das nächste Datenpaket gewartet. Mit diesen zwei Paketen und dem interpolierten Zeitraum dazwischen lässt sich dann eine flüssige Darstellung garantieren.

Die Interpolation ist also eine künstliche Verzögerung, durch welche das Spiel beim Client immer etwas in der Vergangenheit stattfindet.[18] Diese Verzögerung ist jedoch so gering, dass sie für den Spieler nicht spürbar ist und zusätzlich wird sie durch den Server bei der Lag Compensation bereits mit einberechnet und hat somit keine Auswirkung auf das Spielgeschehen.[18]

Eine solche interpolierte Position eines Objektes zwischen 2 Punkten wird bei der einfachsten Variante der Interpolation, der linearen Interpolation, wie folgt berechnet:

$$position = start * (1 - time) + end * time$$

Time ist dabei ein Wert zwischen 0 und 1 und gibt an, zu welchem Zeitpunkt in der Transition man sich befindet.[13]

Wenn man diese Verzögerung verdoppelt, ist auch ein verlorenes Paket nicht mehr dramatisch,

denn es kann immer zwischen dem zuletzt empfangenen und dem vorherigen Datenpaket interpoliert werden.
Sollte jedoch in dieser Interpolationszeit mehr als ein Paket verloren gehen, lässt sich das nicht mehr kompensieren und es wird auf die Extrapolation zurückgegriffen, welche die Bewegung der Objekte anhand der bisher bekannten Positionen darstellt. Extrapolation ermöglich also die näherungsweise Abbildung von verzögerten oder ausbleibenden Paketen, welche bis zur Ankunft der echten Daten korrigierend eingefügt werden können.[19] Bei modernen Spielen werden beide Techniken parallel eingesetzt und an den entsprechend notwendigen Stellen verwendet.

5.3 Delay-based Netcode

5.3.1 Lockstep

Beim sogenannten „Deterministic Netcode“ geht es prinzipiell darum, dass jeder Spieler beziehungsweise Client denselben Spielschritt, also Tick zur selben Zeit verarbeitet. Also kein Client aufgrund einer höheren Latenz oder anderen Verzögerungen eine unterschiedliche oder ältere Version des Spielstandes sieht als der Andere.[20]
Die einfachste Variante und damit das Grundkonzept von Lockstep funktioniert so, dass nichts gemacht wird, bis alle Daten fertig übertragen wurden. Der nächste Tick erfolgt also erst dann, wenn jeder Client seinen eigenen „game tick“ abgeschlossen und seine Datenübertragung beendet hat.[20] Es handelt sich also um ein „Stop-and-Wait“-Netzwerk Protokoll.[21]
Dies hat zur Folge, dass wenn auch nur ein Client aufgrund einer hohen Latenz oder einem leistungsschwachen Computer den nächsten Tick verzögert, alle anderen Spieler warten müssen. Die Tickrate und damit auch die Update-Rate des Spiels hängen also von dem langsamsten Client ab. Das kann schnell zu einem ruckeligen Spielfluss und Verzögerungen für alle Spieler führen oder sogar dazu, dass das Spiel für einen kurzen Moment vollständig einfriert.[20]
Für Spiele, welche für den LAN-Bereich entwickelt wurden, ist dieses Konzept eine mögliche Lösung, da die Latenz dort meistens sehr niedrig ist. Bei Online-Spielen, welche über das Internet gespielt werden können und auf Schnelligkeit angewiesen sind jedoch nicht. Dort kann es durchaus zu Pings von Spielern von über 200ms kommen, wodurch die Tickrate des Spiels auf 5Hz verlangsamt werden würde.[20] Das reicht zwar für Runden basierte Strategiespiele, aber nicht für Echtzeit-Mulitplayer-Shooter, wo teilweise Tickraten von 128Hz gefragt sind.
Für dieses Problem gibt es jedoch eine Lösung, indem man einfach eine zeitliche Obergrenze festlegt, in der ein Spieler seine Aktion bzw. sein getätigtes Event übertragen haben muss. Hat es der Spieler durch Verzögerungen nicht geschafft seine Daten in dem vorgegebenen Zeitraum zu

übertragen, bekommt er auch keine Daten der anderen Spieler und diese ignorieren zusätzlich alle zu spät eintreffenden Pakete.[22]

Ursprünglich wurde das Lockstep-Protkoll von Nathaniel Baughman and Brian Levine als Synchronisations-Methode für P2P Netzwerke entwickelt, mit der das sogenannte „lookahead cheating“ verhindert werden sollte.[22] Als „lookahead cheating“ bezeichnet man das Erlangen eines unfairen Vorteils durch das Abwarten der Eingaben der anderen Spieler, auf dessen Basis man dann im Nachhinein seine eigenen Eingaben übermittelt.
Um dies zu verhindern, müssen die Clients einen kryptographisch sicheren Einweg-Hash ihrer Entscheidung angeben. Erst nachdem alle dieser Hashes eingereicht wurden, werden die Eingaben der Spieler offenbart. Sollte danach ein Spieler noch Änderungen an seinen ursprünglichen Aktionen vornehmen, fällt dies auf, wenn die „Klartext“-Entscheidung des Spielers mit dem vorher eingereichten Hash verglichen wird, da diese dann nicht mehr übereinstimmen würden. Dies ist zum Beispiel bei Echtzeit-Strategiespielen sinnvoll.[21]

5.3.2 Asynchronous Lockstep

Das „Asynchronous Lockstep Protocol“ ist eine Optimierung des Lockstep Protokolls. Dabei verwendet das Spiel standartmäßig keine Lockstep-Technologie. Jeder Spieler nimmt also unabhängig und damit eventuell asynchron von den Mitspielern seine Aktualisierungen vor. Erst in dem Moment, wo die Spieler miteinander interagieren beziehungsweise die Möglichkeit besteht, dass sie sich gegenseitig beeinflussen könnten, geht das Spiel in den Lockstep-Modus über. Durch diese Methode wird die Netzwerkleistung verbessert, da die Clients nicht von der langsameren Verbindung der anderen Clients beeinträchtigt werden, es sei denn, sie interagieren direkt mit ihnen.[20][21]

5.3.3 Deterministic Lockstep

Das Ziel, welches durch den sogenannten „Deterministic Lockstep“ erreicht werden soll ist, dass alle Clients eine Simulation ausführen, welche synchron beziehungsweise identisch zu den Simulationen der anderen Clients ist. Anstatt nun jedoch die Position, Ausrichtung, Geschwindigkeit usw. eines jeden physikalischen Objektes in der Spielwelt bei jeder Aktualisierung an alle Spieler neu zu übertragen, sollen ausschließlich die Eingaben des Spielers an alle anderen gesendet werden.[20] Jeder andere Client verarbeitet anschließend diese Eingaben und ermittelt, wie sich diese auf die Simulation auswirken. Der Sinn dahinter ist die Verringerung der Datenübertragungsgröße, da die Spielereingaben meist deutlich kleiner sind als Informationen über die gesamten Objekte der Spielwelt. Das ist vor allem bei großen Spielwelten mit vielen bewegbaren Objekten sehr sinnvoll, da der Netzverkehr durch die große Menge an Daten riesig

wäre, wenn man jede Änderung am Spielstand und in der Welt immer wieder allen Spielern mitteilen müsste. Stattdessen lässt man diese Änderungen von den Clients anhand der Eingaben der anderen Spieler selbst berechnen.

Die größte Schwierigkeit bei dieser Methode ist der namensgebende Determinismus. Damit die Spielstände der einzelnen Spieler synchronisiert bleiben, muss jeder Client die Möglichkeit haben auf Basis der ihm übertragenen Eingaben den exakt gleichen Spielstand zu produzieren.[20] Schon die kleinsten Abweichungen in den Berechnungen können zu größeren Unterschieden und damit desynchronisierten Simulationen zwischen den Spielern führen. Das Hauptproblem hierbei nennt man „Floating Point Determinism“ und stellt vor allem bei Spielen, welchen ein unterschiedliches Betriebssystem oder unterschiedliche Prozessoren zugrunde liegt ein Problem dar, da beispielsweise verschiedene Prozessorarchitekturen kleine Unterschiede in Berechnungen bei komplexen physikalischen Simulationen aufweisen können. Es erfordert deshalb oft viel Arbeit, die gleichen Ergebnisse bei gleichen Eingaben auf unterschiedlichen Maschinen zu erhalten.[23]

5.3.4 Input Delay

Der größte Vorteil welchen „Deterministic Lockstep“ bietet ist, dass es das „Input-Delay“ ermöglicht, also eine künstlich erzeugte Eingabeverzögerung. Dies ist die einfachste und gebräuchlichste Implementierung von Lockstep-Technologien in Spielen. Wie bereits erwähnt warten die Spieler bei der Lockstep-Technologie auf die Eingabedaten der anderen. Wenn diese aktuellen Eingaben jedoch für einen zukünftigen Tick geplant werden, können in der Zwischenzeit andere Ticks verarbeitet werden, ohne dass auf die Eingaben der anderen Spieler gewartet werden muss.[20]

Solange die Eingabedaten vor der geplanten Zeit eintreffen, muss nicht mehr auf diese gewartet werden und somit gibt es keine Verzögerung und das Spiel kann mit einer so schnellen Tickrate wie nötig laufen.[20]

Ein Beispiel hierfür wäre eine Eingabeverzögerung von 3 Ticks. Die Eingaben werden übertragen und für 3 Ticks zwischengespeichert, bevor sie verwendet werden. Dadurch ist mehr Zeit vorhanden bis die Daten übertragen sein müssen. Wenn also Eingaben eines Remote-Spielers verspätet ankommen, weil sie bei der Übertragung durch eine Verzögerung beeinflusst werden, werden die Eingaben des lokalen Spielers um die gleiche Zeitspanne verzögert. Damit werden beide Eingaben zum gleichen Zeitpunkt verarbeitet und somit im gleichen Tick ausgeführt.[24]

5.3.5 Das Problem

Wenn es zu Netzwerkproblemen oder Verbindungsschwankungen kommt, welche durchaus nicht unüblich sind und oft auftreten können, ist das sehr problematisch für den delay-based Netcode.

Das könnte beispielsweise dazu führen, dass die Eingaben nicht in der vorher eingeplanten Zeit ankommen und das Spiel pausiert wird, was sich durch Ruckeln bemerkbar macht, weil erst noch auf die Eingaben des Spielers für das Fortsetzen der Simulation gewartet werden muss.[24]
Man könnte natürlich die Eingabeverzögerung dynamisch an die Netzwerkbedingungen anpassen. Diese Netzwerkschwankungen sind jedoch schwierig vorherzusagen und bevor die künstliche Verzögerung angepasst werden kann, ist es meist schon zu spät. Außerdem kann eine zu hohe oder inkonstante Eingabeverzögerung dazu führen, dass es für den Spieler zu einem deutlich spürbaren Input Lag kommt.[24]
Der Grund warum es trotzdem noch von manchen verwendet wird ist, dass es im Gegensatz zu anderen Technologien relativ einfach zu implementieren ist, weil die Logik bereits bestehender Offline-Modi der Spiele meist nicht großartig verändert werden muss.

5.4 Rollback Netcode

Anstatt wie beim delay-based Netcode das Spiel zu pausieren, wenn die Eingaben eines Spielers nicht rechtzeitig beziehungsweise verzögert eintreffen, wird beim Rollback Netcode das Spiel weiter simuliert. Wenn diese Eingaben dann irgendwann eintreffen, wird die Simulation zu dem Zeitpunkt zurückgespult, wo der Spieler die Eingabe getätigt hat. Dort werden dann die richtigen Eingaben angewendet und das neue, überarbeitete Ergebnis wird dem Spieler gezeigt.[20]
Frames werden bei Videospielen als ein Maß für die Zeit verwendet. Bei einer Frame-Rate von 60 Frames pro Sekunde entspricht ein Frame 16,66ms. Dieses Wissen wird benötigt, um das nachfolgende Beispiel verstehen zu können.[21]
Wenn ein Spieler A bei Frame 1 seine Eingabe tätigt, beginnt sein Charakter direkt mit der Animation, welche 8 Frames lang andauern wird. Bei Frame 3 kommt die Eingabe von Spieler A bei Spieler B an. Dieser realisiert, dass er 3 Frames hinter Spieler A liegt. In Frame 4 spult Spieler B also die Simulation zu Frame 1 zurück und simuliert die ersten 3 Frames innerhalb eines Frames und fährt mit der Animation von Spieler A zum aktuellen Zeitpunkt fort. In den Frames 5-8 läuft nun also die Animation von Spieler A's Charakter bei beiden Spielern synchron ab.[21]
Um wie viele Frames das Spiel des lokalen Spielers zurückgespult werden muss, also wie viele Frames der Remote-Spieler voraus ist, wird bei einer von Tony Cannon, für P2P-Netzwerke entwickelten Middleware namens GGPO (Good Game, Peace Out) wie folgt berechnet [21]:

$$Last\ received\ Remote\ Frame + \left(\frac{Ping * Frame\ Frequency}{2}\right) - Last\ Local\ Frame$$

5.4.1 Client-side-prediction

Damit das Spielgeschehen trotzdem flüssig abläuft und alle Spielereingaben so gehandhabt werden können, als würden sie lokal geschehen, werden die Eingaben der anderen Remote-Spieler von jedem Client vorhergesagt. Dieses Konzept nennt sich „client-side-prediction“.[13]

Das Spiel verarbeitet also die Ticks mit Hilfe von Vorhersagen darüber, wie die Eingaben der Spieler aussehen könnten. Wenn die richtigen Eingaben eintreffen und von den Vorhergesagten abweichen, wird die Simulation zurückgespult und unter Einbeziehung der tatsächlichen Eingabedaten neu berechnet. Wenn diese Vorhersagen jedoch korrekt waren, fällt gar nicht auf, dass es Verbindungsprobleme gab.[24]

Diese Vorhersagen funktionieren so, dass bei dem Fehlen von Eingabedaten von Spielern basierend auf den vergangenen Aktionen die wahrscheinlichste nächste Aktion bestimmt wird. Eine sehr häufig verbreitete Methode ist es, dass die letzte getätigte Eingabe des Spielers für den aktuellen Frame wiederholt wird. Wenn ein Spieler also eine Taste gedrückt hielt, geht das Spiel davon aus, dass der Spieler die Taste immer noch gedrückt hält und das Spiel wird mit dieser getätigten Vorhersage weiter simuliert. Diese Variante wird auch bei der GGPO-Technologie eingesetzt.[21][24]

Da diese Vorhersagen natürlich auch komplett falsch sein können, kann es dazu kommen, dass die korrigierte Simulation stark von der Vorhergesagten abweicht, wodurch es zu Fehlern, sogenannten „Glichtes“ kommen kann. Dazu zählt beispielsweise das bereits angesprochene Rubberbanding oder dass Spieler plötzlich an einen anderen Ort teleportiert werden. Das geschieht dadurch, dass zum Beispiel ein autoritativer Server die fehlerhaft vorhergesagte Simulation des Spielers korrigiert.[20] Diese Fehler, wie das Teleportieren der Charaktere kommt jedoch durch den Einsatz der häufig verwendeten Technologie GGPO nahezu nie vor. Außerdem liegt die Update-Rate von Servern heutzutage oft bei über 60Hz, wodurch Vorhersagen nur für Bruchteile einer Sekunde getroffen werden müssen und die daraus folgenden Korrekturen gleichermaßen mikroskopisch sind.[13]

Umso länger der Zeitraum ist, über den die Frames vorhergesagt werden müssen, desto größer ist natürlich die Wahrscheinlichkeit, dass diese nicht mehr stimmen. Um also die kurzfristig und fehlerhaft vorhergesagten Spielschritte zu minimieren, kann man zusätzlich noch eine geringe Eingabeverzögerung einbauen, mit dem Unterschied, dass das Spiel nicht pausiert wird, wenn die Eingaben länger benötigen also für die Verzögerung vorgesehen ist, sondern die Simulation mit den Vorhersagen einfach weiterläuft. Außerdem führt eine höhere Tick-Rate des Servers dazu, dass die Zeitabstände welche von den Clients vorhergesagt werden müssen kürzer werden.[20]

Problematisch wird es also erst, wenn die Eingaben des Spielers durch eine zu hohe Latenzzeit verzögert werden und der Zeitraum, wo Vorhersagen getroffen werden müssen sehr groß wird.

Diese „client-side-prediction“ findet noch eine andere Verwendung, nämlich beim lokalen Spieler selbst. Möchte dieser nämlich von der Position (10, 10) einen Schritt nach rechts gehen, würde er dem Server dies mitteilen. Der Server würde dann diese Eingabe bearbeiten, eine Simulation ausführen, die Position des Spielers auf (11, 10) setzen und den Client über seine neue Position informieren. Bei einem Ping von 100ms zwischen Server und Client, würden dann auch mindestens 100ms zwischen Tastendruck und Antwort des Servers und damit dem Start der Bewegungsanimation des Spielers vergehen. Stattdessen kann der Spieler nach dem Tätigen und Übertragen seiner Eingabe seine eigene vorhergesagte Simulation starten, nämlich dass er nachdem er die Taste gedrückt hat auch einen Schritt nach rechts geht.[25] Dadurch muss er nicht darauf warten, dass der Server die Eingabe verarbeitet und es gibt keine Eingabeverzögerung mehr. Der Server validiert diese Eingabe trotzdem, indem er seine eigene Simulation mit ihr durchführt und den Client den autoritativen Hauptspielstand übermittelt, welcher gegebenenfalls den vorhergesagten, also client-seitigen Spielstand korrigiert. Wichtig hierbei ist, dass der Client zusätzlich zu seinem Request eine Sequenznummer mitliefert und der Server die Sequenznummer der letzten verarbeiteten Eingabe mitsendet. Dadurch kann der Client den jeweiligen Request mit der zugehörigen Antwort des Servers vergleichen und es können Fehler vermieden werden. Man spricht dabei von Server Reconciliation.[25] Sollte der Client irgendwelche unerlaubten Änderungen an seinem Spielstand vornehmen, wird das Spielgeschehen der anderen Clients davon nicht beeinflusst, da dieser unabhängig vom Spielstand des Servers ist.
Die größte Anforderung bei Rollback-Technologien ist, dass das Spiel verschiedene Spielzustände speichern und laden sowie die Spiellogik im Hintergrund simulieren können muss. Das alles muss zusätzlich noch extrem schnell gehen. So kann es nämlich vorkommen, dass die Berechnung und Simulation mehrerer Frames in einem einzelnen Frame rechtzeitig abgeschlossen werden müssen. Dadurch ist oft eine enorme Rechenleistung notwendig.[20][24]

6 Praxisbeispiel

Es handelt sich hierbei nur um ein kleines Beispiel, welches die Komplexität dieses Themas keinesfalls annähernd vollständig darstellt. Stattdessen soll es anhand eines einfachen Beispiels zeigen, wie eine Verzögerung ausgeglichen werden kann und wie die Kommunikation der Spieler beziehungsweise Clients mit einem Server aussehen könnte. Aus Gründen der Anschaulichkeit wurden hierfür extra hohe künstliche Verzögerungen erzeugt. Zusammengefasst umfasst das folgende Programm die Bewegung mehrerer Spieler in einem einfachen 2-dimensionalen Raum mit einem simulierten hohen Ping, welcher durch eine client-side-predicition der Bewegung des lokalen Spielers, sowie einem Input-Delay kompensiert wird. Der gesamte Programmcode zu diesem Beispiel befindet sich im Anhang.

```
const express = require("express");
const http = require("http");
const socketio = require("socket.io");

const app = express();
const server = http.createServer(app);
const io = socketio(server);

server.listen(3000, "192.168.178.104", () => {
  console.log("listening on Port: 3000");
});
```

Vorerst wird ein Node.js Webserver basierend auf dem HTTP-Modul und dem Framework Express.js auf dem Port 3000 gestartet. Dieser Arbeitet außerdem mit dem WebSocket-Protokoll. Dieses Netzwerkprotokoll basiert auf TCP und ermöglicht es eine bidirektionale Verbindung zwischen dem eben gestarteten Webserver und der Webanwendung herzustellen. Die Kommunikation startet über einen HTTP-Handshake und wird dann über WebSockets fortgeführt. Dadurch bleibt die Verbindung für den gesamten Zeitraum „geöffnet“, wodurch der Server die ganze Zeit die Möglichkeit hat Daten an die Clients zu senden, ohne dass diese ständig Anfragen zur Verfügbarkeit neuer Daten stellen müssen.[26][27] Das eignet sich besonders gut für Browsergames und erspart das entwickeln eines auf UDP basierenden Custom-Protokolls an dieser Stelle.

```
io.on("connection", (socket) => {
  socket.on("connectedToGame", () => {
    const x = Math.floor(Math.random() * 981)
    const y = Math.floor(Math.random() * 881)
    players.push(new Player(socket.id, x, y));
    socket.emit("setBeginningGameState", players);
  });
```

```
  socket.on("disconnect", () => {
    const index = players.findIndex((player) => player.id === socket.id);
    if (index !== -1) {
      players.splice(index, 1)[0];
    }
  });
});
```

Diese WebSockets basieren auf Events. Verbindet sich ein Client mit dem zuvor erstellten autoritativen Server,

```
const socket = io("http://192.168.178.104:3000/", {
  transports: ["websocket"],
});
socket.emit("connectedToGame");
```

so wird mit einer zufällig erzeugten Position auf dem Canvas und seiner Socket-Id ein „Player“-Objekt erstellt, welches in einem Array gespeichert wird. Dem verbundenen Spieler werden anschließend durch das übertragen dieses Arrays alle notwendigen Informationen über die anderen Spieler und damit der aktuelle Spielstand übertragen. Trennt ein Client seine Verbindung, so wird dieser Spieler aus dem Array entfernt.

Beim Spieler wird dann das Spiel wie folgt eingerichtet:

```
socket.on("setBeginningGameState", (data) => {
  localSocketId = socket.id;
  players = data;
  updatePlayers();
});

function updatePlayers() {
  ctx.fillStyle = "grey";
  ctx.fillRect(0, 0, canvas.width, canvas.height);
  players.forEach((player) => draw(player));
}

function draw(player) {
  ctx.beginPath();
  ctx.rect(player.x, player.y, playerSize, playerSize);
  ctx.stroke();
  ctx.fillStyle = "blue";
  ctx.fill();
}
```

Möchte ein Client seine Position ändern, so sendet er seine getätigte Eingabe an den Server, wo sie verarbeitet wird und die Position des Spielers aktualisiert und an alle Clients gesendet wird. Erst dann ändert sich auch die Position des lokalen Spielers.

Client:

```
window.addEventListener("keydown", function (event) {
  socket.emit("playerMoved", { key: event.key, currentPlayerId: localSocketId});
});
socket.on("update", (data) => {
  players = data;
  updatePlayers();
});
```

Server:

```
  socket.on("playerMoved", (data) => {
    setTimeout(() => {
      const index = players.findIndex((player) => player.id === socket.id);
      if(index != -1){
        switch (data.key) {
          case "ArrowLeft":
            players[index].moveLeft();
            break;
          case "ArrowUp":
            players[index].moveUp();
            break;
          case "ArrowRight":
            players[index].moveRight();
            break;
          case "ArrowDown":
            players[index].moveDown();
            break;
        }
        update();
      }
    }, 500); //Ping 500ms
  });

function update() {
  io.emit("update", players);
}
```

Durch ein künstlich eingebautes Timeout von 500ms, wird ein Ping zwischen Client und Server von 500ms simuliert. Dadurch dauert es auch mindestens 500ms zwischen Tastenanschlag und Bewegung des Spielers, was ein reales Spiel unspielbar machen würde.

Dies lässt sich verhindern, indem man eine clientseitige Vorhersage des Spielstandes nach der Eingabe tätigt und diese anschließend mit der zugehörigen Antwort des Servers vergleicht. Dazu

wird die Eingabe beim Spieler verarbeitet und die daraus resultierende Position und mit einer dazugehörigen Sequenznummer gespeichert. Erhält der Spieler die Antwort des Servers, vergleicht er seine vorhergesagte Position mit dem Hauptspielstand des Servers. Weichen die Simulationen voneinander ab, wird seine Position korrigiert. Dadurch ist keine Verzögerung mehr nach der Eingabe vorhanden.

Hierfür wurden einige Veränderungen am vorherigen Code vorgenommen:

Server:

```
function update(id, sequenznummer) {
  io.emit("update", { players: players, updatedPlayerId: id, sequenznummer });
}
```

Client:

```
window.addEventListener("keydown", function (event) {
  sequenznummer++;
  socket.emit("playerMoved", { key: event.key, sequenznummer: sequenznummer });
  const index = players.findIndex((player) => player.id === localSocketId);
  switch (event.key) {
    case "ArrowLeft":
      moveLeft(players[index]);
      break;
    case "ArrowUp":
      moveUp(players[index]);
       break;
    case "ArrowRight":
      moveRight(players[index]);
      break;
    case "ArrowDown":
      moveDown(players[index]);
      break;
  }
  playerMoves[sequenznummer] = { x: players[index].x, y: players[index].y };
  updateLocalPlayer(players[index]);
});

socket.on("update", (data) => {
  if (data.updatedPlayerId == localSocketId && sequenznummer != 0) {
    const index = data.players.findIndex((player) => player.id === localSocketId);
    if (data.players[index].x == playerMoves[data.sequenznummer].x &&
data.players[index].y == playerMoves[data.sequenznummer].y) {
      for (let i = 0; i < data.players.length; i++) {
        if (data.players[i].id != localSocketId) {
          players[i] = data.players[i];
        }
      }
```

```
    } else {
      players = data.players;
    }
  } else {
    players = data.players;
  }
  updatePlayers();
});
function updateLocalPlayer(player) {
  ctx.fillStyle = "red";
  ctx.fillRect(player.x - playerSize, player.y - playerSize, 60, 60);
  draw(player);
}
```

Bei einem niedrigeren Ping von beispielsweise 200ms (welcher natürlich für ein reales Spiel immer noch viel zu hoch wäre), kann zusätzlich ein Input-Delay, also eine Eingabeverzögerung eingebaut werden, damit nun auch die Verzögerung zwischen lokalem Spieler und den Remote-Spielern ausgeglichen wird.

```
socket.emit("playerMoved", { key: event.key, sequenznummer: sequenznummer });
  setTimeout(() => {
       .
       .
       .
  }, 200);
```

Dazu wird einfach ein Timeout nach dem Senden der getätigten Eingabe und dem Ausführen dieser beim lokalen Spieler gesetzt. Der Wert von 200ms ist natürlich viel zu hoch und dient nur zu Demonstrationszwecken. In Wirklichkeit würde man hier beispielsweise eine Eingabeverzögerung von 50ms wählen, damit der Server zum Beispiel mehr Zeit hat um diese Eingaben zu verarbeiten oder dass beim Ausbleiben von Eingaben die Zeitspannen kürzer werden, in denen er die Eingaben vorhersagen muss oder dass bei dem Einsatz einer Lockstep Technologie mehr Zeit bleibt, bis die Eingaben beim Server sein müssen. Dabei wäre es jedoch besser die Eingaben auf dem Server zwischenzuspeichern und damit dort zu verzögern. Wenn zwei Spieler gleichzeitig eine Taste drücken, wird die Animation durch die künstlich verzögerte Eingabe auch gleichzeitig bei beiden Spielern ausgeführt. Ohne eine Eingabeverzögerung und mit einem sich ständig ändernden Ping würde die Animation erst beim lokalen Spieler und kurze Zeit später beim Remote-Spieler ausgeführt werden. Ein weiterer Vorteil ist, dass diese Eingabeverzögerung dynamisch angepasst und vor allem kontrolliert werden kann. Die restliche jetzt noch übrig gebliebene Latenz würde dann, wenn nötig, durch Lag Compensation oder eine andere vorher vorgestellte Technologie ausgeglichen werden. Da die Spieler hier aber nicht direkt miteinander interagieren, ist das zu dem Zeitpunkt noch nicht relevant.

Zusätzlich kann eine stetige Latenzzeit-Kontrolle durchgeführt werden. Bei einem Ping von über 400ms wird die Verbindung zwischen Server und Client unterbrochen.

Client:

```
setInterval(function () {
  startTime = Date.now();
  socket.emit("ping");
}, 2000);
socket.on("pong", function () {
  let latency = Date.now() - startTime;
  if (latency >= 400) {
    socket.emit("highPing", localSocketId);
  }
  document.getElementById("ping").innerText = "Ping: " + latency
});
```

Server:

```
socket.on("ping", () => {
  socket.emit("pong");
});
socket.on("highPing", (id) => {
  io.sockets.sockets.forEach((socket) => {
     if (socket.id === id) socket.disconnect(true);
  });
})
```

Dieser ermittelte Ping könnte außerdem dazu verwendet werden, um die notwendige Eingabeverzögerung dynamisch anzupassen.

7 Zusammenfassung

In dieser Arbeit wurde eines, wenn nicht sogar das komplexeste und anspruchsvollste Themengebiet der Videospieleentwicklung dargestellt. Dazu wurden erst die Grundlagen aufgezeigt, wie die Clients in Videospielen miteinander vernetzt sind und kommunizieren. Anschließend wurden verschiedene Probleme dargestellt, welche bei der Datenübertragung auftreten können und wie diese das Spielgeschehen beeinflussen können und zum Schluss wie diese Probleme vermindert werden können.

Hierbei ist wichtig zu erwähnen, dass nicht alle möglichen Technologien dargestellt wurden, sondern nur die Grundkonzepte, welche den meisten dieser zugrunde liegen. Jedes Videospiel nutzt unterschiedliche Techniken und die Gewichtung des Einsatzes der hier dargestellten Techniken ist ebenfalls sehr individuell.

Mithilfe des hier vermittelten Wissens sollte es möglich sein, erste eigene Multiplayer-Anwendungen sowie Techniken zur Optimierung dieser zu entwickeln. Diese hier vorgestellten Grundlagen sollen das Fundament für das Verständnis bieten, was die Entwicklung von Echtzeit-Anwendungen, wie schnelle Mulitplayer-Videospiele so anspruchsvoll macht und wie diese funktionieren.

Es wurde festgestellt, dass viele Methoden und Techniken für unterschiedliche Sachen besser geeignet sind als andere, dafür aber eventuell auch kostenintensiver sind und mehr Rechenleistung erfordern. So ist ein Client-Server-Netzwerkmodell für ein optimales Spielerlebnis besser geeignet als ein P2P-Netzwerk, jedoch ist es auch deutlich kostenintensiver und die Implementierung von Rollback ist deutlich anspruchsvoller und arbeitsintensiver als Lockstep, weshalb es vor allem für einzelne Spieleentwickler sinnvoller sein kann, auf die kostengünstigere Variante zurückzugreifen.

All das zeigt, dass es sehr situationsbedingt ist und viele Faktoren, wie beispielsweise die Art des Videospiels eine große Rolle bei der Entscheidung spielen, welche Technologien der hier vorgestellten eingesetzt werden sollten, um den Netcode des Videospiels zu optimieren.

Schlussendlich lässt sich jedoch sagen, dass das Networking eine entscheidende Rolle bei der Entwicklung von Mulitplayer-Spielen spielt. Ein nicht gut durchdachter Netcode mit ständigen Problemen macht ein Videospiel ungenießbar und teilweise auch unspielbar.

Quellenverzeichnis

[1] *Wikipedia – Netzcode [Online]* – Available at: https://de.wikipedia.org/wiki/Netzcode, Stand 14. April 2022

[2] *Wikipedia – Netcode [Online]* – Available at: https://en.wikipedia.org/wiki/Netcode, Stand 14. April 2022

[3] *Medium, Yuan Gao (Meseta) – Netcode Concepts Part 2: Topology (Netcode Konzepte Teil 2: Topologie) [Online]* – Available at: https://meseta.medium.com/netcode-concepts-part-2-topology-ad64f9f8f1e6, Stand 19. April 2022

[4] *PCGamer, Chris „Battle(non)sense“ – How netcode works, and what makes ‚good‘ netcode (Wie Netcode funktioniert und was guten Netcode ausmacht)* – Available at: https://www.pcgamer.com/netcode-explained/, Stand 19. April 2022

[5] *Gabriel Gambetta – Fast-Paced Multiplayer (Part I): Client-Server Game Architecture (Schneller Mehrspielermodus (Teil I): Client-Server-Spielarchitektur) [Online]* – Available at: https://www.gabrielgambetta.com/client-server-game-architecture.html, Stand 19. April 2022

[6] *Pvigier's blog, Pierre – Beginner's Guide to Game Networking (Anleitung für den Einstieg in das Networking von Videospielen) [Online]* – Available at: https://pvigier.github.io/2019/09/08/beginner-guide-game-networking.html, Stand 20. April 2022

[7] *YoutTube, Florian Dalwigk – TCP vs. UDP Die Unterschiede der beiden Protokolle [Online]* – Available at: https://www.youtube.com/watch?v=v8_mM7zq8GE, Stand 20. April 2022

[8] *GafferOnGames, Glenn Fiedler – UDP vs. TCP [Online]* – Available at: https://gafferongames.com/post/udp_vs_tcp/, Stand 20. April 2022

[9] *Wikipedia – Ping (Datenübertragung) [Online]* – Available at: https://de.wikipedia.org/wiki/Ping_(Daten%C3%BCbertragung), Stand 21. April 2022

[10] *Wikipedia – Lag [Online]* – Available at: https://de.wikipedia.org/wiki/Lag, Stand 21. April 2022

[11] *Medium, Yuan Gao (Meseta) – Netcode Concepts Part 1: Introduction (Netcode Konzepte Teil 1: Einleitung) [Online]* – Available at: https://meseta.medium.com/netcode-concepts-part-1-introduction-ec5763fe458c, Stand 21. April 2022

[12] *YouTube, „Battle(non)sense" – Netcode 101 – What You Need To Know (Netcode 101 – Was man wissen muss) [Online]* – Available at: https://www.youtube.com/watch?v=hiHP0N-jMx8, Stand 21. April 2022

[13] *HTWSaar, „Maverick Studer" – Netcode – Networking in Online Games [Online]* – Available at: https://stl.htwsaar.de/tr/STL-TR-2021-02.pdf, Stand 22. April 2022

[14] *Wikipedia – Tickrate [Online]* – Available at: https://de.wikipedia.org/wiki/Tickrate, Stand 22. April 2022

[15] *QRPG – Was ist: Rubberbanding [Online]* – Available at: https://www.qrpg.de/rubberbanding/, Stand 22. Apirl 2022

[16] *Gabriel Gambetta – Lag Compensation [Online]* – Available at https://www.gabrielgambetta.com/lag-compensation.html, Stand 25. April 2022

[17] *YouTube, PlayOverwatch – Netcode in Overwatch [Online]* – Available at https://www.youtube.com/watch?v=vTH2ZPgYujQ, Stand 25. April 2022

[18] *CS-Scene – Interpolation [Online]* – Available at: https://cs-scene.de/counter-strike-global-offensive/configs-scripts/interpolation/, Stand 25. April 2022

[19] *Valve – Source Multiplayer Networking [Online]* – Available at: https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking, Stand 25. April 2022

[20] *Medium, Yuan Gao (Meseta) – Netcode Concepts Part 3: Lockstep and Rollback (Netcode Konzepte Teil 3: Lockstep and Rollback) [Online]* – Available at: https://meseta.medium.com/netcode-concepts-part-1-introduction-ec5763fe458c, Stand 25. April 2022

[21] *HKR, Martin Huynh u. Fernando Valarino - An analysis of continuous consistency models in real time peer-to-peer fighting games [Online]* – Available at: http://hkr.diva-portal.org/smash/get/diva2:1322881/FULLTEXT01.pdf, Stand 25. April 2022

[22] *Wikipedia – Lockstep protocol [Online]* – Available at: https://en.wikipedia.org/wiki/Lockstep_protocol, Stand 25. April 2022

[23] *GafferOnGames, Glenn Fiedler – Floating Poing Determinism [Online]* – Available at: https://gafferongames.com/post/floating_point_determinism/, Stand 26. April 2022

[24] *Arstechnica, Ricky Pusch - Explaining how fighting games use delay-based and rollback netcode [Online]* – Available at: https://arstechnica.com/gaming/2019/10/explaining-how-fighting-games-use-delay-based-and-rollback-netcode/, Stand 26. April 2022

[25] *Gabriel Gambetta – Client-Side Prediction and Server Reconciliation [Online]* – Available at: https://www.gabrielgambetta.com/client-side-prediction-server-reconciliation.html, Stand 26. April 2022

[26] *Wikipedia – WebSocket [Online]* – Available at: https://de.wikipedia.org/wiki/ WebSocket, Stand 27.04.2022

[27] *Medium, Srushtika Neelakantam – Building a realtime multiplayer browser game in less than a day – Part 2/4 [Online]* – Available at: https://medium.com/swlh/building-a-realtime-multiplayer-browser-game-in-less-than-a-day-part-2-4-f1f109761cf3, Stand 27.04.2022

Abbildungsverzeichnis

Selbstständigkeitserklärung

Ich versichere, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form weder veröffentlicht, noch einer anderen Prüfungsbehörde vorgelegt.

_____ _____

Nikolaidis, Simon Ort, Datum

Anhang

Vollständiger Programmcode des Praxisbeispiels:

Server (server.js):

```
const express = require("express");
const http = require("http");
const socketio = require("socket.io");

const app = express();
const server = http.createServer(app);
const io = socketio(server);

let players = [];
let LatencyTimeout;

const canvasHeight = 900;
const canvasWidth = 1000;
const playerSize = 20;

class Player {
  constructor(id, x, y) {
    this.id = id;
    this.x = x;
    this.y = y;
  }

  moveLeft() {
    if (this.x > 0) {
      this.x -= 10;
    } else {
      this.x = canvasWidth;
    }
  }

  moveRight() {
    if (this.x < canvasWidth - playerSize) {
      this.x += 10;
    } else {
      this.x = 0 - playerSize;
    }
  }

  moveUp() {
    if (this.y > 0) {
      this.y -= 10;
    } else {
```

```
43          this.y = canvasHeight;
44        }
45      }
46
47      moveDown() {
48        if (this.y < canvasHeight - playerSize) {
49          this.y += 10;
50        } else {
51          this.y = 0 - playerSize;
52        }
53      }
54    }
55
56    io.on("connection", (socket) => {
57      console.log("a user connected");
58
59      socket.on("connectedToGame", () => {
60        const position = { x: Math.floor(Math.random() * 981), y:
61    Math.floor(Math.random() * 881) };
62        players.push(new Player(socket.id, position.x, position.y));
63        socket.emit("setBeginningGameState", players);
64      });
65
66      socket.on("playerMoved", (data) => {
67        LatencyTimeout = setTimeout(() => {
68          const index = players.findIndex((player) => player.id === socket.id);
69          if (index != -1) {
70            switch (data.key) {
71              case "ArrowLeft":
72                players[index].moveLeft();
73                break;
74
75              case "ArrowUp":
76                players[index].moveUp();
77                break;
78
79              case "ArrowRight":
80                players[index].moveRight();
81                break;
82
83              case "ArrowDown":
84                players[index].moveDown();
85                break;
86            }
87            update(socket.id, data.sequenznummer);
88          }
89        }, 200); //Ping 200ms
90      });
91
```

```
92      socket.on("ping", () => {
93        socket.emit("pong");
94      });
95
96      socket.on("highPing", (id) => {
97        io.sockets.sockets.forEach((socket) => {
98          if (socket.id === id) socket.disconnect(true);
99        });
100     });
101
102     socket.on("disconnect", () => {
103       clearTimeout(LatencyTimeout);
104       console.log("user disconnected " + socket.id);
105       const index = players.findIndex((player) => player.id === socket.id);
106       if (index !== -1) {
107         players.splice(index, 1)[0];
108       }
109     });
110   });
111
112   function update(id, sequenznummer) {
113     io.emit("update", { players: players, updatedPlayerId: id, sequenznummer });
114   }
115
116   server.listen(3000, "192.168.178.104", () => {
117     console.log("listening on Port: 3000");
118   });
```

Client (index.html):

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Game</title>
</head>
<body>
    <canvas id="gameCanvas"></canvas>
    <p id="ping">Ping:</p>
    <script src="node_modules/socket.io/client-dist/socket.io.js"></script>
    <script src="main.js"></script>
</body>
</html>
```

Client (main.js):

```
const socket = io("http://192.168.178.104:3000/", {
  transports: ["websocket"],
});

socket.emit("connectedToGame");

const canvas = document.getElementById("gameCanvas");
const ctx = canvas.getContext("2d");

canvas.width = 1000;
canvas.height = 900;

const playerSize = 20;

let localSocketId;
let players = [];
let playerMoves = [];
let sequenznummer = 0;
let startTime;
sequenznummer++;

window.addEventListener("keydown", function (event) {
  socket.emit("playerMoved", { key: event.key, sequenznummer: sequenznummer });
  setTimeout(() => {
    const index = players.findIndex((player) => player.id === localSocketId);
    switch (event.key) {
      case "ArrowLeft":
        moveLeft(players[index]);
        break;

      case "ArrowUp":
        moveUp(players[index]);
        break;

      case "ArrowRight":
        moveRight(players[index]);
        break;

      case "ArrowDown":
        moveDown(players[index]);
        break;
    }
    playerMoves[sequenznummer] = { x: players[index].x, y: players[index].y };
    updateLocalPlayer(players[index]);
    sequenznummer++;
  }, 180);
});
```

```
48
49  socket.on("setBeginningGameState", (data) => {
50    localSocketId = socket.id;
51    players = data;
52    updatePlayers();
53  });
54
55  socket.on("update", (data) => {
56    if (data.updatedPlayerId == localSocketId && sequenznummer != 0) {
57      const index = data.players.findIndex((player) => player.id ===
58  localSocketId);
59      if (data.players[index].x == playerMoves[data.sequenznummer].x &&
60  data.players[index].y == playerMoves[data.sequenznummer].y) {
61        for (let i = 0; i < data.players.length; i++) {
62          if (data.players[i].id != localSocketId) {
63            players[i] = data.players[i];
64          }
65        }
66      } else {
67        players = data.players;
68      }
69    } else {
70      players = data.players;
71    }
72    updatePlayers();
73  });
74
75  function updatePlayers() {
76    ctx.fillStyle = "grey";
77    ctx.fillRect(0, 0, canvas.width, canvas.height);
78    players.forEach((player) => draw(player));
79  }
80
81  function updateLocalPlayer(player) {
82    ctx.fillStyle = "grey";
83    ctx.fillRect(player.x - playerSize, player.y - playerSize, 60, 60);
84    draw(player);
85  }
86
87  function draw(player) {
88    ctx.beginPath();
89    ctx.rect(player.x, player.y, playerSize, playerSize);
90    ctx.stroke();
91    ctx.fillStyle = "blue";
92    ctx.fill();
93  }
94
95  function moveLeft(player) {
96    if (player.x > 0) {
```

```
      player.x -= 10;
    } else {
      player.x = canvas.width;
    }
  }

  function moveRight(player) {
    if (player.x < canvas.width - playerSize) {
      player.x += 10;
    } else {
      player.x = 0 - playerSize;
    }
  }

  function moveUp(player) {
    if (player.y > 0) {
      player.y -= 10;
    } else {
      player.y = canvas.height;
    }
  }

  function moveDown(player) {
    if (player.y < canvas.height - playerSize) {
      player.y += 10;
    } else {
      player.y = 0 - playerSize;
    }
  }

  //Ping-Time
  setInterval(function () {
    startTime = Date.now();
    socket.emit("ping");
  }, 2000);

  socket.on("pong", function () {
    let latency = Date.now() - startTime;
    if (latency >= 400) {
      socket.emit("highPing", localSocketId);
    }
    document.getElementById("ping").innerText = "Ping: " + latency;
  });
```