Transactional Memory for Distributed Systems

Michael Schöttner, Marc-Florian Müller,
Kim-Thomas Möller, Michael Sonnenfroh

Heinrich-Heine Universität Düsseldorf

Abteilung Betriebssysteme

Vorschau

HEINRICH HEINE UNIVERSITÄT DÜSSELDORF

1. Transaktionaler Speicher
2. Verteilter Transaktionaler Speicher
3. Ergebnisse
4. Fazit & Ausblick

Ausgangssituation

- **Problem: CPU Taktraten wachsen nur noch sehr langsam**
- **Lösung: Mehrkern-Prozessoren**
  → simultane Ausführung paralleler Threads eines Prozesses
- **Konsequenz: bevorzugt parallele Programmierung**

Ausgangssituation

- **Herausforderung: Synchronisierung von Threads**
  - Beim Zugriff auf gemeinsame Variablen
  - Sonst entstehen Wettlaufsituationen, lost update etc.

- **Traditionelle Lösung: Sperren / kritische Abschnitte**
  - Je nach Problem schwierig zu realisieren
  - Gefahr von Fehlern und Verklemmungen
  - Maximale Nebenläufigkeit gewünscht

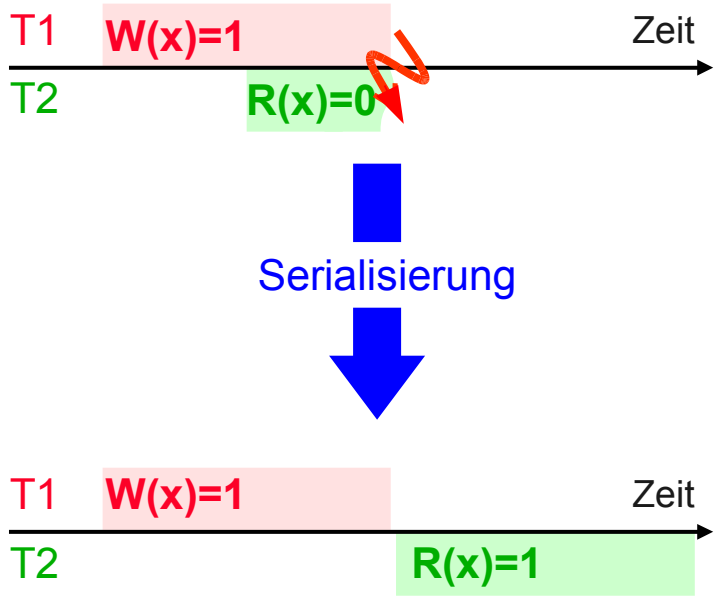
→ **Alternative: transaktionaler Speicher**

Programmiermodell

- **Idee: Spekulative Transaktionen statt Sperren**
  (Transaktionen + optimistische Synchronisierung)
- **Transaktionen haben ACId-Eigenschaften**
- **Programmierer muss Transaktion definieren: BOT & EOT**
- **Oft werden kritische Abschnitte als Transaktion realisiert**
  - ***Java:*** ***synchronized (obj) { … }***
  - ***TM:*** ***BOT { … } EOT***

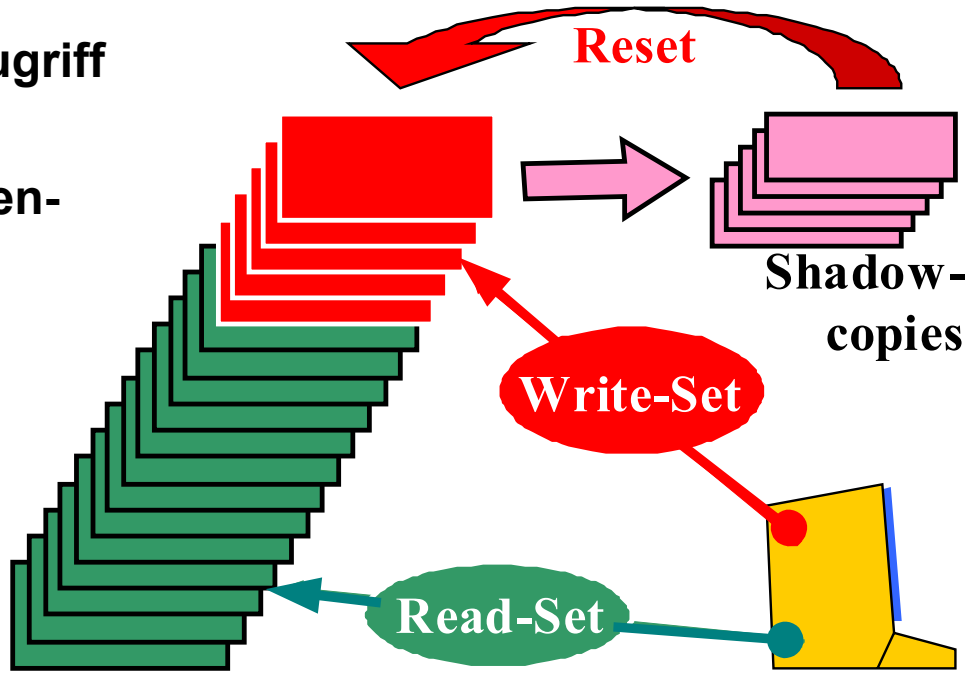
Validierung

- Aufzeichnen spekulativer Lese- und Schreibzugriffe
  --> Read- und Write-Set
- Schreibzugriffe werden erst beim Commit sichtbar
- Validierung: Vergleiche überlappende noch laufende TAs
- Abbruch & Rücksetzung im Konfliktfall →



Rücksetzbarkeit

- **Speicherzugriffe müssen erkannt werden**
  - beim ersten Lese-/Schreibzugriff innerhalb einer Transaktion
  - Bei Schreibzugriffen Schatten-kopien anlegen, notwendig für Rücksetzbarkeit



- **Betriebssystem- bzw. I/O-Aufrufe sind kritisch …**

Implementierungsaspekte

- TM-Systeme in der Regel auf Cache-Ebene implementiert
- Zusätzliche Bits pro Cache-Line zur Verwaltung der Zugriffe
- Schattenkopien von Cache-Lines → Rücksetzbarkeit
- TA-Validierung als Cache-Kohärenzprotokoll
- Großes Interesse in Forschung und Industrie

2. Verteilter Transaktionaler Speicher

XtreemOS

- **Linux-basiertes Grid-Betriebssystem**
  - Für vernetzte Desktop PCs
  - Und für Cluster

- **Integriertes Projekt, gefördert von EU**
  - 19 Partner, ~120 Personen
  - http://www.xtreemos.eu

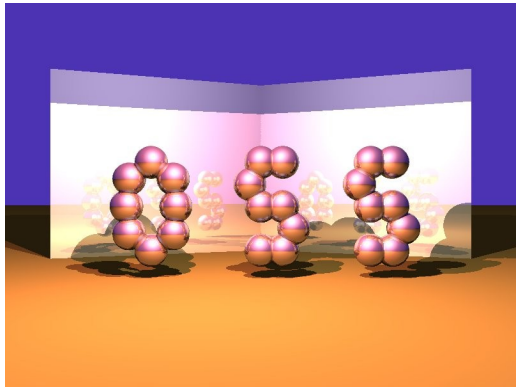
- **HHU-Beteiligung**
  - Grid Checkpointing Service (Job-Migration / Fehlertoleranz)
  - Object Sharing Service (Daten-Management)

Zielsetzung

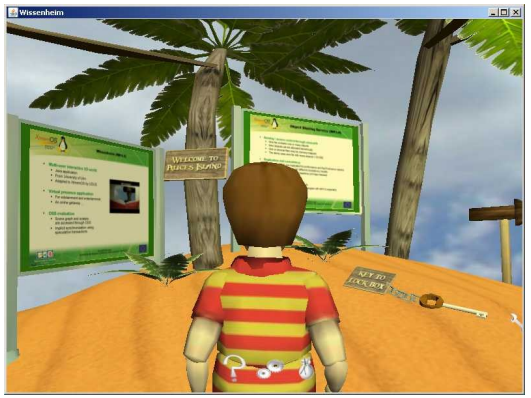
- **Vereinfachung der Entwicklung verteilter und paralleler Anwendungen (im Cluster und Grid-Umgebungen**
- **Transparenter Zugriff auf entfernte Speicherobjekte**
- **Automatische Replikation**
- **Transaktionale Konsistenz**
  - **Strenge Konsistenz --> Komfort**
  - **Änderungen werden gebündelt propagiert**
  - **Optimistische Synchronisierung statt Sperren**

Zielanwendungen

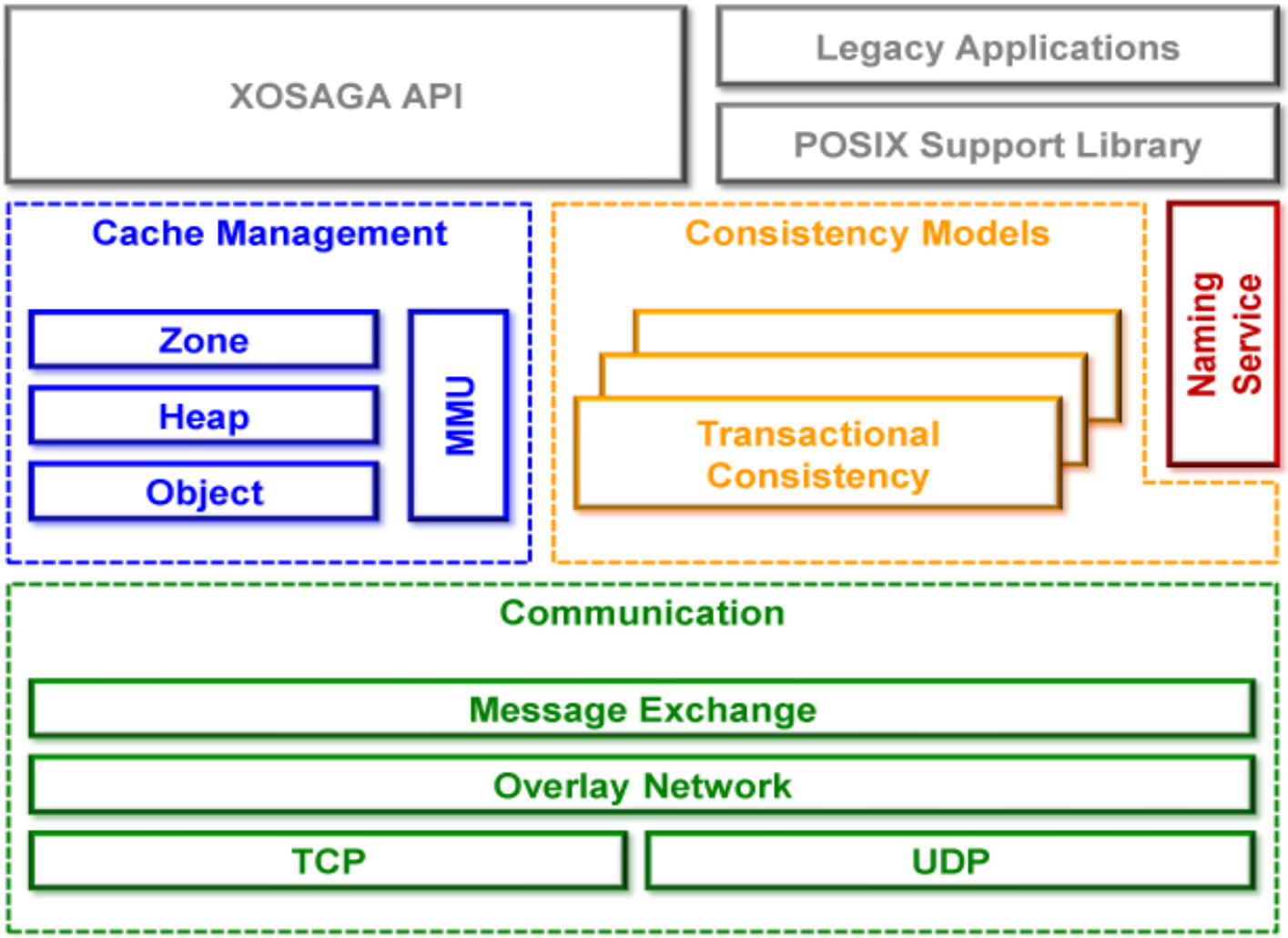
- **Rechenintensive/datenintensive App.**
  - Simulationen, Data Mining, Ray Tracing, … →
  - Daten: Skalare, meist wenige grosse Objekte



- **Interaktive Anwendungen**
  - Mehrbenutzerspiele →
  - Daten: verzeigerte Strukturen, viele kleine Obj.



OSS-Architektur

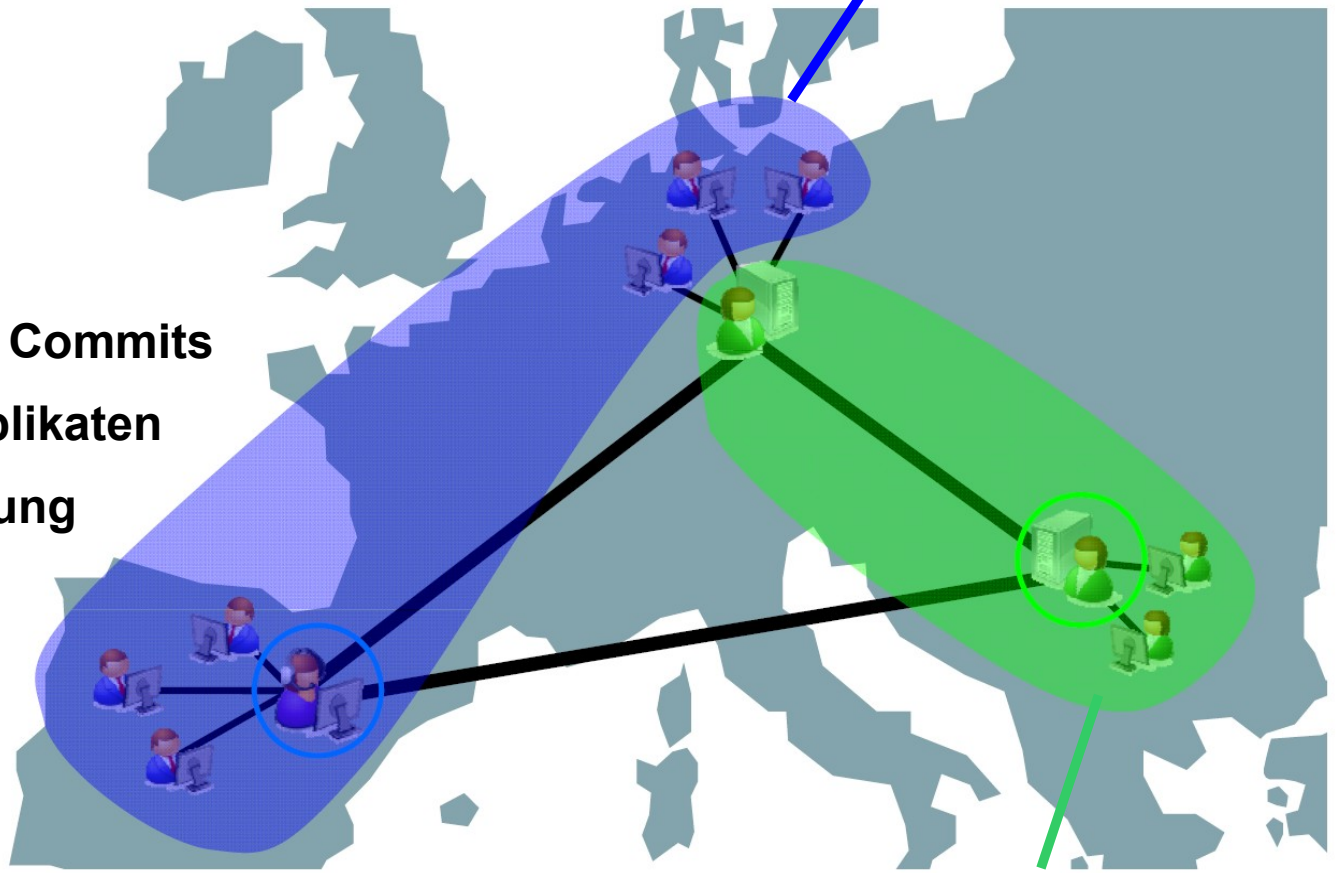


Skalierbare Transaktionen

- **Transaktionale Konsistenzdomänen**
  - Jede Domäne wird separat validiert
- **Lokaler Commit**
  - Falls nur gelesen wurde
  - Oder nur nicht-replizierte Daten verändert wurden
- **Verkettete Transaktionen**
- **Super-Peer Overlay-Netzwerk**

Super-Peer Overlay-Netzwerk

- **Gruppieren geographisch naher Knoten**
- **Aufgaben von Super-Peers**
  - **Validierung von Commits**
  - **Suche nach Replikaten**
  - **Replikatverwaltung**



Programmiermodell

- Grundlegend wie bei TM-Systemen
- Explizite Transaktionen
- Explizite Allokation transaktionaler Daten
- Konflikt-Monitor zur Unterstützung des Programmierers

Programmiermodell

- **Beispiel:**

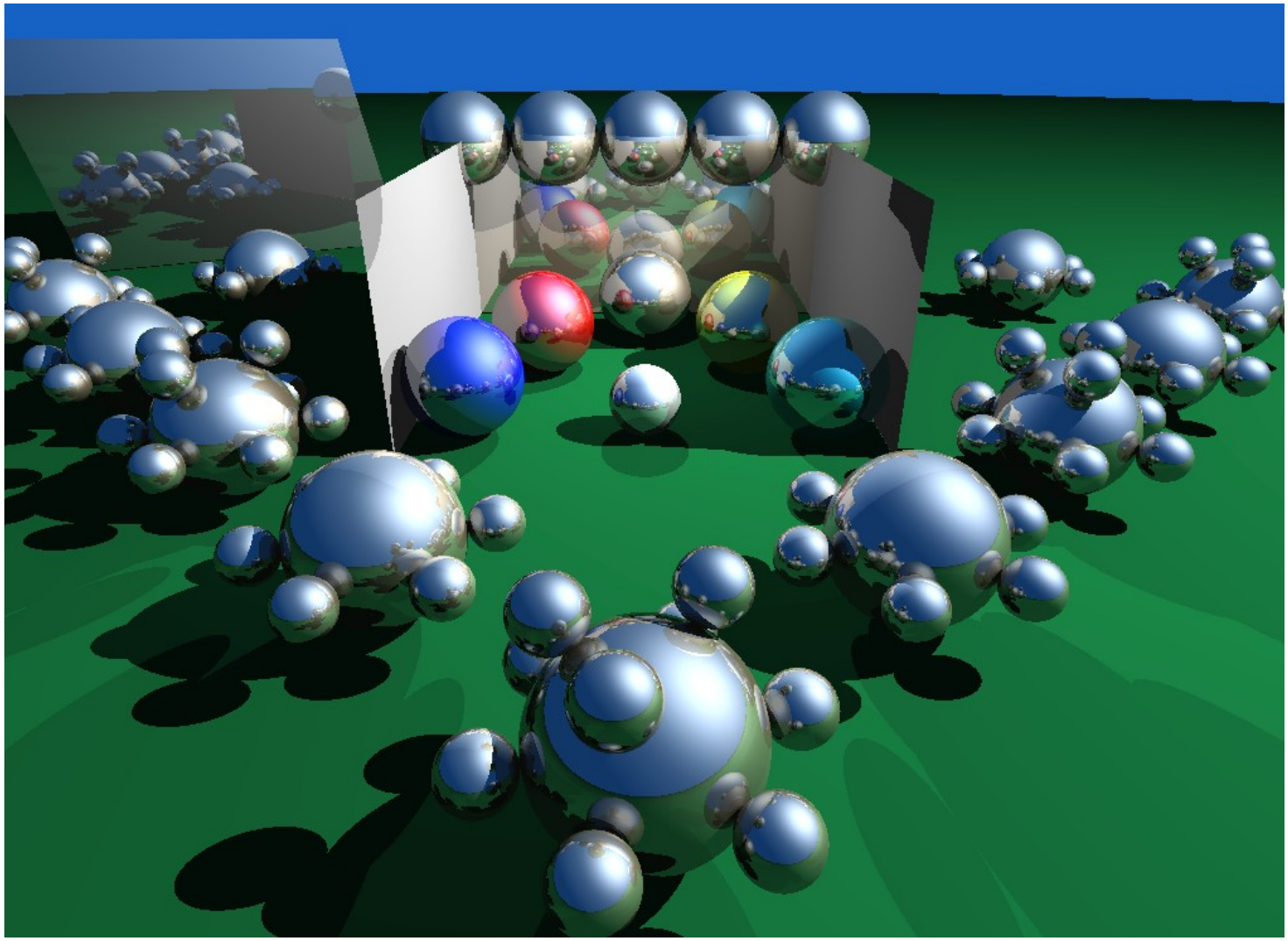
```
int i=5;
Obj obj = oss_malloc(sizeof(Obj),TM);
bot();
   printf("Number %d lives\n",i);
   obj.setPos(12,12);
   i = 8;
eot();
```

3. Experiemente & Leistungsdaten

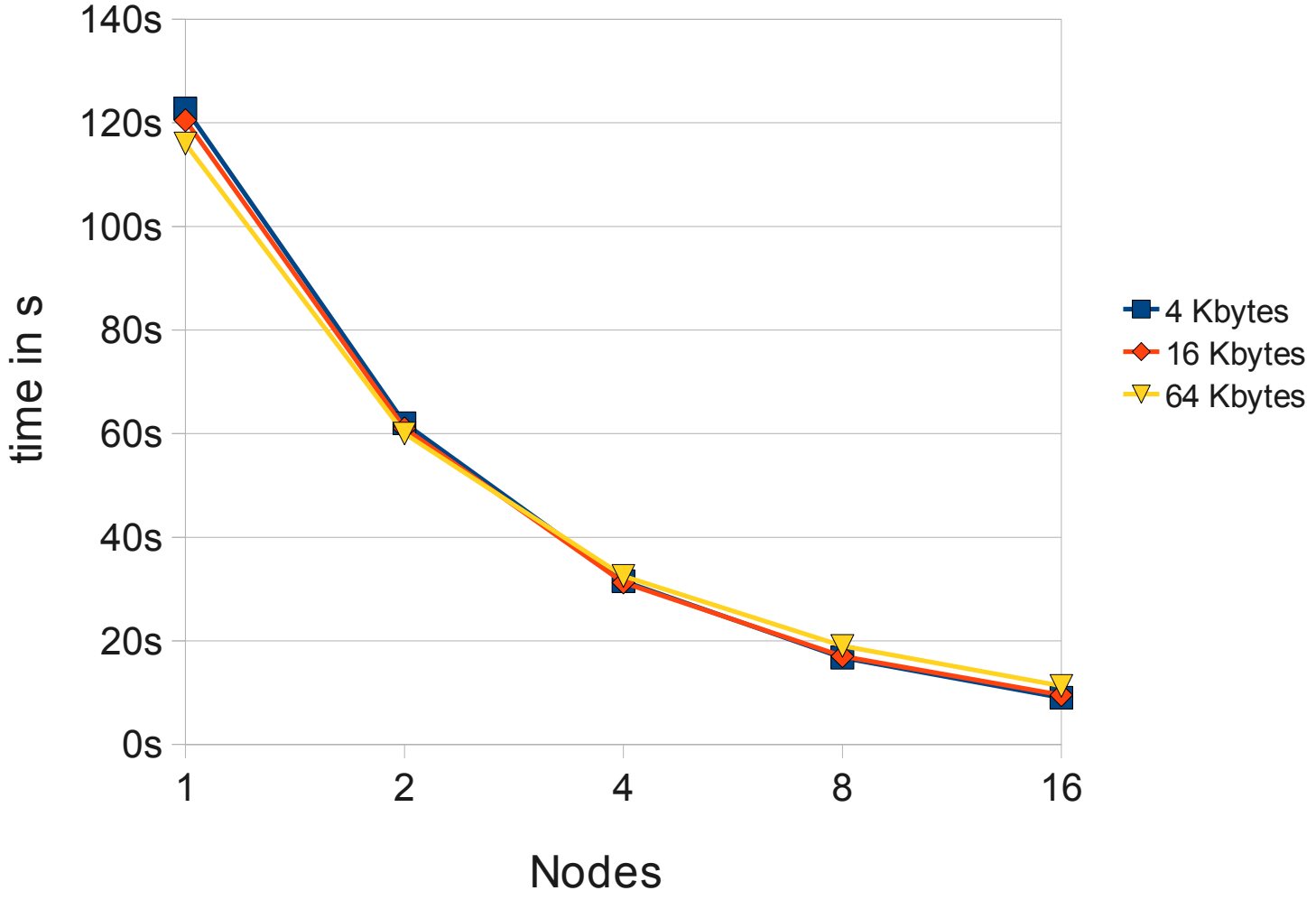
Messumgebung

- Cluster with 16 nodes each with two AMD Opteron CPUs (1,8 GHz), 2 GB RAM Debian Linux 64 (Kernel 2.6.24.3)
- Switched Gigabit Ethernet network
- Zusätzlich Grid'5000

Raytracer

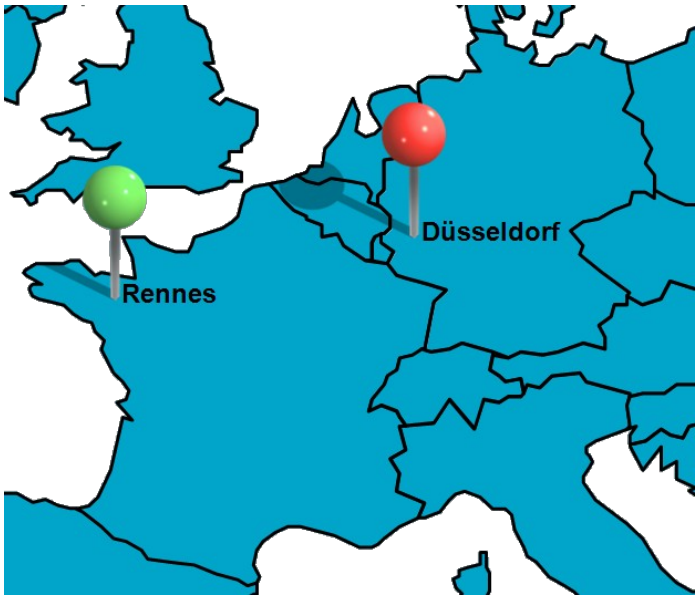


Raytracer


140s
120s
100s
80s
60s
40s
20s
0s
time in s
1
2
4
8
16
Nodes
4 Kbytes
16 Kbytes
64 Kbytes

Wissenheim

- **Getestet mit 2 Knoten:**
  - Einer in Rennes, Frankreich
  - Und einer in Düsseldorf
  - Jeder steuert einen Avatar
- **WAN-Latenz: ~40ms round trip time**
- **Geographische Distanz: ~750km**



Wissenheim

- **Szenengraph verwaltet durch OSS**

- **Synchronisierung:**
  - **Strenge Konsistenz:**
    Änderungen am Szenengraph
    → spekulative Transaktionen
  - **Schwache Konsistenz:**
    Avatar-Bewegung
    → nicht-transaktionale Zugriffe

4. Fazit & Ausblick

Fazit

- **Transaktionaler Speicher (TM) ist nützlich für Thread-Synchronisierung auf Mehrkern-CPUs**
  - **Pro:** weniger Sync.fehler, keine Deadlocks (keine Sperren)
  - **Contra:** Rücksetzbarkeit nicht immer möglich
- **Verteilter TM ist interessant für Cluster und Grid-Systeme**
  - **Pro:** transparente entfernte Zugriffe, strenge Konsistenz, vergleichsweise effizient
  - **Contra:** Commit im Netz, Schattenkopien, Validierung
- **Open Source Code: http://www.xtreemos.eu**

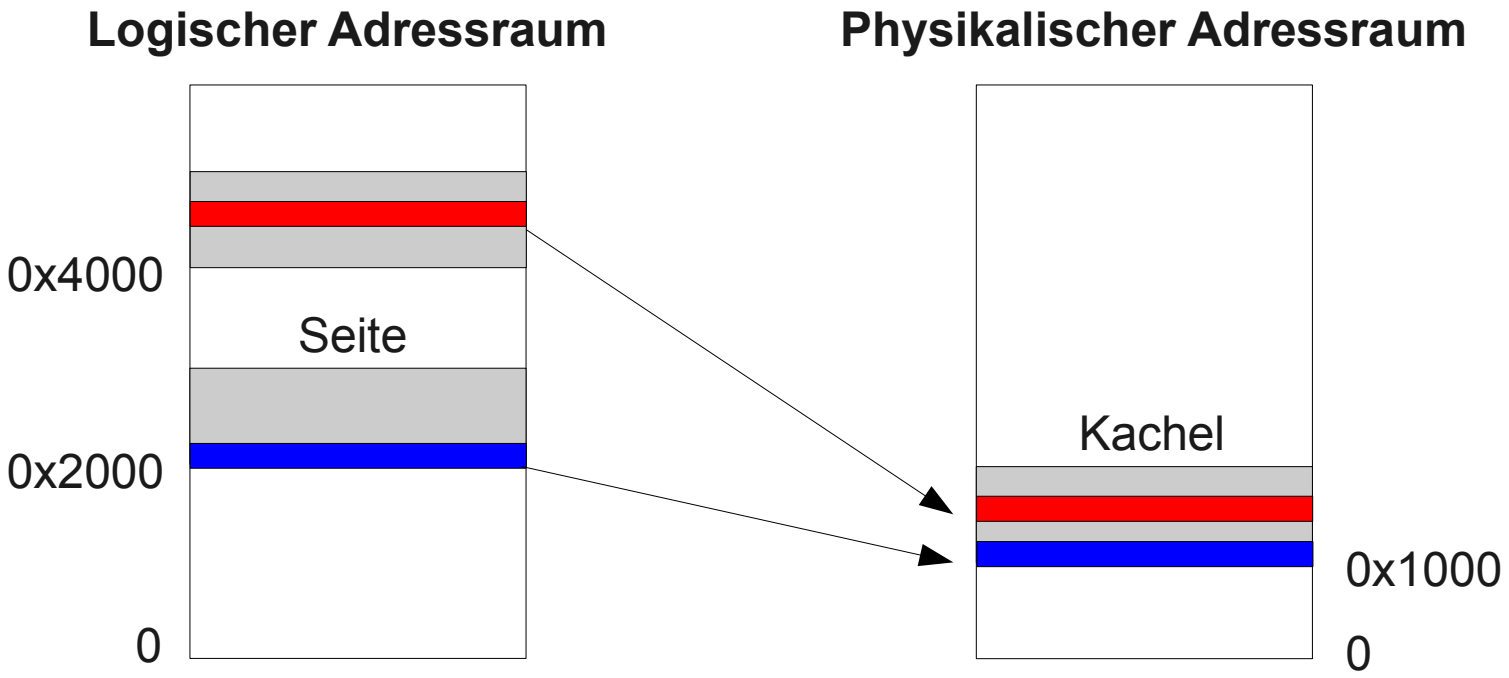
Ausblick

- **Weitere: Anwendungen**
  - **aus dem Bereich Geo- und BioInformatik**
- **Skalierbare Validierungstechniken**
- **Adaptives Cache-Management**
- **Tests auf Grid'5000**

Backup-Folien

Adaptive Konflikteinheitsgröße

- **Millipages zur Vermeidung von False Sharing**
  - **Objektbasierte Zugriffserkennung per MMU**
  - **Aber vielen Seitenfehler & Caching-Effekt geht verloren**



Adaptive Konflikteinheitsgröße

- **Bei einem Seitenfehler auf einer Millipage, alle Objekte/Millipages freischalten**
  - **Entspricht Verhalten bei klassischer Konflikteinheitsgröße von 4 KB**

- **Adaptives Verfahren:**
  - **Aufzeichnen von Millipage-Zugriffsmustern**
  - **dynamisches bündeln von Millipages in Gruppen**

Validierung

- **Nur ein Knoten kann zu einem Zeitpunkt validieren → implementiert durch „zirkulierendes“ Token**
- **Token beinhaltet globalen Commit-Zähler (64-Bit)**
- **Jeder Knoten speichert zuletzt gesehenen Commit in einem lokalen Commit-Zähler**
- **Bei Empfang eines Write-Sets:**
  - **Lokale Replikate invalidieren**
  - **Prüfen, ob eine Seite aus dem Write-Set gelesen wurde, falls ja, laufende Transaktion abbrechen**

Commit

- **Anfordern des Tokens**
- **Nachdem erhalten, Write-Set an alle anderen schicken → First-Wins-Strategie**
  - **Sender wartet nicht auf Bestätigungen → Token sofort freigeben**
  - **Ggf. warten Knoten auf noch ausstehende Commits (Commit-Zähler)**
- **Bem.: Keine verteilten Transaktionen, daher kein aufwändiger 2-Phasen Commit notwendig**

Super-Peer-Commit

- **Nur Super-Peers führen Commits durch**
  → Token „zirkuliert“ nur zwischen wenigen Super-Peers
- **Peers müssen Read- und Write-Set mit Commit-Request an ihren Super-Peer versenden**
- **Wartet ein Super-Peer auf das Token**
  - **So validiert er anhängige Commit-Anfragen gegeneinander**
  - **Allfällige Konflikte handelt er direkt ab**
- **Empfängt er das Token erledigt er mehrere Commits**

Ultra-Peer-Commit

- **Ultra-Peer**
  - Super-Peers wählen einen Ultra-Peer
  - Dieser übernimmt alle Commits
    → Versenden von Write-Sets
  - Validiert alle Anfragen

- **Peers schicken Commit-Request mit Read- und Write-Set direkt an Ultra-Peer oder indirekt über ihren Super-Peer**

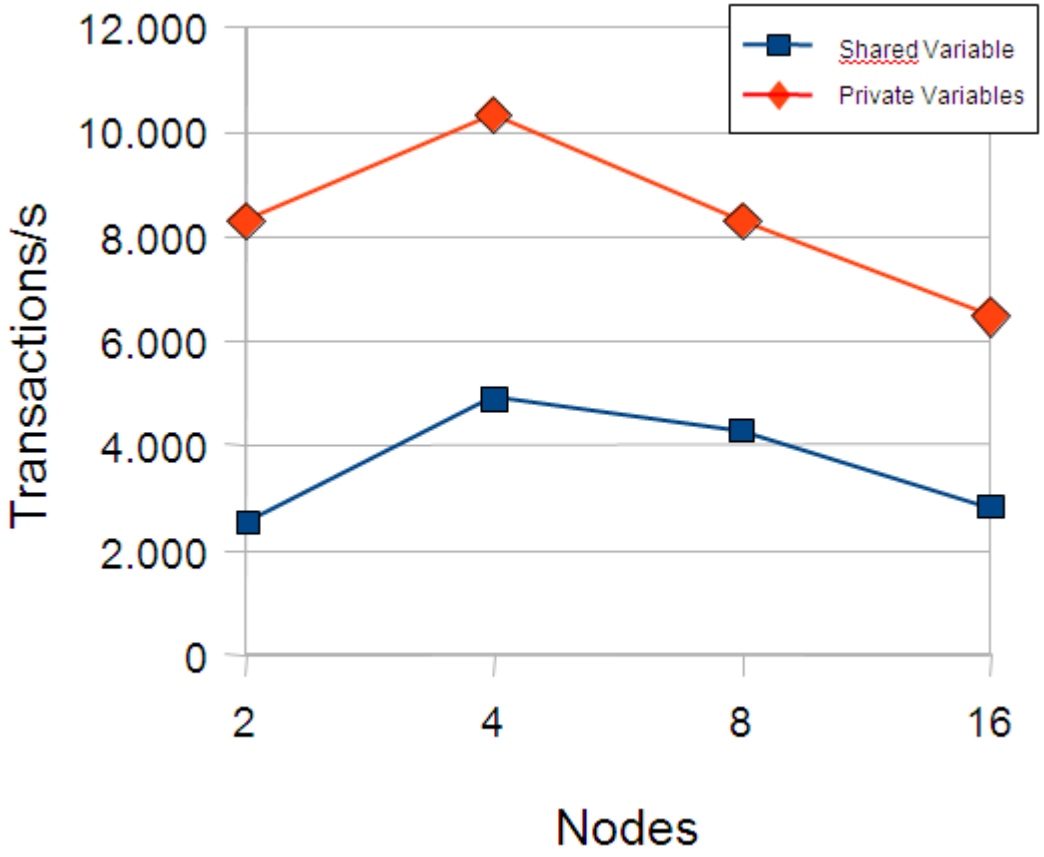
- **Vorteil: Token wird überflüssig**
- **Nachteil: potentieller Flaschenhals**

Synthetisches Zugriffsmuster

- **Worst case: all Knoten inkrementieren eine shared Variable**
- **Best case: all Knoten inkrementieren private Variablen**

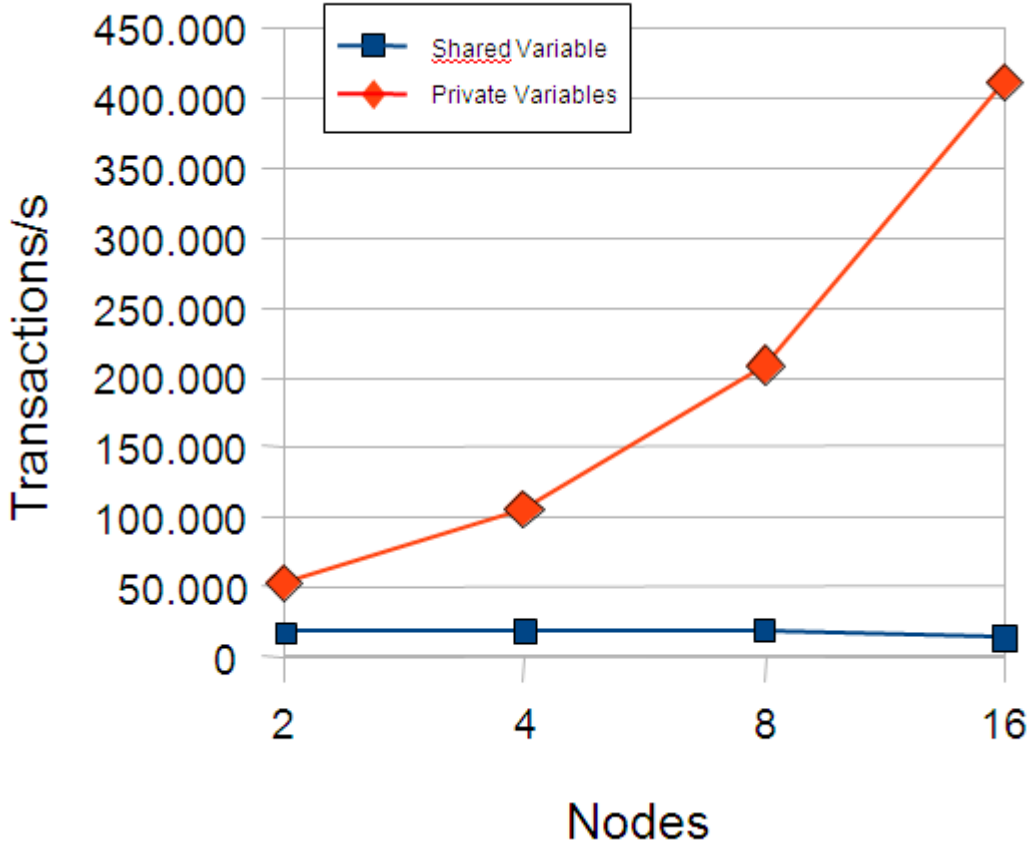
Synthetisches Zugriffsmuster

global commit

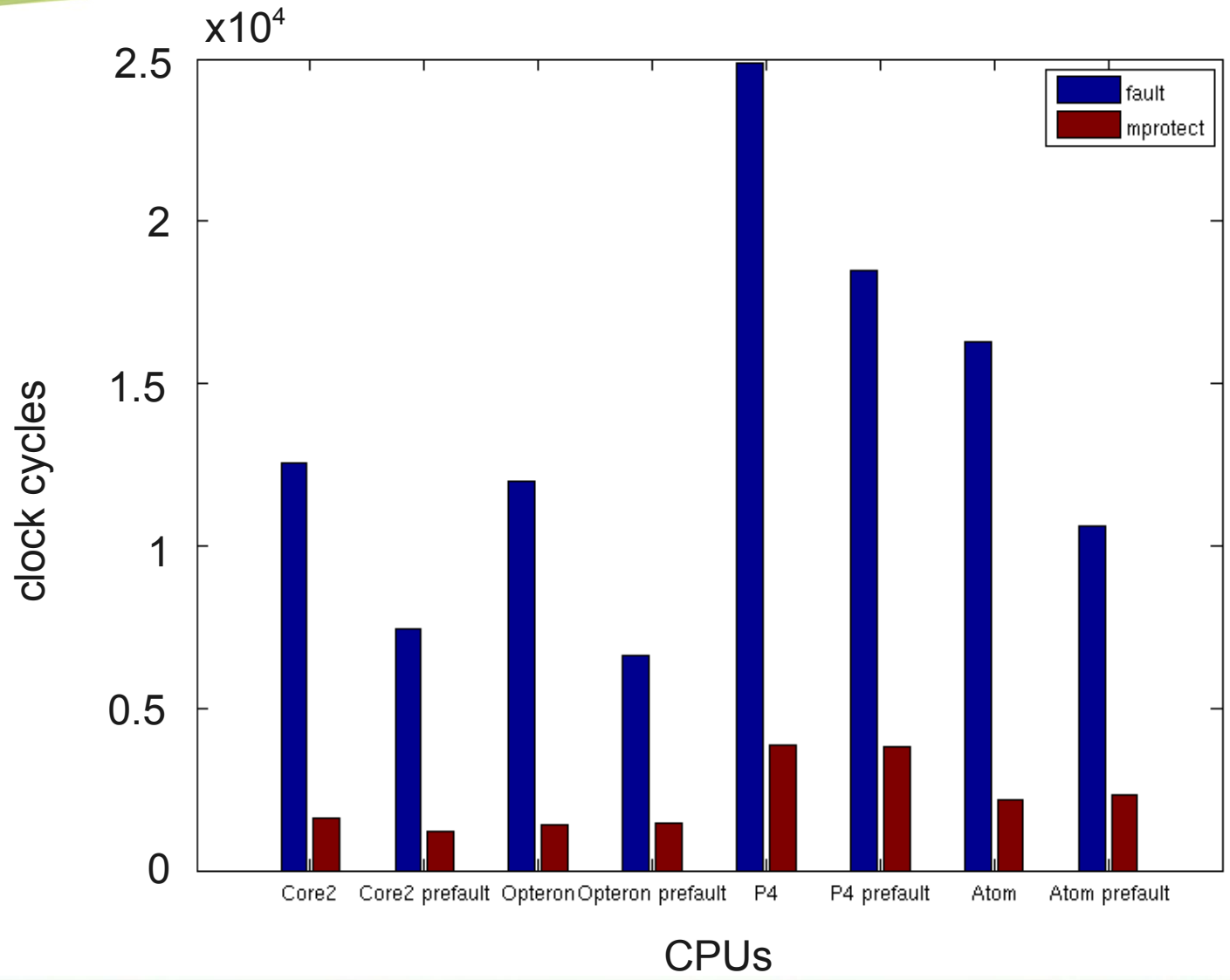


Synthetisches Zugriffsmuster

global/local commit



Kosten der Zugriffserkennung



Zuverlässigkeit

- **Fail-Stop Fehlermodell**
- **Fehlererkennung durch Commit Number (CNR):**
  - Gespeichert im Token
  - wird mit jedem Commit-Paket inkrementiert,
  - jeder Knoten speichert CNR und verschickt diese mit jedem Paket .
- **Transaction History Buffer:**
  - verpasste Write-Sets können bei Bedarf nachgefordert werden,
  - hierzu speichert jeder Knoten Write-Sets in einem Puffer.
- **Weitergehende Fehlertoleranz durch Grid Checkpointing Dienst**