Zuverlässiger verteilter Speicher mit transaktionaler Konsistenz

Dissertation zur Erlangung des Doktorgrades Dr. rer. nat.
der Fakultät für Informatik der Universität Ulm

vorgelegt von

Stefan Martin Frenz
aus Ulm

2006

Amtierender Dekan	Prof. Dr. Helmuth Partsch
Gutachter	Prof. Dr. Peter Schulthess
Gutachter	Prof. Dr. Franz Schweiggert
Tag der Promotion	29. Mai 2006

Für Emma Däubler

Danksagung

Meinen herzlichsten Dank möchte ich Herrn Prof. Dr. Peter Schulthess aussprechen, der mich über viele Jahre hinweg nicht nur bei dieser Arbeit, sondern auch bereits während meines Studiums mit vielen Denkanstößen und mit ständiger Bereitschaft zu produktiven Diskussionen unterstützt hat. Des Weiteren möchte ich mich vielmals bei Herrn Prof. Dr. Franz Schweiggert für die Betreuung im Studium und die Begutachtung dieser Arbeit bedanken.

Bedanken möchte ich mich darüber hinaus herzlich bei Herrn Prof. Dr. Michael Schöttner, der mir stets fachlich sowie menschlich zur Seite stand und dadurch verlässlichen Halt bot. Ebenfalls gilt mein herzlicher Dank Herrn Dr. Moritz Wende für seine Vorarbeiten auf diesem Gebiet sowie für die anregenden Gespräche.

Weiterhin bedanke ich mich bei meinen Kollegen Herrn Dr. Ralph Göckelmann, Herrn Dipl. Inf. Markus Fakler sowie Herrn Dipl. Inf. Michael Sonnenfroh für die zahlreichen Diskussionen und für die angenehme Zusammenarbeit in dem dieser Arbeit übergeordneten Projekt.

Danken möchte ich meinen Eltern Dr. Albrecht und Gertraud Frenz, Frau Gertrud Däubler und meinen Freunden, die mich während des Studiums und der Dissertation begleiteten. Schließlich gilt mein ganz besonderer Dank meiner Verlobten Elisabeth Hahn für das aufgebrachte Verständnis und die kontinuierliche Unterstützung.

Inhaltsverzeichnis

1 Einleitung

In der Abteilung Verteilte Systeme der Universität Ulm wird das verteilte Betriebssystem Plurix für Forschung und Lehre entwickelt, mit dessen Hilfe unter Fortführung der bewährten Konzepte der Oberon-Tradition neue Kommunikationsmodelle, Programmiermethoden und Betriebssystemarchitekturen evaluiert werden. In diesem Kapitel werden relevante Bestandteile dieser Arbeit erläutert, in Bezug zu bestehenden Konzepten gebracht und inhaltlich kurz umrissen.

1.1 Allgemeine Architektur von Betriebssystemen

Die Aufgaben eines Betriebssystems (siehe [TaneMB02], [StalBP03], [ScBoAI03] und [SchuBS04]) sind neben der Bereitstellung von speziellen Systemdiensten die Verwaltung und Vergabe von Betriebsmitteln für Anwendungsprogramme, insbesondere Zuteilung von Rechenzeit, geregelter Zugriff auf Geräte, Verwaltung des logischen und physischen Speichers sowie der Zugang zu Netzwerken.

Herkömmliche Betriebssysteme verfolgen dabei eine strikte Trennung von Kern und Anwendung, um einerseits den Kern vor einem ungültigen Zugriff aus einer Anwendung heraus und andererseits Anwendungen voreinander zu schützen. Des Weiteren liegen die auszuführenden Anwendungen nur in Form von zum Teil modularisierten Dateien vor, deren Referenzen bei der Einlagerung in den Hauptspeicher aufgelöst werden müssen, was zum Teil das Nachladen von Bibliotheken und damit weiteres Binden erforderlich macht.

Die für die jeweiligen Fragestellungen optimierten Protokolle und Mechanismen ermöglichen mit aufwendigen Verwaltungsstrukturen die Kommunikation mit genau einer speziellen Sorte von Gegenstellen, wobei die Protokolle im Regelfall gestapelt sind. Beispielsweise verwendet ein Internet-Browser zur Übertragung eines gesicherten Inhalts HTTP über SSL über TCP über IP, was dann je nach zugrunde liegender Hardware über einem Protokoll für Ethernet oder Token Ring genutzt wird. Der Austausch der Daten unterliegt hierbei unterschiedlichen Güteklassen, so entbindet zum Beispiel TCP das Anwendungsprogramm von einer erneuten Übertragung im Fehlerfall. Für Prozesskommunikation zwischen Knoten stehen unterschiedliche weitere Dienste zur Verfügung wie zum Beispiel RPC, RMI oder Corba. Diese verringern den Aufwand für den Anwendungsprogrammierer gegenüber expliziter Kommunikation deutlich, erfordern jedoch eine unterstützende zusätzliche Schicht, die meistens in Form von Middleware-Systemen vorliegt.

Komplexe Middleware-Systeme wurden als Werkzeug für verschiedene Anforderungen entwickelt, um das Betriebssystem um die für bestimmte Anwendung fehlenden Funktionen zu erweitern. Ziel der Bemühungen im Bereich der Netzwerktechnik ist hier neben der einfacheren Kommunikation für den Anwendungsprogrammierer auch eine einheitliche Schnittstelle auf Knoten mit potenziell unterschiedlichen Betriebssystemen. Beispielhaft soll hier Corba erwähnt werden, dessen Ziel die Erleichterung von sprach- und plattformunabhängigem Datenaustausch ist.

Persistenz (Dauerhaftigkeit) wird bei herkömmlichen Betriebssystemen über Dateien realisiert, die auf Medien wie einer Festplatte, einer CD oder einem USB-Stick abgelegt werden. Diese Medien können, im Gegensatz zum Hauptspeicher, Daten auch ohne Stromversorgung über einen längeren Zeitraum sicher halten, wobei die Länge dieses Zeitraums deutlich von den physikalischen Randbedingungen abhängt. So ist der Inhalt einer gebrannten CD durch UV-Strahlung und der einer Festplatte durch starke Magnetfelder gefährdet.

Wie beim Versenden von Daten zwischen Knoten müssen Daten beim Speichern und Laden serialisiert beziehungsweise zum Aufbau der benötigten Datenstrukturen interpretiert werden, wenn im Hauptspeicher nicht bereits ein linearer Puffer mit allen relevanten Daten vorliegt. Bei den meisten Anwendungen oder Datenbanken ist es jedoch weder möglich noch sinnvoll, mit einem solchen Puffer zu arbeiten, da eine Programmierung mit Verweisen zwischen Objekten erheblich einfacher und natürlicher ist. Ohne spezielle Konvertierung der Verweise kann die Gültigkeit solcher Verweise jedoch bei herkömmlichen Betriebssystemen nach Wiederherstellung der Daten nicht garantiert werden. Zur Vorhaltung eines Datensatzes sind also (mindestens) zwei gleichwertige und ineinander konvertierbare Formate erforderlich. Dabei ist der Aufwand zur Konvertierung häufig hoch und fällt für jedes Format-Paar erneut an. Bei der Änderung eines einzelnen Formats einer Anwendung sind unter Umständen Anpassungen an allen anderen Formaten sowie den dazugehörigen Konvertierungsprogrammen erforderlich.

1.2 Architektur verteilter Betriebssysteme

Verteilte Betriebssysteme (siehe [TaStDS02]) sind zum Einsatz in einem Rechnerverbund konzipiert und enthalten im Vergleich zu nicht-verteilten Betriebssystemen eine zum Teil deutlich ausgefeiltere Schnittstelle zur Kommunikation zwischen Rechnern. Oftmals können Objekte oder ganze Speicherseiten für den Anwendungsprogrammierer transparent zugegriffen und übertragen werden, wobei die Synchronisierung je nach System explizit durch den Programmierer erfolgen muss oder implizit durch das System vorgenommen wird.

Ziel der Systeme ist neben der Vereinfachung der Kommunikation auch die Verwaltung von verteilbaren Betriebsmitteln, jedoch können meist nicht alle Betriebsmittel transparent für den Programmierer verteilt werden, so dass hier eine explizite Kontrolle notwendig ist. Von großem Interesse ist üblicherweise die verteilte Verwaltung von CPU-Zeit und Hauptspeicher, um somit die verfügbaren und sich dynamisch ändernden Ressourcen optimal ausnutzen zu können.

Um den Anwendungsprogrammen auf allen Knoten eine einheitliche Basis anzubieten, ist ein Minimum an kompatiblen Schnittstellen erforderlich. Mit steigender Transparenz wird für diese Schnittstellen eine immer einheitlichere Installation benötigt; eine vollständig einheitliche Sicht für die Anwendung wird beim Single System Image realisiert [BuCoSS01]. Hierbei wird der Anwendung auf jedem Rechner eine identische Sicht auf die zur Verfügung stehenden Ressourcen, insbesondere Bibliotheken und Laufzeitstrukturen, ermöglicht. Im Falle einer vollständig identischen Installation aller beteiligten Rechner ist eine solche einheitliche Sicht zwar garantiert, jedoch ist es mitunter aufwendig oder unmöglich, alle Knoten mit gleichwertigen Bibliotheken auszurüsten. Eine mögliche Alternative zu dieser Art Pflege vieler Knoten besteht darin, statt vieler Systeme mit einheitlichen Bibliotheken auch das Betriebssystem zu verteilen. Somit kann der administrative Aufwand in Grenzen gehalten und dennoch der Anwendung ein Single System Image geboten werden [GoecSB05].

1.3 Verteiltes Betriebssystem Plurix

Das an der Universität Ulm entwickelte verteilte Betriebssystem Plurix verwendet einen seitenbasierten verteilten virtuellen Speicher (Distributed Shared Memory, DSM, siehe zum Beispiel [KeedMS89], [MoRaSD91], [CoCoES96] und [WiMuED96]), der als Halde (engl. Heap) organisiert ist. Der speziell entwickelte Compiler erzeugt aus Java-Quellcode direkt ausführbare Objekte im Speicher, deren Abbild mit Hilfe eines minimalen Bootladers (typischerweise weniger als 400 Bytes Maschinencode) geladen und ausgeführt werden kann.

Die typsichere Sprache Java ermöglicht den Aufbau eines stabilen Systems ohne Trennung von Kern und Anwendungen oder Anwendungen untereinander [WiGuPO92]. Der Austausch von Daten zwischen Kern und Anwendungen und ebenfalls zwischen Anwendungen untereinander geschieht ausschließlich über Objekte. Da Java keine Möglichkeiten zur Manipulation von Zeigern bietet, kann durch kontrolliertes Weitergeben von Referenzen Zugriffsschutz für die Objekte und somit für die Daten gewährleistet werden.

Die Verteilung wird unter Plurix (siehe [GoecSB05]) mit Hilfe der Verteilung der gesamten Halde erreicht, so dass alle Knoten im Cluster eine identische Sicht auf die Halde haben. Statt wie in anderen DSM-Systemen einzelne Zugriffe auf Seiten zu protokollieren und zu publizieren, werden Transaktionen eingeführt, deren Ausführung entweder vollständig erfolgreich ist oder vollständig zurückgesetzt wird. Somit ergibt sich eine semantische Gruppierung von auszuführendem Code, dessen Verhalten mit den von Datenbanken bekannten ACID-Eigenschaften (siehe Kapitel 2.1 und insbesondere Kapitel 2.1.2) vergleichbar ist:

1. Alle Änderungen einer Transaktion erfolgen atomar.
2. Eine Transaktion überführt einen konsistenten Zustand in einen neuen konsistenten Zustand.
3. Alle lokalen Änderungen einer Transaktion sind bis zu deren Bestätigung isoliert.
4. Die Änderungen einer Transaktion sind dauerhaft (persistent).

Das bisherige Plurix-System kann Daten zwar unter Erfüllung der Punkte 1-3 in einem Cluster verteilen, jedoch nicht deren Dauerhaftigkeit auf einem nicht-flüchtigen Speichermedium sicherstellen. Diese Arbeit stellt unter anderem (siehe auch Kapitel 1.5) ein zum Einsatz eines Pageservers verwendbares Verfahren zur Sicherstellung der Dauerhaftigkeit der Daten im DSM vor, und zwar sowohl im fehlerfreien Betrieb als auch beim Ausfall eines oder mehrerer Knoten.

Die Strukturierung der Halde bleibt aufgrund des orthogonalen Sicherungsverfahrens (siehe Kapitel 2.3.1) für den Pageserver transparent. Daher ist die momentan implementierte Integration des Kerns in den verteilten Speicher aus Sicht des hier vorgestellten Verfahrens zur Sicherung der Halde irrelevant. Hinsichtlich der zu erwartenden Performance kann der Kern im verteilten Speicher auch Nachteile bringen, wie in Kapitel 4.2 gezeigt wird. Unberührt davon ist die Anforderung der Rekonstruierbarkeit aller für einen Neustart relevanten Informationen (siehe Kapitel 4.3).

Plurix arbeitet als natives Betriebssystem ohne jegliche zugrunde liegende Softwareschicht, so dass hier zu Gunsten einer homogenen Architektur vollständig auf Middleware verzichtet werden konnte. Dies ist auch für die Lehre hilfreich, für diese Arbeit jedoch zweitrangig, da sich die vorgestellten Verfahren leicht auf andere Systeme mit vergleichbaren Randbedingungen hinsichtlich Konsistenz übertragen lassen.

1.4 Stand der Technik

Die derzeit eingesetzten Cluster stellen hohe Rechenleistungen für bestimmte Anwendungen zur Verfügung und können mit Hilfe spezieller Software und durch Hardware-Redundanz Ausfallsicherheit für Dienste in gewissem Umfang anbieten. Beispielsweise in Heartbeat (siehe

[RobeLH99]) wird die Verfügbarkeit eines Dienstes dadurch sichergestellt, dass ausgefallene Maschinen durch sogenannte Backup-Maschinen ersetzt werden. Im „Tandem NonStop System“ (siehe [EvoyAT81]) hingegen wird für kleine Cluster bis 256 Knoten mit jeweils zwei bis 16 Prozessoren Ausfallsicherheit garantiert, indem für jede nach außen angebotene Funktion mindestens zwei unabhängige Module bereitgestellt werden. Somit kann beim Ausfall eines Moduls das andere identische Modul sofort die Aufgaben des fehlerhaften Moduls übernehmen.

Alternativ dazu hat sich ein allgemeinerer Ansatz zur Bereitstellung von Fehlertoleranz entwickelt: Durch die Erstellung von Schnappschüssen während des fehlerfreien Betriebs kann ein System nach dem Auftreten eines Fehlers auf einen früher gesicherten Zustand zurückgesetzt werden. Dadurch gehen zwar die seit der Erstellung dieser Sicherung geänderten Daten verloren, jedoch ist der Hardware-Bedarf gegenüber vollständiger Redundanz erheblich reduziert. Eine Vielzahl solcher Systeme wird in [ElAlSR96] beschrieben, wobei die Komplexität zur Bildung eines konsistenten Zustands auf der Ebene des Netzwerks aufwendig ist, da Nachrichten weder verloren gehen noch nach einer Rücksetzung doppelt auftauchen dürfen (Orphan Messages). Alternativ zu auf Nachrichten basierenden Systemen haben sich (siehe [KeedMS89]) virtuelle verteilte Speicher (VVS) entwickelt, die den Zugriff auf ein Datum ortstransparent ermöglichen und somit die Kommunikation zwischen Anwendungen deutlich erleichtern. Die Konsistenzmodelle für VVS (siehe [MosbMC93]) sind für verschiedene Anwendungen optimiert und typischerweise entweder eher restriktiv und einfach zu bedienen oder relaxiert und potenziell inkonsistent.

Ein konsistenter Zustand einer laufenden Anwendung besteht jedoch unabhängig vom Konsistenz-Modell nicht nur aus deren Netzwerk-Kommunikation und ihren Daten, sondern hängt auch von der transitiven Hülle ihres Datenpfades in anderen Anwendungen und im lokalen Betriebssystem ab. Die dadurch entstehende Menge an Daten ist immens und erhöht den Zeitaufwand und Speicherbedarf für einen Schnappschuss erheblich. Die veröffentlichten Erkenntnisse werden daher in bestehenden Clustersystemen kaum berücksichtigt, da einerseits der Aufwand zur Umsetzung der theoretischen Vorschläge zu hoch ist und andererseits die im Cluster laufenden Betriebssysteme nicht zur Unterstützung von Fehlertoleranz geeignet sind. Als Beispiel sei hier das Kerrighed System genannt, das die Erstellung eines Schnappschusses durch Erweiterung des Linux-Kerns (siehe [MoLoKS03] und [MoGaTE04]) realisiert. Durch den eingesetzten VVS ist der Datenzugriff feingranular und transparent möglich, so dass sich die Migration von Prozessen und die Erstellung von Schnappschüssen deutlich vereinfacht. Dennoch ist die Erstellung von Schnappschüssen durch die von Linux vorgegebene Architektur mit großem Aufwand verbunden, da die zur Anwendung gehörenden Daten im Kern bei der Erstellung eines Schnappschusses

berücksichtigt werden müssen. Um die Kern-Zustände extrahieren und später wiederherstellen zu können, wird der so genannte Ghost-Mechanismus verwendet (siehe [VaLoGP05]), wofür tiefgreifende Änderungen des Linux-Kernels notwendig sind. Die Herausforderung liegt hier vor allem in der Wahl und Implementierung geeigneter Schnittstellen im Kern und außerdem in der Pflege des Kerrighed-Systems, da die Modifikationen am Kern für jede zu unterstützende Kern-Version manuell anzupassen sind.

Parallel zur Entwicklung der Betriebssysteme besteht seit geraumer Zeit durch die Forschungstätigkeiten auf dem Gebiet der Datenbanken das Konzept der Transaktionen (siehe [HaRePT83], [HuCaEI93], [DadaVD96]), das die Veränderung von Daten in mehrere Phasen einteilt und bestimmte Bedingungen an eine Transaktion knüpft (siehe Kapitel 2.1.2). Die auf diesem Feld gewonnenen Erkenntnisse konnten im Rahmen des dieser Arbeit übergeordneten Plurix-Projekts der Universität Ulm auf verteilte Betriebssysteme für PC-Cluster übertragen werden.

1.5 Beitrag und Abgrenzung dieser Dissertation

Beim Betrieb von Rechnern sind Ausfälle von Hardware, Fehler in der Software sowie nicht reproduzierbare Störungen in der Kommunikation zu erwarten. Stützt sich die Funktionsfähigkeit eines aus vielen Rechnern bestehenden Gesamtsystems auf die Verfügbarkeit aller Teilkomponenten, ist dieses Risiko sogar noch deutlich erhöht. Um die Zuverlässigkeit dennoch ausreichend gewährleisten zu können, ist es wünschenswert, auftretende Fehler zu erkennen und angemessen darauf zu reagieren. Hierfür ist es notwendig, Fehler zu charakterisieren (siehe Kapitel 4.1) und, sofern vorhanden, geeignete Algorithmen zur möglichst verlustfreien Wiederaufnahme der gewünschten Arbeit zu finden.

Die meisten Verfahren zur Behandlung von Fehlern sind jedoch auch im fehlerfreien Betrieb mit zum Teil erheblichen Kosten verbunden und nur auf eine sehr kleine Menge von Fehlern anwendbar. Hier ergibt sich der Bedarf einer unter Umständen nicht völlig verlustfreien, dafür aber möglichst generischen, schnellen und vorzugsweise schlanken Alternative, die im fehlerfreien Betrieb geringen Overhead aufweist. Die in dieser Arbeit vorgestellten Konzepte, insbesondere in Kapitel 3.3 und 4.3, ermöglichen insgesamt eine Implementierung, welche diesen Anforderungen für transaktionale Konsistenz genügen kann.

Diese Arbeit stellt ein leichtgewichtiges Verfahren zur seitenbasierten Unterstützung von Persistenz in einem verteilten Betriebssystem mit transaktionaler Konsistenz vor und erläutert die erforderlichen Grundlagen im Bereich verteilter Systeme, wobei auch auf die Synergien zwischen

transaktionaler Konsistenz und möglichen Sicherungsverfahren eingegangen wird. Der Lösungsraum für einen schnellen Wiederanlauf nach einem behebbaren Fehler wird dabei ebenso durchleuchtet wie derjenige für Möglichkeiten, modifizierte Seiten im fehlerfreien Betrieb elegant einzusammeln und schnell zu sichern. Für die besonders zu berücksichtigenden Zustände von Kern und Gerätetreiber werden Voraussetzungen erarbeitet, mit deren Hilfe ein konsistenter Betrieb auch nach vollständigem Ausfall eines oder mehrerer Knoten gewährleistet werden kann.

Die vorgestellten Verfahren wurden im verteilten Betriebssystem Plurix mit verteiltem virtuellem Speicher getestet, jedoch sind sie von ihrer Anwendbarkeit nicht auf dieses System beschränkt. Allgemeine Modelle zur Architektur eines für schnelle Systemwiederherstellung (engl. Recovery) konzipierten Betriebssystems werden unabhängig von der aktuellen Plurix-Implementierung aufgezeigt und begründet, wobei die jeweiligen Vor- und Nachteile der einzelnen Verfahren beleuchtet werden.

Grundsätzlich wird bei den vorgestellten Verfahren dieser Arbeit die Gutmütigkeit von beteiligten Stationen vorausgesetzt. Sicherungen zur eindeutigen Identifizierung und Authentifizierung sind für manche Anwendungen oder Umgebungen sinnvoll und zusätzlich zu den hier vorgestellten Verfahren implementierbar. Dies ist jedoch ebenso wie die Entwicklung von ausgefeilten Strategien zur Erkennung von Fehlern nicht grundlegender Bestandteil dieser Arbeit.

Die Verwandtschaft der hier vorgestellten und unter Plurix eingesetzten Transaktionen zu denjenigen der Datenbanken wird eingehend beleuchtet und insbesondere die Vergleichbarkeit in Bezug auf Dauerhaftigkeit (Persistenz) herausgearbeitet.

2 Checkpointing mit transaktionaler Konsistenz

2.1 Transaktionale Konsistenz

Wie in jedem verteilten System sind Fragen der Konsistenz und Synchronisierung von zentraler Bedeutung. In diesem Kapitel werden die relevanten Grundlagen aus Sicht des Gesamtsystems dargelegt.

2.1.1 Überblick über Konsistenzmodelle

Beim Entwurf eines verteilten virtuellen Speichers müssen auch Fragen zum verwendeten Konsistenzmodell beantwortet werden. Bekannte Verfahren für strenge Konsistenz (siehe [WendKV03] und [BeGoCC80]) lassen sich in drei Kategorien unterteilen:

1. Pessimistischer Ansatz: Durch vorsorgliche Sperren wird jeglicher gleichzeitige Zugriff, der zu Inkonsistenz führen kann, ausgeschlossen.
2. Optimistische Synchronisierung: Alle Zugriffe werden vorerst gewährt und eventuell entstandene Konflikte werden durch Zurücksetzen von einzelnen Operationen oder von Gruppen von Operationen aufgelöst.
3. Ordnung mit Zeitstempel: Innerhalb eines Zeitintervalls auftretende Zugriffe werden gepuffert, unter Beachtung von Zeitstempeln sowie Abhängigkeiten geordnet und erst dann ausgeführt.

Klassischer Vertreter eines Modells mit pessimistischem Ansatz ist wechselseitiger Ausschluss für Schreiboperationen. Hier können zwar beliebig viele Kopien eines Objektes oder einer Seite existieren, die gelesen werden, jedoch muss für einen Schreibzugriff erst eine Sperre gesetzt werden. Erst nach Freigabe der Sperre werden andere Teilnehmer wieder berechtigt, eine Sperre für diese Seite zu erwerben. Dieses Verfahren garantiert, dass zu jedem Zeitpunkt höchstens ein Schreiber pro Objekt oder Seite existiert. Eine direkte Erweiterung dieses Verfahrens findet zum Beispiel in Multiprozessorsystemen Anwendung, wobei hier bei Erwerb einer Sperre gleichzeitig alle noch im Umlauf befindlichen Kopien invalidiert und neue Kopien erst nach Freigabe der Sperre zugelassen werden. Somit ist zusätzlich garantiert, dass keine ungültigen Kopien existieren. Ungültige System- oder Gerätezustände lassen sich durch längeres Halten von Sperren leicht vermeiden.

Bei optimistischer Synchronisierung (siehe [KuRoOM81]) werden Kopien sowie konkurrierende Zugriffe nicht von vornherein ausgeschlossen, sondern bei schreibenden Zugriffen nachträglich geprüft, ob Konflikte vorliegen. Diese werden durch Zurücksetzen einzelner

Operationen aufgelöst, was die Rücksetzbarkeit von Operationen beziehungsweise die Konfliktfreiheit von nicht rücksetzbaren Operationen impliziert. Für den Hauptspeicher kann Ersteres sehr einfach durch Schattenkopien oder Logs realisiert werden, beim Zugriff auf Geräte ohne Rücksetzungsfunktion wie zum Beispiel Druckern ist jedoch durch geeignete Maßnahmen auf Konfliktfreiheit zu achten (siehe [OzVaPD91] sowie Kapitel 2.1.3 und 2.1.4).

Bei Verfahren mit Ordnung anhand von Zeitstempeln werden alle Zugriffe durch eine zentrale Instanz geprüft. Dabei wird, soweit dies möglich ist, Parallelität beibehalten, jedoch muss bei konkurrierenden Zugriffen auf die gleichen Daten eine konsistente Reihenfolge ermittelt werden, was unter Umständen zu einer Sequentialisierung aller Operationen oder einer Teilmenge davon führt. Dieses Verfahren findet beispielsweise bei Datenbanken Verwendung, wo Anfragen an einen Server geschickt werden, der somit als verwaltender Knoten über alle benötigten Daten verfügt.

Außer diesen Konzepten mit strenger Konsistenz werden auch relaxierte Modelle (für einen Überblick siehe [MosbMC93]) verwendet, wenn Daten unter bestimmten Bedingungen auch veraltet sein dürfen. Hierbei muss der Programmierer selbst an geeigneten Stellen im Programmablauf explizit für eine Synchronisierung sorgen, wenn er auf garantiert aktuelle Daten zugreifen möchte. In der Literatur (siehe dazu auch [TaStDS02], [MoRaSD91] und [MiRaSC95]) werden Modelle mit sequenzieller, kausaler, schwacher und offener Konsistenz diskutiert. Alle Konsistenzmodelle lassen sich aus Sicht der Persistenz auf eines der obigen strengen Modelle abbilden, indem entweder für jede Anfrage eine explizite Synchronisierung vorgenommen wird oder, wenn auf eine solche Synchronisierung verzichtet werden soll, alle gesicherten Daten mit den für das Konsistenzmodell relevanten Zusatzinformationen versehen werden.

2.1.2 Semantik von Transaktionen

Zur Spezifikation von Konsistenz können auch Transaktionen verwendet werden, deren in der Einführung erwähnte Semantik die Einhaltung der ACID-Eigenschaften vorschreibt (siehe auch [HaRePT83], [HuCaEI93], [DadaVD96], [LiCaPP99], [FeShPD99]). Transaktionen gehen über die Definition von Konsistenz, in diesem Fall transaktionaler Konsistenz, jedoch weit hinaus:

1. Atomicity: Die auch als „all-or-nothing“ und „failure-atomicity“ bezeichnete Eigenschaft beschreibt den Umstand, dass Transaktionen entweder vollständig ausgeführt werden (Erfolg, „commit“) oder keinerlei Änderungen des Gesamtsystems hinterlassen (Abbruch, „abort“).
2. Consistency: Im klassischen Fall überführt eine Transaktion das Gesamtsystem von einem konsistenten Zustand in einen neuen konsistenten Zustand. Systeme mit zwischenzeitlichem

Commit (siehe [BlacUT90]) sind möglich, jedoch muss die Konsistenz in diesen Systemen vom Programmierer selbst gewährleistet werden.

3. Isolation: Während der Laufzeit einer Transaktion ändern sich ihre Eingabedaten nicht und ihre Zwischenergebnisse sind für andere Transaktionen nicht sichtbar. Insbesondere resultiert hieraus die Serialisierbarkeit (auch „concurrency-atomicity“ genannt) aller Transaktionen in einem System. Die Resultate nebenläufig ausgeführter Transaktionen müssen also auch von in serialisierter Reihenfolge ausgeführten Transaktionen erzielt werden können (siehe auch [WendKV03]). Ein außerhalb des Gesamtsystems befindlicher Betrachter hat auch bei nebenläufigen Transaktionen den Eindruck von seriellen Operationen.
4. Durability: Die Ergebnisse einer erfolgreichen Transaktion müssen dauerhaft sein, also auch Fehler überleben (siehe auch [StanDO04]). Dies erfordert für die Praxis einerseits die Betrachtung möglicher Fehler und andererseits die Spezifikation von zu tolerierenden Fehlern. Je nach Anwendung und daraus resultierenden Anforderungen an die Dauerhaftigkeit können diese unterschiedlich sein, was eine für den jeweiligen Fall angepasste Implementierung nach sich zieht. In Datenbanksystemen wie Oracle wird die Sicherung auf einem nicht-flüchtigen Speicher als notwendig und ausreichend angesehen. Verschärft wird dies bei Buchungssystemen in Geldinstituten mit redundanter Sicherung, abgeschwächt für höheren Durchsatz (Entkopplung von Transaktion und Datensicherung) mit verzögerten Schreibzugriffen wie zum Beispiel bei Verwendung von ISAM-Tabellen unter MySQL (siehe [WeThMT03]).

Verschiedene Anwendungen und Umgebungen erfordern eine Anpassung der für Transaktionen vorgesehenen Eigenschaften. Beispielsweise können Konsistenz und Dauerhaftigkeit beim Zugriff auf Datenbanken mit mobilen Geräten aufgrund der Randbedingungen (siehe [GrasCA04]) nicht immer vom System sichergestellt werden. Selbst bei möglicher Erfüllung der ACID-Eigenschaften kann es für bestimmte Anwendungen sinnvoll sein, durch Aufweichen der ursprünglich sehr strikten Bedingungen Optimierungen zu erreichen (siehe [RomaCA01] und [PuChAP93]).

Eine Verfeinerung von Transaktionen besteht beispielsweise in ihrer Schachtelung („Nested Transactions“, siehe [ElMoNT81]), die unter anderem in [GrayTC81] beschrieben ist. Das klassische Beispiel ist darin der Reisende auf der Suche nach Hin- und Rückflug, Hotel und Mietwagen, der diese vier Buchungen nur als Gesamtheit akzeptieren möchte. Derartige Verfeinerungen sind für den Programmierer vorteilhaft und beim Design eines Systems zu berücksichtigen. Aus Sicht der Dauerhaftigkeit sind sie jedoch irrelevant, da sie ihre Ergebnisse erst nach Abschluss der übergeordneten Transaktion publizieren dürfen (siehe [DadaSR82]) und somit

aus Sicht der Dauerhaftigkeit auf die übergeordnete Transaktion abgebildet werden (siehe [HaRePT83]).

2.1.3 Kollisionsauflösung für Transaktionen

Um Transaktionen abschließen zu können, wird üblicherweise ein Zwei-Phasen-Commit-Protokoll verwendet. Die erste Phase widmet sich der Validierung, die zum Abbruch einer oder mehrerer Transaktionen führen kann. Findet in dieser Phase kein Abbruch statt, werden die Modifikationen der validierten Transaktion während der zweiten Phase veröffentlicht.

Wie im vorangegangenen Kapitel 2.1.2 erläutert, müssen beim Abbruch einer Transaktion alle bisher von dieser Transaktion durchgeführten Modifikationen verworfen werden. Dazu ist keinerlei Kommunikation mit nebenläufigen Transaktionen erforderlich, da sie aufgrund der Isolationseigenschaft zu diesem Zeitpunkt keinerlei Kenntnis von der Ausführung und den Ergebnissen dieser Transaktion haben. Dieser Abbruch kann nun selbstinitiiert sein, also von der Transaktion selbst explizit erwünscht worden sein, oder durch ein externes Ereignis erzwungen worden sein. Dies ist typischerweise bei der Änderung der Eingangsdaten einer Transaktion der Fall, also beim erfolgreichen Abschluss einer anderen nebenläufigen Transaktion, die schreibend auf die Eingangsdaten der abzubrechenden Transaktion zugegriffen hat. Zur Ermittlung dieser Berechtigung werden die Readsets und die Writesets der Transaktionen in einer Validierungsphase verglichen und die Modifikationen in der Commitphase bei Erfolg der Validierung veröffentlicht. Üblicherweise ist das Writeset einer Transaktion Bestandteil des Readsets dieser Transaktion, so dass dies zur Vereinfachung auch bei den folgenden Erklärungen angenommen wird. Eine detailliertere Beschreibung mit Beleuchtung des theoretischen Hintergrundes findet sich beispielsweise in den Kapiteln 3.4 bis 3.6 von [WendKV03].

Bei der Rückwärtsvalidierung wird während der Validierung einer Transaktion T das Readset von T mit den Writesets aller seit dem Start von T abgeschlossenen Transaktionen verglichen. Tritt ein Konflikt auf (die Schnittmenge mindestens eines Vergleiches ist nicht leer), so muss die Transaktion abgebrochen werden.

Eine Optimierung ist dahingehend möglich, dass die Writesets anderer Transaktionen nicht bis zur Validierungsphase aufbewahrt, sondern bereits bei deren Empfang ausgewertet werden. Treten zu diesem Zeitpunkt bereits Konflikte auf, so werden diese bei der eigenen Validierung zwangsläufig zum Abbruch führen. Treten hingegen zu diesem Zeitpunkt keine Konflikte auf, so kann bei einem Zugriff auf durch andere Transaktionen veränderte Daten bereits konsistent auf diese zugegriffen werden. Bei der Implementierung einer solchen optimierten Variante können

insbesondere die zwei Phasen Validierung und Veröffentlichung zu einer einzigen Phase zusammengeführt werden, da die Validierungsphase für alle beteiligten Transaktionen ein eindeutiges Ergebnis hat.

Im Gegensatz dazu wird bei Vorwärtsvalidierung nicht mit den bereits abgeschlossenen Transaktionen verglichen, sondern mit den noch laufenden, so dass bei einem Konflikt entweder die eigene Transaktion abgebrochen wird oder alle im Konflikt befindlichen anderen Transaktionen abgebrochen werden müssen. Falls immer letzteres Verfahren angewandt wird, handelt es sich effektiv um eine ähnliche Vorgehensweise wie bei der oben beschriebenen Optimierung, bei der nur noch Transaktionen laufen, die in keinem Konflikt zu bereits abgeschlossenen Transaktionen stehen. Im Gegensatz zur Rückwärtsvalidierung ist es bei der Vorwärtsvalidierung jedoch möglich, eine Entscheidung über die abzubrechenden Transaktionen zu fällen. Somit können Prioritäten und Fairness integriert oder die für eine bestimmte Anwendung erforderliche Reihenfolge eingehalten werden. Durch Warten mit Hilfe von Sperren kann versucht werden, einem Konflikt aus dem Weg zu gehen, was jedoch je nach Implementierung der Sperren entweder sehr langes Warten mit sich bringt oder eine Behandlung der eventuell auftretenden Verklemmungen (engl. Deadlocks) erforderlich macht.

Zur Sicherung der Modifikationen einer abgeschlossenen Transaktion ist das Verfahren zur Auswahl einer abschließenden Transaktion irrelevant. Das in dieser Arbeit vorgestellte Verfahren zur Sicherung eines transaktionalen Speichers ist also unabhängig von der Wahl des Verfahrens zur Validierung (siehe auch [RajwSB02], [LeLaEV04] und [WendKV03]), da ausschließlich die Modifikationen bereits erfolgreich abgeschlossener Transaktionen gesichert werden (im Sinne von [DadaSR82] sind die Sicherungen immer logisch konsistent).

2.1.4 Transaktionen in Betriebssystemen

Verschiedentlich wurde erkannt (siehe [StonVM84], [LiskDP88] und [BlacUT90]), dass die Integration von Transaktionen in Betriebssysteme vorteilhaft sein kann. Einerseits für den Programmierer, der bei nebenläufigen Prozessen auf einer Maschine von expliziter Synchronisierung entbunden wird, und andererseits für die Implementierung von Persistenz, da sich durch die Semantik von Transaktionen die gesuchten Konsistenzpunkte (siehe auch [FeShPD99] und [RoHeSP90]) automatisch ergeben.

Eine grundlegende Eigenschaft von Transaktionen ist die Rücksetzbarkeit, die aus Sicht des Betriebssystems für alle von einer Anwendung erreichbaren Ressourcen zur Verfügung gestellt werden muss. Dies betrifft hauptsächlich den Zugriff auf den Hauptspeicher, der mittels

Schattenkopien sehr leicht an die transaktionalen Bedürfnisse angepasst werden kann. Darüber hinaus sind jedoch sämtliche Zugriffe auf die üblicherweise nicht rücksetzbare Hardware in geeigneter Weise zu unterstützen. Zwei kleine Beispiele mögen hierfür genügen:

1. Ein Drucker, der bereits während einer noch nicht bestätigten Druck-Transaktion zu drucken beginnt, kann im Falle eines Abbruchs weder bedruckte Blätter zurückziehen noch die Bedruckung der Blätter rückgängig machen.
2. Der Wert eines analogen Sensors zur Messung einer sich ändernden Größe schwankt im Verlauf einer Transaktion, jedoch muss die mehrfache Abfrage eines Sensors innerhalb einer Transaktion aufgrund der logisch atomaren Ausführung den gleichen Wert ergeben.

Aus (1) wird ersichtlich, dass jegliche Signalisierung oder Veränderung eines nicht-transaktionalen Teils des Gesamtsystems dessen Zustand frühzeitig und unabdingbar ändert. Dies verletzt einerseits die Eigenschaft der Isolation und führt im Falle eines Abbruchs auch zu einem inkonsistenten Gesamtsystem, falls keine Kompensation (Undo-Funktionalität) möglich ist. Die Kommunikation einer Transaktion mit nicht-transaktionalen Komponenten eines Gesamtsystems darf (und muss) also erst im Falle einer erfolgreichen Bestätigung dieser Transaktion durchgeführt werden. Für kritische Operationen ist diese Kommunikation ein nicht unterbrechbarer Bestandteil des Abschlusses einer Transaktion, um die Sicht der Transaktionen auf das Gesamtsystem konsistent zu halten – bei einem Geldautomaten beispielsweise ist die Geldausgabe ein Bestandteil des Transaktionsabschlusses.

Aus (2) lässt sich ablesen, dass die Veränderungen außerhalb des transaktionalen Raums nicht während einer Transaktion sichtbar werden dürfen. Für die vollständige Einhaltung dieser Eigenschaft müsste entweder der Zustand jedes potenziell abgefragten Gerätes zu Beginn jeder Transaktion gesichert oder der aufgrund voriger Anfragen spezifizierte Zustand von relevanten Geräten unter Ausschluss von direktem Zugriff bereitgestellt werden. Dies ist für viele Geräte oder Anwendungen jedoch unkritisch, so dass an dieser Stelle oftmals einerseits der Aufwand für die Pufferung und andererseits die Verzögerung der Eingaben vermieden werden kann: Werden beispielsweise während einer von der Tastatur abhängigen Transaktion zur Steuerung eines Editors weitere Tasten gedrückt, können diese sofort verarbeitet und müssen nicht verzögert werden.

Eine weitere grundlegende Eigenschaft von Transaktionen ist die Dauerhaftigkeit der Daten. Diese ermöglicht im Umfeld von transaktional ausgeführten Programmen, dass Daten und Datenstrukturen zwischen zwei Programmläufen überleben, deren Sichern und Laden also nicht wie in herkömmlichen Systemen von der Anwendung übernommen werden muss und somit ein durchgängiges Typsystem existiert. Bei geeigneten Verfahren mit orthogonaler Persistenz (siehe

auch Kapitel 2.3.1) kann so unter Umständen gänzlich auf eine Schnittstelle zum Dateisystem verzichtet werden, da aus Sicht der Anwendung auf alle benötigten Daten direkt zugegriffen werden kann.

2.1.5 Transaktionale Konsistenz in Plurix

Plurix als verteiltes Betriebssystem (siehe Kapitel 1.3 sowie [SchoPT02] und [FrScPR04]) mit Unterstützung für Transaktionen und einem gemeinsamen verteilten Speicher erschließt die Konzepte von transaktionalen Systemen auch für kooperatives Arbeiten im Cluster. Dabei werden die ACID-Eigenschaften von Transaktionen auch bei der Kommunikation zwischen Rechnern eingehalten, so dass sich aus Sicht des Programmierers der sequenzielle Ablauf von vielen Transaktionen ergibt, die jedoch physisch parallel ausgeführt werden können. Wie in der Einleitung bereits erwähnt, unterstützt Plurix erst mit Hilfe der hier vorgestellten Verfahren auch Dauerhaftigkeit von Transaktionen. Die Verfahren können einerseits in Bezug auf Geschwindigkeit im Cluster optimiert und wie in Plurix implementiert oder andererseits an die striktere Dauerhaftigkeit von Datenbanken angepasst werden (siehe auch Kapitel 2.1.2 und insbesondere das nächste Kapitel 2.1.6).

Plurix verwendet Schattenkopien beim Zugriff auf den Hauptspeicher, einerseits um Änderungen des Hauptspeichers zurücksetzen zu können und andererseits um Seitenanfragen anderer Maschinen mit der unmodifizierten Version zu beantworten. Wie auch bei nicht-verteilten transaktionalen Betriebssystemen sind Zugriffe auf Geräte (siehe voriges Kapitel 2.1.4) stark eingeschränkt, da sie sich außerhalb des transaktionalen Raums befinden. Zur Unterstützung eines Abbruchs ist es in einem transaktionalen Betriebssystem jedoch erforderlich, den transaktional korrekten Zustand von Geräten zu kennen, wobei das Zurücksetzen bei einem Abbruch einer Transaktion für das Gerät identisch zu einem Fehlerfall ist. In Kapitel 4.3 wird darauf gesondert eingegangen und ein Verfahren zur Unterstützung von transaktional korrekter Verwaltung von Gerätezuständen vorgestellt. Auf die jeweiligen Eigenschaften von Plurix wird an geeigneter Stelle hingewiesen, falls diese für das vorgestellte Verfahren relevant sind.

2.1.6 Integration von Dauerhaftigkeit

Bei der Integration von Dauerhaftigkeit in ein bestehendes System sind zum einen die in Kapitel 2.1.2 beschriebenen Anforderungen an die Dauerhaftigkeit, zum anderen aber auch die Schnittstellen zum bestehenden System zu erörtern. Diese Arbeit widmet sich Ersterem in Kapitel

3.1 ausgiebig und stellt hier vorerst die für eine Implementierung von Dauerhaftigkeit in einem transaktionalen verteilten Speicher möglichen Ansatzpunkte vor:

1. Die Speicherverwaltung: Beim Abschluss einer Transaktion sorgt die lokale Speicherverwaltung bei der Übernahme der durch die Transaktion veränderten Werte für eine dauerhafte Sicherung auf dem eigenen Knoten. Die Konsistenzierung des Hauptspeichers wird also mit einer lokalen Sicherung verbunden.
2. Das Protokoll: Beim erfolgreichen Eintritt in die Commitphase werden alle publizierten Modifikationen lokal gesichert oder andere Teilnehmer innerhalb oder außerhalb des Clusters zur Replizierung aufgefordert.
3. Das Netzwerk: Beim oder nach dem Versand von publizierten Modifikationen werden die modifizierten Daten angefordert und außerhalb des transaktionalen Systems gesichert.

Lokale Sicherungen wie bei Möglichkeiten (1) und (2) sind schnell und ohne Belastung des Gesamtsystems realisierbar, bergen jedoch beim Ausfall eines beliebigen Knotens im Cluster die Gefahr des vollständigen Datenverlustes, da nur alle lokalen Sicherungen gemeinsam eine konsistente Version der transaktionalen Daten bilden können. Bei einem auftretenden Fehler müssen die über mehrere Knoten verteilten Zustände zeit- und kommunikationsintensiv analysiert werden, um die relevante Datenbasis bestimmen zu können.

Bei einer Replizierung der lokalen Daten auf n weitere Knoten innerhalb des Clusters unter Zuhilfenahme der mit (2) gegebenen Möglichkeiten kann die Gefahr des vollständigen Datenverlustes bei bis zu insgesamt n Ausfällen abgewandt werden, jedoch werden hiermit alle beteiligten Knoten bei jedem Abschluss einer Transaktion massiv belastet und dadurch die Skalierbarkeit des verteilten Systems eingeschränkt. Des Weiteren muss nach einem Fehler wie bei voriger Variante eine globale Sicht erstellt werden, wobei die zu analysierende Datenmenge enorm vergrößert ist. Eine Replizierung außerhalb des Clusters entspricht Variante (3) mit enger Kopplung zwischen Sicherungsrechnern und Clusterbetrieb.

Unter Verwendung der Möglichkeit (3) ergeben sich mehrere Konsequenzen, die je nach Anwendungsgebiet unterschiedlich bewertet werden müssen und nur im Kontext als Vor- oder Nachteil bezeichnet werden können. Wesentlicher Bestandteil einer Integration von Dauerhaftigkeit auf Basis der Kommunikation über das Netzwerk besteht in der Möglichkeit einer Entkopplung des Clusterbetriebs von dediziert zur Verfügung gestellten Sicherungsrechnern. Zur Vereinfachung wird angenommen, dass es sich um einen einzelnen Sicherungsrechner handelt, wobei die folgende Sammlung an Konsequenzen, auf die in der weiteren Diskussion eingegangen werden soll, davon unberührt bleibt:

- Der für die Sicherung zuständige Rechner ist vom Clusterbetrieb entkoppelt.
    - Das System des Sicherungsrechners kann auf die Bedürfnisse eines solchen Rechners maßgeschneidert und reduziert werden (siehe auch Kapitel 2.3.2).
    - Ein herausgehobener Rechner lässt sich leichter vor Hardwarefehlern und somit fatalen Ausfällen schützen als die vielen und unter Umständen auch räumlich verteilt stehenden Rechner eines Clusters.
    - Der Cluster kann im fehlerfreien Betrieb auch bei Ausfall des Sicherungsrechners uneingeschränkt weiterarbeiten.
    - Der Sicherungsrechner ist von einem Zurücksetzen des Clusters nicht betroffen (siehe insbesondere Kapitel 2.3.3).
    - Fehler im Clusterbetrieb können nicht zu Fehlern im Sicherungsrechner führen.
- Da die Sicherung nicht zentraler Bestandteil beim Abschluss einer Transaktion ist, kann die physikalische Sicherung zeitlich vollständig von der Transaktion entkoppelt sein (siehe dazu auch Kapitel 2.1.2 und 3.1).
- Die Einzelknoten werden bei Rekonstruktion des gültigen Abbildes nicht benötigt.
- Die Schnittstelle zwischen Cluster und Sicherungsrechner ist minimal.
- Für den Sicherungsrechner ist zusätzliche Hardware erforderlich.
- Ein einzelner Sicherungsrechner ist ein „single point of failure“.

Die Entkopplung zwischen Transaktionsabschluss und Sicherstellung der Dauerhaftigkeit ist je nach Anwendung erfreulich, da sich für diesen Fall eine deutlich effizientere Sammlung der Daten realisieren lässt (siehe Kapitel 6.4), oder nachteilig, da im Fehlerfall auch bestätigte Modifikationen von Transaktionen verloren gehen können, was der Semantik von Datenbank-Transaktionen hinsichtlich der Dauerhaftigkeit widerspricht (siehe Kapitel 2.1.2). Prinzipiell lässt sich der Abschluss einer Transaktion auch an die Sicherung im Pageserver koppeln, wodurch die von Datenbanken geforderte Bedingung der Integration erfüllt werden könnte. Jedoch erfordert dies den permanenten Einsatz des Pageservers und eine Verankerung der Kommunikation mit dem Pageserver im Protokoll, zudem wird der Abschluss jeder Transaktion entscheidend verlangsamt.

Die Entkopplung zwischen Pageserver und Cluster hat mehrere Auswirkungen. Zum einen kann das auf dem Pageserver laufende System auf die Bedürfnisse eines Pageservers maßgeschneidert werden, zum Beispiel im Hinblick auf Verwaltung der Halde, Treiberarchitektur oder Benutzerinterface. Des Weiteren kann die Hardware eines einzelnen dedizierten Rechners leichter beziehungsweise kostengünstiger vor Hardwaredefekten geschützt werden, indem zum Beispiel das Netzteil redundant ausgelegt oder mit einer unterbrechungsfreien Stromversorgung

versehen wird, was für alle Rechner eines Clusters zu teuer oder zu aufwendig wäre. Zum Dritten kann ein fehlerfrei laufender Cluster auch bei Fehlern des Pageservers oder der Verbindung zum Pageserver völlig ohne Beeinträchtigung weiterarbeiten. Zuletzt sind Fehler im Cluster für den Zustand des Pageservers unkritisch, da der Pageserver nicht selbst am transaktionalen verteilten Speicher teilnimmt. Würde er direkt am Clusterbetrieb teilnehmen, wäre je nach Fehler (siehe Kapitel 4.1) auch ein Zurücksetzen des Pageservers erforderlich. So jedoch kann der Pageserver in der Zeit, die der Cluster zum Zurücksetzen benötigt, bereits die internen Strukturen aktualisieren und den eigenen Zustand stabilisieren.

Beim Zurücksetzen des Clusters zum Beispiel aufgrund eines schweren Fehlers ist ein außerhalb des Clusters angesiedelter Pageserver imstande, dem neu startenden Cluster ohne dessen Hilfe ein konsistentes Abbild zur Verfügung zu stellen. Wären die Daten eines vollständigen Abbildes auf alle Einzelknoten verteilt, so wäre eine globale Sicht auf die verteilt gesicherten Daten erforderlich, was einen hohen Rechen- sowie Kommunikations- und somit Zeitaufwand beim Sammeln und Auswerten der Daten bedeuten würde. Zudem wäre diese globale Sicht bei einem mittel- oder langfristigen Ausfall eines Knotens (zum Beispiel Defekt an CPU oder Netzwerkverbindung) nur mit Hilfe von zusätzlicher Replikation innerhalb des Clusters möglich. Die dafür benötigte Kommunikation ist mindestens so hoch wie die Kommunikation mit einem Pageserver, belastet jedoch die übrigen Maschinen im Cluster. Das Abbild sollte also außerhalb der abzusichernden Rechner liegen, um bei dauerhaften Hardwarefehlern sofort auf eine gültige Datenbasis zurückgreifen zu können (siehe [LiDuSD97]).

Ein eigenständiger Pageserver kommuniziert mit dem Cluster ausschließlich über das Netzwerk-Protokoll, welches die einzige Schnittstelle zwischen Pageserver und Cluster darstellt. Diese Schnittstelle ist klar definiert und überschaubar und ermöglicht damit eine einfache und effiziente Struktur für den Pageserver (siehe auch [CaFrFG94]). Im Fall von seitenbasierten Systemen wie Plurix wird darüber hinaus die Implementierung mit Hilfe der orthogonalen Persistenz (siehe Kapitel 2.3.1) erleichtert, da in diesem Fall die zu sichernden und zu verwaltenden Daten ausschließlich aus Seiten bestehen und somit die interne Objektstruktur der Daten irrelevant ist.

Wird die Dauerhaftigkeit nicht innerhalb des Clusters realisiert, so ist für den außerhalb des Clusters befindlichen Pageserver zusätzliche Hardware erforderlich, die für Aufgaben innerhalb des Clusters nicht zur Verfügung steht. Dadurch wird jedoch der Cluster entlastet, da die einzelnen Knoten keine Tätigkeiten zur Sicherstellung der Dauerhaftigkeit der Daten wahrnehmen müssen. Die umfassenden Aufgaben des Pageservers in Bezug auf Netzwerkkommunikation und

Festplattenaktivität dürfen diesen dann vollständig auslasten. Nur während einer geringen Auslastung des Pageservers ist die Trennung zwischen Cluster und Pageserver in Bezug auf Rechenleistung nachteilig im Vergleich zu einer Integration der Dauerhaftigkeit in den Cluster, da die noch zur Verfügung stehende Rechenkapazität des Pageservers brachliegt.

Ein einzelner Pageserver ist zur Realisierung von Dauerhaftigkeit nur dann sinnvoll, wenn seine Funktionalität sowie die Erreichbarkeit durch den Cluster sichergestellt werden kann. Da auf dem Pageserver keinerlei Benutzerprogramme laufen und die Funktionalität des Pageservers immer vollständig genutzt wird, ist die Fehlerwahrscheinlichkeit in der Software des Pageservers deutlich niedriger als bei typischer Anwender- und Systemsoftware. Für die Hardware können bei Bedarf zusätzliche Maßnahmen zur Verbesserung der Ausfallsicherheit (siehe oben) getroffen werden. Dennoch empfiehlt sich für die Sicherstellung der Verfügbarkeit eine redundante Auslegung des Pageservers, um einen kritischen Single Point of Failure zu vermeiden. Vorschläge zum Aufbau und Einsatz eines kooperativen oder redundanten Pageserver-Clusters finden sich in Kapitel 6.1. Bei Vernetzung der Knoten mit deutlich schnelleren Komponenten oder einem Cluster aus sehr vielen Knoten ist eine Verteilung im Sinne eines kooperativen Pageservers empfehlenswert, da somit Engpässe bei der Kommunikation oder beim Schreiben auf die Festplatte vermieden werden können.

2.1.7 Zusammenfassung

Es wurde gezeigt, dass der Einsatz von Transaktionen in Betriebssystemen möglich und unter bestimmten Voraussetzungen sinnvoll ist. Die Dauerhaftigkeit der durch Transaktionen erzeugten und veränderten Daten entbindet den Programmierer außerdem von der Implementierung von Dateiformaten und der dafür notwendigen Serialisierung und Deserialisierung. Dies reduziert die Codegröße und somit die möglichen Fehlerquellen in Anwendungsprogrammen und im Betriebssystem, da im Optimalfall gänzlich auf Dateisysteme verzichtet werden kann. Infolge der reduzierten Codegröße wird das Gesamtsystem übersichtlicher und kann leichter gepflegt werden.

In verteilten Betriebssystemen liefert die ACID-Semantik von Transaktionen ein Konzept, das einerseits die Konsistenz des Gesamtsystems sicherstellt und andererseits den Programmierer bei der Entwicklung von verteilten Anwendungen unterstützt. Außerdem wird er von expliziter Synchronisierung befreit, da permanent eine konsistente Sicht auf alle Daten gewährleistet wird und Zugriffskonflikte vom System aufgelöst werden. Die Integration der Dauerhaftigkeit ist im verteilten Betriebssystem mit Transaktionen besonders effektiv und einfach auf Basis der Netzwerkschnittstelle möglich, wobei die Sicherung der Daten je nach Anwendung an den

Transaktionsabschluss gekoppelt oder von diesem entkoppelt werden kann. Der vorgestellte Ansatz eignet sich also sowohl für sicherheitskritische Anwendungen (Sicherung ist an Abschluss gekoppelt) als auch für unkritische Anwendungen, bei denen ein höherer Durchsatz (Sicherung ist vom Abschluss entkoppelt) wünschenswert ist.

2.2 Protokoll-Design

Die bei transaktionaler Konsistenz geforderten Eigenschaften (siehe Kapitel 2.1.2) lassen sich für ein Betriebssystem mit verteiltem gemeinsamem Speicher direkt in Anforderungen an das Kommunikationsprotokoll umsetzen. Hierzu muss das Protokoll hauptsächlich einen konsistenten Datenaustausch zwischen Knoten ermöglichen und die Serialisierbarkeit von Transaktionen gewährleisten (siehe [WendKV03]). In diesem Kapitel werden zuerst minimale Anforderungen erarbeitet und Möglichkeiten zu deren Implementierung aufgezeigt; diese werden anschließend durch ausgefeiltere Ansätze erweitert und optimiert. Tupel-Räume können als Spezialisierung eines virtuellen gemeinsamen Speichers angesehen werden. Diese wurden unter anderem mit Linda (siehe [GeleGC85]) erforscht und finden derzeit bei Sensor-Netzwerken (siehe auch [FoRoMA05] und [HeSiBE01]) neue Anwendung.

2.2.1 Sicherstellung der Synchronität des Clusters

Üblicherweise wird beim Abschluss von Transaktionen mittels Zwei-Phasen-Protokoll (siehe Kapitel 2.1.3) mit Validierung und Veröffentlichung ermittelt, ob die Gesamtheit der Änderungen einer Transaktion konsistent ist und dauerhaft gemacht werden soll (Commit), oder ob sie aufgrund von Konflikten mit anderen Transaktionen wieder rückgängig gemacht werden muss (Abort). Potenzielle Konflikte werden während der Validierungsphase ermittelt und aufgelöst. Dies setzt ein Abstimmungsverfahren voraus, bei dem ermittelt wird, welche Transaktionen abgebrochen und welche Berechnungen folglich verworfen werden müssen, um das Gesamtsystem in einem konsistenten Zustand zu erhalten. Im einfachsten Fall kann man auf die Abstimmung verzichten, indem immer die aktuell zu beendende Transaktion gewinnt und alle im Konflikt befindlichen Transaktionen abgebrochen werden („First Wins“-Strategie). In diesem Fall beschränkt sich die Validierung also auf die Auswahl einer nächsten Transaktion, die gewinnen wird, und den selbstständigen Abbruch aller Transaktionen, die im Konflikt zu dem während der Veröffentlichungsphase verschickten Writeset stehen. Am einfachsten kann eine Auswahl mit Hilfe eines zirkulierenden Tokens zur eindeutigen Ordnung (siehe [ThomMC79]) realisiert werden, das

zum Abschluss einer Transaktion erforderlich ist. Eine Transaktion, die ihre Änderungen publizieren und somit dauerhaft machen möchte, muss also statt der Abstimmung mit anderen Transaktionen nur noch ein Token anfordern und kann, sobald sie dieses erhalten hat, ihre Änderungen publizieren. Diese Vereinfachung der Validierungsphase ist für bestimmte Anwendungen inakzeptabel, da die Erfolgswahrscheinlichkeit sinkt, je länger eine Transaktion läuft und je größer Readset und Writeset werden. Das Verfahren schützt weder vor Aushungerung, noch kann es angemessene Fairness bieten. Dafür kann es jedoch effizient realisiert werden und bietet eine effiziente und generische Lösung zur Validierung von verteilten Transaktionen, die zur Verminderung des Konfliktpotenzials eine kurze Laufzeit aufweisen sollten. Durch die Reduzierung der Kommunikation während der Validierungsphase wird der Cluster insgesamt entlastet und jeder einzelne Transaktionsabschluss beschleunigt.

Damit eine Transaktion einen Konflikt erkennen und somit sich selbst abbrechen kann, muss das Writeset empfangen worden sein, aus dem der Konflikt ermittelt werden kann. Dies erfordert entweder eine zuverlässige Kommunikation zwischen allen Knoten oder, wenn die dafür benötigte Kommunikation vermieden werden soll, zumindest eine Erkennung von Paketverlusten. Für den Clusterbetrieb lässt sich eine Einteilung von Paketen in „unkritisch“ und „kritisch“ vornehmen, deren Einteilung mit den Phasen einer Transaktion zusammenfällt: Während der Ausführung einer Transaktion werden ausschließlich Daten von anderen Stationen angefordert, deren Inhalt zwar aktuell sein muss, deren Ausbleiben aber nicht die Konsistenz des Clusters gefährdet. Im Gegensatz dazu ist ein verschicktes Writeset elementar für die Konsistenz des Clusters, da nur bei sofortiger Überprüfung auf Konflikte die Konsistenz im Cluster gewährleistet werden kann. Die am Cluster beteiligten Rechner können am einfachsten über eine logische Zeit synchronisiert werden, die bei jedem erfolgreichen Abschluss einer Transaktion automatisch inkrementiert und mit jedem Paket verschickt wird. Somit kann jede Maschine beim Empfang überprüfen, ob sie eine mit dem Sender synchrone Zeit hat. Empfängt sie ein Paket mit zu niedriger logischer Zeit, hat die fremde Maschine noch nicht alle Writesets empfangen und benötigt infolgedessen eventuell Hilfe durch erneute Übertragung dieses Writesets. Empfängt eine Maschine ein Paket mit zu hoher logischer Zeit, hat sie selbst ein Writeset verpasst und muss dieses anfordern, um wieder synchron mit dem Cluster zu werden.

Bei Paketverlust ist die erneute Übertragung von Paketen erforderlich. Dazu müssen die Daten, die übertragen wurden, noch vorhanden oder rekonstruierbar sein. Im Fall von unkritischen Paketen während der Ausführungsphase beschränkt sich dies auf die konsistenten Daten der Halde (siehe Kapitel 2.2.3), die aufgrund der Semantik von Transaktionen (siehe Kapitel 2.1.2) ohnehin

dauerhaft sind und somit problemlos erneut übertragen werden können. Im Fall von kritischen Paketen während der Veröffentlichungsphase werden jedoch Informationen über abgeschlossene Transaktionen und über den Status des Clusters (siehe Kapitel 2.2.2) übertragen. Diese Informationen befinden sich außerhalb des transaktionalen Raums und müssen daher gesondert gepuffert werden, um eine erneute Übertragung zu ermöglichen. Auf die jeweiligen Möglichkeiten zur Behandlung von Paketverlusten wird in Kapitel 4.1 genauer eingegangen.

2.2.2 Steuerung des Clusters

Die wichtigsten Aufgaben zur Unterstützung von transaktionaler Konsistenz liegen in der Koordination von Transaktionen. Wie im vorigen Kapitel gezeigt, ist eine einfache Möglichkeit zur Synchronisierung im Cluster auf Basis einer globalen logischen Zeit möglich, die beim Commit inkrementiert wird. Durch die Beschränkungen der Paketgrößen in den meisten Netzen wie auch bei dem von Plurix derzeit genutzten Fast Ethernet kann es vorkommen, dass das Writeset eines Commits nicht vollständig in einem Paket übertragen werden kann. Um die Abarbeitung dennoch zumindest logisch atomar zu ermöglichen, müssen die Pakete eines Writesets entweder bis zum vollständigen Empfang gepuffert werden, oder anstehende sowie neu ankommende Anfragen müssen bis dahin verzögert werden. Dazu muss das Protokoll einen Zustandsübergang anbieten, der die Abarbeitung eines Writesets anzeigt. Dies kann am einfachsten durch eine Integration in die logische Zeit geschehen, die dann auch vor Beginn der Abarbeitung eines Writesets inkrementiert wird. In diesem Fall enthält die logische Zeit alle Informationen, ob die Station synchron zum Cluster ist, und auch, ob sie sich in der Ausführungsphase befindet oder ein Writeset empfängt (siehe Abbildung 2.1, die logische Zeit ist dort als *cnum* bezeichnet).

Die vorgeschlagene Vereinfachung der Validierungsphase im Sinne von „First Wins“ erfordert den sicheren Austausch eines Tokens. Die Absicherung muss für das Token aber im Gegensatz zu den anderen Pakettypen nicht nur gegen Paketverlust erfolgen, sondern auch gegen eine Duplizierung des Tokens. Insbesondere bei möglicherweise identischen Stationsnummern, wie dies beispielsweise bei Plurix während des Beitritts zum Cluster der Fall ist, müssen geeignete Möglichkeiten zur Identifikation des Empfängers vorgesehen werden, um bei einer eventuellen Neuübertragung den richtigen Empfänger garantieren zu können. Geeignete Möglichkeiten für eine solche Identifikation können beispielsweise das Hinzufügen der MAC-Adresse oder eines zufälligen Wertes sein.

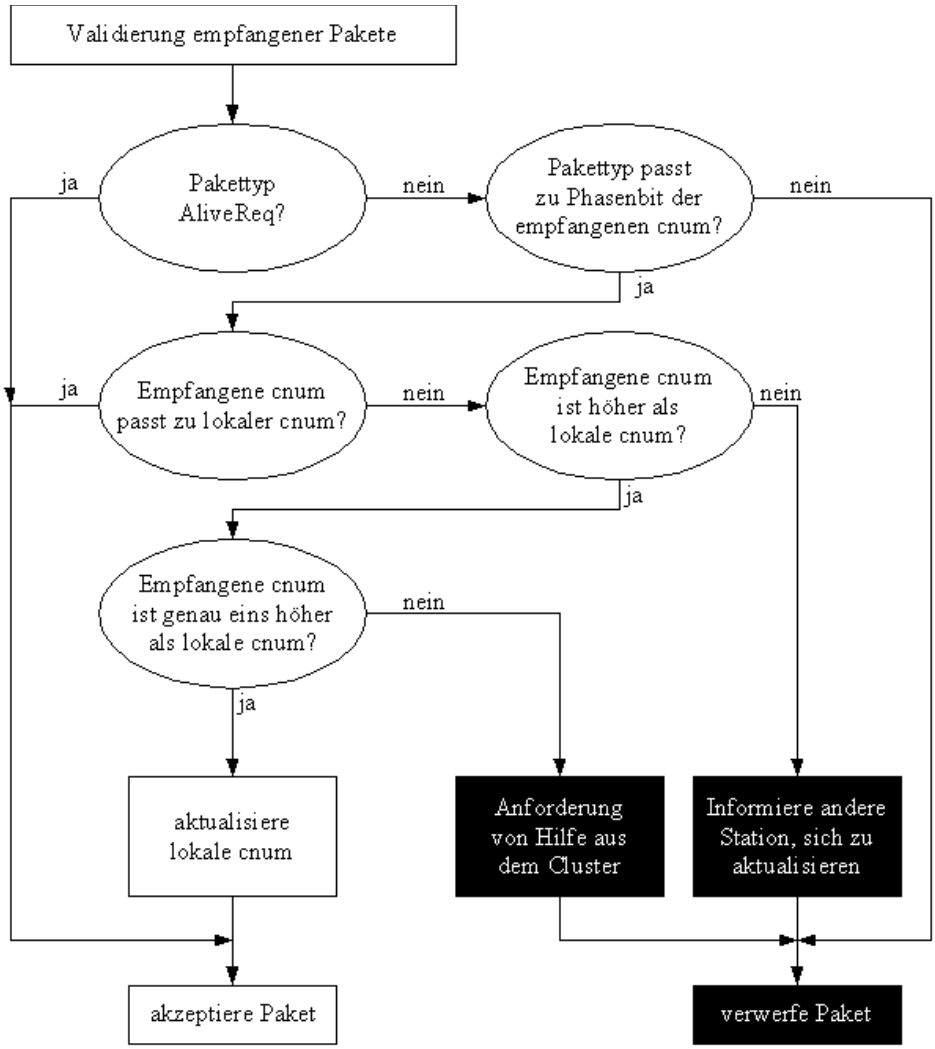

Abbildung 2.1: Validierung empfangener Pakete

Neben Versand und Verwaltung von Commits muss das Protokoll eine Möglichkeit bieten, die Antwortfähigkeit anderer Maschinen zu testen. Dies ist sowohl beim Beitritt zum Cluster erforderlich, um einerseits das Vorhandensein eines Clusters zu erkennen und andererseits die aktuell gültige globale Zeit zu erfahren, als auch im regulären Betrieb, um den Leerlauf des Clusters von einer durch das Netzwerk bedingten Isolation der eigenen Maschine unterscheiden zu können.

2.2.3 Datenaustausch

Für einen Cluster mit gemeinsamem verteiltem Speicher ist der konsistente Datenaustausch in drei Varianten möglich: auf der Basis von Variablen, Objekten oder Seiten. Aufgrund der Lokalität von Zugriffen (siehe [ZaCaHA98], [DWFuST90]) ist eine Granularität von einzelnen Variablen nur in speziellen Ausnahmen sinnvoll. Ein Zugriff auf Objektebene ist zwar einerseits vorteilhaft, da er die Probleme des False-Sharings vermeidet. Andererseits ist er jedoch auch mit zusätzlichem

Aufwand beim Zugriff auf Elemente eines Objekts verbunden, da jeder Zugriff auf das Objekt ausschließlich über spezielle Methoden oder Handles zulässig ist (siehe [LiCaPP99]), damit die Laufzeitumgebung für jeden Zugriff alle erforderlichen Prüfungen vornehmen und eventuell vorhandene Zeiger auf ausgelagerte oder verschobene Objekte anpassen kann. Im Gegensatz dazu ist eine Verteilung auf Basis von Seiten durch die heute zur Verfügung stehende Hardware mit Hilfe der Paging-Funktionalität problemlos möglich (siehe [TrauSK96]). Dabei wird vom Prozessor, genauer der Memory Management Unit (MMU), beim Zugriff auf eine fehlende Seite ein Pagefault ausgelöst. Dieses Verfahren wird in nicht verteilten Systemen zur Aus- und Einlagerung von Seiten auf beziehungsweise von der Festplatte genutzt und kann für Systeme mit gemeinsamem verteiltem Speicher verwendet werden, um mittels Paging über das Netzwerk unnötige Prüfungen zur Laufzeit zu vermeiden, da die Prüfung automatisch von der MMU vorgenommen wird (siehe [KeedMS89] und [RoKeAM92]). Im Fall von Plurix wurde aufgrund der vorteilhaften Bedingungen zur Laufzeit diese Variante gewählt.

Je nach Prozessor kann eine Seite zwischen 4 KB und 4 MB Daten beinhalten. Für die Übertragung auf einem lokalen Netzwerk sind jedoch Paketgrößen bis 1.5 KB üblich, weshalb auch bei feiner Granularität der Seiten oft die Aufteilung einer Seite in mehrere Pakete erforderlich wird. Im Umfeld von Plurix (siehe [GoecSB05]) mit Intel IA32 kompatiblen Rechnern und Fast Ethernet sind drei Pakete zur Übertragung einer Seite erforderlich, sofern keine Kompression (siehe Kapitel 2.2.6) verwendet wird. Auch im Fall eines Abbruchs der laufenden Transaktion oder einer Umordnung der Pakete muss eine korrekte und konsistente Sicht auf die Daten gewährleistet werden können, was durch eine Nummerierung der zu einer Seite gehörenden Pakete und mit der in den letzten beiden Kapiteln vorgestellten logischen Zeit ohne zusätzlichen Aufwand möglich ist.

Paketverlust bei Seitendaten verzögert zwar die anfordernde Transaktion, ist aber für das Gesamtsystem nicht kritisch, da sich die Daten nach wie vor im Cluster befinden. Für eine erneute Anforderung nach Ablauf eines Timers (Timeout) kann entweder das Protokoll selbst oder die Speicherverwaltung sorgen.

2.2.4 Erweiterung für den Betrieb mit Pageserver

Unter Voraussetzung der oben genannten Eigenschaften des Protokolls sind für den Betrieb eines oder mehrerer Pageserver (siehe Kapitel 2.1.6) nur geringfügige Modifikationen im Protokoll erforderlich. Neben den Benachrichtigungen des Pageservers über Änderungen des Speichers muss für die Behandlung eines Fehlerfalls sichergestellt werden, dass der Pageserver das Zurücksetzen der Maschinen veranlassen kann. Das Zurücksetzen des Clusters muss im Gegensatz zur

Übertragung eines Writesets (siehe Kapitel 2.2.2) nicht nur bei vollständig verfügbarem Netzwerk gelingen, sondern es muss garantiert werden, dass fehlerhafte Maschinen vom Cluster ausgeschlossen werden, sofern sie die Aufforderung zur Zurücksetzung nicht erhalten haben oder nicht beachten.

Für den unbeaufsichtigten Dauerbetrieb eines Clusters sind die automatische Erkennung von Fehlerzuständen sowie passende Strategien zur Fehlerbehandlung erforderlich. Die Fehlererkennung kann dabei entweder vom Pageserver zentral übernommen werden, indem dieser periodisch die korrekte Arbeitsweise der Rechner prüft, oder sie kann von den im Cluster befindlichen Rechnern übernommen werden, indem bei nicht lokal behebbaren Fehlern ein Fehler gemeldet wird. In beiden Fällen gibt es generische Strategien, die ohne die Einbeziehung zusätzlicher Daten in der Halde auskommen, und anwendungsspezifische Strategien, die interne Abhängigkeiten der Daten prüfen. In Kapitel 2.2.7 wird auf die Möglichkeiten zur Erkennung eines Fehlers im Cluster gesondert eingegangen.

2.2.5 Überblick über Pakettypen

Zur Umsetzung transaktionaler Konsistenz im Cluster werden für die Kommunikation zwischen teilnehmenden Knoten unterschiedliche Nachrichten benötigt, die auf die Eigenschaften des vorhandenen Netzwerkes maßgeschneidert werden sollten, um höchste Leistungsfähigkeit zu erreichen. Bei der aktuellen Implementierung werden diese Pakete zugunsten der Kompatibilität mit anderen Rechnern im selben Netzwerksegment auf einen ansonsten überflüssigen IP Header aufgesetzt, der die Leistungsfähigkeit in der verwendeten minimalistischen Ausführung jedoch nur unwesentlich beeinträchtigt. Im Folgenden werden die verschiedenen zwingend benötigten Pakettypen unter Berücksichtigung von Fast und Gigabit Ethernet vorgestellt, siehe dazu auch [FrScTC05] und die Spezifikation des Prototypen in Anhang B mit graphischer Darstellung.

Allgemeiner Aufbau eines Pakets

Der Ethernet Header mit 14 Bytes enthält neben den Adressen von Empfänger und Sender noch ein Typfeld, das die darüber liegende IP-Schicht auswählt. Der für IP mindestens erforderliche Header mit 20 Bytes besteht aus den Adressen für Sender und Empfänger, einer Prüfsumme, der Auswahl der darüber liegenden Schicht sowie einigen konstanten Werten, beispielsweise für das Fehlen weiterer Optionen. Für die Validierung eines Pakets ist, wie bereits in Kapitel 2.2.2 diskutiert, die eindeutig identifizierbare logische Zeit des Senders erforderlich, die, an den jeweiligen Verwendungszweck des Clusters angepasst, ausreichend dimensioniert werden muss. In

Anbetracht der maximal möglichen erfolgreichen Transaktionsabschlüsse ist, wie in der aktuellen Implementierung vorgesehen, ein 64 Bit-Wert dafür ausreichend. Bei der parallelen Nutzung mehrerer unabhängiger Cluster bietet sich die Reservierung eines weiteren Feldes an, um die eintreffenden Nachrichten schnell und effizient zuordnen zu können. Zur Spezifikation des übertragenen Pakets enthält der Header ein Byte für den Typ des Pakets sowie ein weiteres Byte für eine eventuell vorhandene genauere Angabe des Typs. Tabelle 2.1 enthält einen Überblick über die nachfolgend diskutierten Pakettypen.

Einsatzgebiet	***Anfrage***	***Antwort***
Lebenszeichen	AliveRequest	AliveAcknowledge
Abschluss-Berechtigung	TokenRequest	TokenGranted
Abschluss-Bestätigung	WritesetRequest	Writeset
Datenaustausch	DataRequest	DataReply
Rücksetzung	RecoverRequest	RecoverOrder

Tabelle 2.1: Pakettypen

Lebenszeichen (Alive)

Während der Bootphase ist einem Knoten weder die Existenz eines Clusters bekannt, noch kennt der bootende Knoten bei Vorhandensein eines Clusters die aktuelle logische Zeit im Cluster. Um die Lauffähigkeit anderer Knoten im Cluster zu ermitteln, muss also eine Anfrage („AliveRequest“) unterstützt werden, die einerseits den Zustand der angefragten Knoten nicht verändert und andererseits auch bei noch nicht initialisierter Senderzeit beantwortet wird („AliveAcknowledge“).

Berechtigung zum Transaktionsabschluss (Token)

Die Reihenfolge der abschließenden Transaktionen muss für alle teilnehmenden Stationen im Cluster identisch sein (Serialisierbarkeit, siehe [RazECO93]). Für die aktuelle Implementierung der First Wins Strategie mit Vorwärtsvalidierung (siehe [WendKV03]) wird am einfachsten ein zirkulierendes Token (siehe auch Kapitel 2.2.1) verwendet, das den Inhaber berechtigt, die Änderungen seiner Transaktionen in den Cluster zu übertragen und somit dauerhaft zu machen. Das Protokoll bietet zur Übertragung des Tokens von einem Knoten auf einen anderen die Pakettypen „TokenRequest“ und „TokenGranted“ an. Sowohl Verlust als auch Duplizierung des Tokens müssen vermieden werden, da einerseits ohne Token keine Transaktion abgeschlossen werden könnte, andererseits könnten bei mehreren Tokens gleichzeitig Änderungen vorgenommen werden, selbst wenn diese in gegenseitigem Konflikt stehen.

Bestätigung eines Transaktionsabschlusses (Writeset)

Der Inhaber des Tokens muss den Transaktionsabschluss im Cluster in Form eines Writesets publizieren, um einerseits veraltete Daten im Cluster verwerfen und andererseits im Konflikt stehende Transaktionen zurücksetzen zu können. Wie bereits in Kapitel 2.2.3 diskutiert, besteht die kleinste Einheit in einem verteilten Speicher aus einer Variablen, aus einem Objekt oder, wie im Fall von Plurix, aus einer Seite. Die Nummern aller veränderten Einheiten müssen im Writeset aufgenommen und atomar abgearbeitet werden, um Inkonsistenz im Cluster zu vermeiden. Wurden innerhalb einer Transaktion mehr Einheiten verändert als in einem Writeset verschickt werden können, müssen mehrere Pakete erzeugt und mit einer Nummerierung versehen werden, die eine atomare und vollständige Abarbeitung ermöglicht (siehe [FrScTC05]). Da die im Writeset enthaltenen Invalidierungsanweisungen äußerst kritisch für die Stabilität des Gesamtsystems sind, empfiehlt sich der Einsatz von Prüfsummen zur Absicherung gegen Übertragungsfehler.

Bei Überlastung oder kurzfristigem Ausfall des Netzwerkes sollten betroffene Knoten die Möglichkeit haben, das ursprüngliche Paket vom Typ „Writeset“ mittels „WritesetRequest“ erneut anfordern zu können. Da diese Daten ansonsten nicht benötigt und daher nach dem Versand sofort verworfen werden, bietet sich dafür ein kleiner, lokaler Ringpuffer (in der aktuellen Implementierung „WritesetBuffer“) an, in dem die zuletzt verschickten Writesets temporär gespeichert werden. Je nach Einsatzgebiet ist auch eine Verriegelung des Puffers oder der Transaktionsabschlüsse denkbar, wenn unter Umständen noch überalterte Maschinen im Cluster vorhanden sind. Wird eine Überalterung aber als selten angenommen, so kann in solchen Fällen auf die generische Rücksetzung (siehe Kapitel 2.3.3 und 4.2) zur Fehlerbehandlung zurückgegriffen und somit auf den für die Verriegelung zusätzlich erforderlichen Aufwand verzichtet werden.

Datenaustausch (Data)

Die Anforderungen zu dem in Kapitel 2.2.3 ausführlich diskutierten Datenaustausch können direkt in die Pakettypen „DataRequest“ und „DataReply“ umgesetzt werden. Im seitenbasierten Fall kann die von der Hardware bei einem Seitenfehler mitgelieferte Seitennummer direkt im Paket vermerkt werden. Der Eigentümer der angefragten Seite antwortet mit bis zu drei (siehe Kapiel 2.2.6) Paketen, aus denen sich der Inhalt dieser Seite rekonstruieren lässt. Da der Inhalt einer Seite besonders kritisch für die Lauffähigkeit und Konsistenz des Clusters ist, empfiehlt sich ebenso wie beim Versand des Writesets die Verwendung einer paketübergreifenden Prüfsumme.

Eine Speicherung der versandten Pakete ist weder erforderlich noch empfehlenswert, da eine Übernahme dieser Funktionalität durch die anfragende Maschine ohne zusätzliche Buchhaltung

implementiert werden kann und die antwortende Maschine die Seitendaten ohnehin im Hauptspeicher hält.

Rücksetzung (Recovery)

Beim Betrieb mit Pageserver benötigt man eine Nachricht, die den Cluster veranlasst, sich auf ein vom Pageserver angebotenes konsistentes Abbild zurückzusetzen. Dazu wird vom Pageserver eine Nachricht „RecoverOrder“ als Reaktion auf eine Fehlererkennung verschickt. Der Pageserver kann einen zur Rücksetzung führenden Fehler entweder selbst erkennen oder diesen von einem Rechner im Cluster mit Hilfe der Nachricht „RecoverRequest“ mitgeteilt bekommen (siehe Kapitel 2.2.7).

2.2.6 Möglichkeiten zur Optimierung

Die im vorigen Abschnitt vorgestellten Pakettypen genügen zur Implementierung von transaktionaler Konsistenz im Cluster. Je nach Anwendung bieten sich zur Leistungssteigerung zusätzlich die in den folgenden Abschnitten diskutierten Optimierungen an.

Kompression der Seitendaten

Empirische Studien im laufenden Betrieb haben gezeigt, dass viele Seiten ausschließlich Nullbytes enthalten und manche Seiten sehr gut über einen LZW-Algorithmus (siehe [ZiLeUA77], [WelcTH84] und [NelsLD89]) komprimiert werden können. Bereits für die Berechnung der Prüfsumme einer zu übertragenden Seite ist die Betrachtung jedes Wertes dieser Seite erforderlich. Dabei kann ohne große zusätzliche Kosten geprüft werden, ob die Seite leer ist. Beim Start des Systems sind alle Seiten leer, und im laufenden Betrieb werden bereits beschriebene Seiten durch die Garbage Collection wieder geleert. Da infolgedessen sehr häufig leere Seiten angefordert werden, wird neben „DataReply“ der neue Typ „DataEmpty“ empfohlen, dessen Pakete keine weiteren Daten enthalten. Eine leere Seite kann dadurch mittels eines einzigen Pakets mit etwa 60 Bytes statt mittels drei Paketen mit insgesamt etwa 4300 Bytes übertragen werden.

Bei der Übertragung von LZW-komprimierten Seiten entstehen sowohl für den Sender als auch den Empfänger erhebliche Kosten, die unter Umständen schwerer wiegen als die Ersparnis bei der Übertragung. Kann durch die Kompression nicht wenigstens eine Paketübertragung eingespart werden, ist auch nach erfolgter Kompression ein unkomprimierter Versand der Seite sinnvoll, um dem Empfänger die Kosten der Dekompression zu ersparen. In Kapitel 5.3 befinden sich umfassende Messungen zu einem im Rahmen dieser Arbeit implementierten Algorithmus zur Kompression mit einer für kleine Datenmengen optimierten LZW-Variante.

Fairness während der Validierung

Die implementierte First Wins Strategie (siehe auch [ScFrCR04]) ist zwar sehr effizient implementierbar und garantiert den Fortschritt im Cluster, kann jedoch das Aushungern einzelner Transaktionen nicht verhindern. Um den Fortschritt jeder Transaktion zu gewährleisten, kann bei mehrmaligem Abbruch einer Transaktion entweder das zum Abschluss berechtigende Token bereits im Vorfeld angefordert oder alternativ die Validierungsphase dahingehend erweitert werden, dass Transaktionen Einspruch gegen den eigenen Abbruch einlegen können. Ersteres würde auch den erfolgreichen Abschluss unbeteiligter Transaktionen verzögern, so dass diese sehr einfach implementierbare Variante nicht empfehlenswert ist. Letzteres erfordert einen Mechanismus, bei dem vor dem Abschluss einer Transaktion potenziell abzubrechende Transaktionen kontaktiert werden. Bei deren Zustimmung kann die anfragende Transaktion ihre Änderungen wie bei der implementierten First Wins Strategie veröffentlichen. Bei einer oder mehreren Ablehnungen muss je nach Dringlichkeit der eigenen Transaktion diese abgebrochen oder gegen den Willen der anderen Transaktionen abgeschlossen werden. Dies kann zum Beispiel durch mehrphasige Commit-Protokolle unterstützt werden, wie sie unter anderem in [MoLiEC83], [GrReTP93] oder [LaLoNP93] diskutiert werden.

Einigungsverfahren mit Mehrheitsprinzip (siehe [ReitSG96], [LampPM01] und insbesondere auch [GrLaCT04]) wie der Paxos-Algorithmus stellen zwar unter bestimmten Bedingungen auch den Fortschritt der beteiligten Maschinen sicher, können das Aushungern einzelner Transaktionen aber nicht verhindern, da bei ausreichend vielen positiven Stimmen negative Stimmen ignoriert werden.

Lokale Transaktionen ohne Veröffentlichungsphase

Für bestimmte Transaktionen, deren Eigenschaften im Folgenden erläutert werden, kann vollständig auf die Veröffentlichung und die dafür notwendige Anforderung des Tokens verzichtet werden. Nach der Veröffentlichung eines Writesets sind die enthaltenen Seiten exklusiv auf der veröffentlichenden Station vorhanden. Diesen Status verlieren sie, sobald sie von anderen Stationen angefordert werden. Soll eine Transaktion ohne Token abschließen, so darf sie zwar lesend auf beliebige Seiten im DSM zugreifen, jedoch nur auf exklusiv vorhandene Seiten schreiben. Wird keine der von der Transaktion modifizierten Seiten von anderen Maschinen angefordert oder verändert, so kann nun auf die Veröffentlichung des Writesets verzichtet werden, da die geänderten Seiten auf keiner anderen Maschine vorhanden sind und somit dort zu keinem Konflikt führen können. Solche Transaktionen entsprechen lokal ausführbaren Transaktionen, wie sie unter anderem

auch in [SunEJB02] und [OraDAG92] verwendet werden. Wird grundsätzlich für alle Seiten vermerkt, ob sie exklusiv sind, kann die Eigenschaft der Lokalität einer Transaktion unter Plurix sehr einfach an ihrem Ende festgestellt werden, indem geprüft wird, ob alle in der Transaktion geänderten Seiten exklusiv auf dem abschließenden Knoten vorhanden sind. Grundsätzlich existieren solche Transaktionen unter Plurix ausschließlich bei entkoppelter Datensicherung (siehe dazu Kapitel 2.1.2, 2.1.6 und 3.1), da andernfalls keine Seiten exklusiv sind.

Die Verwendung eines Pageservers erfordert für diese Optimierung keine spezielle Anpassung, insbesondere nicht am Protokoll oder beim Transaktionsmanagement, und ist insofern unkritisch. Je nach Netzlast wird der Pageserver jedoch mit hoher Wahrscheinlichkeit bestätigte Änderungen anfordern, so dass bereits während der Laufzeit einer potenziell lokalen Transaktion die veränderten Seiten nicht mehr exklusiv sind. Dadurch kann diese Transaktion nicht mehr lokal abgeschlossen werden. Je nach Anwendung sind in diesem Fall Heuristiken zur Verzögerung der Anforderungen durch den Pageserver oder die Vorhaltung älterer Versionen auf den angefragten Maschinen sinnvoll.

2.2.7 Schnittstellen zur Fehlererkennung

Die Funktionalität zur Fehlererkennung kann sowohl im Pageserver als auch in jedem einzelnen Knoten untergebracht werden, wobei die Schnittstellen zur Fehlererkennung jeweils unterschiedlich sind. Die Fehlererkennung kann sich in beiden Fällen auf die Antwortfähigkeit der Rechner oder auf die interne Konsistenz der Anwendungsdaten beziehen. Im Folgenden werden die vier daraus resultierenden Möglichkeiten diskutiert. Bei Auftreten eines Fehlers (eine Klassifizierung von Fehlern findet sich in Kapitel 4.1) wird der Cluster in einen konsistenten Zustand zurückgesetzt und die Arbeit von dort ausgehend neu gestartet.

Knoten meldet anwendungsunabhängiges Fehlverhalten

Bei der im Protokoll eingebauten Plausibilitätsprüfung (siehe Kapitel 2.2.1 und 2.2.2) sowie bei der Überwachung von internen Zuständen durch den Kern können Fehlersituationen erkannt werden, die ein Zurücksetzen des Clusters mit Hilfe des Pageservers erforderlich machen können. Diese Funktionalität ist unabhängig von aktuell laufenden Transaktionen oder von verteilten Daten und ist somit jederzeit verfügbar. Beim Auftreten eines solchen Fehlers kann der betroffene Knoten durch eine Nachricht vom Typ „RecoverRequest“ (siehe Kapitel 2.2.5) ein Zurücksetzen des Clusters durch den Pageserver verlangen.

Die Möglichkeit zur internen Prüfung systemkritischer Zustände sowie zur Erwirkung eines Zurücksetzens des Clusters stellt eine grundlegende Funktionalität zur Fehlererkennung dar und sollte daher in jedem Fall durch das Protokoll und den Kern des Clustersystems wahrgenommen werden.

Transaktion prüft anwendungsspezifische Daten

Transaktionen mit ihrem Zugriff auf den gemeinsamen Speicher können die abgelegten Daten einer Kontrolle unterziehen, die weit über die Semantik der garantierten transaktionalen Konsistenz (siehe Kapitel 2.1.2) hinausgehen. Somit können auch Programmierfehler in der Anwendung erkannt werden. Diese stellen aus Sicht des Pageservers und der Konsistenz des Clusters keinen Fehler dar und können durch erneutes Starten der Anwendung in einem früheren Zustand nicht behoben werden. Wenn Programmierfehler in der Anwendung jedoch ausgeschlossen werden können, lassen sich auf diesem Weg unter Umständen sporadisch oder dauerhaft auftretende Fehler in der Hardware erkennen, wobei erstere durch ein erneutes Starten der Anwendung in einem früheren Zustand ausgeglichen werden können. Dazu wird ebenfalls die in Kapitel 2.2.5 beschriebene Nachricht vom Typ „RecoverRequest“ verwendet. Bei Verwendung der in Kapitel 4.3.2 vorgeschlagenen Kernarchitektur mit einer Benachrichtigung der Anwendungen nach dem Zurücksetzen des Clusters ist eine Unterscheidung zwischen sporadisch und dauerhaft auftretenden Fehlern möglich, so dass die Wiederholung bei mehrfach auftretenden Fehlern erkannt und geeignet behandelt werden kann.

Unabhängig von der verwendeten Programmiersprache und der Systemumgebung ist die Prüfung von Fehlerzuständen innerhalb einer Transaktion bei strukturierter und gewissenhafter Implementierung unabdingbar. Beim Auftreten von nicht durch das Programm erzeugbaren Fehlern bietet die hier vorgestellte Variante eine Möglichkeit, Hardwarefehler zu erkennen und geeignet zu reagieren. Die Implementierung der jeweiligen Tests obliegt hierbei jedoch ausschließlich dem Anwendungsprogrammierer, der bei entsprechender Sachlage und Berechtigung die notwendigen Funktionen des Kerns zum Zurücksetzen des Clusters aufrufen kann.

Pageserver prüft die Antwortfähigkeit der Rechner

Eine einfache Möglichkeit, die Reaktionsfähigkeit sowie die Aktualität einzelner Rechner zu prüfen, stellt der Versand einer Alive-Nachricht (siehe Kapitel 2.2.5) durch den Pageserver dar. Sollte ein zuvor im Cluster angemeldeter Rechner nicht innerhalb einer einstellbaren, auf die gegebenen Verhältnisse des Netzwerkes angepassten Zeitspanne reagieren, kann er als defekt angenommen werden. Ein Rechner, der nur einige Pakete verloren haben sollte, wird auf die Alive-

Anfrage antworten und aufgrund der vom Pageserver verschickten aktuellen logischen Zeit (siehe Kapitel 2.2.2) alle noch ausstehenden Informationen anfordern und sich somit selbst auf den aktuellen Stand bringen.

Für die Feststellung eines Fehlers sind keinerlei Informationen über den Aufbau oder gar den Inhalt des verteilten Speichers erforderlich. Der Ausfall eines Rechners oder einer Verbindung zu einem Rechner kann effizient und mit geringer Belastung der einzelnen Knoten und des Netzwerkes festgestellt werden, so dass sich diese Variante der Fehlererkennung durch den Pageserver grundsätzlich anbietet.

Pageserver prüft anwendungsspezifische Daten

Wie auch schon bei der oben beschriebenen Prüfung von anwendungsspezifischen Daten durch Transaktionen ist für diese Art der Fehlererkennung eine vom Anwendungsprogrammierer erstellte Prüfroutine erforderlich, die in diesem Fall jedoch auf dem Pageserver arbeitet. Dies erfordert eine Installation dieser Prüfroutinen auf dem Pageserver durch den Cluster und deren explizite Anpassung bei Änderungen der Laufzeitstrukturen, da der Pageserver unabhängig vom Aufbau der verteilten Daten implementiert werden sollte (orthogonale Persistenz, siehe Kapitel 2.3.1). Der Pageserver verhält sich während der Prüfung wie einer der Clusterrechner und löst im Fehlerfall ein Zurücksetzen des Clusters aus.

Die erforderlichen Schnittstellen sowie der Aufwand während der Laufzeit zur Unterstützung dieser Fehlererkennungsmöglichkeit sind erheblich und verletzen das vorteilhafte Konzept der orthogonalen Persistenz. Des Weiteren verhält sich der Pageserver während der Ausführung der installierten Prüfroutinen wie ein Clusterrechner nach dem oben beschriebenen Konzept der Prüfung durch eine Transaktion, so dass eine Integration dieser Routinen in einen Clusterrecher naheliegender und deutlich weniger aufwendig ist. Die technisch zwar mögliche Prüfung anwendungsspezifischer Daten durch den Pageserver ist also unnötig umständlich, weshalb von einer Umsetzung dieser Variante abgeraten wird.

2.2.8 Zusammenfassung

Die aus der transaktionalen Konsistenz resultierenden Anforderungen an ein Protokoll zur Kommunikation zwischen mehreren Rechnern lassen sich direkt in eine schlanke Implementierung für verteilt laufende Transaktionen umsetzen. Die wichtige Aufgabe der Steuerung des Clusters sowie der Sicherstellung der Synchronität aller beteiligten Maschinen kann, wie durch den Prototyp belegt wurde, mit geringem Aufwand effizient und mit einer überschaubaren Anzahl an Typen von

Nachrichten gelöst werden. Darüber hinaus wurden wirksame Optimierungen für verschiedene Anwendungsbereiche entwickelt, umgesetzt und getestet. Eine Validierung der theoretischen Grundlagen findet sich in [WendKV03].

Für den Betrieb mit Pageserver sind nur geringfügige Erweiterungen des Protokolls und des Kerns erforderlich, hauptsächlich bestehend aus zwei neuen Nachrichten und einer Funktion zum Zurücksetzen einer Maschine. Obwohl bereits vorhandene Module weiterverwendet werden können, kann die Funktionalität des Pageservers sehr stark vom zugrunde liegenden System abgegrenzt werden. Die für die zuverlässige Fehlererkennung möglichen Strategien wurden ausführlich diskutiert und daraus Nachrichten für das Protokoll abgeleitet, die auf natürliche Weise im System integriert werden können.

2.3 Pageserver-Integration

Die Verbindung zwischen Pageserver und Cluster sowie die Aufgaben und Dienste des Pageservers werden im Folgenden unter unterschiedlichen Gesichtspunkten diskutiert. In Kapitel 2.3.1 wird auf die Rolle des Pageservers aus Sicht der Organisation von Daten eingegangen, Kapitel 2.3.2 widmet sich dem internen Aufbau der Implementierung. Aus Sicht des Clusters dient der Pageserver zur Bereitstellung eines konsistenten Abbildes im Fehlerfall, worauf in 2.3.3 eingegangen wird.

2.3.1 Orthogonale Persistenz

Der Begriff der orthogonalen Persistenz bezeichnet die Möglichkeit, Daten unabhängig von ihrem Format und ihrem Aufbau zu speichern (siehe beispielsweise [AbraHM81], [AtBaAP83], [CockOP83], [ClamDP91], [AtJoDI96] und [HoChOP99], in Bezug auf Betriebssysteme auch [LiBoMU94]). Dadurch kann das sichernde System in Bezug auf die Organisation von Daten vollständig vom zu sichernden System entkoppelt werden. Beispiele für solche Systeme finden sich unter anderem in [AtBuTP87], [DeCoND90], [DeBoGO94] und [AtDaOP96]. Nach der Definition von Atkinson und Morrison (siehe auch [AtMoOP95]) sind für orthogonale Persistenz die folgenden drei Merkmale ausschlaggebend, die auch vom implementierten Prototyp erfüllt werden:

1. Unabhängigkeit des Zugriffs: Der Zugriff auf persistente Objekte ist für den Programmierer transparent, der Code für den Zugriff auf volatile und persistente Objekte ist identisch. Für den Programmierer und den Compiler ist es also irrelevant, ob sich ein referenziertes Objekt im volatilen Speicher befindet oder ob es persistent ist.

2. Unabhängigkeit des Datentyps: Die Persistenz ist unabhängig vom Datentyp, jedes Objekt muss persistent gemacht werden können. In einem objektorientierten System bedeutet dies hauptsächlich, dass es keinen ausgezeichneten Elterntyp für persistente Objekte gibt und somit auch das absolute Wurzelelement persistent gemacht werden kann.
3. Gleichheit der Bezeichner: Die Identifizierung von persistenten Objekten ist unabhängig von allen anderen Elementen der Sprache. Im Gegensatz zu Ansätzen, bei denen Persistenz über die Programmiersprache erreicht wird, müssen alle Sprachelemente für alle Objekte unabhängig von ihrer Persistenz gültig sein. Insbesondere folgt daraus, dass weder die Methoden noch die Art der Allozierung noch das Typsystem ausschlaggebend für die Persistenz sind.

Der virtuelle gemeinsame Speicher auf Seitenbasis bietet zur Implementierung von orthogonaler Persistenz eine gute Grundlage (siehe [LiedPS93]), da aus Sicht des Persistenzmoduls alle zu sichernden Einheiten gleich groß und durch ihre Adresse dauerhaft eindeutig identifizierbar sind. Des Weiteren wird dadurch eine völlige Unabhängigkeit von den im Cluster eingesetzten Programmiersprachen sowie vom Aufbau der Objekte und der Halde erreicht (siehe [RoDeOS97]). Das in Kapitel 2.2.5 diskutierte Protokoll bietet bereits alle erforderlichen Schnittstellen und Funktionen, so dass hierfür keine Erweiterungen erforderlich sind.

2.3.2 Pageserver-Architektur

Mit der in Kapitel 2.1.6 diskutierten Wahl des Netzwerks als Schnittstelle zwischen Cluster und Pageserver und dem Einsatz von orthogonaler Persistenz (siehe das vorige Kapitel 2.3.1) ist der Pageserver in der Implementierung völlig unabhängig von dem im Cluster laufenden System. Dennoch können einige Bestandteile des Gesamtsystems übernommen und somit die Aufwendungen für den Pageserver deutlich reduziert werden. Die Verwendung der identischen Implementierung des Protokolls bietet sich dabei grundsätzlich an, um bei neuen Versionen leicht migrieren zu können und Mehrfachentwicklungen und somit Fehlerquellen zu vermeiden.

Integration in das Gesamtsystem

Die in sich abgeschlossene Funktionalität des Pageservers lässt sich, wie der im Rahmen dieser Arbeit implementierte Prototyp belegt, sehr gut in Form eines Moduls in das bestehende System integrieren. Der Prototyp zeigt auch, dass für die Integration eines Pageservers nur sehr wenige Schnittstellen zu dem als Basis dienenden System erforderlich und dadurch Modifikationen der beiden Teile weitgehend unabhängig voneinander möglich sind. Die folgende Tabelle 2.2 bietet eine Übersicht über die Methoden zur Kommunikation des Systems mit dem Pageserver:

Aufrufer	*gerufene Methode*	*Zweck*
Kern	checkstart	Prüfung auf Einsatzmöglichkeit des Pageservers
Kern	boot	Start des Pageservers
Protokoll	isAlive	Meldung einer antwortenden Maschine
Protokoll	loadSendPage	vom Pageserver zu beantwortende Anfrage
Protokoll	pageReceived	Empfang einer Seite ist vollständig
Protokoll	receivedInvalidate	Invalidierung einer Seite durch ein Writeset
Protokoll	localOwner	lokale Eigentümerschaft einer Seite

Tabelle 2.2: Schnittstelle des Pageservers für das System

Der Pageserver benötigt vom zugrunde liegenden System eine rudimentäre Speicherverwaltung sowie Zugriff auf das zum Cluster kompatible Protokoll; beides ist in einem für Cluster tauglichen Betriebssystem vorhanden (siehe [TrauSK96], [ScBiDO02] und [GoScKR03]). Neben diesen erforderlichen Methoden werden in der prototypischen Implementierung des Pageservers noch Bildschirm und Tastatur angesprochen, um die Interaktion mit einem Administrator zu ermöglichen.

Interne Organisation des Pageservers

Die interne Organisation ist aufgrund der weitgehenden Autonomie des Pageservers sehr flexibel, jedoch ist eine Aufteilung der verschiedenen Funktionen in Module sinnvoll. Durch die schlanke Schnittstelle zum System können die Funktionen so mehrfach unterschiedlich implementiert und bei Bedarf ausgetauscht werden. Für die generell benötigten Module wurde im Rahmen dieser Arbeit eine Struktur erarbeitet, die auch durch den Prototyp wahrgenommen wird:

Name des Moduls	*Sichtbarkeit*	*Zweck*
Pageserver	public	Schnittstelle für Aufrufe aus Kern und Protokoll
Loop	protected	Aktivierung von anstehenden Aufgaben
DiskAccess	protected	logische Verwaltung des Speichermediums
IDE / SCSI	protected	Gerätetreiber zum Zugriff auf das Speichermedium
ServModul	protected	Verwaltung und Beantwortung von Anfragen
UserInterface	protected	Interaktion mit dem Administrator

Tabelle 2.3: Empfohlene Struktur eines Pageservers

Eine weitergehende Aufteilung ist stark abhängig von den gewünschten Funktionen und muss infolgedessen anhand der tatsächlichen Implementierung beurteilt werden. Dabei wird dringend empfohlen, die Übersichtlichkeit der öffentlichen Schnittstelle beizubehalten, um die Vorteile der oben diskutierten Unabhängigkeit nutzen zu können.

2.3.3 Rücksetzung des Clusters

Beim Auftreten eines Fehlers muss der Pageserver in der Lage sein, den Cluster auf ein älteres Abbild zurückzusetzen. Dafür ist im Protokoll (siehe Kapitel 2.2.7) eine Nachricht vorgesehen, die ausreichende Teile aller am Cluster beteiligten Rechner zurücksetzt (für eine genaue Erläuterung dieser Teile siehe Kapitel 4.2). Der Pageserver muss seinerseits alle ungültig gewordenen Daten verwerfen, um eine Inkonsistenz zwischen Cluster und Pageserver zu vermeiden. Die bereits mehrfach angesprochene Trennung zwischen Cluster und Pageserver (siehe Kapitel 2.1.6 und 2.3.2) ist auch in dieser Hinsicht relevant, da das zugrunde liegende System eines am Cluster teilnehmenden Pageservers ebenfalls von der Rücksetzung betroffen wäre. Der Pageserver selbst würde dann beim Zurücksetzen der im Cluster laufenden Maschinen ebenfalls vollständig zurückgesetzt und verlöre infolgedessen alle vorhandenen Informationen. Anfragen durch andere Maschinen könnten so erst nach erfolgter Initialisierung des zugrunde liegenden Systems sowie des Pageservers und nach Prüfung der Datenbasis beantwortet werden. Bei einer vollständigen Trennung von Pageserver und Cluster jedoch muss die Software des Pageservers nicht zurückgesetzt werden, da sie unabhängig von den zurückgesetzten Speicherbereichen ist. Somit kann der Pageserver ohne Wartezeit für das System alle noch gültigen Daten ohne Verzug weiterverwenden. Des Weiteren ist die von einem persistenten Medium einzulesende Datenmenge drastisch reduziert, was die benötigte Zeit bis zur Antwortfähigkeit des Pageservers weiter verringert.

Das korrekte Anlaufen der zurückgesetzten Maschinen wird durch die für Transaktionen bereits vorhandenen Funktionen vollständig unterstützt und geschieht somit für die Anwendung transparent. Für spezielle Tätigkeiten wie zum Beispiel die Initialisierung verwendeter Geräte ist eine Schnittstelle auf den im Cluster laufenden Maschinen vorgesehen, die in Kapitel 4.3 diskutiert wird. Sie ermöglicht gezielte Reaktionen auf die Rücksetzung des Systems, falls diese erforderlich sein sollten.

2.3.4 Zusammenfassung

Das bereits vielfach untersuchte Konzept der orthogonalen Persistenz lässt sich sehr gut mit transaktionaler Konsistenz verbinden. Auf der Basis des gemeinsamen verteilten Speichers kann eine effiziente und von der aktuellen Implementierung der Halde im Cluster unabhängige Sicherung von Seiten erfolgen, wobei die durch das Protokoll gegebenen Möglichkeiten bereits vollständig ausreichen. Die Unabhängigkeit von der Speicherorganisation und der Programmiersprache im Cluster ermöglicht den vielseitigen und unveränderten Einsatz in unterschiedlichen

Anwendungsgebieten, wobei die Verwendung des vorhandenen Protokolls eine schlanke und überschaubare Implementierung des Pageservers ermöglicht.

Für die Integration des Pageservers in ein bestehendes Gesamtsystem wurde gezeigt, dass weite Teile des Systems wiederverwendet werden können und die Schnittstelle zu diesem System äußerst schlank sein kann. Dies erlaubt eine Modularisierung und somit eine effiziente Code-Pflege der für einen Pageserver erforderlichen Funktionalität. Die bestehende Implementierung des Protokolls kann wiederverwendet und somit auf eine Neuentwicklung verzichtet werden, wodurch bei Änderungen die erforderliche Migration erleichtert wird und Fehlerquellen vermieden werden können.

In einem System mit transaktionaler Konsistenz sind bereits die Funktionen zur Rücksetzung von Transaktionen vorhanden, so dass nur noch die Funktionalität zum Zurücksetzen des Systems hinzugefügt werden muss. Bei einem nicht selbst am Cluster teilnehmenden Pageserver kann auf eine Rücksetzung des Pageservers vollständig verzichtet und somit das Verwerfen von noch gültigen Daten vermieden werden, wodurch gleichzeitig Effizienz und Überschaubarkeit der Implementierung gesteigert werden können.

2.4 Synergien zwischen transaktionaler Konsistenz und Checkpointing

Wie in den vorangegangenen Kapiteln gezeigt wurde, lassen sich die Anforderungen für Schnappschüsse (Sicherungspunkte, Checkpoints) leicht und effizient mit Hilfe von transaktionaler Konsistenz umsetzen. Dieses Kapitel beleuchtet mehrere besonders auffällige Aspekte der transaktionalen Konsistenz, auch im Vergleich zu anderen Systemen:

1. Die Änderungsfrequenz der nach außen sichtbaren Daten ist niedrig.
2. Alle nach außen sichtbaren Daten sind Bestandteil des letzten konsistenten Zustands.
3. Trotz äußerst geringem Overhead im Betrieb zur Übertragung der für eine Sicherung erforderlichen Seiten ist eine sehr schnelle Rücksetzung möglich.
4. Die Funktionen zur Rücksetzung von Daten sind Bestandteil des zugrunde liegenden Systems.

In herkömmlichen Systemen wie Ivy, Treadmarks oder Kerrighed (siehe unter anderem auch [LiIVYS88], [KeCoTD94], [MoLoKS03] und [FrLoPC05]) werden schreibende Zugriffe auf Seiten ermöglicht, indem der schreibenden Maschine für eine kurze Zeit exklusiver Zugriff auf diese Seite gestattet wird und die ansonsten im Cluster vorhandenen Daten sofort (strikte Konsistenz), mit kurzer Verzögerung (sequenzielle Konsistenz) oder explizit zu einem späteren Zeitpunkt (abgeschwächte Konsistenz) invalidiert werden. Solange die Zeitspanne für den schreibenden und

somit exklusiven Zugriff nicht abgelaufen ist, können die Anfragen auf diese Seite nicht beantwortet werden. Danach ist für einen schreibenden Zugriff eine erneute Absicherung der Exklusivität erforderlich. Dagegen werden bei transaktionaler Konsistenz Anfragen auf Seiten immer mit der zuletzt veröffentlichten Version beantwortet und beliebig viele Schreibzugriffe gebündelt am Ende einer Transaktion publiziert. Somit ergibt sich die unter (1) aufgeführte niedrige Änderungsfrequenz der nach außen sichtbaren Daten. Dies hat insbesondere Auswirkungen auf Seiten, die von einer Station geschrieben und von vielen Stationen gelesen werden, da diese über einen vergleichsweise langen Zeitraum auf die zuletzt publizierte, also aktuelle Version einer Seite zugreifen können und diese erst nach Invalidierung durch eine abschließende Transaktion erneut anfordern müssen. Dieses Verhalten, das unter (2) in obiger Auflistung genannt ist, hat insbesondere auch Auswirkungen auf das Einsammeln von Daten durch den Pageserver. Denn während der Ausführung von Transaktionen im Cluster sind alle bisher publizierten Daten vom Pageserver aus erreichbar. Somit kann dieser die aktuell gültige Version des Gesamtsystems nebenläufig zu den Änderungen vieler Transaktionen ermitteln und sichern, da erst bei deren Abschluss neue Versionen aus den Ergebnissen der Transaktionen gebildet werden. Im Gegensatz dazu wird in anderen Systemen (siehe auch Kapitel 1.4) häufig versucht, durch bekannte Verfahren zur Koordinierung (siehe [ElAlSR96] und [ChJoDO97]) wie beispielsweise der Lamport-Zeit (siehe [LampTC78]) oder dem Chandy-Lamport-Algorithmus (siehe [ChLaDS85]) einen punktuell gültigen Schnappschuss zu erstellen, wobei außer dem Systemzustand auch die noch auf dem Kommunikationsmedium befindlichen Nachrichten berücksichtigt werden müssen (siehe [TaStDS02]). Bei unabhängigen Schnappschüssen kann es vorkommen, dass mit einer Kombination der Schnappschüsse kein global konsistentes Abbild gefunden werden kann, so dass bei einem Fehler auf den initialen Zustand zurückgesetzt werden muss (Domino-Effekt, siehe [RandSS75]). Dies ist bei transaktionaler Konsistenz vermeidbar, da hier auf einfache Weise ein vollständig konsistentes Abbild erstellt werden kann. Die zwei wesentlichen Unterschiede bestehen also zum Ersten in der Lebensdauer der nach außen sichtbaren Versionen, die sich im Falle von transaktionaler Konsistenz auf den Zeitraum zwischen zwei erfolgreichen Abschlüssen von Transaktionen erstreckt und somit konsistentes Checkpointing (siehe [ElJoPC92]) vereinfacht, während die Aktualität in nicht transaktionalen Systemen beim nächsten Ändern eines Datums oder dem Austausch einer Nachricht verloren geht. Zum Zweiten erfordert die Bestimmung einer konsistenten Datenbasis bei nicht transaktionalen Systemen spezielle Synchronisierung oder Protokollierung, während bei transaktionaler Konsistenz garantiert wird, dass alle sichtbaren Daten zu einem konsistenten Abbild gehören. Zur Verdeutlichung zeigt die folgende Darstellung 2.2 links

den Zustand eines ruhenden Clusters, bei dem die zu einem konsistenten Zustand gehörenden Seiten teilweise im Cluster repliziert sind. Im rechten Teil ist derselbe Cluster nach dem Start von Transaktionen mit der lokal veränderten Sicht auf die noch nicht bestätigten Daten abgebildet, jedoch ist der nach außen repräsentierte Zustand nach wie vor konsistent.

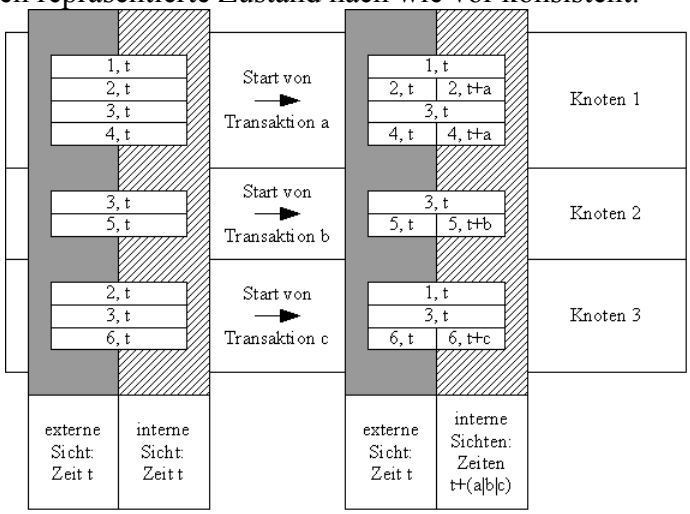

Abbildung 2.2: Nach außen immer konsistenter Zustand

Üblicherweise ist zwischen Geschwindigkeit im fehlerfreien Betrieb und Aktualität der Daten beim Checkpointing ein Kompromiss zu schließen (siehe [DuNaAV04]). Trotz niedriger Belastung des Clusters im laufenden Betrieb ist mit dem implementierten Prototyp eine schnelle Rücksetzung im Fehlerfall möglich (3), wie durch die Messungen in Kapitel 5.4 belegt wird. Auch bei kommerziellen Produkten wie dem IBM LoadLeveler liegen die Intervalle für Checkpoints im Cluster üblicherweise zwischen 15 Minuten und 2 Stunden (siehe [KaRoWM01]), während sie bei Plurix ohne signifikante Beeinträchtigung des Systems auf wenige Sekunden verkürzt werden können. Der Aufwand zur Bereitstellung dieser Funktionalität ist bei Systemen, die Transaktionen unterstützen, äußerst gering, da bereits aufgrund der Rücksetzbarkeit von Transaktionen alle grundlegenden Algorithmen zur Rücksetzung von Anwendungen vorhanden sind (4). Unter Beachtung der in Kapitel 4.3 entwickelten Anforderungen für rücksetzbare Systeme gilt dies sogar für die im System integrierten Gerätetreiber. Wie in Kapitel 2.2.5 gezeigt wurde, muss das Protokoll im fehlerfreien Betrieb für den Einsatz mit Pageserver und somit für Checkpointing nicht erweitert werden, für die Rücksetzung sind lediglich die beiden zusätzlichen Nachrichten „RecoverRequest“ und „RecoverOrder“ erforderlich, um die in Kapitel 2.2.7 diskutierten Schnittstellen zur Fehlererkennung vollständig unterstützen zu können. Mit der integrierten logischen Zeit (siehe Kapitel 2.2.2) wird automatisch eine gültige Datenbasis bei der Konsistenzierung verwendet, was in anderen Systemen spezielle Synchronisierung der Maschinen oder Analyse der verschickten Nachrichten erfordert (siehe [ChLaDS85], [ElAlSR96], [TsWaED99] und [AgbaRH02]).

3 Diskstrukturen zur Sicherung von Seiten

Die in den letzten Kapiteln diskutierte Struktur zur Umsetzung von transaktionaler Konsistenz mit Unterstützung von Sicherungspunkten (Checkpoints) kann als Grundlage zum Entwurf derjenigen Module eines Pageservers dienen, die sich in das bestehende System integrieren und mit dem Cluster kommunizieren. Für die erfolgreiche Implementierung eines dauerhaften Pageservers ist jedoch zusätzlich das Sichern der gesammelten Seiten auf einer Festplatte erforderlich.

Wie in den Kapiteln 2.1.2 und 2.1.6 bereits beschrieben wurde, kann die Sicherung der Daten einer Transaktion vom Ende der Transaktion entkoppelt oder an dieses gekoppelt werden. Bei einer gekoppelten Implementierung werden alle von einer Transaktion geänderten Seiten während der Veröffentlichungsphase an den Pageserver übertragen und die Transaktion erst nach erfolgreicher Sicherung der Daten als bestätigt gewertet. Somit sind alle bestätigten Transaktionen nach dem strengen ACID-Prinzip (siehe Kapitel 2.1.2) tatsächlich dauerhaft und auch bei nachfolgenden Fehlern persistent. Für den Clusterbetrieb mit Anwendungsprogrammen ist eine Entkopplung von Transaktionsbestätigung und Datensicherung jedoch möglich, da es sich bei den Anwendungen typischerweise nicht um unwiederholbare oder sicherheitskritische Steuerungen handelt. Sie ist darüber hinaus sogar wünschenswert, um das Sammeln und Sichern von Seiten optimieren und somit den Durchsatz an Transaktionen steigern zu können (Ansätze zur Optimierung der Strategie zum Sammeln von Seiten bei entkoppelten Verfahren werden in Kapitel 6.4 vorgestellt). Die nachfolgend beschriebenen Möglichkeiten zur Sicherung von Seiten können sowohl bei entkoppelten als auch gekoppelten Verfahren verwendet werden. Für die Verfahren wird angenommen, dass der zu sichernde Speicher durch einen 32 Bit Adressraum begrenzt wird, in maximal 1.048.576 Seiten zu je 4.096 Bytes eingeteilt ist und von einem Pageserver außerhalb des Clusters (siehe Kapitel 2.1.6) auf eine handelsübliche Festplatte mit 512 Bytes pro Sektor gesichert wird.

Die Vorteile beim Einsatz mehrerer Festplatten in einem Pageserver werden bei den vorgestellten Verfahren nicht genutzt, jedoch bleibt die Verwendung von RAID-Systemen davon unberührt, da diese beim logischen Zugriff wie ein einzelnes Medium erscheinen. Der Einsatz von redundanten RAID-Levels (siehe [ChLeRH94]) bietet sich sogar an, um den Betrieb auch bei einem Defekt einer Festplatte aufrecht erhalten zu können und somit die Ausfallsicherheit des Pageservers weiter zu erhöhen.

3.1 Klassifizierung von Diskstrukturen

Um die folgenden beschriebenen Ansätze zum Sichern von Seiten auf eine Festplatte bewerten zu können, werden Teilaspekte der Verfahren betrachtet, die von einer tatsächlichen Implementierung unabhängig sind:

1. Datenorganisation und Verwaltungsstrukturen auf der Festplatte: Die auf der Festplatte abgelegten Daten müssen je nach Verfahren in unterschiedlicher Weise verwaltet werden. Insbesondere Hauptspeicher-Listen mit freien Blöcken oder indizierbare Tabellen können den Zugriff auf Daten beschleunigen, benötigen jedoch auch Speicherplatz und müssen auf dem aktuellen Stand gehalten werden. Ebenso werden zur Speicherung von gleichen Datenmengen je nach Datenorganisation unterschiedlich viele Blöcke auf der Festplatte benötigt. Verschiedene Verfahren müssen bei der Fertigstellung eines Abbildes (Konsistenzierung) große Datenmengen auf der Festplatte kopieren, um mit der Sicherung von neuen Daten fortfahren zu können.
2. Bedarf und Ablauf der Reorganisation von gesicherten Daten: Bei Verfahren mit direkter Abbildung von zu sicherndem Speicher auf Festplattenplatz ist der Speicherplatzbedarf konstant und keinerlei Reorganisierung notwendig. Dagegen ist bei inkrementellen Verfahren vor Erschöpfung der verfügbaren Speicherkapazität eine Reorganisation erforderlich. Diese ist je nach Verfahren atomar oder inkrementell möglich und benötigt unterschiedlich viel Zeit.
3. Verwaltungsstrukturen im Hauptspeicher: Die verschiedenen vorgestellten Verfahren benötigen unterschiedliche Verwaltungsstrukturen und Tabellen im Hauptspeicher, deren Platzbedarf ebenso wie die benötigte Zeit für deren Aktualisierung im ständigen Betrieb und bei der Fertigstellung eines Abbildes verschieden ist.
4. Zugriffsverhalten und Durchsatz im Clusterbetrieb: Im fehlerfreien Clusterbetrieb ist für den Durchsatz des Pageservers das andauernde Schreiben von beliebigen Seiten maßgeblich. Je nach Verfahren sind zur Sicherung von Seitendaten oder zur Aktualisierung der benötigten Verwaltungsinformationen unterschiedlich viele Positionierungen der in der Festplatte vorhandenen Mechanik erforderlich, was im Vergleich zu Zugriffen ohne Positionierung sehr teuer ist (der Durchsatz sinkt etwa um Faktor 50, siehe dazu die Messungen in Kapitel 5.2).
5. Möglichkeiten zur Rekonstruktion bei Fehlern: Auf dem Pageserver muss auch bei Fehlern im Cluster ein konsistentes und möglichst junges Abbild verfügbar sein, um den Cluster wieder auf dieses Abbild zurücksetzen zu können. Fehler im Pageserver werden zwar als selten angenommen und müssen daher nicht hocheffizient behandelt werden, jedoch ist auch in einem solchen Fall eine Möglichkeit zur Wiederherstellung der Daten wünschenswert.

3.2 Bekannte Diskstrukturen

Im Folgenden werden die konzeptionell wichtigsten bekannten Diskstrukturen zur Sicherung von Seiten kurz beschrieben und bewertet sowie in der Zusammenfassung in Kapitel 3.2.5 tabellarisch miteinander verglichen. Ausführlichere Beschreibungen finden sich insbesondere in [SkibTV01], [FrenPT02] und [WendKV03].

3.2.1 Forced Write

Die einfachste Möglichkeit zur Sicherung von Seiten im 32 Bit Adressraum auf einer Festplatte ist das einfache Durchschreiben von Seiten auf eine der jeweiligen Seite eindeutig zugeordnete Position. Dieses in [HaRePT83] als „update in place“ bezeichnete Verfahren versucht, ein initiales, vollständiges Abbild des gesamten Adressraumes zu erstellen und geänderte Seiten sofort ohne zusätzliche Verwaltungsstrukturen zu ersetzen, so dass ein permanent aktuelles Abbild des zu sichernden Speichers auf der Festplatte gehalten wird.

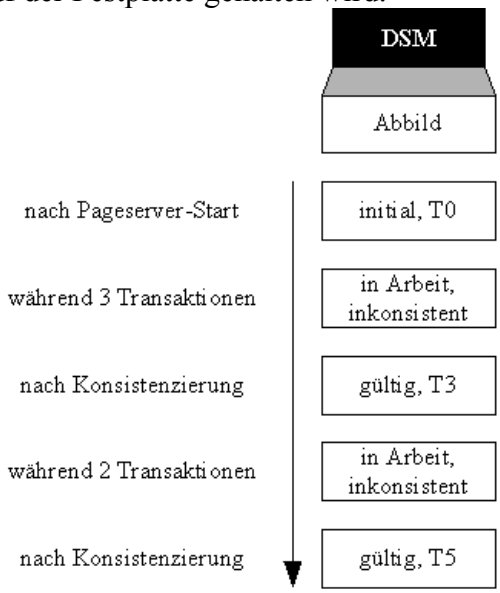

Abbildung 3.1: Forced Write

Das Beispiel zeigt die Daten auf der Festplatte im zeitlichen Verlauf: Das Abbild wird durch eine vollständige Spiegelung des verteilten Speichers initialisiert und repräsentiert somit den Zustand des verteilten Speichers zum Zeitpunkt T0 konsistent. Während der Abarbeitung von drei Transaktionen im Cluster wird das Abbild modifiziert und ist infolgedessen inkonsistent. Erst nach der Konsistenzierung im dritten Schritt ist das Abbild gültig und stellt den Zustand des verteilten Speichers zum Zeitpunkt T3 dar. Das Beispiel enthält zwei nachfolgende Transaktionen bis zur nächsten Konsistenzierung für den Zeitpunkt T5. Entsprechend dieses Musters zeigen die in den nächsten Kapiteln folgenden Graphiken die Arbeitsweise der jeweils vorgestellten Verfahren.

Der Speicherbedarf auf der Festplatte ist konstant und identisch zur Größe des zu sichernden Adressraums, im Hauptspeicher werden keine zusätzlichen Strukturen benötigt, und sowohl bei Lese- als auch Schreibzugriffen ergibt sich die Position der Seite auf der Festplatte direkt aus der Seitenadresse. Durch die bidirektional eindeutige Abbildung von Festplattenspeicher auf zu sichernden Speicher ist keine Reorganisation notwendig. Für die Konsistenzierung sind keine weiteren Operationen erforderlich, da das auf der Festplatte gesicherte Abbild ohne Metainformationen rekonstruiert werden kann. Dem steht jedoch im laufenden Betrieb mit wahlfreiem Zugriff eine sehr hohe Zahl an Kopfbewegungen gegenüber, die den Durchsatz bei aktuellen Festplatten auf unter 1 MB/s sinken lassen. Bei einem Fehler im Cluster ist das auf dem Pageserver vorhandene Abbild nur dann konsistent, wenn noch nicht mit der Sicherung der Seiten einer neuen Transaktion begonnen wurde und die Seiten aller bisher begonnenen Transaktionen vollständig sind. Bei einem Fehler im Pageserver ist keine Rücksetzung möglich, da das aktuelle Abbild typischerweise inkonsistent ist. Aufgrund fehlender Zusatzinformationen kann im Betrieb darüber hinaus gar nicht ermittelt werden, ob das aktuelle Abbild konsistent ist oder nicht.

Dieses sehr einfache Verfahren ist für die praktische Anwendung untauglich, da der Durchsatz viel zu niedrig ist und weder Fehler im Cluster noch Fehler im Pageserver zuverlässig behandelt werden können. Es bildet mit der direkten Umsetzung von Hauptspeicher auf Festplattenspeicher jedoch die Grundlage für das folgende Verfahren.

3.2.2 Copy

Beim sogenannten „Copy“ Verfahren (nach [HaRePT83] „Indirect Page Allocation“) wird ebenfalls ein initiales Abbild des gesamten Adressraums erstellt, das aber nicht direkt aktualisiert wird, sondern erst nach Erstellung einer Kopie. Somit existieren immer zwei vollständige Abbilder des zu sichernden Adressraums: „alt“ und „in Arbeit“. Anfragen werden aus dem konsistenten alten Abbild beantwortet, während geänderte Seiten in das in Arbeit einfließen. Dadurch ist letzteres nur nach einer Konsistenzierung am Ende einer Transaktion konsistent.

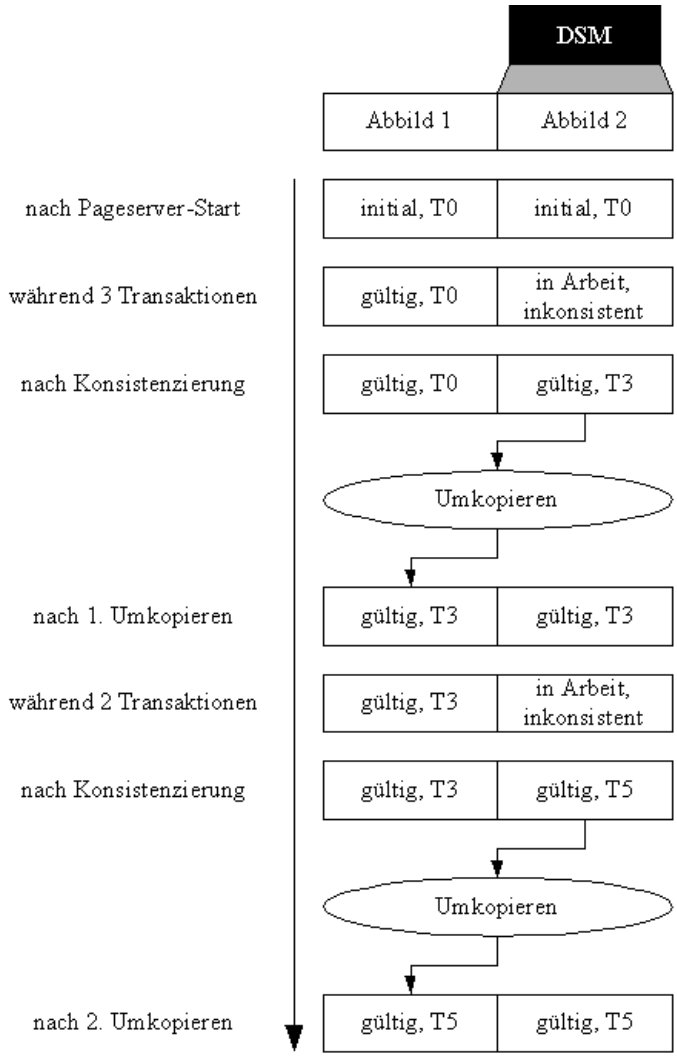

Abbildung 3.2: Copy

Der Speicherbedarf auf der Festplatte ist bei diesem Verfahren konstant und doppelt so hoch wie die Größe des zu sichernden Adressraums, zuzüglich eventuell vorhandener (siehe unten) Sektoren für den aktuellen Zustand. Wie bei „Forced Write“ kann die Festplatte sowohl bei Lese- als auch Schreibzugriffen direkt adressiert werden. Bei der Konsistenzierung des in Arbeit befindlichen Abbildes wird dieses jedoch über das alte Abbild kopiert, was bei heutigen Festplatten selbst bei maximalem Durchsatz etwa drei Minuten dauert. Währenddessen können keine Seiten gesichert und keine Seitenanfragen beantwortet werden.

Statt eines vollständigen Umkopierens zur Initialisierung eines neuen Abbildes kann im laufenden Betrieb auch protokolliert werden, welche Seiten ersetzt wurden. Dann müssen bei der Konsistenzierung des Abbildes nur alle nicht ersetzten Seiten aus der Generation vorher kopiert werden. Die in diesem Fall zu kopierende Datenmenge ist zwar geringfügig kleiner, jedoch ergibt sich aus der zusätzlich erforderlichen Protokollierung eine komplexere Verwaltung.

Durch die sehr hohe Zahl an Kopfbewegungen mit einem resultierenden Durchsatz von weniger als 1 MB/s ist dieses Verfahren zwar genauso schlecht wie „Forced Write“, jedoch kann bei einem Fehler im Cluster auf das hier vorhandene konsistente Abbild zurückgesetzt werden. Sollen auch Fehler im Pageserver nicht zu einem Verlust des letzten Abbildes führen, können auf der Festplatte Beginn und Abschluss des Kopiervorgangs zur Konsistenzierung vermerkt werden. Somit kann beim Neustart des Pageservers ermittelt werden, ob das Abbild „alt“ sofort benutzbar ist (Absturz nicht während des Kopiervorgangs) oder dieses Abbild teilweise überschrieben wurde (Absturz während des Kopiervorgangs) und infolgedessen durch das Abbild „in Arbeit“ ersetzt werden muss. Mehrere Generationen von Abbildern können mit diesem Verfahren leicht unterstützt werden, indem mehrere Bereiche des Typs „alt“ vorhanden sind und zu jedem Bereich das Alter vermerkt wird.

Obwohl dieses Verfahren eine Fehlerbehandlung ermöglicht, ist es für die praktische Anwendung untauglich, da der Durchsatz viel zu niedrig ist und die Konsistenzierung sehr viel Zeit in Anspruch nimmt.

3.2.3 Logging

Die Konsistenz lässt sich alternativ zur vollständigen Kopie wie im vorigen Verfahren auch mit einer Protokollierung von Änderungen (siehe [GiScDB76] und [OuDoBI89], in [SkibTV01] und [WendKV03] als „Log-File“ bezeichnet) sicherstellen. Auch hier wird zu Beginn ein vollständiges Abbild erstellt, jedoch werden im laufenden Betrieb nur geänderte Seiten mit dazugehörenden Metainformationen wie der Seitennummer, einer Prüfsumme und der logischen Zeit fortlaufend in einem dafür vorgesehenen Festplattenbereich gesichert. Bei dessen Erschöpfen müssen die Änderungen in das vollständige Abbild eingearbeitet werden, danach kann der Logbereich erneut genutzt werden. Das aktuell gültige, konsistente Abbild setzt sich aus dem vollständigen Abbild und den zu vollständig gesicherten Transaktionen gehörenden Seiten zusammen. Für die Verwaltung des aktuell gültigen Abbildes bei Seitenanfragen wird eine Tabelle im Hauptspeicher benötigt, die eine Umsetzung von Speicheradressen auf Festplattenpositionen ermöglicht.

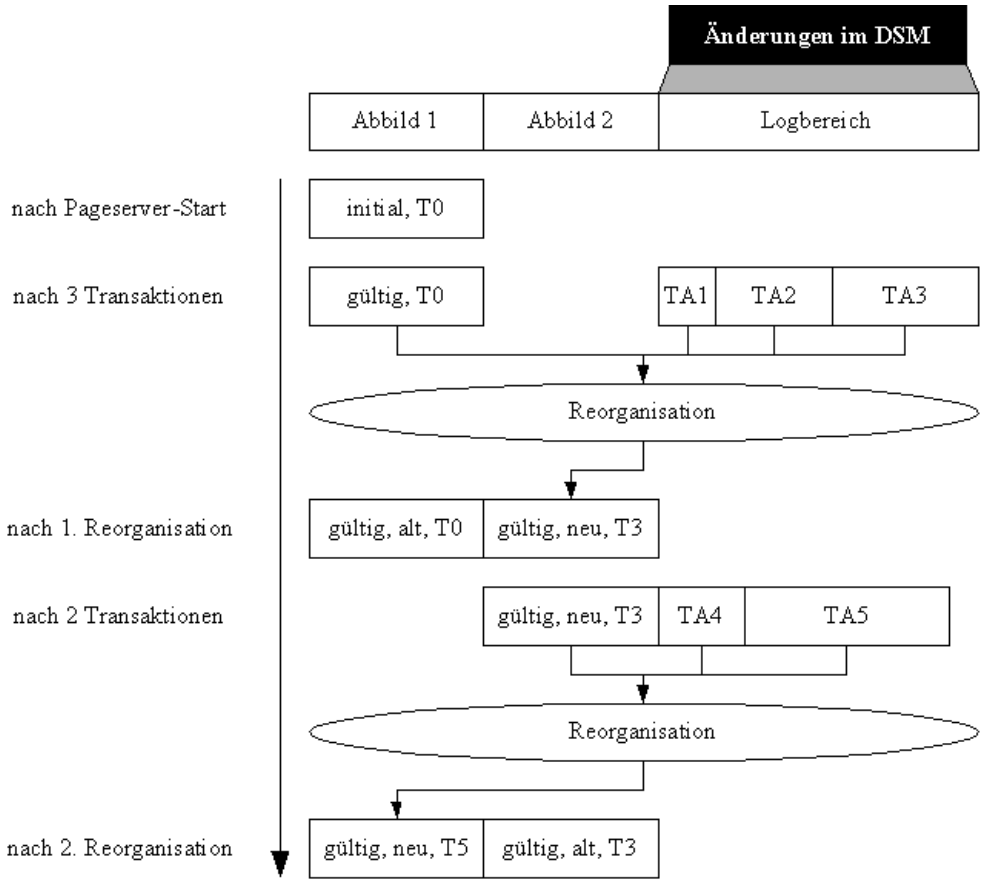

Abbildung 3.3: Logging

Für das initiale Abbild ist ein konstanter Speicherplatz in der Größe des zu sichernden Adressraums erforderlich, der verbleibende Speicherplatz steht dem Logbereich zur Verfügung. Der Logbereich enthält sowohl Seitendaten als auch Metainformationen, wobei letztere im Vergleich zu den Seitendaten wenig Platz (etwa 16 Bytes für Seitennummer, logische Zeit, Prüfsumme und eventuell benötigte Flags) benötigen und auf der Festplatte unterschiedlich um die Seitendaten angeordnet werden können. Für die nachfolgende Abschätzung wird der einfachste Ansatz gewählt, der ohne Kopfbewegung der Festplattenmechanik auskommt und bei dem im Logbereich für jede gesicherte Seite ein Sektor mit Metainformationen existiert (Overhead hier 1:8). Durch diese Anordnung kann im laufenden Betrieb sehr schnell auf die Festplatte geschrieben werden, da keinerlei Kopfbewegung erforderlich ist. Bei Anfragen kann jedoch die Festplattenposition nicht mehr direkt aus der Seitenadresse errechnet werden, sondern muss indirekt über eine Tabelle ermittelt werden. Die Tabelle zur Umsetzung von Speicheradressen auf Festplattenpositionen benötigt pro Seite im zu sichernden Speicher einen Eintrag, der ausreichend Platz für eine Adressierung der Festplatte bietet und somit typischerweise 32 Bit umfasst (entsprechend maximal etwa 18 Terabyte bei Blöcken zu 9 Sektoren mit je 512 Bytes), wodurch sich ein maximaler Bedarf

von 4 MB im Hauptspeicher ergibt. Spätestens bei Erschöpfung des Logbereichs muss eine vollständige Reorganisation durchgeführt werden. Hierbei wird der Logbereich abgearbeitet und die darin vermerkten Seiten in das vollständige Abbild übertragen. Dies kann durch eine weitere Tabelle im Hauptspeicher für eine Abbildung von Speicheradressen auf neue Festplattenpositionen beschleunigt werden, um nicht den gesamten Logbereich abarbeiten zu müssen, sondern nur die jeweils tatsächlich aktualisierten Seiten zu kopieren. Aber auch dieser optimierte Vorgang kann sehr lange dauern, da je nach verfügbarem Puffer im Hauptspeicher und Größe des Logbereichs auf der Festplatte sehr viele Kopfbewegungen erforderlich sind (bereits bei effektiv 10% geänderten Seiten sind je nach Verfahren mindestens 100.000 bis 200.000 Positionierungen notwendig, für die auch eine schnelle Festplatte mit einer Zugriffszeit von 7 ms im besten Fall noch über 12 Minuten benötigt). Bei einem Fehler sowohl im Cluster als auch im Pageserver lässt sich aus dem vollständigen Abbild und dem gesamten Logbereich immer ein konsistentes Abbild rekonstruieren.

Dieses Verfahren ermöglicht einerseits, Fehler im Cluster und im Pageserver zuverlässig zu behandeln, und bietet andererseits während des fehlerfreien Betriebs eine hohe Schreibleistung. Die Reorganisation des Festplattenspeichers erzeugt jedoch für die meisten Anwendungsgebiete inakzeptabel lange Verzögerungen. Wenn der Logbereich so groß dimensioniert werden kann, dass die Reorganisation gezielt in Zeiten äußerst niedriger Last durchgeführt werden kann (zum Beispiel nachts bei Anwendungen in einem Büro), ist der Einsatz dieses Verfahren jedoch vorstellbar.

3.2.4 Age Segment

Das in [SkibTV01] und [WendKV03] als „Age Segment“ bezeichnete Verfahren orientiert sich an den in [RoOuDI91] und [GhIeMS94] vorgestellten Diskstrukturen. Ein vollständiges Abbild des zu sichernden Speichers auf einen kontinuierlichen Festplattenbereich wie in den bisher vorgestellten Verfahren existiert hier nicht mehr. Stattdessen wird die komplette Festplatte als logischer Ringpuffer verwendet und in Segmente unterteilt, die jeweils konstanten Platz für mehrere Seiten und die für diese Seiten erforderlichen Metadaten bieten. Dabei wird für jede Seite mitgeführt und während der Konsistenzierung auf der Festplatte aktualisiert, welche Seiten „aktiv“, „unbestätigt“, „zukünftig“ oder „veraltet“ (siehe unten) sind. Ebenso wird für jedes Segment bei der Konsistenzierung aktualisiert, ob es für das aktuelle Abbild benötigt wird und falls ja, welche Teile genutzt werden. Im Laufe der Zeit werden die Segmente leerer und können schließlich während einer Reorganisation zusammengefasst werden. Dabei wird während der Reorganisation das Ziel verfolgt, volle Segmente zu gruppieren, um die leeren Segmente möglichst aneinander grenzen zu

lassen. Somit kann dort während des laufenden Betriebs linear schreibend und dadurch mit sehr hohem Durchsatz zugegriffen werden.

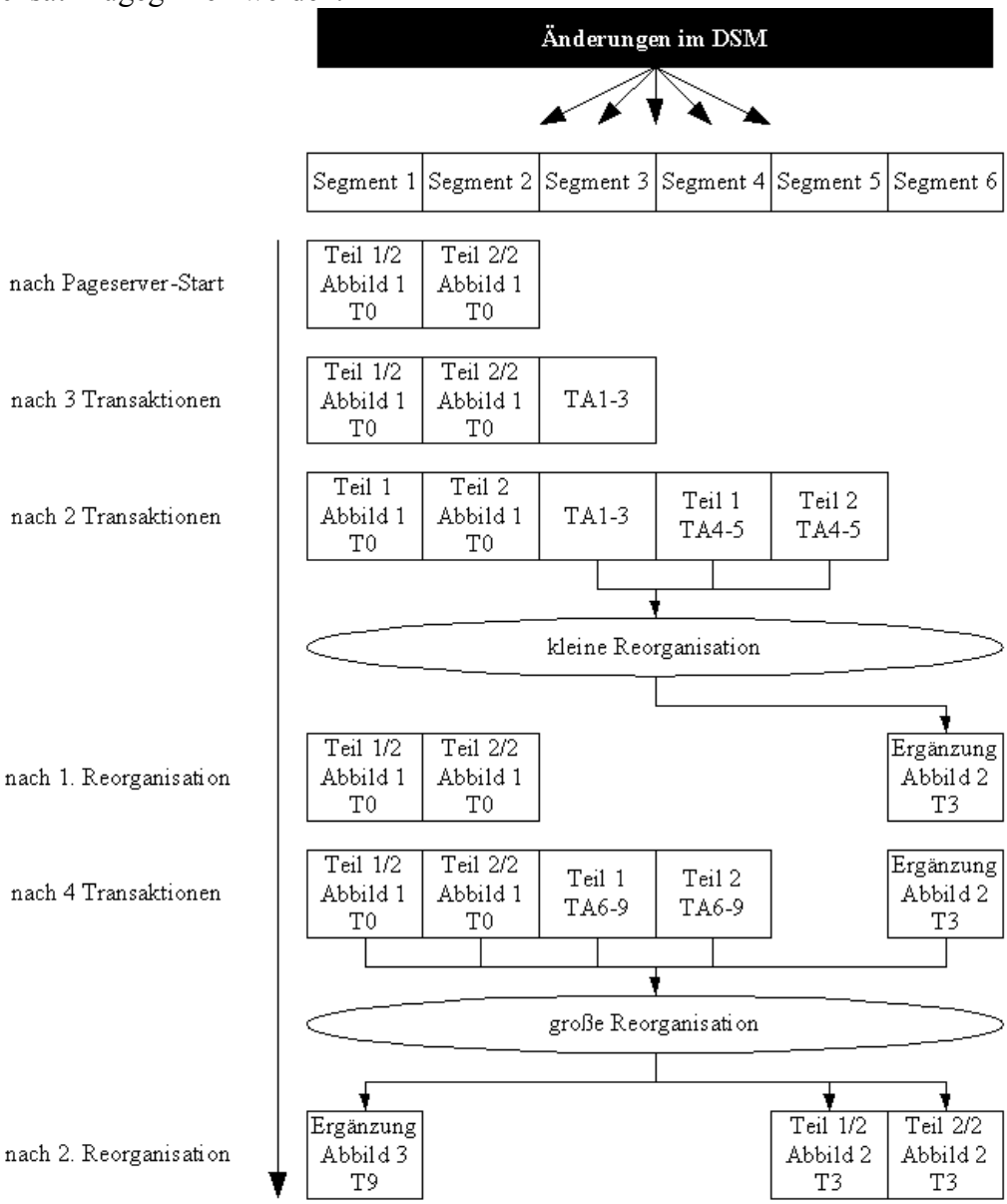

Abbildung 3.4: Age Segment

Wie bei „Logging“ kann der gesamte zur Verfügung stehende Speicherplatz der Festplatte genutzt werden. Die Festplatte wird in gleich große Segmente unterteilt, die jeweils die Daten mehrerer Seiten sowie die für das jeweilige Segment benötigten Metadaten enthalten. Die in [RoOuDI91] empfohlene Segmentgröße von etwa 1 Megabyte bündelt viele Seiten in ein Segment, wobei Daten während der Sicherung als „unbestätigt“ markiert werden und erst während der Konsistenzierung über den Zwischenzustand „zukünftig“ in den Zustand „aktiv“ übergehen. Während des laufenden Betriebs können die Segmente in aufsteigender Reihenfolge und im Optimalfall an direkt aufeinander folgenden Positionen geschrieben werden, so dass sich ein sehr

hoher Durchsatz wie bei Logging ergibt. Für Anfragen wird ebenfalls wie bei Logging eine etwa 4 MB große Tabelle im Hauptspeicher benötigt, um zu der angefragten Seitenadresse eine Festplattenposition ermitteln zu können. Zur Verwaltung der Segmente wird im Hauptspeicher pro Segment gespeichert, ob es frei (1 Bit) oder aktiv ist (1 Bit), wie oft es referenziert (30 Bit) wird und wie alt es ist (64 Bit). Bei einer Festplatte mit 100 Gigabyte ergeben sich etwa 100.000 Segmente, woraus sich ein Speicherbedarf von etwas mehr als einem Megabyte für die Verwaltung der Segmente ergibt. Darüber hinaus muss für jedes seit der letzten Konsistenzierung geschriebene Segment protokolliert werden, welche Seiten im Zustand „unbestätigt“ sind, da die korrespondierenden Einträge auf der Festplatte während der Konsistenzierung aktualisiert werden müssen. Der dafür erforderliche Speicherplatz ist abhängig von der Menge der seit der letzten Konsistenzierung geschriebenen Segmente. Bei einer Konsistenzierung wird dieser Puffer vollständig abgearbeitet, indem alle ersetzten Seiten als „alt“ markiert werden, die neuen Versionen aller Seiten den Zustand „zukünftig“ erhalten, sodann die bisher aktiven Seiten ebenfalls als „alt“ markiert und zum Abschluss alle als „zukünftig“ markierten Seiten „aktiv“ werden. Nach erfolgreicher Aktualisierung der Festplatte kann die Tabelle auf diesen Stand gebracht und mit linearem Schreiben weitergearbeitet werden. Für die Konsistenzierung sind pro geänderter Seite drei Kopfbewegungen (für die drei Übergänge „unbestätigt“ nach „zukünftig“, „aktiv“ nach „alt“, „zukünftig“ nach „aktiv“) erforderlich, bei mehrfach geänderten Seiten für jede zusätzliche Version eine weitere Kopfbewegung (von „unbestätigt“ nach „alt“). Sollte der Puffer für die Protokollierung der Änderungen von Segmenten vor Beginn der Konsistenzierung erschöpft sein, müssen die bisher als „unbestätigt“ markierten und bereits durch neue Versionen ersetzten Seiten als „alt“ markiert werden, um Pufferspeicher freigeben zu können. Die Konsistenzierung eines Abbildes muss atomar erfolgen und benötigt sehr viele Kopfbewegungen, so dass die Konsistenzierung eines Abbildes sehr teuer ist (bei einer auch nur einmaligen Änderung von 2000 Seiten seit der letzten Konsistenzierung benötigt allein die Aktualisierung der Metadaten in den Segmenten bei schnellen Festplatten mit 7 ms Zugriffszeit knapp eine Minute). Eine Reorganisation der Segmente ist bei ausreichend großen Festplatten nur selten notwendig und kann zu beliebigen Zeitpunkten inkrementell erfolgen. Eine Fehlerbehandlung bei Fehlern im Cluster ist jederzeit möglich, da immer ein konsistentes Abbild konstruiert werden kann. Aufgrund der stufenweisen Aktualisierung der Metainformationen auf der Festplatte ist eine Fehlerbehandlung darüber hinaus auch bei Fehlern im Pageserver immer möglich.

Dieses Verfahren ermöglicht einen sehr hohen Durchsatz zwischen zwei Konsistenzierungen und kann Fehler im Cluster und im Pageserver behandeln, jedoch ergibt die hohe Komplexität mit mehrfacher Verwaltung und Protokollierung von Seitenalter und Segementzuständen eine in

[WendKV03] als „mangelhaft“ bezeichnete Verständlichkeit. Für die Protokollierung von geänderten Segmenten ist unter Umständen sehr viel Speicher notwendig, und die für die Konsistenzierung benötigte Zeit ist zu lang. Beim Einsatz mehrerer Festplatten kann dieser Nachteil zwar je nach System abgeschwächt werden, bleibt prinzipiell in dieser Form jedoch bestehen.

3.2.5 Zusammenfassung und Zielsetzung

Die vorigen Kapitel bieten einen Überblick über bekannte Verfahren und Strukturen zur Sicherung von Speicherseiten auf einer Festplatte. Eine Entscheidung für oder gegen ein Verfahren ist abhängig vom jeweiligen Einsatzgebiet, wobei Kompromisse zwischen Platzbedarf für Metadaten, Komplexität des Verfahrens, Zeitbedarf für Konsistenzierung und Reorganisation, Häufigkeit und Unterbrechbarkeit der Reorganisation, benötigtem Hauptspeicher sowie Ausfallsicherheit eingegangen werden müssen. Für das Umfeld von Plurix ist ein hoher Durchsatz bei dennoch niedrigem Aufwand für Konsistenzierung und einer inkrementellen Reorganisation wichtig, um die Arbeitsfähigkeit und Leistung im Cluster nicht übermäßig zu beeinträchtigen. Die folgende Tabelle stellt die einzelnen Verfahren anhand der in Kapitel 3.1 diskutierten Kriterien einander gegenüber und enthält in der letzten Spalte die Eigenschaften eines für Plurix wünschenswerten Verfahrens:

Kriterium	***Forced W.***	***Copy***	***Logging***	***Age S.***	***erwünscht***
Komplexität	sehr niedrig	niedrig	mittel	hoch	niedrig
Durchsatz	sehr niedrig	sehr niedrig	sehr hoch	hoch	hoch
Aufwand Konsistenzierung	keiner	sehr hoch	niedrig	hoch	niedrig
Zwang zur Reorganisation	keiner	keiner	regelmäßig	selten	selten
Aufwand Reorganisation	-	-	hoch	niedrig	niedrig
Ablauf der Reorganisation	-	-	atomar	inkrementell	inkrementell
Bedarf an Hauptspeicher	sehr niedrig	sehr niedrig	niedrig	mittel	niedrig
Fehlerbehandlung möglich	nein	ja	ja	ja	ja

Tabelle 3.1: Vergleich bestehender Verfahren zur Sicherung

Gesucht ist also ein Verfahren, das den sehr guten Durchsatz und die schnelle Konsistenzierung von Logging mit der inkrementell durchführbaren und selten notwendigen Reorganisation von Age Segment kombiniert. Eine niedrige Komplexität ist für den Einsatz des Systems in der Lehre zwar vorteilhaft, ist jedoch ebenso wie ein niedriger Bedarf an Hauptspeicher nicht oberstes Entwicklungsziel. Das folgende Kapitel widmet sich daher der Konstruktion eines aus obiger Sicht möglichst effizienten und dennoch verständlichen Verfahrens.

3.3 Entwickelte Diskstruktur Linear Segment

3.3.1 Entwicklungsansatz

Um bei aktuellen Festplatten mit einer an rotierende Massen und bewegliche Leseköpfe gekoppelten Mechanik einen hohen Durchsatz im fehlerfreien Betrieb zu erreichen, muss aufsteigend und möglichst zusammenhängend geschrieben werden (siehe [RoOuDI91], [ScAiLR03] und [ScAiMD03]). Im Gegensatz zu Datenbanken, bei denen neben der Sicherung von neuen Daten auch sehr viele Anfragen beantwortet werden müssen und dadurch bereits unabhängig von den Schreibzugriffen viele Kopfbewegungen stattfinden, treten Anfragen an den Pageserver wie in anderen für Transaktionen ausgelegten Betriebssystemen auch unter Plurix nur selten auf (siehe [SeStTS90]). Insbesondere bei einer Rücksetzung auf ein altes Abbild werden zwar viele Seiten angefordert, nach erfolgreichem Start des Clusters kann aber im laufenden, fehlerfreien Betrieb von einer hauptsächlich sichernden Tätigkeit des Pageservers ausgegangen werden.

Um die Zahl der Kopfbewegungen minimal zu halten und somit den maximal möglichen Durchsatz zu erreichen, ist also eine Struktur gesucht, die es erlaubt, im laufenden Betrieb linear aufsteigend zu schreiben und die Reorganisation inkrementell durchzuführen. Age Segment liefert dafür eine gute Ausgangsbasis, da außer der Zeit zur Konsistenzierung alle für Plurix wichtigen Kriterien erfüllt sind. Der hohe Aufwand zur Konsistenzierung bei Age Segment resultiert aus der Aktualisierung bisheriger Segmente, die über die gesamte Festplatte verstreut liegen können und für ihre Aktualisierung eine Vielzahl an Kopfbewegungen erfordern. Soll dieser Aufwand vermieden werden, muss die Sicherung von Seiten ohne Modifikation bereits geschriebener Daten erfolgen, insbesondere ohne die Aktualisierung bereits geschriebener Metadaten („Log-Only-Approach“, siehe [NoBrWO97]). Viele Abhängigkeiten innerhalb der Verwaltung sowie die je nach Zeitpunkt unterschiedlichen Bedeutungen von Werten (siehe [SkibTV01]) führen bei Age Segment zu einer negativen Bewertung des als zweitrangig geführten Kriteriums der Komplexität, so dass eine Vereinfachung für ein in der Lehre eingesetztes Verfahren wünschenswert wäre. Durch eine klare Struktur im Aufbau der Segmente und im Datenfluss sowohl bei der Sicherung als auch bei Anfragen kann das Verfahren in der Lehre leicht vermittelt werden. Im folgenden Kapitel wird eine verständliche Struktur vorgestellt, die übersichtlich implementierbar ist und durch die Berücksichtigung der Eigenschaften von Festplatten einen äußerst hohen Durchsatz ermöglicht (siehe dazu auch die Messungen Kapitel 5.4).

3.3.2 Datenorganisation auf der Festplatte

Die Sicherung und Bereitstellung von Seitendaten mit Hilfe eines persistenten Mediums ist die eigentliche Aufgabe des Pageservers. Um die Seitendaten verwalten und im Fehlerfall zuverlässig wiederfinden zu können, sind neben den Seitendaten jedoch zusätzliche Metadaten erforderlich, die die Eigenschaften der gesicherten Seitendaten beschreiben. Beim Einsatz einer einzelnen Festplatte könnte die Festplatte also in einen Bereich für Seitendaten und einen Bereich für Metadaten aufgeteilt werden, was jedoch aufgrund der dann erforderlichen Kopfbewegungen nicht empfehlenswert ist. Stattdessen kann bei einer Verschränkung von Metadaten und Seitendaten beim Zugriff auf eine Seite und ihre Metadaten auf die erforderliche Kopfbewegung zur Positionierung zwischen den beiden Datenbereichen verzichtet werden, beim Zugriff auf fortlaufende Seiten entfällt auch die Bewegung auf die alte Position. Verfahren mit mehreren Festplatten können hier teilweise ausgefeiltere Techniken nutzen, die jedoch meist auch mit erhöhter Komplexität einhergehen.

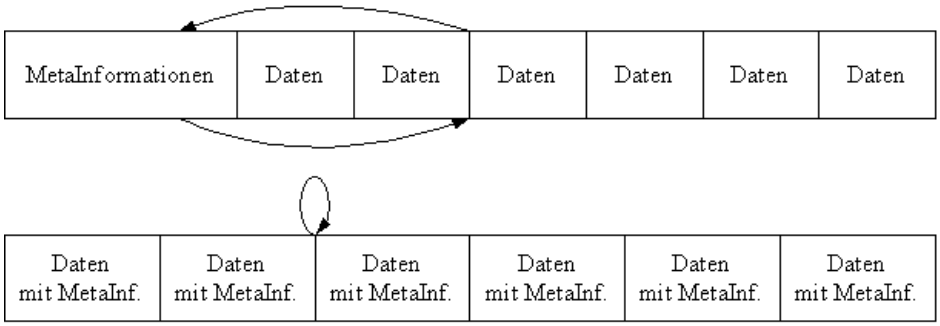

Abbildung 3.5: Kopfbewegungen je nach Position der Metadaten

Die für eine Seite benötigten Metadaten setzen sich wie folgt zusammen (in Klammern ist die im Prototyp verwendete Zahl an Bits und die Bezeichnung angegeben):

- Adresse (32 Bit `logAddrWFlags`): Die logische Adresse der Seite im verteilten Speicher mit den für das Cluster-System relevanten Attributen. Diese werden bei Seitenanfragen durch den Pageserver durch den Kern im Cluster gesetzt und von diesem zur Integration von Seitendaten wieder benötigt.
- CRC Prüfsumme (16 Bit `crc`): Um Übertragungsfehler oder Medienfehler erkennen zu können, wird zu jeder Seite eine Prüfsumme gespeichert. Sollte die Prüfsumme beim Laden einer gesicherten Seite nicht identisch mit der hier abgelegten früheren Prüfsumme sein, kann die Korrektheit der Seitendaten nicht gewährleistet werden, und die Abbilder mit Referenz auf diese Version der Seite müssen verworfen werden.

- Flags (16 Bit `pageflags`): Platz für Flags des Pageservers. In der Implementierung des Prototyps wird nur ein Bit zur Markierung von bereits ungültig gewordenen Seiten verwendet.
- Zeitstempel der letzten Änderung (64 Bit `lChanged`): Die logische Nummer der letzten Transaktion, die diese Seite verändert hat, wird benötigt, um bei einer Rekonstruktion im Fehlerfall die aktuellste Version einer Seite zu ermitteln.
- Zeitstempel beim Empfang (64 Bit `lSeen`): Dieser Zeitstempel enthält die logische Zeit während der Anlieferung dieser Seite. Die gesicherten Seitendaten waren mindestens während der gesamten Zeitspanne von `lChanged` bis `lSeen` gültig. Dieser Wert ist im laufenden Betrieb und bei einfachen Fehlern des Pageservers irrelevant, kann jedoch bei einer Ermittlung von zusätzlichen gültigen Abbildern im Fehlerfall verwendet werden (siehe Kapitel 6.2).
- Zeitstempel bei Sicherung (64 Bit `cmdTime`): Die logische Zeit während der Sicherung dieser Seite ist zur zeitlichen Ordnung der gesicherten Daten im Fehlerfall des Pageservers erforderlich.

Die insgesamt 32 Bytes an Zusatzinformationen ergeben in Summe mit den Seitendaten 4128 Bytes pro Seite, was nicht verlustfrei auf Sektoren mit 512 Bytes Größe abgebildet werden kann. Um den Verlust zu minimieren, können die Metadaten von einigen wenigen Seiten und deren Seitendaten zu einem Segment zusammengefasst werden, das dann aus einem Informationssektor und den je acht Datensektoren einer festen Anzahl (`PAGESPERBLOCK`) an Seiten besteht. Die Größe der Segmente sollte einerseits möglichst klein gewählt werden, um erstens eine schnelle Abarbeitung zu ermöglichen, um zweitens den Overhead bei Anfragen auf eine einzelne Seite eines Segments gering zu halten, und um drittens das Risiko der internen Fragmentierung von Segmenten (siehe Kapitel 3.3.3) zu reduzieren. Andererseits sollten nicht zu wenig Seiten in einem Segment gespeichert werden, um den Platz im Informationssektor so weit wie möglich nutzen zu können. Beim segmentweisen Zugriff auf die Festplatte kann der für alle Seiten dieses Segments identische Zeitstempel (`cmdTime`) einmalig abgelegt werden, wodurch der Bedarf von 32 auf 24 Bytes pro Seite gesenkt wird und sich eine maximale Zahl von 20 Seiten pro Segment ergibt. Da die Daten im Informationssektor äußerst sicherheitskritisch sind, wird eine Absicherung durch eine CRC-Prüfsumme empfohlen, so dass sich für den Informationssektor insgesamt folgender Aufbau ergibt:

- Zeitstempel bei Sicherung (64 Bit `cmdTime`): Die logische Zeit während der Sicherung dieses Segments repräsentiert den logischen Zeitpunkt der Sicherung aller enthaltenen Seiten und ist zur zeitlichen Ordnung der gesicherten Daten im Fehlerfall des Pageservers erforderlich.

- Erkennungsmarke 1 (64 Bit `magicID`): Um zufällig vorhandene Daten auf der Festplatte nicht irrtümlicherweise als Informationssektor zu interpretieren, wird hier eine spezielle Marke zur Erkennung abgelegt.
- Informationen zu den enthaltenen 1 bis 20 Seiten (24*`PAGESPERBLOCK` Bytes): Die oben beschriebenen Informationen für die in diesem Segment gesicherten Seiten werden der Reihe nach abgelegt, insbesondere die Seitenadressen und Prüfsummen zu jeder einzelnen Seite.
- Padding-Puffer (488-24*`PAGESPERBLOCK` Bytes): Je nach Zahl der Seiten pro Segment sind unterschiedlich viele Bytes im Informationssektor ungenutzt.
- Flags (32 Bit `segflags`): Platz für Flags des Pageservers. In der Implementierung des Prototyps wird nur ein Bit zur Markierung des letzten Segments für ein vollständiges Abbild verwendet.
- CRC Prüfsumme (16 Bit `crc`): Diese Prüfsumme ermöglicht die Erkennung von Fehlern im Informationssektor. So werden insbesondere auch die im Informationssektor abgelegten Adressen und Prüfsummen der Seitendaten geschützt.
- Erkennungsmarke 2 (16 Bit `idcd`): Die zweite Erkennungsmarke reduziert in Kombination mit der ersten die Wahrscheinlichkeit einer irrtümlichen Interpretation alter Daten auf der Festplatte weiter. Dadurch kann üblicherweise auf ein initiales Löschen der Festplatte verzichtet werden.

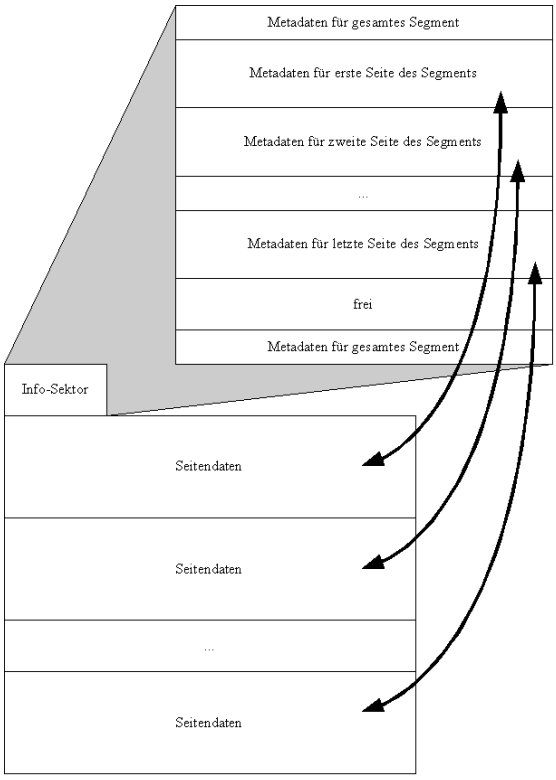

Abbildung 3.6: Aufbau eines Segments

Durch die Verwendung der hier vorgestellten Struktur kann im laufenden Betrieb mit ausschließlich linearem Zugriff auf die Festplatte geschrieben werden. Die Metadaten sind sehr nahe an den Seitendaten, so dass bei Anfragen jeweils das gesuchte Segment vollständig in den Hauptspeicher geladen und die angefragte Seite effizient validiert werden kann. Zur schnellen Abarbeitung von Anfragen empfiehlt sich das Vorhalten einer Tabelle (genannt: Seiten-Segment-Tabelle), in der für jede im verteilten Speicher die Nummer desjenigen Segments abgelegt wird, in dem sich die Daten zu dieser Seite auf der Festplatte befinden. Sollen mehrere Abbilder unterstützt werden oder soll auch der Rückfall auf eine weit zurückliegende Version schnell durchführbar sein, können mehrere solche Tabellen vorgehalten werden, deren Platzbedarf bei jeweils knapp 4 MB liegt (pro Seite im verteilten Speicher ist bei 20 Seiten pro Segment ein 32 Bit Wert ausreichend, um eine Festplatte oder Partition mit 322 Terabyte zu adressieren). Diese Tabellen sollten entweder bei der Konsistenzierung aller oder ausgewählter Abbilder oder vor dem Herunterfahren des Pageservers auf die Festplatte gesichert werden, um schnelle Anlaufzeiten beim Neustart des Pageservers zu ermöglichen. Werden die Tabellen nicht nur beim Herunterfahren des Pageservers gesichert, so kann auch bei unerwartetem Ausfall des Pageservers schnell auf ein konsistentes Abbild zurückgegriffen und die Suche über die gesamte Festplatte nach den aktuellen Versionen des letzten Abbildes vermieden werden. Für die Sicherung der Tabellen wird ein konstanter Bereich am

Anfang oder Ende der Festplatte empfohlen, so dass die Tabellen nicht auf mehrere Segmente aufgeteilt werden müssen und das Laden der Tabellen kein Suchen der Segmente und keinerlei Kopfbewegungen erfordert. Die Kosten für die Kopfbewegung vor dem Schreiben der Tabellen sind beim Herunterfahren des Pageservers irrelevant, bei einer Sicherung im Rahmen der Konsistenzierung sind die zwei Kopfbewegungen zum Tabellenbereich und zurück zum Datenbereich im Verhältnis zur Datenmenge der Tabellen und in Anbetracht der deutlichen Vereinfachung vertretbar.

Da die bisher geschriebenen Metadaten nicht aktualisiert werden müssen, sondern das aktuelle Abbild entweder über die vorgehaltenen Tabellen oder, falls diese aufgrund eines Stromausfalls oder Ähnlichem nicht gesichert werden konnten, über eine Suche nach der jeweils aktuellsten Version aller Seiten wiederhergestellt werden kann, kann mit dem vorgestellten Verfahren das in [NoBrWO97] empfohlene „Log-Only“-Verhalten mit der damit verbundenen maximal möglichen Schreibleistung erreicht werden. Das benötigte logisch unendlich große Medium ist physikalisch jedoch begrenzt, so dass spätestens bei Erreichen der Festplattenkapazität entweder eine Reorganisation der Daten oder zumindest eine Suche nach freigewordenen Segmenten erfolgen muss.

3.3.3 Möglichkeiten zur Reorganisation

Zu Beginn der Sicherung sind alle Segmente der Festplatte leer, nach ihrem ersten Beschreiben sind sie üblicherweise vollständig genutzt. Durch die erneute Sicherung von aktualisierten Seitendaten in anderen Segmenten werden die Daten aber teilweise nicht mehr benötigt, so dass die Segmente nach und nach irrelevante Daten enthalten.

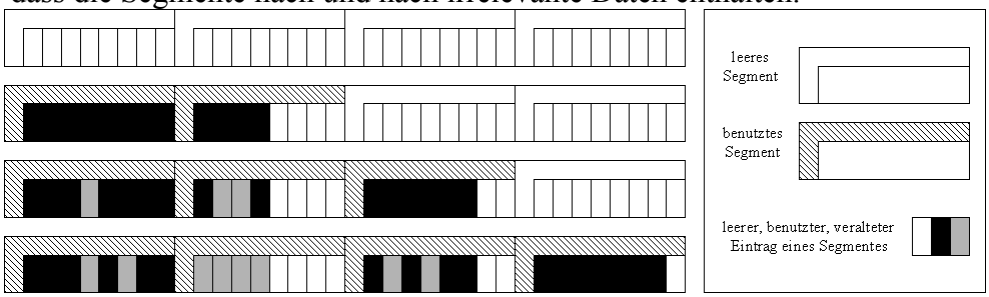

Abbildung 3.7: Beispiel für Segmente im Lauf der Zeit

Spätestens beim Beschreiben der letzten freien Segmente müssen in den bereits benutzten Segmenten freie Stellen gesucht werden, um diese wiederverwenden zu können. Dafür stehen die in der folgenden Abbildung 3.8 dargestellten und anschließend diskutierten Möglichkeiten zur Verfügung:

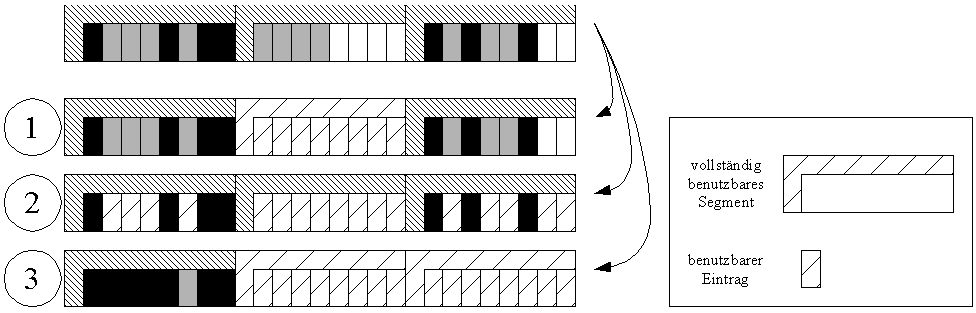

Abbildung 3.8: Möglichkeiten der Reorganisation

1. Verwendung von vollständig nicht mehr benötigten Segmenten: Dies ist die einfachste Möglichkeit, bereits beschriebene Segmente wiederzuverwenden, da keine alten Seitendaten von der Festplatte gelesen werden müssen. Die interne Fragmentierung von Segmenten kann allein mit diesem Verfahren jedoch nicht gezielt verringert werden.
2. Verwendung des Speichers von nicht mehr benötigten Seiten innerhalb von Segmenten: Diese sehr feingranulare Wiederverwendung beseitigt die interne Fragmentierung von Segmenten vollständig, jedoch muss für jedes Segment analysiert werden, welche Seiten noch benötigt werden. Die Datenbereiche dieser Seiten müssen dann entweder beim Wiederbeschreiben des Segments übersprungen oder vorher gelesen werden, um später wieder ein vollständiges Segment schreiben zu können.
3. Zusammenfassen der noch benötigten Seiten innerhalb von teilweise belegten Segmenten mit anschließender Verwendung der frei gewordenen Segmente in zwei Schritten: Im ersten Schritt werden die noch benötigten Seiten aus vielen teilweise belegten Segmenten in einen Puffer geladen, neu gruppiert und in noch freie Segmente geschrieben, so dass im zweiten Schritt die teilweise belegten Segmente freigegeben werden können. Da weniger vollständig gefüllte Segmente geschrieben werden als teilweise belegte Segmente gelesen wurden, stehen somit mehr freie Segmente als vor der Operation zur Verfügung. Hierbei reduziert sich nicht nur die interne Fragmentierung, sondern es erfolgt auch eine Gruppierung nach Alter. Dadurch wird die erneute Fragmentierung der soeben geschriebenen Segmente weniger wahrscheinlich, da bei dem relativ hohen Alter der noch gültigen Seiten davon ausgegangen werden kann, dass deren Seitendaten auch in Zukunft nicht geändert werden. Zum Sichern neuer Daten werden wie beim ersten Verfahren vollständige freie Segmente verwendet.

Die Qualifizierung der Segmente in „frei“, „teilweise belegt“ und „vollständig genutzt“ kann entweder auf Basis der Metainformationen im Segment oder auf Basis der im Hauptspeicher vorhandenen Seiten-Segment-Tabellen (siehe Kapitel 3.3.2) für jede Seite erfolgen. Ersteres betrachtet die Informationssektoren der zu analysierenden Segmente einzeln und benötigt aufgrund

dessen relativ wenig Hauptspeicher, jedoch erfordert die häufige Positionierung der Festplattenmechanik auf die Informationssektoren aller zu analysierender Segmente sehr viel Zeit. Letzteres kommt gänzlich ohne Zugriff auf die Festplatte aus, benötigt dafür aber einen kleinen Puffer im Hauptspeicher. Beide Möglichkeiten, die im Folgenden genauer diskutiert werden, erfordern eine Tabelle zur Ermittlung des zugehörigen Segments für jede Seite aus jedem noch gültigen Abbild. Diese Seiten-Segment-Tabellen sind jeweils knapp 4 MB groß und stellen somit für aktuelle Rechner keine Belastung dar. Die für die Reorganisation benötigten Daten können auf zwei unterschiedliche Arten gewonnen werden, worauf in den nächsten beiden Abschnitten eingegangen wird.

Qualifizierung anhand der Daten auf der Festplatte

Auf der Basis der in den Informationssektoren enthaltenen Daten lässt sich feststellen, ob das zum Informationssektor gehörende Segment frei, teilweise belegt oder vollständig genutzt ist, indem die Zahl der Referenzen auf das aktuelle Segment ermittelt werden. Dazu werden die Nummern der in diesem Segment gespeicherten Seiten ermittelt, und für jede dieser Seiten wird über die für jede Generation vorgehaltene Seiten-Segment-Tabelle geprüft, ob ein Verweis auf das aktuelle Segment existiert (siehe Abbildung 3.9). Die Zahl der noch referenzierten Seiten ergibt direkt den Status des Segments: Befinden sich die Daten zu allen Seiten in anderen Segmenten, wird das aktuelle Segment nicht mehr genutzt und kann sofort überschrieben werden. Werden noch alle Seiten referenziert, wird das Segment vollständig genutzt und muss nicht reorganisiert werden. Wird jedoch nur ein Teil der Seiten referenziert, ist das Segment fragmentiert und kann teilweise wiederverwendet (Punkt 2 in obiger Liste) oder mit anderen Segmenten reorganisiert werden (Punkt 3 in obiger Liste). Die Analyse eines Segments kann entweder bei der Suche nach einem freien Segment im laufenden Betrieb an der aktuellen Position der Festplattenmechanik erfolgen oder in speziellen Phasen zur Reorganisation mit der Analyse anderer Segmente zusammengefasst werden. Eine Analyse an der aktuellen Position hat den Vorteil, dass vorerst keinerlei Kopfbewegung erforderlich ist, jedoch muss für jedes weitere betrachtete Segment die Mechanik der Festplatte bewegt werden, wodurch die Suche nach einem freien Segment unter Umständen sehr lange dauern kann. Bei der Reorganisation von mehreren Segmenten ist mit diesem Verfahren aber eine schnelle Prüfung der noch benötigten Teile der betrachteten Segmente möglich, so dass die betrachteten Segmente schnell defragmentiert und zusammengefasst mit aktualisierten Metadaten wieder auf die Festplatte geschrieben werden können.

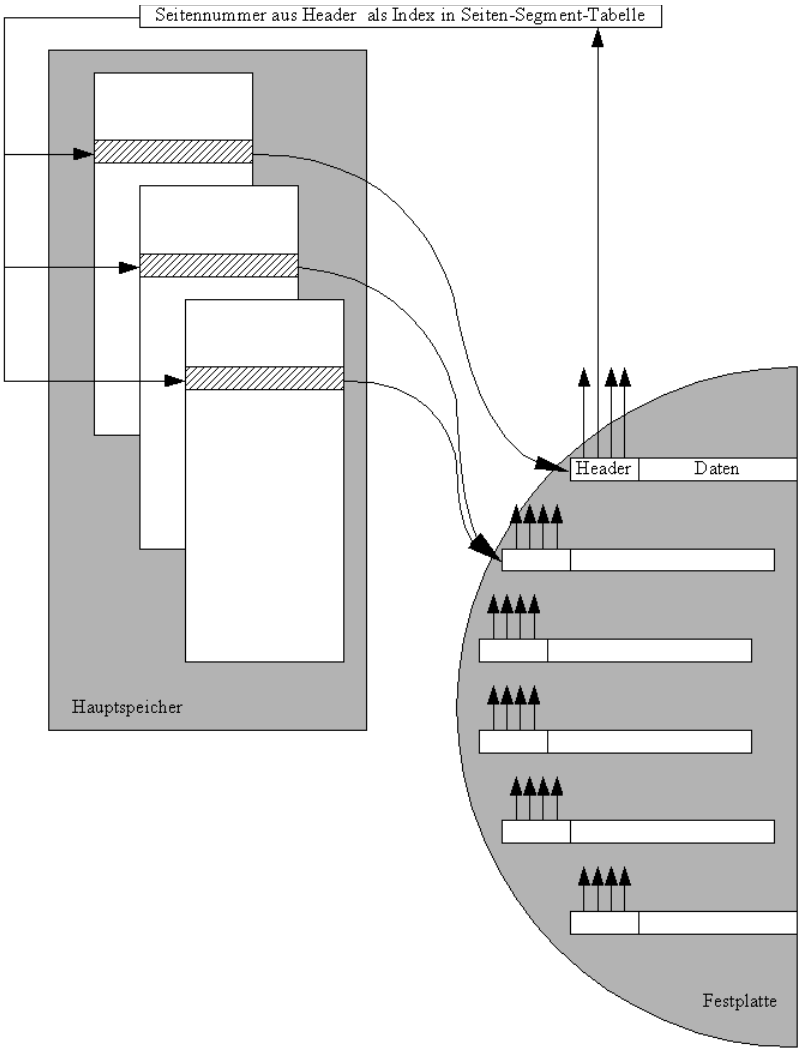

Abbildung 3.9: Qualifizierung anhand der Daten auf der Festplatte

Qualifizierung anhand der Daten im Hauptspeicher

Alternativ zur Analyse mit Hilfe der Informationssektoren lässt sich die Belegung der Segmente auch ohne Zugriff auf die Festplatte ermitteln, indem anhand einer Auswertung der Seiten-Segment-Tabellen eine Belegungsliste erstellt wird. Dabei wird für jedes auf der Festplatte vorhandene Segment entweder ein Bit (frei/belegt) oder ein ausreichend dimensionierter Referenzzähler (die maximale Zahl an Seiten muss zählbar sein, also beispielsweise 5 Bit bei 20 Seiten pro Segment) vorgesehen und mit 0 initialisiert. Für jede Seite wird in jeder Seiten-Segment-Tabelle das referenzierte Segment ermittelt und in der Belegungsliste das zu diesem Segment gehörende Bit gesetzt bzw. der zu diesem Segment gehörende Zähler inkrementiert (siehe Abbildung 3.10). Dabei ist der Aufwand zur Ermittlung der Blockzustände konstant und hängt ausschließlich von der Zahl der Generationen und der Größe der zu durchsuchenden Seiten-Segment-Tabellen ab. Nach Abschluss der Analyse enthält die Belegungsliste die notwendigen

Informationen zur Bewertung der Segmente in frei oder belegt und bei der zählenden Variante auch über den Grad der Fragmentierung. Der Hauptspeicherbedarf hängt in beiden Fällen von der Zahl der Segmente auf der Festplatte ab und liegt bei einer Festplatte mit 200 GB und 20 Seiten pro Segment bei etwa 320 KB für die Bitmap-Variante bzw. 2,5 MB für die zählende Variante (bei einem 8-Bit-Zähler). Der Hauptspeicherbedarf beider Varianten ist also im Vergleich zur heutigen Ausstattung als gering zu bezeichnen.

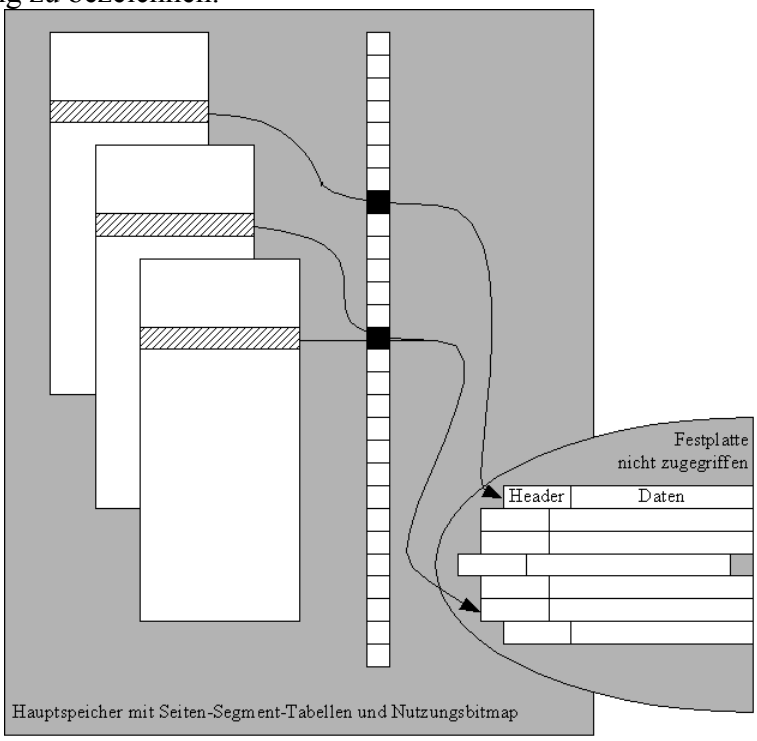

Abbildung 3.10: Qualifizierung anhand der Daten im Hauptspeicher

Da freie Blöcke über die vorgestellte Bitmap-Variante sehr schnell (siehe Messungen in Kapitel 5.4) und ohne Zugriff auf die Festplatte ermittelt werden können, bietet sich dieses Verfahren grundsätzlich als erster Schritt auf der Suche nach freien Segmenten an. Bei ausreichend großen Festplatten kann damit schnell und mit sehr hoher Wahrscheinlichkeit ein freier Block gefunden werden. Dadurch kann die aufwendige Reorganisation mit der Verschiebung von Daten im Sinne des obigen Punktes 3 entfallen oder auf Phasen niedriger Last verschoben werden.

3.3.4 Möglichkeiten zur Optimierung

Obwohl beim hier vorgestellten Verfahren bereits sehr hohe Geschwindigkeit, geringer Aufwand und sehr wenig Overhead entstehen, bieten sich bei der Speicherung von Seiten ähnliche Optimierungen an wie bei deren Übertragung durch das Protokoll (siehe Kapitel 2.2.6). Da die

Struktur des zugrunde liegenden Mediums jedoch völlig unterschiedlich ist, wird in diesem Kapitel auf die bei der Speicherung möglichen Optimierungen gesondert eingegangen.

Zusammenfassen von leeren Seiten

Wie in Kapitel 2.2.6 bereits erwähnt, sind unter Plurix viele Seiten vollständig leer. Diese Seiten dennoch mit 4096 Nullbytes auf der Festplatte zu sichern, ist vor allem in Anbetracht der bereits vorhandenen Metadaten, in denen die Eigenschaft „leer“ mit einem einzigen Bit gespeichert werden könnte, verschwenderisch. Der grundsätzlich zwar mögliche Ansatz, die Nummern von leeren Seiten in einer speziellen Tabelle abzulegen, ist nicht empfehlenswert, da dies das Log-Only-Verhalten vernichten und Kopfbewegungen zur Positionierung auf den gesonderten Bereich erfordern würde. Stattdessen empfiehlt es sich, die jeweils 24 Bytes an Metadaten für bis zu 170 leere Seiten in einer speziellen Leerliste zusammenzufassen. Eine solche maximal 4080 Bytes belegende Leerliste kann innerhalb eines Segments anstelle der Daten einer Seite gespeichert werden. Die für diesen Bereich vorgesehenen Metadaten im Informationssektor des Segments enthalten dann nur einen Hinweis, dass es sich bei den Seitendaten um eine Leerliste handelt. Die nachfolgende Abbildung verdeutlicht dieses Konzept.

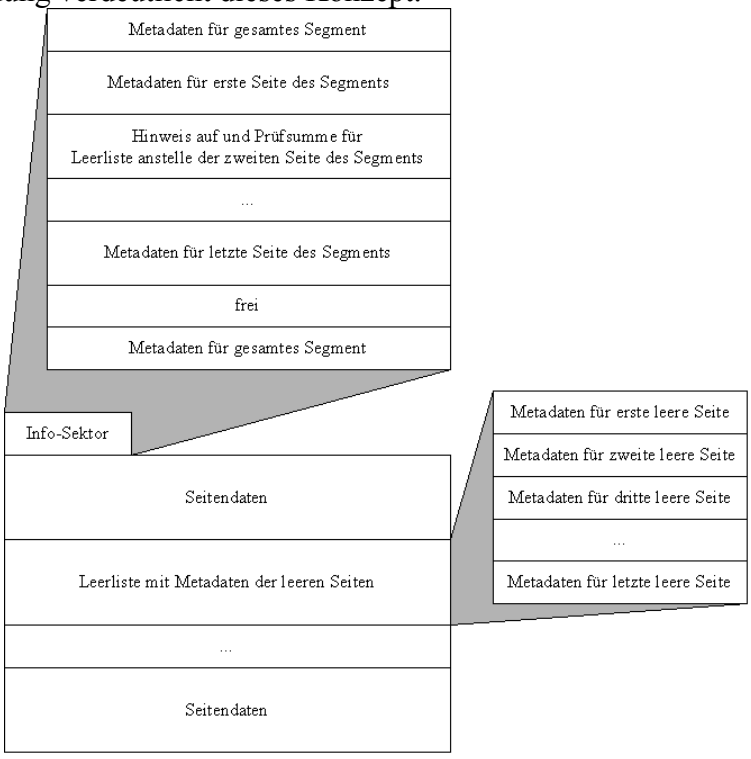

Abbildung 3.11: Segmentaufbau bei Verwendung von Leerlisten

Mit Hilfe dieser Optimierung lassen sich leere Seiten deutlich effizienter und platzsparender auf der Festplatte speichern, ohne dabei Abstriche bei den Metadaten in Kauf nehmen zu müssen. Der auf der Festplatte benötigte Speicherbedarf reduziert sich um Faktor 170 für leere Seiten von

4120 Bytes pro Seite auf 4120 Bytes für 170 Seiten, wodurch der Durchsatz beim Sichern deutlich steigt (siehe dazu auch die Messungen in Kapitel 5.4). Im Hauptspeicher werden für diese Optimierung nur 4080 Bytes zur Zwischenspeicherung der Leerliste benötigt, so dass die Unterstützung einer Leerliste insbesondere bei Systemen wie Plurix mit vielen leeren Seiten dringend empfohlen wird.

Kompression von Seitendaten

Unter Beibehaltung der Struktur auf der Festplatte können die Daten der gesicherten Seiten komprimiert und somit mehr Seitendaten gesichert werden. Der Kompressionsfaktor fällt bei unterschiedlichen Seitendaten verschieden aus, so dass nicht präzise vorhergesagt werden kann, wie viele Seiten in einem Segment mit konstanter Länge untergebracht werden können. Für die Praxis empfiehlt sich daher eine empirische Ermittlung des durchschnittlichen Kompressionsfaktors mit anschließender Anpassung der Segmentgröße, üblicherweise kann mit einer Reduktion der Datenmenge auf etwa 20-50% gerechnet werden (siehe Messungen des Protokolls in Kapitel 5.3). Die Adressierung der gespeicherten Seiten erfolgt dann nicht nur über deren Index, sondern wie in Abbildung 3.12 dargestellt über einen zusätzlich für jede Seite im Informationssektor abzulegenden Offset, dessen Speicherung die maximale Zahl an Seiten pro Segment reduziert (bei einem zusätzlichen 32 Bit Offset sind statt der bisher 20 Seiten nur noch 17 möglich). Da die Kompression unter Umständen erfolglos ist, sollte wie beim Protokoll die Möglichkeit vorgesehen werden, Seitendaten trotz eingeschalteter Kompression unkomprimiert abzulegen. Die Information darüber sollte ebenfalls bei den für jede Seite vorgesehenen Metadaten im Informationssektor (siehe Kapitel 3.3.2) abgelegt werden, beispielsweise im Offset oder in einem der noch verfügbaren Flags.

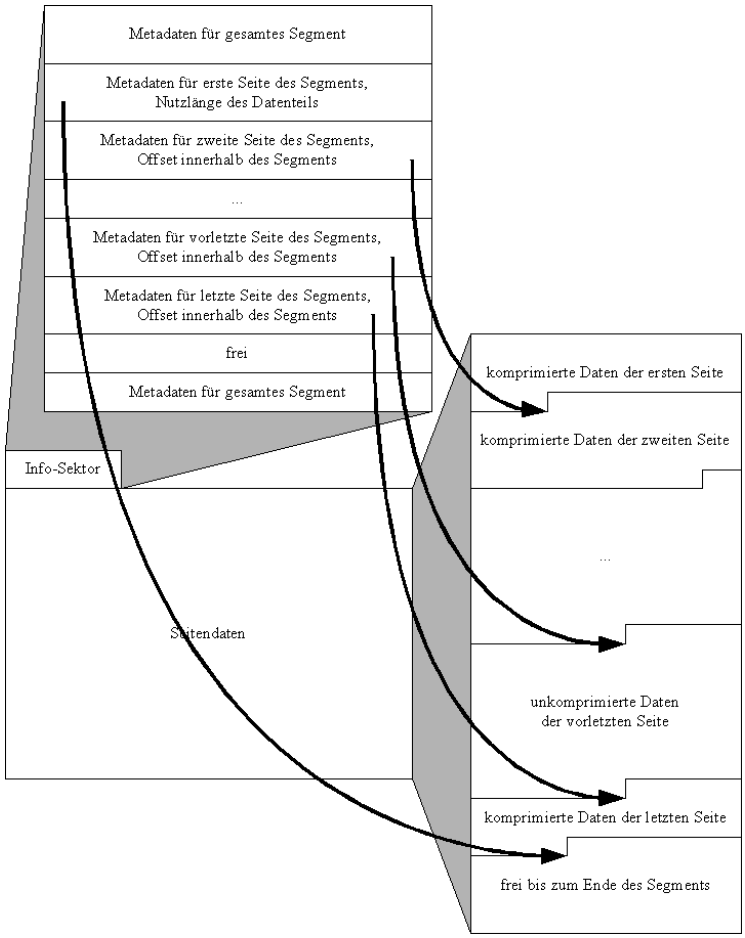

Abbildung 3.12: Segmentaufbau bei Verwendung von Kompression

Die Kompression der Seitendaten erfordert CPU-Leistung sowie ein Umkopieren von Daten zur Anordnung der komprimierten Daten in einem zu speichernden Segment, wofür zusätzlicher Hauptspeicher benötigt wird und verwaltet werden muss. Mit dieser Optimierung können zwar etwa doppelt so viele Seiten auf der Festplatte gespeichert werden, bei den Größen der aktuell verfügbaren Festplatten ist dieser Vorteil im Vergleich zum Aufwand jedoch nur in Ausnahmefällen hinreichend, so dass der Einsatz dieser Optimierung für den jeweiligen Anwendungsbereich genau geprüft werden sollte.

3.3.5 Zusammenfassung

Ausgehend von bestehenden Verfahren zur Sicherung von Seiten auf einer Festplatte wurde die Struktur „Linear Segment“ entwickelt, die einen hohen Durchsatz bei gleichzeitig niedrigem Aufwand ermöglicht. Unter Berücksichtigung der Eigenschaften aktueller Festplatten konnte ein System mit Log-Only-Verhalten entworfen werden, das Kopfbewegungen im sichernden Betrieb nur zur Reorganisation erfordert. Der Aufbau der Segmente bietet dennoch auch bei

unvorhersehbaren Störungen wie Stromausfall alle notwendigen Informationen zur Wiederherstellung eines konsistenten Gesamtsystems. Die Varianten der nach Bedarf stattfindenden Reorganisation wurden ausführlich diskutiert sowie im Hinblick auf Betrieb und Implementierung bewertet. Trotz bereits sehr hoher Effizienz bestehen Möglichkeiten zur Optimierung, die geprüft, beschrieben und deren Vor- wie Nachteile herausgearbeitet wurden.

Die Messungen belegen, dass mit Linear Segment die Aufwandsgrenze zur Sicherung aller für ein verteiltes Betriebssystem relevanten Daten nicht nur überwunden werden kann, sondern das Intervall zur Sicherung von vollständigen und konsistenten Abbildern sogar ohne übermäßige Beeinträchtigung der Leistung des Clusters äußerst klein gewählt und somit die Gefahr von Datenverlust minimiert werden kann. Darüber hinaus ist das entwickelte Verfahren kompakt implementierbar und didaktisch leicht vermittelbar, wodurch sich dieses Verfahren auch zum Einsatz in der Lehre eignet.

4 Architektur rücksetzbarer Betriebssysteme

So vielfältig und unterschiedlich die heutigen Betriebssysteme für die verschiedensten Einsatzgebiete und Prozessoren auch sind, lassen sich ihre Aufgaben dennoch abstrahieren und verallgemeinern (nach P. Schulthess in [SchuBS04]): „Das Betriebssystem macht den Anwendungssystemen die Betriebsmittel zugänglich; insbesondere Verarbeitungsleistung, Speicher und Kommunikationswege.“ Es „kontrolliert alle Hardware- und Software-Komponenten eines Rechners und teilt sie effizient den einzelnen Nachfragern zu“ und „stellt Basis-Dienstleistungen [...] zur Verfügung“ (nach F. Schweiggert et al. in [ScBoAI03]). Im Kontext dieser Arbeit geht es dabei weniger um den allgemeinen Aufbau von Betriebssystemen, sondern vielmehr um die Unterstützung von Fehlerbehandlung und Rücksetzung in einem auf Transaktionen basierenden Betriebssystem wie Plurix (siehe [GoecSB05]). In diesem Kapitel werden Konzepte und Grundlagen diskutiert, die im Fehlerfall eine effiziente und möglichst verlustarme Behandlung des Fehlers durch das Gesamtsystem erlauben, um somit die zwei Aufgabenbereiche Bequemlichkeit und Effizienz (siehe [StalBP03]) eines Betriebssystems verbessern zu können. Aus Sicht des Systems wird in diesem Kapitel jedes Fehlverhalten entsprechend der Definition nach DIN 44300 beleuchtet: „Ein Fehler ist jegliche Abweichung in Inhalt, Aufbau und Verhalten eines Testobjekts zwischen ermittelten, beobachteten, gemessenen Daten einerseits und den entsprechenden, in den Zielvorgaben spezifizierten oder theoretisch gültigen Daten andererseits.“

Beispielsweise stellt die lokale Dysfunktion einer Transaktion aufgrund eines Programmierfehlers zwar eine Beeinträchtigung des Betriebs aus Sicht des Anwenders und einen Fehler aus Sicht des Programmierers dar, jedoch wird die Systemstabilität dadurch nicht beeinträchtigt. Der normale Betrieb kann durch lokales Entfernen der fehlerhaften Transaktion ohne weitere Beeinträchtigung fortgesetzt werden und stellt keinen Fehler aus Sicht des Systems dar.

Fehlerbehandlung und Fehlervermeidung

Viele Forschungsrichtungen beschäftigen sich nicht mit der Frage, wie ein Fehler, nachdem er eingetreten ist, behandelt werden kann, sondern möchten fehlerhaftes Verhalten des Gesamtsystems von vornherein eindämmen. Dabei sind nicht Programmierfehler oder Anwendungsfehler gemeint, die mittels geeignetem Einsatz von Software-Engineering-Techniken wie Analyse, Spezifizierung und Tests (siehe unter anderem auch [MaBoTC00], [MoBrEM04], [ChJaOO04], [BundVM05]) drastisch reduziert werden können, sondern Fehler im System oder der Hardware. Solche Fehler

können zwar nicht verhindert werden, jedoch sollen deren Auswirkungen für den Anwender und die Anwendungssoftware weitestgehend abgemildert beziehungsweise unsichtbar werden. Das System übernimmt also aktiv die Bereitstellung von fehlertoleranten Diensten für die darüber liegenden Schichten. Hierfür werden komplexe und vielschichtige Dienste sowie dazu passende Kommunikationsmodelle entworfen, die dem Anwendungsprogrammierer eine komfortable Schnittstelle anbieten und dann mittels Replizierung und Voting auch im Fehlerfall einen konsistenten Betrieb anhand der noch funktionierenden Replikate aufrecht erhalten (siehe [EvoyAT81], [GrLaCT04] und [ReBaRS05]). Um diese Funktionalität bereitstellen zu können, entstehen auch im fehlerfrei laufenden Betrieb Kosten für zusätzliche Hardware zur unabhängigen Ausführung sowie für die Kommunikation zwischen den Replikaten ([FeNaES04]).

Alternativ zur Implementierung von Fehlertoleranz in einer oder mehreren Schichten zwischen Betriebssystem und Anwendungsprogrammen kann in einem transaktionalen Betriebssystem wie Plurix die Fähigkeit zur Rücksetzung dazu genutzt werden, nicht nur die Anwendungen, sondern das Gesamtsystem fehlertolerant zu machen und gleichzeitig auf die zusätzlichen Schichten zu verzichten. Mit der in Kapitel 2.1.6 diskutierten Integration von Dauerhaftigkeit auf der Netzwerkebene sowie dem in Kapitel 3.3 vorgestellten effizienten Verfahren zur Sicherung von Seiten lassen sich periodische Schnappschüsse des Gesamtsystems erstellen, für die im fehlerfreien Betrieb nur geringe Kosten entstehen (siehe Messungen in Kapitel 5.4). Die in das System integrierte Funktionalität für Rücksetzung ermöglicht eine schnelle und schlank implementierbare Fehlerbehandlung, die nur die geänderten Teile des Systems verwirft und durch das konsistente Abbild ersetzt. Besondere Beachtung erfordern dabei die üblicherweise nicht transaktionalen Gerätetreiber, da die Einstellungen der Hardware an den Stand des geladenen Abbildes angepasst werden müssen, worauf in Kapitel 4.3 ausführlich eingegangen wird. Die Kapitel 4.1 und 4.2 behandeln jedoch zunächst die verschiedenen Klassen und die Strategien zur Behandlung von Fehlern.

4.1 Klassifizierung von Fehlern

4.1.1 Autonom behebbare Fehler

Nicht jedes Fehlverhalten aus Sicht des Programmierers stellt tatsächlich ein Fehlverhalten der Hardware oder des Systems dar. Zum Beispiel erwartet der Programmierer typischerweise, dass ein abgeschicktes Paket beim Empfänger eintrifft. Jedoch ist bei Ethernet spezifiziert, dass Pakete

ohne Meldung an den Sender verworfen werden können, was unter hoher Last auch tatsächlich vermehrt auftritt. Die Übertragung kann also ohne zusätzlichen Aufwand in der Software nicht garantiert werden.

In vielen Fällen kann der Systemprogrammierer bei genauer Analyse der zugrunde liegenden Hardware solche häufig auftretenden Probleme berücksichtigen und seine Software mit nur geringem Aufwand dahingehend erweitern, dass das unerwünschte Verhalten erkannt und automatisch korrigiert wird. Im obigen Beispiel kann der vollständige Verlust eines Pakets durch erneute Übertragung behoben werden, wobei je nach Kommunikationsmodell die Erkennung und Behandlung entweder vom Sender nach fehlender Bestätigung oder vom Empfänger bei ausbleibender Information vorgenommen werden kann (siehe dazu auch Kapitel 2.2.5).

Diese Art von Fehlern kann autonom von den beteiligten Knoten behandelt werden und betrifft dann nicht den gesamten Cluster. Dies ist zwar konzeptuell vorteilhaft, jedoch muss von Fall zu Fall ermittelt werden, ob die Kosten im Verhältnis zum Nutzen eine spezielle Implementierung rechtfertigen. Die Kosten setzen sich aus Komplexität der Implementierung der Fehlerbehandlung sowie dem Aufwand im laufenden Betrieb zusammen; der Nutzen wird maßgeblich durch die erwartete Fehlerhäufigkeit und die gewonnene Leistung beeinflusst. Falls der Aufwand zu hoch ist oder der Fehler nur sehr selten auftritt, kann unter Umständen auch auf die explizite Fehlerbehandlung verzichtet werden und das Gesamtsystem mit Hilfe der periodischen Sicherung wieder in einen konsistenten Zustand gebracht werden.

4.1.2 Assistiert behebbare Fehler

Wird ein nicht autonom behebbarer Fehler vom System oder einer Anwendung erkannt, muss der Cluster auf das letzte konsistente Abbild zurückgesetzt werden. Die dafür erforderlichen Grundlagen und Konzepte wurden in dieser Arbeit bereits diskutiert, insbesondere in den Kapiteln 2.1.6, 2.2.7, 2.3.2 und 2.3.3. Die relevanten Teile des Hauptspeichers werden durch das gesicherte Abbild ersetzt und die Gerätetreiber in den zum Abbild passenden Zustand gebracht, worauf in den folgenden Kapiteln 4.2 und 4.3 ausführlich eingegangen wird. Die Ausführung der laufenden Programme kann dann sofort konsistent fortgesetzt werden (siehe auch [ScFrCR04]).

Sporadische Fehler können einfach durch erneute Ausführung behandelt werden, wohingegen dauerhafte Fehler wie ein Hardwaredefekt des Netzwerks menschliches Eingreifen erfordern. Nach Beheben der Fehlerursache oder Deaktivierung der defekten Maschine kann der Betrieb jedoch konsistent fortgesetzt werden. Dieser Ansatz ist unabhängig von der Fehlerursache und eignet sich somit für jegliche Art von Fehlern (siehe auch [DeBoGO94], [WiMuED96]).

4.1.3 Nicht behebbare Fehler

Obwohl sich diese Arbeit der Behebung von fehlerhaften Situationen widmet, bleibt dennoch die Klasse von nicht behebbaren Fehlern bestehen. Sie zerfällt weiter in die zwei Gruppen „erkennbare, aber nicht behebbare Fehler“ und „nicht erkennbare Fehler“, wobei letztere aufgrund ihrer Unkenntlichkeit generell nicht direkt behandelt werden können.

Erkennbare Fehler, die nicht mit dem im vorigen Abschnitt erläuterten Verfahren behoben werden können, entstehen bei Hardware-Problemen im Cluster, die auch durch menschlichen Eingriff nicht ohne weiteres zu lösen sind. Beispielsweise kann der Cluster die Folgen zu großer Stromschwankungen im Versorgungsnetz zwar unter Umständen in Form von häufigen Knotenausfällen erkennen, jedoch stellen diese für ihn eine unüberwindliche Hürde dar.

Da der Pageserver in der vorgestellten Version einen Single Point of Failure (siehe auch Kapitel 2.1.6) darstellt, sind dauerhafte Ausfälle des Pageservers aus Sicht des Clusters ebenfalls nicht behebbare Fehler. Dies kann einerseits ohne Änderung des Verfahrens durch ausfallsichere Hardware wie unabhängige Stromversorgung, doppelt ausgeführtes Netzteil und spiegelndes RAID-Plattensystem (siehe [ChLeRH94]) entschärft werden. Andererseits kann das Verfahren in Richtung vollständige Replizierung oder durch Pageserver-Cluster (siehe Kapitel 6.1) weiterentwickelt und somit die Problematik des single point of failure behoben werden. Ist nach einem Ausfall der Neustart des Pageservers möglich, kann der Fehler bereits ohne Erweiterung des vorgestellten Verfahrens behoben werden, da die Datenstrukturen auch nach unerwartetem Neustart des Pageservers die Rekonstruktion eines konsistenten Abbildes erlauben (siehe Kapitel 3.3.2).

4.2 Strategien zur assistierten Fehlerbehandlung

Wenn beim Erkennen eines Fehlers die Entscheidung für eine Rücksetzung des Clusters auf ein altes Abbild gefallen ist, verwirft der Pageserver die seit Erstellung dieses Abbildes geänderten Seiten und veranlasst den Cluster, ebenfalls alle betroffenen Seiten zurückzusetzen und die Einstellungen der Gerätetreiber (siehe Kapitel 4.3) zu überprüfen. Ein vollständiger Reset der Maschine funktioniert bei allen behebbaren Fehlern, da hierbei auch die Hardware automatisch vollständig zurückgesetzt wird. Die Zeiten für ein vollständiges Booten eines Knotens sind jedoch schon aufgrund der teilweise sehr langen Startzeiten des BIOS sehr hoch, so dass es besser ist, je nach Schwere des Fehlers sowie je nach Abhängigkeiten zwischen Kern und Transaktionen nur bestimmte Teile der Maschine zurückzusetzen. Die folgende Auflistung bietet einen nach Aufwand sortierten Überblick, der nachfolgend diskutiert wird.

1. Zurücksetzen der seit dem letzten Abbild geänderten Speicherbereiche: Sind die Nummern aller geänderten Seiten protokolliert und keine für das System kritische Seite geändert worden, kann das System durch ein Zurücksetzen ausschließlich der geänderten Seiten normal weiterlaufen.
2. Zurücksetzen aller für das System unkritischen Speicherbereiche: Bei seit dem letzten Abbild unveränderter Laufzeitumgebung beziehungsweise unverändertem Kern kann das System durch ein Zurücksetzen aller außerhalb des Kerns befindlichen Seiten normal weiterlaufen.
3. Zurücksetzen des Systems und der Laufzeitumgebung: Wurde der Kern oder die Laufzeitumgebung seit Erstellung des letzten Abbildes geändert, so muss der Kern entfernt und durch die alte Version ersetzt sowie vollständig neu gestartet werden, bevor die außerhalb des Kerns befindlichen Seiten angefordert und die Transaktionen neu gestartet werden können.
4. Neustart des Knotens: Bei Verklemmung oder Defekt der Hardware kann eine Maschine erst nach vollständigem Neustart wieder dem Cluster beitreten; sie erhält dann alle relevanten Daten des letzten Abbildes und kann den Betrieb wieder aufnehmen.

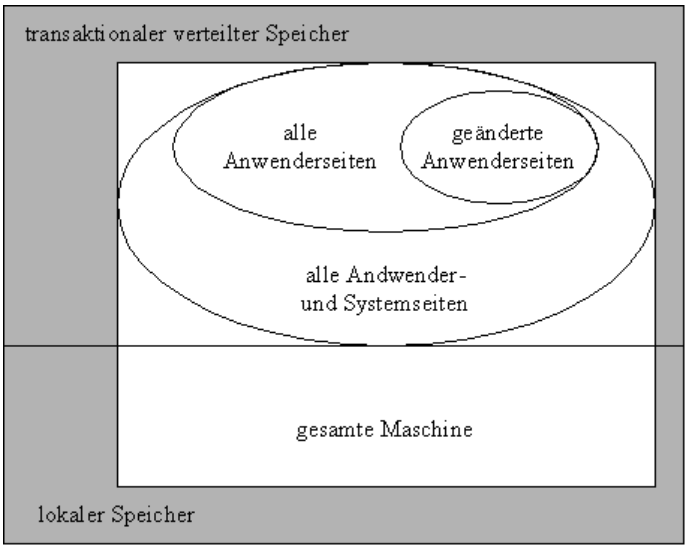

Abbildung 4.1: Position der verschiedenen Seiten

Die erste Variante zur Zurücksetzung ausschließlich der geänderten Seiten ist zwar die schnellste, jedoch ist dazu entweder die Ermittlung beziehungsweise Protokollierung der Nummern der seit dem letzten Abbild geänderten Seiten erforderlich, was aufwendig beziehungsweise ansonsten überflüssig ist. Die zweite Variante mit der Zurücksetzung aller Seiten außerhalb des Kerns und der Laufzeitumgebung ist dagegen einfach durchführbar, da hier jede am Cluster beteiligte Maschine alle Nicht-System-Seiten, die bereits für das System besonders markiert sein müssen (siehe [GoecSB05]), mit dem für die transaktionale Konsistenz bereitgestellten Mechanismus invalidiert. Der implementierte Prototyp zeigt, dass die Geschwindigkeit der zweiten Variante ausreichend ist und somit der Aufwand zur Implementierung der ersten Variante

vermieden werden kann. Noch einfacher und schneller wäre diese Variante allerdings implementierbar, wenn der Kern nicht im verteilten Speicher läge, da dann die Bereiche für Seiten des Kerns und Seiten für die verteilte Laufzeitumgebung völlig getrennt wären und somit nicht jede einzelne Seite auf diese Eigenschaft hin überprüft werden müsste, wie dies in der momentanen Implementierung von Plurix (siehe ebenfalls [GoecSB05]) der Fall ist. Damit würde auch die dritte Variante in obiger Liste überflüssig werden, da der dann sehr kleine Kern unveränderbar bleiben könnte, siehe folgende Graphik:

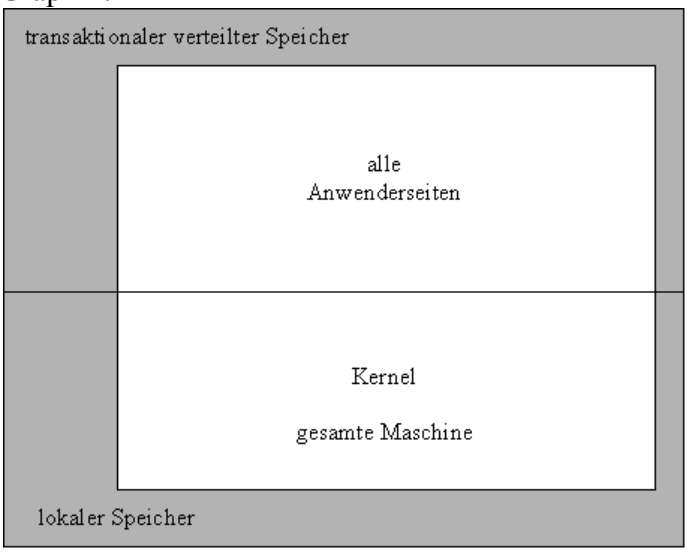

Abbildung 4.2: Vereinfachte Position der Seiten

In der aktuellen Implementierung kann der Kern jedoch auch zur Laufzeit ausgetauscht und invalidiert werden, so dass der neu installierte Kern bei einer Rücksetzung verworfen und die alte Version des Kerns vom Pageserver angefordert werden muss. Die Implementierung des Prototyps sieht dazu vor, den in [GoecSB05] beschriebenen kleinen Kern („Joiner“) zum Anfordern des verteilten Kerns im Speicher zu halten und wie beim ersten Start der Maschine mit seiner Hilfe den verteilten Kern zu laden und zu starten. Die langsamste, jedoch aufgrund der vollständigen Initialisierung jeglicher Hardware sicherste Rücksetzung ist die vierte Variante. Hierbei werden über einen per Software ausgelösten Reset alle Komponenten inklusive Prozessor, Hauptspeicher und Bussysteme sowie alle daran angeschlossenen Peripheriegeräte wie beim Einschalten der Maschine zurückgesetzt, wodurch auch Verklemmungen der Hardware behoben werden können. In allen Fällen muss auf die Einstellung von Geräten besondere Rücksicht genommen werden, was im folgenden Kapitel diskutiert wird.

4.3 Behandlung von Geräten bei einer Rücksetzung

4.3.1 Definition von Gerätezuständen

Die möglichen Zustände und die Komplexität eines Gerätes hängen sehr stark von der Art des Gerätes (Netzwerk- oder Graphikkarte, serielle oder parallele Schnittstelle, Controller für Festplatten) und der tatsächlichen Ausführung des Gerätes (Intel Gigabit Ethernet oder 3Com Fast Ethernet, nVidia GeForce oder ATI Radeon) ab. Zwar muss jeder Treiber speziell für ein bestimmtes Gerät implementiert werden, jedoch lassen sich konzeptuelle Gemeinsamkeiten erkennen: Üblicherweise werden Geräte in einem bestimmten Modus initialisiert, um dann im laufenden Betrieb die ausgewählte Funktionalität bereitzustellen. Die Initialisierung besteht bei einer Netzwerkkarte im Setzen der Übertragungsgeschwindigkeit und weiterer Parameter wie MAC-Adresse und Duplex-Art, bei einer Graphikkarte können verschiedene Auflösungen und Wiederholraten gewählt werden, und bei einer Festplatte am IDE-Kanal muss die Geschwindigkeit der Übertragung festgelegt werden. Im laufenden Betrieb werden zwar die Zustände mancher Geräte ebenfalls verändert, jedoch sind diese Änderungen im Normalfall wiederholbar und haben keinen Einfluss auf neue Befehle. Beispielsweise wird das Setzen eines Pixels unabhängig von weiteren Pixeln sein, bei einer seriellen Schnittstelle ändert das Absetzen eines Bytes nicht die Einstellungen der Übertragung. Dementsprechend lassen sich zwei Gruppen von Zuständen definieren: kritische (grundlegende) Einstellungen, die untereinander Abhängigkeiten enthalten können und für die Art und Weise der Funktion eines Gerätes entscheidend sind, und unkritische (betriebliche) Einstellungen, die zwar von den kritischen Einstellungen abhängig, untereinander jedoch unabhängig sind.

Diese beiden Kategorien von Einstellungen der Gerätetreiber erlauben eine Unterstützung zuverlässiger Rücksetzung von Knoten, indem die kritischen Einstellungen durch das System rekonstruiert werden, so dass die Anwendung die verwendeten Geräte nach einer Rücksetzung ohne spezielle Behandlung sofort in den von ihr eingestellten Modi weiterverwenden kann. Dennoch kann es für manche Anwendungen nach einer Rücksetzung wichtig sein, spezielle unkritische Daten an das Gerät zu schicken (wie zum Beispiel eine erneute Initialisierung einer Verbindung auf höherer Ebene oder auch das vollständige Zeichnen des Bildschirms), so dass das Betriebssystem die Anwendungen hierüber informieren sollte. Die folgenden Kapitel widmen sich daher den Voraussetzungen und Schnittstellen für dieses Konzept.

4.3.2 Anforderungen an das Betriebssystem und die Gerätetreiber

Um eine fehlerfreie und schnelle Rücksetzung des Gesamtsystems zu ermöglichen, muss der Zustand des Betriebssystems aus dem gesicherten Speicher rekonstruierbar und wie in der aktuellen Version von Plurix (siehe [GoecSB05]) transaktional sein. Die kritischen Geräteeinstellungen werden durch das transaktionale System erst beim erfolgreichen Abschluss von Transaktionen durchgeführt (siehe ebenfalls [GoecSB05]), so dass während dieser Änderungen ein im gesicherten verteilten Speicher befindliches Objekt an diese Einstellungen angepasst werden kann. Dieses Objekt enthält dann bei einer Rücksetzung den Zustand des Gerätes zum Zeitpunkt der Konsistenzierung des Abbildes und ermöglicht damit eine vollständige Initialisierung des Gerätes mit den zu diesem Zeitpunkt herrschenden Einstellungen. Da kritische Geräteeinstellungen typischerweise nur sehr selten geändert werden und der Zugriff auf das jeweilige Konfigurationsobjekt sehr schnell und im Vergleich zu den übrigen Aufwendungen beim Transaktionsabschluss irrelevant ist, behindert diese zusätzliche Speicherung der Einstellungen den laufenden Betrieb nicht. Die folgende Abbildung 4.3 verdeutlicht den Zusammenhang der einzelnen Komponenten im Kontext eines virtuellen verteilten Speichers mit transaktionaler Konsistenz.

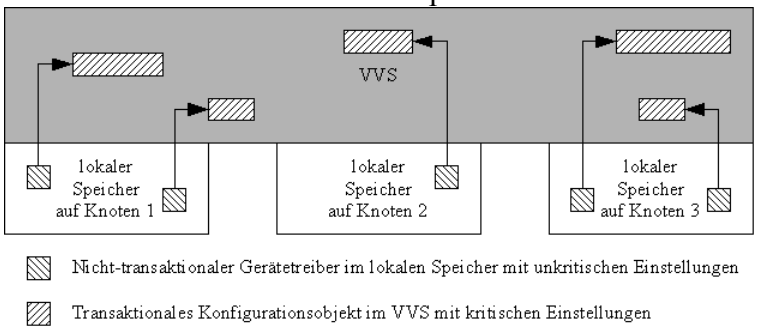

Abbildung 4.3: Positionierung der Konfigurationsobjekte

Üblicherweise kann nach einem vollständigen Neustart einer Maschine nicht auf die alten Instanzen und somit nicht direkt auf deren Einstellungen zugegriffen werden. Daher bietet der implementierte Prototyp für jeden Knoten eine durch das System verwaltete Liste im gesicherten verteilten Speicher an, in der die Zustände für alle Geräte abgelegt sind. Nach einem Neustart prüft der Kern, ob für den aktuellen Knoten eine solche Liste vorhanden ist und reicht sie bei Vorhandensein an die installierten Gerätetreiber weiter, so dass jeder Treiber die für sein Gerät passenden Einstellungen aus dem zu ihm passenden Konfigurationsobjekt auslesen kann. Ist keine solche Liste vorhanden, so wurden bisher für die aktuelle Maschine noch keine Objekte registriert. Die Geräte nehmen dann ihre Grundeinstellung vor und registrieren diese beim Kern, um bei späteren Änderungen von Einstellungen direkt auf ihr Konfigurationsobjekt zugreifen zu können.

4.3.3 Schnittstellen für Gerätetreiber und Anwendungen

Zur Verwaltung der bei einer Rücksetzung benötigten Informationen wird, wie im vorigen Kapitel erläutert, von jedem Treiber ein Konfigurationsobjekt angelegt. Im Prototyp ist dieses Objekt vom Typ DeviceFallbackInfo, welches eine interne Verkettung unterstützt. Jeder Gerätetreiber nutzt eine spezielle DeviceFallbackInfo-Instanz, welche alle für das Gerät relevanten Einstellungen reflektiert. Der Kern liefert den Gerätetreibern bei ihrer Initialisierung den Einstieg in die Liste mit, so dass jeder Treiber in der Liste nach den ihn betreffenden Informationen suchen und das Gerät dementsprechend initialisieren kann. Wird, beispielsweise beim ersten Start einer Maschine oder nach Einbau neuer Hardware, kein passendes Objekt gefunden, liefert die Initialisierungsphase des Treibers ein neues Konfigurationsobjekt zurück, das durch den Kern in die zum aktuellen Knoten gehörende Liste eingetragen wird und somit bei der nächsten Rücksetzung zur Verfügung steht. Die folgende Abbildung 4.4 verdeutlicht den für die Rücksetzung relevanten Aufbau der Gerätetreiber und Konfigurationsobjekte.

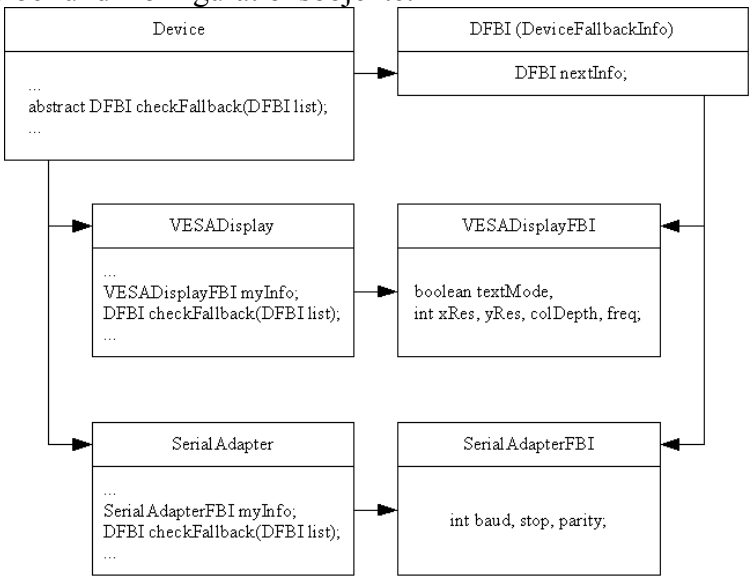

Abbildung 4.4: Treiber und Konfigurationsobjekte

Für den Desktop ist es nach einer Rücksetzung unerlässlich, die intern verwendeten Referenzen auf Ausgabetreiber zu aktualisieren sowie den Bildschirm auf den zum Abbild passenden Zustand zu setzen. Um dies zu ermöglichen, wird die Methode initAfterFallback des Desktop mit dem ab sofort zu verwendenden Ausgabetreiber aufgerufen. Nach erfolgter Initialisierung aller Geräte und Aktualisierung des Desktop wird der für jede Maschine vorhandene Zähler für Rücksetzungen inkrementiert, so dass interessierte Transaktionen durch eine Überprüfung dieses Wertes erfolgte Rücksetzungen feststellen können. Der vollständige Datenfluss während einer Rücksetzung wird im folgenden Schaubild 4.5 gezeigt.

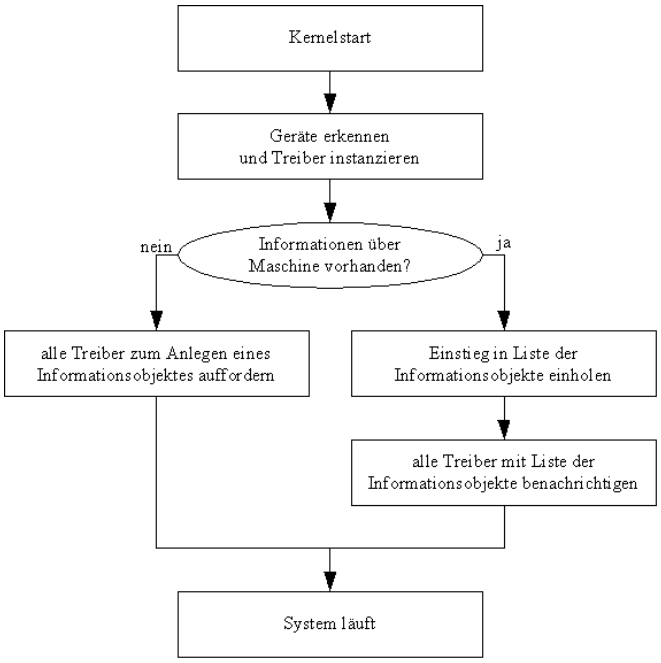

Abbildung 4.5: Start einer Maschine

Alternativ ist zusätzlich auch eine explizite Benachrichtigung aller laufenden Transaktionen denkbar, was dem Nachrichtenmodell moderner Systeme eher entspräche. Die Abfrage des Zählers durch die Transaktion ist jedoch sehr schnell möglich, und die im aktuellen System vorhandenen Transaktionen sind typischerweise nicht am Vorgang der Rücksetzung interessiert, so dass eine explizite Benachrichtigung jeder Transaktion mehr Overhead als das Prüfen einer Variablen durch interessierte Transaktionen bedeuten würde.

4.4 Zusammenfassung

Im Zusammenspiel mit der in Kapitel 2 beschriebenen transaktionalen Konsistenz und dem in Kapitel 3.3 entwickelten Verfahren zur Sicherung eines verteilten transaktionalen Speichers ergibt sich mit den in diesem Kapitel beschriebenen Algorithmen ein umfassendes Konzept zur Behandlung von Fehlern. Dazu wurden zuerst die aus Sicht eines verteilten Betriebssystems relevanten Fehlerklassen definiert und eingehend beleuchtet, wobei die besondere Bedeutung der beiden Fehlerklassen „autonom behebbare Fehler“ und „assistiert behebbare Fehler“ herausgearbeitet wurde. Erstere kann sehr effizient und ohne Abhängigkeit im Cluster behandelt werden, während Letztere eine allgemeine und auch bei Hardwaredefekten zuverlässige Behandlung von Fehlern erfordert.

Die Definition zweier unterschiedlicher Kategorien von Zuständen in Geräten ermöglicht eine sinnvolle Aufteilung in einerseits seltene und kritische Einstellungen (Konfigurationszustände)

sowie andererseits oftmalige und unkritische Einstellungen (Betriebszustände). Anhand dieser Unterscheidung wurden schlanke Schnittstellen für Betriebssystem, Gerätetreiber und Anwendungsprogramme definiert, mit deren Hilfe sich bei transaktionaler Konsistenz eine schnelle und effiziente Rücksetzung implementieren lässt. Für eine Rücksetzung von Gerätetreibern sind dann nur die folgenden drei Funktionalitäten erforderlich: Erstens muss der Treiber im laufenden Betrieb die Möglichkeit haben, seine Einstellungen in einem transaktionalen Konfigurationsobjekt abzulegen; zweitens muss der Treiber über eine Rücksetzung benachrichtigt werden, und drittens muss ihm danach Zugriff auf sein früher erstelltes Konfigurationsobjekt zur Initialisierung des Gerätes ermöglicht werden.

Wie der implementierte Prototyp zeigt, kann die Aufwandsgrenze zur Erstellung von Schnappschüssen auch dann überwunden werden, wenn die Schnappschüsse die Gerätezustände berücksichtigen. Im Gegensatz zu herkömmlichen Wiederanlaufverfahren können somit unter Beachtung der hier vorgestellten Konzepte nicht nur die verteilten Anwendungen, sondern auch das Betriebssystem und die installierten Geräte mit Hilfe von Schnappschüssen zuverlässig zurückgesetzt und dadurch der Clusterbetrieb auch im Fehlerfall aufrecht erhalten werden. Der Prototyp zeigt darüber hinaus, dass die intrinsische Komplexität zur Rücksetzung eines Clusters niedrig ist und unter Verwendung der in dieser Arbeit vorgestellten Konzepte elegant implementiert werden kann.

5 Messungen

5.1 Eingesetzte Hardware

Der für die Messungen eingesetzte Cluster besteht aus Rechnern mit jeweils einer CPU vom Typ Athlon XP 2500+ mit der spezifizierten Taktrate von 1.8 GHz. Jeder Rechner verfügt über ein Asus A7V8X-X Mainboard mit 512 MB DDR-RAM und eine 3Com905 Netzwerkkarte. Diese ermöglichen mittels Twisted Pair Verkabelung an einem Allied Telesyn MR912TX Hub den Betrieb eines 100 MBit/s Fast Ethernet Netzwerkes. Der Pageserver ist darüber hinaus mit einer Hitachi Deskstar HDS722516VLAT80 Festplatte ausgestattet, die über 164 GB Speicherkapazität sowie 8 MB Cache verfügt und an einem UDMA5-Kanal (nominell 100 Megabyte pro Sekunde, siehe [T13ATA02]) angeschlossen ist.

In den folgenden Kapiteln werden zunächst die beiden Komponenten Festplatte und Netzwerk einzeln und zum Abschluss im realen Betrieb gemessen und in Bezug zur theoretisch maximalen Leistung gesetzt. Weitere themenverwandte Messungen, die bereits veröffentlicht wurden, finden sich unter anderem in [ScFrFT04], [ScFrCR04], [FrLoPC05] und [FrScTC05].

5.2 Festplatte

5.2.1 Sequentieller Zugriff

Zur Messung der mit dem implementierten Treiber und der Architektur des Betriebssystems erreichbaren Geschwindigkeit wurden große Datenblöcke zu jeweils 2 Megabyte gelesen bzw. geschrieben. Bei der eingesetzten Hardware können maximal 32 Sektoren entsprechend 16 Kilobyte in einem Befehl beauftragt werden, so dass für einen Block bereits mehrere Befehle an die Festplatte geschickt werden müssen.

	Durchsatz		
	1. Lauf	2. Lauf	3. Lauf
lesender Zugriff	43246 KB/s	82700 KB/s	82694 KB/s
schreibender Zugriff	34741 KB/s	34907 KB/s	35205 KB/s

Tabelle 5.1: Sequentieller Zugriff

Deutlich sichtbar ist beim lesenden Zugriff der Einsatz des Caches, der für den gesamten Block Platz bietet. Somit kann der angeforderte Block in den folgenden Läufen 2 und 3 direkt aus

dem Cache und ohne Zugriff auf das Speichermedium geliefert werden. Beim schreibenden Zugriff wird der Cache nicht genutzt, sondern direkt auf das Speichermedium geschrieben. Dies hat zwar den Nachteil, dass nicht mit der maximal möglichen Übertragungsrate gearbeitet wird, jedoch sind die von der Festplatte als geschrieben bestätigten Daten tatsächlich auf dem Speichermedium. Die unterschiedliche Übertragungsrate zwischen lesendem und schreibendem Zugriff im ersten Lauf resultiert aus der Tatsache, dass die Festplatte bei einem lesenden Zugriff nicht nur die angefragten Sektoren, sondern noch eine kleine (von der Firmware abhängige) Anzahl weiterer Sektoren in den Cache liest. Im Gegensatz zu einem schreibenden Zugriff, bei dem immer alle Daten aus dem Hauptspeicher zur Festplatte und dort auf das Speichermedium übertragen werden müssen, kann durch das vorauseilende Lesen eine Anfrage auf sequentiell nachfolgende Sektoren somit teilweise ohne Zugriff auf das Speichermedium bedient werden.

Die effektive Datenrate errechnet sich aus dem Quotienten der angeforderten Datenmenge und der dafür benötigten Zeit, welche sich aus der Zeit zur Übertragung des Befehls, der Übertragung der Daten aus dem Cache in den Hauptspeicher und der dafür erforderlichen Steuerzyklen zusammensetzt. Da die für die Steuerzyklen erforderliche Zeit von mehreren Komponenten (IDE-Kanal, PCI-Bus, Anbindung des Hauptspeichers etc.) abhängt und sich leider nicht messen lässt, muss eine Abschätzung des erreichten Potenzials anhand der Messungen genügen.

$$p_{eff} = \frac{d_{Nutzdaten}}{t_{gesamt}} = \frac{d_{Nutzdaten}}{t_{Befehl} + t_{Daten}} = \frac{d_{Nutzdaten}}{t_{Befehlsworte} + t_{Befehlssteuerung} + t_{Nutzdatenworte} + t_{Nutzdatensteuerung}}$$

Die gemessene Zeit zur Übertragung eines Befehls zum Lesen von 16 Kilobyte Daten liegt inklusive der im System benötigten Aufrufe im Mittel bei etwa 15 Mikrosekunden. Zur Übertragung von 80,7 MB werden demnach 5164 Befehle entsprechend etwa 77 Millisekunden pro Sekunde benötigt.

$$p_{eff} = \frac{d_{Nutzdaten}}{t_{Befehl} + t_{Daten}} = \frac{80{,}7\,\mathrm{MB}}{1\,\mathrm{s}} = \frac{80{,}7\,\mathrm{MB}}{0{,}077\,\mathrm{s} + 0{,}923\,\mathrm{s}}$$

$$p_{nom} = \frac{d_{Nutzdaten}}{t_{Nutzdatenworte}} = 100\,\frac{\mathrm{MB}}{\mathrm{s}}$$

Da während der Übertragung des Befehls keine Nutzdaten übertragen werden, werden die 81 MB Nutzdaten innerhalb von 0,923 Sekunden übertragen. Von der nominellen Bandbreite von 100 MB/s können also nur 92% genutzt werden. Die erreichten 80,7 MB entsprechen also etwa 88% der theoretisch möglichen Datenrate und werden maßgeblich von der Geschwindigkeit des Betriebssystems, des Treibers, der Festplatte und der zwischen den Komponenten erforderlichen Kommunikation beeinflusst.

5.2.2 Wahlfreier Zugriff

Zur Ermittlung der Kosten für eine Kopfbewegung wurden mehrere zehntausend zufällige, über die gesamte Festplatte verteilte Sektoren gelesen und die benötigte Zeit gemessen, wenn ein einzelner Sektor oder wenn acht Sektoren auf einmal gelesen wurden.

	1 Sektor			***8 Sektoren***		
	1. Lauf	2. Lauf	3. Lauf	1. Lauf	2. Lauf	3. Lauf
Durchsatz	43 KB/s	43 KB/s	43 KB/s	342 KB/s	344 KB/s	344 KB/s
benötigte Zeit pro Block	11,618 ms	11,573 ms	11,611 ms	11,672 ms	11,616 ms	11,624 ms

Tabelle 5.2: Wahlfreier Zugriff

An den benötigten Zeiten pro Block lässt sich direkt ablesen, dass die Zeit zur Übertragung der Daten deutlich kleiner ist als diejenige für eine Kopfbewegung. Genauer können diese Werte aus dem Durchschnitt der einzelnen Läufe berechnet werden, wenn die für die Übertragung von Sektoren benötigte Zeit als proportional zur Zahl der Sektoren angenommen wird:

$$t_{1,lesen} = t_{positionieren} + t_{1,übertragen} \quad \text{und} \quad t_{8,lesen} = t_{positionieren} + t_{8,übertragen} = t_{positionieren} + 8 * t_{1,übertragen}$$

$$t_{1,übertragen} = \frac{t_{8,lesen} - t_{1,lesen}}{7} = \frac{11{,}637\,\text{ms} - 11{,}601\,\text{ms}}{7} \approx 0{,}005\,\text{ms}$$

Die benötigte Zeit zur Übertragung eines Sektors liegt mit 5 Mikrosekunden sehr nahe an der für UDMA5 spezifizierten minimalen Zeit von 4,9 Mikrosekunden. Der resultierende Durchsatz von etwa 94 MB/s stimmt mit der im vorigen Kapitel ermittelten maximal nutzbaren Bandbreite von 92 MB/s im Rahmen der Messgenauigkeit überein.

$$t_{positionieren} = \frac{8 * t_{1,lesen} - t_{8,lesen}}{7} = \frac{92{,}805\,\text{ms} - 11{,}637\,\text{ms}}{7} \approx 11{,}6\,\text{ms}$$

Die errechnete durchschnittliche Zeit zur Positionierung der Festplattenmechanik liegt mit 11,6 Millisekunden zwischen den vom Hersteller spezifizierten Zeiten (siehe [HitaDD05]) für eine „übliche Positionierung“ (8,8 ms) und für eine „Positionierung mit maximaler Distanz“ (15,1 ms).

Die Messungen bestätigen somit die in dieser Arbeit vertretene Aussage, dass Kopfbewegungen beim Zugriff auf Festplatten nach Möglichkeit vermieden werden sollten. Bei Verfahren, die intensiven Gebrauch von wahlfreiem Zugriff machen müssen, empfiehlt sich daher der Einsatz von alternativen Speichermedien wie CompactFlash-Medien oder RAMdisks mit niedriger Auswahlzeit. Sie bringen jedoch Vor- und Nachteile in Bezug auf Durchsatz und Ausfallsicherheit mit sich, deren Beschreibung den Rahmen dieser Arbeit sprengen würde.

5.3 Protokoll

5.3.1 Durchsatz

Um Aussagen über die Effizienz des implementierten Protokolls sowie der darunter liegenden Schichten (Netzwerktreiber und Betriebssystem) machen zu können, wurde im Vorfeld zu den Messungen des Protokolls der maximal erreichbare Durchsatz gemessen. Die folgende Tabelle 5.3 zeigt den gemessenen Durchsatz beim Versenden von Paketen durch eine einzelne Maschine in Abhängigkeit von der Menge an Nutzdaten. Das tatsächlich verschickte Paket ist jeweils um 34 Bytes vergrößert, um Platz für die benötigten Ethernet- und IP-Header zu bieten. Die genutzte Bandbreite bezieht sich auf das theoretische Maximum von 12800 KB/s bei 100 MBit/s auf einem Segment.

Nutzdaten	*Paketgröße*	*Paket-Durchsatz*	*Byte-Durchsatz*	*genutzte Bandbreite*
1466 Bytes	1500 Bytes	8172 Pakete/s	11970 KB/s	93,51%
966 Bytes	1000 Bytes	12155 Pakete/s	11870 KB/s	92,73%
734 Bytes	768 Bytes	15714 Pakete/s	11785 KB/s	92,07%
542 Bytes	576 Bytes	20734 Pakete/s	11663 KB/s	91,12%
350 Bytes	384 Bytes	30467 Pakete/s	11425 KB/s	89,26%
238 Bytes	272 Bytes	41958 Pakete/s	11145 KB/s	89,07%
126 Bytes	160 Bytes	40263 Pakete/s	6291 KB/s	49,15%
62 Bytes	96 Bytes	37472 Pakete/s	3513 KB/s	27,45%
30 Bytes	64 Bytes	38096 Pakete/s	2381 KB/s	18,60%

Tabelle 5.3: Durchsatz und genutzte Bandbreite

Deutlich wird die Beschränkung auf etwa 11-12 Megabytes pro Sekunde und etwa 40.000 Pakete pro Sekunde bei einem einzelnen Sender, entsprechend einer Verarbeitungszeit von etwa 25 Mikrosekunden pro versandtem Paket. Bei vielen Sendern können durch Verwendung eines Switches mehrere Segmente gleichzeitig Daten übertragen, so dass die aggregierte Bandbreite deutlich höher sein kann als die hier erzielten Werte beim Einsatz einer einzelnen sendenden Maschine.

Bei vielen kleinen Paketen ist durch die niedrige Auslastung des Netzwerkes sichergestellt, dass auch bei Verwendung eines einzelnen Segments mehrere Sender ihr Paket auf dem Netzwerk platzieren können und gleichzeitig die Übertragung von großen Paketen nicht allzu sehr behindert wird. Daher ist trotz der bei einem einzelnen Sender niedrigen Auslastung des Netzwerkes die Verwendung von nur 64 Bytes großen Paketen zum Beispiel zur Übertragung des Tokens sinnvoll.

5.3.2 Latenz und Kompression

In einem seitenbasierten verteilten Speicher mit Transaktionen sind die Latenz bei Seitenanfragen sowie die Latenz zum Transaktionsabschluss von größter Bedeutung. Bei einer Implementierung mit „First Wins“-Strategie lässt sich die Latenz des Transaktionsabschlusses aus Sicht des Netzwerkes auf die Latenz des Tokens reduzieren (siehe Kapitel 2.2.5).

Sowohl die Anfrage auf das Token als auch die Bestätigung zum Erhalt des Tokens wird in der aktuellen Implementierung als 64 Byte großes Paket versandt. Da allein der Versand von zwei Paketen dieser Größe etwa 50 Mikrosekunden beansprucht, wird eine Latenz von mindestens 50 Mikrosekunden erwartet.

Lauf 1	***Lauf 2***	***Lauf 3***	***Lauf 4***	***Lauf 5***
70 µs	81 µs	66 µs	66 µs	66 µs

Tabelle 5.4: Tokenlatenz

Die gemessene minimale Zeit von 66 Mikrosekunden als Token-Latenz liegt erfreulich nahe bei den 50 Mikrosekunden zum Versand von zwei Paketen. Die Differenz von 16 Mikrosekunden erklärt sich aus der Summe von Paketlaufzeit und der im Softwaresystem benötigten Bearbeitungszeit. Die Bearbeitungszeit teilt sich weiter auf in die von Netzwerkkarte, Interrupts, Netzwerktreiber und Protokoll benötigte Zeit. Die einzelnen Bestandteile wurden nicht weiter gemessen, da die äußeren Umstände bereits ähnlich großen Einfluss haben, wie Lauf 1 und 2 zeigen, bei denen noch externer Verkehr auf dem Netzwerk vorhanden war.

Etwas vielschichtiger als die Latenz des Tokens, das angefragt und geliefert wird, ist die Latenz von Seiten, da hier der antwortende Knoten entscheiden muss, ob die Seite leer oder nicht leer ist. Bei eingeschalteter Kompression für nicht leere Seiten muss er zusätzlich ermitteln, ob die Übertragung komprimiert oder unkomprimiert erfolgen soll.

Unkomprimierte Seiten werden grundsätzlich in drei Paketen verschickt (siehe Kapitel 2.2.5), bei eingeschalteter Kompression wird die komprimierte Übertragung nur dann genutzt, wenn sie mindestens ein Paket einspart (siehe Kapitel 2.2.6). Die nachfolgende Tabelle 5.5 bezieht sich auf die Anzahl der übertragenen Seiten, die daraus resultierende Anzahl an Paketen sowie die dazugehörige durchschnittliche Latenz einmal mit aktivierter und einmal mit deaktivierter Kompression für die folgenden drei Szenarien:

1. Heapcheck: die Übertragung aller Seiten, die zur Konsistenzprüfung aller Zeiger erforderlich ist,
2. Apfel: (1) und zusätzlich diejenigen Seiten, die die Bilddaten eines Apfelmännchens enthalten,
3. RayT: (2) und zusätzlich diejenigen Seiten, die die Bilddaten einer Raytracer-Szene enthalten.

Szenario &		*leer*		*1 Paket*		*2 Pakete*		*3 Pakete*		*nicht-leer*	
Kompression		*Anz.*	*Latenz*	*Anz.*	*Latenz*	*Anz.*	*Latenz*	*Anz.*	*Latenz*	*Σ Pakete*	*Ø Latenz*
Heap	aus	26	60 µs	-		-		649	546 µs	1947	546 µs
	an			462	174 µs	187	416 µs	-		836	244 µs
Apfel	aus	795	68 µs	-		-		2212	526 µs	6636	526 µs
	an			2021	162 µs	191	416 µs	-		2403	184 µs
RayT	aus	795	68 µs	-		-		3765	529 µs	11295	529 µs
	an			3127	163 µs	632	437 µs	6	709 µs	4409	210 µs

Tabelle 5.5: Seitenlatenz mit und ohne Kompression

Die Anzahl der leeren Seiten ist unabhängig von der Aktivierung der Kompression, die Latenz liegt aufgrund der gleich großen Pakete auf ähnlichem Niveau wie die Anforderung eines Tokens. In der Tabelle wird daher bei der Berechnung die Anzahl der Pakete und bei der durchschnittlichen Latenz nur die Übertragung von nicht leeren Seiten berücksichtigt. Da die ersten beiden Szenarien Heap und Apfel im Szenario RayT enthalten sind, wird in den folgenden Graphiken und in der nachfolgenden Diskussion dieses Szenario betrachtet.

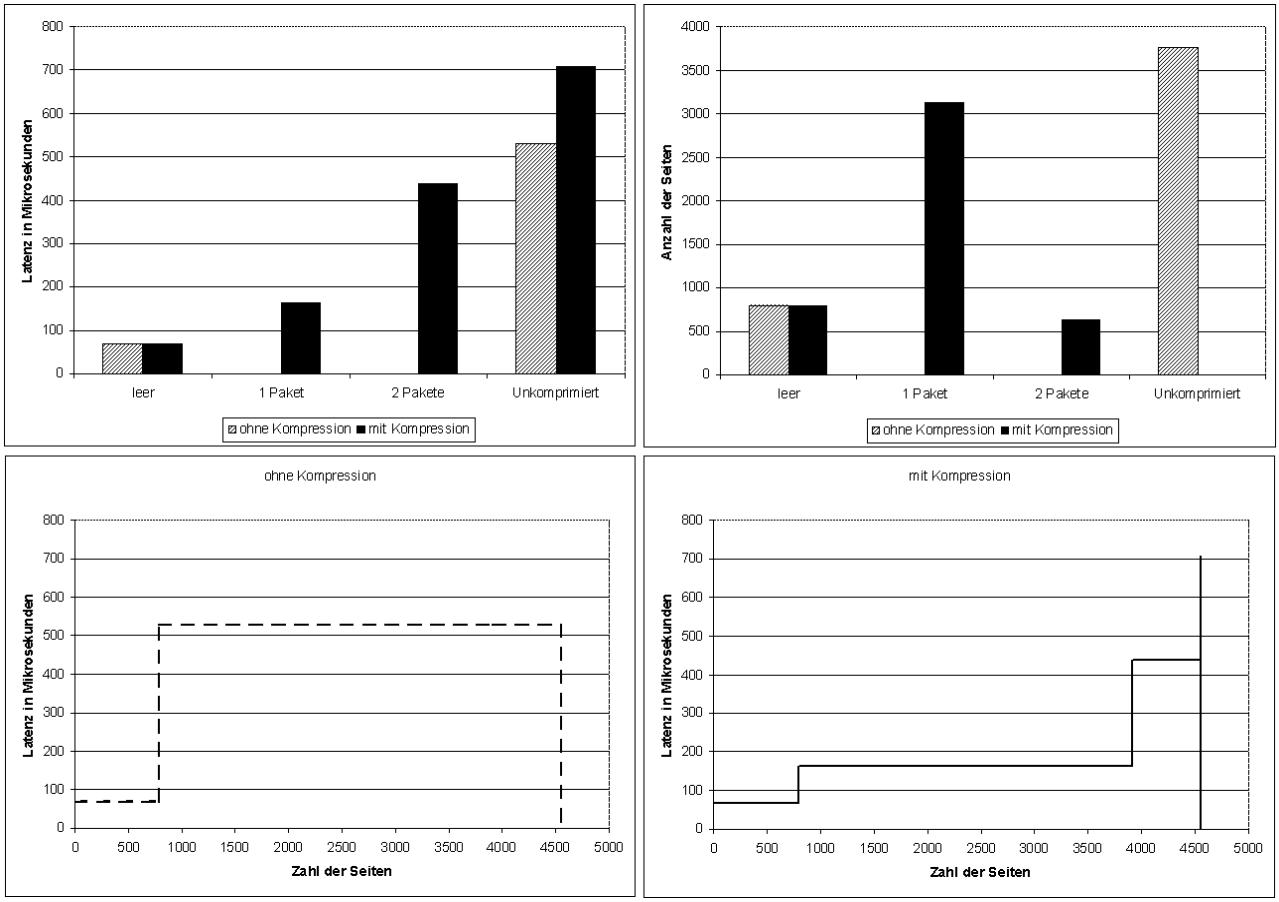

Abbildung 5.1: Latenz von Seiten ohne und mit Kompression

Beim Einsatz der Kompression wird die Latenz von unkomprimiert übertragenen Seiten zwar deutlich von etwa 529 auf 709 Mikrosekunden erhöht, jedoch ist die Kompression nur bei weniger als 0,2% der übertragenen Seiten erfolglos. Bei der deutlichen Mehrheit von gut 99,8% wird durch die Kompression mindestens ein Paket auf dem Netzwerk eingespart und die durchschnittliche Latenz deutlich verringert. Bei über 80% der Seiten wird die durchschnittliche Latenz von 529 auf 163 Mikrosekunden reduziert, insgesamt sinkt die durchschnittliche Latenz auf 210 Mikrosekunden. Gleichzeitig können mit eingeschalteter Kompression über 60% der ohne Kompression erforderlichen Pakete vermieden werden, darüber hinaus sind die Pakete teilweise deutlich kleiner als diejenigen der unkomprimierten Übertragung, so dass weitere Bandbreite auf dem Netzwerk gespart wird.

Die in Bezug auf die Auslastung des Netzwerkes und auf die Latenz beim anfragenden Knoten vorteilhafte Kompression erfordert jedoch auf dem angefragten Knoten Rechenleistung, um die angefragte Seite zu komprimieren. Wie Messungen ergeben haben, steht beim eingesetzten Kompressionsverfahren (siehe Kapitel 2.2.6) die benötigte Rechenleistung in direktem Zusammenhang mit der resultierenden Datenmenge. Die Linien im folgenden Diagramm 5.2 links zeigen die Zeit zum Versenden eines Paketes in Abhängigkeit von der resultierenden Datenmenge; zum Vergleich ist in Balkenform die benötigte Zeit beim Versand ohne Kompression abgebildet. Im Diagramm 5.2 rechts ist angegeben, welche Kompressionsraten bei den im gesamten System existenten Seiten nach Berechnung des Apfelmännchens und des Raytracer-Bildes erreicht werden können. Nicht enthalten sind die gänzlich leeren Seiten sowie die sechs nicht erfolgreich komprimierbaren Seiten. Im Gegensatz zu obiger Auflistung sind ansonsten aber nicht nur die übertragenen Seiten, sondern alle Seiten im gesamten System berücksichtigt. Die gestrichelte Linie zeigt waagrecht die Verteilung in Prozent sowie senkrecht die Grenze zwischen einer potenziellen Übertragung in einem oder in zwei Paketen.

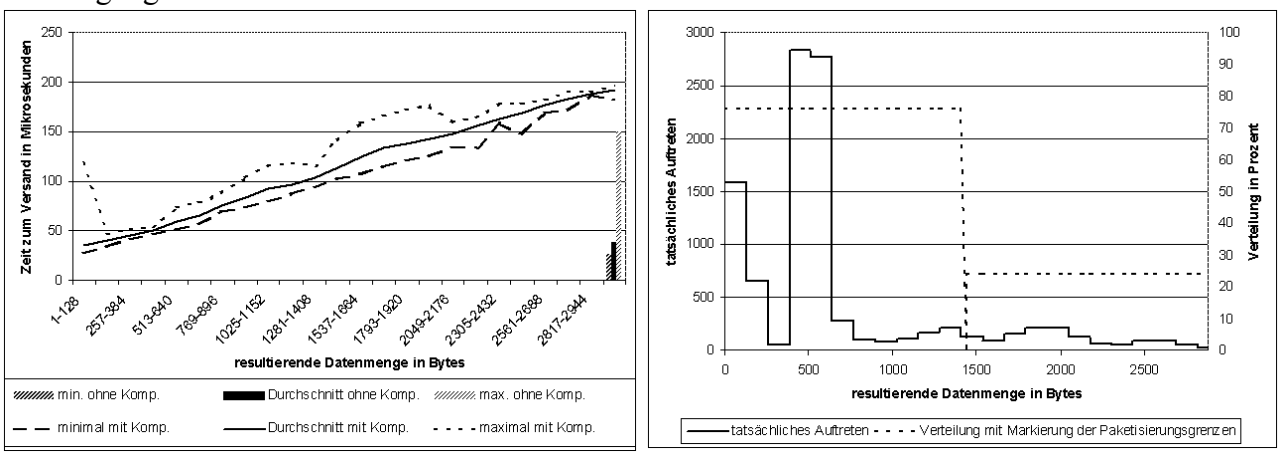

Abbildung 5.2: Aufwand beim Sender ohne und mit Kompression

Für die Mehrzahl der im System vorhandenen Seiten dauert das Versenden mit Kompression nur unwesentlich länger als ohne Kompression, so dass der Einsatz von Kompression auch aus Sicht des Senders vertretbar ist.

5.4 Cluster-Betrieb

5.4.1 Aufwand im fehlerfreien Betrieb

Der Aufwand im fehlerfreien Betrieb bezeichnet im Folgenden die Verzögerung eines Anwendungsprogramms und gliedert sich in die Belastung der im Cluster arbeitenden Maschinen durch Seitenanfragen und in die Verzögerung durch Inanspruchnahme des Tokens während der Konsistenzierung. Das Token ist bei der Konsistenzierung notwendig, um alle noch ausstehenden Seiten aus dem Cluster anzufragen und dabei zu verhindern, dass weitere Seiten modifiziert werden. Eine alternative Implementierung könnte mit einer speziellen Verwaltung von alten Seiten die Blockierung des Tokens vermeiden, jedoch erfordert dies tiefgreifende Änderungen in der Speicherverwaltung. Wird jedoch die Konsistenzierung durchgeführt, wenn nur noch wenige Seiten ausstehen, ist der Unterschied sehr klein. Daher wurde der Aufwand zur Umstrukturierung des als Basis dienenden Systems nicht betrieben.

Dazu kann der implementierte Prototyp optional bereits im laufenden Betrieb ohne Blockierung des Tokens geänderte Seiten einsammeln, die zwar potenziell bis zur Konsistenzierung nochmals verändert werden, andernfalls jedoch bei der Konsistenzierung bereits gesichert sind und somit nicht mehr behandelt werden müssen. Im Idealfall, also bei keiner ausstehenden Seite, benötigt der Pageserver 25 Millisekunden zur Konsistenzierung. Gegenüber den Messungen in Kapitel 5.3.2 konnte die durchschnittliche Latenz um 0,4 Millisekunden verringert werden, indem mehrere Seiten gleichzeitig angefragt und somit die Leerlaufzeiten reduziert wurden. Somit werden bei der Konsistenzierung für jede noch ausstehende Seite etwa 0,17 Millisekunden zusätzlich zu den minimalen 25 Millisekunden benötigt.

Zur Messung der im fehlerfreien Betrieb erzeugten Verzögerung durch den Einsatz eines Pageservers wurde die Ausführungszeit eines Raytracers gemessen. Im nachfolgenden Schaubild 5.3 wird in Abhängigkeit von der Zeit zwischen zwei Konsistenzierungen gezeigt, wie lange die Berechnung einer Szene dauert, wenn der Pageserver für die Erzeugung des konsistenten Abbildes die ausstehenden Seiten (a) parallel und (b) punktuell erst bei der Konsistenzierung einsammelt. Zum Vergleich ist die Zeit (c) eingetragen, die der Raytracer ohne Pageserver benötigt. Der linke

Teil des Schaubildes enthält eine lineare Zeitachse mit Nulldurchgang am Koordinatenursprung, im rechten Teil des Schaubildes ist der relevante Teil mit verschobenem Nulldurchgang vergrößert. Die nachfolgende Tabelle 5.6 enthält die für das Schaubild relevanten Daten, die aus dem Durchschnitt von jeweils drei Läufen bestehen, sowie die daraus ableitbaren Kenngrößen.

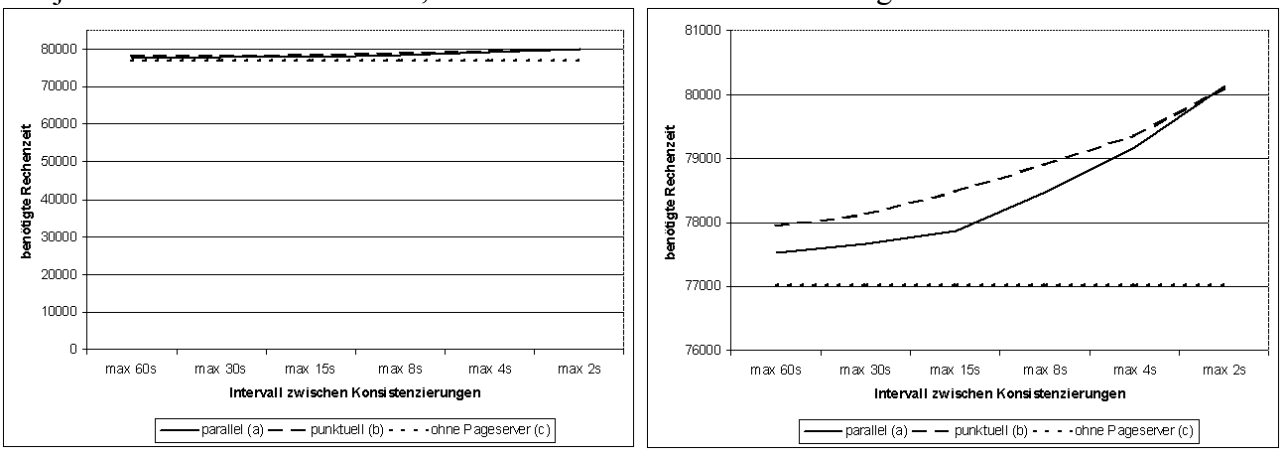

Abbildung 5.3: Rechenzeit des Raytracers je nach Konsistenzierungsintervall

	max. 60s	***max. 30s***	***max. 15s***	***max. 8s***	***max. 4s***	***max. 2s***
parallel (a)	77527 ms	77678 ms	77867 ms	78465 ms	79158 ms	80132 ms
punktuell (b)	77945 ms	78131 ms	78489 ms	78910 ms	79362 ms	80106 ms
ohne Pageserver (c)	77023 ms					
Verzögerung parallel	0,65%	0,85%	1,10%	1,87%	2,77%	4,04%
Verzögerung punktuell	1,20%	1,44%	1,90%	2,45%	3,04%	4,00%
Zahl der Konsistenzierungen	1	2	5 bis 6	9 bis 10	19 bis 20	38 bis 39
Zeit pro Sicherung parallel	504 ms	328 ms	~169 ms	~160 ms	~112 ms	~82 ms
Zeit pro Sicherung punktuell	922 ms	554 ms	~293 ms	~210 ms	~123 ms	~81 ms

Tabelle 5.6: Rechenzeit des Raytracers je nach Konsistenzierungsintervall

Die Fertigstellung der Berechnung ist direkt abhängig von der Größe des Intervalls zwischen zwei Konsistenzierungen und somit von der Häufigkeit der Konsistenzierung. Bis zu einem Intervall von vier Sekunden ist die parallele Sicherung gegenüber der punktuellen Sicherung deutlich im Vorteil, da zwischen zwei Konsistenzierungen viele Seiten bereits vor der Konsistenzierung nicht mehr geändert werden, so dass solche bereits eingesammelten Seiten während der blockierenden Phase nicht mehr angefordert werden müssen. Bei kleineren Intervallen ist es sinnvoll, die wenigen Daten nicht während der Berechnung, sondern gebündelt bei der Konsistenzierung anzufordern, um die mehrfache Sicherung von oft veränderten Seiten zu vermeiden.

Insgesamt ist eine Verzögerung von weniger als 3% bei paralleler Sicherung mit einem Intervall von nur vier Sekunden äußerst niedrig, insbesondere im Vergleich zu anderen Systemen (zum Beispiel beträgt beim IBM LoadLeveler das minimale Intervall 15 Minuten, siehe [KaRoWM01]).

5.4.2 Aufwand im Fehlerfall

Rücksetzung des Systems

Bei einem Fehler im Cluster wird dieser zurückgesetzt (siehe Kapitel 2.2.4, 2.2.7 und 2.3.3). Beim implementierten Prototypen lässt sich in Anlehnung an Kapitel 4.2 konfigurieren, ob das gesamte Betriebssystem mit allen Daten der Benutzer oder nur die Daten der Benutzer zurückgesetzt werden sollen. Da in der aktuellen Implementierung nicht sichergestellt ist, dass der Kern und die Treiber seit der letzten Konsistenzierung nicht verändert wurden, muss die Überprüfung dieses Umstandes bis zu einer Reorganisation des Systems durch den Benutzer vorgenommen werden. Die folgende Tabelle 5.7 zeigt die bei dem jeweiligen Verfahren benötigte Zeit, um einen Cluster aus ein, zwei, drei oder vier Rechnern zurückzusetzen.

	mit Neustart des Kerns		***ohne Neustart des Kerns***	
	minimal	***maximal***	***minimal***	***maximal***
1 Maschine	460 ms		244 ms	
2 Maschinen	587 ms	595 ms	246 ms	247 ms
3 Maschinen	572 ms	768 ms	249 ms	251 ms
4 Maschinen	589 ms	994 ms	252 ms	256 ms

Tabelle 5.7: Zeit zur Rücksetzung eines Clusters

Die große Differenz zwischen minimaler und maximaler Zeit bei einer Zurücksetzung mit Neustart des Kerns liegt an der in der aktuellen Implementierung erzwungenen Serialisierung beim gleichzeitigen Start von mehreren Maschinen. Im Gegensatz dazu kann die Rücksetzung der Daten des Benutzers parallel ablaufen und deutlich besser skalieren.

Anfragen auf Seiten

Bei der Rücksetzung des Clusters werden viele Seiten beim Pageserver angefordert, die bei ausreichend großem Hauptspeicher in der aktuellen Version automatisch im Speicher belassen werden, um weitere Anfragen auf die gleiche Seite schneller bedienen zu können. Die folgende Tabelle 5.8 zeigt die Latenz von Seiten nach Anfragen aus dem Cluster (bei der Messung existierten keine Seiten, die nicht erfolgreich komprimiert werden konnten).

	leer		*1 Paket*		*2 Pakete*		*nicht-leer*	
	Anz.	***Latenz***	***Anz.***	***Latenz***	***Anz.***	***Latenz***	***Σ Pakete***	***Ø Latenz***
Erster Rechner	795	64 µs	1604	3279 µs	88	2919 µs	1780	3260 µs
Zweiter Rechner	792	65 µs	1604	185 µs	73	402 µs	1750	194 µs

Tabelle 5.8: Seitenlatenz nach einer Rücksetzung

Die aktuelle Implementierung des Pageservers erfordert bei Anfragen auf leere Seiten keinen Zugriff auf die Festplatte, so dass die Latenz ähnlich wie im fehlerfreien Fall ist. Der erste Zugriff auf eine Seite erfordert jedoch den Zugriff auf die Festplatte, so dass die Latenz von durchschnittlich 210 Mikrosekunden auf 3.260 Mikrosekunden steigt. Da viele Seiten in ähnlicher Reihenfolge angefordert werden, wie sie geändert und somit gesichert wurden, kann die Festplatte hier mit dem integrierten Cache einige Positionierungen einsparen, so dass die Latenz dennoch deutlich unter der mittleren Zeit zur Positionierung des Lesekopfes (siehe Kapitel 5.2) liegt. Bei nachfolgenden Anfragen ist wiederum kein Zugriff auf die Festplatte erforderlich, da der Pageserver über ausreichend Hauptspeicher verfügt, um die 1.780 Seiten entsprechend knapp sieben Megabyte zu puffern.

6 Ausblick

Die im Rahmen dieser Dissertation entwickelten Konzepte und Verfahren sind in sich abgeschlossen und durch den implementierten Prototyp belegt. Darüber hinaus ermöglichen sie jedoch noch eine Fülle an weiterführenden Entwicklungen, deren Vielfältigkeit und Potenzial in diesem Kapitel aufgezeigt und mit grundlegenden Lösungsansätzen verknüpft werden, wobei eine erschöpfende Behandlung weiteren Arbeiten vorbehalten bleibt. Solche Arbeiten sind bereits in der Vorbereitung und werden durch die Deutsche Forschungsgemeinschaft (DFG) in Form von Mitarbeiterstellen gefördert (zum Beispiel das Projekt DTDOS: „Dependable Transactional Distributed Operating System“, eingereicht durch P. Schulthess und M. Schöttner; im Antrag wird bereits auf die Ergebnisse der vorliegenden Arbeit Bezug genommen).

6.1 Pageserver-Cluster

Der prototypisch implementierte Pageserver arbeitet auf einer einzelnen Maschine, die vorgestellten Verfahren und Konzepte lassen sich jedoch auf einen verteilten Pageserver erweitern. In weiteren Arbeiten ist zu prüfen, wie die gewonnenen Erkenntnisse auf einen verteilten Pageserver übertragen und welche darüber hinausgehenden Algorithmen entwickelt werden können.

Zum einen ist die Verteilung erstrebenswert, um den bereits sehr hohen Durchsatz des Pageservers weiter zu erhöhen, beispielsweise im Hinblick auf andere Netzwerke wie Gigabit-Ethernet, Quadrics oder Myrinet. Die hier deutlich höhere Übertragungskapazität würde den internen Durchsatz eines einzelnen Pageservers mit derzeit erhältlichen Festplatten überfordern und verlangt nach einer Aufteilung des Datenvolumens auf mehrere Pageserver. Die Aufteilung der Daten kann dabei beispielsweise nach Adressen vorgenommen werden, die veränderten Seiten könnten abwechselnd angefordert werden oder alternativ könnte auch eine Partitionierung des Clusters mit einer festen oder variablen Zuordnung zwischen Knoten und Pageservern erfolgen. Hier sind bei geeigneter Implementierung auch Kombinationen vorstellbar, so dass sich ein weites Feld für Forschungen und Optimierungen ergibt.

Zum anderen kann eine Verteilung genutzt werden, um den Pageserver redundant auszulegen und die Daten somit gegen Hardware-Fehler zu schützen. Ein paralleler Betrieb mehrerer gleichberechtigter Pageserver ist dabei ebenso denkbar wie ein Master-Slave-Szenario, bei dem ein als Master deklarierter Pageserver die bisherige Funktion des einzelnen Pageservers wahrnimmt und

die als Slave konfigurierten Knoten eine redundante Speicherung der Daten vornehmen, jedoch nicht auf Anfragen aus dem Cluster antworten. Sollte der Master-Pageserver ausfallen, kann ein vorhandener Slave-Pageserver die Aufgaben des Pageservers nahtlos übernehmen, da eine identische Datenbasis besteht und die Quelle der Sicherungskopie für die Knoten im Cluster irrelevant ist.

Beim Einsatz mehrerer Rechner, die zusammen den Dienst des Pageservers bereitstellen sollen, lassen sich durch die Kombination von paralleler Datenaufzeichnung und redundanter Hardware gezielt Verfahren entwickeln, die sowohl höheren Durchsatz als auch verbesserte Ausfallsicherheit bieten. Dabei kann unter Umständen das Konzept des transaktional verteilten Speichers für die Kommunikation der Pageserver untereinander Verwendung finden. Die Erfahrung bei der Programmierung des transaktional verteilten Speichers im Cluster hat gezeigt, dass insbesondere die Übersichtlichkeit des Datenaustauschs dem Entwickler hilft, sich auf die intrinsische Komplexität zu konzentrieren und somit unnötige Fehler zu vermeiden. Eine Verwendung dieses Konzepts bei der Weiterentwicklung eines verteilten Pageservers scheint daher nicht nur möglich, sondern sogar sehr empfehlenswert.

6.2 Rückfall-Optimierungen

6.2.1 Abhängigkeitsgraphen

Das einstellbare Intervall zwischen der Erstellung zweier konsistenter Abbilder kann auf einen so kleinen Zeitraum reduziert werden, dass der Datenverlust von wenigen Sekunden bei assistiert behebbaren Fehlern im üblichen Betrieb toleriert werden kann. Werden mehrere Generationen von Seiten-Segment-Tabellen (siehe Kapitel 3.3.2) vorgehalten, kann der Cluster auch sehr schnell auf ein älteres Abbild zurückgesetzt werden. Da die Zahl der Seiten-Segment-Tabellen jedoch beschränkt ist, müssen veraltete Tabellen verworfen werden, um neuen Generationen Platz zu bieten. Anstatt nun wie beim implementierten Prototypen immer die älteste Generation zu verwerfen, beispielsweise bei fünf Generationen und zwei Sekunden zwischen den Abbildern eine nur zehn Sekunden alte Generation, können die Tabellen auch selektiv verworfen werden. Die Auswahl erfolgt dabei so, dass Tabellen mit sehr unterschiedlichem Alter verbleiben, beispielsweise für die Zeitpunkte vor ungefähr einer Minute, einer Stunde, einem Tag und einer Woche. Der dann nicht mehr benötigte Platz auf der Festplatte kann mit den in Kapitel 3.3.3 vorgestellten Konzepten ermittelt und für neue Daten verwendet werden. Sollte nun wider Erwarten eine Generation benötigt

werden, zu der keine Seiten-Segment-Tabelle mehr existiert, könnte mit Hilfe der in den Segmenten gespeicherten Metainformationen ein Abhängigkeitsgraph erstellt werden. Der Pageserver prüft anhand dieses Graphen für die noch vorhandenen Daten, ob aus ihnen ein konsistentes Abbild näher am gewünschten Zeitpunkt konstruiert werden kann.

Der Aufwand einer solchen Analyse ist erheblich und wohl nur in seltenen Fällen gerechtfertigt. Ist eine Rücksetzung auf solche Abbilder mit nicht im Hauptspeicher haltbaren Seiten-Segment-Tabellen notwendig, wäre es unter Umständen sinnvoll, die Seiten-Segment-Tabellen der für einen längeren Zeitraum interessanten Abbilder auf einer zweiten Festplatte zu speichern. Alternativ kann auf einer zweiten Festplatte regelmäßig und mit niedriger Priorität eine Kopie der relevanten konsistenten Abbilder erstellt werden, so dass bei einer Rücksetzung alle Daten von dort gelesen werden können. Inwieweit sich weitere Verfahren mit mehr als einer Festplatte für die Optimierung dieser Problemstellung eignen, bleibt im Rahmen einer eigenen Arbeit zu prüfen.

6.2.2 Vorausschauendes Lesen nach einer Rücksetzung

Da die nach einer Rücksetzung aktiven Transaktionen mit hoher Wahrscheinlichkeit auf ähnlich alte Daten zugreifen werden, kann der Pageserver zur Beschleunigung des späteren Zugriffs zusätzlich zu einer angefragten Seite auch bereits ähnlich alte oder im gleichen Segment gespeicherte Seiten lesen und im Hauptspeicher einlagern. Nachfolgende Anfragen auf diese Seiten können dann ohne Verzögerung der Festplatte beantwortet werden, wodurch die Latenz von Seiten erheblich reduziert wird. Zu erwarten wäre, dass die durchschnittliche Latenz dadurch auch bei ersten Anfragen in den Bereich der wiederholten Anfragen kommt, sich also um etwa Faktor neun (siehe Kapitel 5.3.2) verringert.

Ziel weiterer Arbeiten könnte die Erstellung eines effektiven und allgemein verwendbaren Verfahrens zur Ermittlung der potenziell bald benötigten Seiten darstellen. Im Gegensatz zu den bisher bereits eingesetzten Caches in Datenbanken und Dateisystemen kann mit neuen Verfahren unter Umständen eine bessere Vorhersage getroffen werden, da die Transaktionen vor der Rücksetzung bereits bearbeitet wurden und somit eine Art Blick in die Zukunft existiert. Gleichzeitig können Kennzahlen anhand der Zugriffe auf eingelagerte Seiten erstellt werden, um eine sinnvolle Bewertung der eingelagerten Seiten vornehmen zu können und somit eine verbesserte Auslagerungsstrategie zu erreichen. Dies ermöglicht das Verwerfen von wahrscheinlich nicht mehr benötigten Seiten und schafft somit Platz für voraussichtlich benötigte Seiten. Genauere Informationen über das laufende System ermöglichen eine wirkungsvolle Optimierung.

Beispielsweise können in einem Cluster für Software-Entwickler die Seiten des Compilers und der Entwicklungsumgebung bereits vor der ersten Anfrage geladen werden.

6.2.3 Lokale Anwendungsdaten

Die im Umfeld dieser Arbeit vorgestellte Systematik und Architektur für rücksetzbare Systeme impliziert eine Verteilung der Anwendungsdaten sowie der für eine Rücksetzung von Gerätetreibern erforderlichen Daten. Dadurch kann die Komplexität für die Erstellung von Schnappschüssen und für eine Rücksetzung auf bereits erstellte Schnappschüsse erheblich reduziert werden, wie in Kapitel 4 ausführlich erläutert wurde.

Dennoch sind Anwendungen denkbar, deren Daten nicht zwingend transaktional behandelt werden müssen und infolgedessen auch ausschließlich lokal gehalten werden könnten, um das Risiko von Kollisionen zu vermindern und den Cluster nicht unnötig zu belasten. Die Anwendung sollte diese Daten von Zeit zu Zeit publizieren, um eine konsistente Rücksetzung zu ermöglichen. Unter Umständen entsteht durch die vergrößerte Zeitspanne zwischen den Veröffentlichungen jedoch bei einer Rücksetzung ein erhöhter Datenverlust, so dass einer Anwendung mit lokalen Daten bei einer Rücksetzung ohne Verlust des lokalen Hauptspeichers der Zugriff auf die noch vorhandenen Daten ermöglicht werden sollte. Dadurch könnte die Menge der verlorenen Informationen mit Hilfe von explizit in der Anwendung programmierten Verfahren eventuell reduziert werden.

Da die Konsistenz dieser Daten nicht gewährleistet werden kann und somit deren Prüfung dem Anwendungsprogrammierer obliegt, ist dies nur mit größter Vorsicht zu nutzen. Weitere Arbeiten könnten prüfen, inwieweit sichere Schnittstellen zwischen den verschiedenen Ebenen des Zugriffs erarbeitet werden können, die die Konsistenz des Systems garantieren und welche Anwendungen von diesem Ansatz überhaupt sinnvollen Gebrauch machen können.

6.3 Pageserver-Fehlerbehandlung

Die vorgestellten Verfahren und Konzepte gehen auch auf Seiten des Pageservers von einem Fail-Stop-Verhalten aus. Im Falle einer zerstörten Festplatte oder fälschlicherweise überschriebenen Daten ist eine Fehlerbehandlung ohne Einführung von Redundanz äußerst schwierig. Dennoch lassen sich in solchen Fällen unter Umständen aus den mit Prüfsummen versehenen und somit einzeln validierbaren Datenbeständen noch konsistente Abbilder erstellen.

In noch vergleichsweise trivialen Fällen mit ausschließlich unvollständig geschriebenen Daten, verursacht zum Beispiel durch einen nicht überbrückbaren Stromausfall, kann durch Analyse der in den Segmenten vorhandenen Metainformationen festgestellt werden, ob alle zum letzten konsistenten Abbild gehörenden Segmente vorhanden und gültig sind, so dass die Arbeit des Clusters an diesem logischen Zeitpunkt wieder aufgenommen werden kann. Bei teilweise zerstörten Segmenten hingegen ist zu prüfen, ob entweder ein früheres Abbild konsistent wiederhergestellt werden kann oder anhand der Analyse des Abhängigkeitsgraphen (siehe Kapitel 6.2.1) eine neue, bisher nicht explizit als konsistent markierte Zusammenstellung von Seiten in validen Segmenten erstellt und der Cluster somit auf diesen Punkt zurückgesetzt werden kann.

Wird der einzeln oder verteilt arbeitende primäre Pageserver seinerseits durch einen sekundären Pageserver abgesichert, lassen sich unter Umständen auch Fehler des primären Pageservers durch eine Rücksetzung behandeln. Da sich die Hardware des Pageservers sehr effektiv gegen Hardware-Fehler schützen lässt und der Umfang des „Trusted Code“ beim Pageserver auf ein Minimum reduziert werden kann, ist die Wahrscheinlichkeit eines auch mit mehrstufigen Pageservern nicht behebbaren Fehlers als äußerst gering einzustufen. Daher ist es unter Umständen sinnvoller, eine Verifizierung des im Pageserver verwendeten Codes anzustreben. Andernfalls sollte bei der Entwicklung neuer Konzepte für den Pageserver berücksichtigt werden, dass die vom Pageserver intern verwendeten Daten auf Plausibilität geprüft werden sollten, um derartige interne Fehler überhaupt erkennen zu können. Gleichzeitig steigt durch das Hinzufügen von weiteren Komponenten die Wahrscheinlichkeit von Fehlern an, so dass sorgfältig abgewogen werden sollte, wie die effektive Ausfallsicherheit tatsächlich noch verbessert werden kann.

6.4 Strategien zum Sammeln von Seiten

Wie aus den Messungen in Kapitel 5.4.1 bereits ersichtlich wurde, ist der Zeitpunkt zum Sammeln von Seiten bei kurzen Abständen zwischen zwei Konsistenzierungen nicht ausschlaggebend für die Belastung des Clusters. Ebenfalls wurde jedoch gezeigt, dass für längere Intervalle die geeignete Auswahl der anzufordernden Seiten durchaus relevant ist. In weiteren Arbeiten kann diese Fragestellung genauer beleuchtet werden, wobei die folgenden Systematisierungen einen ersten Überblick über die dann zu diskutierenden Möglichkeiten geben.

6.4.1 Klassifizierung von Seiten

Um einen geeigneten Zeitpunkt zum Sammeln von Seiten bestimmen zu können, müssen die ausstehenden Seiten zuerst bewertet werden. Die Einteilung in heiße, warme, kühle und kalte Seiten für sehr oft, häufig, manchmal und selten geänderte Seiten einerseits oder selten bis vielfach im Cluster replizierte Seiten andererseits sei im Folgenden ausreichend, in der Praxis werden wohl feinere Abstimmungen sinnvoll sein.

Zunächst bietet sich eine Einteilung anhand der vergangenen logischen oder tatsächlichen Zeit zwischen letzter und vorletzter Änderung an. Geht man von einer gewissen Regelmäßigkeit der Zugriffe aus, kann dieser einzelne Wert direkt Auskunft über die Wärme einer Seite geben. Unter Umständen bietet sich auch die Protokollierung von mehreren Werten an, so dass mit einer bestimmten Anzahl an letzten Änderungen ein Durchschnitt oder eine Prognose erstellt werden kann. Alternativ könnte zur Bestimmung des Alters einer Seite sowie zur Abschätzung ihrer wahrscheinlichen Lebenszeit auch die Zählung der Änderungen in einem festen Zeitintervall genutzt werden, wobei eine hohe Änderungsfrequenz eine heiße, eine niedrige Änderungsfrequenz eine kalte Seite implizieren würde.

Des Weiteren könnten zur Klassifizierung von Seiten die Art der Zugriffe auf diese Seiten betrachtet werden. Das in Kapitel 2.2.5 vorgestellte Protokoll würde dann optimalerweise beim Versand des Writesets auch das Readset veröffentlichen, um die lesenden Zugriffe von Transaktionen beim Pageserver bekannt zu machen. Eine veränderte, jedoch im Cluster noch nicht gelesene Seite wurde folglich nicht repliziert und könnte somit als heiß eingestuft werden. Eine seit der letzten Änderung mehrfach gelesene Seite dagegen wurde bereits im Cluster repliziert und stünde somit auch bei einem Ausfall des Eigentümers innerhalb des Clusters zur Verfügung und könnte als kalt gelten. Ebenso wie bei der Bestimmung des Alters einer Seite ist hier auch die Veränderung über die Zeit interessant, um Vorhersagen machen zu können, ob eine momentan als heiß eingestufte Seite wahrscheinlich demnächst von vielen Maschinen angefragt wird und somit abkühlt, oder ob die Seite bei früheren Änderungen von keiner Maschine angefordert wurde und daher damit zu rechnen ist, dass auch die letzte Änderung für keine Maschine relevant ist und die Seite somit heiß bleibt. Heiße Seiten sind hier bei einem Ausfall des Knotens kritisch, kalte könnten ohne Hilfe des Pageservers im Cluster bereitgestellt werden.

In Kombination entfalten die vorgestellten Möglichkeiten zur Ermittlung der Temperatur einer Seite eine noch größere Aussagekraft, darüber hinausgehende Mechanismen sind zur vollständigen Ausleuchtung des Lösungsraums jedoch zusätzlich zu untersuchen.

6.4.2 Zeitpunkt für Seitenanfragen

Nachdem die anzufordernden Seiten klassifiziert wurden, können die möglichst kalten Seiten ohne Blockierung des Clusters angefragt und gesichert werden. Je näher die nächste Konsistenzierung jedoch kommt, desto wärmere Seiten sollten angefragt werden. Erst bei der tatsächlichen Konsistenzierung, die ohne ein Versionsmanagement eine Blockierung des Tokens erfordert (siehe Kapitel 5.4.1), werden die heißen Seiten angefordert.

Die direkt aus der Temperatur einer Seite ableitbare Strategie zur Ermittlung eines geeigneten Zeitpunktes für eine Seitenanfrage kann durch weitere Parameter verfeinert werden. Insbesondere zu erwähnen sind hier die beiden für den Pageserver wichtigen Kanäle zum Cluster und zur Festplatte. Einerseits wird der Cluster bei bereits ausgelastetem Netzwerk stärker behindert, als dies bei wenig ausgelastetem Netzwerk der Fall ist, andererseits muss der Pageserver alle Daten, die auf der Festplatte gespeichert werden sollen, im Hauptspeicher zwischenlagern. Ist die Datenrate der Festplatte jedoch mit den bisher vorhandenen Daten bereits voll nutzbar, müssen keine weiteren Daten spekulativ angefordert werden, denn sie benötigen Platz im Hauptspeicher und werden bis zu ihrem Transfer auf die Festplatte eventuell bereits wieder invalidiert.

Auch für die Wahl des Zeitpunkts zum Anfordern von geänderten Seiten lassen sich die angesprochenen Verfahren kombinieren und mit Sicherheit weiter verfeinern. Mit steigendem Durchsatz des Netzwerkes wird insbesondere die Bandbreite zur Festplatte wichtiger werden, so dass eine Steuerung mit Hilfe dieses Parameters für zukünftige Entwicklungen lohnend scheint.

6.5 Zusammenfassung

Die Erforschung von Konzepten für Pageserver, die wiederum einen Cluster bilden, scheint durchaus lohnend. Die Möglichkeiten zur Aufteilung der Aufgaben eines Pageservers sind vielfältig und vielversprechend, da sowohl der Durchsatz als auch die Ausfallsicherheit noch weiter erhöht werden können. Zur Erreichung dieser beiden Ziele sollten auch die Möglichkeiten bei Verwendung von mehreren Festplatten beleuchtet werden, wobei die zugrunde liegende Ansteuerung der einzelnen Festplatten teilweise oder sogar ganz aus dieser Arbeit übernommen werden kann.

Durch gezielt vorausschauendes Lesen kann ebenfalls unter Verwendung der in dieser Arbeit vorgestellten Konzepte die Latenz bei erstmaligen Anfragen auf Seiten nach einer Rücksetzung drastisch reduziert werden, so dass eine Beschleunigung der bereits sehr schnellen Rücksetzung möglich wird. Eine darüber hinausgehende Optimierung unter Zuhilfenahme von systemabhängigen Randbedingungen scheint je nach Einsatzgebiet Erfolg versprechend.

Aufbauend auf dieser Arbeit kann die Wahl geeigneter Zeitpunkte für die Anfrage von Seiten genauer erforscht werden, ein erster Überblick über denkbare Strategien wurde gegeben. Dieser Bereich ist ebenso wie die Verwendung von mehreren Festplatten für die Weiterentwicklung sinnvoll, um auch bei den zukünftig deutlich erhöhten Kapazitäten im Netzwerk gut funktionierende Konzepte bereitstellen zu können.

7 Zusammenfassung der Dissertation

Die in herkömmlichen Betriebssystemen vorhandene strikte Trennung zwischen Kern und Anwendungen zum Schutz des Systems einerseits und andererseits auch der Anwendungen untereinander kann alternativ unter Verwendung einer typsicheren Sprache umgangen werden, wie durch die Forschungen des Oberon-Projekts (siehe [WiGuPO92]) belegt wurde. Darauf aufbauend wird an der Universität Ulm ein verteiltes Betriebssystem namens Plurix entwickelt, mit dem neuartige Kommunikations- und Konsistenzmodelle, Programmiermethoden und Betriebssystemarchitekturen erforscht werden. Ziel der Forschung ist es, ein für Programmierer, Studenten und Anwender schlüssiges, verständliches und dabei dennoch effizientes System zu entwickeln.

In bisherigen Arbeiten wurde dazu unter anderem ein eigenständiger Compiler mit speziellen Laufzeitstrukturen (siehe [SchoPT02]), ein Kommunikationsmodell (siehe [WendKV03]) und eine für ein verteiltes Betriebssystem geeignete Speicherverwaltung (siehe [GoecSB05]) entworfen. Die identische Sicht aller im Cluster arbeitenden Knoten auf eine verteilte Halde hat sich dabei als sehr vorteilhaft erwiesen und erlaubt mit den von Datenbanken bekannten Transaktionen (siehe [DadaSR82]) eine einfache, verständliche und zuverlässige Programmierung. Die Vorteile von transaktional arbeitenden Betriebssystemen wurde bereits unter anderem in [StonVM84], [LiskDP88] und [BlacUT90] erkannt. Auch im verteilten Fall lassen sich die Eigenschaften Atomarität, Konsistenz und Isolierung in einem fehlerfrei arbeitenden System mit den in [WendKV03] und in dieser Arbeit vorgestellten Konzepten effizient implementieren. Wenn Fehler auftreten, ergeben sich allerdings Herausforderungen bei der Wiederherstellung eines konsistenten Zustands, deren Lösungsansätze bisher jedoch in einem sehr hohen Aufwand auch im fehlerfreien Betrieb resultierten oder sehr aufwendige Analysen im Falle eines Fehlers erforderten.

Die sich daraus ergebenden Fragestellungen wurden in dieser Arbeit ausführlich diskutiert und anhand des ausgeleuchteten Lösungsraums zu einem leichtgewichtigen und dennoch hocheffizienten Verfahren entwickelt, um Schnappschüsse in einem verteilten und auf Transaktionen basierenden Betriebssystem unter Berücksichtigung der Gegebenheiten heutiger Hardware zu erstellen und somit Persistenz für solche Systeme anzubieten. Die Messungen belegen eindrucksvoll, dass die Erstellung auch in äußerst kurzen Abständen von unter vier Sekunden möglich ist und dabei den Cluster dennoch nicht über Gebühr belastet – im Falle der gemessenen Anwendungen verbleiben über 96% der verfügbaren Zeit für die Anwendung selbst.

Wie auch bei nicht-verteilten transaktionalen Systemen erfordert die Integration von Gerätetreibern besonderes Augenmerk, da sich die Geräte außerhalb des transaktionalen Raums befinden. Das System muss dafür Sorge tragen, dass sich die Geräte beim Abbruch einer Transaktion wieder in dem konsistenten Zustand befinden, der zum logischen Zeitpunkt vor der abgebrochenen Transaktion passt. Die in dieser Arbeit entwickelte Systematik erlaubt eine effiziente Verwaltung solcher Informationen, so dass bei einer Rücksetzung nach einem Fehler auch die Gerätetreiber in den zum transaktionalen Speicher konsistenten Zustand versetzt werden können. Anhand des implementierten Prototyps wurde so belegt, dass die bisher bestehende Aufwandsgrenze zur Erstellung eines Schnappschusses selbst dann überwunden werden kann, wenn die Daten des Betriebssystems und ausgewählte Informationen der Gerätetreiber ebenfalls Bestandteil des Schnappschusses sind. Somit kann auch beim vollständigen Ausfall von Knoten die gesamte Umgebung der laufenden Transaktionen einschließlich der Zustände der Geräte wiederhergestellt werden. Obwohl die Erstellung von Schnappschüssen sehr effizient erfolgt und somit eine niedrige Geschwindigkeit bei einer Rücksetzung zu erwarten ist (siehe [DuNaAV04]), ist auch die Rücksetzung des Clusters auf ein gesichertes konsistentes Abbild äußerst schnell möglich. Der implementierte Prototyp erlaubt eine Rücksetzung des Systems mit Fast Ethernet in etwa 250 Millisekunden, wobei mit einem zusätzlichen Aufwand von ungefähr fünf Millisekunden pro Knoten eine zur Anzahl der Knoten annähernd lineare Skalierung erreicht wurde.

Der erzielte Durchsatz sowohl im fehlerfreien Betrieb als auch bei der Rücksetzung nach einem Fehler konnte nur erreicht werden, indem bestehende Konzepte wie übersichtliche Betriebssysteme (siehe [WiGuPO92]), orthogonale Persistenz (siehe auch [AtBaAP83], [DeCoND90], [LiBoMU94] und [HoChOP99]), transaktionale Konsistenz (siehe [WendKV03], [FrScTC05] und [FrLoPC05]) und Linear Segment (siehe [FrenPT02]) konsequent verfeinert und mit den in dieser Arbeit entwickelten Konzepten kombiniert wurden. Die in der Zukunft möglichen Veränderungen durch neuartige Speichermedien oder schnellere Netzwerke wurden diskutiert und Lösungsansätze für die dann auftretenden Problemstellungen aufgezeigt. Darüber hinaus kann auf Basis dieser Arbeit die Forschung in weiteren Richtungen fortgesetzt werden, beispielsweise anhand von gezielter Nutzung von Zusatzinformationen nach einer Rücksetzung oder zur Verteilung des Pageservers.

A Literatur

[AbraHM81] D.A. Abramson: „Hardware Management of a Large Virtual Memory“, Proceedings of the 4th Australian Computer Conference, p1-13, 1981

[AgbaRH02] A. Agbaria: „Reliability in High Performance Distributed Computing Systems“, PhD-Thesis at the Israel Institute of Technology, Haifa, Israel, 2002

[AtBaAP83] M.P. Atkinson, P. Bailey, K.J. Chisholm, W.P. Cockshott, R. Morrison: „An Approach to Persistent Programming“, The Computer Journal, vol26(4), p360-365, November 1983

[AtBuTP87] M.P. Atkinson, O.P. Buneman: „Types and Persistence in Database Programming Languages“, ACM Computing Survey, vol19(2), p105-190, Juni 1987

[AtDaOP96] M.P. Atkinson, L. Daynes, M.J. Jordan, T. Printezis, S. Spence: „An Orthogonally Persistent Java“, ACM SIGMOD 1996, vol25(4), p68-75, Dezember 1996

[AtJoDI96] M.P. Atkinson, M.J. Jordan, L. Daynes, S. Spence: „Design Issues for Persistent Java: A Type-safe Object-oriented, Orthogonally Persistent System“, Proceedings of the 7th International Workshop on Persistence and Java, p33-47, 1996

[AtMoOP95] M.P. Atkinson, R. Morrison: „Orthogonally Persistent Object Systems“, International Journal on Very Large Data Bases 4, p319-401, 1995

[BeGoCC80] P. Bernstein, N. Goodman: „Concurrency Control in Distributed Data Bases“, TODS, 03.1980, p18-51, 1980

[BlacUT90] A.P. Black: „Understanding Transactions in the Operating System Context“, Proceedings of the 4th Workshop on ACM SIGOPS European Workshop, p1-4, 1990

[BuCoSS01] R. Buyya, T. Cortes, H. Jin: „Single System Image (SSI)“, The International Journal of High Performance Computing Applications, vol15(2), p124-135, Sage Publications, 2001

[BundVM05] Entwicklungsstandard für IT-Systeme des Bundes: „Das V-Modell: Planung und Durchführung von IT-Vorhaben“, Stand September 2005

[CaFrFG94] M.J. Carey, M.J. Franklin, M. Zaharioudakis: „Fine-Grained Sharing in a Page Server OODBMS“, Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, p359-370, 1994

[ChJaOO04] M. Christerson, I. Jacobson, L.L. Constantine: „Object-Oriented Software Engineering“, Addison-Wesley, 2004

[ChJoDO97] R. Chow, T. Johnson: „Distributed Operating Systems and Algorithm Analysis“, Addison Wesley, 1997

[ChLaDS85] K.M. Chandy, L. Lamport: „Distributed Snapshots: Determining Global States of Distributed Systems“, ACM Transactions on Computer Systems, vol3(1), p63-75, Februar 1985

[ChLeRH94] P.M. Chen, E.K. Lee, G.A. Gibson, R.H. Katz, D.A. Patterson: „RAID: High-Performance, Reliable Secondary Storage“, ACM Computing Surveys (CSUR), vol26(2), p145-185, Juni 1994

[ClamDP91] S.M. Clamen: „Data Persistence in Programming Languages, A Survey“, CMU-CS-91-155, Pittsburgh, 1991

[CockOP83] W.P. Cockshott: „Orthogonal Persistence“, PhD Thesis, University of Edinburgh, Edinburgh TR CSR-21-83, Februar 1983

[CoCoES96] J. Cordsen, M. Cordsen, A. Gerischer, B. Oestmann, F. W. Schröer: „Evaluation of the SODA DVSM Consistency Framework“, in „The SODA Project“, Studien der GMD Nr. 301, ISBN 3-88457-301-2, Oktober 1996

[DadaSR82] P. Dadam: „Synchronisation und Recovery in verteilten Datenbanken, Konzepte und Grundlagen“, Dissertation an der FernUniversität Hagen, 1982

[DadaVD96] P. Dadam: „Verteilte Datenbanken und Client/Server-Systeme – Grundlagen, Konzepte, Realsierungsformen“, Springer-Verlag, Heidelberg, 1996

[DeBoGO94] A. Dearle, R. di Bona, J. Farrow, F. Henskens, A. Lindström, J. Rosenberg, F. Vaughan: „Grasshopper: An Orthogonally Persistent Operating System“, Computer Systems, vol7(3), p289-312, 1994

[DeCoND90] A. Dearle, R. Conner, F. Brown, R. Morrison: „Napier88 – A Database Programming Language?“, Proceedings of the 2nd International Workshop on Database Programming Languages, 1990

[DuNaAV04] T. Dumitras, P. Narasimhan: „An Architecture for Versatile Dependability“, DSN Workshop on Architecting Dependable Systems, Florenz, Italien, Juni 2004

[DWFuST90] D. DeWitt, P. Futtersack, D. Maier, F. Velez: „A Study of Three Alternative Workstation-Server Architectures for Objected-Oriented Database Systems“, Proceedings of the 16th International Conference on Very Large Databases, p107-121, Australien, 1990

[ElAlSR96] E.N. Elnozahy, L. Alvisi, Y.M. Wang, D.B. Johnson: „A Survey of Rollback-Recovery Protocols in Message-Passing Systems“, Technical Report CMU-CS-96-181, Carnegie Mellon University, Oktober 1996

[ElJoPC92] E.N. Elnozahy, D.B. Johnson, W. Zwaenepoel: „The Performance of Consistent Checkpointing“, Proceedings of Reliable Distributed Systems, p39-47, 1992

[ElMoNT81] J. Eliot, B. Moss: „Nested Transactions: An Approach to Reliable Distributed Computing“, Ph.D. Thesis at the Department of Electrical Engineering and Computer Science, MIT, April 1981

[EvoyAT81] D. McEvoy: „The Architecture of Tandem’s NonStop System“, Proceedings of the ACM’81 Conference, p245, 1981

[FeNaES04] P. Felber, P. Narasimhan: „Experiences, Strategies and Challenges in Building Fault-Tolerant CORBA Systems“, IEEE Transactions on Computers, vol53(5), p497-511, Mai 2004

[FeShPD99] P. Ferreira, M. Shapiro, X. Blondel, O. Fambon, J. Garcia, S. Kloosterman, N. Richer, M. Robert, F. Sandakly, G. Coulouris, J. Dollimore, P. Guedes, D. Hagimont, S. Krakowiak: „PerDiS: design, implementation, and use of a PERsistent DIstributed Store“, Lecture Notes in Computer Science, vol1752, p427-452, 1999

[FoRoMA05] C.L. Fok, G.C. Roman, C. Lu: „Mobile Agent Middleware for Sensor Networks: An Application Case Study“, Proceedings of the 4th International Conference on Information Processing in Sensor Networks (IPSN’05), Los Angeles, USA, 2005

[FrenPT02] S. Frenz: „Persistenz eines transaktionsbasierten verteilten Speichers“, Diplomarbeit an der Universität Ulm, Oktober 2002

[FrLoPC05] S. Frenz, R. Lottiaux, M. Schöttner, C. Morin, R. Göckelmann, P. Schulthess: „A Practical Comparison of Cluster Operating Systems Implementing Sequential and Transactional Consistency“, Proceedings of the 6th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP), Melbourne, Australien, 2005

[FrScPR04] S. Frenz, M. Schöttner, R. Göckelmann, P. Schulthess: „Parallel Ray-Tracing with a Transactional DSM“, Proceedings of the 4th IEEE/ACM International Symposium on Cluster Computing and the Grid, Chicago, USA, 2004

[FrScTC05] S. Frenz, M. Schöttner, R. Göckelmann, P. Schulthess: „Transactional Cluster Computing“, Proceedings of the 2005 International Conference on High Performance Computing and Communications (HPCC-05), Sorrent, Italien, 2005

[GeleGC85] D. Gelernter: „Generative Communication in Linda“, ACM Transactions Programming Language Systems, vol7(1), p80-112, 1985

[GhIeMS94] S. Ghandeharizadeh, D.J. Ierardi, R. Zimmermann: „Management of Space in Hierarchical Storage Systems“, Technical Report USC-CS-94-598, University of Southern California, November 1994

[GiScDB76] N.J. Giordano, M.S. Schwartz: „Data Base Recovery at CMIC“, ACM-SIGMOD, Conference on Management of Data, Juni 1976

[GoecSB05] R. Göckelmann: „Speicherverwaltung und Bootstrategien für ein Betriebssystem mit transaktionalem verteilten Heap“, Dissertation an der Universität Ulm, 2005

[GoScKR03] R. Göckelmann, M. Schöttner, S. Frenz, P. Schulthess: „A Kernel Running in a DSM – Design Aspects of a Distributed Operating System“, Proceedings of the IEEE International Conference on Cluster Computing, Hong Kong, 2003

[GrasCA04] M.A. Grasso: „Clinical Applications of Handheld Computers“, Proceedings of the 17th IEEE Symposium on Computer-Based Medical Systems (CBMS'04), 2004

[GrayTC81] J. Gray: „The Transaction Concept: Virtues and Limitations“, Proceedings of the 7th International Conference on Very Large Data Bases, p144-154, September 1981

[GrReTP93] J. Gray, A. Reuter: „Transaction Processing: Concepts and Techniques“, Morgan Kaufmann, 1993

[GrLaCT04] J. Gray, L. Lamport: „Consensus on Transaction Commit“, Microsoft Research, MSR-TR-2003-96, 2004

[HaRePT83] T. Haerder, A. Reuter: „Principles of Transaction-Oriented Database Recovery“, Computing Surveys, vol15(4), p287-317, Dezember 1983

[HeSiBE01] J. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, D. Ganesan: „Building Efficient Wireless Sensor Networks with LowLevel Naming“, Proceedings of the 18th Symposium of Operating System Principles, SOSP, 2001

[HitaDD05] Hitachi Global Storage Technologies: „Deskstar 7K250 Datasheet“, siehe http://www.hitachigst.com/hdd/support/d7k250/d7k250.htm, Stand 2005

[HoChOP99] A.L. Hosking, J. Chen: „PM3: An Orthogonally Persistent System's Programming Language – Design, Implementation, Performance“, Proceedings of the 25th VLDB Conference, Edinburgh, Schottland, 1999

[HuCaEI93] C. McHugh, V. Cahill: „Eiffel**: An Implementation of Eiffel on Amadeus, a Persistent, Distributed Applications Support Environment“, TOOLS Europe '93 Conference Proceedings, p47-62, TCD-CD-93-36, 1993

[KaRoWM01] S. Kannan, M. Roberts, P. Mayes, D. Brelsford, J.F. Skovira: „Workload Management with LoadLeveler“, IBM Redbooks, IBM Corporation 2001

[KeCoTD94] P. Keleher, A.L. Cox, S. Dwarkadas, W. Zwaenepoel: „TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems“, Proceedings of the Winter 1994 USENIX Conference, 1994

[KeedMS89] J.L. Keedy: „The MONADS-PC System: A Programmer's Overview“, Report 8/89, Universität Bremen, 1989

[KuRoOM81] H.T. Kung, J.T. Robinsons: „On Optimistic Methods for Concurrency Control“, Proceedings of ACM Transactional Database Systems, vol6(2), p213-226, Juni 1981

[LaLoNP93] B. Lampson, D. Lomet: „A New Presumed Commit Optimization for Two Phase Commit“, Digital Equipment Corporation Cambridge Research Lab, CRL 93/1, 1993

[LampPM01] L. Lamport: „Paxos Made Simple“, ACM SIGACT News on Distributed Computing, vol32(4), Dezember 2001

[LampTC78] L. Lamport: „Time, Clocks, and the Ordering of Events in a Distributed System“, Communications of the ACM, vol21(7), p558-565, Juli 1978

[LeLaEV04] V.C.S. Lee, K.W. Lam, T.W. Kuo: „Efficient Validation of Mobile Transactions in Wireless Environments“, Journal of Systems and Software Archive, vol69(1-2), p183-193, Januar 2004

[LiBoMU94] A. Lindström, R. di Bona, J.M. Farrow, F. Henskens, J. Rosenberg, A. Dearle, F. Vaughan: „A Model for User-Level Memory Management in a Distributed, Persistent Environment“, 17th Australian Computer Science Conference, Neuseeland, 1994

[LiCaPP99] B. Liskov, M. Castro: „Providing Persistent Objects in Distributed Systems“, Proceedings of ECOOP'99, 1999

[LiDuSD97] J.L. Lin, M.H. Dunham, M.A. Nascimento: „A Survey of Distributed Database Checkpointing“, Distributed and Parallel Databases, vol5(3), p289-319, 1997

[LiedPS93] J. Liedtke: „A Persistent System in Real Use“, Proceedings of the International Workshop on Object-Orientation in Operating Systems (IWOOOS), 1993

[LiIVYS88] K. Li: „IVY: A Shared Virtual Memory System for Parallel Computing“, Proceedings of the International Conference on Parallel Processing, 1988

[LiskDP88] B. Liskov: „Distributed Programming in Argus“, Communications of the ACM Archive, vol31(3), p300-312, 1988

[MaBoTC00] S. Mann, A. Borusan, H. Ehrig, M. Große-Rhode, R. Mackenthun, A. Sünbül, H. Weber: „Towards a Component Concept for Continuous Software Engineering“, Fraunhofer Institut ISST, Interner Bericht 55/00, Oktober 2000

[MiRaSC95] M. Mizuno, M. Raynal, J.Z. Zhou: „Sequential Consistency in Distributed Systems: Theory and Implementation“, LNCS 938, Theory and Practice in Distributed Systems, p224-241, 1995

[MoBrEM04] K. Moll, M. Broy, M. Pizka, T. Seifert, K. Bergner, A. Rausch: „Erfolgreiches Management von Software-Projekten“, Informatik-Spektrum, Oktober 2004, Springerverlag, 2004

[MoGaTE04] C. Morin, P. Gallard, R. Lottiaux, G. Vallée: „Towards an Efficient Single System Image Cluster Operating System“, Future Generation Computer Systems, vol20 (2), Januar 2004

[MoLiEC83] C. Mohan, B. Lindsay: „Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions“, Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing, 1983

[MoLoKS03] C. Morin, R. Lottiaux, G. Vallée, P. Gallard, G. Utard, R. Badrinath, L. Rilling: „Kerrighed: a Single System Image Cluster Operating System for High Performance Computing“, Proceedings of Europar 2003: Parallel Processing, vol2790 of Lecture Notes in Computer Science, p1291-1294, August 2003

[MoRaSD91] A. Mohindra, U. Ramachandran: „A Survey of Distributed Shared Memory in Loosely-Coupled Systems“, Technical Report GIT-CC-91/01, College of Computing, Georgia Institute of Technology, Januar 1991

[MosbMC93] D. Mosberger: „Memory Consistency Models”, Technical Report, University of Arizona, USA, 1993

[NelsLD89] M. Nelson: „LZW Data Compression“, Dr. Dobb’s Journal, Oktober 1989

[NoBrWO97] K. Norvag, K. Bratbergsengen: „Write Optimized Object-Oriented Database Systems“, Proceedings of SCCC’97, Valparaiso, Chile, November 1997

[OraDAG92] Oracle9i Database Administrator’s Guide, Release 2 (9.2), siehe dazu auch http://www.lc.leidenuniv.nl/awcourse/oracle/server.920/a96521/ds_txns.htm

[OuDoBI89] J. Ousterhout, F. Douglis: „Beating the I/O Bottleneck: A Case for Log-Structured File Systems“, Operating Systems Review, vol23(1), p11-28, 1989

[OzVaPD91] M.T. Özsu, P. Valduriez: „Principles of Distributed Database Systems“, Prentice-Hall International, 1991

[PuChAP93] C. Pu, S.F. Chen: „ACID Properties Need Fast Relief: Relaxing Consistency Using Epsilon Serializability“, Proceedings of the 5th International Workshop on High Performance Transaction Systems, Asilomar, Kalifornien, USA, September 1993

[RandSS75] B. Randell: „System Structure for Software Fault-Tolerance“, IEEE Transactions on Software Engineering, vol1(2), Februar 1975

[RajwSB02] R. Rajwar: „Speculation-Based Techniques for Lockfree Execution of Lock-Based Programs“, Ph.D. Thesis, University of Wisconsin, Madison, USA, 2002

[RazECO93] Y. Raz: „Extended Commitment Ordering or Guaranteeing Global Serializability by Applying Commitment Order Selectivity to Global Transactions“, Proceedings of the 12th ACM Symposium on Principles of Database Systems, ACM Press, p83-96, Washington, USA, 1993

[ReBaRS05] H.P. Reiser, U. Bartlang, F.J. Hauck: „A Reconfigurable System Architecture for Consensus-based Group Communication“, Proceedings of the 17th IASTED International Conference on Parallel and Distributed Computing Systems, PDCS, 2005

[ReitSG96] M.K. Reiter: „A Secure Group Membership Protocol“, IEEE Transactions on Software Engineering, vol22(1), Januar 1996

[RobeLH99] A. Robertson: „Linux High Availability“, Linux High Availability Project, 1999, siehe dazu auch http://www.linux-ha.org/

[RoDeOS97] J. Rosenberg, A. Dearle, D. Hulse, A. Lindström, S. Norris: „Operating System Support for Persistent and Recoverable Computations“, Communications of the ACM, 1997

[RoHeSP90] J. Rosenberg, F.A. Henskens, A.L. Brown, R. Morrison, D. Munro: „Stability in a Persistent Store Based on a Large Virtual Memory“, Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence of Information, Springer, p229-245, 1990

[RoKeAM92] J. Rosenberg, J. Keedy, D. Abramson: „Addressing Mechanisms for Large Virtual Memories“, The Computer Journal, vol35(4), p369-375, November 1992

[RomaCA01] A. Romanovsky: „Coordinated Atomic Actions: How to Remain ACID in the Modern World“, ACM SIGSOFT Software Engineering Notes, vol26(2), p66-68, März 2001

[RoOuDI91] M. Rosenblum, J.K. Ousterhout: „The Design and Implementation of a Log-Structured File System“, Proceedings of the 13th ACM Symposium on Operating System’s Principles, 1991

[ScAiMD03] J. Schindler, A. Ailamaki, G.R. Ganger: „Matching Database Access Patterns to Storage Characteristics“, Proceedings of the VLDB 2003 PhD Workshop, Berlin, Deutschland, 2003

[ScAiLR03] J. Schindler, A. Ailamaki, G.R. Ganger: „Lachesis: Robust Database Storage Management Based on Device-specific Performance Characteristics“, Proceedings of the 29th VLDB Conference, Berlin, Deutschland, 2003

[ScBiDO02] P. Schulthess, T. Bindhammer, R. Goeckelmann, M. Schoettner, M. Wende: „A DSM Operating System for Persistent Objects“, Proceedings of the 14th IASTED International Conference on Parallel and Distributed Computing and Systems, Cambridge, USA, 2002

[ScBoAI03] F. Schweiggert, A. Borchert, M. Grabert, J. Mayer: „Allgemeine Informatik III“, Vorlesungsunterlagen, Universität Ulm, 2003

[ScFrCR04] M. Schöttner, S. Frenz, R. Göckelmann, P. Schulthess: „Checkpointing and Recovery in a transaction-based DSM Operating System“, Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks, Innsbruck, Österreich, 2004

[ScFrFT04] M. Schöttner, S. Frenz, R. Göckelmann, P. Schulthess: „Fault Tolerance in a DSM Cluster Operating System“, Workshop on „Dependability and Fault Tolerance“ within the International Conference on Architecture of Computing Systems, Augsburg, Deutschland, 2004

[SchuBS04] P. Schulthess: „Betriebssysteme“, Vorlesungsunterlagen, Universität Ulm, 2004

[SchoPT02] M.F.W. Schöttner: „Persistente Typen und Laufzeitstrukturen in einem Betriebssystem mit verteiltem virtuellen Speicher“, Dissertation an der Universität Ulm, 2002

[SeStTS90] M. Seltzer, M. Stonebraker: „Transaction Support in Read Optimized and Write Optimized File Systems“, Proceedings of the 16th VLDB Conference, Brisbane, Australien, 1990

[SkibTV01] S.M. Skibicki: „Transaktionssicherung im verteilten virtuellen Speicher“, Diplomarbeit an der Universität Ulm, Juni 2001

[StalBP03] W. Stallings: „Betriebssysteme: Prinzipien und Umsetzung“, Übersetzung der 4. Auflage, Prentice Hall, 2003

[StanDO04] J. Stanton: „Distributed Operating Systems“, Lecture CS 251, 2004/6, Department of Computer Science, George Washington University, USA, 2004

[StonVM84] M. Stonebraker: „Virtual Memory Transaction Management“, ACM SIGOPS Operating Systems Review, vol18(2), p8-16, 1984

[SunEJB02] Sun Enterprise Java Beans, „Global and Local Transactions“, siehe dazu auch http://docs.sun.com/source/816-7151-10/detrans.html

[T13ATA02] Technical Committee T13: „AT Attachment - 6 with Packet Interface“, ANSI NCITS 361-2002, siehe dazu auch http://www.t13.org

[TaneMB02] A.S. Tanenbaum: „Moderne Betriebssysteme“, Übersetzung der 2. Auflage, Prentice Hall, 2002

[TaStDS02] A.S. Tanenbaum, M.v. Steen: „Distributed Systems: Principles and Paradigms“, Prentice Hall, 2002

[ThomMC79] R.H. Thomas: „A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases“, ACM Transactional Database Systems, vol4(2), p180-209, Juni 1979

[TrauSK96] S. Traub: „Speicherverwaltung und Kollisionsbehandlung in transaktionsbasierten verteilten Betriebssystemen“, Dissertation an der Universität Ulm, 1996

[TsWaED99] J. Tsai, Y.M. Wang, S.Y. Kuo: „Evaluations of Domino-Free Communication-Induced Checkpointing Protocols“, Information Processing Letters, vol69(1), p31-37, Januar 1999

[WelcTH84] T. Welch: „A Technique for High-Performance Data Compression“, Computer, Juni 1984

[WendKV03] M. Wende: „Kommunikationsmodell eines verteilten virtuellen Speichers“, Dissertation an der Universität Ulm, 2003

[WeThMT03] L. Welling, L. Thomson: „MySQL Tutorial: A concise introduction to the fundamentals of working with MySQL“, MySQL Press, November 2003

[WiGuPO92] N. Wirth, J. Gutknecht: „Project Oberon, The Design of an Operating System and Compiler“, ACM Press, 1992

[WiMuED96] T. Wilkinson, K. Murray: „Evaluation of a Distributed Single Address Space Operating System“, IEEE Proceedings of the 16th ICDCS, 1996

[VaLoGP05] G. Vallée, R. Lottiaux, D. Margery, C. Morin, J.-Y. Berthou: „Ghost Process: a Sound Basis to Implement Process Duplication, Migration and Checkpoint/Restart in Linux Clusters“, 4th International Symposium on Parallel and Distributed Computing, Lille, Frankreich, p97-104, Juli 2005

[ZaCaHA98] M. Zaharioudakis, M.J. Carey: „Hierarchical, Adaptive Cache Consistency in a Page Server OODBMS“, IEEE Transactions on Computers, vol47(4), p427-444, 1998

[ZiLeUA77] J. Ziv, A. Lempel: „A Universal Algorithm for Sequential Data Compression“, IEEE Transactions on Information Theory, Mai 1977

B Referenz-Paketspezifikation

Die in Kapitel 2.2.5 dargestellten Anforderungen an ein Kommunikationsprotokoll für transaktionale Konsistenz können direkt in eine Spezifikation umgewandelt werden, die auch vom implementierten Prototypen genutzt wird. Der dazugehörige Aufbau der Pakete ist in diesem Teil des Anhangs dargestellt. Die Pakete sind von links nach rechts und von oben nach unten zu lesen, wobei auf die Wiederholung der bei allen gleich aufgebauten Header für Ethernet- und IP-Schicht verzichtet wird.

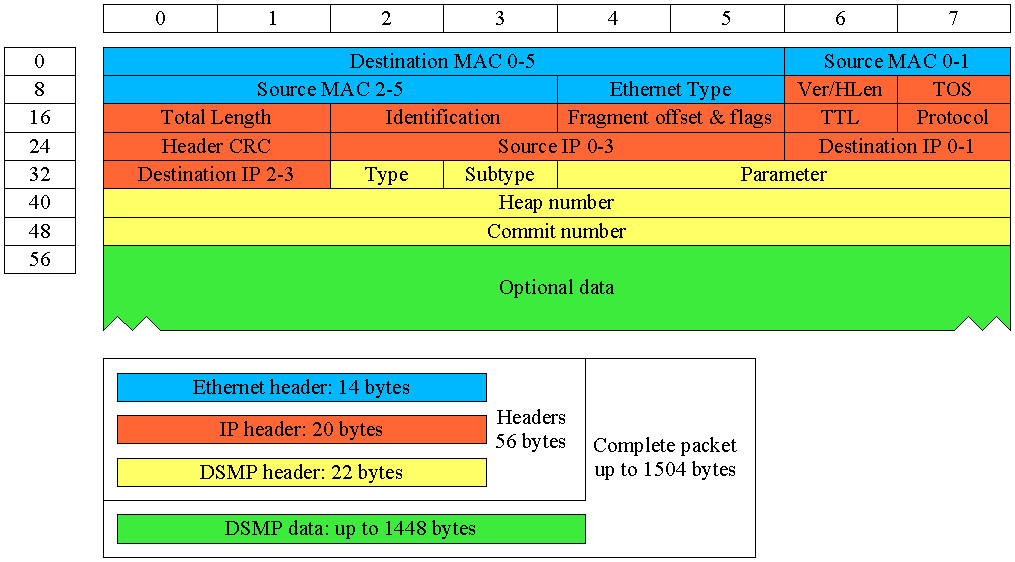

Abbildung B.1: Allgemeiner Paketaufbau

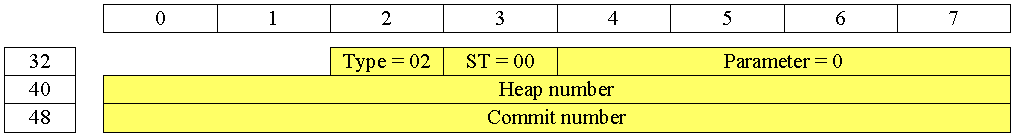

Abbildung B.2: Clustererkennung (AliveReq)

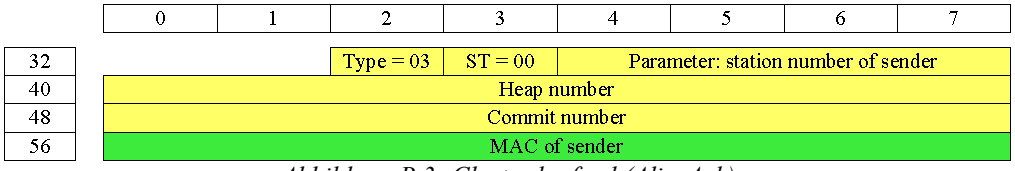

Abbildung B.3: Cluster laufend (AliveAck)

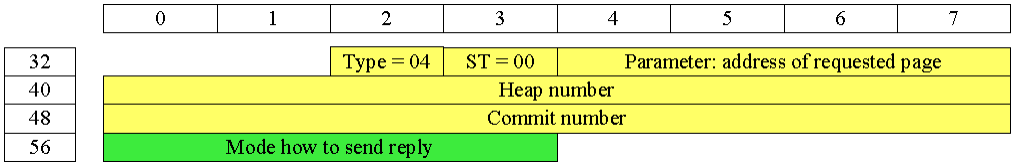

Abbildung B.4: Seitenanfrage (PageReq)

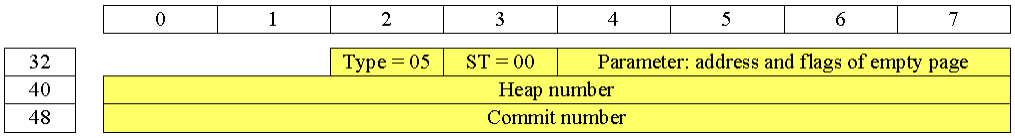

Abbildung B.5: Seite ist leer (PageEmpty)

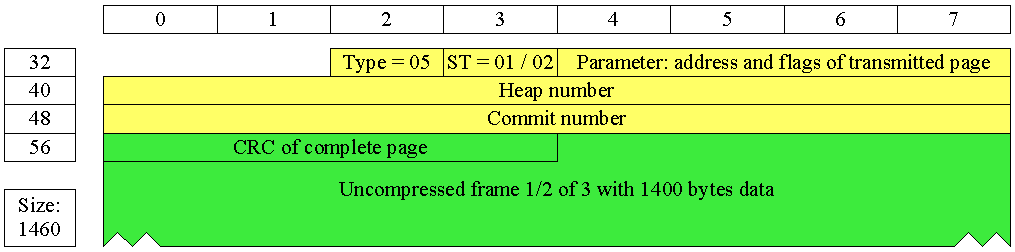

Abbildung B.6: Unkomprimierte Seitendaten Teil 1/2 von 3 (PageFrame1/2)

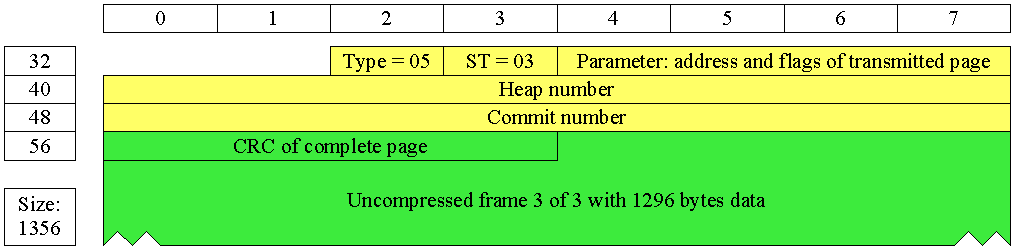

Abbildung B.7: Unkomprimierte Seitendaten Teil 3 von 3 (PageFrame3)

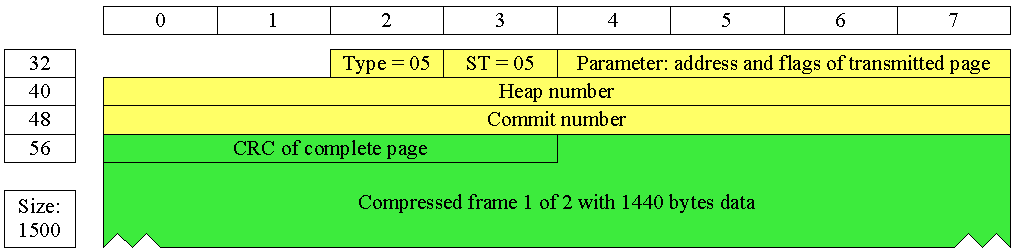

Abbildung B.8: Komprimierte Seitendaten Teil 1 von 2 (PageCompr1)

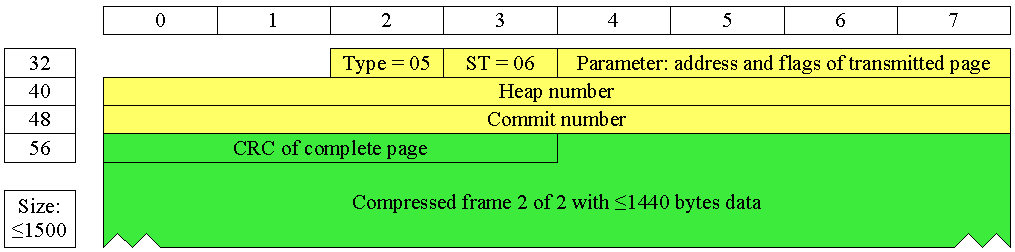

Abbildung B.9: Komprimierte Seitendaten Teil 2 von 2 (PageCompr2)

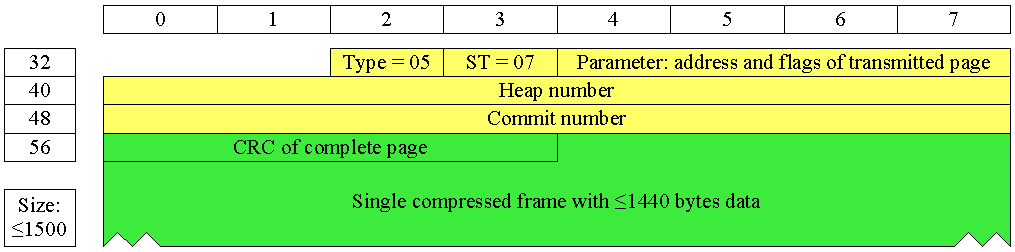

Abbildung B.10: Komprimierte vollständige Seitendaten (PageComprSingle)

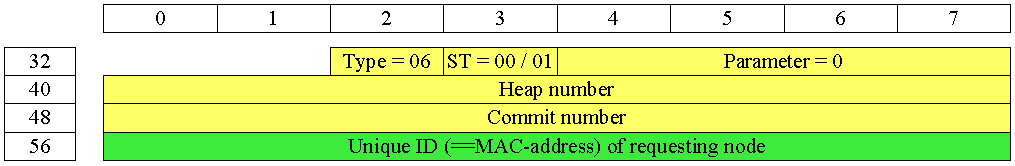

Abbildung B.11: Tokenanfrage (TokenReq)

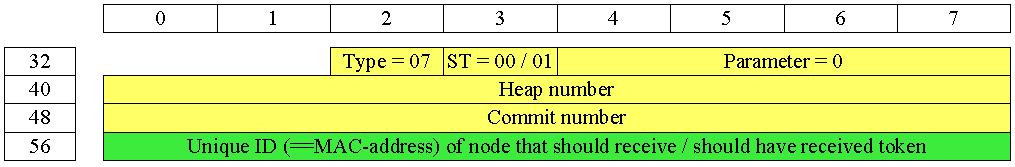

Abbildung B.12: Tokengenehmigung (TokenGranted)

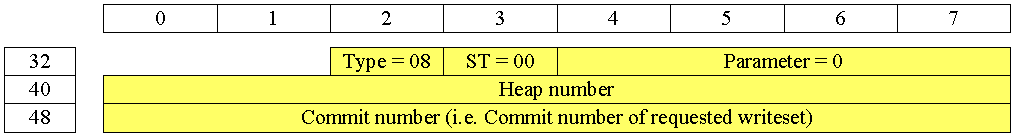

Abbildung B.13: Anfrage auf altes Writeset (WritesetReq)

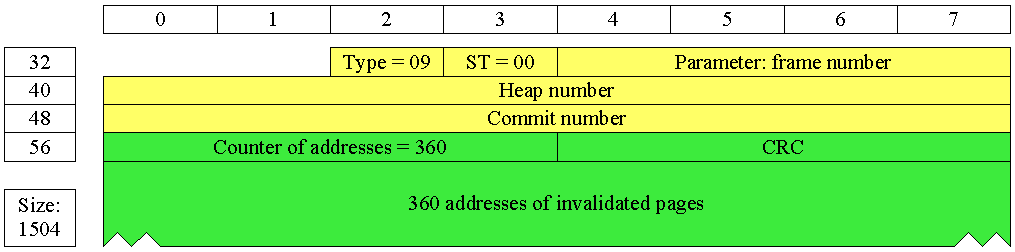

Abbildung B.14: Fragment eines Writesets (WritesetFrag)

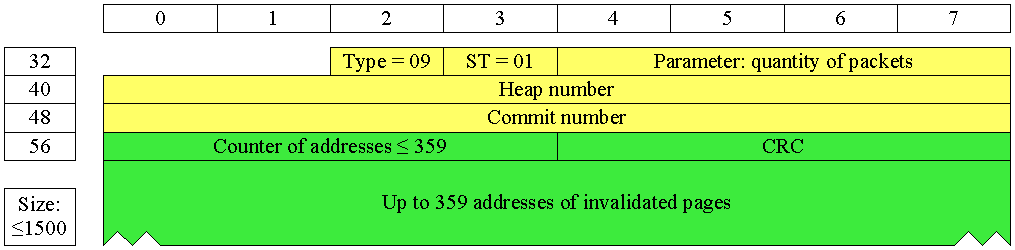

Abbildung B.15: Vollständiges Writeset oder letztes Fragment (WritesetFinal)

C Abbildungsverzeichnis

D Tabellenverzeichnis

E Lebenslauf

Vor- und Zuname:	Stefan Martin Frenz
Geburtstag und Geburtsort:	16. April 1978 in Ulm
Staatsangehörigkeit:	deutsch
Familienstand:	ledig

Ausbildung:

1984 – 1988	Grundschule in Stuttgart-Ostheim
1988 – 1997	Eberhard-Ludwigs-Gymnasium in Stuttgart
Juni 1997	Abitur am Eberhard-Ludwigs-Gymnasium in Stuttgart
Okt. 1997 – Nov. 2002	Studium der Informatik mit Nebenfach Mathematik an der Universität Ulm
Nov. 2002	Informatik-Diplom der Universität Ulm
seit Jan. 2003	wissenschaftlicher Mitarbeiter in der Abteilung Verteilte Systeme an der Universität Ulm

Praktische Tätigkeiten vor 2003:

Juli 1997 – März 1998	Anstellung im DaimlerBenz-Forschungszentrum (heute DaimlerChrysler) in Ulm im Bereich Datenbankerstellung, WebServices und Dokumentenpflege
Feb. 2000 – Feb. 2002	Anstellung in der Abteilung Verteilte Systeme der Universität Ulm im Bereich Treiberentwicklung und Dateizugriffsschnittstellen für das Plurix-Projekt
Apr. 2000 – Sep. 2000	Anstellung in der Abteilung für Stochastik der Universität Ulm im Bereich Simulation stochastischer Modelle
Apr. 2002 – Juli 2002	Anstellung in der Abteilung für Analysis der Universität Ulm zur Betreuung der vorlesungsbegleitenden Übungen

Veröffentlichungen:

S. Frenz: „Persistenz eines transaktionsbasierten verteilten Speichers“, Diplomarbeit, Universität Ulm, Deutschland, 2002

R. Göckelmann, M. Schöttner, S. Frenz, P. Schulthess: „A Kernel Running in a DSM – Design Aspects of a Distributed Operating System“, IEEE International Conference on Cluster Computing, Hong Kong, 2003

R. Göckelmann, S. Frenz, M. Schöttner, P. Schulthess: „Compiler Support for Reference Tracking in a type-safe DSM“, Joint Modular Languages Conference (JMLC), Klagenfurt, Österreich, 2003

M. Schöttner, S. Frenz, R. Göckelmann, P. Schulthess: „Checkpointing and Recovery in a transaction-based DSM Operating System“, IASTED International Conference on Parallel and Distributed Computing and Networks, Innsbruck, Österreich, 2004

R. Göckelmann, M. Schöttner, S. Frenz, P. Schulthess: „Plurix, a Distributed Operating System Extending the Single System Image Concept“, IEEE Canadian Conference on Electrical and Computer Engineering, CCECE04, Niagara Falls, Kanada, 2004

S. Frenz, M. Schöttner, R. Göckelmann, P. Schulthess: „Parallel Ray-Tracing with a Transactional DSM“, 4th IEEE/ACM International Symposium on Cluster Computing and the Grid, Chicago, USA, 2004

M. Schöttner, S. Frenz, R. Göckelmann, P. Schulthess: „Fault Tolerance in a DSM Cluster Operating System“, Workshop on „Dependability and Fault Tolerance“, im Rahmen der International Conference on Architecture of Computing Systems, Augsburg, Deutschland, 2004

R. Göckelmann, M. Schöttner, S. Frenz, P. Schulthess: „Type-Safe Object Exchange Between Applications and a DSM Kernel“, International Workshop on „Operating Systems, Programming Environments and Management Tools for High-Performance Computing on Clusters“ (COSET-1), im Rahmen der ACM International Conference on Supercomputing (ICS ’04), Saint Malo, Frankreich, 2004

M. Fakler, S. Frenz, R. Göckelmann, M. Schöttner, P. Schulthess: „An interactive 3D world built on a transactional operating system“, IEEE Canadian Conference on Electrical and Computer Engineering, CCECE05, Saskatoon, Kanada, 2005

M. Fakler, S. Frenz, R. Göckelmann, M. Schöttner, P. Schulthess: „Project Tetropolis – Application of Grid Computing to Interactive Virtual 3D Worlds“, International Conference on Hypermedia and Grid Systems, Opatija, Kroatien, 2005

S. Frenz, R. Lottiaux, M. Schöttner, C. Morin, R. Göckelmann, P. Schulthess: „A Practical Comparison of Cluster Operating Systems Implementing Sequential and Transactional Consistency“, 6th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP), Melbourne, Australien, 2005

S. Frenz, M. Schöttner, R. Göckelmann, P. Schulthess: „Transactional Cluster Computing“, 2005 International Conference on High Performance Computing and Communications (HPCC-05), Sorrent, Italien, 2005