# Internet-Echtzeitspiele für mobile Netzwerke

## Studienarbeit

von
cand. inform. Tobias Schröter

### Aufgabenstellung und Betreuung

Dipl.-Inform. Oliver Wellnitz

Institut für Betriebssysteme und Rechnerverbund
Technische Universität Braunschweig

Braunschweig, 02.12.2003

# Erklärung

ich versichere,
die vorliegende Arbeit
selbstständig
und nur unter Benutzung
der angegebenen Hilfsmittel
angefertigt zu haben.

Braunschweig,
Dienstag, 2. Dezember 2003

_____

# Kurzfassung

Diese Arbeit beschäftigt sich mit den zu erwartenden Problemen von Mehrbenutzer-Echtzeitspielen, im Speziellen *First Person Shooter,* in zukünftigen mobilen Ad-Hoc Netzen und deren Lösungen. Dazu werden zunächst, aktuell im Einsatz befindliche und desweiteren experimentelle, Netzwerk-Architektur-Modelle hinsichtlich ihrer Eignung für mobilen Ad-Hoc Netze überprüft. Als nächstes werden die Ursachen und Arten von Netzwerk-Verzögerungen und damit der kritischste Punkt, beim Spielen über einen Kommunikationskanal, durchleuchtet. Anknüpfend werden die Techniken aktueller Internet-Echtzeitspiele zur Bekämpfung dieses und weiterer Probleme vorgestellt. Zur Veranschaulichung wird danach auf einige konkrete Realisierungen aus populären *First Person Shooter* Spielen eingegangen. Im Anschluss werden die auftretenden Probleme in mobilen Ad-Hoc Netzen erläutert. Es werden Methoden zu ihrer Bekämpfung vorgestellt und untersucht, in wie weit sich herkömmliche Lösungsmethoden auch in Spielen für mobile Ad-Hoc Netze anwenden lassen.

**Originales Aufgabenblatt**

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Einleitung 1

## 1.1 Danksagung

Bedanken möchte ich mich bei Oliver Wellnitz für die Idee zu dieser Arbeit. Es ist ein Lichtblick hinsichtlich der Einbeziehung des Bereichs der Computerspiele-Entwicklung in den universitären Lehrbereich.

## 1.2 Einführung

Mehrbenutzerspiele repräsentieren mittlerweile eine der populärsten Arten der Gruppenkommunikation im Internet. In den letzten Jahren wurden Computerspiele mit Mehrspieler-Option bzw. reine Online-Mehrbenutzerspiele, immer beliebter und kommerziell erfolgreich. Letzt genannte sind eine der wenigen Internet-Service deren Nutzer bereit sind, eine monatliche Gebühr zu entrichten [SIG03]. In beiden Bereichen kann man alle Spiele-Genre wieder finden, hierzu zählen die ***F**irst **P**erson **S**hooter (Quake, Unreal Tournament, Counterstrike … )*, die ***R**ealtime **S**trategy **G**ames (Age of Empires, Command & Conquer…)* und die *Adventures (Ultima Online, Everquest, Diablo…)*. Man muß jedoch im Bereich der Onlinespiele, auf Basis der unterstützten Anzahl von Mitspielern pro Spielsession, zwischen denen mit weniger als 100 und den Massive-Multiplayer-Spielen *(MMP)* mit weit über 1000 Mitspielern, unterscheiden. Diese Arbeit befasst sich nun schwerpunktmäßig mit FPS-Spielen, die einen Online-Mehrspieler-Modus besitzen und bis zu 64 Spieler[1] pro Session unterstützen.

Im Moment zeichnet sich der Trend ab, dass durch die zunehmende Verfügbarkeit des Internets, vor allem durch mobile Geräte, die Zahl der Nutzer von Mehrbenutzerspielen einem schnellen Wachstum unterliegt und sich so ein zukünftiger Massenmarkt entwickelt. Eine diesen Trend unterstützende Entwicklung ist die der Mehrbenutzerspiele in mobilen Ad-hoc Netzen. Ad-hoc Netze sind drahtlose Netzwerke, deren Teilnehmer sich spontan, ohne Unterstützung von Infrastruktur, zusammenfinden und kommunizieren können. Die daraus folgende, größte Schwierigkeit bei einer evtl. Portierung eines *First Person Shooter*, ist die

[1] aktuelle obere Grenze in FPS, gesetzt von *Battlefield 1942*

Mobilität der kooperierenden[2] Endgeräte. Leistungsfähigkeit und Laufzeit der Geräte sind obendrein in vielen Fällen noch nicht ausreichend, um als Plattformen für Echtzeitspiele zu fungieren. Das gilt teilweise leider auch für die zur Verfügung stehende Bandbreite. Es ist jedoch abzusehen, dass die technologischen Vorraussetzungen in der Zukunft geschaffen werden. Zumindest insofern, das grafisch abgespeckte und *veraltete* Versionen aktueller *First Person Shooter* auf der Mehrheit der Geräte gespielt werden können. Die jeweils aktuellen Spiele werden aufgrund der enormen und stetig wachsenden Hardwareanforderung nur einigen wenigen Endgeräten[3] vorbehalten bleiben.
In dieser Arbeit werden die Problemlösungen heutiger Mehrbenutzerspiele in Hinblick auf ihre Anwendung in mobilen Ad-Hoc Netzwerken untersucht.

## 1.3 Aufbau der Arbeit

In Kapitel 2 werden zunächst einmal die verschiedenen Netzwerkarchitektur-Modelle, auf denen die aktuellen Mehrbenutzerspiele basieren vorgestellt, analysiert und ihre Anwendung in mobilen Ad-Hoc Netzen untersucht. Kapitel 3 beschäftigt sich mit den Ursachen von, während der Internet-Kommunikation auftretenden, Verzögerungen. Im anschließenden Kapitel 4 werden Konzepte zur Lösung dieses und weiterer Probleme von Mehrbenutzerspielen durchleuchtet. Überdies werden einige Implementierungen, der vorgestellten Konzepte, aus aktuellen Mehrbenutzer-Echtzeitspielen in Kapitel 5 aufgezeigt. Kapitel 6 beschäftigt sich letztlich, nach einem Überblick der Schwierigkeiten von Mehrbenutzerspielen in mobilen Ad-Hoc Netzen, mit der Anwendung der gewonnenen Erkenntnisse zu ihrer Bekämpfung. Das letzte Kapitel stellt eine Zusammenfassung der zuvor präsentierten Erkenntnisse dar und gibt einen Ausblick auf, im Rahmen der Arbeit, interessante Fragestellungen und zukünftige Untersuchungen.

[2] d.h. jedes Gerät leitet Nachrichten für andere Geräte weiter *(Forwarding im multi-hop Netzwerk)*
[3] z.B. Laptops mit hoher Performance oder zukünftig spez. entwickelte Spiel-Endgeräte

# Netzwerkarchitektur-Modelle 2

In diesem Kapitel werden die verschiedenen Netzwerkarchitektur-Modelle, auf denen die Kommunikation aktueller Mehrbenutzerspiele basiert, vorgestellt und analysiert. Es wird weiterhin untersucht, in wie weit sich diese Modelle in mobilen Ad-Hoc Netzwerken anwenden lassen.

## 2.1 Peer-to-Peer-Netzwerkmodelle

Die ersten Mehrbenutzerspiele, wie *Doom, Descent,…* nutzten diese, am leichtesten zu implementierende, Art der Kommunikation. Wesentliches Merkmal dieser Modelle ist das jeder Teilnehmer mit allen anderen verbunden ist. Man unterscheidet je nach gebotener Funktionalität mehrere Arten:

### 2.1.1 Unicasting-Systeme

Ein Peer-to-Peer Unicasting-Modell ist die **einfachste Art** der verwendeten Systeme. Es besteht aus nur einer Applikation, die aus der Spiellogik und einem Modul das Nachrichten senden, empfangen, und bearbeiten kann, zusammengesetzt ist.
Bei *n* Teilnehmern muss ein Zustandsupdate eines Spielers, zu *n-1* Mitspielern gesendet werden. Die Folge ist eine hohe Anzahl von Nachrichten und evtl., je nach Art des verwendeten Protokolls *(3.2)*, offenen Verbindungen. Das System skaliert somit **quadratisch** zu der Anzahl der Mitspieler und ist deshalb für große Teilnehmerzahlen ungeeignet.
Nachteile ergeben sich auch bei der Skalierbarkeit der internen Bildwiederholrate *(framerate)*. Diese wird durch die folgende Hauptprogrammschleife bestimmt: Zustands-Updates aller Spieler empfangen → Input des Spielers lesen → neuen Spielerzustand berechnen und versenden → Bild anzeigen ↑. Die Bildwiederholrate muss bei allen Mitspielern auf demselben Level[4] *(frame-locking)* basieren [Treg02], um die Simulationen konsistent zu halten, d.h. die Hauptprogrammschleife kann nicht unterschiedlich

[4] es wird eine Zeitlang gewartet, bis alle Updates von allen Spielern eingetroffen sein müssen

schnell durchlaufen werden. Dies macht es schwierig unterschiedliche Hardwarepotentiale[5] auszunutzen.

Nachteilig ist ebenfalls das sich die Spieler verabreden müssen, um das Spiel gemeinsam zu starten, wenn sie sich nicht im selben Ethernet-Segment befinden und somit Broadcasting *(2.1.2)* nicht zur Spielerfindung verfügbar ist. Neue Spieler können dann auch nicht im Verlauf des Spiels teilnehmen. Eine Lösung des Problems stellt der Einsatz eines bekannten[6] **Slot-Servers** dar. Dessen Aufgabe sich auf die Initialisierung des Peer-to-Peer-Modus, durch Übermittlung einer Teilnehmeradressliste, beschränkt. Jeder Server hat eine gewisse Anzahl von *Slots* zu vergeben, die der maximalen Spieleranzahl pro Session entspricht. Sodass auch noch während des Spiels, solange noch *Slots* frei sind, Clients durch Anfrage an den Server dem Spiel beitreten können. Der Einsatz eines Slot-Servers markiert den Übergang zu einer Client/Server-Architektur *(2.2)* und beschreibt auf diese Weise ein **hybrides System**.

### 2.1.2 Broadcasting/Multicasting-Systeme

Einige der Probleme des oben vorgestellten Systems werden gelöst, wenn das zugrunde liegende Netzwerk das Versenden von Broadcast- oder Multicast-Nachrichten unterstützt. Mit Hilfe von Broadcasting kann man den Kommunikationsaufwand auf **O(n)** reduzieren. Dies bezieht sich auf den Grossteil der Nachrichten, die häufig und unzuverlässig[7] übertragenen Zustandsupdates. Für zuverlässige Übertragungen erhöht sich der Aufwand wegen evtl. wiederholt gesendeter Nachrichten *(retransmits)*. Bei sehr hoch skalierten Systemen mit mehreren hundert Teilnehmern ist der Aufwand trotzdem zu groß, da jeder Client jede Zustandsänderung empfangen und bearbeiten muss, auch wenn die Änderung nicht seine nähere Umgebung beeinflusst und somit irrelevant für das nächste Bild ist. Genau hier setzen Multicasting-Systeme an. Durch eine dynamische Partitionierung des Virtuellen Raums *(4.1.2)* werden Multicast-Gruppen gebildet und somit die Anzahl der Update-Nachrichten reduziert.

[5] höherwertige Hardware ermöglicht eine höhere Framerate und somit flüssige Simulationen höherer Qualität
[6] feste IP-Adresse oder URL
[7] Broadcasting/Multicasting-Übertragungen sind stets unzuverlässig

## 2.2 Client/Server-Netzwerkmodelle

Wegbereiter für die Verwendung der Client/Server-Architektur in der Spielentwicklung war das Spiel *Quake*. Bei diesem Modell sind alle Spieler bzw. Clients nur noch mit einem Server verbunden und nicht mehr untereinander. Die Server-Applikation empfängt und bearbeitet alle Nachrichten bzw. Anfragen *(bewege, erzeuge…)* der Clients und versendet dann gültige Zustandsupdates. Die Bandbreite der Serveranbindung und die Rechenleistung des Servers bestimmen folglich die maximale Anzahl von Spielern pro Server. Dieses Problem des **Server-Flaschenhalses** tritt an die Stelle des Problems der Anzahl der Nachrichten und Verbindungen bei Peer-to-Peer-Architekturen.
Da die Server-Applikation die Zustandsupdates verteilt, kann sie mit Hilfe von Sichtbarkeits-Algorithmen oder anderen Kontext bezogenen Regeln entscheiden, für welche Clients die Updates relevant sind *(4.1.2)*. So kann der Kommunikationsaufwand, ähnlich wie beim Peer-to-Peer Multicasting, reduziert werden.
Der Server kann auch als vertrauenswürdige Instanz, zum Schutz vor Cheatern und Hackern, dienen *(4.1.3)*.
Eine Client/Server-Architektur stellt letztlich die **Beste Lösung**, bei hoher Bandbreite und geringer Latenz, dar. Deshalb wird sie auch von allen aktuellen FPS eingesetzt und im weiteren Verlauf der Arbeit favorisiert.

### 2.2.1 Arten von Client/Server-Modellen

Es gibt verschiedene Arten der Verwendung von Client/Server-Modellen, die durch die Funktionserfüllung *(4.2)* der Clients gekennzeichnet sind. Bei den ersten Mehrbenutzerspielen wie *Quake*, die Client/Server-Modelle verwendeten, spricht man von einer rigorosen *(monolithischen)* Anwendung. Die Clients waren reine Anzeige-Terminals welche die Benutzereingaben an den Server sendeten und daraufhin eine Liste von zu anzuzeigenden Objekten empfingen. Mit der Entwicklung von *Quake 2* wurde dann zunehmend Funktionalität bzw. Eigenverantwortung auf die Clients übertragen. *Unreal* führte schließlich ein **generalisiertes Client/Server-Modell** bei den Mehrbenutzerspielen ein. In diesem Modell bleibt der Server zwar die einzige, über die Evolution des Spielzustands entscheidende Instanz, jedoch spiegelt der Server einen Teil des Gesamtzustands auf jeden der Clients. Dadurch kann der Client, durch Anwendung derselben Spiellogik wie der Server, den

Spielfluss voraussagen. In der Folge kommt es eher selten vor, dass der Server die Position des Clients ändern muss *(5.1)*.

## 2.3 Anwendung in mobilen Ad-Hoc Netzwerken

Das sich im Internet stellende Problem, der Verfügbarkeit der Broadcasting/Multicasting-Unterstützung, wird in mobilen Ad-Hoc Netzen gelöst. Da es sich um lokale Netze handelt sind beide Dienste einsetzbar und man kann die beschriebenen Features nutzen. Der größte Vorteil eines Einsatzes einer **Peer-to-Peer-Architektur** in mobilen Ad-Hoc Netzen liegt jedoch in der **hohen Fehlertoleranz** und in der geringeren Latenz, im Vergleich zu einer Client/Server-Architektur. Der Ausfall eines Spielers ist relativ einfach zu kompensieren, da der Spielzustand zwischen allen bzw. durch alle Teilnehmer propagiert wird. Dessen ungeachtet überwiegen die in Tabelle 2.1 zusammengefassten Nachteile, insbesondere die Anfälligkeit gegenüber Cheatern und die Anwendung des Frame-Locking-Verfahrens. Des Weiteren nutzen die meisten am Markt positionierten Spiele Client/Server-Architekturen. Sodass ein verbreiteter Einsatz dieser Architektur nicht zu erwarten ist.

Das größte Problem der **Client/Server-Architektur**, in Bezug auf mobile Ad-Hoc Netze ist die **Fehleranfälligkeit** des Servers. Auf Grund von fehlender Infrastruktur und der Mobilität der Clients, ist die Übertragung der Serverfunktionalität auf einen der Clients nicht ausreichend fehlertolerant, da beim Ausfall des Servers das gesamte Spiel zusammenbricht und somit alle Clients betroffen sind. Einen ersten Ansatz dieses Problem zu lösen stellt die in [CFK01] vorgestellte **Mirrored-Server-Architektur** dar. Der Spielzustand wird hier auf mehrere topologisch verteilte Mirror-Server gespiegelt. Jeder der teilnehmenden Clients verbindet sich mit dem am nächsten liegenden Mirror-Server *(sogen. ingress mirror)*. Die Server übertragen per zuverlässigen[8] *Multicasting* die eingehenden Kommandos ihrer Clients an die anderen Server *(sogen. egress mirror)* und jeder der Server berechnet den daraus resultierenden aktuellen Spielzustand. Dieser wird den Clients wiederum per *Unicasting* mitgeteilt. Entgegen den Clients sind die Mirror-Server in dieser Architektur über ein privates Netzwerk mit Multicasting-Fähigkeit und geringer Latenz verbunden. Durch diese Trennung der Server-Server-Kommunikation von der Client-Server-Kommunikation und die Aufteilung der Last auf mehrere Server, wird ein weiterer Vorteil dieser Architektur, die abfallende Bandbreitenanforderung der Mirror-Server bei steigender Clientanzahl[9], im Vergleich zu

[8] unabhängig von der Art des Ereignisses, welches das Kommando repräsentiert
[9] topologisch gleichmäßige Verteilung wird unterstellt

einer zentralisierten Version erreicht. In einer hierfür speziell angepassten Quake-Version wurde z.B. eine Last von 2Mbs an den Mirror-Servern und im Vergleich dazu 6Mbs am zentralen Server gemessen, bei 4 Mirror-Servern und 32 Clients *(s.a. Abbildung 2.1)*.

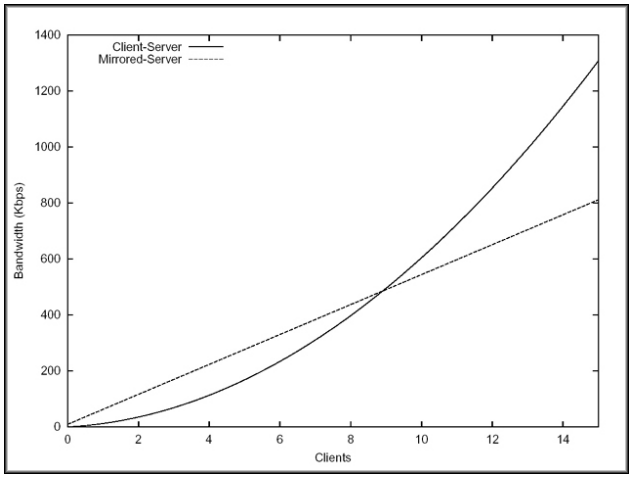


**Abbildung 2.1**
Client/Server und Mirrored-Server Bandbreiten-anforderungen einer modifizierten Quake-Version [CFK01].

Diese Trennung und damit der Vorteil, kann jedoch nicht in mobilen Ad-Hoc Netzen aufrechterhalten werden, da alle Informationen über dasselbe Netz ausgetauscht werden müssen.

Die Motivation für die Entwicklung dieser Architektur lag in der Verringerung der Latenzkosten, die bei der herkömmlichen Kommunikation der Clients mit dem zentralisierten Server entstehen. Explizite Methoden zur Erhöhung der Fehlertoleranz sind deshalb nicht vorgesehen. Einzig die Aufteilung des Spielzustands auf mehrere Server kann man als ersten Schritt in Richtung erhöhte Fehlertoleranz werten.

Den Ansatz der Verteilung des Spielzustands aufgreifend und die Fehlertoleranz im Blickfeld wurde in [RWW03] eine Erweiterung der Standard Client/Server-Architektur, die **Zone-based-Gaming-Architektur**, vorgestellt. Welche speziell zur Anwendung in mobilen Ad-Hoc Netzen entworfen wurde. Bei dieser Architektur werden mehrere Clients zusätzlich als **Zonen-Server**[10] ausgewiesen und sind dann für eine Teilmenge der am Spiel beteiligten Clients verantwortlich. Je nach Leistungsfähigkeit der Client-Hardware kann der Spieler entscheiden ob sein System zusätzlich als Zonen-Server fungieren soll. Die Zonen-Server empfangen die Updates ihrer Clients, berechnen den sich daraus ergebenen Spielzustand und senden diesen per *Multicasting*[11] an die anderen Zonen-Server. Nachdem ein Zonen-

[10] nicht zu verwechseln mit den Zoned-Server aus *(4.1.1)*
[11] zuverlässiges und unzuverlässiges *Multicasting* wird unterstützt

Server eine solche Änderung empfangen hat oder selbst einen neuen Spielzustand generiert hat, wird dieser wiederum mit Hilfe von *Multicasting* an die mit ihm verbundenen Clients verschickt. Im Angesicht der *First Person Shooter* ist die Übertragung der Spielzustände zwischen den Zonen-Servern, im Gegensatz zu einer Übermittlung der Client-Kommandos, wie z.B. zwischen den Mirror-Servern, allerdings eher als kritisch zu bewerten. Einerseits können die sich aus der Verarbeitung der Client-Kommandos ergebenen Zustände weitaus mehr Bandbreite konsumieren und andererseits sind die vorzunehmenden Änderungen an dem Server-Code einer bestehenden Client/Server-Implementierung wesentlich komplizierter [CFK01].
Ebenfalls zu beachten ist, dass man in beiden Architekturen nun nicht mehr nur eine autoritative Instanz hat, sondern das sich die Server für ihre Entscheidungen untereinander abstimmen müssen. Eine Lösung für die Konsistenzerhaltung, des durch die Server gemeinsam verwalteten Spielzustands ist in der Zonen-Server-Architektur noch nicht realisiert. Es wird jedoch auf einige, noch zu implementierende, Synchronisationsverfahren verwiesen, wovon das für den Bereich der *First Person Shooter* relevanteste Verfahren[12] *(trailing state synchronisation)* in dieser Arbeit *(6.1.3)* noch näher erläutert wird.
Die für mobile Ad-Hoc Netze notwendige Fehlertoleranz wird erreicht, indem die Clients sich im Fehlerfall an einen geeigneten neuen Zonen-Server wenden können. Dies geschieht falls der momentan zuständige Server ausfällt oder die von den Clients überwachte Latenz zu groß wird. Des Weiteren wird, in dem Fall das ein Zonen-Server den Kontakt zu allen anderen Zonen-Servern verliert, dass Spiel aufgeteilt und der Server wird allein autoritativ.
Auch wenn in diesem Entwurf noch nicht alle Teilprobleme wie:

- Synchronisation des Spielzustands
- alternatives Versenden von Kommandos zwischen den Servern für existierende FPS
- automatische Zuweisung der Geräte als Zonen-Server
- Integration neuer Zonen-Server nach der Isolation eines Zonen-Servers
- Wiedervereinigung zuvor aufgeteilter Spiele

gelöst sind. Stellt die Verwendung von mehreren Servern, die den Spielzustand gemeinsam verwalten, als Erweiterung der Client/Server-Architektur, die Architektur zukünftiger Mehrbenutzerspiele in mobilen Ad-Hoc Netzen dar. Solange die Server-Server-Kommunikation nicht zu große Verzögerungen verursacht, erreicht man durch eine solche

[12] wird auch in der zuvor beschriebenen Mirrored-Server-Architektur eingesetzt

Architektur die Vereinigung der Vorteile der Client/Server-Architektur mit einer ausreichenden Fehlertoleranz.

## 2.4 Zusammenfassung

| Architektur-Modell | Pro | Contra |
|---|---|---|
| **Peer-to-Peer** | **- einfache Implementierung**<br>**- hohe Fehlertoleranz**<br>**- Latenzkompensations-Techniken** | **- Anfälligkeit gegenüber *Cheatern***<br>**- Spielzustands-Synchronisierung**<br>**- Frame-Locking-Verfahren** |
| ▪ Unicasting | | - hoher Kommunikationsaufwand<br>- quadratische Skalierung<br>- schwierige Spielinitialisierung |
| ▪ Hybrides System | | - hoher Kommunikationsaufwand<br>- quadratische Skalierung |
| ▪ Broadcasting/Multicasting | - lineare Skalierung<br>- Filterung relevanter Informationen[13] | |
| **Client/Server** | **- Schutz vor *Cheatern***<br>**- Filterung relevant. Informationen**<br>**- autoritativer Server** | **- geringe Fehlertoleranz**<br>**- max. Spieleranzahl pro Server** |
| ▪ Monolithisches System | | - keine Latenz-Kompensation |
| ▪ Generalisiertes System | - Latenzkompensations-Techniken | |
| **Client/Server-Server**[14] | **- Schutz vor *Cheatern***<br>**- Filterung relevant. Informationen**<br>**- abstimmende autoritative Server**<br>**- Latenzkompensations-Techniken**<br>**- hohe Fehlertoleranz** | **- max. Spieleranzahl pro Server**[15]<br>**- Spielzustands-Synchronisierung**[16] |

**Tabelle 2.1**

Zusammenfassung der Netzwerkarchitektur-Modelle und ihrer Eignung für Mehrbenutzerspiele

[13] im Fall von *Multicasting*
[14] zukünftige Architektur wie bspw. *Zone-based Gaming Architekture for Ad-Hoc Networks* [RWW03]
[15] wird durch die Aufteilung der Spieler auf mehrere Server relativiert
[16] Aufwand für die Synchronisierung des Gesamtzustands zwischen den Servern

# Anforderungen an ein Netzwerkprotokoll 3

## 3.1 Ursachen von Latenz in Echtzeitspielen

Allgemein gilt als oberstes Gebot beim Design eines Netzwerkcodes für ein Echtzeitspiel, dass die Verzögerungen *(delays)* durch die Kommunikation der Spielinformationen nicht die Spielbarkeit, das so genannte *Gameplay*, beeinflussen. Diese Verzögerungen auf dem Kommunikationskanal entstehen durch folgende interagierende Faktoren:

- Anzahl der Mitspieler[17]
- Datenmenge der zu übertragenden Spielinformationen
- Bandbreite und Wartezeiten des Kommunikationskanals
- Hardware die der Verbindungsherstellung dient
- Grad der nötigen Zuverlässigkeit bei der Übertragung
- Aufwand bei der Bearbeitung der Nachrichten auf dem Server

Echtzeitspiele wie FPS sind durch rasante Bewegungen und häufige Richtungswechsel gekennzeichnet. Aus diesem Grunde sind sie auf einen ständigen Informationsaustausch zwischen Server und Client angewiesen, um die Simulationen konsistent zu halten. Sodas geringere Bandbreiten mit niedrigeren Wartezeiten ein besseres Spielerlebnis ermöglichen als hohe Bandbreiten mit hohen Wartezeiten.
Hohe Latenz wird dem Spieler auf unterschiedliche Weise bewusst. Bei Spielen mit Client/Server-Architekturen entsteht ein Ruckeln der Simulation bzw. eine verzögerte Reaktion, da jede Aktion von dem Server verarbeitet werden muss. Bei Peer-to-Peer-Architekturen läuft die Simulation flüssig, jedoch entstehen Sprünge in den Spielerpositionen und Spielzuständen. Solche Sprünge entstehen auch in den erweiterten, nicht monolithischen Client/Server-Architekturen, aufgrund von Vorhersagefehlern *(4.2)*. Hierbei sind fluktuierende Latenzzeiten für den Menschen schlechter zu kompensieren, als konstante Latenzen bis zu einer oberen Grenze.
Selbstverständlich spielt auch die Hardware, über die man mit dem Internet oder Ethernet verbunden ist, eine Rolle. So erhöhen zum Bsp. Standard-Modems die Latenz durch

[17] außer bei Peer-to-Peer Multicasting/Broadcasting

Wartezeiten, bedingt durch interne Buffer, Kompressions- und Fehlerkorrekturverfahren. Durch Ausschalten solch entsprechender „Features“, erreicht man das die Pakete so schnell wie möglich versendet werden. Die erste Ebene auf der man Kommunikationsverzögerungen entgegenwirken kann, ist die Wahl des Netzwerkprotokolls in Abhängigkeit von der Art der zu übertragenden Spielinformationen.

## 3.2 Informationsabhängige Wahl des Netzwerkprotokolls

Bei der Wahl des zu verwendenden Protokolls gilt es zu beachten, dass die Verzögerungen auf dem Kommunikationskanal invers Proportional zur Zuverlässigkeit der Datenübertragung sind. Des Weiteren muss die auftretende Latenz der versendeten Informationen im Bereich unterhalb von 225ms liegen, um ein akzeptables *Gameplay* zu erreichen [Hend01]. Weitere Untersuchungen im Rahmen von *Quake III Arena* haben ergeben, das die Latenz, besonders im Bereich der *First Person Shooter*, sogar unter 150 ms liegen muss, um die Spieler für längere Zeit *bei Laune zu halten* [Armit01]. Die Folge ist eine Aufteilung der Spieleinformationen in:

- Daten bei denen es auf die Reihenfolge und die zuverlässige Übertragung ankommt
    - z.B. Objekte erzeugen, übereignen, entfernen
    - Allgemein wird in dieser Kategorie TCP verwendet

- Daten die sehr oft und besonders schnell übertragen werden müssen
    - z.B. Spielerpositionen
    - In dieser Kategorie wird allgemein UDP eingesetzt und zusätzlich evtl. Sequenznummern und einfache Bestätigungen *(acknowledgements) (5.2.2).*

In einigen Spielen wird nur auf unzuverlässige Übertragung gesetzt und je nach Art der Information entschieden, wie Paketverluste kompensiert werden. Im ersten Fall wird dann eine zuverlässige Übertragung simuliert, indem Bestätigungen und Paketwiederholungen eingesetzt werden. Andererseits werden Paketverluste im zweiten Fall nicht speziell behandelt sondern geduldet, da eine neue, aktuellere Übertragung ohnehin kurz bevorsteht. Entsprechendes gilt auch für den etwaigen Verlust der Reihenfolge der Übertragungen, indem zu spät eintreffende Pakete verworfen werden.

Ansätze zur Entwicklung eines zuverlässigen Protokolls basierend auf UDP, speziell für die Übertragung von Spieldaten, haben sich als nicht praktikabel erwiesen [Treg02].

## 3.3 Zusammenfassung

Zusammenfassend kann man 3 Ebenen, auf denen Verzögerungen entstehen können, identifizieren:

**Verzögerungen auf Hardwareebene**, bedingt durch:

- Geringe Bandbreiten und hohe Wartezeiten des Kommunikationskanals
- Die Netzwerkverbindung herstellende Hardware
  - Standardmodems, Satellitenverbindungen

**Verzögerungen auf Protokollebene**, bedingt durch:

- Verwendete Netzwerkprotokolle
  - Schnelles, unzuverlässiges UDP im Gegensatz zum langsamen, zuverlässigen TCP.
  - Bei zuverlässigen Übertragungen wird die Latenz durch TCP- oder UDP-Paketwiederholungen[18] erhöht.
  - Das Warten auf eigene oder automatische Bestätigungen vergrößert die Latenz.

**Verzögerungen auf Applikationsebene**, bedingt durch:

- Verwendetes Netzwerkarchitektur-Modell
  - Die Client/Server-Architekturen erzeugen eine höhere Latenz, als Peer-to-Peer-Architekturen, da die Informationen erst zum Server und wieder zurück übertragen werden müssen.
- Zeit für die Informationsverarbeitung auf dem Server
  - Hängt von der Anzahl der Mitspieler, dem Aufwand zur Berechnung des neuen Zustands und der evtl. nötigen Server-Server-Kommunikation ab.
- Datenmenge der zu übertragenden Spielinformationen

Im Rahmen von mobilen Ad-Hoc Netzen stellen insbesondere die aus den Verzögerungen auf Hardwareebene resultierenden Probleme, die Herausforderung für Mehrbenutzer-Echtzeitspiele dar *(Kap. 6)*. Die für Internet-Mehrbenutzerspiele entwickelten Techniken zur

[18] im Fall von TCP sind automatische Wiederholungen *(retransmits)* gemeint

Kompensation der Verzögerungen auf Protokoll- und Applikationsebene lassen sich auch in mobilen Ad-Hoc Netzen anwenden und werden im folgenden Kapitel vorgestellt.

# Problemlösungen für Mehrbenutzerspiele 4

Dieses Kapitel befasst sich mit Problemen, die bei der Entwicklung von Mehrbenutzerspielen auftreten und stellt jeweils aktuelle Techniken zu ihrer Bekämpfung vor. Die meisten Probleme resultieren aus den Auswirkungen der in Kapitel 3 vorgestellten Latenz in Mehrbenutzerspielen.
Im Folgenden werden Techniken zur Erweiterung der Client/Server-Architektur, kategorisiert nach ihrer Zugehörigkeit zur Client- oder Server-Seite, vorgestellt. Pioniere auf dem Gebiet waren *QuakeWorld* und *Quake 2*, welche erstmals zusätzliche Simulations- und Vorhersage-Techniken auf der Client-Seite einführten.

## 4.1 Techniken der Server-Applikation

### 4.1.1 Unterteilung des virtuellen Raums

Die **Skalierbarkeit** der Client/Server-Architektur kann erhöht werden, in dem man die Virtuelle Welt auf mehrere Server aufteilt. Jeder der Server behandelt dann nur die Clients, die in seinem Teil der Welt agieren. Man unterscheidet zwei Konzepte, die **Seamless**- und die **Zoned-Server**, nach der Wahrnehmbarkeit der Teilweltübergänge durch den Spieler. In einer nahtlosen *(seamless)* Welt interagiert der Spieler, an den Randbereichen des Servers auf dem er gerade spielt, unbewusst mit Objekten die jeweils von einem anderen Server kontrolliert werden können. Abbildung 4.1 stellt ein Beispiel für einen nahtlosen Übergang zwischen 2 Servern dar, indem der Spieler durch den Höhleneingang mit Objekten und Spielern des anderen Servers interagieren kann. Bei einer in Zonen aufgeteilten Welt, interagiert der Spieler nur mit Objekten die sich auch auf seinem Server befinden. Die Übergänge zwischen den Teilwelten werden bei dieser Art der Unterteilung durch Portale, Teleporter oder ähnliches ermöglicht, das heißt man kann nicht während der Übergänge interagieren und die simulierte Welt erscheint somit stellenweise unrealistischer. Die Unterteilung des Spieluniversums erfolgt jeweils, bei auf Terrain basierenden Spielen, über ein Welt umspannendes 2D Raster. Neben der Komplexität ist das wesentlichste Seamless-Server-

**Abbildung 4.1**
Beispiel eines Seamless-Server aus *Asheron's Call 2* [Thor03].

Problem, die Server-Grenzen überschreitenden Sichtbarkeits-Bereiche *(allg. Interessensbereiche)* der Spieler. Die gemeinsamen Randbereiche adjazenter Server-Regionen müssen mindestens gleich groß dimensioniert sein, wie die Radien der Sichtbarkeits-Bereiche der Spieler *(Abbildung 4.2)*. Dadurch werden Fehler bei der Erzeugung verteilter Proxy-Objekte, welche temporäre Referenzen von Objekten anderer Server repräsentieren, innerhalb der Randbereiche vermieden. Diese Proxy-Objekte werden auf allen Servern die den Randbereich teilen erzeugt, wenn ein Objekt diesen Randbereich betritt. Je nachdem auf welcher Server-Seite der Randbereich verlassen wird, wird die Kontrolle eventuell auf einen der Proxies und damit auf einen anderen Server übergeben, die restlichen Proxies werden wieder zerstört. Ein Beispiel für einen zu kleinen Randbereich beschreibt Abbildung 4.3: Spieler 1 kann mit dem Proxy von Spieler 2 interagieren, weil es in seinem Sichtbarkeits-Bereich liegt. Spieler 2 ist dies jedoch nicht möglich weil sich Spieler 1 noch nicht im Randbereich befindet und somit kein Proxy angelegt wurde, obwohl es in seinem Sichtbarkeits-Bereich liegt.

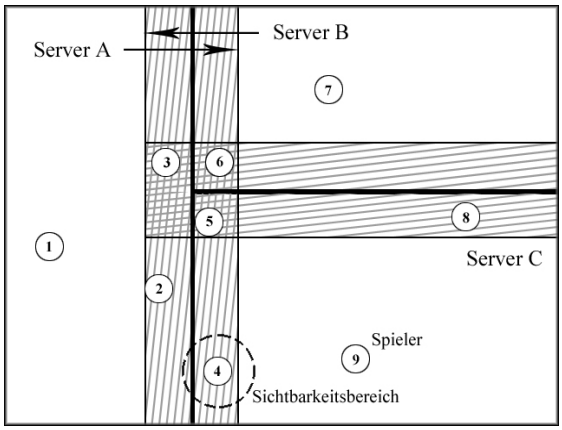


**Abbildung 4.2**
Server-Randbereiche sind schraffiert dargestellt und von Obj. 5 müssen z.B. 2 Proxies auf A, B erstellt werden [Thor03].

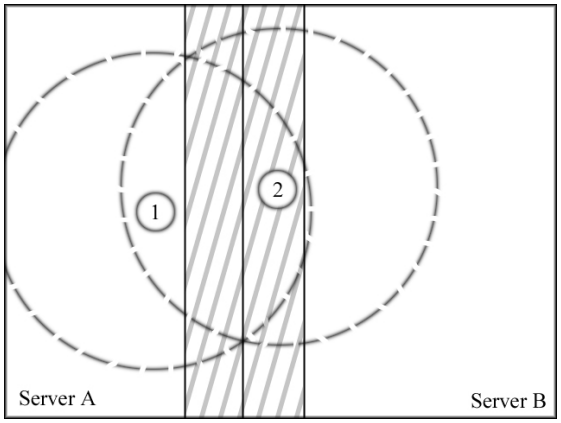


**Abbildung 4.3**
Ein zu klein dimensionierter Server-Randbereich [Thor03].

Beide Konzepte dienen der Erhöhung der unterstützten Anzahl von Mitspielern und zur Vergrößerung der Ausmaße der Spielwelt. Die Vorteile des Einsatzes von Seamless-Servern gegenüber Zoned-Servern sind:

- Realistischer wirkenden Welten.
- Flexiblere Skalierbarkeit, da die Aufteilung der Welt aufgrund von beobachteten Spielerpopulationen nachträglich angepasst werden kann.
- Höhere Zuverlässigkeit. Beim Ausfall eines Servers können die Servergrenzen angepasst und somit die Last auf die verbleibenden Server verteilt werden.
- Wenn sich die Servergrenzen[19] sogar während des Spiels ändern können, sind die Vorteile automatisch bzw. zur Laufzeit einsetzbar.
- Ein Vorteil für die Client-Seite ist, dass lange Map-Ladezeiten[20] wegfallen, da immer nur kleine Teile der Welt zur Laufzeit geladen werden.

Der nicht zu unterschätzende Nachteil des Seamless-Konzepts ist die hohe Komplexität einer solchen Imlplementierung, die alle Bereiche der Spielentwicklung beeinflusst z.B.:

- Server/Server-Kommunikation zur Synchronisierung der Proxy-Objekte, wobei dieselben Probleme wie bei divergierenden Client/Server-Simulationen auftreten.
- Akkuratheit-Durchsatz-Tradeoff bei der Repräsentation der Proxies.
- Spielweltgestaltung

Eine detaillierte Ausführung der Pro/Contra-Argumente einer Seamless-Server-Architektur findet man in [Thor03].
Andere Verfahren zur Aufteilung der Serverlast, wie **separate KI- oder Physik-Server**, bei denen ein Front-End-Server sich um die Anfragen des Clients und die Verteilung der Aufgaben auf die spezial Server kümmert, sind wesentlich nachteiliger hinsichtlich Erweiterbarkeit und gleichmäßiger Verteilung. Die Effizienz dieses Ansatzes hängt davon ab, wie gleichmäßig sich die zur Berechnung des neuen Spielzustands anstehenden Aufgaben aufteilen lassen. Insbesondere in *First Person Shooter* Spielen besteht die Hauptlast aus physikalischen Berechnungen *(Bewegung, Kollisionserkennung)*, so dass eine solche Aufteilung keinen Skalierungsvorteil bringt und schlechter skaliert als Raumunterteilungs-Verfahren.

[19] dynamische vs. statische Implementierung
[20] die Spielumgebung beschreibende Daten, wie Geometrie, Texturen…

### 4.1.2 Filterung relevanter Informationen

Neben der Aufteilung der Last auf mehrere Server, wie im obigen Abschnitt beschrieben, kann man zusätzlich noch **pro Server** eine Art Datenverteilungs-Management implementieren. Damit nicht mehr alle Nachrichten von allen beteiligten Spielern an alle Clients gesendet werden müssen. Dies ist möglich, da jedes Objekt einen Sichtbarkeits- bzw. Einflussbereich besitz, außerhalb dessen müssen Änderungen am Spielzustand nicht unmittelbar dem Objekt propagiert werden. Objekte können mehrere solcher *„Bereiche des Interesses“* besitzen, je nach Art *(Audio, Visuell)* sind sie unterschiedlich dimensioniert.
Die Methoden zur Filterung ähneln bzw. sind mit den Methoden zur Kollisionserkennung und Raumunterteilung verwandt.

#### 4.1.2.1 Rasterbasierte Filterung

Bei dieser Methode wird das Spiel-Terrain wieder durch ein 2D-Raster in quadratische Zellen unterteilt und jeder dieser Zellen wird eine Multicast-Adresse zugewiesen. In Client/Server-Architekturen muss evtl. ein Konzept für Kommunikations-Gruppen *(Multicasting Alternativen)* implementiert werden. Updates sind dann für einen Spieler nur noch relevant, wenn sie aus der Menge ihn umgebender Zellen stammen, Nachrichten anderer Clients werden nicht empfangen. Wenn sich das vom Client gesteuerte Objekt bewegt wird diese Menge dynamisch angepasst. Die Größe der Zellen muss einerseits an die Interessensbereiche der Objekte angepasst werden und andererseits bestimmt sie die Effizienz des Verfahrens.
Eine weitere Möglichkeit ist die Unterteilung der Spielwelt in konvexe Volumen, wie sie in *Quake* eingesetzt wurde *(5.2.1)*. Hierbei wird unter zu Hilfenahme von vorberechneten Sichtbarkeitsinformationen jeweils entschieden, welche der den Spieler umgebenden 3D-Zellen und damit die darin enthaltenen Objekte, für den Spieler relevant sind.

#### 4.1.2.2 Objektbasierte Filterung

Ein feinkörnigerer Relevanztest, der eine höhere Verkehrsreduktion erreicht ist die Objektbasierte Filterung. Bei dieser Methode werden zur Bestimmung der zu versendenden Updates nur noch die Interessensbereiche der Objekte untersucht. Wenn sich also der Interessensbereich des Spielers mit denen einiger Objekte oder anderer Spieler schneidet, werden etwaige Zustandsänderungen an den Objekten dem Spieler bzw. dem jeweiligen Client propagiert. Zu beachten ist das der Aufwand auf dem Server für diesen Relevanztest nicht zu groß wird, d.h. die Anzahl der zu untersuchenden Objekte bestimmt die Effizienz des Verfahrens.

Als ein Implementierungsbeispiel ist die *Unreal-Engine* zu nennen, sie verwendet eine Variante eines solchen Relevanz-Tests *(5.1.1.1)*.

### 4.1.3 Schutz vor Cheatern und Hackern

Böswillige Veränderungen von sensitiven Spieldaten in Peer-to-Peer-Architekturen zu verhindern ist hoffnungslos, da man keinem der Clienten trauen kann. Bei Client/Server-Architekturen hingegen eignet sich der Server als vertrauenswürdige Instanz und somit als Speicherort für alle sicherheitskritischen Daten. Dadurch sind allerdings auch die Konsequenzen von Manipulationen am Server gravierender und können zur Destabilisierung des Spiels führen und somit alle Spieler betreffen.
Die meisten Protokoll Hacker ändern einfach einige Bytes in den Paketen und werten die Resultate aus. Eine mögliche Abwehr würde die Berechnung von **Checksummen**, über jedes einzelne Paket, darstellen. Eine Checksumme ist eine kurze Zahl die durch eine verfahrensabhängige Kombinierung der Paketbytes *(Header und Userdaten)* gebildet wird. Diese wird dann zusammen mit dem Paket versendet. Der Empfänger vergleicht dann die selbst berechnete mit der übertragenen Checksumme und weist nicht übereinstimmende Pakete ab. Perfekte Checksummen, die jeden Byte-Fehler erkennen, erzeugen jedoch einen zu großen Daten-Overhead pro Paket. **One-Way-Hashfunktionen** wie MD-5, eine Online-Public-Domain-Implementierung findet man in [Plumb93], sind in diesem Fall als Checksummen-Generatoren besser geeignet. Sie sind weit verbreitet und schnell[21] genug für Echtzeitspiele. Eine Hashfunktion $h$ erzeugt aus einer beliebig langen Nachricht $M$ eine Zusammenfassung $h(M)$ einer festen Größe, 128Bit im Fall von MD5. Dieser Fingerabdruck $z$ hängt von der gesamten Nachricht ab und man spricht von einer One-Way-Hashfunktion, wenn es für dieses $z$ berechnungsmäßig praktisch unmöglich ist eine Nachricht $M$ mit $h(M) = z$ zu finden, also auf die Ursprungsnachricht zurück zu schließen [Wätjen00]. Durch diese Eigenschaften eignen sich die, vor allem bei der Signatur von Dokumenten eingesetzten, Funktionen auch zur Erzeugung von Checksummen. Bei solch einer Verwendung der Fingerabdrücke als Checksummen, muss man sich allerdings noch zweier möglicher Angriffe bewusst werden:

#### 4.1.3.1 Paket Replay

Ein unfairer Spieler kann mit Hilfe von sogenannter *Sniffer Software* gültige Pakete abfangen und duplizieren. Durch wiederholtes senden dieser gültigen Pakete kann man eventuell die

vom Client festgelegte Anzahl der Ausführungen von Spielkommandos unerlaubt erhöhen. Die gültige Anzahl von Ausführungen an ein Zeitintervall zu binden, ist in der Praxis auf Grund der Internet-Latenz ungünstig. Denn eine Verzögerung kann dazu führen das legale, ursprünglich zeitlich versetzte Pakete unmittelbar nacheinander eintreffen und dann als illegal zurückgewiesen werden. Besser sind hier deshalb einzigartige Identifikations-Informationen pro Paket. Wie **linear kongruente Zufallszahlen** *z* der Art $z = (z_{alt} \div a) * b$ wobei *a, b* speziell gewählte Zahlen sind [Knuth98], und *z* vom Sender und Empfänger pro Paket inkrementiert $z_{alt} = z_{alt} + 1$ und neu generiert wird. Erreicht wird so eine sukzessive Folge von Zufallszahlen für sukzessive Pakete, die auf beiden Seiten erzeugt und damit verglichen werden können. Die Initialisierung der Generatoren übernimmt der Server, indem er zufällige Startwerte für $z_{alt}$ generiert und diese den Clients in der ersten Nachricht übermittelt. Eine hierfür geeignete Quelle von Zufallszahlen, mit höherer Genauigkeit als die standardisierte *rand()* C-Funktion, liefert z.B. eine Implementierung in [Booth97]. Bei unzuverlässigen Verbindungen kann die Synchronität der Zufallszahl-Generatoren *z,* aufgrund von Paketverlusten und Reihenfolgeverlusten verloren gehen, da dann die Inkrementierungen divergieren und so unsynchrone *z* erzeugt werden. Eine Lösung stellt in diesem Fall die erweiterte Nutzung von herkömmlichen Paket-Sequenznummern *s* dar, welche meist auch bei unzuverlässigen Übertragungen genutzt werden. Die Distanz[22] der Sequenznummern wird dann als Inkrement für die Berechnung des aktuellen gültigen *z* benutzt. Sollen auch verspätete Pakete *(Out of order)* zugelassen werden müssen alte Zustände *(z, s)* von *z* gespeichert werden.

Diese Technik der sukzessiven Zufallszahlen kann man des Weiteren einsetzen, um den Aufwand des Hackers für die Paketanalyse zu erhöhen. So sollen Pakete in denen die Nutzerdaten identisch sind, durch eine bitweise XOR-Verknüpfung der Nutzerdaten mit einem solchen *z*, den Anschein für unterschiedlichen Inhalt waren. Der Empfänger erhält die eigentlichen Daten wiederum durch eine erneute Verknüpfung mit demselben, aber von ihm generierten *z*. Weiterhin können Pakete mit identischen Längen durch das Anhängen von nutzlosen Daten *(junk data)* maskiert werden. Die Länge der Paketanhänge wird durch einen weiteren Generator bestimmt. Daneben muss man jedoch die steigenden Bandbreitenanforderungen beachten und deshalb die Anhangslänge der Paketlänge wesentlich unterwerfen.

---

[21] MD5 verarbeitet 0.9 Mbytes/s

[22] Differenz aus der Sequenznummer des letzten gültigen Pakets und der aktuell zu prüfenden

#### 4.1.3.2 Reverse Engineering

Das Problem bei allen Checksummen-Verfahren ist, dass die Client-Applikation den Berechnungsalgorithmus enthält. Dadurch ist es immer möglich, unter Zuhilfenahme von Reverse-Engineering-Methoden, den Algorithmus zu ermitteln und damit gültige Checksummen für jedes beliebige Paket zu berechnen. Eine Lösung ergibt sich daraus das der Aufwand für den Hacker in der Regel viel zu groß ist und es sich somit für ihn nicht lohnt. Einige Ansätze um den Reverse-Engineering-Aufwand zu erhöhen sind:

- Entfernung aller Debug-Informationen aus jeglichen Veröffentlichungen
- Funktionen zur Datensicherung nicht isolieren sondern mit dem Netzwerkcode kombinieren
- Initiale Zufallszahlen oder ähnliches erst zur Laufzeit berechnen und nicht in den ausführbaren Dateien festlegen

#### 4.1.3.3 Verschlüsselter Datenverkehr

Die Verschlüsselung der Nutzerdaten innerhalb der Netzwerkpakete ist eine weitere, zusätzliche Möglichkeit zur Erhöhung der Sicherheit während der Übertragung. Angelehnt an die Erkenntnisse aus IPSec (*Internet Protocoll Security)* [RFC2401] wird in [Treg02] unter anderem die Verwendung des symmetrischen[23] Verschlüsselungs-Algorithmus **DES** [RFC2405] mit 64Bit Schlüsseln vorgeschlagen. Die Performance des DES ist ausreichend für Echtzeitspiele, lediglich die durch das Verschlüsselungsverfahren bedingten zusätzlichen Informationen pro Paket, sprich die nicht irrelevant steigende Bandbreitenanforderung[24] muss beachtet werden. So das, wie bei jedem Paket-Overhead größere, seltener gesendete Pakete die verfügbare Bandbreite effizienter ausnutzen. In Kombination mit häufigen Schlüsselwechseln wird trotzdem ein guter Kompromiss aus Sicherheit, Performance und Bandbreitenanforderung erreicht.

Durch die Verwendung eines zweiten Paars Kryptographiescher-Schlüssel kann man auch das oben beschriebene Szenario der Hash-Checksummen vereinfachen und sicherer gestalten. Indem abgeschnittene, Hash-basierte Nachrichten-Authentifikations-Codes *(HMAC)* [RFC2104], als authentische Checksummen eingesetzt werden. **HMAC** werden auch durch einen Fingerabdruck einer Hashfunktion *(z.B. MD5)* über ein Paket gebildet, allerdings wird zusätzlich ein geheimer symmetrischer Schlüssel einbezogen. Der Empfänger kann wiederum

[23] gleicher Schlüssel zum chiffrieren und dechiffrieren
[24] im Implementierungsbeispiel aus [Treg02] sind das 21-36 Bytes Overhead, zusätzlich zum UDP-Header

etwaige Manipulationen am Paket erkennen und gleichzeitig wird eine Reverse-Engineering-Attacke nutzlos, da der Hacker auf diese Weise nicht an die nötigen Schlüssel kommt und keine eigenen gültigen HMAC, selbst unter Kenntnis des verwendeten Algorithmus, erzeugen kann. In der Folge werden auch Paket-Replay-Attacken durch normale unverschlüsselte Sequenznummern und eine Sliding-Window-Technik nutzlos. So wird zunächst der HMAC eines Paketes berechnet und mit dem übertragenen verglichen. Ist ein Paket validiert und befindet sich seine empfangene Paketnummer innerhalb des Fensters, wird sie in einer Liste der bereits empfangenen Pakete innerhalb des Fensters *(window)* gesucht. Falls sie nicht gefunden wird ist sie gültig und es ist kein Replay-Paket. Liegt die Nummer rechts außerhalb des Fensters wird dieses verschoben, liegt sie links außerhalb wird das Paket verworfen. Dementsprechend werden auch verspätete Pakete unterstützt. Dem Hacker verbleibt schließlich nur ein direkter Angriff[25] auf die verschlüsselten Daten, um die Schlüssel zu berechnen und dann gültige Pakete zu erzeugen. Deshalb sollten die Schlüssel regelmäßig gewechselt werden, z.B. bei jedem Serverlogin. Des Weiteren sollten die Performance unkritischen Schlüsselaustauschverfahren [Schnei96] eine höhere Sicherheitsstufe, in der Regel längere Schlüssel, aufweisen.
Aufgrund der Sicherheitsvorteile der krytologischen Verfahren werden sich diese wohl in der Zukunft auch im Bereich der Mehrbenutzerspiele durchsetzen.

### 4.1.4 Ephemeral Channels

Als generelle Regel nimmt man an, dass 70% der zur Verfügung stehenden Bandbreite für die Übertragung von Spielerpositionen verwendet werden [Thor03]. Der Client währe in der Lage Positons-Updates pro angezeigtes Bild zu generieren. Dies würde allerdings bereits bei einer minimalen Framerate von 15, konventionellen 20 Bytes pro Update und 30 Spielern, die sich alle gegenseitig sehen, zu einer Datenrate von 9kB pro Sekunde führen. Womit man die theoretisch erreichbaren Fähigkeiten eines 56kbps Modems gesprengt hätte. Zur Überprüfung, ob eine Bewegung legal ist, benötigt der Server allerdings nicht eine solch hohe Refresh-Rate. Deshalb wird bei Spielen mit **hohen Spieleranzahlen** bei denen die Anforderungen an die Aktualität der Positionsdaten nicht Spiel bestimmend sind, ein so genanntes Ephemeral-Channel-Konzept eingesetzt. Der Client generiert zwar weiterhin Framerate-gekoppelte Updates, allerdings wird jeweils nur das Aktuellste innerhalb eines längeren Update-Zyklus versendet. Der Server überprüft dann den Bewegungsabschnitt zwischen der letzt

[25] Brute-Force-Angriff auf einen normalen DES-Code dauerte auf einem Spezialrechner 1998 56 Stunden

empfangenen und der aktuellen Position an einigen Stellen *(lineare Interpolation)*. Zu beachten ist, dass die verwendete Abtastrate kleiner als das kleinste Kollisions- oder Schalter-Volumen sein muss, damit der Spieler nicht irgendwo „durchschlüpfen“ kann. In *First Person Shooter* Spielen wäre dieses Konzept nur in Verbindung mit kurzen Update-Zyklen einsetzbar und verfehlt damit sein eigentliches Ziel, die Erhöhung der maximal zulässigen Spielerzahlen.

## 4.2 Techniken der Client-Applikation

In den nächsten Abschnitten werden Verfahren vorgestellt, die zu dem Bereich der Abschätzung zukünftiger Bewegungen, der so genannten *Movement Prediction*, gehören.

### 4.2.1 Methoden der Interpolation und Extraplolation

Mit Hilfe dieser beider Verfahren wird versucht, die Auswirkungen größerer Latenz zu kompensieren. **Extrapolation** wird angewendet, wenn kein aktuelles Positions-Update eines von Opponenten *(Clients)* oder vom Server kontrollierten Objektes *(Ghost)* vorliegt, um die wahrscheinlich aktuelle Position des Objektes abzuschätzen. Abbildung 4.4 zeigt eine Situation in der ab dem Zeitpunkt $t_0$ keine neuen Positions-Updates mehr eintreffen. Der Client extrapoliert auf Basis der letzten Position, Geschwindigkeit und Beschleunigung *(p, v, a)* des *Ghost* dessen wahrscheinlichste Position. Da das eigentliche Objekt aber nach dem Zeitpunkt $t_0$ einen Richtwechsel durchführt, divergieren die Simulationen. Zum Zeitpunkt $t_1$ trifft ein aktuelles Update ein woraus sich zwei Möglichkeiten ergeben. Die einfachste ist den *Ghost* sofort an seine tatsächliche Position zu setzen, diese Methode setzt man bei sehr niedrigen und extrem hohen Latenzen ein oder wenn der im Folgenden berechnete Pfad blockiert ist. Die zweite Möglichkeit ist, mit Hilfe der aktuellen Zustands-Informationen *(p, v, a)* eine zukünftige Position zu extrapolieren und diese dann innerhalb eines Zeitintervalls, durch **lineare Interpolation** [Treg00] von der momentanen Position aus, zu erreichen. In einigen Fällen wird die Inter- oder Extrapolationszeit begrenzt, in QuakeWorld z.B. auf 100ms, um die evtl. entstehenden Fehler zu begrenzen.

In einem *First Person Shooter* tendieren die Teilnehmer dazu, durch ständiges hüpfen das Zielen für ihre Opponenten zu erschweren. Dies führt, bei der Verwendung der obigen Methode, eventuell zu einer schwebenden Bewegung der Spieler *(siehe Pfeil in Abbildung 4.5)*. Aus diesem Grund wird häufig entlang einer Kurve interpoliert anstatt linear. So

verwendet *Half-Life* einige der letzten Updates zur Generierung einer solchen Kurve, mit deren Hilfe eine springende Bewegung interpoliert werden kann. Abbildung 4.5 zeigt eine Version, in der ein Catmull-Rom-Spline [Treg00] zwischen den letzten beiden bekannten, wegen dem Ausfall der Updates ab $t_0$, Positionen des Objekts der Client-Simulation und den zwei extrapolierten Positionen, auf Basis des Update $t_1$, als Interpolationspfad verwendet wird. Spline-Kurven mit noch mehr Freiheitsgraden kommen dem Absprungspunkt immer näher.

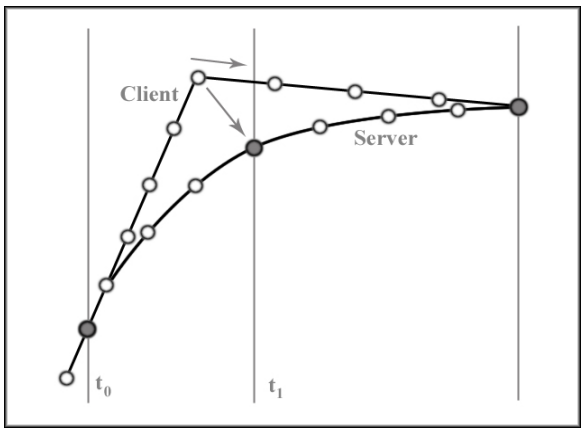


**Abbildung 4.4**
Überblendung der divergierten Client-Sicht in die der Server-Simulation durch Interpolation [Thor03].

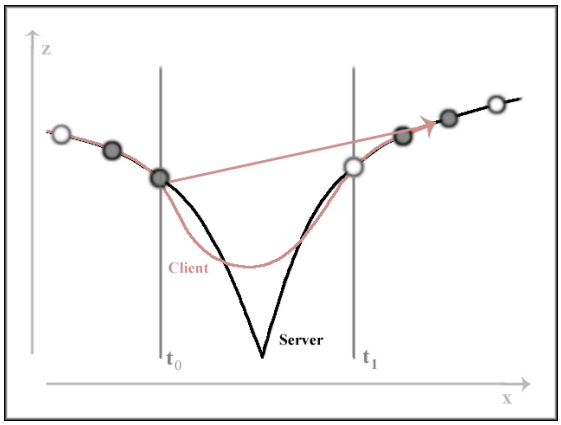


**Abbildung 4.5**
Catmull-Rom-Spline Interpolation für kontinuierlich springende Objekte [Thor03].

### 4.2.2 Reversible Simulationen

Ein Problem das sich beim Studium des obigen Abschnitts aufdrängt ist das so genannte **Aiming-Lag**. Wenn die Client-Simulation die Position eines *Ghost's* aufgrund von Latenz extrapolieren muss werden etwaige Treffer eventuell vom Server zurückgewiesen, weil das Objekt sich in Wirklichkeit an einer anderen Position befindet. Dies ist natürlich frustrierend für den Spieler und eine Möglichkeit dem zu begegnen sind reversible Simulationen, eingesetzt in *Half-Life.* Theoretisch würde es genügen sich auf die Verteilung des Ereignisses *„habe Objekt XY getroffen"* durch den Client, zu verlassen. Wegen des dadurch entstehenden Sicherheitsrisikos muss jedoch zusätzlich eine Server-Kontrolle stattfinden *(„Never trust the client").* Zur Durchführung dieser Kontrolle versetzt sich der Server in den, in der Vergangenheit liegenden, Spielzustand des Clients, um dessen Entscheidungen zu verifizieren. Der Server erreicht dies indem er in seiner *Update-History*[26] das letzte, vor dem

[26] Liste der gespeicherten und an den Client versendeten Zustands-Updates

Ereignis versendete und vom Client empfangene Zustands-Update sucht. Auf dessen Basis werden dann alle anderen Objekte auf einen ihrer vergangenen Zustände zurückversetzt. Dann extrapoliert der Server die Position des *Ghost's* auf dem Client zum Zeitpunkt des Ereignisses und fällt die Treffer-Entscheidung. Dieses Verfahren erfordert synchronisierte Uhren zwischen Server und Client, um anhand der Zeitstempel zu entscheiden auf welchem der Server-Updates die momentane Clientsicht basiert [Treg02]. Ein Nachteil dieser Methode ist, dass die Entscheidung des Servers evtl. den Client der das eigentliche Objekt steuert benachteiligt. Wenn z.B. seine gezielt ausweichende Bewegung mit einer Netzwerkverzögerung zusammenfällt, wird er evtl. trotzdem getroffen! In Mehrbenutzerspielen wie *First Person Shooter*, die durch kurze Respawn-Zeiten[27] gekennzeichnet sind, in denen somit das Ausscheiden den Spieler nicht viel „kostet", fördert diese Methode letztendlich dennoch den Spielspaß!

### 4.2.3 Dead-Reckoning-Algorithmen

Die durchschnittliche, voraussetzbare Qualität der Anbindungen an das Internet, wird zwar in Zukunft wachsen, allerdings wird niemals genug Bandbreite zur Verfügung stehen, um komplette Spielzustands-Updates zu übertragen. Deshalb muss man sich stets mit der Reduzierung der Informationen in den Updates und ihrer Anzahl beschäftigen. Im ersten Fall bieten sich Standardverfahren zur **Datenkomprimierung**, wie Reduktion der Genauigkeit der versendeten Daten oder, aus dem Bereich der Video Komprimierung, das Übertragen von Differenzen anstatt absoluter Werte. Leider lassen sich diese Methoden nicht für alle Spiele sinnvoll anwenden. In *First Person Shooter* Spielen wird in einigen Fällen *(5.1.1.3)* die Komponentenlänge[28] der Positionsvektoren beschnitten, allerdings werden die Positions-Updates mit Hilfe unzuverlässiger Protokolle übertragen, sodass das Versenden von Differenzen zu bereits übertragenen Werten nicht einsetzbar ist. Zur Kompensation des **Durchsatz-Konsistenz-Tradeoff** verbleiben also im Wesentlichen Methoden zur Verringerung der Anzahl bzw. Häufigkeit der versendeten Nachrichten.

Viele Mehrbenutzerspiele verwenden eine konstante Rate von Zustands-Updates zu ihren Clients, welche zwischen 4 und 20 Updates pro Sekunde liegt. Half-Life sendet beispielsweise 20 Updates pro Sekunde zu jedem Client. Ein Verfahren das diese Rate der jeweiligen Spielsituation anpasst, somit eine Methode des zweiten Falls, ist **Dead-Reckoning**,

[27] Zeitintervall vom Ausscheiden der Spielerfigur *(fraggen)* bis zum Wiedereintritt bzw. Fortsetzen des Spiels
[28] Einschränkung des Wertebereichs *(Bitlänge)* der Koordinaten *(x, y, z)* eines Vektors

ursprünglich ein Protokoll aus DIS *(Distributed Simulator Networking, IEEE Standard 1278)*. Der Dead-Reckoning-Algorithmus übernimmt die Steuerung der Ghost's *(4.2.1)* auf dem Client, in dem er auf Basis der letzten übertragenen Zustands-Informationen *(p, v, a)* zukünftige Positionen durch Extrapolations- und Interpolations-Techniken *(4.2.1)* berechnet. Zum Erhalt der Synchronität der Simulationen, ermittelt jeder der Clients zusätzlich durch den Dead-Reckoning-Algorithmus die Differenz zwischen der extrapolierten und realen Position, der von ihm kontrollierten Objekte. Wenn diese einen Grenzwert überschreitet, müssen die aktuellen Spieler-Informationen neu versenden werden. Der Erfolg der Methode hängt vom Verhältnis der Informations-Änderungen zur Frequenz der Informations-Übertragungen ab und ist deshalb nicht für jedes Spiel-Genre geeignet. Insbesondere *First Person Shooter* sind durch, im Verhältnis zu Vehikle-Simulationen, langsame Spieler-Bewegungen und häufige Richtungswechsel gekennzeichnet und damit weniger für diesen Algorithmus prädestiniert.

Eine Vereinfachung der Methode die nur noch auf Positions-Updates basiert und damit weniger anfällig gegen Geschwindigkeits und Beschleunigungsänderungen ist, wurde von Singhal und Cheriton [SiCh94] vorgestellt *(Abbildung 4.6)*. Ähnlich dem Dead-Reckoning-Algorithmus werden Positions-Updates übertragen, wenn die Differenz zur abgeschätzten Position einen Grenzwert überschreitet. Zur Extrapolation wird eine Kurve *(Tracking curve)* durch die letzten 3 empfangenen Positionen gelegt, je nach Winkel $\theta$ zwischen diesen Positionen werden Polynome 1. oder 2. Grades für die Kurve verwendet *(Abbildung 4.7)*. Durch den Verlauf der Kurve erhält man dann Abschätzungen über zukünftige Spielerpositionen. Nach dem Empfang einer aktuellen Position wird die Tracking-Kurve angepasst und eine Konvergenz-Kurve abgeleitet. Welche die Position des Ghost fließend auf die Tracking-Kurve zurückführt, innerhalb der Update freien Periode $\Delta t$. Die Schätzungen dieser Art des Dead-Reckoning liegen auch bei einem *First Person Shooter* häufiger richtig und sind bei dieser Art von Spielen zu bevorzugen.

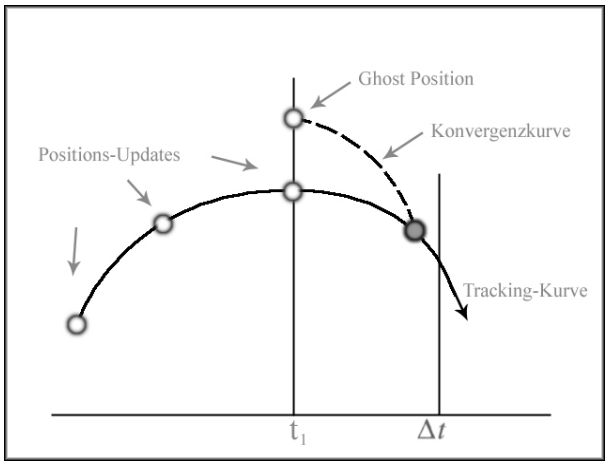


**Abbildung 4.6**
Auf Positions-Updates basierende Version des Dead-Reckoning-Algorithmus [Watt01].

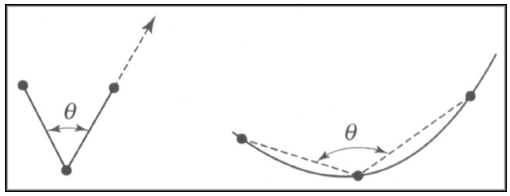


**Abbildung 4.7**
Bestimmung des Grads der Tracking-Kurve [Watt01].

### 4.2.4 Ansätze der Vorhersage-Techniken

Bei der Verwendung von Vorhersage- bzw. Abschätzungs-Techniken (*Movement Prediction)* muss man sich zweier Zielsetzungen bewusst sein. Einerseits werden Probleme durch verspätete, ausgefallene Zustands-Updates kompensiert, indem die Steuerung der Ghost's *(zeitweise)* durch diese Algorithmen übernommen wird. Andererseits wird jede, vom Server unbestätigte, Spielerbewegung des Clients als vorausgesagt betrachtet, da der Server jederzeit korrigierend eingreifen kann *(5.1.1.4)*. Wodurch die Anforderungen an die Bandbreite gesengt werden.

### 4.2.5 Event-Locking

Im Rahmen von Echtzeitspielen möchte ich am Rande noch auf ein Konzept für RTS-Spiele eingehen, da in den neueren Spiele-Genre-Mixen auch eine Anwendung in FPS-Spielen denkbar ist. Die Bewegung einer Menge von Objekten zu einem vorgegebenen Ziel, mit Hilfe von Pfadfindungs-Algorithmen, beschreibt das Szenario für dieses Konzept. Anstatt nun wiederholt die Positionen der Objekte entlang des berechneten Pfades zu versenden, werden lediglich der Zielort und die Ankunftszeit übertragen. Synchronisierte Uhren innerhalb des verteilten Systems vorausgesetzt [Treg02], kann nun durch Anpassung der Geschwindigkeiten der Objekte auf jedem der Clients, etwaige auftretende Latenz kompensiert werden. Im Beispiel aus Abbildung 4.8 erhält der Server nach 200ms *(2)* die Bewegungsanfrage für den Panzer und startet die geprüft legale Bewegung mit 10m/s. Nach weiteren 200ms *(3)* haben alle Clients die Vorgaben des Servers, also die Ankunftszeit und den Ort für den Panzer in der

Serversimulation, erhalten und passen die Geschwindigkeiten ihrer Panzer an. Dadurch wird erreicht, dass letztlich alle Panzer zur selben Zeit am selben Ort ankommen und somit die Konsistenz wieder hergestellt ist.

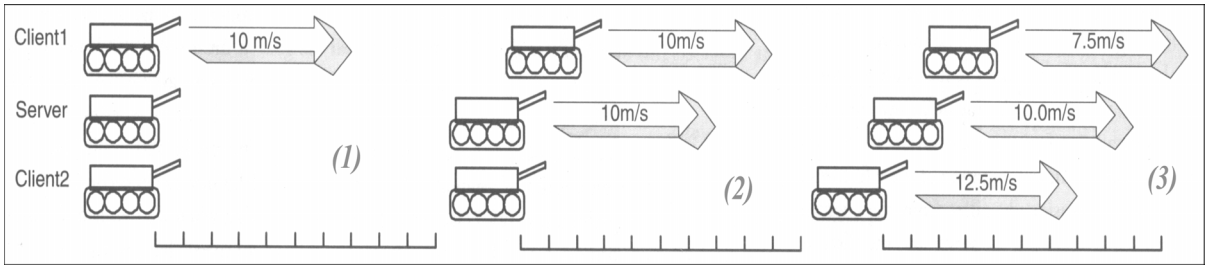


**Abbildung 4.8**

Momentaufnahmen alle 200ms, einer vom Client Nr. 1 initiierten Bewegung eines Panzers [Thor03].

# Implementierungen in Echtzeitspielen 5

Im Anschluss, an die im vorherigen Kapitel vorgestellten Konzepte zur Lösung von auftretenden Problemen bei der Internetkommunikation, sollen nun einige Realisierungen aus aktuellen Mehrbenutzer-Echtzeitspielen beispielhaft vorgestellt werden.

## 5.1 *Unreal Tournament - Epic MegaGames*

### 5.1.1 Reduktion der Bandbreitenanforderung

#### 5.1.1.1 Relevante Aktoren

*Unreal* bezeichnet als Aktoren alle Objekte mit der Fähigkeit sich unabhängig zu bewegen und mit anderen Aktoren zu interagieren. Die Menge der aktiven Aktoren in einem Level beschreibt den jeweiligen Spielzustand. Durch die Bildung einer relevanten Menge von Aktoren entscheidet der Server welche der Aktoren für einen Client sichtbar oder ihn eventuell beeinflussen können. Unreal führt somit eine Art der Objektbasierten Filterung *(4.1.2)* durch. Jeder der Clients wird angewiesen seine Menge von Aktoren zu kopieren und zu erhalten *(5.1.2)*. Ein Aktor ist für den *Unreal*-Client unter anderem dann relevant, wenn:

- ein Sichtbarkeitstests zwischen Aktor und Spieler, den Aktor als sichtbar ausweist
- der Aktor vor 2-10s *(je nach Performance-Einstellung)* sichtbar war
- der Aktor ein vom Spieler kontrolliertes[29] Objekt ist
- der Aktor eine, von einem sichtbaren Spieler, geführte Waffe ist

Für eine komplette, detaillierte und insbesondere spielabhängige Auflistung der Regeln zur Bestimmung der relevanten Aktoren verweise ich auf [Sweeny99].

#### 5.1.1.2 Aktoren verschiedener Priorität

*Unreal* verwendet zusätzlich eine Last-Verteilungs-Technik *(load-balancing)* unter den Aktoren, um die zur Verfügung stehende Bandbreite, entsprechend ihrer Priorität für das

[29] der Spieler ist der Besitzer des Objektes

Gameplay, zu verteilen. Jeder der Aktoren besitzt ein *NetPriority*-Attribut dessen Wert den Anteil zu verwendender Bandbreite, in Relation zu den anderen, angibt. Einige Beispielwerte für *NetPriority*:

▪ Bots: 8.0 ▪ Movers: 7.0 ▪ Projektile: 6.0 ▪ Dekorative Kreaturen: 2.0

#### 5.1.1.3 Einschränkung des Wertebereichs

Zur Erhöhung der Bandbreiten-Effizienz beschränkt *Unreal* optional den Wertebereich der Positionsvektoren und den der Werte für die Rotation. Die Komponenten der Positionsvektoren werden auf 16-Bit Integer *(-32768..32767)* und die Rotationskomponenten *(pitch, yaw, roll)* werden auf 8-Bit *(0..255)* beschränkt.

#### 5.1.1.4 Abschätzung von Spielerpositionen

Der Client beschreibt seine Bewegung als einen 3D Beschleunigungsvektor *(Acceleration)* und sendet diesen zusammen mit einem Zeitstempel *(Timestamp)* als replizierten Funktionsaufruf *(5.1.2)(5.1.2.1) ServerMove()*[30] an den Server:

```
function ServerMove
(
    float       TimeStamp,
    float       AccelX,         // absolute Beschleunigung
    float       AccelY,
    float       AccelZ,
    float       LocX,           // absolute Position
    float       LocY,
    float       LocZ,
                                // auf die Eingaben bezogene Informationen
    byte        MoveFlags,      // Art der Bewegung geduckt…
    eDodgeDir   DodgeMove,      // Richtung einer Spezial-Bewegung (dodging)
    rotator     Rot,            // evtl. rotierende Bewegung
    int         ViewPitch,      // Blickrichtungswinkel
    int         ViewYaw
);
```

Unmittelbar danach führt der Client die Bewegung lokal aus *(MoveAutonomous())* und speichert die Aktion in einer Liste *(History)*. Solange der Server kein Veto einlegt, erreicht *Unreal* somit ein Verzögerungsfreie Client-Bewegung.

Wenn der Server den replizierten Funktionssaufruf empfängt, bestimmt er mit Hilfe des aktuellen und des letzten empfangenen Zeitstempels ein $\Delta t$. Mit den empfangenen Parametern und $\Delta t$ wird die aktuelle Position des Clients abgeschätzt. Woraus sich eine leicht divergierte Server-Sicht ergeben kann.

[30] Sämtliche Kommentare sind Schlussfolgerungen bzw. Anmerkungen und entstammen nicht aus den veröffentlichten Quellen von *Epic Megagames.*

Durch einen Aufruf der replizierten Funktion *ClientAdjustPosition()* kann der Server seine Kontrollfunktion wahrnehmen, falls dies erforderlich ist.

```
function ClientAdjustPosition
(
    float      TimeStamp,
    name       newState,
    EPhysics   newPhysics,
    float      NewLocX,
    float      NewLocY,
    float      NewLocZ,
    float      NewVelX,
    float      NewVelY,
    float      NewVelZ
);
```

Empfängt der Client einen solchen Aufruf justiert er seine Position und Geschwindigkeit und löscht alle Einträge in seiner Bewegungs-History die vor dem empfangenen Zeitstempel liegen. Da die vom Server bestimmte Position aber bereits wieder in der Vergangenheit liegt und somit veraltet ist, durchläuft der Client seine *History* und ruft mit jedem nachfolgendem Eintrag *(MoveAutonomous())* auf. Der Client läuft den Server-Anweisungen somit immer eine Zeitlang voraus und zwar genau so weit wie die Hälfte seiner Ping-Zeit, woraus eine fließende Client-Bewegung resultiert.

### 5.1.2 Konsistenzerhaltung durch Replikation

*Unreal* fasst das Problem der Konsistenzerhaltung einer verteilten Simulation der virtuellen Realität als ein Problem der Replikation des Spielzustands zwischen Server und Client auf. Auf Basis der replizierten Daten werden drei Arten der Replikation unterschieden:

- **Aktoren-Replikation**. Der Server erstellt die Menge der relevanten Aktoren für den Client und weist diesen an, sie zu replizieren.
- **Variablen-Replikation**. Einzelne Aktorenvariablen können vom Client repliziert werden, wenn sie für ihn Relevants besitzen.
- **Function-Call-Replikation**. Funktionen die auf der Server- oder Client-Seite aufgerufen werden, können zur gegenüberliegenden Seite versendet und ausgeführt werden. Nach dem Absetzen eines Funktionsaufrufs fährt der Absender mit der Ausführung fort ohne auf eine etwaige Antwort zu warten.

#### 5.1.2.1 Unzuverlässige/Zuverlässige Replikation

Die Spiellogik wurde in Unreal komplett von dem Netzwerkcode getrennt, indem der Spielzustand durch eine erweiterbare, Objektorientierte Script-Sprache *(UnrealScript)* beschrieben wird. In *UnrealScript* wird damit auch wahlweise festgelegt, ob eine Übertragung bzw. eine Replikation zuverlässig *(Anweisung „reliable if(...)“)* oder unzuverlässig *(„unreliable if(...)“)* erfolgen soll.

```
replication
{
        …
        // Bsp. einer Variablen-Replikation
        reliable if( Role==ROLE_Authority )
                Weapon;

        // Bsp. einer Funktions-Replikation
        unreliable if( Role==ROLE_Authority )
                ClientHearSound, ClientMessage;
        …
}
```

Zusätzlich zur Art der Replikation, wird in der Anweisung auch noch die Bedingung *( if(...) )* für die Replikation festgelegt. Jeder Aktor enthält eine Variable[31] *(Role)*, welche die Kontrolle, die der lokale[32] Rechner über den Aktor besitzt, angibt. Die Auswertung dieser Variablen stellt die wichtigste Bedingung für die eventuelle Replikation dar. Die relevanten Aktoren-Rollen in *Unreal* sind:

- ***ROLE_ DumbProxy***. Der Aktor ist ein temporäres, approximiertes Proxy-Objekt auf dem Client, welches keine physikalischen Simulationen durchführt und nur durch neue, vom Server replizierte Informationen verändert wird.
- ***ROLE_ SimulatedProxy***. Der Aktor ist ein temporäres, approximiertes Proxy-Objekt auf dem Client, welches physikalische Simulationen wie Gravitationsbewegungen, Kollisionen oder ähnliches durchführen kann.
- ***ROLE_ AutonomousProxy***. Der Aktor ist der lokale Spieler auf dem Client. Spezielle

[31] die Variable *(RemoteRole)* gibt die dazu inverse Kontrolle der anderen Teilnehmer über den Aktor wieder
[32] der das Script ausführende Server oder Client

Voraussage-Logik wird anstelle der Simulations-Logik angewendet.

- ***ROLE_ Authority***. Der lokale Rechner ist der Server und besitzt die absolute, autoritative Kontrolle über den Aktor.

Neben der Rolle des Aktors kann der *UnrealScript*-Programmierer weitere Bedingungen durch eine Verknüpfung mit Boolschen-Operatoren hinzufügen, um die Notwendigkeit der Replikation zu bestimmen.
Variablen werden in *Unreal* immer zuverlässig und nur dann übertragen, wenn sich ihr Wert seit dem letzten Update geändert hat und genügen Bandbreite zur Verfügung steht. Replizierte Funktionen hingegen, werden auf beide Weisen, immer und unmittelbar übertragen. *Unreal* verwendet zur Übertragung der Daten über das Internet standardmäßig UDP und des weiteren eigene Erweiterungen zur Unterstützung von zuverlässigen Übertragungen [Sweeny99].

## 5.2 *Quake – id Software*

### 5.2.1 Filterung relevanter Informationen

*Quake* versendet Updates über Objektzustände nur dann an Clients, wenn diese im Sichtbereich des Spielers liegen. Dieser Sichtbereich wird durch einen BSP-Baum[33] bestimmt, welcher den virtuellen Raum durch 3 Ebenen in 8 Regionen *(Octree),* pro Iteration unterteilt *(Abbildung 5.1)*. So wird z.B. ein Raum einem Blatt zugewiesen und der relevante Bereich ergibt sich aus diesem und allen von ihm aus sichtbaren, adjazenten Räumen. Diese Liste der sichtbaren Räume wird vorberechnet, so dass der Server zur Laufzeit schnell entscheiden kann, welche Blätter des Baumes zusätzlich zum aktuellen zur Bestimmung des Sichtbereichs in frage kommen. Diese Methode gehört zu einer Art der zuvor vorgestellten Rasterbasierten Filterung *(4.1.2.1)*. Raumunterteilung mit Hilfe von BSP-Bäumen ist ein sehr komplexes Thema und kann hier nur am Rande erwähnt werden. Weitergehende Informationen über *Quakes* BSP-Bäume findet man z.B. in [Abrash00].

[33] *Binary Space Partition*

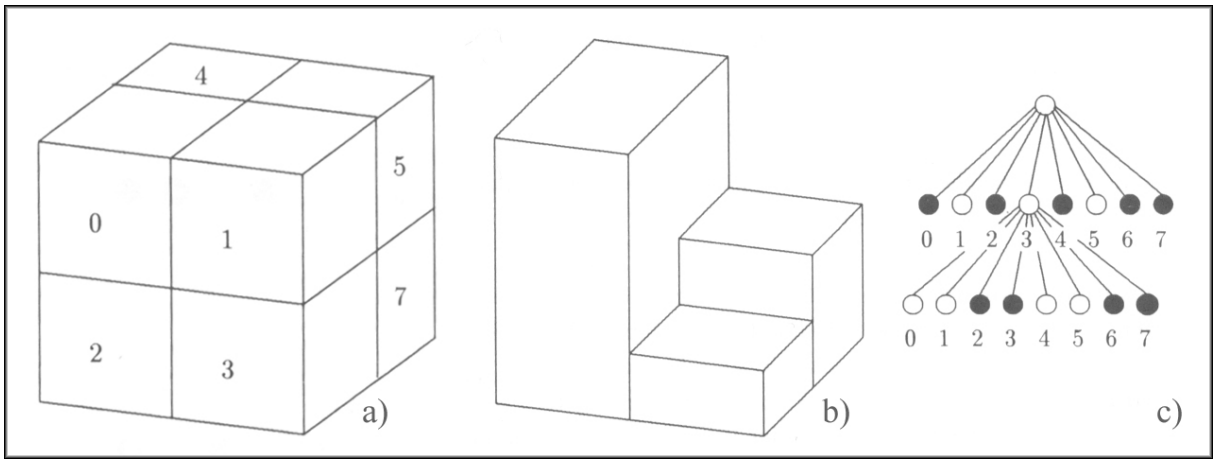


**Abbildung 5.1**

a) Numerierung der einzelnen Oktanten des Octrees

b) Einige Beispielobjekte innerhalb des virtuellen Raumes

c) Octree-Darstellung entsprechend der Numerierungsregeln, dabei bedeutet ein schw.-Blatt das der zugehörige Raum komplett von einem Objekt*(-teil)* ausgefüllt wird [Fellner92].

### 5.2.2 Das Netzwerkprotokoll von *Quake I*

*Quake* verwendet als grundlegendes Netzwerkprotokoll ausschließlich UDP, um zusätzlich verlässliche Übertragungen zu ermöglichen werden eigene Paketnummern, Bestätigungen und *Retransmits* eingesetzt.

Alle Pakete beginnen mit einem 4-Byte Header, der aus dem Pakettyp *(2 Byte)* und der Paketlänge *(inkl. Header)* zusammengesetzt ist. Die verwendeten **Spielpakete** bestehen aus einer 4-Byte Paketnummer gefolgt von der Paketlänge und dem anschließenden Datenbereich *(message block)*. *Quake* unterscheidet zwei Paketarten hinsichtlich des Übertragungstyps.

**Zuverlässige Pakete:**

```
short       packet_type       0x0001 or 0x0009 – Pakettyp   (2 Byte)
short       packet_len        Paketlänge inklusive Header
long        packet_number     Paketnummer                   (4 Byte)
mesgblk     message_block     Zu versendende Daten (Byte orientiert)
```

Die entsprechenden Bestätigungspakete *(acknowledgements)* sind 8-Byte groß und enthalten die Paketnummer des letzt empfangenen Paketes:

```
short       packet_type       0x0002 - Acknowledgements
short       packet_len        Paketlänge inklusive Header
long        packet_number     Nummer des zuletzt empfangenen Pakets
```

Sie werden zusammen mit anderen ausgehenden Nachrichten *(Huckepack)* versendet.
**Unzuverlässige Pakete** sind vom Aufbau identisch zu ihren zuverlässigen Pendants, bis auf die Typnummer *0x0010***.** Weitere Informationen über *Quake's* Netzwerkprotokoll findet man in [Kreim98].

### 5.2.3 Positions- und Input-Updates in *Quake II*

Als Beispiel für die so oft erwähnten Update-Nachrichten, soll hier der **Aufbau** der häufigsten Updates am Beispiel von *Quake 2* gezeigt werden.
Der Client sendet pro angezeigtem Bild eine Nachricht der 8-Byte großen *usercmd_t* Struktur, die unter anderem die aktuellen Benutzereingaben widerspiegelt:

```
typedef struct usercmd_s
{
      byte        msec;          // Zeit fürs letzte Frame*1000 [0-250]
      byte        buttons;                          // gedrückte Tasten
      short       angles[3];                        // Blickrichtung
      // Geschwindigkeiten
      short       forwardmove, sidemove, upmove;    // relative Bewegung
      // 3D-Engine spezifische Daten
      byte        impulse;
      byte        lightlevel;    // Ambientes-Licht am Spielerstandpunkt
} usercmd_t;
```

Die Struktur, der vom Server versendeten **Spielerpositions-Updates**, heißt in *Quake 2* *pmove_state_t* und ist wie folgt definiert:

```
typedef struct
{
      pmtype_t    pm_type;          // Typ des Spielers, für die Vorhersage
                                    // (Normal, Zuschauer…)
      short       origin[3];        // Positionsvektor
      short       velocity[3];      // Geschwindigkeitsvektor
      byte        pm_flags;         // Bewegungsart (geduckt, springend…)
      byte        pm_time;          // mehrere Bedeutungen, pro Einheit 8 ms
      short       gravity;
      short       delta_angles[3];  // Bestimmung der Blickrichtung, ver-
                                    // ändert von rot. Objekten, Teleporter
}pmove_state_t;
```

Diese 11-Byte große Struktur nutzt der Client nach dem Empfang zur Bestätigung oder Aktualisierung der Spielerbewegung.

Für weitere Nachforschungen ist der komplette Quellcode älterer *id*-Spiele, wie *Quake I und II*, online unter [ID03] verfügbar.

### 5.2.4 Verteilung der Paketgrößen einer Spielsession

Die im vorherigen Abschnitt beispielhaft vorgestellten Paketstrukturen kennzeichnen natürlich nur einen Teil der in *Quake II* versendeten Pakete. So dass mit Hilfe der in [Armit00] gesammelten Werte, ein Blick auf die Verteilung der auftretenden Paketgrößen während einer *Quake-II*–Session, geworfen werden soll.

#### 5.2.4.1 Client-to-Server

Abbildung 5.1 zeigt eine Verteilung, die durch eine 14 min Spielsession mit 5 Spielern und ca. 30000 Paketen pro Spieler erzeugt wurde. Die Paketgröße der von den Clients versendeten Informationen liegt zwischen 68 und 88 Bytes.

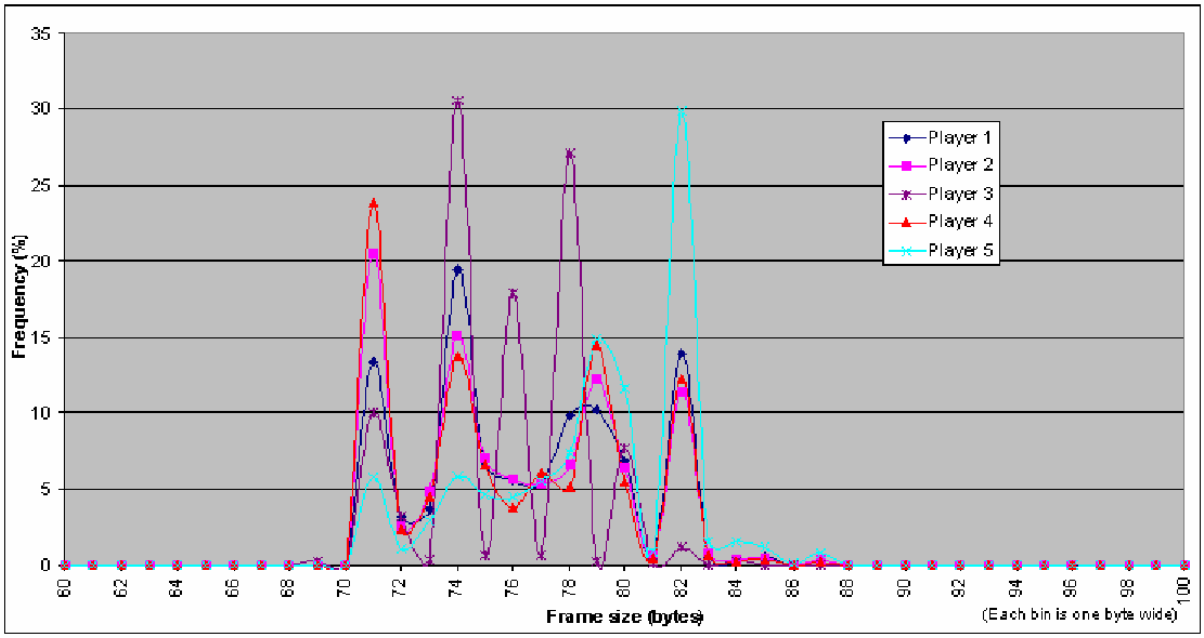


**Abbildung 5.1**

Verteilung der Paketgrößen währen einer Spielsession, von den Clients zum Server [Armit00].

*(Prozentsatz des Auftretens im Verhältnis zu der entsprechenden Paketgröße)*

### 5.2.4.2 Server-to-Client

Abbildung 5.2 zeigt eine Verteilung, die durch eine 11 min Spielsession mit 5 Spielern und ca. 7000 Paketen pro Spieler erzeugt wurde. Die Paketgröße der vom Server versendeten Informationen liegt zwischen 80 und 240 Bytes.

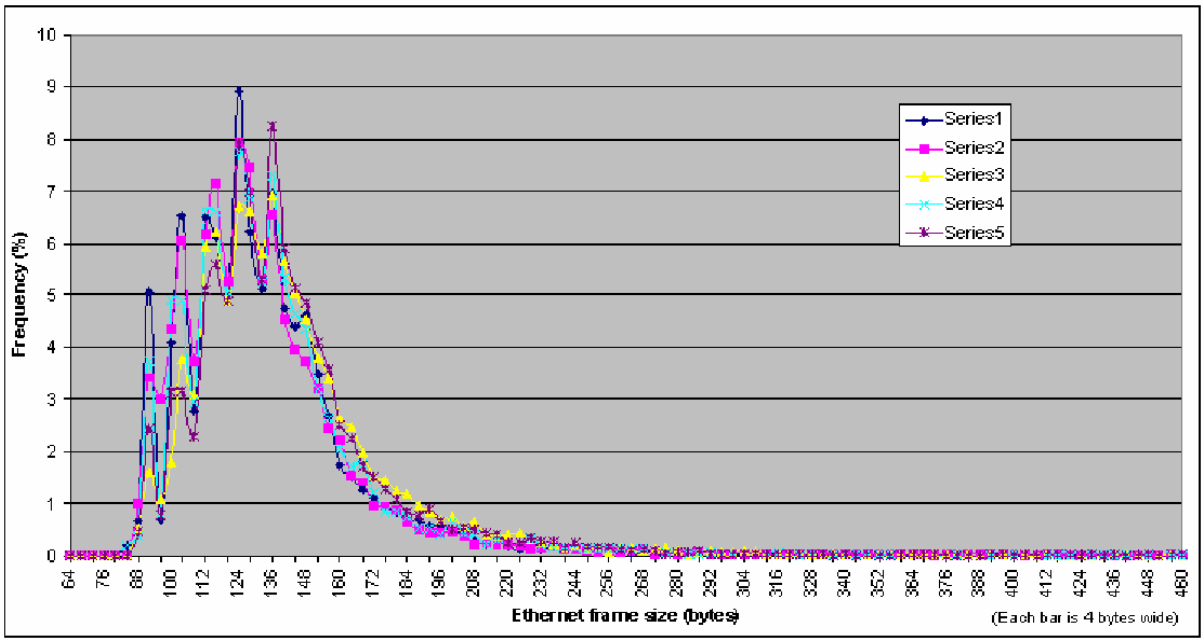


**Abbildung 5.2**

Verteilung der Paketgrößen währen einer Spielsession, vom Server zu den Clients [Armit00].

### 5.2.4.3 Bandbreitenanforderungen

Zur Gewinnung einer Vorstellung über die gesamte Client- und Server-Seitig angeforderte Datenrate von *Quake,* soll eine Messung aus [CFK01] vorgestellt werden. In dieser Arbeit wurde die Bandbreitenanforderungen von *Quake* bei der Unterstützung von 8 Clients gemessen. Dabei stellte sich heraus das die Clients jeweils eine Verbindung mit 50Kbps unterhielten und das der Server sogar eine Datenrate von 400Kbps der Verbindung abverlangte.

## 5.3 *Half-Life – Valve Software*

### 5.3.1 Abschätzung von Spielerpositionen

In *Half-Life* wird, anders als in *Unreal*, jedes Client-Update durch den Server bestätigt. Solch ein *Acknowledgement* zeigt an, welches Client-Kommando *(user input)* als letztes ausgeführt wurde und beinhaltet den daraus resultierenden, momentanen Spielerzustand auf dem Server. Die in *Half-Life* verwendete Struktur für diese Server-Updates heißt *entity_state_*s und ist in der *entity_state.h* Datei definiert [Valve03]. Die für das Client-Kommando verwendete *usercmd_t* Struktur unterscheidet sich nur unwesentlich von der bereits vorgestellten Struktur aus *Quake II*. Zur Vorhersage werden, ausgehend von der letzten Server-Bestätigung, alle Benutzer-Kommandos lokal ausgeführt und zusammen mit ihren Zeitstempeln in einer Liste gespeichert. Server und Client verwenden hierbei dieselbe Logik *(pm_shared.c)* um die Spielerposition zu bestimmen. Da der Client schneller Updates generiert[34], als aufgrund der Latenz Bestätigungen vom Server empfangen werden können, wird die Liste mehrfach durchlaufen. Erst wenn ein *Acknowledgement* für ein Update eintrifft, wird dieses aus der Liste entfernt *(Sliding-Window-Technik).* Damit keine Fehler bei der wiederholten Ausführung der Kommandos auftreten (*Sound-,Visuelle-Effekte),* werden Elemente wie z.B. Schrittgeräusche, als einmal ausgeführt markiert und beim erneuten durchlaufen nicht noch einmal aufgerufen [Bern01].

## 5.4 Zusammenfassung

In diesem Kapitel wurden 2 Realisierungen der Server-Applikations-Technik: Filterung relevanter Informationen vorgestellt. Im speziellen verwendet *Unreal* eine Objektbasierte Filterung, während *Quake* eine Art der Rasterbasierten Filterung zur Reduktion der Bandbreitenanforderung einsetzt. Weiterhin wurden zwei verschiedenen Umsetzungen von Vorhersage-Techniken, aus *Unreal* und *Half-Life*, zur Latenzkompensation präsentiert. Eine Bewertung der Implementierungen ist an dieser Stelle jedoch nicht möglich, da es sich einerseits, um 3D-Engine-Technologie handelt und andererseits zusätzliche Studien über das Spielerverhalten auf *Unreal-* und *Half-Life*-Servern, bei qualitativ unterschiedlichen

[34] gekoppelt an die jeweilige Framerate

Netzwerkbedingungen, durchgeführt werden müssten. Beide Fragestellungen sprengen den Rahmen dieser Arbeit.

Die zusammengetragenen Messwerte über die Bandbreitenanforderungen eines einfachen[35] FPS wie *Quake* zeigen, dass der kritische Punkt, durch die Anforderungen der Server-Applikation bestimmt ist. In mobilen Ad-Hoc Netzen müssen Client und Server mit der angebotenen Bandbreite auskommen, es gibt keine Möglichkeit einer speziellen, besseren Verbindung für die Server, wie z.B. im Internet. Hinzu kommt, dass mehrere Server pro Spielsession eingesetzt werden müssen, um eine ausreichen Fehlertoleranz zu gewährleisten. Die Clients werden dann zwar auf die Server verteilt, was auf eine sinkende Anforderung des einzelnen Servers schließen lässt, jedoch wird wieder zusätzlicher Verkehr durch die Server/Server-Kommunikation erzeugt. Mit Hilfe der aktuellen Server-Anforderungen kann man somit ein Schätzung für die, zum Betreiben von FPS nötige, Bandbreite in zukünftigen mobilen Ad-Hoc Netzen abgeben.

[35] Aufgrund komplexer werdender FPS, werden auch die Nutzer-Eingaben vielfältiger. Folglich müssen die Updates mehr Daten enthalten und werden mehr Bandbreite konsumieren.

# Mobile Ad-Hoc Netze 6

## 6.1 Zukünftige Probleme von Echtzeitspielen

Bei der Betrachtung der entstehenden Probleme und ihrer Lösungen wird angenommen, dass eine Weiterentwicklung einer Client/Server-Netzwerkarchitektur in den mobilen Ad-Hoc Netzen, eingesetzt wird.

### 6.1.1 Mobilität der Endgeräte

Aufgrund der mobilen Endgeräte entstehen diverse Probleme für die Spielapplikation. Zum einen können sich die kooperierenden Geräte *(7.1.2)* während eines Spiels soweit von einander entfernen, bis sie sich außerhalb ihrer Empfangsbereiche befinden. Da keine Infrastruktur vorhanden ist verbleiben nur zwei Reaktionen. Falls sich nur einige Spieler aus dem Netz entfernen muss der jeweils zuständige Server, aufgrund des Ausbleibens von Zustandsupdates der Clients, innerhalb eines Intervalls *(I > Latenz + Ausfallzeit)*, die Spieler aus der Session entfernen. Je nach Art des Spiels muss schließlich ab der Unterschreitung einer Mindestspieleranzahl die Session geschlossen werden.
Im Gegensatz zu einer endgültigen geographischen Entfernung kann auch ein temporärer Ausfall der Netzverbindung, z.B. wegen einer Abschattung durch Gebäude während der Bewegung, auftreten. Die so entstehende Ausfallzeit kann man mit einer sprunghaft ansteigenden Latenz vergleichen. Daraus folgt das beim Wiedereintritt in den Empfangsbereich, die Kommunikation, nach eventueller Reinitialisierung des Kommunikationskanals, fortgesetzt werden kann. Damit der Ausfall für den Spieler unbemerkt bleibt, bieten sich die vorgestellten Vorhersage-Techniken *(4.2)* zur Latenz-Kompensation an. Die Schwierigkeit liegt darin das sich diese Ausfallzeiten zu den Verzögerungszeiten des jeweiligen Netzes und der Architektur addieren, sodass der Latenz-Bereich [0...150ms] *(3.2),* in dem frustfreies Spielen möglich ist, aufgeteilt werden muss. Folglich lässt sich ein temporärer Ausfall um so besser kompensieren, je höher die Bandbreite in zukünftigen mobilen Ad-Hoc Netzen (7.1.2) ist. Denn je geringer die Verzögerung durch die Übertragung der Spieldaten, desto mehr Potential verbleibt in dem Latenzintervall zur

unbemerkten Kompensation etwaiger Ausfälle. Für Überlegungen hinsichtlich einer oberen Grenze für die Ausfallzeit, muss man zunächst einmal, zur Generalisierung der verschiedenen zukünftigen verfügbaren Bandbreiten in mobilen Ad-Hoc Netzen, von einem idealen Kommunikationskanal[36] ausgehen. Des Weiteren muss die Frequenz der temporären Ausfälle in Betracht gezogen werden. Ausfälle die in kurzen Intervallen auftreten, müssen unterhalb der 150ms Grenze bleiben, da sie wie eine fluktuierende Latenz auf das Spiel wirken und somit nur, wie bereits gezeigt, bis zu dieser Grenze sinnvoll kompensiert und der Spielspass erhalten werden kann. Wenn die Ausfälle in großen zeitlichen Abständen *(empirisches*[37] $\Delta t > 10min$*)* auftreten, kann trotz Ausfallzeiten von mehr als 1min, z.B. durch Funkstörungen, der Spielspass erhalten bleiben. Dies liegt an der Art der Interaktion in einem *First Person Shooter*, nach einer 10min Partie *Deathmatch*[38] ist der Spieler *„schon fast erfreut“* über eine Zwangspause. Der Spieler muss zwar das Ausscheiden *(fraggen)* seines Avatars hinnehmen, aber aufgrund von kurzen Respawn-Zeiten und dem Erhaltenbleiben des eigenen Spielstands *(Punktestands)*, ist eine solche erzwungene Auszeit zu Gunsten der Mobilität hinnehmbar. Der Vorteil den die Opponenten aus einem Reaktionsausfall des Spielers ziehen können ist eher unwesentlich, da der Unterhaltungswert dieser Art von Spielen aus einer aktionsreichen und kurzweiligen Interaktion zwischen den Teilnehmern besteht und die dabei erreichten Statistiken zweitrangig sind. Zusammenfassend wird eine für den Spieler akzeptable Ausfallzeit durch das Verhältnis des Betrags zur Frequenz der Ausfallzeiten und das jeweilige Spiel *(-Genre)* bestimmt. Eine weitergehende Untersuchung ist damit erst mit erhobenen Messdaten über charakteristische Ausfallzeiten und deren Frequenz in zukünftigen mobilen Ad-Hoc Netzen möglich.

### 6.1.2 Limitierte Bandbreiten

Die zur Verfügung stehende Bandbreite ist in mobilen Ad-Hoc Netze genau wie im Internet, ein limitierender Faktor bei der Entwicklung von Mehrbenutzer-Echtzeitspielen. Eine Besonderheit ist in diesem Fall, das die mobilen Geräte kooperierend kommunizieren. Vorausgesetzt ein Gerät kann nicht direkt mit seinem Ziel kommunizieren, dann nutzt es andere Knoten *(mobile Geräte)* des Netzes um die gewünschten Nachrichten zu übertragen. Das Manko ist, dass diese Nachrichten weiterleitende Eigenschaft *(forwarding)*, den maximal

[36] verursacht keine Latenz bei der Übertragung der Spielinformationen
[37] ermittelt in *Unreal Tournament 2003* Partien
[38] typischer Spielmodus eines FPS, gleichbedeutend mit *„jeder gegen jeden“*

gebotenen Durchsatz beeinträchtigt und die Latenz erhöht. Ebenfalls muss bedacht werden das gleichzeitig neben dem Spiel-Verkehr, auch noch andere Dienste *(Internet-Surfen, Email, ...)* genutzt werden können, welche ebenfalls den maximalen Durchsatz verringern und die Latenz erhöhen. Sodas die effiziente Nutzung der Bandbreite für Zustands-Updates, auch bei mobilen Ad-Hoc Netzen im Vordergrund steht. Die präsentierten Konzepte zur Datenkomprimierung *(4.2.3)* und zur Filterung relevanter Information *(4.1.2)* sind hierfür bestens geeignet.
Bei herkömmlichen Client/Server-Netzwerkarchitekturen für Mehrbenutzer-Echtzeitspiele ist die Bandbreitenanforderung der Spiel-Server noch um ein Vielfaches höher als die der Clients *(5.2.4.3)*. In mobilen Ad-Hoc Netzen in denen mehrere Clients gleichzeitig auch als Server fungieren sollen *(2.3)*, um die Fehlertoleranz zu erhöhen, wird dieser Effekt noch verstärkt. Dieser Sachverhalt muss bei der Verteilung der zur Verfügung stehenden Bandbreite beachtet werden.
Trotz der oben genannten negativ wirkenden Faktoren sollte die derzeit maximal gebotene Bandbreite von 11Mbit/s *(IEEE 802.11b (WLAN))* in vielen Anwendungs-Szenarien ausreichen.

### 6.1.3 Synchronisation des verteilten Spielzustands

In zukünftigen Mehrbenutzer-Echtzeitspielen für mobile Ad-Hoc Netzwerke, ist die Verwendung einer Instanz des Spielzustands auf einem autoritativen Server, aufgrund der inakzeptablen Fehlertoleranz *(2.3),* nicht mehr ausreichend. Es wird vielmehr zu einer Verteilung des Spielzustands auf mehrere Server und des daraus resultierenden Problems der Synchronisierung des gemeinsam verwalteten Spielzustands kommen. In [CFK01] wurde dieses Problem, im Rahmen einer erweiterten Client/Server-Architektur für *First Person Shooter*, der Mirrored-Server-Architektur *(2.3)*, untersucht. Zur Konsistenzerhaltung werden die an den Servern ankommenden Client-Kommandos mit einem Zeitstempel versehen und dann unabhängig von ihrem Ereignistyp *(move, fire…)* an die jeweils restlichen Server versand *(command-based consistency protocol)*. Jeder Server berechnet sodann den Spielzustand zunächst optimistisch auf Basis der Reihenfolge der eingehenden Kommandos.
Alternativ könnte man je nach Ereignistyp und dadurch bestimmter Konsistenzanforderung, die Ereignisse auf den Ingress-Servern unterschiedlich behandeln *(event-based consistency protocol)*, z.B. Bewegungen unter Vorbehalt unmittelbar ausführen. Allerdings bedingt dies wiederum einen autoritativen Server, was zu einer inakzeptablen Fehlertoleranz führt und

zusätzlich die Anpassung eines existierenden Spiels weitaus komplexer werden lässt. Die erst genannte Art der Konsistenzerhaltung ist somit in mobilen Ad-Hoc Netzen zu favorisieren.

#### 6.1.3.1 Trailing-State-Synchronisation

Da beim Verteilen der Kommandos zwischen den Servern nicht gewährleistet ist, dass die Reihenfolge der Kommandos erhalten bleibt muss ein Synchronisations-Algorithmus einen solchen Fall erkennen und den Spielzustand korrigieren. Die Konsistenz entsteht somit durch ein globales[39] Ordnen der Ereignisse, sodass jeder Server auf Basis derselben Daten den korrekten Spielzustand berechnet. Man erreicht folglich, vom Spielzustand aus gesehen, dieselbe Situation, wie bei einer zentralisierten Client/Server-Architektur. In [CFK01] wurden auch aktuelle Synchronisations-Mechanismen, konservative *(lock-step, time-bucket synch.…)* wie auch optimistische *(Time Warp synch. …)*, hinsichtlich ihrer Eignung für Mehrbenutzer-Echtzeitspiele untersucht. Die Erkenntnis der Autoren ist das keiner der existierenden Algorithmen den Ansprüchen *(kurze Antwortzeiten, geringer Speicher- und Prozessor-Overhead)* aktueller *First Person Shooter* genügt und so wurde unter Anleihen existierender Algorithmen die **T**railing-**S**tate-**S**ynchronisation entworfen.

Zur **Behebung einer Inkonsistenz** unterhält TSS nun mehrere Kopien des Spielzustands, jede zu einer anderen, in der Vergangenheit liegenden, Simulationszeit *t*. Das heißt der aktuelle Zustand *(leading state)*, welcher den Clients mitgeteilt wird, resultiert aus der Verarbeitung der Ereignisse bis zur aktuellen Simulationszeit *t*, welche über die Differenz der realen Zeit und einer geringen oder keinen Synchronisationsverzögerung $d_0$ definiert ist. Der erste Trailing-Zustand steht für die Ausführung der Kommandos bis zur Simulationszeit *t* minus einer Synchronisationsverzögerung $d_1$, der zweite bis $t - d_2$ und so weiter. So entsteht eine Folge von Synchronisierern, jedes Mal mit einem etwas längeren $d_i$. Jeder höhere Trailing-Zustand wartet folglich länger mit der Ausführung und hat infolgedessen mehr Zeit die Ereignisse zu ordnen. Auf diese Art und Weise wird die Anzahl der zu speichernden Zustände, im Vergleich zu anderen Verfahren, minimiert und trotzdem können alle Inkonsistenzen erkannt und behoben werden. Zur **Entdeckung einer Inkonsistenz** vergleicht ein Synchronisierer nun die, durch die Ausführung der Kommandos entstehende, Spielzustandsänderung mit den aufgezeichneten Änderungen seines direkt übergeordnenten Trailing-Zustands. Wird eine Diskrepanz auf diesem Weg entdeckt, überschreibt der Synchronisierer den übergeordneten Trailing-Zustand *(rollback)* und alle Kommandos nach dem Rollback-Zeitpunkt werden in die Liste, der noch auszuführenden Kommandos *(pending*

[39] d.h. über alle Server

*list)*, des übergeordneten Synchronisierers übertragen. Ein solcher Rollback kann je nach Ereignistyp durchgeführt werden, so verwendet die Beispielimplementierung schwach und strikt konsistente Ereignisklassen. In der ersten Klasse werden Fehler in den Zuständen innerhalb eines kleinen Spielraums geduldet *(z.B. MOVE-Ereignis)*, während in der zweiten Klasse stets ein *Rollback* erfolgt *(z.B. FIRE-Ereignis)*. Der letzte Trailing-Zustand wird nicht mehr synchronisiert, so dass seine Synchronisationsverzögerung größer sein sollte, als die zu erwartenden Netzverzögerungen.

Die Abbildungen 6.1 und 6.2 beschreiben ein einfaches Beispiel der TSS. Sie zeigen drei Zustände mit den Synchronisationsverzögerungen von 0, 100 und 150ms. Zwei Kommandos sind hervorgehoben. Kommando A wird zum Zeitpunkt *t = 150* auf dem Server lokal ausgelöst[40] und unmittelbar danach im Leading-Zustand ausgeführt. Zum Zeitpunkt *t = 250* erreicht der erste Trailing-Zustand die Simulationszeit *t = 150* und führt das Kommando A aus. Da der Leading-Zustand in der Zeit lag, wird richtiger Weise keine Inkonsistenz festgestellt, auch nicht vom letzten Synchronisierer *(Abbildung 6.1)*.

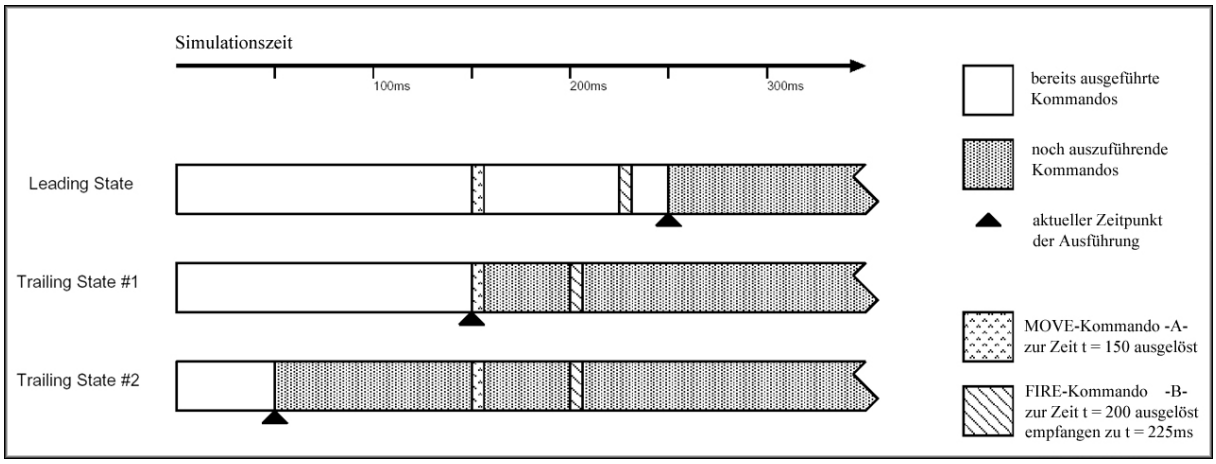


**Abbildung 6.1**

Beispiel einer Trailing-State-Synchronisation anhand zweier Kommandos [CFK01].

Kommando B wird zum Zeitpunkt *t = 200* auf einem entfernten Server ausgelöst und nach dem Eintreffen *t = 225* sofort im Leading-Zustand ausgeführt. Ferner wird das Kommando in den Pending-Listen der anderen Synchronisierer an der korrekten Position *(t = 200)* eingefügt. Zum Zeitpunkt *t = 300* führt der erste Trailing-Zustand das Kommando B aus. Der Synchronisierer kann allerdings beim darauf folgenden Vergleich kein FIRE-Ereignis zu *t =200* im Leading-Zustand bzw. in dessen Liste der bereits ausgeführten Kommandos finden. Demgemäß wird ein *Rollback* durchgeführt *(Abb. 6.2)*. Der Zustand des Trailing-Zustands

[40] d.h. ein von den, mit ihm verbundenen, Clients empfangenes Kommando

wird bis zu *t = 200* auf den Leading-Zustand kopiert. Dieser wiederum markiert alle Kommandos nach *t = 200* als unausgeführt und wiederholt deren Ausführung bis zum Zeitpunkt *t = 300*, was den *Rollback* abschließt.

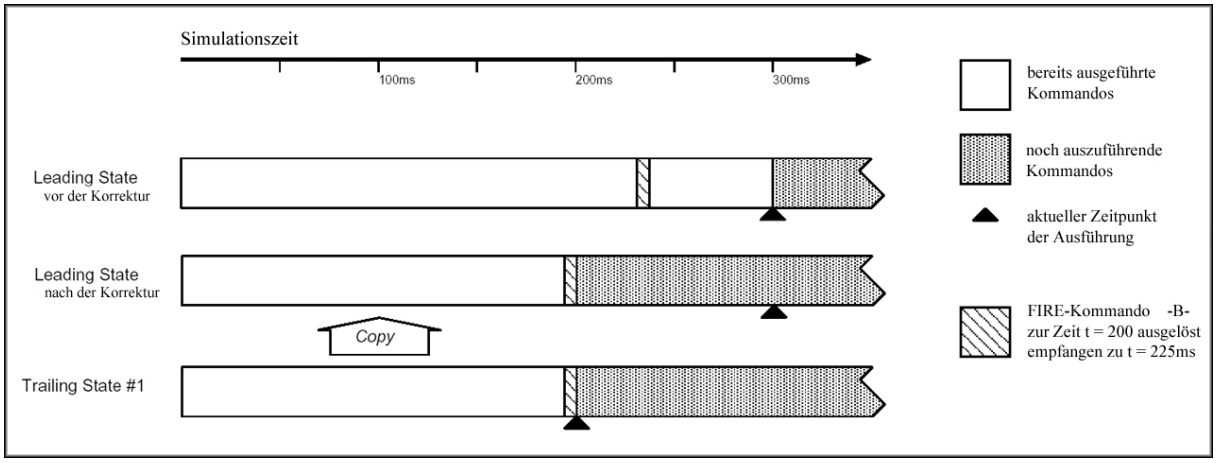


**Abbildung 6.2**

Beispielhafte Behebung einer Inkonsistenz anhand des Kommandos B mit Hilfe der Trailing-State-Synchronisation [CFK01].

Die Anzahl der Synchronisierer und die Länge der Synchronisationsverzögerungen bestimmen die Performance und Qualität der TSS. Sie müssen an die jeweilige Anwendung angepasst werden. Konkrete Untersuchungen bzw. Vorgaben einer optimalen Einstellung existieren in dieser Hinsicht noch nicht. Die resultierende Erkenntnis ist, dass TSS eine flexible und leistungsfähige Methode ist, um verteilte Spielzustände eines *First Person Shooter* zu synchronisieren. TSS ist zwar für FPS-Spiele über das Internet entwickelt wurden allerdings stimmen die Anforderungen an ein Synchronisations-Verfahren für Internet-Mehrbenutzerspiele mit denen für mobile Ad-Hoc Netze überein. Weshalb sich TSS auch zur Konsistenzerhaltung in zukünftigen *First Person Shooter* Spielen für mobile Ad-Hoc Netzen eignet.

### 6.1.4 Automatisches Finden von Mitspielern

Ein Feature von Spielen in mobilen Ad-hoc Netzen soll das spontane Zusammenfinden von Teilnehmern zum Starten einer Spielsession sein. Ein Mechanismus der genau dies ermöglicht wird in [RWW03] vorgestellt. In dem dort präsentierten Szenario versucht ein Client mit Hilfe des SLP-Protokolls *(Service Location Protocol)* [GTVD99] existierende Zonen-Server

zu finden und dann einer Spielsession beizutreten. Jeder Zonen-Server lässt dazu einen SLP-Service-Agenten und jeder Client einen SLP-User-Agenten laufen. Zur Kontaktaufnahme schickt der Client eine Multicast-Service-Anfrage an die Zonen-Server. Worauf ein Server, mit einer URL die den angebotenen Service *(u.a. Name, Port)* beschreibt, antwortet. Der Client entscheidet sich dann für den Zonen-Server, der die beste Antwortzeit aufweist und das entsprechende Spiel unterstützt.
Dieses Szenario beschreibt eine Möglichkeit eines der einfacheren Probleme in mobilen Ad-Hoc Netzen zu lösen.

### 6.1.5 Sicherheitsrelevante Aspekte

Die in Kapitel 4 vorgestellten Konzepte zur Erhöhung der Sicherheit, zur Gewährleistung fairer Mehrbenutzerspiele, können auch in mobilen Ad-Hoc Netzen eingesetzt werden, um die beschriebenen Angriffe zu erschweren bzw. zu verhindern. Die Schwierigkeit liegt jedoch darin, dass die Verfahren von schwer zu manipulierenden, da von den Spielentwicklern kontrollierten, Servern ausgehen. In mobilen Ad-Hoc Netzen werden die Server-Applikationen jedoch auf denselben Systemen wie die Client-Applikationen laufen. Folglich ergeben sich neue Angriffsmöglichkeiten, die noch untersucht und für die noch Sicherheitsmaßnahmen gefunden werden müssen.

## 6.2 Portierung eines FPS auf mobile Ad-Hoc Netze

Im Rahmen der Portierung möchte ich noch einmal auf die nötige Hardware zum Spielen in mobilen Ad-Hoc Netzen kurz eingehen. Die Portierung eines existierenden *First Person Shooter* gestaltet sich natürlich umso einfacher, je verwandter die Quell- und Ziel-Plattformen sind. So sind insbesondere Leistungsstarke Laptops mit 3D Graphikbeschleuniger und WLAN-Netzwerkkarte ein lukratives Zielsystem. Wenn die Portierung auf Grund drastischer Architektur-Unterschiede sehr aufwendig ist, eventuell sind nur Graphik-Set, Story und Spielnahme verwendbar, muss die Zielplattform einen hohen Grad der Verbreitung aufweisen, wie z.B. zukünftige Handy-Generationen, um eine Portierung zu rechtfertigen.
Ist das Plattform-Problem gelöst, muss eine Unterstützung für mobile Ad-Hoc Netze implementiert werden. Dies ist mit den Techniken der vorangegangenen Abschnitte realisierbar. Verallgemeinert könnten folgende Schritte durchgeführt werden:

- Erweiterung der Netzwerk-Architektur
  - Implementierung der Fehlertoleranten-Aspekte der Zonen-Server-Architektur

- Erweiterung der Server-Applikation
  - Eingehende Client-Kommandos werden mit einem Zeitstempel versehen
  - *Multicast* dieser Kommandos an alle Server und Verarbeitung auf den Servern
  - Synchronisierung der Spielzustände durch TSS
  - *Unicast/Multicast* Kommunikation[41] zwischen Server und Client

- Erweiterung der Client und Server-Applikation
  - Dienst zum Finden von Spielpartien z.B. durch SLP

## 6.3 Zusammenfassung

Die Mobilität der Endgeräte ist wohl das kritischste Problem bei der Entwicklung oder Portierung zukünftiger *First Person Shooter* für mobile Ad-Hoc Netze. Aus ihr resultieren die Forderungen nach einer erhöhten Fehlertoleranz und die damit verbundene Verteilung des Spielzustands auf mehrere Server. Mit Hilfe einer Netzwerkarchitektur die dies ermöglicht *(z.B. Zone-based-Gaming-Architektur)* und einem Synchronisations-Mechanismus *(z.B. Trailing-State-Synchronisation)* der für die Konsistenz des Spielzustands verantwortlich ist, kann man diesem Problem allerdings begegnen. Überdies lassen sich die, durch die Mobilität bedingen, temporären Netzausfälle im Zeitraum [0...150ms] durch die präsentierten Latenzkompensations-Techniken *(4.2)* vor dem Spieler verbergen. Das Problem der limitierten Bandbreite in mobilen Ad-Hoc Netzen ist verwandt mit dem entsprechenden Problem im Internet, so dass sich die dort angewendeten Konzepte zur optimalen Nutzung des verfügbaren Durchsatzes auch hier Verwendung finden. Eine Möglichkeit, um das automatische Finden von Mitspielern zu ermöglichen, ist die Verwendung des SLP-Protokolls.

[41] je nachdem, welche Art sich besser an die bestehende Client/Server Kommunikation anpassen lässt

# 7 Zusammenfassung und Ausblick

## 7.1 Zusammenfassung

In dieser Arbeit wurden zunächst die verschiedenen Netwerkarchitektur-Alternativen einander gegenübergestellt. Dabei stellte sich heraus, dass zum einen die Client/Server-Architektur dem Peer-to-Peer-Modell, u.a. wegen der Sicherheits-Aspekte, vorzuziehen ist. Zum anderen, das in zukünftigen mobilen Ad-Hoc Netzen eine Erweiterung einer Client/Server-Architektur *(z.B. Zone-based, Mirrored server…)*, die den Spielzustand auf mehrere Server verteilt, wegen der Fehleranfälligkeit in einer mobilen Umgebung, zum Einsatz kommen wird.
Weiterhin wurden die Gründe für das Auftreten von Latenz in Internet-Echtzeitspielen untersucht und es wurde festgestellt, dass die Latenz für ein frustfreies spielen von *First Person Shooter* Spielen unter 150ms liegen muss. Es wurde gezeigt, dass je nach Art der zu übertragenen Spielinformationen, zwischen TCP und UDP, für *(un)*zuverlässige Übertragung, gewechselt wird.
Ferner wurden Techniken zur Erhöhung der Skalierbarkeit und Reduktion der Bandbreitenanforderung, wie Seamless-Server und Filterung relevanter Informationen, als Techniken der Server-Applikation vorgestellt. Auf den Aspekt der Sicherung der Spieldaten vor Manipulationen wurde ebenso eingegangen, wie auf die zunehmende Bedeutung der kryptographischen Verfahren. Im Abschnitt über die Techniken der Client-Applikation wurden die grundlegenden Verfahren der Vorhersage-Techniken, die Interpolation und die Extrapolation, näher erläutert. Es wurde beschrieben, dass aufgrund der ständigen Richtungswechsel sich die Dead-Reckoning-Algorithmen nicht besonders für den Bereich der FPS eignen. Des Weiteren wurde eine Methode präsentiert, reversible Simulationen, um die Entscheidungen des Clients auf dem Server zu verifizieren.
Im darauf folgenden Kapitel wurden 2 Realisierungen, aus *Quake* und *Unreal*, der Filterung relevanter Informationen vorgestellt. Außerdem wurden zwei verschiedenen Umsetzungen von Vorhersage-Techniken, aus *Unreal* und *Half-Life*, zur Latenzkompensation präsentiert. Abschließend wurde mit Hilfe der Messwerte über die Bandbreitenanforderungen von *Quake* geschlussfolgert, dass die aktuellen Server-Anforderungen eine Schätzung für die, zum Betreiben von FPS nötige, Bandbreite in zukünftigen mobilen Ad-Hoc Netzen abgeben.

Das letzte Kapitel stellte auftretende Probleme in mobilen Ad-Hoc Netzen, wie Auswirkungen mobiler Endgeräte, limitierter Durchsatz und Synchronisation des verteilten Spielzustands vor. Es werden ebenfalls Konzepte zu deren Lösung aufgezeigt, insbesondere ein Verfahren zur Synchronisierung verteilter Spielzustände, die Trailing-State-Synchronisation. Ferner wurde eine mögliche Schrittfolge zur Portierung eines FPS auf mobile Ad-Hoc Netze beschrieben.

## 7.2 Ausblick

Eine anschließende, interessante Untersuchung währe eine über die Dauer und Frequenz der typischsten Ausfallzeiten in mobilen Ad-Hoc Netzen, zur Bestimmung des Intervalls, welches z.B. durch Latenzkompensations-Techniken zu überbrücken ist. Ähnlich gelagert wäre eine Studie über typische Anwendungs-Szenarien, d.h. welche Dienste *(Internet, Email)* werden häufig simultan eingesetzt und welche verbleibende Bandbreite resultiert daraus, in den einzelnen Fällen. Zur Abschätzung der Rentabilität etwaiger Portierungen, währen Daten über die häufigsten Typen von mobilen Spiel-Endgeräten *(Laptops, Handys…)* und deren Leistungsfähigkeit von Nutzen.

Allgemein stellt sich im Rahmen von Internet-Echtzeitspielen die Frage, ob man zukünftig auf Basis des Einsatzes kryptographischer Verfahren dazu übergeht den Clients zu vertrauen und nicht mehr vom Client ausgelöste Ereignisse *(Treffer…)* über den Server verifiziert.

# Literaturverzeichnis

**[Armit00]** Armitage Grenville, *Snapshot of Quake II Ethernet Frame Sizes*, SIGCOMM Internet Measurement Workshop, http://gja.spac e4me.com/things/quake2pktsize-040600.html, 2000.

**[Armit01]** Armitage Grenville, *Lag over 150 milliseconds is unacceptable*, SIGCOMM Internet Measurement Workshop, http://gja.spac e4me.com/things/quake3-latency-051701.html, 2001.

**[Abrash00]** Abrash Michael, *Michael Abrash's Graphics Programming Black Book*, Coriolis Group und in *Ramblings in Realtime,* http://www.bluesnews.com/abrash/contents.shtml, 2000.

**[Bern01]** Bernier Y. W., *Latency compensating methods in client/server in-game protocol design and optimization*, Proceedings of the 15th Games Developers Conference, San Jose, http://www.gdconf.com/archives/2001/ bernier.doc, 2001.

**[Booth97]** Booth Rick*, Inner Loops*, Addison-Wesley Developer Press, 1997.

**[CFK01]** Cronin E., Filstrup B., Kurc A., *A distributed multiplayer game system*, Technical Report EECS589 Term Project, University of Michigan, http://www.eecs.umich.ed/~bfilstru/quakefinal.pdf, 2001.

**[Fellner92]** Fellner Wolf-Dietrich, *Computergrafik*, BI-Wissenschaftsverlag, 1992.

**[GTVD99]** Gutman E., Perkins C., Veizades J., Day M., *RFC 2608: SLPv2: A service location protocol*, 1999.

**[Hend01]** Henderson Tristan, *Latency and User Behaviour on a Multiplayer Game Server*, Proceedings of the 3rd Workshop on Networked Group Communication, London, http://www.cs.ucl.ac.uk/staff/T.Henderson/doc/ngc2001.ps.gz, 2001.

**[ID03]** ID Software, *Quellcode älterer id-Spiele*, ftp://ftp.idsoftware.com/idstuff/source/, 2003.

**[Knuth98]** Knuth Donald, *The Art of Computer Programming, Volume 2: Semi numerical Algorithms*, Addison-Wesley Longman, Inc., 1998.

**[Kreim98]** Kreimeier Bernd, *Unofficial Quake Network Protocol Specs*, http://www.gamers.org/dEngine/quake/QDP/qnp.html, 1998.

**[Mann01]** Manninen Tony, *Virtual Team Interactions in Networked Multimedia Games Case: "Counter-Strike" – Multi-player 3D Action Game*, http://www.tol.oulu.fi/~tmannine/publications/PRESENCE2001_Virtual_Team_Interactions_in_Networked_Multimedia_Games.pdf, 2001.

**[Plumb93]** Plumb Collin, *MD-5 Public-Domain-Implementierung*, http://src.openresources.com/debian/src/admin/html/s/rpm_2.4.12.orig%20rpm-2.4.12%20lib%20md5.c.html, 1993.

**[RFC2104]** *HMAC: Keyed-Hashing for Message Authentication*, http://www.ietf.org/rfc/rfc2104.txt, 1997.

**[RFC2401]** *Security Architecture for Message Authentication*, http://www.ietf.org/rfc/rfc2401.txt, 1998.

**[RFC2405]** *The ESP DES-CBC Cipher Algorithm with Explicit IV*, http://www.ietf.org/rfc/rfc 2405.txt, 1998.

**[RWW03]** Riera S., Wellnitz O., Wolf L., *A Zone-based Gaming Architecture for Ad-Hoc Networks*, Proceedings of the Workshop on Network and System Support for Games, Redwood City, http://www.ibr.cs.tu-bs.de/users/wellnitz/papers/netgames2003/zone server.pdf, 2003.

**[SiCh94]** Singhal S. K., Cheriton D. R., *Exploiting position history for efficient remote rendering in networked VR.* Prescence: Teleoperators and Ves.

**[SIG03]** Special Interest Group *NetGame*, http://www.informatik.uni-mannheim.de/netgame/, 2003.

**[Sweeny99]** Sweeny Tim, *Unreal Networking Architecture*, http://unreal.epicgames.com/Network.htm, 1999.

**[Schnei96]** Schneider Bruce, *Applied Cryptography,* John Wiley & Sons, 1996.

**[Thor03]** Thor Alexander, *Massively Multiplayer Game Development*, Charles River Media, Inc., 2003.

**[Treg00]** Treglia Dante, *Game Programming Gems I*, Charles River Media, Inc., 2000.

**[Treg02]** Treglia Dante, *Game Programming Gems III*, Charles River Media, Inc., 2002.

**[Valve03]** Valve Software, *Half-Life Standard SDK 2.0*, http://download.com.com/3001-2121-3422522.html, 2003.

**[Varga99]** Varga Andreas, *PARSEC: Building the networking architecture for a distributed virtual universe*, University of Technology Vienna/Austria, http://www.cg.tuwien.ac.at/studentwork/CESCG 99/AVarga/paper.ps.gz, 1999.

**[Watt01]** Watt Alan, Policarpo Fabio, *3D Games Real-time Rendering and Software Technology*, Addison-Wesley Longman, Inc., 2001.

**[Wätjen00]** Wätjen Dietmar, *Kryptologie I Vorlesungsskript*, Technische Universität Braunschweig, http://www.iti.cs.tu-bs.de/TI-INFO/waetjen/krypto.ps, 2000.