Internet-Echtzeitspiele für mobile Netzwerke

Studienarbeit

von

cand. inform. Tobias Schröter

Aufgabenstellung und Betreuung

Dipl.-Inform. Oliver Wellnitz

Inhaltlicher Überblick

- NETZWERKARCHITEKTUR-MODELLE
  - 2.1 Peer-to-Peer-Netzwerkmodelle
  - 2.2 Client/Server-Netzwerkmodelle
  - 2.3 Anwendung in mobilen Ad-Hoc Netzwerken
- ANFORDERUNGEN AN EIN NETZWERKPROTOKOLL
  - 3.1 Ursachen von Latenz in Echtzeitspielen
  - 3.2 Informationsabhängige Wahl des Netzwerkprotokolls
- PROBLEMLÖSUNGEN FÜR MEHRBENUTZERSPIELE
  - 4.1 Techniken der Server-Applikation
  - 4.2 Techniken der Client-Applikation
- IMPLEMENTIERUNGEN IN ECHTZEITSPIELEN
- MOBILE AD-HOC NETZE
  - 6.1 Zukünftige Probleme von Echtzeitspielen und deren Lösungen

Netzwerkarchitektur-Modelle

Architektur-Modell	Pro	Contra
Peer-to-Peer	- **einfache Implementierung** - **geringe Latenz** - **hohe Fehlertoleranz**	- **Anfälligkeit gegenüber *Cheatern*** - **Spielzustands-Synchronisierung** - ***Frame-Locking-Verfahren***
▪ Unicasting		- hoher Kommunikationsaufwand - quadratische Skalierung - schwierige Spielinitialisierung
▪ Broadcasting/ Multicasting	- lineare Skalierung - Filterung relevanter Informationen	
Client/Server	- **Schutz vor *Cheatern*** - **Filterung relevant. Informationen** - **autoritativer Server**	- **geringe Fehlertoleranz** - **max. Spieleranzahl pro Server**

Netzwerkarchitektur-Modelle

- keine der herkömmlichen Architekturen eignet sich unverändert für den Einsatz in mobilen Ad-Hoc Netzen
- desshalb wurden die zwei folgenden Erweiterungen der Client/Server-Architekturen untersucht

Erweiterte Client/Server Architekturen

- **Mirrored-Server-Architektur**
  - Spielzustand wird auf mehrere topologisch verteilte Mirror-Server gespiegelt
  - Server übertragen per zuverlässigem *Multicasting* die eingehenden Kommandos ihrer Clients
  - alle Server berechnen optimistisch den resultierenden Spielzustand
  - Konsistenzerhaltung durch *Trailing-State-Synchronisation*
  - per *Unicasting* wird dieser den Clients mitgeteilt
  - Nachteil
    - keine expliziten Methoden zur Erhöhung der Fehlertoleranz

Erweiterte Client/Server Architekturen

- **Zone-based-Gaming-Architektur**
  - mehrere Clients werden zusätzlich als **Zonen-Server** ausgewiesen
  - Server empfangen die Updates ihrer Clients, berechnen den Spielzustand und senden diesen per *Multicasting* an die anderen Zonen-Server
  - nach Empfang oder Generierung eines neuen Spielzustands wird dieser per *Multicasting* an die Clients versandt
  - Fehlertolerante-Aspekte:
    - nach einem Server-Ausfall können sich die Clients an einen anderen Zonen-Server wenden
    - verliert ein Server den Kontakt zu allen anderen Servern wird er allein autoritativ
  - Nachteil
    - versenden von Spielzuständen anstatt Kommandos verbraucht mehr Bandbreite und lässt sich schwerer an existierende FPS anpassen
    - Synchronisierung der Spielzustände ist noch nicht gelöst

Erweiterte Client/Server Architekturen

- Kombination der beiden Architekturen stellt eine mögliche Architektur für FPS in mobilen Netzen dar
  - Erweiterung der Netzwerk-Architektur
    - Implementierung der Fehlertoleranten-Aspekte der Zonen-Server-Architektur
  - Erweiterung der Server-Applikation
    - eingehende Client-Kommandos werden mit einem Zeitstempel versehen
    - zuv. *Multicast* dieser Kommandos an alle Server und Verarbeitung auf den Servern
    - Synchronisierung der Spielzustände durch *Trailing-State-Synchronisation*
    - *Unicast/Multicast* Kommunikation zwischen Server und Client

Trailing-State-Synchronisation

- Konsistenz entsteht durch ein globales Ordnen der Ereignisse über alle Server
  - da beim Verteilen der Client-Kommandos deren Reihenfolge verloren gehen kann

- TSS unterhält mehrere Kopien des Spielzustands, jede zu einer anderen Simulationszeit *t*, um Inkonsistenzen zu entdecken

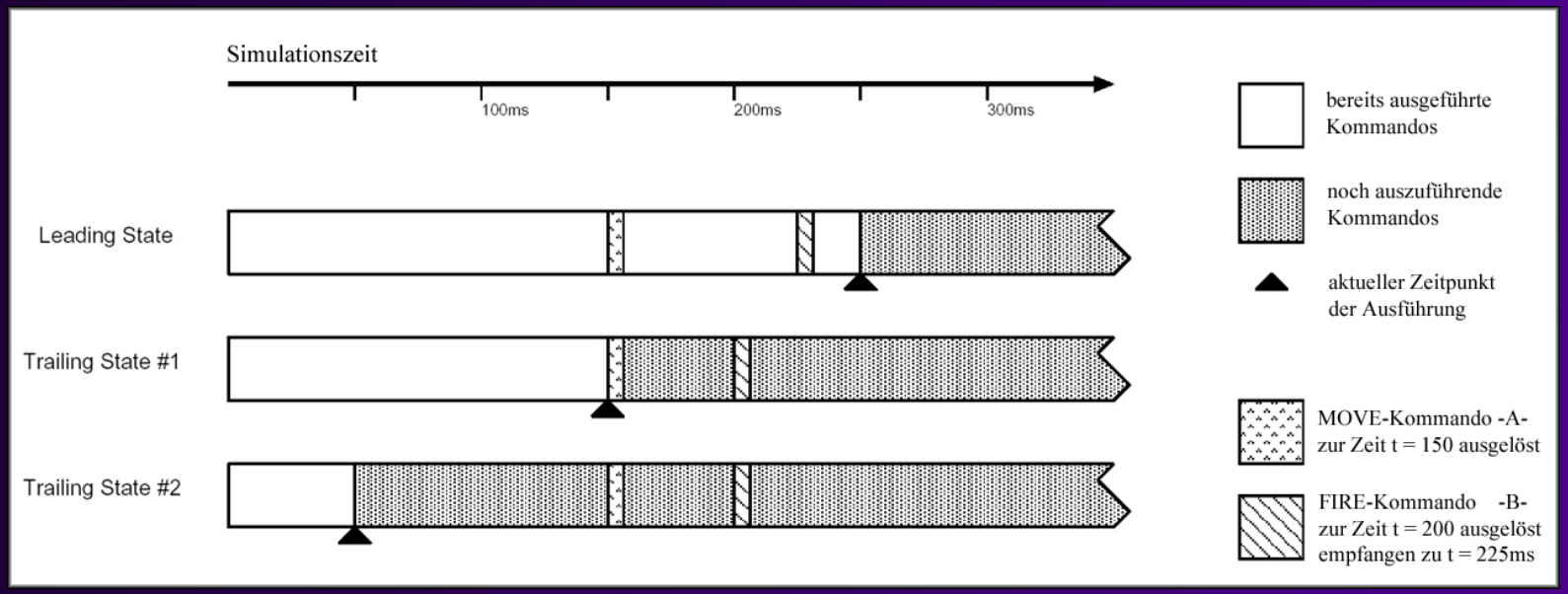
  - aktuelle Zustand *(leading state)* resultiert aus der Verarbeitung der Ereignisse bis zur akt. Sim. *t*

  - 1. Trailing-Zustand steht für die Ausführung der Kommandos bis zur Simulationszeit *t* minus einer Synchronisationsverzögerung d_1 der zweite bis $t - d_2$

  - so entsteht eine Folge von Synchronisierern, jeder mit einem etwas längeren d_i und infolgedessen mit mehr Zeit, um die Ereignisse zu ordnen

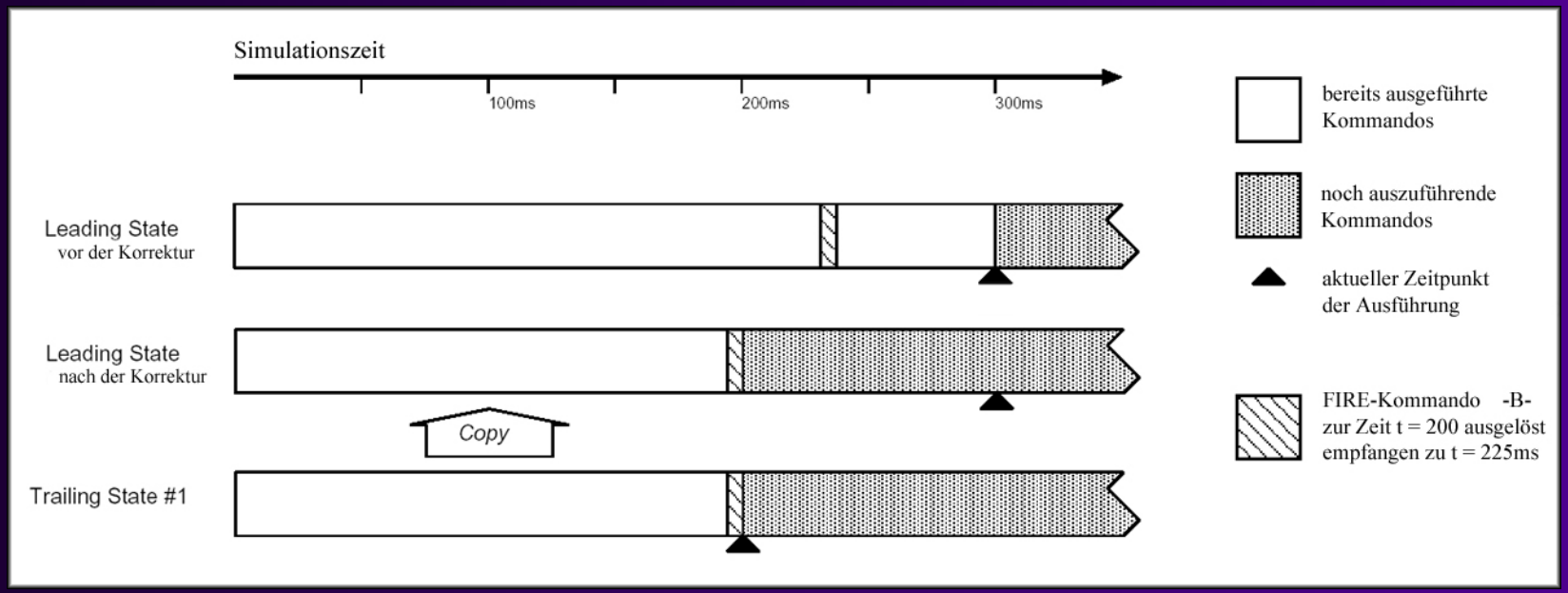
Trailing-State-Synchronisation

- Synchronisierer vergleicht seine erzeugte Zustandsänderung mit den aufgezeichneten Änderungen des übergeordnenten Trailing-Zustands
- im Fall einer Diskrepanz
  - überschreibt er den übergeordneten Trailing-Zustand *(rollback)*
  - Anwendung des Rollback je nach Ereignistyp
    - schwach und strikt konsistente Ereignisklassen
  - alle Kommandos nach disem Zpkt. werden in die Liste, der auszuführenden Kommandos *(pending-list)*, des übergeordneten Synchronisierers übertragen
- Synchronisationsablauf an Hand eines Bsp.:



Trailing-State-Synchronisation

- 1. Synchronisierer kann beim Vergleich kein FIRE-Ereignis zu *t =200* im Leading-Zustand bzw. in dessen Liste der bereits ausgeführten Kommandos finden →
  - Rollback
    - Zustand des Trailing-Zustands wird bis zu *t = 200* auf den Leading-Zustand kopiert
    - dieser markiert alle Kommandos nach *t = 200* als unausgeführt und wiederholt deren Ausführung bis *t = 300*



Latenz in Echtzeitspielen

- Verzögerungen auf dem Kommunikationskanal entstehen vor allem durch folgende Faktoren:
  - Spieleranzahl und Datenmenge der zu übertragenden Spielinformationen
  - Bandbreite und Wartezeiten des Kommunikationskanals
  - Grad der nötigen Zuverlässigkeit bei der Übertragung
- fluktuierende Latenzzeiten für den Menschen schlechter zu kompensieren als konstante
- Latenz im Bereich der *First Person Shooter* muss <150 ms für ein akzeptables *Gameplay* sein
- Verzögerungen auf dem Kommunikationskanal sind invers Proportional zur Zuverlässigkeit
  - 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
  - einige Spiele *(Quake, Unreal)* nutzen ausschließlich UDP und zusätzlich eigene Erweiterungen wie
    - Sequenznummern und einfache Bestätigungen *(acknowledgements)* für zuverl. Übertragungen

Problemlösungen für Mehrbenutzerspiele

- Techniken der Server-Applikation *-Überblick-*
  - Unterteilung des virtuellen Raums
    - Skalierbarkeit der C/S-Architektur wird erhöht
    - Aufteilung der Virtuellen-Welt auf mehrere Server
      - 2 Konzepte Seamless- und Zoned-Server
  - Filterung relevanter Informationen
    - Art Datenverteilungs-Management pro Server zur Verringerung der Nachrichtenanzahl
    - jeder Spieler besitzt einen Sichtbarkeits- bzw. Einflussbereich, außerhalb dessen müssen Änderungen am Spielzustand nicht übertragen werden
      - 2 Konzepte Rasterbasierte- und Objektbasierte-Filterung
  - Schutz vor Cheatern und Hackern
    - Hashfunktionen, Paket Replay, Verschlüsselter Datenverkehr

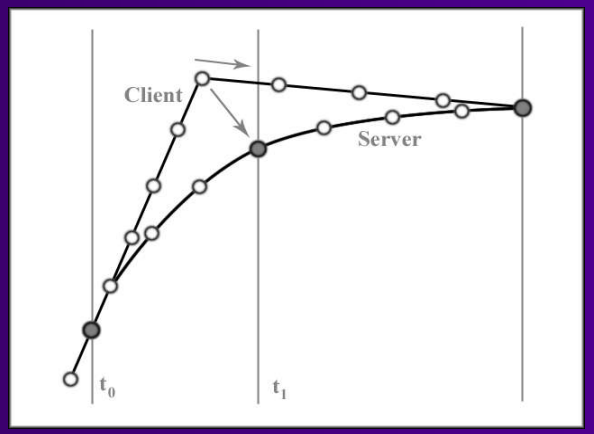
Techniken der Client-Applikation

- Movement Prediction
  - Verfahren zur Abschätzung zukünftiger Bewegungen
  - Ansätze der Vorhersage-Techniken
    - Einerseits werden Probleme durch verspätete, ausgefallene Zustands-Updates kompensiert, indem die Steuerung der Ghost's *(von Opponenten oder vom Server kontrollierte Objekte)* zeitweise durch diese Algorithmen übernommen wird
      - Interpolation und Extrapolation
      - Dead-Reckoning-Algorithmen
    - Andererseits wird jede, vom Server unbestätigte, Spielerbewegung des Clients als vorausgesagt betrachtet, da der Server jederzeit korrigierend eingreifen kann Wodurch die Anforderungen an die Bandbreite gesengt werden.

Movement Prediction

- Methoden der Interpolation und Extrapolation
  - mit Hilfe dieser beider Verfahren wird versucht, die Auswirkungen größerer Latenz zu kompensieren

- Extrapolation
  - wird angewendet, wenn kein aktuelles Positions-Update eines von Opponenten oder dem Server kontrollierten Objektes *(Ghost)* vorliegt, um die wahrscheinlich aktuelle Position des Objektes abzuschätzen



- Interpolation
  - wird angewendet, um einen fließenden Übergang aus einer divergierten, extrapolierten Position in eine zukünftige, auf Basis neuer Daten, extrapolierte Position zu ermöglichen

Movement Prediction

- Ein Verfahren das die Rate der Client-Zustands-Updates der jeweiligen Spielsituation anpasst,ist der

- **D**ead-**R**eckoning-**A**lgorithmus

  - übernimmt die Steuerung der Ghost's auf dem Client
  - in dem er auf Basis der letzten übertragenen Zustands-Informationen *(p, v, a)* zukünftige Positionen durch Extrapolations- und Interpolations-Techniken berechnet

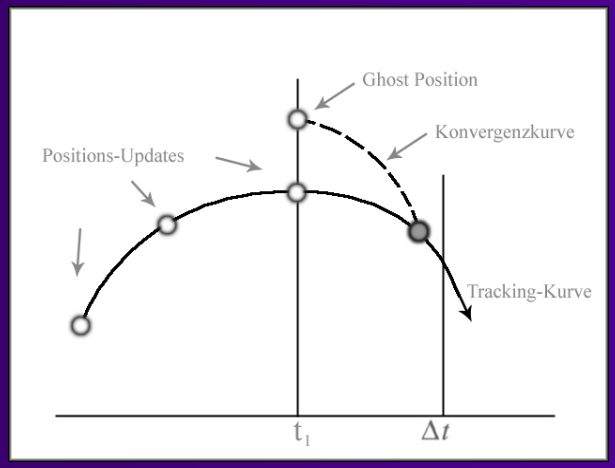
  - Synchronität der Simulationen
    - ermittelt jeder der Clients zusätzlich durch den DRA die Differenz zwischen der extrapolierten und realen Position, der von ihm kontrollierten Objekte.
    - Grenzwert überschritten, müssen die aktuellen Spieler-Informationen neu versenden werden.

- Der Erfolg der Methode hängt vom Verhältnis der Informations-Änderungen zur Frequenz der Informations-Übertragungen ab

  - nicht für jedes Spiel-Genre geeignet
  - insbesondere *FPS* sind durch, im Verhältnis zu Vehikle-Simulationen, langsame Spieler-Bewegungen und häufige Richtungswechsel gekennzeichnet und damit weniger für diesen Algorithmus prädestiniert

Movement Prediction

- Singhal und Cheriton, Vereinfachung des DRA
  - die nur noch auf Positions-Updates basiert
  - weniger anfällig gegen v- und a-Änderungen
  - Ähnlich DRA werden 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
  - Verlauf der Kurve bestimmt Abschätzungen über zukünftige Spielerpositionen
  - Nach dem Empfang einer aktuellen Position wird die Tracking-Kurve angepasst und Konvergenz-Kurve abgeleitet
    - Welche die Position des Ghost fließend auf die Tracking-Kurve zurückführt
  - Schätzungen dieser Art liegen auch bei FPS häufiger richtig und sind zu bevorzugen

Zukünftige Probleme von Echtzeitspielen

- zwei der wichtigsten Probleme neben der erwähnten Synchronisierung der Spielzustände
- Mobilität der Endgeräte
  - Geräte können sich aus ihren Empfangsbereiche heraus bewegen
    - Stellt einen Ausfall des Servers oder Clients dar
    - benötigt fehlertolerantes Netzwerk-Architektur-Modell
  - temporärer Ausfall der Netzverbindung
    - z.B. wegen einer Abschattung durch Gebäude während der Bewegung
    - ist mit sprunghaft ansteigender Latenz gleichzusetzen
    - Kompensation durch Methoden der *Movement Prediction*

Zukünftige Probleme von Echtzeitspielen

- Limitierte Bandbreiten
  - mobilen Geräte kommunizieren kooperierend, d.h. andere Knoten leiten Nachrichten weiter
    - geht zu Lasten der Bandbreite und Latenz
  - gleichzeitig können andere Dienste *(Internet-Surfen, Email, ...)* genutzt werden
    - geht zu Lasten der Bandbreite und Latenz
  - dem entgegen wirken
    - Methoden der Datenkomprimierung
      - Reduktion der Genauigkeit der versendeten Daten beschränkt
        - z.B. beschränkt *Unreal* optional die Komponenten der Positionsvektoren auf 16-Bit Integer
    - Filterung relevanter Information

ENDE