Automatisiertes Testen von *Lag*-Kompensationalgorithmen in Netzwerkspielen

Fabian Meisinger

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im September 2013

Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Hagenberg, am 30. September 2013

Fabian Meisinger

Inhaltsverzeichnis

Kurzfassung

Netzwerkverzögerungen sind ein Problem, mit dem die meisten Onlinespiele zu kämpfen haben. Aus diesem Grund werden Algorithmen eingesetzt, die den Auswirkungen auf das Spielgeschehen entgegenwirken sollen. Wie wirksam die eingesetzten Maßnahmen sind, wird entweder durch groß angelegte Tests mit Endnutzern oder gar nicht überprüft.

Nach einem ausführlichen Einblick in die Problematik wird im Rahmen dieser Arbeit eine Methode vorgestellt, die es vergleichsweise einfach ermöglicht, die Funktionalität und Qualität solcher *Lag*-Kompensationstechniken zu testen. Es wird der Grundstein für ein *Framework* gelegt, welches die Simulation von verschiedenen Netzwerkbedingungen ermöglicht und mit Hilfe von automatisierten Tests verschiedene Spielsituationen nachstellen kann. Anhand eines Testspiels wird die Funktionsfähigkeit der vorgeschlagenen Methode demonstriert und ein Einblick in die generierten Testergebnisse geboten.

Die Arbeit richtet sich vorwiegend an Spielentwickler mit Erfahrung in der Netzwerkprogrammierung und Forscher, die sich mit der Thematik der *Lag*-Kompensation auseinandersetzen.

Abstract

Network delays are a problem many online games have to cope with. For that reason, algorithms that reduce the impact they have on the course of the game are put in place. How effective these measures are, is either verified through large-scale tests with end users or completely ignored.

After an in-depth look into the problem, a new method, which makes it comparatively easy to test the functionality and quality of such lag compensation techniques, is introduced in the context of this thesis. The foundation for a framework, which allows the simulation of various network conditions and is able to re-enact different game situations with the help of automated tests, is laid. The viability of the proposed method is demonstrated with the aid of a test game and an insight into the generated results is provided.

This work is directed towards game developers with experience in network programming and scientists who devote themselves to the topic of lag compensation.

Kapitel 1

Einleitung

Jedem, der einmal ein Onlinespiel über eine schlechte Internetverbindung gespielt hat, sollte klar sein, dass die Qualität des Spiels darunter leidet. Verschiedene Studien wie [1, 3, 8] haben festgestellt, das Verzögerungen bis ca. 100 ms von den meisten Spielern nicht wahrgenommen werden, darüber hinaus jedoch unangenehm auffallen. Außerdem fallen Variationen in der Latenz unangenehmer auf, als eine konstant bleibende längere Verzögerung.

Viele Spieler tun ihren Unmut auch öffentlich kund, wie eine Suche nach den Begriff *Lag* und *Unfair* im Internet zeigt. Diverse Suchmaschinen liefern dafür unzählige Ergebnisse, wie unter anderem [22–24] zeigen. Wie beispielsweise in [35, 43] berichtet wird, ist davon auszugehen, dass Onlinespiele in Zukunft noch weiter an Bedeutung gewinnen werden. Damit wird jedoch gleichzeitig auch die Anzahl der betroffenen Spiele weiter steigen, was mehr Frustration bedeutet und somit schlecht ist.

1.1 Problemstellung

Die *Lag*-Problematik ist sehr vielschichtig und es gibt leider keine ultimative Lösung, die für alle Spiele gültig ist. Es ist zwar klar, dass die Auswirkungen von Latenz einen negativen Einfluss auf das Spielgeschehen haben und es werden auch viele Lösungen vorgeschlagen, die den Problemen entgegenwirken sollen. Es gibt jedoch keine klare Definition, wodurch sich eine gute *Lag*-Kompensation auszeichnet. Zum Überprüfen, ob ein derartiger Algorithmus seinen Zweck erfüllt, müssen aufwendige Tests mit vielen Spielern durchgeführt werden, die jedoch viel Zeit in Anspruch nehmen und auch erst nachdem ein funktionsfähiger Prototyp vorhanden ist, stattfinden können.

1.2 Zielsetzung

Langfristiges Ziel ist es, ein plattform- und spielunabhängiges *Framework* zu schaffen, welches das Testen von *Lag*-Kompensationsalgorithmen auto-

matisiert und vereinfacht. Um dies zu erreichen, muss im ersten Schritt, der in dieser Arbeit behandelt wird, die Voraussetzungen für ein derartiges Unterfangen ermittelt und die Realisierbarkeit geprüft werden. Es soll ein Verfahren zum automatisierten Generieren und Auswerten von Spieldaten entwickelt werden. Zu diesem Zweck sollen verschiedene Latenzen simuliert werden, während *Packet Loss* und *Jitter* außer Acht gelassen werden. Die Beurteilung der verwendeten *Lag*-Kompensation wird dabei anhand der wenigen in der Literatur beschrieben Qualitätskriterien erfolgen. Um die Funktionalität des Prototyps zu überprüfen, sollen anschließend unter Zuhilfenahme eines bereits bestehenden Spiels jeweils zwei Spielsituationen und *Lag*-Kompensationsalgorithmen getestet werden.

1.3 Aufbau der Arbeit

In Kapitel 2 werden die Ursachen von Netzwerkverzögerungen und die Probleme, die dadurch in verteilten Mehrspieler-Spielen auftreten, untersucht. Es werden bekannte Lösungsansätze beschrieben, die den Einfluss der Latenz reduzieren und so die Qualität des Spielerlebnisses erhöhen sollen. Des Weiteren wird auf automatisierte Tests als Werkzeug zum Erhöhen der Qualität von Software und deren geringe Verbreitung in der Spielindustrie eingegangen. Ein Einblick in die derzeit verwendeten Methoden zur Analyse der Qualität von *Lag*-Kompensation wird ebenfalls geboten.

Kapitel 3 behandelt das entwickelte *Framework*, welches automatisiert den Zustand eines Test-Spiels aufzeichnet und daraus Werte berechnet, die mit anderen Ergebnissen für dasselbe Spiel vergleichbar sind. Es wird ein Weg zur simultanen Simulation von beliebigen Latenzen für mehrere Spielteilnehmer beschrieben und ein bestehendes HTML5-basiertes Browserspiel so modifiziert, dass Spielabläufe automatisiert ausgeführt werden können. Die verwendeten Algorithmen zur Berechnung der Ergebnisse werden eingehend untersucht und die Implementation beschrieben.

In Kapitel 4 wird durch mehrere Tests die Funktionalität der beschriebenen Methode bestätigt und ihr Nutzen demonstriert. Es werden zwei verschiedene Testszenarios und zwei verwendete Kompensationsalgorithmen beschrieben und die Ergebnisse der Tests interpretiert.

Abschließend wird in Kapitel 5 aufgeschlüsselt, welche Verbesserungen notwendig sind und welche Punkte in der Arbeit nicht behandelt werden. Eine Einschätzung des Nutzens der entwickelten Methode wird ebenfalls geboten.

Kapitel 2

Stand der Technik

2.1 Ursachen für Latenz

In der Informatik wird die Zeit, die ein Datenpaket braucht, um von einem Sender zu einem Empfänger zu gelangen, als Latenz bezeichnet, wie auch in [2, S. 69] beschrieben wird. Für gewöhnlich wird jedoch die einfacher zu ermittelnde *Round Trip Time*[1] (RTT) verwendet. Die RTT beschreibt die Zeitspanne, die ein Paket vom Sender zum Empfänger und wieder zurück benötigt. Computerspiele verwenden häufig den Begriff *Lag*[2] anstelle von RTT, um die Verbindungsqualität anzugeben. Entstehen können derartige Verzögerungen aus verschiedenen Gründen, wie im Nachfolgenden näher erläutert wird.

2.1.1 Hardware

In der Regel ist ein Spieler nicht direkt mit dem Server im Internet verbunden, sondern sendet Nachrichten über eine Anzahl von Netzwerkgeräten. Diese müssen jeweils warten, bis sie ein Paket komplett empfangen haben, bevor dieses weitergesendet werden kann. Die Zeit, die ein Paket bei den einzelnen Geräten verbringt, summieren sich deshalb auf, wie in [8, 9] beschrieben. Zusätzlich befindet sich auf dem Weg noch etliche transparente Hardware wie z. B. *Switches* oder *Repeater*, welche ebenfalls eine geringfügige Latenz aufweisen. Falls unterwegs auch noch schlecht konfigurierte, überlastete oder defekte Geräte angetroffen werden, kann sich dies schnell zuspitzen.

Moderne Router senden Pakete im Idealfall in unter 100 µs weiter, können jedoch durchaus auch länger brauchen, wie Tests von *Cisco* in [41] zeigen. Werden all diese Verzögerungen zusammengezählt, kommt man bereits auf Werte im Millisekundenbereich. Wie in [37] festgestellt wird, ist üblicherweise die Latenz zwischen einem Haushalt und dem Internetanbieter am höchsten.

[1] englisch für Rundreisezeit
[2] englisch für Verzögerung

Dies kann unter Windows mit dem Programm `tracert` getestet werden, welches mehrere *pings* zu allen sichtbaren Geräten, auf dem Weg zu einer Zieladresse, sendet.

```
>tracert -d google.at
Tracing route to starforcedelta.com [78.142.142.29]
over a maximum of 30 hops:
  1    <1 ms    <1 ms    <1 ms  10.0.0.138
  2     9 ms     9 ms     9 ms  93.82.71.254
  3    11 ms     9 ms     9 ms  195.3.66.149
  4    11 ms    11 ms    12 ms  195.3.68.157
  5    11 ms    12 ms    17 ms  195.3.68.126
  6    11 ms    13 ms    12 ms  193.203.0.105
  7    12 ms    14 ms    12 ms  130.244.71.76
  8    12 ms    12 ms    12 ms  130.244.49.118
  9    12 ms    12 ms    11 ms  212.152.193.89
 10    12 ms    16 ms    12 ms  81.189.132.90
 11    12 ms    14 ms    11 ms  78.142.142.1
 12    12 ms    13 ms    11 ms  78.142.142.29
Trace complete.
```

Der lokale Router ist in unter 1 ms erreichbar, während der erste externe Hop bereits zwei Drittel der RTT zur Zieladresse ausmacht.

2.1.2 Datenübertragungsrate

Laut [25] bezeichnet Datenübertragungsrate die Anzahl der Bits, die pro Sekunde über eine Netzwerkverbindung gesendet werden können. Diese ist durch die eingesetzte Hardware und Leitungen limitiert, kann aber auch durch den Netzbetreiber künstlich begrenzt werden. Auf die Latenz wirkt sich diese Einschränkung dadurch aus, dass Netzwerkgeräte überschüssige Pakete in einem Puffer zwischenspeichern, wie auch [9] bestätigt. Dadurch kann es passieren, dass Daten, die für ein Spiel wichtig sind, für eine unbestimmte Zeit auf diesem Gerät verweilen. *Quality of Service* (QoS)-fähige Hardware kann in diesem Fall Abhilfe schaffen, da diese ein Paket analysiert und abhängig vom Inhalt anders priorisiert. Somit können Pakete von interaktiven Anwendungen beim Abarbeiten des Puffers bevorzugt und schneller weitergesendet werden.

Wie in [34] gezeigt, steigt die maximal verfügbare Datenübertragungsrate jährlich quadratisch an, während der Verbrauch exponentiell mit einem Faktor von derzeit nur etwa 1,5 wächst. Das bedeutet, dass in Zukunft die Datenübertragungsrate wahrscheinlich weniger Auswirkungen auf die Latenz haben wird.

2.1.3 Paketverluste

In [2, S. 44f.] wird festgestellt, dass bei Internetspielen normalerweise als Transportprotokoll entweder das *Transmission Control Protocol* (TCP) oder das *User Datagram Protocol* (UDP) eingesetzt wird. Als solche sind TCP

und UDP dafür zuständig, dass größere Datenpakete beim Senden zerteilt und beim Empfangen wieder zusammengebaut werden. TCP kümmert sich weiterhin darum, dass Pakete in der richtigen Reihenfolge an die Anwendung übergeben werden und bei Verlusten auch erneut gesendet werden. Durch die andauernd gesendeten Empfangsbestätigungen kann TCP des Weiteren feststellen, ob die Verbindung zum Empfänger verloren gegangen ist. Dadurch bietet TCP zwar eine hohe Zuverlässigkeit, kann bei schlechten Netzwerkbedingungen aber auch für längere Zeit keine Daten an die Anwendung liefern. UDP dagegen liefert einfach nur die Pakete an die Anwendung. Die Empfangsreihenfolge und Paketverluste müssen hier vom Entwickler selbst berücksichtigt werden.

Welches Transportprotokoll besser geeignet ist, hängt vom Spiel selbst ab. Während Stragiespiele von der Zuverlässigkeit von TCP profitieren, kann ein Actionspiel durch das implizite Warten auf verlorene Pakete unspielbar werden. Bei kabelgebundenen Verbindungen werden solche Paketverluste zwar durch bessere Hardware immer seltener, bei WLAN-Verbindungen sind Paketverluste jedoch sehr häufig, wie in [21] festgestellt wird. Paketverluste im Bereich von 10 bis 70 Prozent sind durchaus üblich, wodurch sich Verzögerungen von mehreren 100 ms ergeben können, wie auch in [37] erwähnt. Dies ist vor allem bei *Tablet*- und *Smartphone*-Spielen ein Problem, welche in den letzten Jahren stark an Bedeutung gewonnen haben. Wie in [42] berichtet, ist zu erwarten, dass bis 2016 ein Großteil aller Spiele auf mobilen Endgeräten ausgeführt wird.

Ein Test mit dem `ping`-Programm zeigt, dass selbst wenn keine anderen Geräte und WLANs in der Umgebung sind und der Abstand zum WLAN-*Access Point* gering ist, viele Pakete mehrmals gesendet werden müssen.

```
>ping /n 100 10.0.0.138
Pinging 10.0.0.138 with 32 bytes of data:
Reply from 10.0.0.138: bytes=32 time=1ms TTL=64
Reply from 10.0.0.138: bytes=32 time=1ms TTL=64
...
Reply from 10.0.0.138: bytes=32 time=7ms TTL=64
Reply from 10.0.0.138: bytes=32 time=29ms TTL=64
Reply from 10.0.0.138: bytes=32 time=51ms TTL=64
...
Reply from 10.0.0.138: bytes=32 time=11ms TTL=64
Reply from 10.0.0.138: bytes=32 time=34ms TTL=64
...
Ping statistics for 10.0.0.138:
    Packets: Sent = 100, Received = 100, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 51ms, Average = 2ms
```

Bei hundert gesendeten Pings wurden fünf Pakete mehrmals gesendet, wie an der Sendezeit erkannt werden kann. Das bedeutet, dass selbst unter idealen Bedingungen 5 % Paketverlust auftreten.

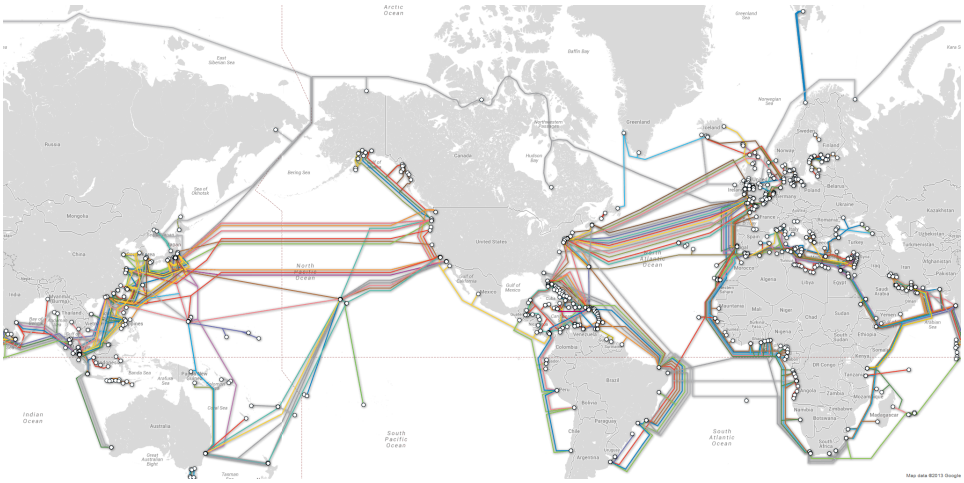


Abbildung 2.1: Karte der weltweit verlegten und geplanten Unterseekabel. Bildquelle: [28].

2.1.4 Signalgeschwindigkeit

Das fundamentale Problem in der Kommunikation zwischen Client und Server stellt, wie in [2, S. 71] und [8] erwähnt, die Physik selbst dar. Die Ausbreitungsgeschwindigkeit von Signalen liegt bei etwa $0{,}64c$ in Netzwerkkabeln und bei ca. $0{,}658c$ in Glasfaser, wie aus [26, 27] hervorgeht. Somit kann die Zeit, die ein Signal innerhalb von Glasfaser benötigt, um eine bestimmte Strecke zurückzulegen, mit

$$t(d) = \frac{d}{v_S} = \frac{d}{0{,}658c} \tag{2.1}$$

berechnet werden. Zum Zurücklegen eines ein Kilometer langen Stücks Glasfaserleitung benötigt ein Signal also

$$t[1\,\mathrm{km}] = \frac{1000\,\mathrm{m}}{0{,}658 \cdot 3 \cdot 10^8 \frac{\mathrm{m}}{\mathrm{s}}} \approx 5{,}0659\,\mu\mathrm{s}. \tag{2.2}$$

Wird der Erdumfang mit ca. 40.000 km angenommen, ergibt sich daraus, dass ein Datenpaket zwischen zwei gegenüberliegenden Punkten auf der Erdoberfläche derzeit nie in weniger als 200 ms hin und zurück übertragen werden kann. Wie die Karte in Abbildung 2.1 zeigt, sind Glasfaserkabel meist nicht in einer direkten Linie zwischen zwei Orten verlegt, wodurch in der Realität die Latenz um einiges Höher ausfällt.

2.1.5 Verbindungstest

Diese Vermutung kann durch einen Verbindungstest zwischen einem Server in einem Amazon Rechenzentrum in Singapur und einem Server in einem Rechenzentrum in Wien bestätigt werden. Die Distanz zwischen diesen beiden

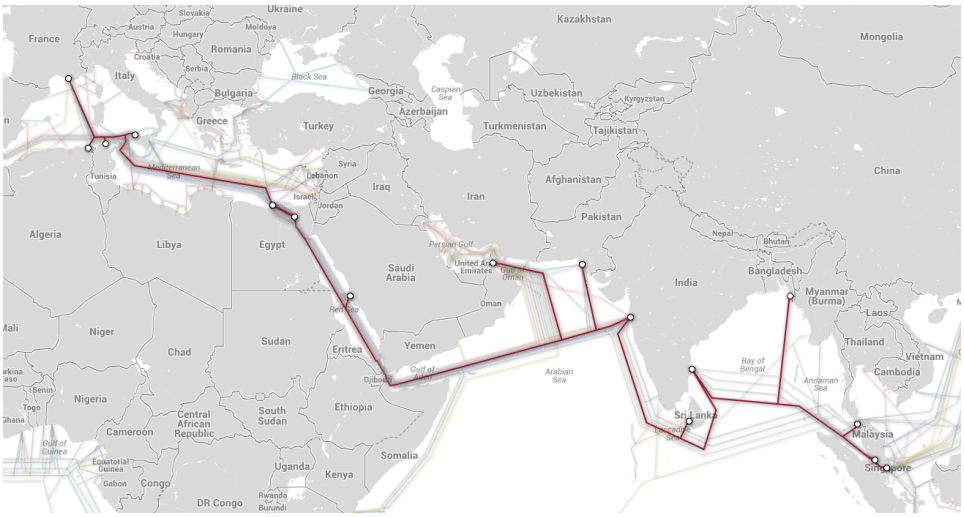


Abbildung 2.2: „SeaMeWe-4“ verbindet Europa, Afrika und Asien. Bildquelle: [28].

Servern beträgt ca. 9.700 km Luftlinie[3], wodurch sich eine RTT von ca. 100 ms berechnen lässt. Die neueste und kürzeste Glasfaserverbindung zwischen Europa und Asien ist „SeaMeWe-4“, welche in Abbildung 2.2 zu sehen ist.

Zum Testen der Verbindung wird das Linux-Programm `traceroute` verwendet. Ausgeführt wird der Befehl auf dem Server in Singapur, da dessen Firewall ICMP Pakete blockt und somit vom Server in Wien ausgehend keine RTT ermittelt werden kann.

```
traceroute to 78.142.142.29, 30 hops max, 60 byte packets
 1  175.41.128.234  0.446 ms  0.619 ms  0.621 ms
 2  203.83.223.75  1.785 ms  1.788 ms  1.761 ms
 3  203.83.223.59  1.561 ms  1.568 ms  1.758 ms
 4  63.218.213.205  2.295 ms  2.485 ms  2.492 ms
 5  206.126.236.161  247.106 ms  247.825 ms  247.544 ms
 6  130.244.38.218  330.306 ms  330.153 ms  329.731 ms
 7  130.244.71.125  339.430 ms  337.659 ms  338.879 ms
 8  130.244.49.110  444.457 ms *  444.354 ms
 9  130.244.49.118  351.146 ms  350.445 ms  350.473 ms
10  212.152.193.89  353.458 ms  353.244 ms  353.302 ms
11  81.189.132.90  355.980 ms  355.748 ms  355.685 ms
12  78.142.142.1  343.701 ms  343.570 ms  343.431 ms
13  78.142.142.29  360.263 ms !X  360.652 ms !X  360.435 ms !X
```

Die IP Adressen in Zeile 1 bis 3 sind, wie zu erwarten war, auf *Amazon* registriert[4]. In Zeile 4 ist eine IP von *PCCW Global* zu sehen, während Zeile 5 eine IP von *Equinix* zeigt. Gemeinsam mit der RTT von rund 250 ms kann man deshalb darauf schließen, dass die Verbindung über Amerika geleitet

[3]ermittelt mit http://www.luftlinie.org/
[4]ermittelt mit http://www.fixedorbit.com/ und http://www.whoisstuff.net/

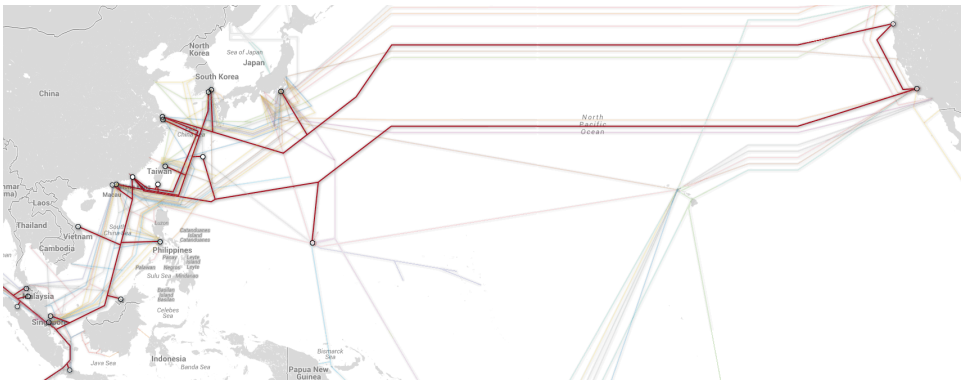


Abbildung 2.3: „SeaMeWe-3“ und „China-U.S. Cable Network (CHUS)“ verläuft von Asien nach Nord-Amerika. Bildquelle: [28].

wird. Ab Zeile 6 sind IPs im Besitz von *Tele2* zu sehen, welche sich bereits in Europa befinden.

Um eine grobe Abschätzung der zurückgelegten Entfernung zu erhalten, wird angenommen, dass die Signale über die Unterseekabel in Abbildung 2.3 und 2.4 geleitet werden. Die gesamte zurückgelegte Strecke beträgt somit mindestens 27.500 km in eine Richtung, wodurch sich mit Gleichung 2.1 ein Mindestwert für die RTT von ca. 280 ms berechnen lässt. Die Werte in der Ausgabe zeigen jedoch deutlich, dass dieser Wert in der Realität um etwa 70 ms höher ist. Dies kann zum Einen an der ungenauen Schätzung der Entfernung liegen und zum Anderen an der transparenten Hardware, die in der Ausgabe von `traceroute` nicht aufscheint. Der Anstieg in der RTT zwischen Zeile 12 und 13 liegt vermutlich an der Konfiguration der Firewall im Datenzentrum, welche ICMP Pakete mit niedriger Priorität abarbeiten dürfte.

Warum das Signal einen weiten Umweg über Amerika macht, ist ein Rätsel, ein möglicher Grund könnte jedoch die Anbieterwahl von Amazon sein, welche mit einem günstigeren Preis zusammenhängen dürfte.

2.2 *Lag*-Kompensationstechniken

Prinzipiell hat *Lag*-Kompensation zwei Ziele zu erfüllen. Einerseits soll bei schlechter Netzwerkverbindung ein reibungsloser Spielverlauf ermöglicht werden. Andererseits sollen Spieler mit verschieden guter Verbindung möglichst die gleiche Chance haben, ein bestimmtes Ziel zu erreichen. Da diese beiden Ziele in gewissem Sinne Gegensätze darstellen, müssen hier Kompromisse eingegangen werden. Weiterhin können mehrere Arten von Algorithmen unterschieden werden, die an verschiedenen Punkten in der Kommunikation ansetzen und sich ergänzen können.

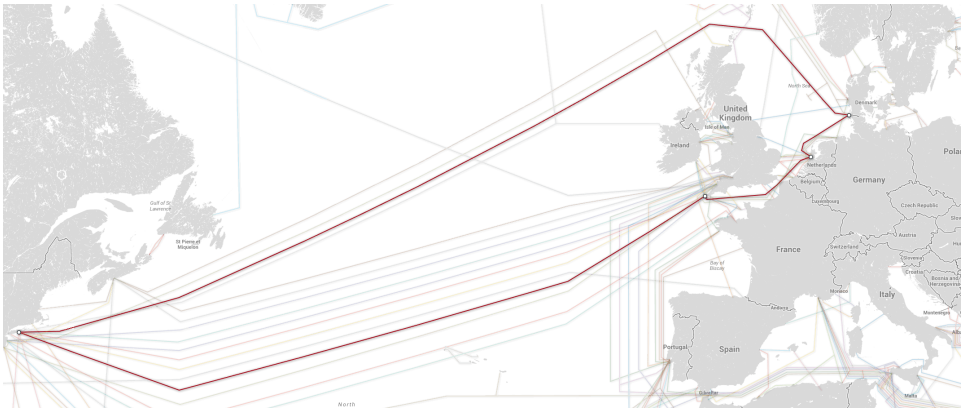


Abbildung 2.4: „Atlantic Crossing-1 (AC-1)“ liegt zwischen Nord-Amerika und Europa. Bildquelle: [28].

2.2.1 Reduktion

Um den Auswirkungen von Latenz entgegenzuwirken, bietet es sich an, so wenig Daten wie möglich über das Netzwerk zu versenden. Es können z. B. Zustände von Spielfiguren, die der Empfänger nicht sehen kann, weggelassen werden. Des Weiteren ist es von Vorteil, wenn anstelle von vielen kleinen Paketen ein gesammeltes Paket gesendet wird, da dadurch der Mehraufwand reduziert wird. Dies setzt zwar voraus, dass Daten nicht sofort versendet werden, dadurch können aber beispielsweise Aktionen, die sich gegenseitig aufheben, ganz weggelassen werden.

Eine weitere einfache Möglichkeit, wie die Datenmenge reduziert werden kann, bietet sich durch Verwendung von verlustfreien Kompressionsalgorithmen wie z. B. *Deflate*, oder dem *Lempel-Ziv-Markow-Algorithmus* (LZMA). Anstatt Daten immer zwischen Client und Server zu senden, können Updates auch direkt zwischen Clients ausgetauscht werden, falls diese eine bessere Verbindung zueinander haben. Es sollte jedoch bedacht werden, dass Betrüger hier eventuell eine Möglichkeit bekommen, zu ihrem Vorteil veränderte Daten an alle Spieler zu senden.

2.2.2 Vorhersage

Zu dieser Kategorie können alle Algorithmen gezählt werden, die den Zustand der Spielwelt auf Basis von bereits erhaltenen Daten vorberechnen.

Dead Reckoning (DR)

Die primitivste Variante von Extrapolation wurde schon lange vor der Entwicklung des Computers in der Seefahrt eingesetzt, wie in [16] erwähnt wird. Auch verschiedene Tiere verwenden die Prinzipien hinter *Dead Reckoning*,

um ihre Position abzuschätzen, wie in [19] erläutert wird. In [33] wird beschrieben, dass DR in der Informatik erstmals Verwendung in verteilten militärischen Simulationen fand, die diese Technik einsetzten, um Verzögerungen in der Kommunikation auszugleichen.

Anhand der Position (P_0), Geschwindigkeit (V_0) und Beschleunigung (A_0) eines Akteurs kann mit

$$P(t) = P_0 + V_0 t + \frac{1}{2} A_0 t^2 \tag{2.3}$$

extrapoliert werden, wo er sich zu einem beliebigen Zeitpunkt befinden wird. Dadurch wird die Anzahl der notwendigen Positionsupdates reduziert und der Spielablauf kann unabhängig von der Regelmäßigkeit der Netzwerkkommunikation berechnet werden. Allerdings sind diese Berechnungen nur Annäherungen an die echte Position und sobald die ermittelte Position zu weit abweicht, müssen diese beiden Positionen wieder zusammengeführt werden.

Im einfachsten Fall überschreibt die vom Server erhaltene Position den Zustand am Client, was jedoch zu sprunghaften Bewegungen führt und somit die Immersion des Spielers reduziert. Eine bessere Möglichkeit besteht durch Interpolation der beiden Positionen über einen bestimmten Zeitraum. Dies kann z. B. mit Überblenden der Geschwindigkeit oder mit Kurveninterpolation erreicht werden, wie aus [29] entnommen werden kann. Der zum Überblenden verwendete Faktor T kann durch lineare Interpolation der Empfangszeit t_0 und der maximalen Überblendzeit t_1 mit

$$T(t) = \frac{t - t_0}{t_1 - t_0}, \quad \text{mit } t \in [t_0, t_1] \tag{2.4}$$

berechnet werden. Die interpolierte Geschwindigkeit wird anschließend als

$$V_b(t) = V_0 + (V_0' - V_0) T(t) \tag{2.5}$$

ermittelt und anstelle der Ausgangsgeschwindigkeit V_0 in Gleichung 2.3 eingesetzt, während die zweite Position $P'(t)$ weiterhin auf Basis von V_0 berechnet wird. Abschließend werden die beiden vorliegenden Positionen linear interpoliert und somit die endgültige Position

$$Pos(t) = P(t) + (P'(t) - P(t)) \cdot T(t) \tag{2.6}$$

berechnet. Bei Verwendung einer kubischen Bézierkurve kann die Position als

$$Pos(t) = (1-T)^3 P(t_0) + 3(1-T)^2 T P(t_1) + 3(1-T) T^2 P'(t_0) + T^3 P'(t_1) \tag{2.7}$$

berechnet werden. Dabei werden jeweils die extrapolierte Position und erhaltene Position zum Empfangszeitpunkt t_0 und die extrapolierten Positionen zum gewünschten Zusammenführungszeitpunkt t_1 als Kontrollpunkte

benutzt. $P(t)$ und $P'(t)$ verwenden jeweils die Definition in (2.3) und T ist als $T(t)$ wie in (2.4) definiert.

DR bietet sich vor allem für Spiele mit trägen Akteuren, wie z. B. Rennspiele oder Flugsimulationen an, da die Bewegungen dort vorhersehbar sind. Bei Spielen, in denen sich die Spielfigur willkürlich bewegen kann, ist diese Methode meist weniger geeignet, da der extrapolierte Pfad oft weit vom eigentlichen Pfad abweicht und dadurch der Vorteil verloren geht.

Interessen-basierte Extrapolation

Da DR, wie in [12, 20] beschrieben, bei hohen Latenzen nur noch schlecht funktioniert, werden bei Techniken wie *AntReckoning* und *Interest Schemes*, die Interessen des Spielers in die Berechnung miteinbezogen.

Es wird davon ausgegangen, dass Spieler bestimmte Punkte, Objekte und andere Spielfiguren entweder meiden oder ansteuern wollen. Dadurch kann die Abweichung zwischen dem echten Weg und dem extrapolierten Pfad minimiert werden. Bei *Interest Schemes* werden die Spieler zusätzlich in drei Kategorien klassifiziert:

- Spieler, die als oberste Priorität ein Objekt aufheben wollen,
- Spieler, die andere Spieler besiegen wollen,
- Spieler, die sich neutral verhalten.

Der Verhaltenswert wird quantifiziert und beeinflusst, wie stark sich die Kräfte auf den Spieler auswirken. Da bei niedrigen Latenzen die Verbesserung durch diese Methode nur gering ausfällt, wird dazu geraten, erst ab einem Schwellwert die zusätzlichen Berechnungen durchzuführen.

Parallele Simulation

Bei dieser Methode wird die Spielwelt, wie in [5] beschrieben, auf allen Clients komplett berechnet, wodurch es ausreicht, nur die Eingaben der Spieler zu senden. Voraussetzung dafür ist, dass die Simulation deterministisch abläuft, da sonst Inkonsistenzen zwischen den verschiedenen Teilnehmern entstehen.

Besonders gut geeignet ist diese Technik für Echtzeit-Strategiespiele, da bei diesen in der Regel eine Partie bei allen Spielern gleichzeitig beginnt und Spieler nur indirekt auf die Spielwelt einwirken können. Andere Spiele, die nicht für alle Spieler die gleiche Ausgangssituation bieten, müssen einen Weg bieten, um den initialen Zustand für nachträglich beitretende Spieler zu übertragen.

Da die Spielwelt auf der Benutzerseite berechnet wird, können Eingaben sofort angewandt werden, wodurch Netzwerkverzögerungen im Normalfall nicht mehr wahrnehmbar sind, wie auch in [4] beschrieben. Wenn jedoch eine Eingabe vom Server abgelehnt wird, muss ähnlich wie bei DR ein Weg gefunden werden, wie der lokale Zustand wieder an den des Servers angeglichen wird.

2.2.3 Zeitmanipulation

Zu dieser Kategorie zählen Algorithmen, die auf die ein oder andere Weise den Ablauf des Spiels verzögern oder nachträglich verändern, um den korrekten Ablauf von Ereignissen zwischen den verschiedenen Darstellungen der Spielwelt zu bewahren.

Local Lag (LL)

Wie unter anderem in [13] und [2, S. 93f] beschrieben, zielt dieser Algorithmus darauf ab, dass alle beteiligten Spieler dieselbe Latenz haben, um Inkonsistenz in der Darstellung auf den einzelnen Clients zu eliminieren. Hierfür wird die höchste Latenz ermittelt und anschließend entweder bei allen Spielern mit weniger Latenz die Darstellung künstlich verzögert, oder das Paket entsprechend später vom Server weitergesendet. Dadurch erhalten alle Spieler eine korrekte Darstellung der Spielwelt, jedoch auf Kosten der Interaktivität.

Time Warp

Sollten Verzögerungen in der Kommunikation auftreten, die durch LL nicht mehr kompensiert werden können, muss der Spielzustand rückwirkend korrigiert werden, wie in [13] beschrieben. Um die korrekte Reihenfolge aller Ereignisse sicherzustellen, wird am Server eine Liste mit vergangenen Zuständen der Spielwelt geführt und eine zweite Liste mit allen von den Spielern erhaltenen Befehlen. Diese Listen sind jeweils nach einer synchronisierten Zeit sortiert, wodurch die alten Zustände beim Erhalt eines verspäteten Befehls im Schnelldurchlauf aktualisiert werden können und der neue Zustand auf Basis des korrigierten, vorhergehenden Zustandes, berechnet werden kann. Zustände die älter als die höchste gemessene RTT sind, können abschließend ohne Weiteres aus dieser Liste gelöscht werden.

2.2.4 Spieldesign

In vielen Fällen kann auch durch geschicktes Kombinieren von Spielelementen Abhilfe geschaffen werden, die Möglichkeiten hängen hier jedoch stark vom jeweiligen Spiel ab. In einem Rennspiel ist es beispielsweise nicht zwingend notwendig, dass die konkreten Positionen aller Teilnehmer übertragen werden, solange Schlüsselereignisse, wie das Überholen, richtig dargestellt werden und zum Ende des Rennens die Reihenfolge, in der die Spieler das Ziel erreichen, stimmt. In vielen Spielen kann auch durch Animationen mehr Zeit zum Übertragen der Daten verschafft werden. So kann z. B. eine Tresortür eine längere Öffnungssequenz zeigen, während auf die Antwort vom Server gewartet wird.

2.3 Testen von Software

Softwaretests sind ein oder mehrere Prozesse, die entworfen werden, um sicherzustellen, dass der Programmcode genau das macht, was er machen soll und nichts anderes. Software sollte vorhersehbar und konsistent sein und den Benutzer nicht überraschen, wie in [14, S. 8] festgestellt wird. In [14, S. 10f.] wird des Weiteren eine mögliche Definition für Softwaretests geboten, die folgendermaßen lautet:

> Testing is the process of executing a program with the intent of finding errors.

Diese Definition berücksichtigt jedoch nur einen Teil der etablierten Testmethoden, da es auch Tests gibt, die ein Programm nicht ausführen. In [30] wird beschrieben, dass Softwaretests aus zahlreichen Einzelmaßnahmen bestehen, die Fehler aufdecken sollen, um deren Wirkung im Produktivbetrieb vorzubeugen. Das Testen soll dabei Vertrauen in die Qualität der Software schaffen.

2.3.1 Klassifikation von Tests

Wie schon an der sehr weit gefassten Definition erkannt werden kann, gibt es auch zahlreiche unterschiedliche Kriterien, nach denen Softwaretests eingeordnet werden können. Tests können, wie in [30] beschrieben, z. B. nach der Prüftechnik, nach dem verwendeten Testkriterium, nach dem Zeitpunkt in der Produktentwicklung, nach Testintensität, nach Informationsstand über die zu testende Komponente, u. v. m. unterschieden werden.

Auf Grund des Themas dieser Arbeit bietet sich vor allem die Unterscheidung von manuell und automatisch ablaufenden Tests an. Bei *manuellen Softwaretests* handelt es sich um Tests, die von ein oder mehreren Personen durchgeführt werden müssen und je nach Art einem festgelegten Ablauf folgen, oder einfach zufällig die Funktionalität der Software testen. *Automatische Tests* können dagegen von einem Computer ohne Beaufsichtigung durchgeführt werden.

Diese Tests können weiterhin in statische und dynamische Tests unterschieden werden. *Statische Tests* prüfen den produzierten Programmcode auf seine formelle Richtigkeit, ohne diesen auszuführen. Viele moderne Entwicklungstools können neu geschriebenen Code bereits während der Eingabe auf syntaktische Korrektheit testen. Zusätzlich können jedoch auch Maßnahmen eingesetzt werden, um typisch auftretende, problematische Muster im Code zu finden, die bei der Ausführung zu Fehlern führen könnten, wie in [36] erklärt. *Dynamische Tests* arbeiten dagegen mit dem laufenden Programm oder Teilen davon und testen, ob für eine bestimmte Eingabe das erwartete Ergebnis produziert wird. Diese Testfälle werden von einem Programmierer festgelegt und sollten bei jeder Ausführung dasselbe Ergebnis liefern.

Dadurch können Fehler in der Software bereits während des Entwicklungsprozesses erkannt werden.

Dynamische Tests können wieder in mehrere Unterarten gegliedert werden. *Komponententests* sind dafür vorgesehen, einen Programmteil, wie z. B. eine Funktion, isoliert vom Rest des Programms zu testen. Dies kann durch Ersetzen von externen Abhängigkeiten durch Mock-Objekte erzielt werden. Dadurch kann sichergestellt werden, dass eine bestimmte Funktion die gewünschten Ergebnisse liefert. *Integrationstests* testen dagegen das Zusammenspiel von verschiedenen Komponenten unter kontrollierten Bedingungen. Auch hier können Mock-Objekte zum Einsatz kommen, um den Test auf bestimmte Programmteile zu beschränken. *Systemtests* gehen einen Schritt weiter und testen das vollständige Programm auf Nebenwirkungen, die andere Programme beeinflussen könnten.

2.3.2 Softwaretests in der Spielentwicklung

In der Spielindustrie wird selten automatisiert getestet, wie aus [38, 40] hervorgeht. Im Normalfall werden Spiele gegen Ende des Entwicklungsprozesses in Alpha- und Betatests von Testspielern gespielt, um die Anzahl der bestehenden Fehler vor der Veröffentlichung zu reduzieren. Spielfirmen beauftragen auch oft Qualitätssicherungsfirmen, oder haben eine eigene Abteilung, um ihr Produkt zu testen. *Alpha-Tests* werden in der Regel von den Entwicklern selbst durchgeführt und können daher relativ einfach organisiert werden. *Beta-Tests* werden durch außenstehende Personen, die dem Zielpublikum entspringen, durchgeführt. Die Software sollte in dieser Phase bereits weitgehend fehlerfrei sein.

2.4 Testen von *Lag*-Kompensation

Zum Testen der Qualität der eingesetzten *Lag*-Kompensation werden in der Regel Teststudien mit mehreren Benutzern eingesetzt. Dazu werden mehrere Personen eingeladen, die das Spiel in einer präparierten Umgebung spielen, in der verschiedene Latenzen künstlich erzeugt werden können. Diese Testspieler müssen während oder nach Abschluss der Tests auf eine zuvor festgelegte Weise ihre Meinung zur Qualität des Spiels abgeben. Die Ergebnisse werden ausgewertet und erlauben so Rückschlüsse auf die Brauchbarkeit der eingesetzten *Lag*-Kompensation.

Ein derartiger Testaufbau besteht gewöhnlich aus einem Server und einer beliebigen Anzahl von Clients, die in einem lokalen Netzwerk über einen Proxy mit dem Server kommunizieren. Der Proxy kann die Latenz in der Kommunikation über Software wie z. B. *Dummynet* oder *tc* steuern. Die einzelnen Client-PCs können zusätzlich mit je drei Kameras gefilmt werden, die auf das Gesicht des Testers, die Tastatur und den Bildschirm gerichtet sind. Dadurch können nach Abschluss der Tests zusätzliche Rückschlüsse auf die

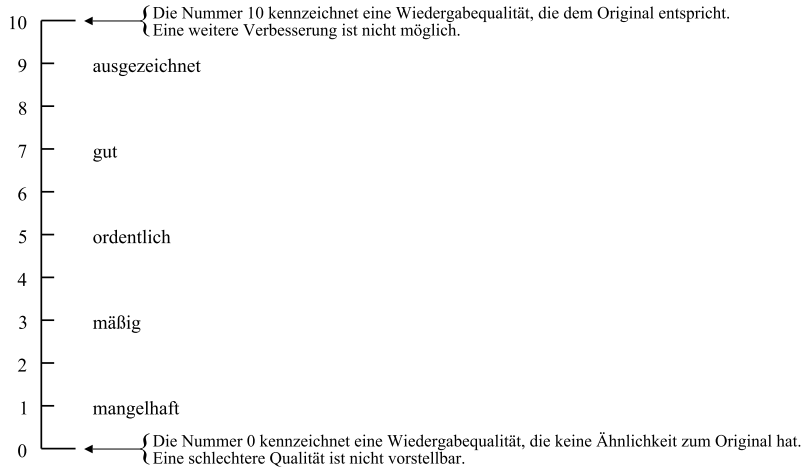


Abbildung 2.5: Elfteilige, numerische Qualitätsskala wie in [10] gezeigt.

von den Testern angegebenen Werte gezogen werden. Die Benutzer spielen das Spiel für einen bestimmte Zeit in diesem Testnetzwerk, während auf dem Proxyserver verschiedene Netzwerkzustände simuliert werden. Anschließend werden die Aussagen der Benutzer auf eine der nachfolgend beschriebenen Arten ausgewertet.

2.4.1 *Absolute Category Rating* (ACR)

Bei dieser in [10] beschriebenen Methode wird jeweils eine Testsequenz für einige Sekunden durchlaufen, woraufhin die Testperson einige Sekunden Zeit hat, die Qualität auf einer Skala zu bewerten. Es wird empfohlen, dass die Testsequenzen je nach Anwendungsfall für etwa 10 Sekunden dargestellt werden und den Testern maximal 10 Sekunden Zeit für die Bewertung gegeben wird. Im Falle der Vielzahl an komplexen Zusammenhängen von *Lag*-Kompensationstests ist jedoch eine längere Testdauer wahrscheinlich von Vorteil.

Üblicherweise wird bei ACR eine 5, 9 oder 11 Punktskala, wie in Abbildung 2.5 gezeigt, verwendet und nach einem absoluten Wert gefragt. Alternativ kann jedoch auch eine Linie mit jeweils einer negativen und einer positiven Beschriftung an den Enden verwendet werden, auf der die Testpersonen eine Markierung an einer beliebigen Position setzen dürfen. Vorteil der zweiten Variante ist, dass die Ergebnisse weniger von der subjektiven Interpretation der Beschriftung auf der Skala abhängen.

2.4.2 *Degradation Category Rating* (DCR)

Ebenfalls in [10] festgelegt, folgt dieser Auswertungsmodus denselben Regeln wie ACR, mit dem Unterschied, dass jeweils eine Referenz- und Testsequenz direkt nacheinander dargestellt werden, bevor diese bewertet werden. Dadurch können die Testpersonen einen direkten Vergleich zum Normalzustand ziehen und Auswirkungen von persönlichen Erfahrungen minimiert werden. Ein gravierender Nachteil bei diesem Verfahren ist, dass die Testpersonen hier wissen, dass die zweite Darstellung schlechter als die erste sein soll und dadurch dazu verleitet sind, diese auch schlechter zu bewerten, wenn sie womöglich gleich gut sind.

2.4.3 *Pair Comparison Method* (PC)

Die Dritte, in [10] erwähnte Methode, umgeht den Nachteil von DCR, indem alle möglichen Kombinationen getestet werden, wodurch der Benutzer nicht weiß, welche Darstellung den Idealzustand repräsentiert. Außerdem können die Ergebnisse validiert werden, da die Benutzer jede Paarung zwei Mal dargestellt bekommen (AB und BA). Im Gegensatz zu den beiden vorhergehenden Methoden wird die Wertung hier vereinfacht und es kann nur gewählt werden, welche der beiden gezeigten Sequenzen bevorzugt wird.

2.4.4 *OneClick*

Bei dieser Testvariante, die in [6] beschrieben wird, erfolgt die Bewertung nicht über eine Punkteskala, sondern die Testbenutzer können zu jeder Zeit während des Tests eine Schaltfläche drücken, wenn sie unzufrieden sind. Die Qualität der Anwendung kann danach durch die Anzahl der Klicks in einer bestimmten Zeit ermittelt werden. Dies bietet den Vorteil, dass die Tester nicht darüber nachdenken müssen, wie gut eine Darstellung war und somit zeitnahes Feedback während der Tests geben können.

2.4.5 *Group Synchronized Control*

Dieses in [11] beschriebene Verfahren bietet keine eigene Wertung, sondern kann mit den zuvor beschriebenen Verfahren verwendet werden. Beim Testaufbau wird jedoch zusätzlich darauf geachtet, dass mehrere Benutzer zur selben Zeit denselben Level an Latenz haben. Dadurch können die Ergebnisse von mehreren Benutzern verglichen werden, wodurch die Qualität der individuellen Bewertungen bestätigt werden kann.

Kapitel 3

Test *Framework*

Um die Funktionalität und Qualität von den in einem Spiel eingesetzten *Lag*-Kompensationstechniken ermitteln zu können, bieten sich automatisierte Tests besonders an, da diese den Gesamtzustand eines Spiels, übergreifend zwischen Server und Clients, testen können.

Nach der in Kapitel 2 beschriebenen Kategorisierung von Testverfahren kann ein solches *Framework* als spezielle Form von Integrationstest eingeordnet werden, da nur bestimmte Teile des Programms auf ihr Zusammenspiel getestet werden. Beispielsweise werden für solche Tests keine Ein- und Ausgabesysteme für Endbenutzer benötigt und deshalb deaktiviert. Zum Zeitpunkt, zu dem diese Arbeit verfasst wird, gibt es jedoch keine Rückschlüsse auf den Einsatz derartiger Techniken in der Spielindustrie. Auch gibt es in der Literatur nur eine einzige formelle Beschreibung eines Bewertungssystems, welches für die Verwendung mit einem solchen System gedacht ist. Dieses wird im Verlauf dieses Kapitels noch ausführlich beschrieben.

3.1 Testspiel

Als Testspiel wird eine modifizierte Version des HTML5 basierten Browserspiels *Swords 'n' Bombs*[1] verwendet, welches im Laufe eines vorangegangenen Semesters entwickelt wurde. Das Spiel, zu sehen in Abbildung 3.1, war als Prototyp und Testumgebung für moderne Webtechnologien geplant und sollte es hunderten Spielern ermöglichen, gleichzeitig miteinander zu spielen. Die gesetzten Ziele wurden weitestgehend erreicht, jedoch konnte die Skalierbarkeit des Spiels, auf Grund von mangelnden Testern, nur beschränkt überprüft werden.

Ziel des Spiels ist es, möglichst viele Mitspieler mit dem Schwert oder einer Bombe zu bezwingen. Die verwendeten Grafiken stammen aus dem *Open Source*-Spiel *Browserquest*[2], welches von *Mozilla* als Vorzeigeprojekt

[1] ©2012 Daniel Kaneider, Fabian Meisinger und Andreas Jagel
[2] http://browserquest.mozilla.org/

Abbildung 3.1: Das Spiel ist eine Mischung aus *Zelda* und *Bomberman*.

für HTML5 Technologien etwa zwei Monate nach Beginn des Projektes veröffentlicht wurde und bis dahin dem Team nicht bekannt war.

3.1.1 Server

Der Server wurde in *Java* implementiert und verwendet die *Jetty*-Bibliothek[3], da diese zum Zeitpunkt der Entwicklung bereits das neuartige *WebSocket*-Protokoll unterstützte und außerdem gleichzeitig als Webserver zum Ausliefern der HTML- und Ressource-Dateien dient. Weiterhin wird zum Simulieren der Spielwelt die *Java*-Portierung der *Box2D*-Physikbibliothek[4] eingesetzt. Diese ist zwar eigentlich für die Simulation von realistischen Abläufen gedacht, kann jedoch auch dazu benutzt werden, um die Position von Figuren zu berechnen, die direkt von Spielern gesteuert werden.

Damit die Netzwerkkommunikation von der Berechnung der Physik entkoppelt ist, wurden die entsprechenden Programmteile in eigene *Threads* ausgelagert, wie in Abbildung 3.2 zu sehen ist. Für die *WebSockets* sieht *Jetty* einen *Threadpool* vor, während die Verteilung von Nachrichten an die Clients in einem zusätzlichen *Thread* verarbeitet wird. Die Kommunikation

[3] http://www.eclipse.org/jetty/
[4] http://www.jbox2d.org/

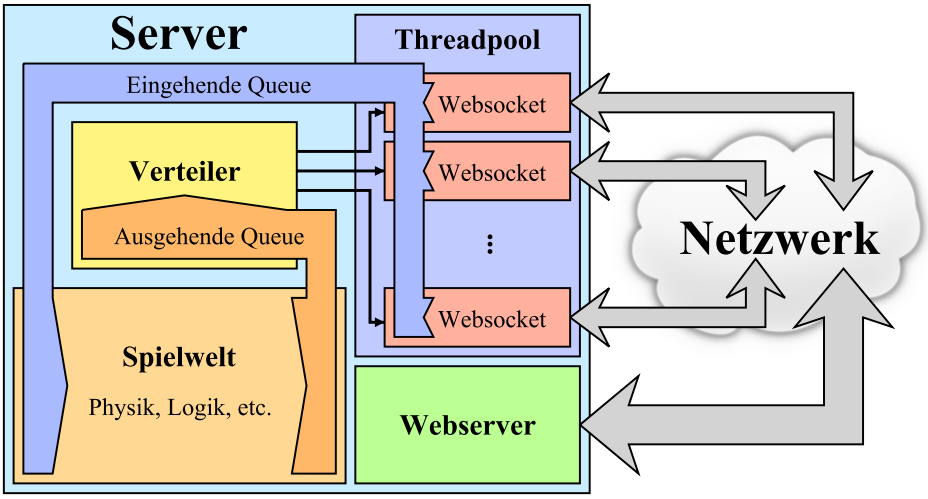


Abbildung 3.2: Der Server ist in verschiedene *Threads* aufgeteilt, welche mittels *Queues* miteinander kommunizieren.

zwischen den *Threads* erfolgt über zwei priorisierte *Queues*, die eingehende und ausgehende Nachrichten transportieren.

3.1.2 Client

Der Client wurde komplett in *JavaScript* (JS) umgesetzt und verwendet neueste HTML5 Technologien, wie z. B. *WebSockets* und *WebGL*, welche bisher nur durch *Mozilla Firefox* und *Google Chrome* vollständig unterstützt werden. Obwohl das Spiel 2D-Grafik verwendet, wurde für die Darstellung die bereits weit ausgereifte 3D-*Engine Three.js*[5] ausgewählt, da diese gut dokumentiert ist, aktiv entwickelt wird und unter anderem auch Features wie *Sprite-Rendering* bietet. Die restlichen notwendigen Programmteile wurden selbst entwickelt.

Um den Fokus auf die Skalierbarkeit des Spiels zu legen, wurde im ursprünglichen Projekt entschieden, dass der Client rein zur Darstellung der am Server berechneten Spielwelt dient und somit keine Logik simuliert und auch keine *Lag*-Kompensierenden Maßnahmen enthält. Der Client ist daher wie in Abbildung 3.3 gezeigt, relativ simpel aufgebaut. Da JS grundsätzlich in einem *Thread* abläuft, wurde für die Kommunikation mit dem Server ein *Web Worker* angedacht, der in einem getrennten *Thread* läuft. Wegen eines Bugs in *Firefox*[6] war dies jedoch nur teilweise möglich und so wurde zumindest die Deserialisierung der Servernachrichten ausgelagert.

[5] http://threejs.org/
[6] https://bugzilla.mozilla.org/show_bug.cgi?id=504553

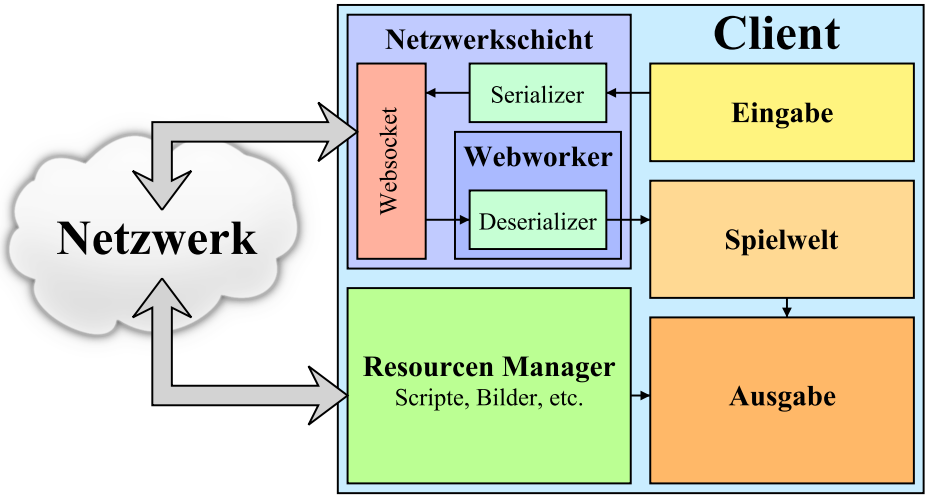


Abbildung 3.3: Der Client dient rein zur Ein- und Ausgabe.

Um während der Entwicklung die Performance bei vielen gleichzeitigen Nutzern einfach testen zu können, wurde ein Stresstest Script entwickelt, wofür es notwendig war, dass die Grafikausgabe komplett deaktivierbar wird und Benutzereingaben simuliert werden können.

3.1.3 Kommunikation

Client und Server sind mittels *WebSocket* zueinander verbunden. *WebSockets* sind eine Erweiterung des *Hypertext Transport Protokolls* (HTTP), welche eine von JS ausgehende, socket-artige Verbindung ermöglichen. Darunter findet im Prinzip eine persistente TCP-Verbindung zum Server statt, über jene die Datenpakete gesendet werden.

In der ursprünglichen Variante des Spiels, wurden die Daten mit einem selbst definierten Protokoll serialisiert, wodurch die zu sendende Datenmenge auf ein Minimum beschränkt wurde. Dies hatte zur Folge, dass Änderungen am Protokoll nur schwer umsetzbar waren, da für den entwickelten Prototyp die Serverkommunikation jedoch ständig verändert und weiterentwickelt werden musste, wurde das Protokoll durch *Apache Thrift*[7] ersetzt.

Bei *Thrift* handelt es sich um ein *Open Source*-Projekt, welches die Kommunikation zwischen verschiedenen Anwendungen erleichtern soll. Die Kommunikation wird, wie in Bild 3.4 gezeigt, in mehrere Schichten aufgeteilt, welche bei Bedarf einfach modifiziert oder ausgetauscht werden können. So wird für die Protokollebene neben JSON auch ein wesentlich effizienteres binäres Protokoll geboten, und für die Transportschicht können neben Socket-

[7] http://thrift.apache.org/

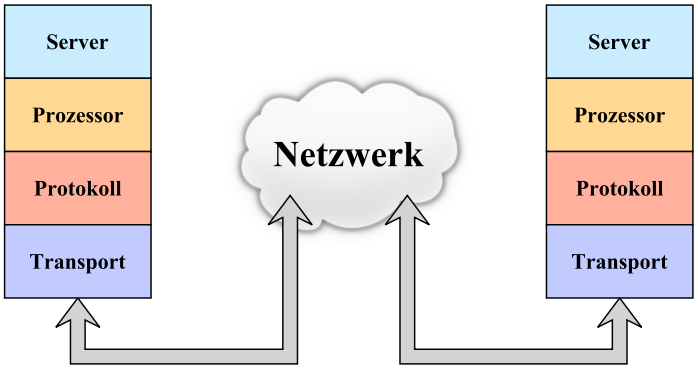


Abbildung 3.4: Darstellung der Thrift-Architektur.

verbindungen auch herkömmliche HTTP-Aufrufe gewählt werden.

Die in der Prozessorschicht verwendeten Datenstrukturen und Funktionen werden in einem sprachunabhängigen Format festgelegt und ein Compiler erstellt anschließend die notwendigen Klassen in der jeweiligen Zielsprache. Dadurch sinkt der Arbeitsaufwand und die Fehleranfälligkeit, wenn im Projekt mehrere verschiedene Programmiersprachen eingesetzt werden. Da das WebSocket-Protokoll nicht unterstützt wird, wurde die bereits bestehende Kommunikationsschicht weiterverwendet, was dank der sauberen Architektur von *Thrift* sehr einfach möglich ist. Des Weiteren ist in der JS-Bibliothek von *Thrift* das gewählte *compact protocol* nicht vorhanden, weshalb dieses von Java portiert wurde.

3.2 Testumgebung

Um die Auswirkung einer schlechten Netzwerkverbindung gezielt testen zu können, ist es notwendig, alle äußeren Einflüsse möglichst zu eliminieren. Hierzu zählen im Falle von *Lag*-Kompensation alle Quellen von Latenz, die in Kapitel 2 beschrieben werden. Um die Nachteile einer physikalischen Netzwerkverbindungen zu vermeiden, werden daher Server und Clients in derselben virtuellen Maschine (VM) betrieben und lokal miteinander verbunden.

Außerdem ist zu beachten, dass sich hohe CPU Auslastung oder langsame Festplattenzugriffe ebenfalls auf die Testergebnisse auswirken können. Dies ist vor allem bei der Verwendung einer VM problematisch, da diese meist nur einen Teil der Ressourcen des *Hostsystems* verwenden. Als *Hostsystem* wird deshalb ein hochwertiges Spiele-Notebook[8] mit ausreichend CPU Leistung, Arbeitsspeicher und schnellem Festplattenzugriff verwendet.

[8] i7-2670QM@2,2 GHz, 16 GB DDR3, Festplatte: M4-CT256M4SSD2

3.2.1 Verwendete Software

Zum Erstellen der VM wird das *Freeware*-Programm *VirtualBox* von *Oracle*[9] verwendet und als Betriebssystem wird *Ubuntu Server 64-bit*[10] gewählt. Die benötigten Netzwerkeigenschaften werden mit dem in Linux integrierten Programm `tc` (traffic control), künstlich erzeugt. Mit `tc` ist es möglich, alle Aspekte der Netzwerkkommunikation zu steuern, allerdings ist die Syntax relativ komplex, wie aus verschiedenen Anleitungen in [31, 44, 45] hervorgeht. Um mehrere verschiedene Netzwerkverzögerungen parallel simulieren zu können, ist es weiterhin notwendig, dass der Spielserver über mehrere Ports erreichbar ist. Dies kann mit Hilfe von *HAProxy*[11] einfach und flexibel erreicht werden. Da das Projekt bereits automatische Komponententests verwendet, werden die notwendigen Tests mit Hilfe von *JsTestDriver* (JSTD)[12] durchgeführt. JSTD startet dazu einen Server, auf den ein oder mehrere Browser verbunden werden, und führt in diesen den vorgegebenen Testcode aus. Als Browser wird *Mozilla Firefox* gewählt, da dieser neben *Google Chrome* der einzige Browser ist, der zu diesem Zeitpunkt alle benötigten HTML5-Features unterstützt und dem Autor bereits bekannt ist.

3.2.2 Simulation von *Lag*

Eine schlechte Netzwerkverbindung setzt sich, wie in [2, S. 69f.] beschrieben, aus *Latenz*, *Jitter* und *Paketverlusten* zusammen. Während Latenz und Paketverluste, wie bereits in Kapitel 2 beschrieben, die Zeit zwischen dem Senden und Empfangen von Daten beeinflussen, bezeichnet *Jitter* die Unterschiede in der Latenz zwischen aufeinander folgenden Übertragungen.

Wie aus [17] hervorgeht, wirkt sich *Jitter* im Vergleich zu Latenz jedoch geringer auf die empfundene Spielqualität aus. Aufgrund der zusätzlichen Komplexität bei der Auswertung der Ergebnisse, bei vergleichsweise geringer Auswirkung auf das Spielerlebnis, wird daher die Simulation von *Jitter* vernachlässigt. Da WebSockets auf TCP basieren, werden Pakete, die verlorengehen, automatisch erneut gesendet, weswegen sich Paketverluste bei dieser Art von Verbindung nur als Multiplikator für die Latenz auswirken. Für einfache Tests, die nicht auf reale Bedingungen angewiesen sind, können diese deshalb ebenfalls ignoriert werden.

Mehrere Latenzen gleichzeitig simulieren

Damit Kombinationen von verschieden schlechten Netzwerkbedingungen getestet werden können, ist es notwendig, den Netzwerkverkehr zum Server zu differenzieren. Die einfachste Möglichkeit, um dies zu erreichen, bietet

[9] https://www.virtualbox.org/
[10] http://www.ubuntu.com/server
[11] http://haproxy.1wt.eu/
[12] http://code.google.com/p/js-test-driver/

sich durch die Verwendung von mehreren Ports. Damit das Programm selbst nicht verändert werden muss, kann hierfür Proxy-Software wie z. B. *HAProxy* eingesetzt werden. *HAProxy* kann über den *Package-Manager* von *Ubuntu* installiert werden:

```
sudo apt-get install haproxy
```

Anschließend muss nur noch in der mitgelieferten Konfigurationsdatei in /etc/haproxy/haproxy.conf folgender Eintrag hinzugefügt werden:

```
listen latency-ports
        mode tcp
        option tcplog
        bind :8081-8200
        server gameserver 127.0.0.1:8080
```

Alle anderen `listen`-Einträge können, falls nicht benötigt, entfernt werden. Danach kann *HAProxy* gestartet werden, wodurch der auf Port 8080 liegende Server auf allen Ports von 8080 bis 8200 erreichbar wird.

```
sudo haproxy -f /etc/haproxy/haproxy.cfg
```

Testen der Latenz

Um auch testen zu können, ob die eingestellte Latenz wirklich wie erwartet funktioniert, wurde ein simples *Python*-Script entwickelt, welches mittels *Threads* einen Server und einen Client simuliert, die über die zu testenden Ports ein Paket hin und zurück senden.

Der Client speichert die aktuelle Zeit, sendet ein Paket, wartet auf die Antwort des Servers und speichert wiederum die Systemzeit.

```
time1 = time.time()
sock.send(' ')
sock.recv(1)
time3 = time.time()
```

Am Server wird auf das Paket gewartet, die Zeit gespeichert und anschließend eine Antwort gesendet.

```
conn.recv(1)
self.time2 = time.time()
conn.send(' ')
```

Da beide Threads im selben Script laufen, kann der Client auf die Zeit im Server-Thread direkt zugreifen, wodurch die Latenz in beide Richtungen berechnet und angezeigt werden kann.

```
time2 = self.server.time2
up = (time2 - time1) * 1000
down = (time3 - time2) * 1000
rtt = (time3 - time1) * 1000
print 'up: %.2f ms, down: %.2f ms, rtt: %.2f ms' % (up, down, rtt)
```

Um aussagekräftige Ergebnisse zu erhalten, wird nach Aufbau der Verbindung für eine kurze Zeit gewartet und der oben gezeigte Vorgang mehrmals wiederholt, mit jeweils einer weiteren Pause zwischen den Messungen. Die Ausgabe des Scripts sieht somit folgendermaßen aus:

```
up: 0.20 ms, down: 0.10 ms, rtt: 0.30 ms
up: 0.19 ms, down: 0.20 ms, rtt: 0.39 ms
up: 0.19 ms, down: 0.17 ms, rtt: 0.35 ms
-----------------------------------------------------------
result for 3 samples:
up:   min=0.19 ms, max=0.20 ms, mean=0.19 ms, stdev=0.01 ms
down: min=0.10 ms, max=0.20 ms, mean=0.16 ms, stdev=0.04 ms
rtt:  min=0.30 ms, max=0.39 ms, mean=0.35 ms, stdev=0.04 ms
```

Es werden zuerst die Ergebnisse der einzelnen Durchläufe angezeigt und zuletzt die statistische Auswertung. Das vollständige Script und eine Anleitung ist in Anhang B zu finden.

Latenzsimulation mittels `tc`

Mit `tc` ist es durch Definition von verschiedenen Regeln möglich, die Abarbeitung von Paketen innerhalb der Netzwerkadapter zu beeinflussen. Die gewünschten Netzwerkeigenschaften werden in sogenannten *Queueing Disciplines*[13] und Klassen festgelegt. Standardmäßig verwendet ein Adapter, der nicht speziell konfiguriert wurde, eine FIFO (First-In First-Out) Reihung, um Pakete zu transportieren.

Beim Festlegen der Regeln muss zwischen eingehenden Daten (*ingress*) und ausgehenden Daten (*egress*) unterschieden werden, da `tc` für *ingress* wegen der Funktionsweise des IP-Protokolls nur begrenzte Regeln zur Verfügung stellen kann, wie auch in [15] besprochen wird. Auf dem *Loopback*-Adapter spielt dies zwar keine Rolle, da jeder Datenverkehr als *egress* angesehen werden kann, um die simulierte Latenz jedoch auch außerhalb der VM beobachten zu können, wird ein *Intermediate Functional Buffer* (IFB)-Adapter eingesetzt, über welchen der eingehende Verkehr umgeleitet wird.

Die Befehle zum Einrichten zweier Dummy-Adapter lauten wie folgt:

```
modprobe ifb numifbs=2
ip link set dev ifb0 up
ip link set dev ifb1 up
```

Anschließend muss nur noch der eingehende Verkehr jeweils für eth0 und lo gespiegelt werden:

```
tc qdisc add dev eth0 handle ffff: ingress
tc filter add dev eth0 parent ffff: protocol ip u32 match u32 0 0 action
     mirred egress redirect dev ifb0
tc qdisc add dev lo handle ffff: ingress
tc filter add dev lo parent ffff: protocol ip u32 match u32 0 0 action
    mirred egress redirect dev ifb1
```

[13] englisch für Reihungs-Disziplinen

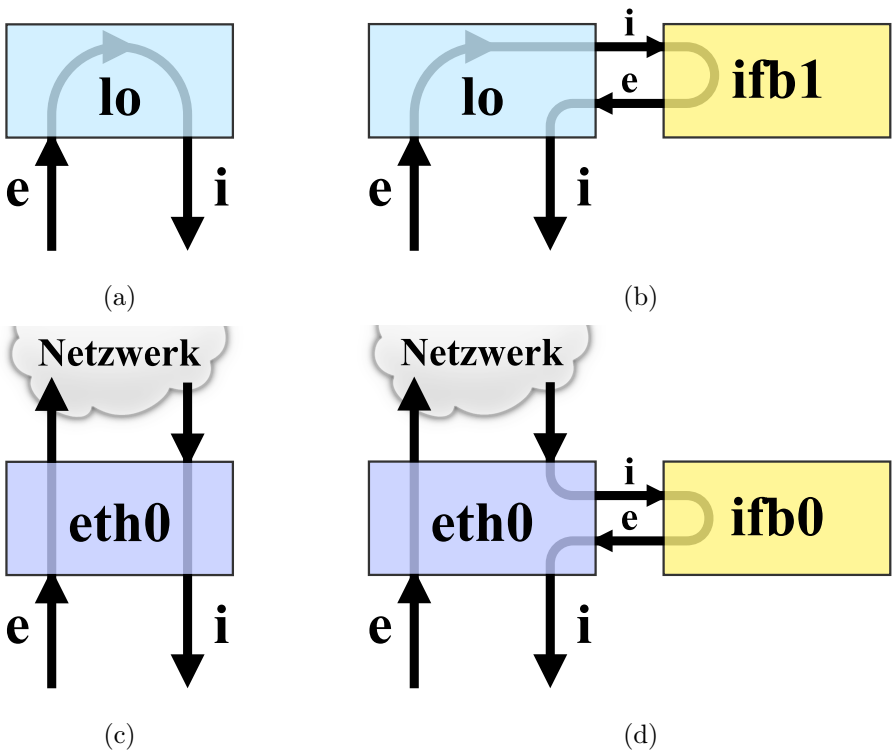


Abbildung 3.5: Schematische Darstellung der Datenströme. `lo` ohne `ifb` (a), `lo` mit `ifb` (b), `eth0` ohne `ifb` (c) und `eth0` mit `ifb` (d).

Auf diese Weise wird der Datenverkehr, wie in Abbildung 3.5 dargestellt, umgeleitet und es können somit alle Regeln, die für den ausgehenden Verkehr zulässig sind, auch auf eingehende Datenströme angewandt werden.

Um nun die Latenzen gezielt auf einzelne Ports anwenden zu können, muss eine klassenbasierte Disziplin gewählt werden, damit anschließend mittels Filter ein Port eingestellt werden kann. Dazu wird der *Hierarchical Token Bucket* (HTB) ausgewählt, welcher für jeden Port jeweils eine HTB Klasse mit entsprechenden Filterregeln zugewiesen bekommt. Wie bereits erwähnt, würde es für den lokalen *Loopback* zwar ausreichen, die Regeln für beide Richtungen auf demselben Adapter zu definieren. Da sich dadurch die notwendigen Befehle jedoch von denen für `eth0` unterscheiden würden und das Anlegen der Regeln schlussendlich von einem *Bash*-Script übernommen werden soll, werden für die lokale und externe Variante jeweils dieselben Regeln verwendet.

Damit Klassen auf einen Adapter zugewiesen werden können, muss als erstes jeweils der Wurzelknoten der Hierarchie angelegt werden. Im Nachfolgenden werden die hierfür notwendigen Regeln am Beispiel von `lo` und `ifb1` gezeigt.

```
tc qdisc add dev lo handle 1: root htb
tc qdisc add dev ifb1 handle 1: root htb
```

Danach kann die Verzögerung für die einzelnen Ports angelegt werden.

```
tc class add dev lo parent 1: classid 1:1 htb rate 100Mbps
tc filter add dev lo protocol ip parent 1: prio 1 u32 match ip sport
    8081 0xffff classid 1:1
tc qdisc add dev lo parent 1:1 handle 10: netem delay 50ms

tc class add dev ifb1 parent 1: classid 1:1 htb rate 100Mbps
tc filter add dev ifb1 protocol ip parent 1: prio 1 u32 match ip dport
    8081 0xffff classid 1:1
tc qdisc add dev ifb1 parent 1:1 handle 10: netem delay 50ms
```

Da die zulässige Datenrate bei einem HTB zwingend angegeben werden muss, wurde der Wert mit 100 Mbps groß genug gewählt, damit das Ergebnis nicht unerwünscht beeinflusst wird. Den beiden Interfaces wird jeweils die gewünschte Latenz in Millisekunden zugewiesen, woraus sich die RTT ergibt.

Die oben angegebenen Befehle wurden in einem *Bash*-Script zusammengestellt, welches in Anhang C zu finden ist. Dadurch können mit einem simplen Befehl die gewünschten Latenzen einfach und flexibel angewendet werden.

```
./portdelay.sh 100 300 100
```

Dem Script wird die niedrigste und die höchste RTT und die Schrittweite übergeben, wodurch nach Ausführung des gezeigten Befehls die Ports 8081 bis 8083, je eine RTT von 100 ms, 200 ms und 300 ms aufweisen. Dies kann nun mit dem in Sektion 3.2.2 gezeigten Testscript überprüft werden, indem dieses für die drei Ports je einmal aufgerufen wird. Zusätzlich werden auch Port 8080 und 8084 getestet, welche keine Latenz aufweisen sollten.

```
python latencytest.py -r 100 -s 8070 8070
python latencytest.py -r 100 8081
python latencytest.py -r 100 8082
python latencytest.py -r 100 8083
python latencytest.py -r 100 8084
```

Der Parameter `r` gibt dabei an, wie viele Testpakete gesendet werden sollen und `s`, auf welchen Port der Server gebunden wird.

Wie aus den Endergebnissen in Tabelle 3.1 hervorgeht, liegt die RTT ohne eingestellter Latenz auf einem Port der nicht durch *HAProxy* weitergeleitet wird, bei einem Durchschnitt (μ) von 0,35 ms, mit einer Standardabweichung (σ) von 0,03 ms. Durch den Einsatz des Proxys steigt μ auf 0,56 ms und σ auf 0,07 ms an. Mit den Regeln von tc wird der Unterschied zur jeweils eingestellten RTT um weitere 0,8 ms überschritten, während σ auf Werte von etwas über 0,5 ms ansteigt. Die Abweichungen liegen somit weit unter 10% und die erzeugten Latenzen sind daher für die Tests mit dem *Framework* ausreichend genau.

Tabelle 3.1: Latenz für verschiedene Einstellungen bei je 100 Samples.

Richtung	Min (ms)	Max (ms)	μ (ms)	σ (ms)
0 ms ohne HAProxy				
auf	0,16	0,29	0,19	0,02
ab	0,09	0,26	0,17	0,02
beide	0,30	0,48	0,35	0,03
0 ms mit HAProxy				
auf	0,23	0,53	0,32	0,06
ab	0,15	0,30	0,24	0,02
beide	0,39	0,79	0,56	0,07
100 ms				
auf	50,34	51,57	50,64	0,39
ab	50,31	51,70	50,69	0,44
beide	100,69	102,76	101,33	0,56
200 ms				
auf	100,33	101,59	100,67	0,40
ab	100,30	101,60	100,61	0,39
beide	200,71	203,05	201,28	0,57
300 ms				
auf	150,33	152,41	150,68	0,43
ab	150,32	155,01	150,77	0,78
beide	300,65	305,78	301,45	0,88

3.2.3 Automatisiertes Testen

JSTD ist ein Komponententest-*Framework* für JS, welches Aufgaben wie das Starten der Browser und Aufrufen der Testfunktionen erleichtert und Funktionen zum Abbilden von Abläufen und Testen von Annahmen zur Verfügung stellt. Um unter Linux einen Browser mit dem JSTD-Server verbinden zu können, ist es jedoch notwendig, dass dieser ohne Bildschirmausgabe ausgeführt werden kann. Die Installation des dafür notwendigen Setups mit Firefox wird in Anhang D detailliert beschrieben. Mit diesen Vorkehrungen sollte einem erfolgreichen automatischen Testdurchlauf nichts mehr im Wege stehen.

JsTestDriver Testlauf

Da JsTestDriver ein Java-Programm ist, ist es notwendig, dass eine Java Laufzeitumgebung installiert wird.

```
sudo apt-get install openjdk-7-jre
```

Um nun einen Probelauf mit JSTD durchführen zu können, muss ein minimaler Testfall festgelegt werden. Dieser besteht in der Regel aus drei Dateien. Eine Datei HelloWorld.js enthält den zu testenden Code:

```
1 function helloWorld() {
2   return "Hello World!";
3 }
```

Eine zweite Datei HelloWorldTest.js definiert den Testfall.

```
1 TestCase("HelloWorldTestCase", {
2   testHelloWorld: function() {
3     var hello = helloWorld();
4     assertEquals("Hello World!", hello);
5   }
6 });
```

Die letzte Datei HelloWorldTest.conf konfiguriert JSTD und gibt an welche Dateien geladen und getestet werden sollen.

```
1 server: http://localhost:42442
2 load:
3   - HelloWorld.js
4 test:
5   - HelloWorldTest.js
```

Danach kann der Test mit folgendem Befehl ausgeführt werden.

```
xvfb-run java -jar JsTestDriver.jar --port 42442 --browser firefox \
--config HelloWorldTest.conf --tests all
```

Wenn die Konfiguration korrekt ist und JSTD funktioniert, erscheint auf der Konsole nun folgende Ausgabe.

```
setting runnermode QUIET
.
Total 1 tests (Passed: 1; Fails: 0; Errors: 0) (1.00 ms)
  Firefox 18.0 Linux: Run 1 tests (Passed: 1; Fails: 0; Errors 0) (1.00
    ms)
```

3.3 Anforderungen an das Spiel

Damit die *Lag*-Kompensation sinnvoll getestet werden kann, muss das zu testende Spiel einige Features unterstützen, die im Nachfolgenden erläutert werden. Da *Swords 'n' Bombs* von Grund auf selbst entwickelt wurde, konnten notwendige Änderungen relativ einfach implementiert werden. Für umfangreichere, bereits bestehende Spiele, kann es jedoch unter Umständen sehr schwierig sein, im Nachhinein notwendige Features einzubauen.

3.3.1 Serialisierbarkeit

Die grundlegende Funktion, die für das Testen von *Lag*-Kompensation notwendig ist, ist eine Möglichkeit, die Zustände der Spielwelt von allen beteiligten Teilnehmern vergleichen zu können. Dafür ist es unumgänglich, dass die Spielwelt auf Client- und Serverseite zu jedem *Frame*[14] gesammelt und exportiert werden kann. Da in *Swords 'n' Bombs* auf Client- und Serverseite unterschiedliche Sprachen und Implementationen verwenden, wird der Zustand am Server durch einen Client, der ohne künstliche Latenz verbunden ist, dargestellt. Dadurch reicht es aus, wenn der Client den entsprechenden Code zum Serialisieren und Ausgeben der Daten enthält. Es müssen somit keine weiteren Maßnahmen getroffen werden, um die Daten des Servers in dieselbe Form wie die Daten der Clients zu bringen.

Um den Einsatz des *Frameworks* mit anderen Spielen sinnvoll zu ermöglichen, sollten die Daten in einer abstrakten Form übergeben werden, damit nicht für jedes weitere Spiel die Auswertung geändert werden muss. Für die in dieser Arbeit beschriebene erste Version des *Frameworks* reicht jedoch die gewählte Methode aus.

3.3.2 Parallelisierbarkeit

Um ein Spiel in einer geschlossenen Umgebung testen zu können, müssen mehrere Instanzen des Clients und der Server auf demselben Rechner ausführbar sein. Falls Clients und Server zu diesem Zweck in verschiedenen VMs ausgeführt werden, ist es unbedingt notwendig, dass alle Instanzen des Spiels dieselbe Uhrzeit verwenden, damit gespeicherte *Frame*-Zustände im Nachhinein zueinander zugeordnet werden können. Dies kann einerseits durch Synchronisieren der Systemuhr erreicht werden, oder falls das Spiel einen eigenen Zeitgeber verwendet, auch innerhalb des Spiels gelöst werden, wie in [18] beschrieben wird.

Weil *Swords 'n' Bombs* mit JSTD getestet wird und somit mehrere Clients im selben Browserfenster ausgeführt werden, ist besonders darauf zu achten, dass diese sich nicht gegenseitig beeinflussen, wie es z. B. durch die Verwendung von globalen Variablen oder Singletons passieren könnte.

Nachdem mehrere Instanzen der Software auf demselben System laufen müssen, ist es vor allem für aufwendigere Spiele notwendig, dass die verbrauchte Leistung auf ein Minimum reduziert werden kann. Idealerweise können zu diesem Zweck jegliche unbenötigten Teile des Spiels, wie z. B. audiovisuelle Ausgabe, deaktiviert werden, ansonsten sollten zumindest die Grafikeinstellungen auf die niedrigste Stufe gesenkt werden.

Um mehrere Clients gleichzeitig, automatisch steuern zu können, ist es außerdem notwendig, dass über ein Interface Befehle an das Programm übergeben werden können. Im Falle von *Swords 'n' Bombs* wurde bereits während

[14] englisch für Einzelbild

der Entwicklung auf diese Punkte geachtet, wodurch problemlos bis zu siebzig Clients gleichzeitig im selben Browserfenster ausgeführt werden können.

3.3.3 Netzwerkverbindung

Wie bereits in Abschnitt 3.2.2 beschrieben, werden *Lags* auf verschiedene Ports angewandt. Daher ist es notwendig, dass im Client eingestellt werden kann, auf welchem Port die Verbindung zum Server hergestellt wird. Außerdem können mit dem derzeitigen Setup nur Spiele, die über TCP kommunizieren, getestet werden, da *HAProxy* UDP Verkehr nicht umleiten kann.

3.4 Implementation von Szenarien

Obwohl JSTD eigentlich für Komponententests gedacht ist, können durch die Verwendung von asynchronen Testfällen jedoch auch Integrationstests abgebildet werden. Dazu wird eine Warteschlange angeboten, die es ermöglicht, den Test in einzelne Schritte zu unterteilen, welche bei Aufruf der übergebenen Funktion zum nächsten Schritt wechselt, oder nach einer gewissen Zeit der Inaktivität den Test abbricht. Dadurch können einfach und effektiv relativ komplizierte Abläufe simuliert werden, ohne dass zusätzliche Software entwickelt werden muss.

Eine zu testende Spielsituation wird in Form eines `AsyncTestCase` implementiert.

```
1 AsyncTestCase("SampleTestCase", {
2   setUp : function() {
3     this.clients = [];
4   },
5
6   tearDown : function() {
7     while (this.clients.length > 0) {
8       this.clients.pop().destroy();
9     }
10  },
11
12  testSituation : function(queue) {
13  ...
14  },
15 });
```

Die Definition der Tests wird in Zeile 1, in Form eines JS-Objekts an die `AsyncTestCase`-Funktion von JSTD übergeben. Das Objekt enthält eine oder mehrere Funktionen, deren Name mit `test` beginnt. Wie auch von anderen *Unittest-Frameworks* bekannt, können eine `setUp`- und eine `tearDown`-Methode definiert werden, welche jeweils vor und nach jedem Test aufgerufen werden.

Damit ein Client sicher vom Server entfernt wird, ist es ratsam, die Verbindung in der `tearDown`-Methode zu terminieren, da diese auch beim Fehl-

schlag eines Tests aufgerufen wird. Zu diesem Zweck wird in Zeile 3 ein Array initialisiert, welches Referenzen auf alle erstellten Spieler sammelt. In Zeile 8 werden diese in einer Schleife zerstört, was dazu führt, dass ihre Verbindung zum Server beendet wird.

Die Aktionen der Clients werden im Testspiel durch Simulation von Benutzereingaben gesteuert. Um den Testcode und Spielcode möglichst getrennt zu halten, wird hier auf ein Adapter-Entwurfsmuster gesetzt.

Für einen simplen Testfall, in dem die Spielfigur einfach nach rechts läuft, wird innerhalb einer solchen Test-Methode zuerst ein Spieler angelegt.

```
1 testWalkToRight : function(queue) {
2   var testuser1 = client("testuser#1");
```

Die Variable `queue` wird, wie bereits erwähnt, von JSTD bereitgestellt und enthält eine Methode `call`, die zum Anlegen der einzelnen Schritte dient.

```
3   queue.call("initialize client", function(callbacks) {
4     testuser1.teleportTo(2275, 2150).realize(callbacks, this.clients);
5   });
```

Das Initialisieren des Spiels und der Serververbindung ist in der `realize`-Methode versteckt. Die `teleportTo`-Methode wird vor dem Initialisieren aufgerufen, damit der Spieler sofort an der richtigen Position erscheint.

```
6   queue.call("wait for client to become ready", function(callbacks) {
7     setTimeout(callbacks.noop(), 500);
8   });
9   queue.call("send move command", function(callbacks) {
10    testuser1.moveToDirection(1, 0).flush();
11  });
```

Bevor der Befehl zum Bewegen gegeben wird, wird noch für eine kurze Zeit gewartet, damit der Server nach dem Login alle notwendigen Daten an die Spieler senden kann. Mit `flush` werden alle angesammelten Befehle an den Server gesendet.

```
12  queue.call("wait before disconnecting", function(callbacks) {
13    setTimeout(callbacks.noop(), 1500);
14  });
15 },
```

Nach dem Veranlassen der Bewegung wird noch für eine bestimmte Zeit (z. B. eineinhalb Sekunden) gewartet, damit das nachfolgende Verhalten erfasst wird. Danach wird in der `tearDown`-Methode die Verbindung automatisch beendet.

Alternativ könnte man `teleportTo` auch später aufrufen.

```
1   queue.call("initialize client", function(callbacks) {
2     testuser1.realize(callbacks, this.clients);
3   });
```

Dadurch wird aber auch ein zusätzlicher Aufruf von `flush` in einem getrennten Aufruf von `queue.call` erforderlich.

```
1   queue.call("send teleport command", function(callbacks) {
2     testuser1.teleportTo(2275, 2150).flush(callbacks);
3   });
```

3.5 Testablauf

In dieser Sektion wird beschrieben, welche Schritte notwendig sind, um ein bestimmtes Szenario bei verschiedenen Latenzen zu testen und wie die Daten für die weitere Verwendung gesammelt werden.

3.5.1 Vorbereitung

Bevor die Tests durchgeführt werden können, muss noch festgelegt werden, welche Verzögerungen getestet werden sollen und somit welche Ports für die Verbindung zum Server benutzt werden. Daher wird nach Festlegen der Netzwerkeigenschaften mit portdelay.sh eine Datei hostconfig.js erstellt, die ein Objekt mit allen *Host*-Adressen enthält.

```
1 var Host = {
2   WITH_100MS_LAG : "localhost:8081",
3   WITH_200MS_LAG : "localhost:8082",
4   WITH_300MS_LAG : "localhost:8083",
5 };
```

Eine Möglichkeit, wie die Testfälle für jede Kombination von Latenz definiert werden können, bietet sich durch die Verwendung von verschachtelten Schleifen.

```
1 var testCase = {}, c = 0;
2 for ( var i in Host) {
3   for ( var j in Host) {
4     (function(host1, host2) {
5       testCase["testShootOnPassingEnemy" + (c++)] = function(queue) {
6         var observer = client("observer");
7         var testuser1 = client("testuser#1", host1);
8         var testuser2 = client("testuser#2", host2);
9         ...
10      };
11    })(Host[i], Host[j]);
12  }
13 }
```

In Zeile 1 wird ein Objekt, welches die Testfälle enthält und später an `AsyncTestCase` übergeben wird, und eine Zählvariable initialisiert. Anschließend wird in Zeile 2 und 3 jeweils über die in hostconfig.js definierte Auflistung der möglichen *Hosts* iteriert. Da der Code in der Testmethode in Zeile 5 erst aufgerufen wird, nachdem die Schleifen bereits verlassen wurden, müssen die Werte in einen eigenen Gültigkeitsbereich übergeben werden. Dazu wird in Zeile 4 eine anonyme Funktion angelegt, die in Zeile 11 sofort aufgerufen wird und die beide *Hosts* übergeben bekommt. Dadurch ändern sich

die Werte für `host1` und `host2` nicht, wenn die Schleife weiterläuft. Der Beobachter in Zeile 6 bekommt keinen speziellen *Host* übergeben und verbindet sich somit über Port 8080, ohne Latenz zum Server.

Damit sich nun die Testclients auf den Server verbinden können, muss dieser noch gestartet werden.

```
java -server -Xmx256M -Xms128M -XX:+UseConcMarkSweepGC -XX:+
    CMSIncrementalPacing -XX:ParallelGCThreads=2 -XX:+AggressiveOpts -
    Dorg.eclipse.jetty.util.log.class=java.util.logging.Logger -jar
    websocketgame.jar
```

Es werden einige Parameter an die JRE übergeben, welche die Performance des Servers beeinflussen und eine Klasse für die Ausgaben von *Jetty* festlegen.

3.5.2 Datensammlung

Während des Tests werden die Zustände der jeweiligen Clients mit Hilfe eines `FrameRecorder`-Objekts gesammelt. Dieser bekommt zu diesem Zweck beim Erstellen das Client-Objekt übergeben und erhält dadurch Zugriff auf dessen inneren Zustand.

Aus technischen Gründen funktionieren innerhalb von JSTD die von JS bereitgestellten Methoden zum regelmäßigen Aufrufen von Funktionen nicht wie erwartet, solange sie nicht direkt innerhalb der Testmethoden aufgerufen werden. Daher wird ein *Frame* aufgezeichnet, sobald auf dem *Observer*-Client ein `POSITION_UPDATE` Paket empfangen wird, wodurch die *Framerate* der des Servers entspricht und zusätzlich automatisch auf allen Clients synchronisiert ist. Es ist jedoch zu beachten, dass Positionsupdates nur gesendet werden, solange sich ein Spieler bewegt. Aus diesem Grund wird der Beobachter vor Beginn des eigentlichen Tests damit beauftragt, in eine Richtung zu laufen.

Die Methode `record` wird somit zu jedem *Frame* aufgerufen und speichert eine fortlaufende Nummer, die aktuelle Zeit aus der Sicht des Clients und die Position, Geschwindigkeit und Beschleunigung aller Spieler und Entitäten.

```
1 this.record = function() {
2   var frame = {
3     number : ++frameNumber,
4     time : core.playerManager.lastUpdateTime,
5     players : core.playerManager.getState(),
6     entities : core.entityManager.getState(),
7     messages : core.backendConnection.getState()
8   };
9   frameStates.push(frame);
10 };
```

Die Aufzeichnung kann mittels `start`- und `stop`-Funktion gesteuert werden und, um die Ergebnisse nicht zu stark zu beeinflussen, wird in den `getState`-Methoden jeweils nur ein simples Abbild der Daten gespeichert. Im `PlayerManager` wird über alle Spieler iteriert und jeweils eine Kopie der

relevanten Daten erstellt und in einem Array abgelegt.

```
this.getState = function() {
  var state = [];
  for ( var id in players) {
    var player = players[id];
    state.push({
      id : id,
      position : player.getPosition().clone(),
      velocity : player.getVelocity().clone(),
      updateTime : player.getLastUpdateTime()
    });
  }
  return state;
};
```

Der `EntityManager` funktioniert analog dazu und zeichnet anstelle der Spieler alle Nicht-Spieler-Entitäten auf. In der `BackendConnection` werden alle ankommenden und abgesendeten Nachrichten in zwei Listen abgelegt, die in `getState` durch leere Arrays ausgetauscht und zurückgegeben werden.

```
  that.getState = function() {
    var incoming = incomingList;
    incomingList = [];
    var outgoing = outgoingList;
    outgoingList = [];
    return {
      incoming : incoming,
      outgoing : outgoing
    };
  };
```

3.5.3 Datenkonvertierung

Die Aufbereitung der gesammelten Rohdaten findet nach Abschluss des eigentlichen Tests statt. Es wird dabei eine Liste mit allen Entitäten und Spielern, die in den Daten vorkommen, angelegt. Des Weiteren wird auch eine Liste mit allen Paketen, die zum Server gehen, und den Verknüpfungen zu den entsprechenden Antwortpaketen erstellt.

Anschließend werden die gesammelten und vorbereiteten Daten als JSON auf der JSTD-Konsole ausgegeben.

```
jstestdriver.console.log("##begin##");
testuser1.printRecord(observerId);
testuser2.printRecord(observerId);
observer.printRecord(observerId);
jstestdriver.console.log("##end##");
```

Die Zeichenketten `##begin##` und `##end##` markieren jeweils den Anfang und das Ende der Ergebnisse eines Tests im JSTD-Report. Die einzelnen Datenbestände der Clients und Server werden zusätzlich mit `##viewdata##`

markiert, um sie von anderen Ausgaben, die während des Tests getätigt werden, unterscheiden zu können.

3.6 Auswertung der Daten

Nach welchen Kriterien nun die vorliegenden Daten beurteilt werden können, ist nicht einfach zu beantworten, da der Einfluss der Latenz auf das Spielerlebnis subjektiv ist und somit stark vom Genre des Spiels und der Situation, in der sich ein Spieler befindet, abhängt. Eine Möglichkeit, wie die Qualität gemessen werden kann, wird in [13] beschrieben, wo eine formelle Definition für die *Konsistenz* und *Korrektheit* von Netzwerk-Spielen beschrieben wird. Es wird davon ausgegangen, dass zwischen dem Auftreten von Inkonsistenzen und der Reaktion von derartigen Applikationen eine Verbindung besteht und diese gegeneinander ausgetauscht werden können.

Die Konsistenz und Korrektheit des Spiels kann für jeden beliebigen Zeitpunkt festgestellt werden, solange alle Benutzereingaben von allen Teilnehmern empfangen wurden. Zu diesem Zweck wird die Empfangsfunktion R einer Ansicht i als

$$R_i(t, o_{j,t^0,t^*}) = \begin{cases} \textit{falsch} & \text{für Erhalt von } o_{j,t^0,t^*} \text{ an } i \text{ nach } t, \\ \textit{wahr} & \text{andernfalls} \end{cases} \tag{3.1}$$

definiert. Eine Aktion, die von Client j zur Zeit t^0 ausgeführt wurde und eine maximale Lebensdauer bis t^* besitzt, wird dabei als o_{j,t^0,t^*} bezeichnet. Dieser Ausdruck ist somit *falsch*, solange die Aktion zum Zeitpunkt t noch nicht empfangen wurde.

Die Konsistenz der virtuellen Welt kann danach mit dem Ausdruck

$$\begin{aligned} &\forall t, i, j \mid \forall t^* \leq t, o_{w,t^0,t^*} \in O \mid \\ &R_i(t, o_{w,t^0,t^*}) \wedge R_j(t, o_{w,t^0,t^*}) \Rightarrow (s_{i,t} = s_{j,t}) \end{aligned} \tag{3.2}$$

beschrieben werden. Das bedeutet, das Spiel ist zu einem beliebigen Zeitpunkt t konsistent, wenn alle Clients, bei denen alle ausstehenden Aktionen vorliegen, eine identische Spielwelt s aufweisen. Clients, die eine Aktion noch nicht erhalten haben, werden nicht berücksichtigt, da ansonsten dieses Konsistenzkriterium niemals erfüllt werden könnte.

Die Korrektheit eines Spiels wird als

$$\begin{aligned} &\forall t, i \mid \forall t^* \leq t, o_{w,t^0,t^*} \in O \mid \\ &R_i(t, o_{w,t^0,t^*}) \Rightarrow (s_{i,t} = s_{P,t}) \end{aligned} \tag{3.3}$$

festgelegt. Im Gegensatz zur Konsistenz wird der Zustand der Clients hier mit dem Idealzustand der Spielwelt verglichen. Der Zustand $s_{P,t}$ beschreibt somit einen Teilnehmer, der alle Änderungen ohne Latenz übermittelt bekommt und somit zu jedem Zeitpunkt konsistent ist. Auf diese Weise kann

die Funktionalität der *Lag*-Kompensation zu jedem Zeitpunkt, zu dem keine Aktionen ausstehen, überprüft werden.

Für Fälle, in denen die Kriterien nicht angewendet werden können, kann die in [7] beschriebene Methode zur Berechnung der *View Inconsistency*[15] (VI) herangezogen werden. Auch hier wird von einer Austauschbeziehung zwischen der Dauer einer Aktion und der räumlichen Abweichung ausgegangen, was sich mit den Behauptungen in [13] deckt. Es wird jedoch ein anderer Ansatz verfolgt, bei dem nicht auf den Gesamtzustand der Spielwelt geachtet wird, sondern die Auswirkungen von einzelnen Aktionen auf bestimmte Entitäten überprüft wird.

Auch hier wird angenommen, dass in solchen Situationen niemals perfekte Synchronizität zwischen allen Teilnehmern erreicht werden kann, solange Latenz vorhanden ist. Das Spielerlebnis einzelner Spieler und die Fairness zwischen allen Spielern soll jedoch durch den Einsatz von *Lag*-Kompensation verbessert werden können. Um die Auswirkungen von *Lag* auf eine Aktion berechnen zu können, werden für eine Situation die drei Metriken *Reaktion* (R), *Präzision* (P) und *Unfairness* (U) definiert. Während R und P die Situation für einen einzelnen Spieler widerspiegeln, gibt U einen Wert an, der für alle Spieler gültig ist.

Die *Reaktion* ist die Zeitspanne, die eine Aktion benötigt, um für einen Spieler sichtbar zu werden, nachdem er sie in Auftrag gegeben hat. Sie kann somit als

$$R = t_1 - t_0 = t_{RTT} \tag{3.4}$$

berechnet werden, wobei t_0 der Zeitpunkt der Eingabe und t_1 der Zeitpunkt der Ausgabe beim Spieler ist, weshalb der Wert ohne Einsatz von *Lag*-Kompensation der RTT entsprechen.

Die *Präzision* bezeichnet die Differenz im Abstand zwischen zwei Entitäten e_1 und e_2, in der Ansicht i des Spielers zum Zeitpunkt 0, zu dem er die Aktion auslöst und dem autoritativen Zustand s am Server, zu dem Zeitpunkt 1, zu dem die Aktion dort ankommt. Sie wird damit als

$$P = |\vec{p}_{i,e_1,0} - \vec{p}_{i,e_2,0}| - |\vec{p}_{s,e_1,1} - \vec{p}_{s,e_2,1}| = |\vec{v}_{e_1} - \vec{v}_{e_2}| \cdot t_{RTT} \tag{3.5}$$

festgelegt. Das bedeutet, solange keine Kompensationsalgorithmen im Einsatz sind, sieht der Spieler einen Zustand, der eine halbe RTT alt ist. Außerdem kommt sein Befehl erst eine weitere halbe RTT später am Server an. Dadurch ist P bei einer konstanten Geschwindigkeit der Entitäten ebenfalls mit der RTT gleichzusetzen.

Aus diesen beiden Werten ergibt sich die VI-Metrik

$$\Phi = \sqrt{P^2 + R^2}, \tag{3.6}$$

die den objektiven Zustand des Spiels wiedergibt. Ein Spieler nimmt P und R jedoch verzerrt wahr, daher müssen die beiden Werte relativiert werden,

[15] englisch für Ansichtsinkonsistenz

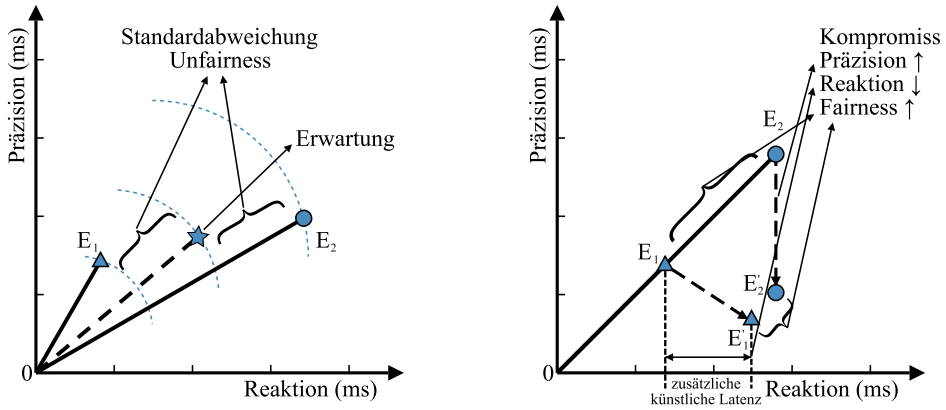


Abbildung 3.6: Fairnessraum nach [7], (a) Unfairness zwischen Spieler E_1 und E_2, (b) Austausch von P gegen R zum Reduzieren von U.

um die subjektive VI-Metrik

$$\phi = \sqrt{\left(\frac{P}{l}\right)^2 + \left(\frac{R}{\lambda}\right)^2} \tag{3.7}$$

zu erhalten. Durch l wird die Größe der vom Spieler anvisierten Entität bezeichnet, was bedeutet, dass für größere Spielfiguren der subjektive Einfluss der Latenz geringer ausfällt. Die Reaktion wird dagegen durch die Lebensdauer λ der Aktion, die bereits in Gleichung 3.1 als t^* aufgeführt wird, beeinflusst. Beispielsweise kann ein Projektil, das sich langsam auf den Spieler zubewegt, auch erst später in seiner Ansicht auftauchen, solange ihm genug Zeit zum Ausweichen bleibt.

Die *Unfairness* U stellt schlussendlich den Unterschied der VI mehrerer Spieler zueinander dar und wird als Standardabweichung aller VI-Werte berechnet. Die Formel lautet somit

$$U = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(\Phi_i - \left(\frac{1}{N}\sum_{i=1}^{N}\Phi_i\right)^2\right)}. \tag{3.8}$$

Da die VI im unkompensierten Zustand direkt von der RTT abhängt, besteht somit zwischen zwei Spielern mit unterschiedlicher Latenz eine Unfairness. Dieser Zusammenhang ist im sogenannten *Fairness Space*[16], der in Abbildung 3.6 gezeigt wird und die VI als Funktion der Präzision über die Reaktion darstellt, einfach ersichtlich. Solange keine *Lag*-Kompensationsalgorithmen

[16]englisch für Fairnessraum

eingesetzt werden, sind P und R gleich groß und die Unfairness zwischen Spieler E_1 und E_2 ist in diesem Fall proportional zur Differenz in der RTT.

Zusammenfassend kann also festgestellt werden, dass das Ziel von *Lag*-Kompensation die Verbesserung des Spielerlebnisses ist. Dazu wird die Korrektheit und Konsistenz der Spielzustände aller Teilnehmer sichergestellt und in Situationen, in denen diese Kriterien nicht geprüft werden können, die Fairness zwischen den Spielern erhöht, indem Präzision gegen Reaktion getauscht wird.

3.7 Darstellung der Daten

Damit leicht ersichtlich wird, ob sich die Qualität der *Lag*-Kompensation nach einer Änderung am Spiel verbessert oder verschlechtert hat, werden daher folgende Fragen gestellt:

- Ist der Ablauf konsistent?
- Ist der Ablauf korrekt?
- Wie ist der subjektive Eindruck des Spielers?
- Wie fair ist das Spiel?

Die Antworten sollen in Form eines HTML-Reports gegeben werden, der mit Hilfe der zuvor gesammelten Daten erstellt wird.

3.7.1 Erstellen des Reports

Um die gesammelten Zustände der Spielwelt, die in Form eines JSTD-Reports vorliegen, in eine nutzbare Form zu bringen, muss dieser zuerst konvertiert werden. Dies erfolgt mit Hilfe von *Apache Ant*[17], welches viele Funktionen zur Automatisierung von Abläufen zur Verfügung stellt. Unter anderem wird durch den `scriptdef`[18]-Block eine Möglichkeit bereitgestellt, um JS mit *Ant* verwenden zu können. Zusätzlich zu herkömmlicher JS-Syntax, können in derartigen Scripts auch Java-Methoden aufgerufen werden, wodurch das Lesen und Schreiben von Dateien möglich wird.

Die Konvertierung erfolgt somit mit Hilfe eines Scripts, welches die Log-Ausgaben im Report nach den zuvor festgelegten Zeichenketten durchsucht und damit anschließend die Vorlage des Reports befüllt.

```
1 var logLines = input.split("\\[LOG\\] ");
```

Da JSTD-Ausgaben Zeilenumbrüche enthalten können, ist es notwendig, die Daten nach anderen Kriterien zu unterteilen. Hierfür bietet sich die Zeichenkette `[LOG]` an, welche von JSTD vor jede Ausgabe gestellt wird.

```
2 for ( var i in logLines) {
3   var line = logLines[i];
```

[17] http://ant.apache.org/
[18] http://ant.apache.org/manual/Tasks/scriptdef.html

```
4   if (line.substr(0, 9) === "##begin##") {
5     data = {};
6     ...
```

Wird das Symbol für den Anfang der Testdaten gefunden, wird ein leeres Objekt zum Speichern der Daten eines Testdurchlaufs angelegt.

```
7   } else if (line.substr(0, 12) === "##viewdata##") {
8     var parsedData = parseView(line);
9     ...
```

Die Daten der einzelnen Spieler werden ausgelesen und anschließend im Datenobjekt abgelegt.

```
10  } else if (line.substr(0, 7) === "##end##") {
11    ...
12    testRuns.push(data);
13  }
14 }
```

Zuletzt werden die gefundenen Daten in die Liste der Testdurchläufe eingefügt.

Die vom `FrameRecorder` ausgegebenen Daten werden in der `parseView`-Methode zeilenweise ausgelesen und, falls nötig, wieder in JS-Objekte umgewandelt.

```
1 function parseView(rawData) {
2   var lines = rawData.split("\n");
3   return {
4     playerToken : "" + lines[1].trim(),
5     playerName : "" + lines[2].trim(),
6     entities : JSON.parse(lines[3]),
7     data : JSON.parse(lines[4]),
8   }
9 }
```

Zum Schreiben des Reports wird das vorbereitete HTML-*Template* eingelesen.

```
1 var templateLines = readFileToArray(templateFile);
2 var outputLines = [];
3 for ( var i in templateLines) {
4   var match = templateLines[i].match(includeExpression);
5   if (match) {
6     ...
```

Wenn die entsprechende Stelle in der Vorlage gefunden wird, kann zeilenweise das `testRuns`-Array als JS-Code abgelegt werden.

```
7       outputLines.push("var testRuns = [];");
8       for ( var i in testRuns) {
9         outputLines.push("testRuns.push(" + JSON.stringify(testRuns[i])
      + ");");
10      }
11    ...
```

Ansonsten wird einfach der eingelesene Quellcode wieder ausgegeben. Abschließend werden die gesammelten Zeilen in eine neue Datei geschrieben.

```
12   } else {
13     outputLines.push(templateLines[i]);
14   }
15 }
16 writeArrayToFile(targetFile, outputLines);
```

Im resultierenden HTML-Report können nun beliebige Berechnungen durchgeführt werden und die Ergebnisse mit Hilfe von *JQuery*[19] als Text, Tabellen oder mit der *Flot*-Library[20] auch als Graphen dargestellt werden.

3.7.2 Berechnung der Konsistenz

Wie in Gleichung 3.2 aufgezeigt, ist das Spiel dann konsistent, wenn alle Spieler, die bereits alle Befehle empfangen haben, denselben Zustand der Spielwelt haben. Im Report wird daher zum Berechnen der Konsistenz zu einem Zeitpunkt t eine Funktion erstellt, die folgendermaßen aussieht:

```
 1 function isConsistent(clients, actions, data, entities, t) {
 2   var consistent = true;
 3   for ( var i = 0; i < clients.length - 1; ++i) {
 4     for ( var j = i + 1; j < clients.length; ++j) {
```

Es werden alle übergebenen Clients paarweise kombiniert.

```
 5       var canCompare = true;
 6       for ( var w in actions) {
 7         var action = actions[w];
 8         if (t >= action.time && t <= action.deadline) {
 9           canCompare = (canCompare && receivedBy(data[clients[i]],
       action, t) && receivedBy(data[clients[j]], action, t));
10         }
11       }
```

Für jede übergebene Aktion wird anschließend überprüft, ob sie bereits stattgefunden hat und die Lebensdauer noch nicht abgelaufen ist. Sollte das der Fall sein, wird geprüft, ob beide Clients die Aktion bereits empfangen haben.

```
12       if (canCompare) {
13         consistent = (consistent && hasEqualState(data, entities,
       clients[i], clients[j], t));
14       }
15     }
16   }
17   return consistent;
18 }
```

Nur im Fall, dass alle Aktionen von Beiden empfangen wurden, kann der Spielstand verglichen werden. Sollten alle auf diese Weise geprüften Teilnehmer denselben Zustand haben, ist das Spiel konsistent.

[19] http://jquery.com/
[20] http://www.flotcharts.org/

Die Funktion `receivedBy` entspricht der Empfangsfunktion R aus der Gleichung 3.1 und wird dabei durch folgenden Code abgebildet.

```
1 function receivedBy(clientData, action, t) {
2   for ( var frameNumber = action.time; frameNumber <= t; frameNumber++)
      {
3     if(clientData[frameNumber].actions[action.id] === true) {
4       return true;
5     }
6   }
7   return false;
8 }
```

Es wird in einer Schleife geprüft, ob in einem der vorangegangenen *Frames* die Antwort für die übergebene Aktion empfangen wurde.

In der `hasEqualState` Methode wird über alle Entitäten, die im Testszenario jemals existieren, iteriert und deren Zustand in den beiden Clients verglichen.

```
1 function hasEqualState(data, entities, clientA, clientB, t) {
2   var stateA = data[clientA][t];
3   var stateB = data[clientB][t];
4
5   for ( var id in entities) {
6     var entityA = stateA[id];
7     var entityB = stateB[id];
```

Sollte eine Entität in einem Client existieren, jedoch nicht im anderen, sind die beiden Zustände verschieden, es sei denn sie ist in keinem der beiden Clients zu diesem Zeitpunkt vorhanden.

```
8     if (!entityA || !entityB) {
9       if (entityA !== entityB) {
10        return false;
11      } else {
12        continue;
13      }
14    }
```

Trifft keiner der beiden Fälle zu, wird die Position und die Geschwindigkeit in den beiden Zuständen verglichen.

```
15    if (distance(entityA.position, entityB.position) >
      POSITION_TOLERANCE) {
16      return false;
17    }
18    if (distance(entityA.velocity, entityB.velocity) >
      VELOCITY_TOLERANCE) {
19      return false;
20    }
21  }
22  return true;
23 }
```

Die Funktion `distance` berechnet dazu die euklidische Distanz zwischen den Vektoren. Da die beiden Clients nur selten exakt dieselben Werte enthalten, wird beim Vergleich ein Toleranzwert eingesetzt. Dieser Wert hängt vom jeweiligen Spieltyp und den eingesetzten *Lag*-Kompensationsalgorithmen ab, sollte jedoch zumindest die Ungenauigkeiten von Gleitkommazahlen ausgleichen.

3.7.3 Berechnung der Korrektheit

Wie bereits erwähnt, wird die Korrektheit ähnlich wie die Konsistenz berechnet. Nur dass hier die einzelnen Zustände mit dem Idealzustand der Spielwelt verglichen werden. Dieser könnte zum Beispiel nach Abschluss des Tests im Nachhinein errechnet werden, indem die aufgezeichneten Befehle zum Zeitpunkt ihrer Eingabe direkt am Server ausgeführt werden. Alternativ könnte auch während des Tests ein zweiter Server laufen, an den alle Aktionen ebenfalls ohne Latenz gesendet werden. Im Rahmen dieser Arbeit wird dies jedoch vernachlässigt und davon ausgegangen, dass der Server auch so bereits den idealen Zustand repräsentiert.

```
1 function isCorrect(clients, observer, actions, data, entities, t) {
2   var correct = true;
3   for ( var i = 0; i < clients.length; ++i) {
4     var canCompare = true;
5     for ( var w in actions) {
6       var action = actions[w];
7       if (t >= action.time && t <= action.deadline) {
8         canCompare = (canCompare && receivedBy(data[clients[i]], action,
        t));
9       }
10    }
11    if (canCompare) {
12      correct = (correct && hasEqualState(data, entities, clients[i],
      observer, t));
13    }
14  }
15  return correct;
16 }
```

Die `isCorrect`-Funktion sieht der `isConsistent`-Funktion sehr ähnlich, unterscheidet sich jedoch darin, dass zusätzlich der Beobachter, der hier den Idealzustand repräsentiert, übergeben wird und für diesen nicht extra geprüft wird, ob er alle Aktionen erhalten hat, da davon ausgegangen wird, dass er alle Befehle ohne Verzögerung erhalten hat.

3.7.4 Berechnung der *View Inconsistency*

Die VI wird für jede ausgeführte Aktion berechnet, weshalb sie nur für Befehle, die eine Antwort erzeugen, ermittelt werden kann. In *Swords 'n' Bombs* sind zwar keine echten Antworten für Befehle vorgesehen, es können jedoch

im Nachhinein verschiedene Pakete verknüpft werden. So wird, z. B. nachdem der Server eine `ATTACK_COMMAND`-Nachricht erhält, ein `playerAttack`-Paket an alle Spieler gesendet. Da diese Implementierungsdetails im Report jedoch keine Rolle spielen sollen, werden diese Verknüpfungen bereits beim Abschließen der Tests durchgeführt.

Die absolute VI Φ kann also mit den vorliegenden Daten folgendermaßen berechnet werden:

```
1 function viewInconsistency(data, observer, action, e1, e2) {
2   var t0 = action.time;
3   var t1 = action.finished;
4
5   var clientView = data[action.player][t0];
6   var observerView = data[observer][(t1-t0) / 2];
7   var d0 = positionDiff(clientView[e1].position, clientView[e2].position
      );
8   var d1 = positionDiff(observerView[e1].position, observerView[e2].
      position);
```

Da der Zeitpunkt, zu dem der Server die Aktion empfängt und bearbeitet, momentan nicht ermittelt wird, kann angenommen werden, dass dies näherungsweise nach der Hälfte der Zeit, die eine Aktion benötigt hat, um zum Client zurückzukommen, geschieht. Dies sollte sich, dank der eingestellten symmetrischen Verzögerungen, relativ gut mit dem echten Zeitpunkt decken.

```
9   var D = normalizePrecision(d1 - d0, clientView, e1, e2);
10  var T = normalizeResponsiveness(t1 - t0);
11  return {
12    P : D,
13    R : T,
14    phi : Math.sqrt(D * D + T * T)
15  };
16 }
```

In den Methoden `normalizePrecision` und `normalizeResponsiveness` werden die beiden Werte jeweils in Millisekunden umgewandelt. Für die *Präzision* hängt die Umwandlung von der Bewegungsrichtung und Geschwindigkeit der beobachteten Entitäten ab, während die *Reaktion* mit der *Framerate* umgewandelt werden kann. In *Swords 'n' Bombs* ist diese fix auf 60 *Frames* pro Sekunde eingestellt. Zusätzlich zum berechneten VI-Wert werden auch die beiden Komponenten zurückgegeben, damit auf Basis des Ergebnisses der Funktion die subjektive VI berechnet werden kann.

```
1 function relativeViewInconsistency(vi, lambda, size) {
2   var D = vi.P / size;
3   var T = vi.R / lambda;
4   return {
5     P : D,
6     R : T,
7     phi : Math.sqrt(D * D + T * T)
8   };
9 }
```

Die Frist für die Ausführung der Aktion `lambda` und die Größe der Spielfigur `size` hängen von der jeweiligen Aktion ab und müssen daher ebenfalls bereits beim Abschluss des Tests, wenn die Aktionen verknüpft werden, festgelegt werden.

3.7.5 Berechnung der Unfairness

Damit die Unfairness berechnet werden kann, werden mehrere Testdurchläufe mit verschiedenen Latenzen durchgeführt, wodurch verschiedene VI-Metriken ermittelt werden. Bei drei verschiedenen Latenz und zwei zu beobachtenden Spielfiguren, bedeutet das, dass 3^2 Tests durchgeführt werden und somit neun VI-Werte vorliegen.

Die Ergebniswerte der subjektiven und objektiven VI-Berechnung von jedem Testdurchlauf werden in je einem Array gesammelt und anschließend die Standardabweichung, wie in Gleichung 3.8, berechnet.

```
 1 function calculateStdev(data) {
 2   var r = 0;
 3   var m = 0;
 4   for ( var i in data) {
 5     m += data[i];
 6   }
 7   m /= data.length;
 8   for ( var i in data) {
 9     var temp = data[i] - m;
10     r += temp * temp;
11   }
12   return Math.sqrt(r / data.length);
13 }
```

3.7.6 Das Resultat

Um die berechneten Konsistenz- und Korrektheitswerte übersichtlich zu visualisieren, wird in einem Graph die Anzahl der Tests, bei denen der Zustand korrekt bzw. konsistent ist, über die Zeit angezeigt. Die Zeiträume, in denen die Werte auf Grund von ausstehenden Aktionen nicht ermittelt werden können, sind farblich hinterlegt, um einen Überblick zu haben, wann die Werte nur eine eingeschränkte Aussagekraft haben. Zusätzlich wird jeweils ein prozentualer Wert für die *Konsistenz* und *Korrektheit* angezeigt, der widerspiegelt, wie gut der ideale Zustand erreicht wird, in dem alle Spieler eine konsistente und korrekte Kopie der Spielwelt besitzen.

Für die Visualisierung der VI wird für jede beobachtete Aktion jeweils ein Graph mit dem *Fairnessraum* erstellt, der sowohl die objektive als auch die subjektive VI anzeigt. Der Wert für die Unfairness wird wiederum als Text ausgegeben, wodurch Verbesserungen zwischen Reports einfach abgelesen werden können. Abschließend wird noch eine Auswahlbox für die einzelnen

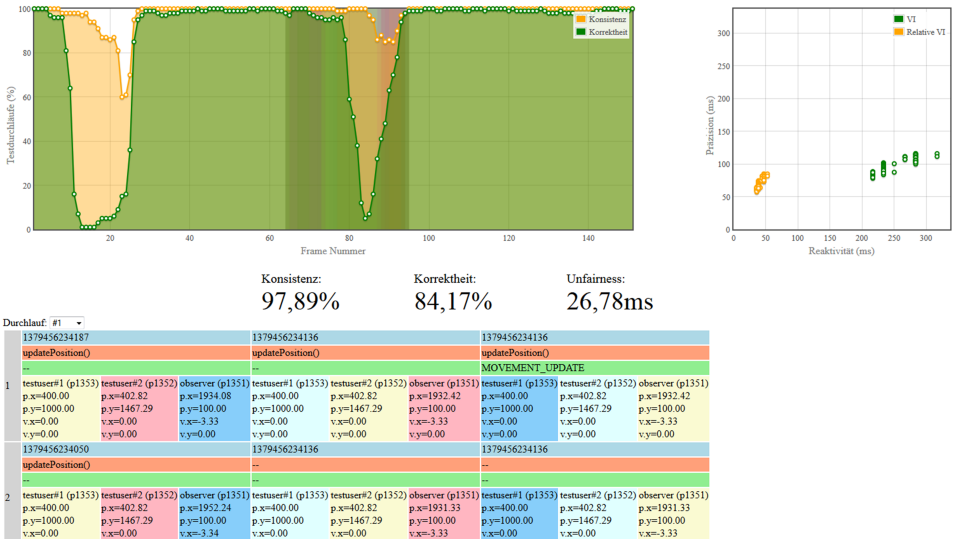


Abbildung 3.7: Exemplarische Screenshot eines HTML-Reports.

Testdurchläufe zur Verfügung gestellt, über die jeweils eine Tabelle mit allen gesammelten Daten eines Testdurchlaufs zugänglich ist.

Ein Beispiel, wie ein solcher Report aussehen kann, ist in Abbildung 3.7 zu sehen. Die *Konsistenz* und *Korrektheit* ist im linken Graphen zu sehen, während der *Fairnessraum* rechts angezeigt wird. Die Tabelle mit den Rohdaten ist mit Farben kodiert, damit zusammengehörige Werte leichter erkannt und zugeordnet werden können.

Kapitel 4

Implementierung und Ergebnisse

Mit Hilfe von *Swords 'n' Bombs* konnten mehrere Reports erstellt werden, welche eindeutig zeigen, dass mit der beschrieben Methode die durch den Einsatz von *Lag*-Kompensation erreichte Steigerung der Qualität eines Spiels erfasst und dargestellt werden kann.

4.1 Implementierte Szenarien

Zum Testen der Funktionalität wurden Testfälle für zwei unterschiedliche Spielsituationen implementiert. Einerseits wurde das in [7] beschriebe Szenario mit einem Schützen, der auf einen vorbeilaufenden Gegner schießt, nachgestellt, andererseits wurde ein neuer Testfall erdacht, in welchem zwei Spieler aufeinander zulaufen und einer der Beiden ausweicht, sobald der andere Spieler ihm zu nahe kommt.

4.1.1 Angriff

Der Angreifer steht während des gesamten Tests auf einer Position und blickt nach Oben. Der verteidigende Spieler läuft in 215 Pixel Entfernung vor dem Angreifer von Links nach Rechts. Sobald der Angreifer in seiner Ansicht den Gegner 60 Pixel vor seiner Schusslinie sieht, feuert er seine Waffe ab, in der Hoffnung, ihn zu treffen.

Beobachtete Größen

Wie bereits in Kapitel 3 erklärt wurde, muss zum Berechnen der VI eine Aktion und ihre Antwort betrachtet werden. Daher wird auf die ausgehende `ATTACK_COMMAND`-Nachricht des Angreifers geachtet. Als Antwort vom Server wird die nachfolgende `playerAttack`-Nachricht interpretiert, die an alle Spieler gesendet wird.

Für die Berechnung der Präzision werden die Positionen von Angreifer und Verteidiger beobachtet. Die Größe einer Spielfigur liegt bei 41 Pixel, womit die subjektive Präzision ebenfalls ermittelt werden kann. Bei der Berechnung der *Deadline* kann davon ausgegangen werden, dass der Angriff vor dem Eintreffen des Projektils beim Verteidiger bereits dargestellt werden soll. Die *Deadline* kann somit als

$$\lambda = \frac{\Delta p_y - l}{v_b} \tag{4.1}$$

berechnet werden, wobei Δp_y der vertikale Abstand der beiden Spieler, l die Größe der Spielfigur und v_b die Geschwindigkeit des Projektils ist. Da sich das Projektil mit 410 Pixel pro Sekunde fortbewegt, ergibt sich daraus eine Obergrenze von etwa 25 *Frames* für die *Deadline*. Um dem Spieler auch noch Zeit zum Ausweichen zu geben, sollte auch noch die Reaktionszeit und die Geschwindigkeit der Spielfigur berücksichtigt werden. Der Auflistung in [32] zufolge liegt die Reaktionszeit von Menschen bei etwa 100 ms, was ca. 6 *Frames* entspricht. Die Spielfigur bewegt sich mit einem Drittel der Geschwindigkeit des Projektils, wodurch bei der eingestellten Größe der Figur etwa 9 *Frames* für ein Ausweichmanöver benötigt werden. Daher wird die Lebensdauer der Aktion schlussendlich mit 10 *Frames* festgelegt.

Implementierung

Bevor der Test beginnt, wird eine halbe Sekunde gewartet, da in aufeinanderfolgenden Tests die Spieler erst in der `tearDown`-Methode ausgeloggt werden und durch die verschiedenen Latenzen eventuell noch nicht entfernt wurden, bevor die neuen Clients eingeloggt werden.

```
1 queue.call("wait for the previous clients to log out", function(
      callbacks) {
2   setTimeout(callbacks.noop(), 500);
3 });
```

Zu Beginn des Testablaufes müssen alle Spieler auf ihre entsprechenden Startpositionen teleportiert werden, da sie eine zufällige Position zugewiesen bekommen.

```
4 queue.call("initialize clients", function(callbacks) {
5   observer.teleportTo(1000, 4000).realize(callbacks, this.clients);
6   testuser1.teleportTo(2275, 2150, 0, -1).realize(callbacks, this.
      clients);
7   testuser2.teleportTo(2030, 1935).realize(callbacks, this.clients);
8 });
```

Der Beobachter wird an eine Stelle in der Spielwelt transportiert, wo er ungestört geradeaus laufen kann, während die Spieler festgelegte Abstände zueinander einnehmen. Für den Angreifer wird zusätzlich die x- und y-Ausrichtung an die `teleportTo`-Methode übergeben, damit dieser, wie erfordert, nach Oben blickt. Um, wie in Kapitel 3 erwähnt, alle *Frames* während

des Tests aufzeichnen zu können, wird der Beobachter eine halbe Sekunde vor dem Test in Bewegung versetzt.

```
 9 queue.call("wait before starting the actual test", function(callbacks) {
10   setTimeout(callbacks.noop(), 500);
11 });
12 queue.call("move the observer to generate position updates", function(
       callbacks) {
13   setTimeout(callbacks.noop(), 500);
14   observer.moveToDirection(1, 0).flush();
15 });
```

Sobald die Wartezeit vor dem eigentlichen Test vorbei ist, wird der Beobachter mit dem Start der Aufzeichnung beauftragt.

```
16 queue.call("send move command", function(callbacks) {
17   observer.startRecording([ testuser1, testuser2 ]);
```

Dieser registriert dazu seine eigene `record`-Methode und die der übergebenen Clients in seiner eigenen `BackendConnection`-Klasse, sodass diese jeweils beim Erhalt eines `POSITION_UPDATES` aufgerufen werden. Danach wird eine Rückrufmethode für den Angriff angelegt, welcher den `ATTACK_COMMAND` an den Server sendet und die Warteschlange in den nächsten Zustand versetzt.

```
18   var shot = false;
19   var finished = callbacks.add(function() {
20     testuser1.attack().flush();
21     shot = true;
22   });
```

Damit dieser nur einmal aufgerufen wird, wird zusätzlich ein Bit-Flag angelegt, welches gesetzt wird, sobald die Funktion zum ersten Mal ausgeführt wird. Um festzustellen, ob die Attacke ausgeführt werden soll, wird bei jedem Positionsupdate vom Server überprüft, ob sich der verteidigende Spieler bereits an der gewünschten Position befindet.

```
23   testuser1.onPositionUpdate(function(updateTime, positionMap) {
24     if (shot)
25       return;
```

Falls der Schuss bereits abgefeuert wurde, kann die Suche übersprungen werden.

```
26     for ( var i in positionMap) {
27       if (positionMap[i].key === testuser2.getId() && positionMap[i].
       value.position.x > 54) {
28         finished();
29         return;
30       }
31     }
32   });
```

Andernfalls wird in den vom Server empfangenen Daten nach einem Eintrag mit der Nummer des gehenden Spielers gesucht und geprüft, ob dieser eine entsprechende horizontale Position aufweist. Da die Positionsdaten noch in

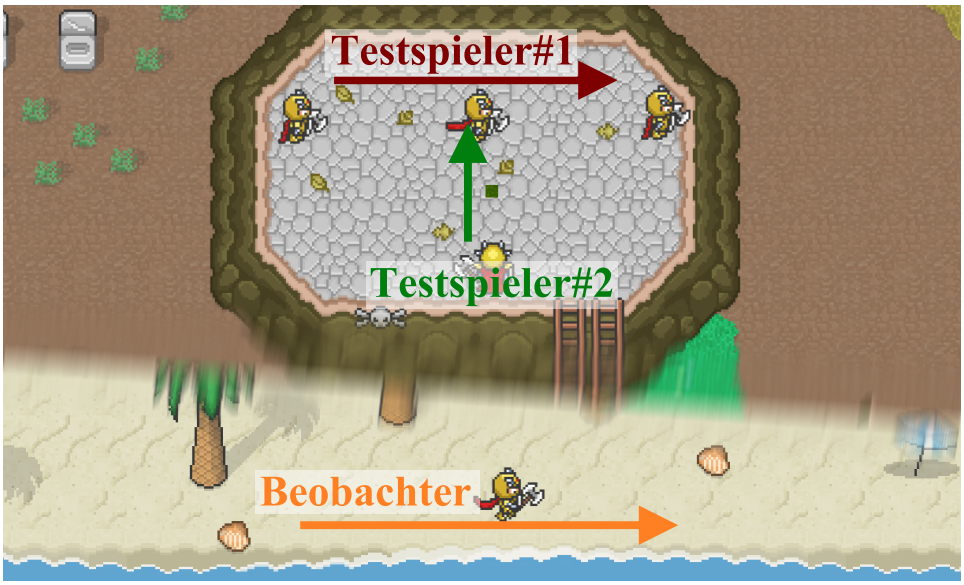


Abbildung 4.1: Die Testspieler bekriegen sich, während der Beobachter einen Strandspaziergang macht.

der Größenskalierung des Servers vorliegen, ergibt sich die Zahl 54 dadurch, dass die gesuchte Position bei $2275 - 60 = 2215$ Pixel liegt. Durch den Umrechnungsfaktor von 41 Pixel pro Meter erhält man somit etwa 54 m. Zuletzt wird der verteidigende Spieler noch damit beauftragt, nach rechts zu laufen.

```
33   testuser2.moveToDirection(1, 0).flush();
34 });
```

Die daraus resultierende Testsituation ist in Abbildung 4.1 zu sehen.

4.1.2 Ausweichen

Bei diesem Testszenario werden zwei Spieler auf einer vertikalen Linie mit 500 Pixel Abstand zueinander positioniert. Anschließend laufen sie direkt aufeinander zu, bis der obere Spieler diagonal, nach rechts unten, ausweicht. Der ausweichende Spieler möchte früh genug mit seiner Aktion beginnen, damit er, wie in Abbildung 4.2 dargestellt, nicht mit dem anderen Spieler zusammenstößt. Da er sich nach Beginn des Ausweichens in einem 45 Grad Winkel bewegt, kann die Zeitdauer t für den Beginn des Vorgangs mit

$$t = \frac{2 \cdot l}{\cos\left(\frac{\pi}{8}\right) \cdot v} \tag{4.2}$$

berechnet werden. Beim gegebenen Spielerdurchmesser l und der Geschwindigkeit v ergibt sich daraus $t = 0{,}859$ s. Weshalb das Ausweichen beginnen

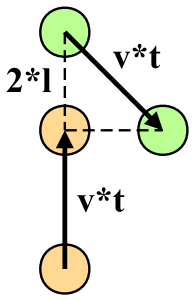


Abbildung 4.2: Die Ausweichbewegung im Überblick.

muss, sobald der Abstand a zwischen den Spielern kleiner als

$$a = 2 \cdot l + v \cdot t \approx 200 \text{ px} \tag{4.3}$$

wird. Dadurch ist sichergestellt, dass immer genug Abstand bleibt, sodass sich die Spieler nicht verfangen und in eine andere Richtung bewegen.

Beobachtete Größen

In diesem Test wird die `MOVEMENT_UPDATE`-Nachricht, die der obere Spieler zum Ausweichen sendet, beobachtet. Die Antwort ist in diesem Fall etwas versteckt, da ständig `positionUpdate`-Nachrichten empfangen werden. Daher wird das erste Positionsupdate, in dem sich die Geschwindigkeit des oberen Spielers verändert, als Antwort gewertet.

Zum Berechnen der Präzision werden, wie im ersten Test, die beiden Spieler herangezogen. Für die Deadline gibt es diesmal keine echten Anhaltspunkte, daher wird sie mit der Reaktionszeit gleichgesetzt und somit auf 6 *Frames* festgelegt.

Implementierung

Der Testcode bleibt zu großen Teilen identisch, es werden jedoch andere Startpositionen für die Spieler gewählt.

```
1 queue.call("initialize clients", function(callbacks) {
2   observer.teleportTo(2000, 100).realize(callbacks, this.clients);
3   testuser1.teleportTo(400, 1000).realize(callbacks, this.clients);
4   testuser2.teleportTo(400, 1500).realize(callbacks, this.clients);
5 });
```

Anstelle der Attacke sendet der Spieler dieses mal eine Bewegung.

```
1   var done = false;
2   var finished = callbacks.add(function() {
```

```
3     testuser1.moveToDirection(1, 1).flush();
4     done = true;
5   });
```

In den Positionsupdates wird anstelle der x-Position die y-Koordinate überprüft. Der Wert dafür ergibt sich aus dem Abstand der Spieler $d = y_1 - y_0$ und dem Mindestabstand a für das Ausweichen,

$$y_{min} = \frac{y_1 - \left(\frac{d-a}{2}\right)}{41} \approx 33 \text{ m}. \tag{4.4}$$

```
6   testuser1.onPositionUpdate(function(updateTime, positionMap) {
7     if (done)
8       return;
9
10    for ( var i in positionMap) {
11      if (positionMap[i].key === testuser2.getId() && positionMap[i].
      value.position.y < 33) {
12        finished();
13        return;
14      }
15    }
16  });
```

Außerdem werden dieses mal beide Spieler in Bewegung versetzt.

```
17  testuser1.moveToDirection(0, 1).flush();
18  testuser2.moveToDirection(0, -1).flush();
19 });
```

In Abbildung 4.3 ist wiederum die im Spiel entstehende Situation abgebildet.

4.2 Verwendete *Lag*-Kompensationsalgorithmen

Um feststellen zu können, ob der Report wie erwartet eine Steigerung der Qualität anzeigt, wurde zu diesem Zweck ein simpler *Dead Reckoning* (DR)-Algorithmus und *Local Lag* (LL) eingesetzt, wie bereits in Kapitel 2 beschrieben.

4.2.1 *Dead Reckoning* Implementation

Im Gegensatz zu der in Gleichung 2.3 beschriebenen Variante werden nur die Ausgangsposition und Geschwindigkeit berücksichtigt, da die Spielfiguren nur gehen oder stehen können und somit keine Beschleunigung besitzen. Die Methode `extrapolate` sitzt im `PlayerManager` und wird zu jedem *Frame*, das heißt in diesem Fall durch den `FrameRecorder`, aufgerufen.

```
1 that.extrapolate = function() {
2   for ( var i in players) {
3     var player = players[i];
4     if (player.lastUpdateTime && player.velocity) {
```

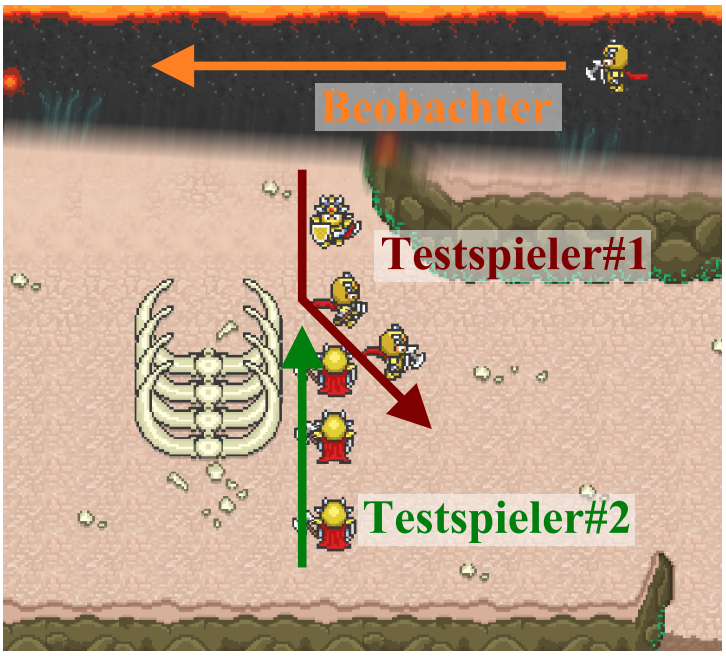


Abbildung 4.3: Der Beobachter spaziert auf einem Vulkan, während die Testspieler das Feiglingsspiel spielen.

```
5       var velocity = player.velocity;
6         if (Math.abs(velocity.x) > 0.01 || Math.abs(velocity.y) > 0.01)
      {
```

Die Extrapolation kann nur für Spieler durchgeführt werden, die bereits ein Positionsupdate erhalten haben. Daher wird überprüft, ob die notwendigen Werte bereits vorhanden sind. Weiterhin ändert sich für Entitäten, die stillstehen, nichts und daher wird im Vorhinein geprüft, ob die Geschwindigkeit größer als 0 ist, um unnötige Berechnungen zu vermeiden.

```
7       var time = player.lastUpdateTime;
8       time = new dcodeIO.Long(time.low, time.high, time.unsigned);
9       var now = dcodeIO.Long.fromNumber(Date.now());
10      var delta = now.subtract(time).toNumber() / 1000;
```

Da die Zeit als 64-bit Wert übertragen wird und JS keine 64-bit Integer unterstützt, kommt eine Bibliothek für *Long*-Werte[1] zum Einsatz. Eigentlich ist `lastUpdateTime` beim Empfang bereits ein *Long*-Objekt, da die Deserialisierung jedoch in einem *Web Worker* abläuft und dieser nur sprach-eigene JS-Klassen an den Haupt-*Thread* übergeben kann, muss ein neues *Long*-Objekt mit den vorliegenden Werten initialisiert werden, damit die Klassen-Methoden verwendet werden können. Die Zeit muss anschließend noch von

[1] https://github.com/dcodeIO/Long.js

Millisekunden in Sekunden umgewandelt werden, da die Geschwindigkeit ebenfalls auf Sekundenbasis vorliegt.

```
11      var origin = player.originalPosition;
12      var newX = origin.x + delta * velocity.x;
13      var newY = origin.y + delta * velocity.y;
14      player.setPosition(newX, newY);
15      ...
```

4.2.2 *Local Lag* Implementation

Die Implementation von LL besteht aus drei Teilen:

- Berechnung der notwendigen Verzögerung,
- zurückhalten der empfangenen Nachrichten und
- verzögern der zu sendenden Pakete.

Da die `SCORE_BOARD_UPDATE` Nachricht alle RTTs aus der Sicht des Servers enthält, ist die Berechnung der notwendigen Verzögerung relativ einfach. Für die Tests wird die maximale RTT jedoch fix eingestellt, da nicht alle Spieler gleichzeitig im Spiel sind und daher in den unterschiedlichen Testdurchläufen unterschiedliche Verzögerungen berechnet werden würden. Die notwendige Verzögerung T_d, kann mit

$$T_d = \frac{RTT_{max} - RTT_p}{2} \tag{4.5}$$

berechnet werden, wobei RTT_{max} die höchste RTT bezeichnet und RTT_p die des jeweiligen Spielers.

Die empfangenen Pakete werden nach der Deserialisierung im *Web Worker* zurückgehalten, indem die Übergabe mittels `setTimeout` verzögert wird. Dazu wird nach der Berechnung der Verzögerung der Wert mittels Aufruf der `postMessage`-Funktion an den *Web Worker* übergeben und dort zwischengespeichert.

Da in JSTD `setTimeout` nicht wie erwartet funktioniert, innerhalb des *Web Workers* jedoch schon, werden die zu sendenden Pakete beim Aufruf der *send*-Methode in einer Liste zwischengespeichert und der jeweilige Index an den *Web Worker* übergeben. Dieser kennt die notwendige Verzögerungszeit bereits und sendet den Index mit `setTimeout` zurück an den Haupt-*Thread*. Dort wird anschließend die ursprüngliche Funktion zum Senden mit den in der Liste gespeicherten Werten aufgerufen.

4.3 Auswertung der Testdurchläufe

Für beide Szenarien wurden die Tests jeweils entweder ohne Kompensation, nur mit DR, nur mit LL oder mit einer Kombination von beiden Techniken durchgeführt. Innerhalb der Tests werden für die beiden Spieler je eine Kombinationen aus *Lags* von 100 ms, 200 ms und 300 ms verwendet.

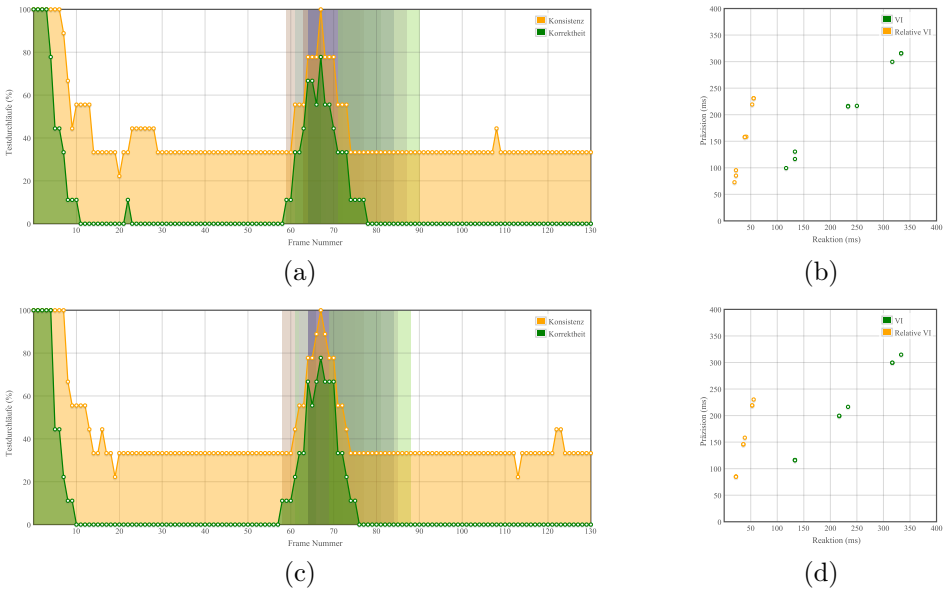


Abbildung 4.4: Die Diagramme zweier aufeinanderfolgender Tests. (a) und (c) zeigen jeweils die Korrektheit und Konsistenz, (b) und (d) dagegen den Fairness Space.

4.3.1 Genauigkeit

Wie in Abbildung 4.4 anhand von zwei aufeinander folgenden Reports zu sehen ist, liefern Tests, die mit unveränderten Einstellungen erstellt wurden, zur Zeit keine identischen Ergebnisse. Dies kann daran liegen, dass `tc` die Verzögerungen, wie in [15] beschrieben, in Schüben erzeugt und somit leichte Abweichungen entstehen können. Eine weitere Quelle für Unterschiede kann die Selbstoptimierung der JRE sein, welche mit längerer Laufzeit wesentlich effizienter arbeitet als direkt nach dem Start. Weitere Differenzen können durch die Spiel-Physik selbst entstehen, welche nicht deterministisch abläuft.

Aus diesem Grund wird jede Latenz mehrere Male hintereinander getestet, um statistisch aussagekräftige Werte für die Reports zu erhalten. Abbildung 4.5 zeigt zwei Reports, bei denen die einzelnen Testdurchläufe je 10 Mal wiederholt wurden. Die Ergebnisse sind zwar nach wie vor nicht identisch, jedoch können zumindest größere Abweichungen in den Werten erkannt werden. Die Gesamtdauer für die Erstellung eines Reports erhöht sich damit zwar auf beinahe 10 Minuten, was jedoch angesichts der besseren Qualität der Reports ein kleiner Preis ist. Aus diesem Grund wird für den restlichen Verlauf der Tests diese Anzahl beibehalten.

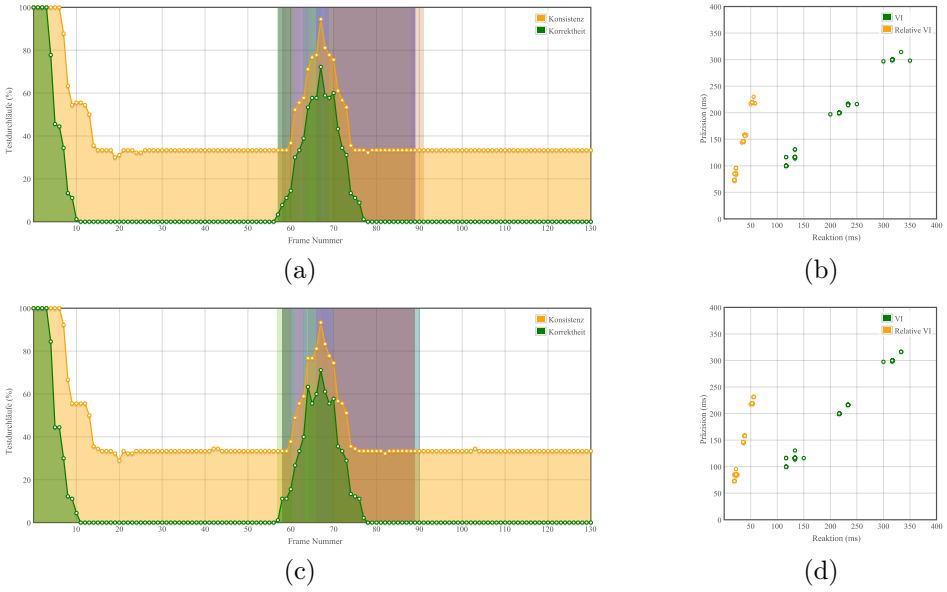


Abbildung 4.5: Die Diagramme zweier aufeinanderfolgender Tests mit je 10 Wiederholungen jeder Kombination von Latenzen. (a) und (c) zeigen jeweils die Korrektheit und Konsistenz, (b) und (d) dagegen den Fairness Space.

4.3.2 Angriffsszenario

Wie in Abschnitt 4.1.1 beschrieben wurden mehrere Tests durchgeführt und Reports erstellt, deren Gesamtergebnis in Tabelle 4.1 zu sehen ist. Wie eindeutig zu erkennen ist, steigt die Korrektheit und Konsistenz des Spiels bei Einsatz von *Lag*-Kompensation, während die Unfairness sinkt. Der verwendete DR Algorithmus steigert Korrektheit und Konsistenz beträchtlich und kann auch die Unfairness ein wenig reduzieren. Die gewählte LL Implementierung halbiert die Unfairness zwischen den Spielern und steigert auch die Konsistenz, während die Korrektheit sich nicht wesentlich ändert. Werden beide Algorithmen kombiniert, können zwar nicht die besten Werte bei Korrektheit und Unfairness erzielt werden, jedoch ist das Ergebnis ausgeglichener, als wenn nur eine der beiden Methoden zum Einsatz kommt.

Korrektheit und Konsistenz

Wie in Abbildung 4.6 zu sehen ist, kann ohne Einsatz von Kompensationsalgorithmen die Korrektheit der Spielsituation nicht gewährleistet werden. Die Konsistenz ist wie zu erwarten für ein Drittel der Durchläufe gegeben, da bei selber Latenz zwei Spieler einen ähnlichen Zustand haben. Die hohen Werte zu Beginn sind dadurch zu erklären, dass alle Teilnehmer mit demselben Zustand starten. Erst nachdem der Befehl für die Bewegung des Spielers

Tabelle 4.1: Ergebniswerte des HTML-Reports im Überblick.

	Korrektheit	Konsistenz	Unfairness
Unkompensiert	12,47%	43,07%	102,80 ms
Dead Reckoning	61,08%	75,40%	80,01 ms
Local Lag	19,35%	72,75%	51,24 ms
Kombination	50,90%	89,51%	60,07 ms

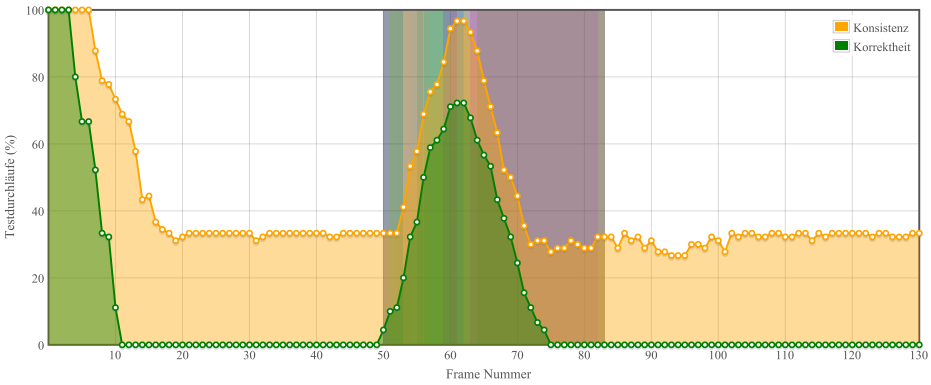


Abbildung 4.6: Korrektheit und Konsistenz beim Angreifen, wenn keine Kompensation eingesetzt wird.

eingeht, ändern sich abhängig von der Latenz die Zustände der Spieler. Die zweite Spitze, in der Mitte des Tests, entspricht dem Zeitraum, in dem der Angriffsbefehl gesendet wird. Da der Zustand der Spieler in dieser Periode nicht vergleichbar ist, wird angenommen, dass Korrektheit und Konsistenz gegeben sind.

Durch den Einsatz von DR wird die Konsistenz und die Korrektheit gesteigert. Wie in Abbildung 4.7 abgelesen werden kann, ist dies jedoch nur teilweise der Fall. Der Einbruch in der Konsistenz nach dem Angriff kann durch den Stoß, den der laufende Spieler vom Projektil bekommt, erklärt werden. Dadurch weicht die berechnete Position zwischen den Clients ab, da sie auf Basis von unterschiedlichen Daten extrapolieren. Sobald alle Clients die Daten mit der neuen Ausgangsposition haben, sollte eigentlich die Korrektheit und Konsistenz rasch wieder auf denselben Level wie vor dem Angriff ansteigen. Wieso diese allerdings nur langsam ansteigen, konnte nicht geklärt werden, weist aber auf ein Problem in der verwendeten DR Implementation hin.

In Abbildung 4.8 wird das Ergebnis bei Einsatz von LL gezeigt. Wie erwartet verbessert sich nur die Konsistenz, da dieser Algorithmus nur auf

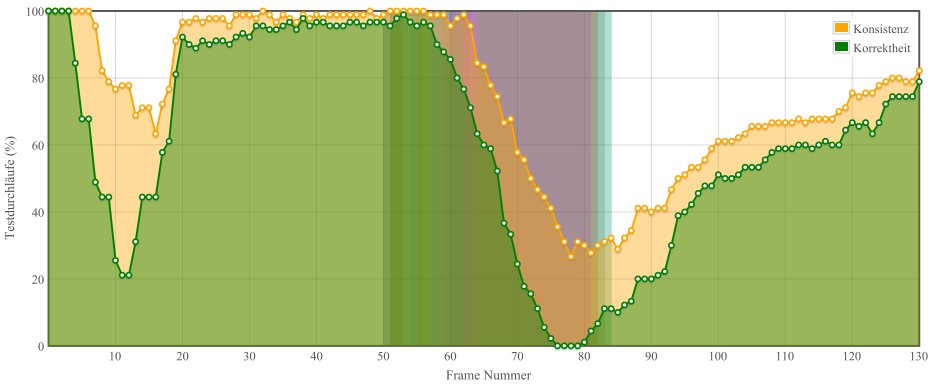


Abbildung 4.7: Korrektheit und Konsistenz beim Angreifen, unter Einsatz von DR.

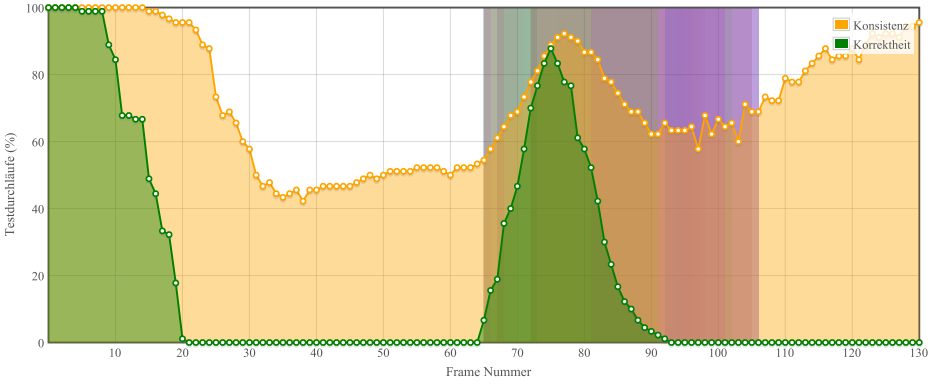


Abbildung 4.8: Korrektheit und Konsistenz beim Angreifen, mit LL.

Benutzerseite zum Einsatz kommt und im Prinzip die Latenz aller Clients angleicht. Auch hier gibt es offensichtlich ein Problem mit dem verwendeten Code, da die Konsistenz langsam über die Zeit ansteigt, anstatt konstant zu sein.

Zuletzt kann in Abbildung 4.9 eine Verbesserung der Konsistenz und Korrektheit durch den kombinierten Einsatz von LL und DR festgestellt werden. Der Einbruch in der Korrektheit nach dem Angriff ist nach wie vor gegeben, die Konsistenz bleibt jedoch wesentlich höher, da alle Benutzer ähnliche Daten für die Extrapolation verwenden. Die Probleme mit den beiden Algorithmen verstärken sich allerdings weiter, wodurch sich die Korrektheit nach dem Angriff noch langsamer erholt.

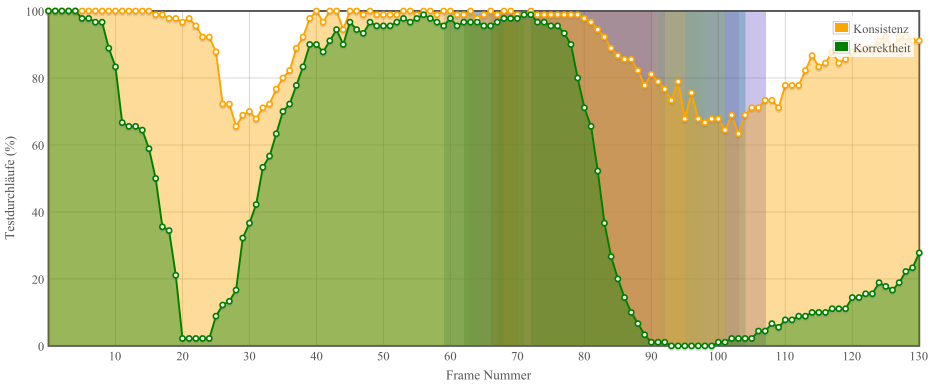


Abbildung 4.9: Korrektheit und Konsistenz beim Angreifen, bei einer Kombination von DR und LL.

View Inconsistency

In Abbildung 4.10 ist die Gegenüberstellung der Fairnessräume zu sehen. Ohne *Lag*-Kompensation sind die Präzision und Reaktion rein von der Latenz abhängig. Durch den Einsatz von DR kann die Präzision erhöht werden, während LL die Reaktion aller Spieler angleicht. Wenn beide Techniken kombiniert werden, kann die Unfairness zwischen den Spielern wesentlich reduziert werden, wie auch aus Tabelle 4.1 abgelesen werden kann. Die starke Streuung bei Einsatz von LL zeigt eindeutig, dass der verwendete Algorithmus nicht wie erwartet alle Latenzen auf denselben Wert bringt und überarbeitet werden muss.

4.3.3 Ausweich-Szenario

Diesmal werden mehrere Tests mit dem in Abschnitt 4.1.2 beschriebenen Szenario durchgeführt. Die Ergebnisse der Reports zeigen in Tabelle 4.2 wiederum eine Verbesserung in den Hauptmetriken bei Einsatz von Kompensationstechniken. Im Gegensatz zum vorhergehenden Szenario zeigt LL keine Verbesserung in der Fairness gegenüber DR, dafür ist der Unterschied in der Konsistenz wesentlich deutlicher zu erkennen. Auch hier stellt die Kombination der beiden Techniken den besten Kompromiss dar.

Korrektheit und Konsistenz

Die Korrektheit und Konsistenz sind, wie in Abbildung 4.11 gezeigt, wiederum niedrig, solange keine Kompensation im Einsatz ist. Die Spitze ist diesmal kleiner als beim Angriffsszenario, da die Deadline für die Aktion kürzer ist.

Wie in Abbildung 4.12 zu sehen ist, versagt DR nach einer Geschwindigkeitsänderung jeweils zu Beginn und in der Mitte des Tests für einen kurzen

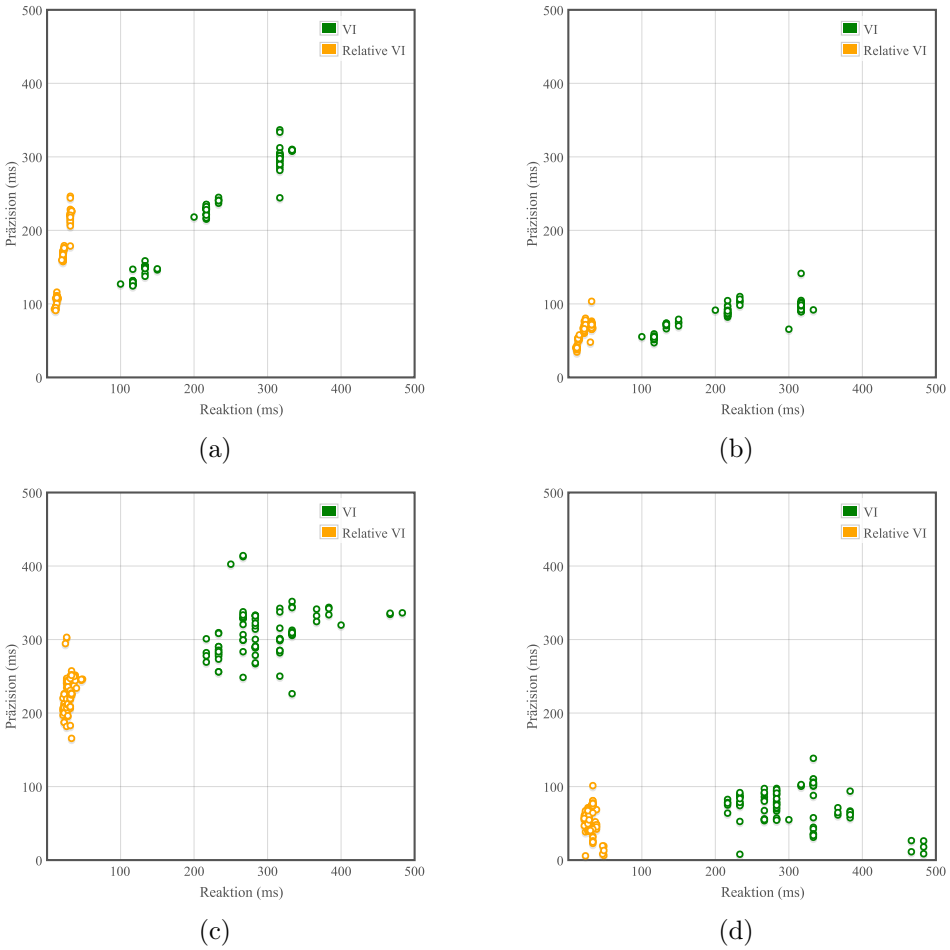


Abbildung 4.10: Der Fairness Space beim Angreifen, im unkompensierten Zustand (a), unter Einsatz von DR (b), LL (c) und mit einer Kombination von beiden Techniken (d).

Tabelle 4.2: Korrektheit und Konsistenz im Überblick.

	Korrektheit	Konsistenz	Unfairness
Unkompensiert	9,41%	41,36%	109,91 ms
Dead Reckoning	86,66%	94,02%	84,46 ms
Local Lag	14,91%	67,81%	81,58 ms
Kombination	77,16%	94,78%	59,32 ms

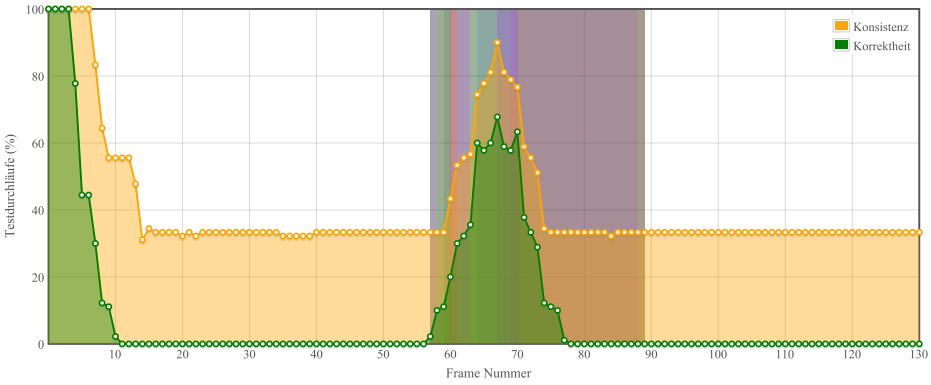


Abbildung 4.11: Korrektheit und Konsistenz beim Ausweichen, wenn keine Kompensation eingesetzt wird.

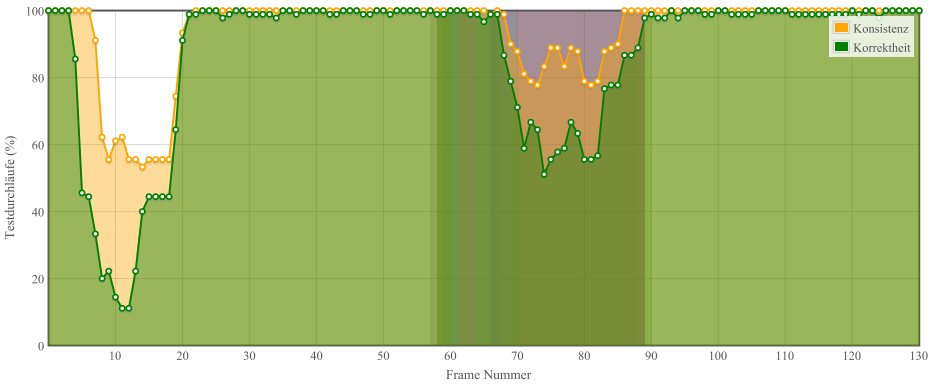


Abbildung 4.12: Korrektheit und Konsistenz beim Ausweichen, unter Einsatz von DR.

Moment, bis die neuen Daten empfangen wurden. Da beim zweiten Einbruch nur die Richtung um 45 Grad verändert wird, ist dieser nicht so stark wie der Erste, bei dem die Spielfiguren zuvor stillstehen und plötzlich losgehen. Bei einer einfachen Bewegung funktioniert der Algorithmus offensichtlich wie erwartet und die Werte steigen nach der Aktion wieder rasch an.

Der in Abbildung 4.13 gezeigte Verlauf bei der Verwendung von LL weist dieselbe kontinuierliche Steigerung der Konsistenz auf, die auch in Abbildung 4.8 zu erkennen ist. Das bedeutet, die Probleme in diesem Algorithmus sind unabhängig vom Szenario immer vorhanden.

Wenn, wie in Abbildung 4.14 dargestellt, beide Techniken gemeinsam eingesetzt werden, ist eindeutig eine Verschlechterung in der Korrektheit gegenüber DR zu erkennen, während die Einbrüche in der Konsistenz lediglich nach hinten verschoben werden.

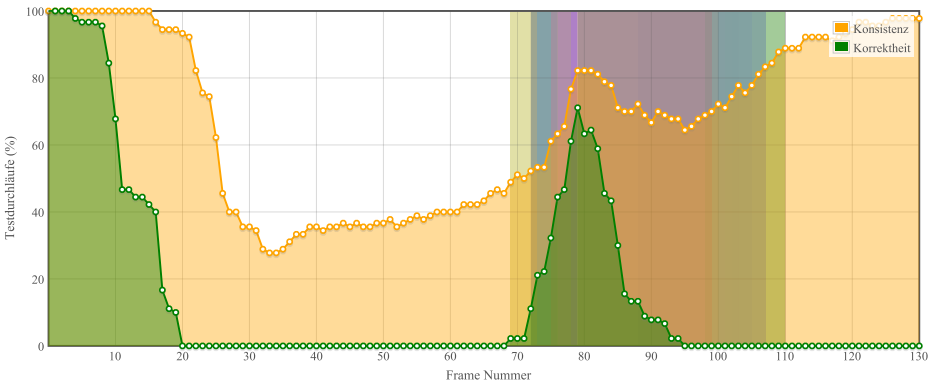


Abbildung 4.13: Korrektheit und Konsistenz beim Ausweichen, mit LL.

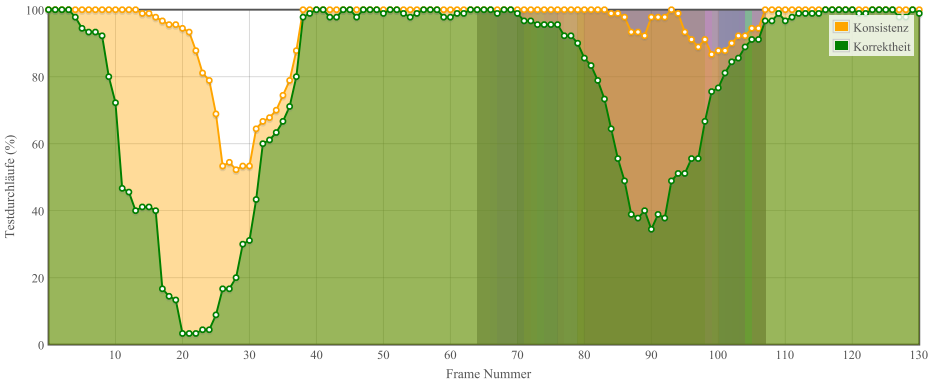


Abbildung 4.14: Korrektheit und Konsistenz beim Ausweichen, bei einer Kombination von DR und LL.

View Inconsistency

Die Fairnessräume in Abbildung 4.15 zeigen ebenfalls wieder, wie sich die Werte durch den Einsatz von *Lag*-Kompensation annähern, jedoch ist die Auswirkung nicht so stark wie im vorhergehenden Szenario. Die Streuung bei Einsatz von LL ist zwar noch stärker als zuvor, jedoch kann bei der Kombination der beiden Techniken durch DR die Verschlechterung der Präzision verhindert werden.

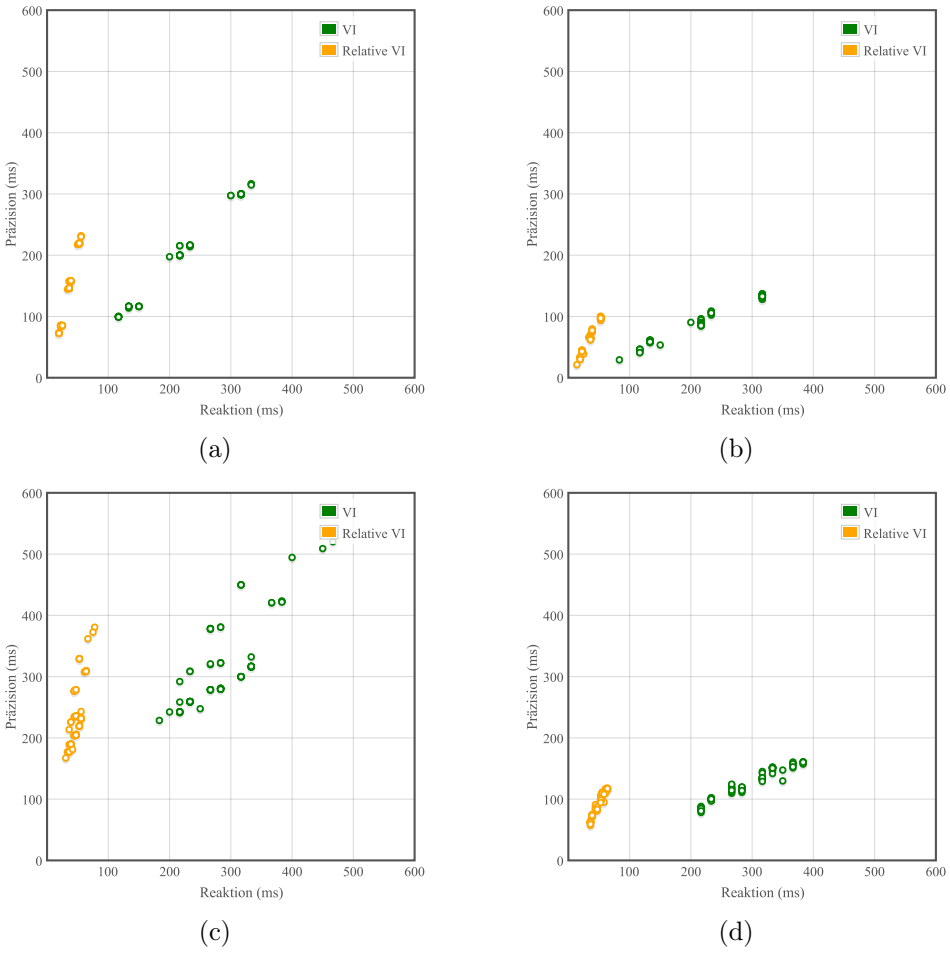


Abbildung 4.15: Der Fairness Space beim Ausweichen, im unkompensierten Zustand (a), unter Einsatz von DR (b), LL (c) und mit einer Kombination von beiden Techniken (d).

Kapitel 5

Schlussbemerkungen

Wie erwartet zeigen die Reports, ob die *Lag*-Kompensation für die getesteten Szenarien funktioniert, und bieten, dank der in [7] und [13] beschriebenen Methoden, Einblicke in Funktionalität der gewählten Algorithmen und in die zu erwartende Qualität des Spielerlebnisses. Des Weiteren konnten, mit Hilfe der gesammelten Daten, während der Entstehung der Arbeit Probleme mit den verwendeten Algorithmen einfach erkannt und behoben werden. Es bestehen zwar weiterhin Probleme, die jedoch aus Zeitgründen nicht mehr behoben werden konnten.

5.1 Verbesserungsmöglichkeiten

Das vorgestellte *Framework* ist zwar bereits funktionsfähig, es gibt jedoch viele Punkte, die noch verbessert werden können.

5.1.1 Genauigkeit

Wie in Kapitel 4 gezeigt wurde, schwanken die Ergebnisse der Testdurchläufe bei unverändertem Programm und Testumgebung relativ stark. Es ist daher sicher angebracht, die Ursache dieser Differenzen zu ermitteln und das Testsetup in dieser Hinsicht weiter zu verbessern. Als mögliche Gründe wurden bereits die von `tc` angewandte Methode zum Erzeugen der *Lags*, die Selbstoptimierung der JRE und die Physik-*Engine* des Spiels genannt.

Wie in [15] erwähnt, kann die Qualität der *Lags* durch den Einsatz eines hochauflösenden Zeitgebers erhöht werden. Um den Auswirkungen der Selbstoptimierung der JRE entgegenzuwirken, könnte vor Beginn der eigentlichen Tests eine Art Kalibrierdurchlauf durchgeführt werden. Die dritte erwähnte Fehlerquelle wird wahrscheinlich ohne Eingriffe in die grundlegende Funktionalität des Spiels kaum ausgemerzt werden können. Daher sollte im Idealfall bereits während der Entwicklung darauf geachtet werden, dass eine Situation auf Basis von gegebenen Ausgangswerten reproduzierbar ist.

5.1.2 Ablaufoptimierung

Da die Ausführung eines einzelnen Tests ca. 5 Sekunden dauert und somit der vollständige Testdurchlauf bei einer Wiederholung derzeit bereits etwas mehr als eine Minute in Anspruch nimmt, ist es empfehlenswert, den Ablauf der Testsituation zu optimieren. Dafür gibt es mehrere Möglichkeiten. So kann z. B. die Wartezeit vor und nach den einzelnen Testdurchläufen von insgesamt 2,5 s auf eine halbe Sekunde reduziert werden. Außerdem können die von den Testspielern zurückgelegten Strecken ohne Probleme noch verkürzt werden, wodurch neben der Dauer auch die Datenmenge reduziert wird. Dadurch wird auch die Zeit, die zum Schreiben des Reports benötigt wird, verringert.

5.1.3 Offene Punkte

Da JS keine Möglichkeit bietet, einen *Datagramsocket* zum Server zu öffnen, wurde in dieser Arbeit nicht beschrieben, wie die Tests für eine Kommunikation über UDP aufgebaut sein müssen. Aus demselben Grund wurden bei der Konfiguration der Netzwerksimulation auch die Limitierung der Datenrate und Paketverluste ignoriert, da diese sich bei TCP nur als zusätzliche Latenz abzeichnen. Wie sich *Jitter* auf das Ergebnis der Tests auswirkt, wurde ebenfalls außer Acht gelassen, um das Setup und die Auswertung im ersten Schritt nicht unnötig zu verkomplizieren.

5.2 Weitere Schritte

Damit die Ergebnisse noch weiter verifiziert werden können, ist es auf jeden Fall notwendig, noch mehr *Lag*-Kompensationsverfahren und Spielsituationen zu implementieren und zu testen. Des Weiteren ist klar, dass die derzeit eingesetzten Berechnungsmethoden noch viel Spielraum für Verbesserungen zulassen. So wird im Moment z. B. komplett ignoriert, wie stark sich der Schaden eines Projektils auf die subjektive Wahrnehmung eines Spielers auswirkt, oder welche Bedeutung ein auf Grund von Lag fehlgeschlagener Angriff hat. Wie gut sich der berechnete Wert für die subjektive VI mit der Wahrnehmung von echten Spielern deckt, verbleibt daher ebenfalls unklar. Es muss auch noch ein Weg gefunden werden, wie die Veränderung der Werte zwischen den einzelnen Tests dargestellt werden kann.

Um die Brauchbarkeit des *Frameworks* zu erhöhen, sollten weitere Spiele, auch aus unterschiedlichen Genres, getestet werden. Dazu ist es notwendig, dass die Berechnung der Ergebnisse und die Erstellung des Reports soweit abstrahiert werden, dass sie unabhängig vom Spiel funktionieren. Ein Ziel sollte dabei sein, dass am zu testenden Spiel nur geringfügige Änderungen vorgenommen werden müssen.

5.2.1 Persönliche Einschätzung

Wie aus der Arbeit hervorgehen sollte, denke ich, dass die erarbeitete Version des *Frameworks* bereits ihren Nutzen beweisen konnte. Besonders für die Fehlersuche haben sich die gesammelten Daten als hilfreich erwiesen, da ich einige Probleme mit den implementierten *Lag*-Kompensationsalgorithmen, die sonst nur sehr schwer ersichtlich gewesen währen, schnell und unproblematisch lösen konnte. Die Verbesserung in den Metriken zwischen den einzelnen Änderungen konnten mich auch immer motivieren, weitere Variationen auszuprobieren, da sie ein Ziel bieten, auf das ich hinarbeiten kann. Angesichts der vielen Vorteile des beschriebenen Verfahrens verwundert es mich ein wenig, dass bisher noch niemand ein derartiges *Framework* entwickelt hat.

Wie beschrieben, gibt es noch viele Verbesserungsmöglichkeiten, die ich gerne einbauen möchte, leider verbleibt dafür jedoch keine Zeit mehr. Nach Abschluss dieser Arbeit werde ich mich jedoch bemühen, das *Framework* für moderne Onlinespiele fit zu machen, um Tests mit diesen durchführen zu können und die Frage zu klären, ob diese Spiele bei unterschiedlicher Verbindungsqualität der Spieler wirklich fair sind.

Anhang A

Inhalt der DVD

A.1 PDF-Dateien

Pfad: /

Meisinger13.pdf . . . . Masterarbeit

Pfad: /webseiten/

Aronson97.pdf . . . . . Kopie von [33]
Cisco07.pdf . . . . . Kopie von [41]
Cloonan11.pdf . . . . . Kopie von [34]
Dawson13.pdf . . . . . Kopie von [36]
GDSE.pdf . . . . . Kopie von [38]
Grigorik12.pdf . . . . . Kopie von [37]
InstallFirefox.pdf . . . . Kopie von [39]
Llopis05.pdf . . . . . Kopie von [40]
Peterson13.pdf . . . . . Kopie von [42]
PRWeb.pdf . . . . . Kopie von [35]
Takahashi11.pdf . . . . Kopie von [43]
TcHowTo.pdf . . . . . Kopie von [44]
TcManPage.pdf . . . . Kopie von [45]

Anhang B

Script zum Messen der Latenz

Diese *Python*-Script erstellt einen Server und einen Client jeweils in verschiedenen *Threads*. Es werden eine beliebige Anzahl von Paketen zwischen ihnen hin und her gesendet, und jeweils die Sendezeiten aufgezeichnet. Abschließend wird ausgewertet und ausgegeben, wie lange die Pakete in beide Richtungen benötigt haben.

```
1  import socket
2  import time
3  import threading
4  import argparse
5  import numpy
6
7  class ClientThread(threading.Thread):
8      def __init__(self,port,retries,server):
9          threading.Thread.__init__(self)
10         ADDR = ('localhost', port)
11         self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
12         self.sock.connect((ADDR))
13         self.retries = retries
14         self.server = server
15         self.results = []
16
17     def run(self):
18         try:
19             for i in range(self.retries):
20                 self.send(self.sock)
21         finally:
22            self.sock.close()
23
24     def send(self, sock):
25        time.sleep(0.5)
26        time1 = time.time()
27        sock.send(' ')
28        sock.recv(1)
29        time3 = time.time()
30        time.sleep(0.5)
31
```

```
32        time2 = self.server.time2
33        up = (time2 - time1) * 1000
34        down = (time3 - time2) * 1000
35        rtt = (time3 - time1) * 1000
36        self.results.append((up, down, rtt))
37
38        print 'up: %.2f ms, down: %.2f ms, rtt: %.2f ms' % (up, down, rtt
      )
39
40 class ServerThread(threading.Thread):
41     def __init__(self,port,retries):
42         threading.Thread.__init__(self)
43         ADDR = ('localhost', port)
44         self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
45         self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
46         self.sock.bind(ADDR)
47         self.sock.listen(1)
48         self.retries = retries
49
50     def run(self):
51         conn, addr = self.sock.accept()
52         try:
53             for i in range(self.retries):
54                 conn.recv(1)
55                 self.time2 = time.time()
56                 conn.send(' ')
57         finally:
58             conn.close()
59
60 parser = argparse.ArgumentParser(description="test up and downstream
      latency on a port.")
61 parser.add_argument("port", type=int, help="the port to connect to")
62 parser.add_argument("-r", default=5, type=int, help="specifies how many
      samples should be taken")
63 parser.add_argument("-s", default=8080, type=int, help="the port of the
      server")
64 args = parser.parse_args()
65
66 retries = args.r
67 thread1 = ServerThread(args.s, retries)
68 thread2 = ClientThread(args.port, retries, thread1)
69 thread1.start()
70 thread2.start()
71 thread1.join()
72 thread2.join()
73
74 nums = numpy.array(thread2.results)
75 min_up, min_down, min_rtt = nums.min(axis=0)
76 max_up, max_down, max_rtt = nums.max(axis=0)
77 mean_up, mean_down, mean_rtt = nums.mean(axis=0)
78 std_up, std_down, std_rtt = nums.std(axis=0)
79
80 print '-------------------------------------------------------------'
81 print 'result for %d samples:' % nums.shape[0]
```

```
82 print 'up:   min=%.2f ms, max=%.2f ms, mean=%.2f ms, stdev=%.2f ms' % (
       min_up, max_up, mean_up, std_up)
83 print 'down: min=%.2f ms, max=%.2f ms, mean=%.2f ms, stdev=%.2f ms' % (
       min_down, max_down, mean_down, std_down)
84 print 'rtt:  min=%.2f ms, max=%.2f ms, mean=%.2f ms, stdev=%.2f ms' % (
       min_rtt, max_rtt, mean_rtt, std_rtt)
```

Beim Ausführen ohne Parameter wird der folgende Hilfetext angezeigt.

```
usage: latencytest.py [-h] [-r R] [-s S] port

test up and downstream latency on a port.

positional arguments:
  port        the port to connect to

optional arguments:
  -h, --help  show this help message and exit
  -r R        specifies how many samples should be taken
  -s S        the port of the server
```

Anhang C

Script zum Einrichten der Verzögerungen

Das nachfolgende *Bash*-Script benötigt drei Parameter: Startverzögerung, Endverzögerung und Schrittweite. Es werden initial alle bestehenden Regeln auf den Adaptern `lo`, `eth0`, `ifb0` und `ifb1` gelöscht und die Wurzelknoten für nachfolgende Regeln und die Umleitung von *ingress* auf *egress* eingerichtet. In einer Schleife werden die notwendigen Regeln für eine bidirektionale Latenz auf den einzelnen Ports auf die Adapter angewandt.

```
#!/bin/bash
startport=8080
count=0

clear_rules() {
  echo "clear rules on $1 and $2"
  tc qdisc del dev $1 root
  tc qdisc del dev $1 ingress
  tc qdisc del dev $2 root
}

init_rules() {
  echo "initialize $1 and $2"
  tc qdisc add dev $1 handle 1: root htb
  tc qdisc add dev $1 handle ffff: ingress
  tc filter add dev $1 parent ffff: protocol ip u32 match u32 0 0 action
     mirred egress red$
  tc qdisc add dev $2 handle 1: root htb
}

add_delay() {
  latency=$3
  latency=$((latency/2))ms
  port=$((startport+count))
  echo "add $latency latency on $1 and $2 for port $port"
  delay_port $1 $count $port $latency sport
  delay_port $2 $count $port $latency dport
```

```
27 }
28
29 delay_port() {
30   handle=$2
31   handle=$((handle))0
32   tc class add dev $1 parent 1: classid 1:$2 htb rate 100Mbps
33   tc filter add dev $1 protocol ip parent 1: prio 1 u32 match ip $5 $3 0
       xffff classid 1:$2
34   tc qdisc add dev $1 parent 1:$2 handle $handle: netem delay $4
35 }
36
37 clear_rules lo ifb1
38 clear_rules eth0 ifb0
39 init_rules lo ifb1
40 init_rules eth0 ifb0
41 for (( i=$1; i<=$2; i+=$3)) do
42     count=$((count+1))
43     add_delay lo ifb1 $i
44     add_delay eth0 ifb0 $i
45 done
```

Anhang D

Firefox unter Linux

Nachdem die Version von *Firefox* in den *Package-Repositories* meist hinter der aktuellen Version nachhinkt, muss eine neuere Version von Hand installiert werden. Die aktuelle Version der Software ist am *Mozilla* FTP-Server[1] verfügbar und kann, wie in [39] beschrieben, nach dem Download entpackt, in den Ordner /opt/firefox verschoben und verlinkt werden.

```
tar xjf firefox-23.0.1.tar.bz2
mv firefox /opt/firefox
sudo mv /usr/bin/firefox /usr/bin/firefox-old
sudo ln -s /opt/firefox/firefox /usr/bin/firefox
```

Auf einem frisch installierten Server-System sind allerdings noch einige zusätzliche Schritte notwendig, da üblicherweise einige Pakete fehlen. Außerdem muss für gewöhnlich der Installationspfad erst noch in die Umgebungsvariable `LD_LIBRARY_PATH` gespeichert werden.

```
export LD_LIBRARY_PATH=/opt/firefox/
```

Die fehlenden Pakete können mit folgendem Befehl ermittelt werden:

```
ldd /opt/firefox/libxul.so|grep not
```

In der Regel sollte folgender Befehl alle notwendigen Pakete nachinstallieren:

```
apt-get install libxrender1 libasound2 libpango1.0-0 libgtk2.0-0
```

Um nun testen zu können, ob *Firefox* ohne Probleme funktioniert, muss ein virtueller *Framebuffer* installiert werden, der die Bildschirmausgaben entgegennimmt. Weiter wird *ImageMagick* benötigt, um einen Screenshot der Ausgaben erstellen zu können, damit sichtbar wird, ob Firefox wie erwartet arbeitet. Die Installation von *Xvfb* und *ImageMagick* erfolgt einfach über `apt-get`.

```
apt-get install xvfb imagemagick
```

[1] ftp://ftp.mozilla.org/pub/firefox/releases/latest/linux-x86_64/en-GB/

Mit den folgenden Befehlen kann nun ein Screenshot erstellt werden, welcher im Erfolgsfall die Menüleiste von *Firefox* und eine leere Seite, ohne Fehlermeldungen, zeigen sollte:

```
Xvfb :1 -screen 0 1280x960x24 &
DISPLAY=:1 firefox about:blank &
DISPLAY=:1 import -window root -crop 1264x948+0+0 -quality 90 /tmp/
    screen.jpg
```

Anschließend können der Browser und *Xvfb* mittels `pkill` beendet werden.

```
pkill -9 firefox
pkill -9 Xvfb
```

Konfiguration von Firefox

Um Probleme beim Browserstart zu vermeiden, ist es empfehlenswert, gewisse Einstellungen des Browsers zu deaktivieren und das Profil vor jedem Start zu löschen. Um automatische Updates und andere Einstellungen, die den Startvorgang beeinflussen könnten, permanent zu deaktivieren, muss eine spezielle Konfigurationsdatei erstellt werden. Diese Datei muss im Installationsverzeichnis von *Firefox* liegen und kann z. B. mozilla.cfg heißen. Als erste Zeile muss diese Datei zwei Slashs enthalten, danach folgen die gewünschten Einstellungen.

Mit den folgenden Angaben können etwaige Abfragen, die das Ausführen der JSTD-Tests verhindern könnten, deaktiviert werden.

```
//
lockPref("browser.shell.checkDefaultBrowser", false);
lockPref("browser.search.update", false);
lockPref("signon.rememberSignons", false);
lockPref("extensions.update.enabled", false);
lockPref("extensions.update.autoUpdate", false);
lockPref("app.update.enabled", false);
lockPref("app.update.auto", true);
lockPref("app.update.mode", 0);
lockPref("browser.sessionstore.resume_from_crash", false);
lockPref("browser.privatebrowsing.autostart", true);
lockPref("datareporting.healthreport.service.enabled", false);
```

Damit *Firefox* diese Einstellungen nun berücksichtigt, muss die Datei /opt/firefox/browser/defaults/preferences/firefox.js folgende Zeilen enthalten:

```
pref("general.config.obscure_value", 0);
pref("general.config.filename", "mozilla.cfg");
```

Falls der Pfad und die Datei nicht existieren, müssen sie erst noch erstellt werden:

```
mkdir -p /opt/firefox/browser/defaults/preferences
touch /opt/firefox/browser/defaults/preferences/firefox.js
```

Danach sucht *Firefox* beim Start im Installationspfad nach mozilla.cfg.

Trotz dieser Einstellungen kann es dennoch vorkommen, dass der Browser nicht richtig gestartet wird, da Firefox in den neueren Versionen einige Bugs enthält. Daher ist es empfehlenswert, vor dem Testdurchlauf den Profil Ordner zurückzusetzen, wodurch diese Probleme umgangen werden. Dazu reicht es, den Inhalt des Profilordners zu löschen und die Datei prefs.js neu zu erstellen.

```
rm -Rf ~/.mozilla/firefox/#profilname#/*; touch prefs.js
```

Der Pfad zum Profil muss entsprechend angepasst werden.

Quellenverzeichnis

Literatur

[1] Rahul Amin u. a. „Assessing the Impact of Latency and Jitter on the Perceived Quality of Call of Duty Modern Warfare 2“. In: *Human-Computer Interaction. Users and Contexts of Use*. Hrsg. von Masaaki Kurosu. Bd. 8006. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, S. 97–106.

[2] Grenville Armitage, Mark Claypool und Philip Branch. *Networking and Online Games: Understanding and Engineering Multiplayer Internet Games*. Chichester: John Wiley & Sons, Ltd., 2006.

[3] Tom Beigbeder u. a. „The Effects of Loss and Latency on User Performance in Unreal Tournament 2003“. In: *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*. ACM. Portland, Aug. 2004, S. 144–151.

[4] Yahn W. Bernier. *Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization*. Präsentiert auf der GDC2001. März 2001. URL: http://www.gamedevs.org/uploads/latency-compensation-in-client-server-protocols.pdf.

[5] Paul Bettner und Mark Terrano. *1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond*. Präsentiert auf der GDC2001. März 2001. URL: http://gamedevs.org/uploads/1500-archers-age-of-empires-network-programming.pdf.

[6] Kuan-Ta Chen, Cheng-Chun Tu und Wei-Cheng Xiao. „OneClick: A framework for measuring network quality of experience“. In: *INFOCOM 2009, IEEE*. IEEE. 2009, S. 702–710.

[7] Peng Chen und Magda El Zarki. „Perceptual View Inconsistency: An Objective Evaluation Framework for Online Game Quality of Experience (QoE)“. In: *Proceedings of the 10th Annual Workshop on Network and Systems Support for Games*. IEEE. Okt. 2011. Kap. 2, S. 1–6.

[8] Stuart Cheshire. *Latency and the Quest for Interactivity*. White paper commissioned by Volpe Welty Asset Management, L.L.C., for the Synchronous Person-to-Person Interactive Computing Environments Meeting. San Francisco, Nov. 1996. URL: http://stuartcheshire.org/papers/LatencyQuest.pdf.

[9] Aiman Erbad und Charles Krasic. „Sender-side buffers and the case for multimedia adaptation“. In: *Communications of the ACM* 55.12 (2012), S. 50–58.

[10] ITU-T Study Group 9. *Subjective video quality assessment methods for multimedia applications*. ITU-T Recommendation P.910. International Telecommunication Union, Apr. 2008. URL: http://www.itu.int/rec/T-REC-P.910-200804-I.

[11] Yoshiaki Ida u. a. „QoE Assessment of Interactivity and Fairness in First Person Shooting with Group Synchronization Control“. In: *Proceedings of the 9th Annual Workshop on Network and Systems Support for Games*. IEEE. Nov. 2010. Kap. 10, S. 1–2.

[12] Shaolong Li, Changja Chen und Lei Li. „A new method for path prediction in network games“. In: *Computers in Entertainment (CIE)* 5.4 (März 2008). Artikel 8, S. 1–12.

[13] Martin Mauve u. a. „Local-lag and Timewarp: Providing Consistency for Replicated Continuous Applications“. In: *IEEE Transactions on Multimedia* 6.1 (Feb. 2004), S. 47–57.

[14] Glenford J. Myers. *The Art of Software Testing*. 2. Aufl. Hoboken, New Jersey: John Wiley & Sons, Inc., 2004.

[15] Lucas Nussbaum und Olivier Richard. „A Comparative Study of Network Link Emulators“. In: *Proceedings of the 2009 Spring Simulation Multiconference*. Society for Computer Simulation International. 2009. Kap. 85, S. 1–8.

[16] Douglas T. Peck. „The History of Early Dead Reckoning and Celestial Navigation: Empirical Reality Versus Theory“. In: *Early Seafaring Exploration Series* 2.4 (2002), S. 1–18.

[17] Michal Ries, Philipp Svoboda und Markus Rupp. „Empirical Study of Subjective Quality for Massive Multiplayer Games“. In: *Proceedings of IWSSIP 2008. 15th International Conference on Systems, Signals and Image Processing*. IEEE. Juni 2008, S. 181–184.

[18] Zachary B. Simpson. *A stream based time synchronization technique for networked computer games*. März 2000. URL: http://www.mine-control.com/zack/timesync/timesync.html.

[19] Ian Q. Whishaw, Dustin J. Hines und Douglas G. Wallace. „Dead reckoning (path integration) requires the hippocampal formation“. In: *Behavioural Brain Research* 127.1 (2001), S. 49–69.

[20] Amir Yahyavi, Kévin Huguenin und Bettina Kemme. „Interest modeling in games: the case of dead reckoning“. In: *Multimedia Systems* 19.3 (Juni 2011), S. 255–270.

[21] Hua Yu, Daniel Wu und Prasant Mohapatra. „Experimental anatomy of packet losses in wireless mesh networks“. In: *Proceedings of the 6th Annual IEEE communications society conference on Sensor, Mesh and Ad Hoc Communications and Networks*. SECON'09. Piscataway, NJ, USA: IEEE Press, Dez. 2009, S. 126–134.

Online-Quellen

[22] URL: https://www.google.at/#q=lag+unfair (besucht am 20.09.2013).

[23] URL: http://www.bing.com/?q=lag+unfair (besucht am 20.09.2013).

[24] URL: https://duckduckgo.com/?q=lag+unfair (besucht am 20.09.2013).

[25] URL: http://de.wikipedia.org/wiki/Datenrate (besucht am 05.09.2013).

[26] URL: http://en.wikipedia.org/wiki/Category_5_cable#Characteristics (besucht am 11.08.2013).

[27] URL: http://en.wikipedia.org/wiki/Optical_fiber#Index_of_refraction (besucht am 11.08.2013).

[28] URL: http://www.submarinecablemap.com/ (besucht am 11.08.2013).

[29] URL: http://en.wikipedia.org/wiki/Dead_reckoning (besucht am 05.09.2013).

[30] URL: http://de.wikipedia.org/wiki/Softwaretest (besucht am 06.09.2013).

[31] URL: https://wiki.archlinux.org/index.php/Advanced_Traffic_Control (besucht am 07.09.2013).

[32] URL: http://de.wikipedia.org/wiki/Gr%C3%B6%C3%9Fenordnung_%28Zeit%29 (besucht am 19.09.2013).

[33] Jesse Aronson. *Dead Reckoning: Latency Hiding for Networked Games*. Kopie auf DVD (Datei Aronson97.pdf). Sep. 1997. URL: http://www.gamasutra.com/view/feature/3230/ (besucht am 05.09.2013).

[34] Tom Cloonan. *Bandwidth Trends on the Internet... A Cable Data Vendor's Perspective*. Kopie auf DVD (Datei Cloonan11.pdf). ARRIS. Sep. 2011. URL: http://www.ieee802.org/3/ad_hoc/bwa/public/sep11/cloonan_01a_0911.pdf.

[35] *DFC Intelligence Forecasts Worldwide Online Game Market to Reach $79 Billion by 2017*. Kopie auf DVD (Datei PRWeb.pdf). Juni 2013. URL: http://www.prweb.com/releases/2013/6/prweb10796698.htm (besucht am 20.09.2013).

[36] Bruce Dawson. *Two Years (and Thousands of Bugs) of Static Analysis*. Kopie auf DVD (Datei Dawson13.pdf). Juni 2013. URL: http://randomascii.wordpress.com/2013/06/24/two-years-and-thousands-of-bugs-of-static-analysis/ (besucht am 06.09.2013).

[37] Ilya Grigorik. *Latency: The New Web Performance Bottleneck*. Kopie auf DVD (Datei Grigorik12.pdf). Juli 2012. URL: http://www.igvita.com/2012/07/19/latency-the-new-web-performance-bottleneck/ (besucht am 05.09.2013).

[38] *How common is automated testing in game development?* Kopie auf DVD (Datei GDSE.pdf). URL: http://gamedev.stackexchange.com/questions/21397/ (besucht am 11.08.2013).

[39] *How to install Firefox 24 on Linux*. Kopie auf DVD (Datei InstallFirefox.pdf). URL: http://www.libre-software.net/how-to-install-firefox-on-ubuntu-linux-mint (besucht am 28.09.2013).

[40] Noel Llopis. *Stepping Through the Looking Glass: Test-Driven Game Development (Part 1)*. Kopie auf DVD (Datei Llopis05.pdf). Feb. 2005. URL: http://gamesfromwithin.com/stepping-through-the-looking-glass-test-driven-game-development-part-1 (besucht am 06.09.2013).

[41] *Performance-Comparison Testing of IPv4 and IPv6 Throughput and Latency on Key Cisco Router Platforms*. Kopie auf DVD (Datei Cisco07.pdf). Cisco Systems, Inc. 2007. URL: http://www.cisco.com/web/strategy/docs/gov/IPv6perf_wp1f.pdf (besucht am 08.08.2013).

[42] Steve Peterson. *Mobile to be "primary hardware" for gaming by 2016*. Kopie auf DVD (Datei Peterson13.pdf). Apr. 2013. URL: http://www.gamesindustry.biz/articles/2013-04-30-mobile-to-be-primary-hardware-for-gaming-by-2016 (besucht am 05.09.2013).

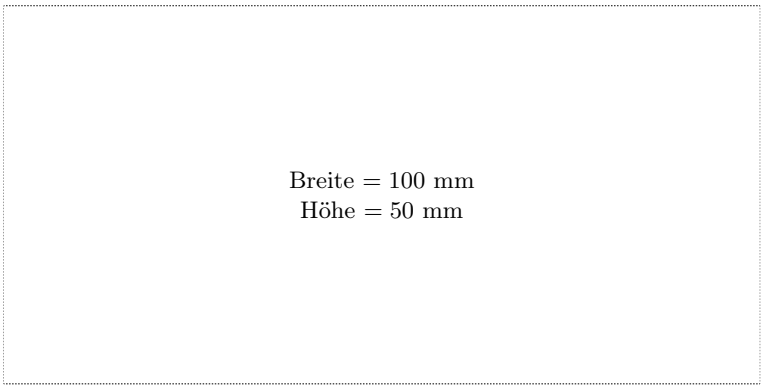
[43] Dean Takahashi. *With online sales growing, video game market to hit $81B by 2016 (exclusive)*. Kopie auf DVD (Datei Takahashi11.pdf). Sep. 2011. URL: http://venturebeat.com/2011/09/07/with-online-sales-growing-video-game-market-to-hit-81b-by-2016-exclusive/ (besucht am 28.09.2013).

[44] *Traffic Control HOWTO*. Kopie auf DVD (Datei TcHowTo.pdf). URL: http://www.tldp.org/HOWTO/Traffic-Control-HOWTO/ (besucht am 07.09.2013).

[45] *Traffic Control Manual*. Kopie auf DVD (Datei TcManPage.pdf). URL: http://lartc.org/manpages/tc.txt (besucht am 07.09.2013).

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —