

Skript zur Vorlesung  
**Managing and Mining Multiplayer Online Games**  
im Sommersemester 2015

# Kapitel 4: Das Persistenz System

Skript © 2012 Matthias Schubert

[http://www.dbs.ifi.lmu.de/cms/VO\\_Managing\\_Massive\\_Multiplayer\\_Online\\_Games](http://www.dbs.ifi.lmu.de/cms/VO_Managing_Massive_Multiplayer_Online_Games)

## Kapitelübersicht

---

- Anforderungen an ein Persistenz System
- SaveGames und Replays
  - State Logs
  - Transition Logs
  - Action Logs
- Persistenz in MMOs
- Check-Point-Recovery Verfahren
  - Naive Snapshot
  - Copy-on-update
  - Wait-free Zigzag
  - Wait-free PingPong

# Aufgaben des Persistenz Systems

---

## 1. Sichern eines Teils ( $gs \in GS$ ) des aktuellen Game States $GS$

- Erlaubt das Spiel zu einem anderen Zeitpunkt fortzusetzen
- Bei Systemabsturz einen konsistenten Spielstand zu erzeugen
- Spielstände können bewusst nur an bestimmten Orten gespeichert werden (z.B. Resident Evil, Diablo III, ...)
- Es können bewusst bestimmte Teile nicht gesichert werden (Position der NPC/Monster/Gegner, Zufallskarten, ...)

## 2. Speichern des Spielverlaufs ( $gs_1, \dots, gs_{end}$ )

- Erlaubt das Nachvollziehen und die Analyse des Spielverlaufs
- Spielverläufe werden in der Regel auf den Clients angelegt
- Je nach Speichermethode können diese „Replays“ sehr groß werden
- Da der letzte GS auch ein Teil des Replays ist, können Replays auch zu Sicherung verwendet werden

3

# Anforderungen an das Persistenz System

---

- Speichern darf das Spiel nicht verlangsamen  
=> Tick-Dauer darf das Zeitlimit nicht überschreiten
- der Spielstand sollte möglichst aktuell sein  
=> GS sollte möglichst in jedem Tick abgespeichert werden
- Laden eines Spielstandes soll einen konsistenten Zustand erzeugen  
=> Alle GEs im Spielstand sollten gleich aktuell sein  
=> Minimalziel Game State muss logisch konsistent sein  
(jede GE kommt nur einmal vor ...)

### Wichtige Einflussgrößen:

- Größe des Game States
- Anforderungen an Aktualität und Tick-Rate
- Rolle der Ladezeit beim Wiederherstellen
- Teil des Game States / Spielverlaufs der gesichert werden soll

4

# Methoden für Replays und Save-Games

---

**Save-Game/Replay:** Lokale Datei, die Game State/Spielverlauf enthält.

## State-Log:

Jede Game Entity wird in jedem Tick gesichert

⇒ Sequentielle Datei mit Folge von Game States

**Beispiel:** Demo-Files von Quake/Halflife/Counterstrike  
(geparsed und in XML transformiert) (Bachelorarbeit: J. Rummel 2011)

```
<replay path="c:\data\ncsdemos\ndust210.dem" duration="3379,459" noOfRounds="39"
  mapname="dedust2" maxClients="16" serverName="HLTV.org/VeryGames.net">
  <rounds>
  <roundnumber="1" roundBegin="0" roundEnd="40,496184" endingReason="Bombing" winner="Terrorists">
  <teamScore ct="2" t="1" />
  <ticks>
  <ticktime="1">... </tick>
  <ticktime="2">... </tick>
  <ticktime="3">
  <players>
  <playerid="765611887383" localName="q" 15 team="Terrorist" kills="3" deaths="7" x="680" y="819" z="164"
  angle0="2" angle1="60" moveType="" weaponModel="172,, modelIndex="149" isHit="Helmet" outOfAmmo="
  Rifle" />
  ....
```

5

## Diskussion State-Log

---

### Vorteile:

- dokumentiert echte Folge von Game States
- Wahlfreier Einsprung zu jedem Zeitpunkt ist möglich
- Ladevorgang sehr einfach und schnell

### Nachteile:

- bei geringer Änderungsrate sehr redundant
- großes Datenvolumen durch hohe zeitliche Auflösung (je Tick)
- max. Schreiblast => evtl. nicht auf große Game States anwendbar

6

# Transition-Log

---

Logge alle Änderungen des Game States durch:

- Zeitstempel
- ID\_GameEntity
- Attribut
- neuer Wert

## Vorteil:

- kompakter als Snap-Shot

## Nachteil:

- mehr Aufwand bei der Rekonstruktion des Game States
- Änderungen müssen beim Persistenz-System registriert werden

7

# Action-Log

---

- enthält die zeitliche Abfolge aller Benutzereingaben
- benötigt das Spiel um Spielstände mittels der Benutzereingaben „nachzuspielen“
- Zufallereignisse müssen gespeichert werden (Ergebnisse oder Zahlen)

**Beispiel:** Starcraft II (\*.sc2replays Dateien) nach Parsing mit SC2gears  
(<http://sites.google.com/site/sc2gears/>)

```
0:00 TSLHyuN   Select Hatchery (10230)
0:00 TSLHyuN   Select Larva x3 (1027c,10280,10284), Deselect all
0:00 TSLHyuN   Train Drone
0:01 TSLHyuN   Train Drone
0:01 roxkisSlivko Select Hatchery (10250)
0:01 roxkisSlivko Select Larva x3 (10270,10274,10278), Deselect all
0:01 TSLHyuN   Select Drone x6 (10234,10238,1023c,10240,10244,10248), Deselect all
0:01 roxkisSlivko Train Drone
0:01 TSLHyuN   Right click; target: Mineral Field (10114)
0:01 roxkisSlivko Select Egg (10270), Deselect 1 unit
0:01 roxkisSlivko Select Drone x6 (10254,10258,1025c,10260,10264,10268), Deselect all
0:01 roxkisSlivko Right click; target: Mineral Field (10164)
0:01 TSLHyuN   Deselect 6 units
0:01 roxkisSlivko Right click; target: Mineral Field (10164)
0:02 TSLHyuN   Right click; target: Mineral Field (10170)
0:02 roxkisSlivko Deselect 6 units
....
```

8

# Diskussion Action-Log

---

## *Vorteile:*

- Speicherung kann kompakter sein  
(Action per Minute APM vs. Ticks per Second)
- keine Redundanz
- kann mehr Informationen enthalten als Game State  
=> Benutzereingaben die den GS nicht verändert haben  
(z.B. Maus-Bewegungen, Anzeigen, ...)

## *Nachteil:*

- hoher Aufwand bei der Wiederherstellung des letzten Zustands durch Simulation des Spielverlaufs
- Prinzip wird schwierig bei hohem Anteil an Zufallselementen
- Computergesteuerte Spieler/Objekte:
  - erfordert deterministisches Verhalten  
(NPC Verhalten ist Teil des Spiels und kann ebenfalls nachsimuliert werden)
  - AI muss über dieselben Befehlsprimitive gesteuert werden wie menschliche Spieler

9

---

## Spielstände in MMOs

---

**Normale Spiele:** „kleine“ Game States bei dezentraler Speicherung

=> lokale Clients schreiben dezentrale Spielstände in Dateien

### **Bei MMO-Games:**

- kompletter und konsistenter Game State liegt nur beim Server vor  
=> Persistenz muss zentral am Server realisiert werden  
=> Bei Verbindungsverlust zählt der Zustand auf dem Server
- Game State ist wesentlich umfangreicher  
=> Performanz beim Wegschreiben kann Game Loop bremsen  
=> unstrukturiertes Abspeichern in einer Datei ist unpraktisch  
(selektives Laden von Spielern nach einem Login)  
=> historische Information über Spielverlauf verursachen  
teils sehr große Datenvolumen

10

# MMOGs und relationale Datenbanken

---

Verwaltung großer Mengen fest strukturierter Objekte

=> Verwendung einer relationalen Datenbank

## **Vorteile bei der Verwendung einer relationalen Datenbank:**

- Datenbank übernehmen bestimmte Konsistenzprüfungen (keine Duplikate, ...)
- Datenbanken unterstützen selektive Anfragen durch effiziente Index-Strukturen
- aktueller Game State steht direkt zur Verfügung
- Datenbanksysteme verfügen über eigene Recovery-Mechanismen (Absicherung gegen System- und Hardwarefehler )

## **Nachteile:**

- strukturiertes Abspeichern und Vermeidung von Anomalien erhöhen die Bearbeitungszeiten von Änderungsoperationen

11

# Persistenz über Log-Dateien

---

Schnelle Speicherung aller Änderungen im Game State

=> Logging mit sequentiellen Dateien

## **Vorteile:**

- System hat kaum Overhead
- Schreiben ans Ende eines sequentiellen Files  
=> minimale Wartezeiten

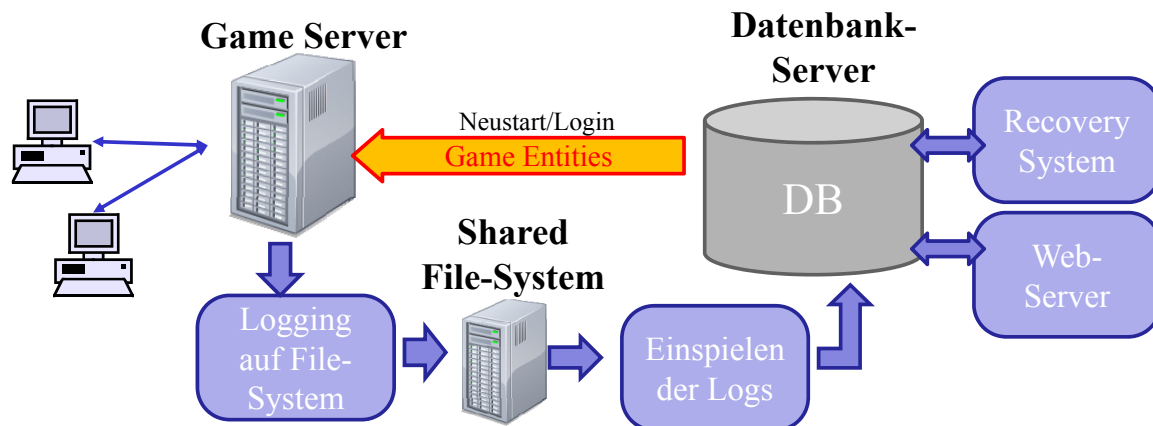
## **Nachteile:**

- keine Sicherung bei Platten- oder Systemfehlern
- keine Unterstützung von selektiven Anfragen
- Laden des letzten konsistenten Game States erfordert eventuell Check-Point als Ausgangspunkt für relative Änderungen

12

# Beispiel für eine Hybride Architektur

- Wegschreiben der Daten vom Game Server auf das Persistenz System über Logging-Verfahren => minimale Belastung der Tick-Zeiten
- Im Persistenz-Server: Einspielen der Log-Dateien in einen Datenbankserver
  - dauerhafte und sichere Speicherung von Game States
    - konsistente und redundanzfreie Version des Game States
    - mit Recovery Mechanismen (ggf. Remote-Sicherung)
  - vom Spiel entkoppelte Information für Anfragedienste (z.B. Armory, ...)



13

## Offene Punkte

- Welche Logging-Methode ist die beste für volatile Systeme?
  - Änderungsrate pro Objekte (Wie viele Objekte ändern sich in einem Tick?)
  - Komplexität der Änderungen (sind Aktionen kompakter als die resultierenden Attributänderungen?)
  - Burstiness der Änderungen (Fallen Änderungen periodisch in großer Zahl an?)
- Was muss vom Game State gespeichert werden?
  - alle beweglichen Objekte
  - Zustände aller Spieler
  - räumliche Positionen von Spielern und Objekten
- Aktualität und Rückstand
  - Welche Aktionen müssen wie schnell gesichert werden? (Laufen vs. Looten)

14

# Check-Point Recovery Verfahren für Spiele

- **Check-Point:** Konsistenter Zustand des Game States
- **Check-Point Phase:** Zeitraum, der benötigt wird um einen Check-Point zu erstellen.
- **Ziel:** Wegschreiben des Game State mit minimalem Zeitaufwand im Game Loop  
=> Minimale Auswirkung auf Latenz
- **Idee:** Speichere Informationen nicht direkt, sondern kopiere alle Informationen zunächst in einen Schattenspeicher
  - ⇒ Daten im Schattenspeicher werden nicht mehr durch Aktionen verändert
  - ⇒ Game Loop braucht nicht auf das IO-System zu warten (Verwendung eines asynchronen Schreibe-Threads)
  - ⇒ Schreiben kann mehrere Ticks dauern, Persistenz System hängt etwas hinterher
- **Unterscheidung der Strategien nach:**
  - Bulk-Copies vs. selektives Kopieren
  - Locking von Einzel-Objekten
  - Rücksetzen von Dirty-Bits
  - Speicherverbrauch

15

## Naive-Snapshot

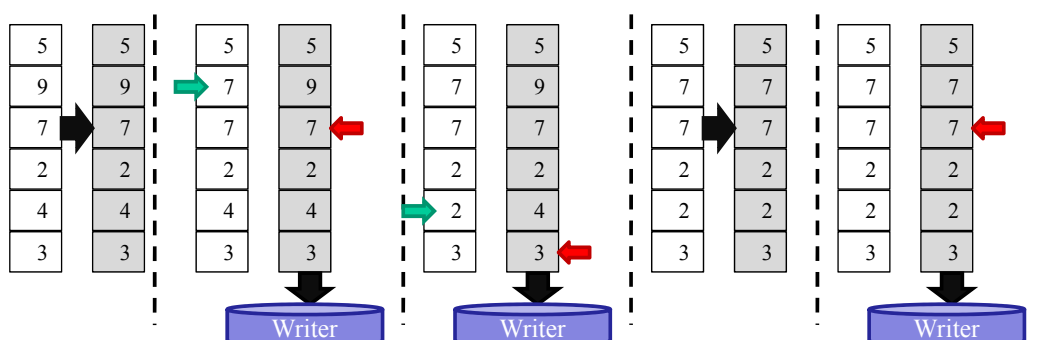
- Sofern Schreibe-Thread mit dem letzten Check-Point fertig ist, kopiere am Tick-Ende den gesamten Game State in den Schattenspeicher
- nach Ende der Kopie startet der nächste Tick und der Schreibe-Thread schreibt den kopierten Game State aus dem Schattenspeicher

### Vorteile:

- kein Overhead durch Locking oder Bit-Resets
- Effizient bei sehr vielen Änderungen

### Nachteile:

- bei begrenzten Änderungen viel Overhead beim Kopieren und Schreiben
- periodisch sehr viel Aufwand in Ticks an denen Game State kopiert wird



16



# Copy-On-Update

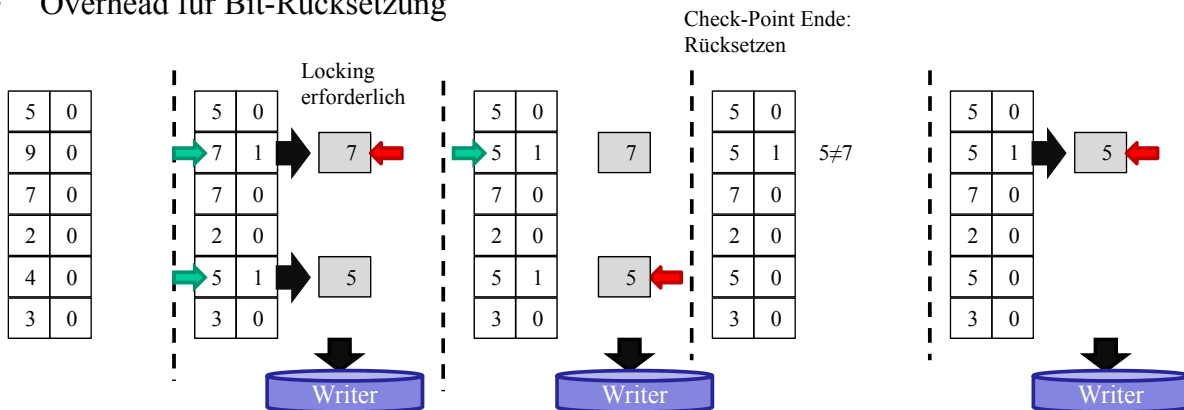
- Objekte werden bei Änderung in den Schattenspeicher kopiert und markiert (Dirty-Bits)
- Objekte werden pro Phase nur 1 Mal kopiert
- nachdem Check-Point geschrieben wurde, werden Markierungen zurückgesetzt

## Vorteil:

- weniger Änderungsvolumen
- bessere Umverteilung der Kopien auf mehrere Ticks

## Nachteile:

- erfordert Locking, damit nicht gleichzeitig geändert und kopiert wird
- Overhead für Bit-Rücksetzung



17

# Wait-Free Zigzag

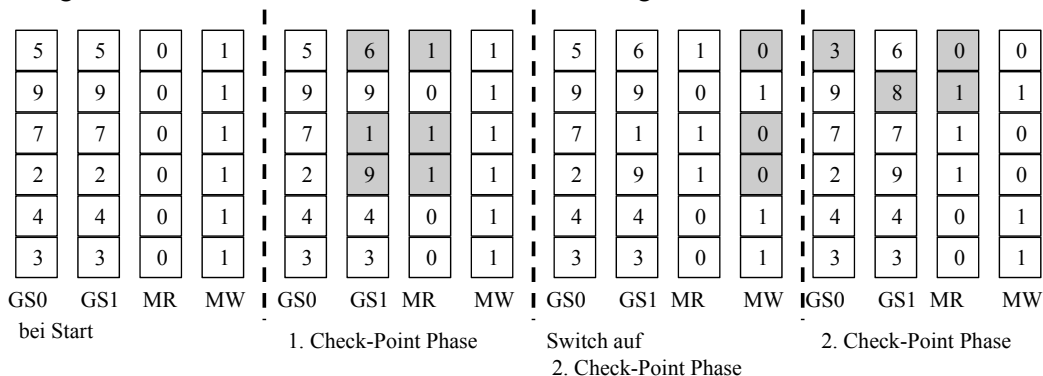
- jedes Objekt hat 2 Flags, die auf einen GS zeigen: MW (Schreibe-State) und MR (Lese-State) für Aktionsverarbeitung
- Eintrag in GS[MW] wird während der Phase nicht geändert
- Bei Änderung wird MR auf MW gesetzt
- Writer-Thread liest jeweils das Element aus GS[ $-MW_i$ ] für Objekt  $i$

## Vorteil:

- kein Locking notwendig
- Änderung können über die Zeit hinweg verteilt geschrieben werden

## Nachteil:

- benötigt immer noch Switch-Phase mit Bit-Rücksetzungen



18

# Wait-Free Ping-Pong

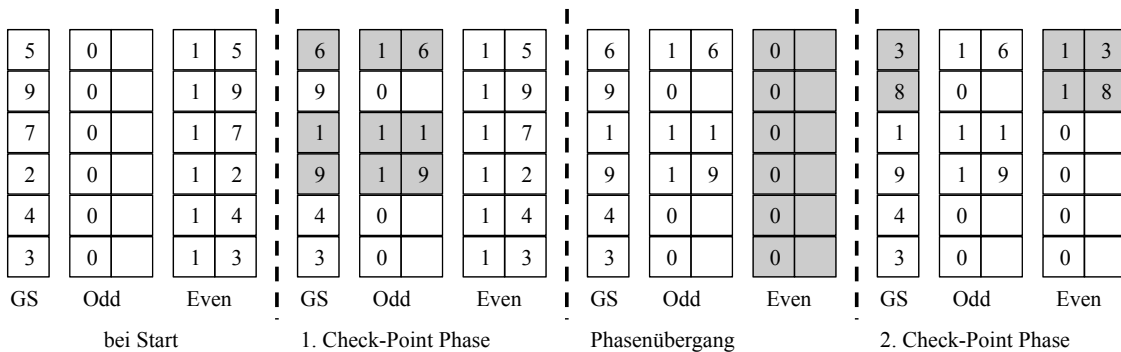
- Verwendung von 3 Game States:  
Aktionsverwaltung (GS), Persistenz-System (read), Persistenz-System (write) (odd oder even)
- Update erfolgen immer in GS und Persistenz-System (write)
- Writer-Thread liest Persistenz-System (read)
- bei Phasenwechseln vertauschen von Persistenz-System (write) und Persistenz-System (read)

## Vorteil:

- weder Locking noch Bit-Rücksetzungen beim Phasenwechsel

## Nachteil:

- dreifacher Speicherbedarf statt bisher nur doppelter



## Diskussion

- Naive-Snapshot ist bei sehr volatilen Systemen mit vielen Änderungen am einfachsten umzusetzen
- Je weniger Änderungen auftreten, desto vorteilhafter werden die übrigen Verfahren
- Wait-Free Ping-Pong und Wait-Free Zig-Zag verhindern Locking der Game Entities durch das Persistenzsystem
- Wait-Free Zig-Zag reduziert sogar den Overhead beim Phasenwechsel, verbraucht aber extrem viel Speicher

# Lernziele

---

- Aufgaben des Persistenz Systems
  - Speicherung eines Game States
  - Speicherung einer Folge von Game States (Replay)
- Arten von Save-Games und Replays:  
State-Log, Transition-Log, Action-Log
- Persistenz in MMOs:  
Datenbanken, Logging und Hybride Architekturen
- Check-Point Recovery Verfahren für MMOs
  - Naive-Snap Shot
  - Copy-on-update
  - Wait-Free Zigzag
  - Wait-Free Ping-Pong

21

---

# Literatur

- Tuan Cao, Marcos Vaz Salles, Benjamin Sowell, Yao Yue, Alan Demers, Johannes Gehrke, Walker White  
**Fast checkpoint recovery algorithms for frequently consistent applications**  
In Proceedings of the 2011 International Conference on Management of Data, 2011.
- Marcos Antonio Vaz Salles, Tuan Cao, Benjamin Sowell, Alan J. Demers, Johannes Gehrke, Christoph Koch, Walker M. White  
**An Evaluation of Checkpoint Recovery for Massively Multiplayer Online Games**  
PVLDB, 2(1): 1258-1269, 2009.
- Kaiwen Zhang, Bettina Kemme, and Alexandre Denault. 2008. **Persistence in massively multiplayer online games**. In *Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games (NetGames '08)*, ACM, New York, NY, USA, 53-58.

22