Modellierung des Online-Steerings von Gridjobs als Zugriffe auf verteilten gemeinsamen Speicher

Vom Fachbereich Elektrotechnik und Informatik der Universität Siegen

zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation

von

Dipl.-Inform. Daniel Lorenz

1. Gutachter: Prof. Dr. Roland Wismüller
2. Gutachter: Prof. Dr. Christoph Ruland
Vorsitzender: Prof. Dr. Hans Wojtkowiak

Tag der mündlichen Prüfung: 15. 7. 2009

gedruckt auf alterungsbeständigem holz- und säurefreiem Papier

Danksagung

Diese Arbeit wäre ohne die Unterstützung verschiedener Personen, denen ich an dieser Stelle dafür danken möchte, sicherlich nicht möglich gewesen.

Zuerst möchte ich Prof. Dr. Roland Wismüller danken, der diese Arbeit betreut und sich durch verschiedene unausgereifte Entwürfe gearbeitet hat. In verschiedenen Gesprächen gab er immer wieder Anregungen und wichtige Impulse, die das Entstehen dieser Arbeit beeinflusst haben.

Dr. Wolfgang Walkowiak und Dr. Torsten Stahl aus der Fachgruppe Experimentelle Teilchenphysik danke ich für ihre Quellenhinweise und ihre Erklärungen über die physikalischen und anwendungsbezogenen Hintergründe zum ATLAS Experiment und der dort verwendeten Software. Dadurch haben sie für mich die Einarbeitung in diesen Bereich deutlich vereinfacht.

Mein Dank gilt auch Prof. Dr. Peter Buchholz, der durch seine Unterstützung wesentlich zur Fertigstellung dieser Arbeit beigetragen hat.

Prof. Dr. Christoph Ruland danke ich, dass er sich bereit erklärt hat das Zweitgutachten für diese Arbeit zu erstellen.

Schließlich möchte ich meinen Eltern für alle emotionale Unterstützung danken, die sie mir haben zukommen lassen, während ich an dieser Dissertation gearbeitet habe.

II

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

Abkürzungsverzeichnis

Abb.	Abbildung
AktP.	Aktualisierungsprotokoll
bzgl.	bezüglich
bzw.	beziehungsweise
CA	Certification Authority (Zertifizierungsstelle für Zertifikate)
CE	Computing Element, Frontend-Rechner eines Rechenressource
DSM	distributed shared memory, Verteilter gemeinsamer Speicher
evtl.	eventuell
GB	Gigabyte
Gb	Gigabit
GSI	Grid-Sicherheits-Infrastruktur (Grid security infrastructure)
HEP	Hochenergiephysik
InvP.	Invalidierungsprotokoll
Kap.	Kapitel
kB	Kilobyte
kb	Kilobit
LB	Logging and Bookkeeping Service
LCG	LHC Computing Grid
LHC	Large Hadron Collider (Großer Hadronenbeschleuniger)
MB	Megabyte
Mb	Megabit
MD	Molekulardynamik (molecular dynamics)
ND	Namensdienst
oBdA.	ohne Beschränkung der Allgemeinheit
PKI	Public Key Infrastrukture
PLU	Problem-Lösungs-Umgebung (problem solving environments)
RB	Ressource Broker
sog.	sogenannt
SSK	Spezielle Schwache Konsistenz
Std.abw.	Standardabweichung
Tab.	Tabelle
u.a.	unter anderem
UI	User Interface (Benutzerschnittstelle)
VD	Verbindungsdienst
VO	virtuelle Organisation (virtual organization)
VOMS	VO membership service (VO Mitgliedschaftsdienst)
VSK	Verzögerte Schwache Konsistenz
WN	Worker Node, Arbeitsknoten einer Gridsite
z.B.	zum Beispiel

Kapitel 1

Einleitung und Motivation

In vielen Bereichen der Wissenschaft, z.B. der Hochenergiephysik, Medizin, Astronomie, Biologie und Geologie, spielen Computersimulationen oder aufwendige, computergestützte Berechnungen eine immer wichtigere Rolle. Typischerweise werden diese Berechnungen auf speziellen Großrechenanlagen oder Rechnerfarmen in sogenannten Batchsystemen durchgeführt. Diese Batchsysteme besitzen einen sichtbaren Rechner, der die gesamte Anlage nach außen repräsentiert, das *Computing Element* (CE) und mehrere *Worker Nodes* (WN) genannte Arbeitsknoten. Dabei ist das Computing Element (CE) nur dafür zuständig, die Aufträge entgegenzunehmen und an ein Worker Node weiterzuleiten. Die Anzahl der Worker Nodes kann mehrere hundert Stück pro Computing Element betragen. Aber auch Computing Elemente mit wenigen Worker Nodes sind vorhanden. Ein Arbeitsauftrag wird Job genannt. Solch ein Job kann mehrere Stunden oder Tage rechnen und wird autonom ausgeführt. Interaktion mit einem Job ist nicht ohne weiteres möglich. In vielen Fällen sind die Worker Nodes durch Firewalls geschützt oder liegen in einem privaten Netzwerk und nur das Computing Element ist nach außen sichtbar.

In zunehmendem Maße werden Ressourcen in sogenannten Grids [56, 53, 15] zusammengeschlossen. Hinter dem Grid steckt die Idee, Ressourcen mit anderen zu teilen. Die Ressourcen können dabei unterschiedlichster Art sein, z.B. Rechner, Speicher, Teleskope, Mikroskope, etc. Die Ressourcen werden von unterschiedlichen, unabhängigen Betreibern zur Verfügung gestellt. Eine *Virtuelle Organisation* (VO) [56] definiert die Regeln für die Benutzung der Ressourcen, die Rechte der Benutzer und Ressourcenbetreiber sowie die Ressourcenbelegung. Die Ressourcen und Dienste einer VO bilden somit ein konkretes Grid. Für die Verwaltung der Ressourcen, Auftragsverteilung, Zustandskontrollen, Monitoring, Accounting, etc. werden entsprechende Dienste benötigt, die die Grid-Middleware bilden. Wenn ein Benutzer einen Job ins Grid submittiert, weist die Grid-Middleware dem Job nach den entsprechenden Regeln der VO eine passende freie Ressource zu. Das Grid geht von nicht-interaktiven Jobs aus. Das Ergebnis seines Jobs sieht der Benutzer erst nachdem sein Job erfolgreich beendet wurde. Falls er nicht erfolgreich beendet wurde, erhält er bei einigen populären Grid-Middlewares wie Globus [51, 164] oder gLite [64] keinerlei Ausgaben zurück.

Während Interaktivität im Grid nicht vorgesehen wurde, wünschen sich

Wissenschaftler interaktive Kontrolle über ihren Job [24]. Das Ziel des Online-Steering ist es, interaktiv während der Laufzeit der Anwendung:

- Zwischenergebnisse, Parameter und/oder Performance-relevante Daten der Anwendung zu visualisieren.
- Parameter oder Konfiguration der Anwendung zu ändern.
- die Ausführung der Anwendung zu steuern. Z.B. den Job anhalten, neu starten oder terminieren oder Lastverteilung bei mehreren Prozessen ausgleichen.

Durch die Benutzung von Online-Steering-Systemen können Ressourcen und Zeit gespart werden. Typischerweise erstellt und konfiguriert der Benutzer zunächst seinen Gridjob und submittiert den Job anschließend ins Grid. Während der Ausführung des Gridjobs, kann der Benutzer nicht mit seinem Job interagieren, er muss also bis zur vollständigen Beendigung der Berechnung warten. Nachdem der Job beendet ist, kann der Benutzer auf Ausgaben zugreifen, sofern der Job erfolgreich beendet wurde. Falls die Ergebnisse fehlerhaft oder nutzlos sind, z.B. weil ein Parameter nicht korrekt gesetzt war, muss der ganze Kreislauf wiederholt werden (siehe Abb 1.1). Oft geht es auch darum einen bestimmten Parameter zu optimieren, wobei der Wertebereich in mehreren Durchläufen immer mehr eingeschränkt wird. In diesen Fällen muss der gesamte Kreislauf mehrmals wiederholt werden. Mit Online-Steering kann hier mit lediglich einem Job online experimentiert werden, wodurch lange Wartezeiten und viele unnötige Rechenschritte wegfallen. Der Benutzer kann die Zwischenergebnisse des Jobs zur Laufzeit ansehen und hat dadurch viel früher Zugang zu Ergebnissen. Dies erlaubt es dem Benutzer, eher Entscheidungen für das weitere Vorgehen zu treffen und frühzeitig auf die Ergebnisse zu reagieren, z.B. indem Parameter in einem Job online angepasst werden. Ein anderer Anwendungsfall ist die Suche nach Fehlern, die aus den (möglicherweise fehlenden) Ausgaben nicht einfach zu rekonstruieren sind und deren Suche mehrere Durchläufe mit unterschiedlichen Konfigurationen benötigt. Mit Online Steering kann der Job interaktiv schrittweise ausgeführt und die Entstehung der Ergebnisse im Detail nachvollzogen werden.

Die meisten bestehenden Online-Steering-Systeme, wie z.B. Progress [178], Magellan [181], Autopilot [149] oder Falcon [70], basieren auf einem Client/Server Modell, bei dem die Anwendung als Server gesehen wird, der Daten bereit stellt, und das Steuerungswerkzeug als Client. Die Grundidee des DSM Modells für Online-Steering beruht auf der Beobachtung, dass die Anwendung eine Berechnung auf lokal vorhandenen Daten durchführt. Mit dem Steering-System greift ein zusätzlicher Prozess auf dieselben Daten zu, um diese zu visualisieren oder zu modifizieren. Folglich kann man diese gemeinsame Nutzung desselben Datenobjekte durch einen virtuellen gemeinsamen Speicher darstellen.

Die Verwendung eines Modells, welches auf verteiltem gemeinsamen Speicher (DSM, distributed shared memory) beruht, bringt einige Vorteile mit sich:

- Datenkonsistenz: Bisher wurden die Effekte der Modifikation von Daten einer Anwendung durch einen externen Steeringprozess kaum untersucht. Lediglich in Pathfinder [74] wird ein Mechanismus beschrieben, der sequentielle Konsistenz [103] garantiert. Ein allgemeines Konsistenzmodell wurde

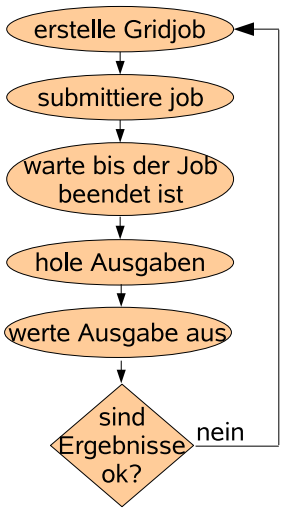


Abbildung 1.1: Arbeitsablauf bei der Verwendung von Gridjobs ohne Online-Steering

nie definiert. In der Regel wird die Konsistenz der Daten der Verantwortung des Anwenders überlassen. Ein DSM-Modell nimmt dem Anwender diese Aufgabe ab, indem es, in Abhängigkeit von den entsprechenden Bedingungen, Konsistenzmodelle bereitstellt, die die Konsistenz der Daten in der Anwendung sicher stellen.

- Transportoptimierung: Es gibt für bestehende DSM Systeme etliche Untersuchungen und Techniken, um die Performance zu verbessern und die Kommunikation zu verringern. Bestehende Steering-Systeme optimieren die Performance im Wesentlichen durch Filterung der Ereignisse oder durch Reduzierung der überwachten Daten. Dies wird erreicht, indem Sensoren abgeschaltet werden oder gegen Sensoren ausgetauscht werden, die weniger Daten generieren. Diese Techniken lassen sich bei einem DSM-orientierten Ansatz auf Online-Steering übertragen.

- Instrumentierung: Der Aufwand eine Anwendung zu instrumentieren ist ein wichtiger Grund dafür, dass Online-Steering-Systeme oft nicht verwendet werden. Oft müssen komplexe Objektwrapper-Klassen erzeugt, oder der gesamte Quellcode mit Aufrufen zur Steering-Bibliothek instrumentiert werden. DSM-Systeme bieten eine intuitive Schnittstelle, die die Integration von Online Steering in bestehende Systeme vereinfacht.

1.1 Ziele und wissenschaftlicher Gewinn der Arbeit

Es gibt etliche Steering-Systeme die auf einem Cluster oder auf Mehrprozessorsystemen arbeiten, aber Steering-Systeme die an eine inhomogene Gridumgebung angepasst sind existieren bisher nicht. Ziele dieser Arbeit sind:

- ein Online-Steering-Modell zu entwickeln, dass Online-Steering als Zugriffe auf einen verteilten gemeinsamen Speicher betrachtet. Dafür werden die Konsistenzanforderungen für eine Online Steering System analysiert. Dabei ist es möglich, dass innerhalb einer Anwendung die Daten unterschiedliche Konsistenzbedingen haben. Aus der Analyse wurden mehrere Konsistenzmodelle abgeleitet.

- Implementierung und Evaluierung des neuen Modells für sequentielle Anwendungen.

- Entwicklung eines Steering-Systems zum Online-Steering von Gridjobs. Das Steering-System muss die speziellen Bedürfnisse der Gridumgebung berücksichtigen. Dazu gehört, dass die Kommunikation Domain-übegreifend stattfinden, was u.a. bedeutet, dass eine Sicherung der Kommunikation zwischen den Prozessen mit den Grid-typischen Sicherheitsmechanismen implementiert werden muss. Außerdem wird die direkte Kommunikation durch Sicherheitsvorkehrungen wie Firewalls oder private Netzwerke verhindert. Die Sicherheitseinrichtungen einzelner Sites müssen erhalten bleiben und dürfen durch ein Steering-System nicht kompromittiert werden.

- Evaluation der entwickelten Konsistenzmodelle und Protokolle.

Es wird RMOST (Result Monitoring and Online Steering Tool) [112, 151] präsentiert, in welchem das entwickelte Modell implementiert wurde und Online Steering von Grid Jobs ermöglicht. RMOST wurde für das Steering von Gridjobs des Hochenergiephysik (HEP) Experiments ATLAS [172, 173] verwendet. Für das ATLAS Experiment müssen im Grid Datenmengen von mehreren PB verarbeitet werden. An diesem Praxisbeispiel konnte gezeigt werden, dass RMOST effektiv mit riesigen Datenmengen umgehen kann. Eine Ereignisstrom-basierte Lösung würde ein Vielfaches an Netzwerkbandbreite erfordern. Dabei kann die meiste und wichtigste Funktionalität in der ATLAS Software integriert werden, ohne dass Quellcodeänderungen nötig sind.

Anhand der Implementierung wurde das Verhalten das Steering-Systems in verschiedenen Bedingungen untersucht und die wesentlichen Gründe für das Performanceverhalten analysiert. In dieser Arbeit werden neue Erkenntnisse gewonnen

- ob und wie Online-Steering als verteilter gemeinsamer Speicher dargestellt werden kann und wie dieser Ansatz praktisch umsetzbar ist.

- über die Performance der Konsistenzprotokolle für das Steering sequentieller Anwendungen und daraus resultierender Optimierungsmöglichkeiten. Dazu zählt insbesondere die Adaptivität der Invalidierungsprotokolle, die ihre Aktualisierungsfrequenz der verfügbaren Bandbreite anpassen.

- über möglichen Sicherheitsarchitekturen für und Konzepte zum Aufbau einer interaktiven Gridkommunikation sowie Performamceeigenschaften des Kommunikationskanals, der Skalierung und der Performance bei unterschiedlichen Sicherheitseinstellungen, z.B. mit zusätzlicher Ende-zu-Ende Sicherheitsschicht und/oder Verschlüsselung.

- über das Zusammenspiel von Nachrichtengröße, Threadscheduling und TCP-Puffer und deren Auswirkungen auf die Performance des Online-Steering-Systems zum Steering.

1.2 Vorgehensweise und Aufbau

Zunächst wird in Kap. 2 die Umgebung beleuchtet und relevante Aspekte verwendeter Technologien vorgestellt. Anschließend wird der Stand der Forschung (Kap. 3) dargestellt, wobei insbesondere auf die Bereiche Online-Steering, verteilter gemeinsamer Speicher und interaktive Grid Systeme eingegangen wird.

In Kapitel 4 werden die Probleme theoretisch untersucht und Lösungsansätze entwickelt. Um überhaupt mit Gridjobs interagieren zu können, muss eine Möglichkeit zur Kommunikation mit dem Gridjob bestehen. Daher wird in 4.1 zuerst ein Kommunikationskanal entwickelt, der die Interaktion eines Benutzers mit seinem Gridjob erlaubt.

Für die Integration mit einer Anwendung muss das Steering System auf Datenzugriffe reagieren können. Daher werden in Kap. 4.4 Zugriffsmöglichkeiten beschrieben und klassifiziert.

Die Entwicklung des Online-Steering Modells lässt sich in die folgenden Schritte aufteilen:

- Formalisierung das Modellansatzes (siehe Kap. 4.2).
- Entwicklung der benötigten Konsistenzmodelle (siehe Kap. 4.2).
- Entwicklung von Protokollen, die die neuen Konsistenzmodelle umsetzten. (siehe Kap. 4.3).

Der letzte Teil der Analyse und Modellierung (siehe Kap. 4.5) entfällt auf die theoretische Untersuchung der Performanceeigenschaften von Konsistenzprotokollen und der Entwicklung von Algorithmen zur automatisierten Protokollwahl.

Um das entwickelte Modell zu untersuchen, wurde es in RMOST (Result Monitoring and Online Steering Tool) für sequentielle Anwendungen implementiert. Die Implementierung ist in Kapitel 5 beschrieben. Als reale Anwendung für das Online-Steering wurde das Hochenergiephysikexperiment ATLAS [1] verwendet. Kapitel 6 beschreibt die Integration von RMOST in die Experimentsoftware des ATLAS Experimentes. Dadurch wird die einfache Integration von DSM-basiertem Steering demonstriert. Anhand der Implementierung in RMOST wurde die Performance der verschiedenen Protokolle und der Gridverbindung evaluiert (siehe Kapitel 7).

Mit der Entwicklung des neuen Modells für Online Steering wird eine Tür geöffnet zu einer Reihe weiterer Forschungsmöglichkeiten. Bisher ist die Ausarbeitung und Evaluation nur für sequentielle Jobs erfolgt. Außerdem ergeben sich eine Reihe weiterer Aspekte, die untersucht werden können, von denen einige in Kap. 8 beschrieben werden.

Teile dieser Arbeit wurden bereits veröffentlicht in [112, 125, 19, 114, 113, 111, 115, 110, 109].

Kapitel 2

Grundlagen

In diesem Kapitel werden zunächst einige Themengebiete erläutert, auf denen diese Arbeit aufbaut. Das Thema dieser Arbeit, „Modellierung des Online-Steerings von Gridjobs als Zugriffe auf verteilten gemeinsamen Speicher" , verknüpft die Bereiche Online-Steering, verteilter gemeinsamer Speicher und Grid miteinander.

Zunächst wird eine Einführung in das Thema Online-Steering gegeben. Eine ausführliche Darstellung des aktuellen Forschungsstandes über Online-Steering gibt es in Kap. 3.1.

Anschließend wird die angestrebte Umgebung für das zu entwickelnde Online-Steering-System, das Grid, beschrieben (Kap. 2.3). Da es sich beim Grid um ein komplexes System handelt, erwachsen aus der Umgebung spezielle Anforderungen an ein Online-Steering-System. Aus der Beschreibung der relevanten Konzepte und Funktionsweise des Grids leiten sich daher Rahmenbedingungen ab, die bei der Konzeption des Steering-Systems berücksichtigt werden müssen und insbesondere die Kommunikation betreffen.

Online Steering ermöglicht die Interaktion mit einem laufen Job. Da die Kommunikation über das Internet erfolgt, ist sie ein potentielles Ziel für Angriffe auf die übertragenden Daten und Gridressourcen. Um die Kommunikation zu schützen, müssen Sicherheitsvorkehrungen getroffen werden. Außerdem ist eine funktionierende Sicherheitsarchitektur essentiell für jede Gridinfrastruktur. Die Sicherheit des Grids darf durch das Online-Steering nicht gefährdet werden. Daher werden zunächst in Kap. 2.2 Angriffsmöglichkeiten und verbreitete Konzepte zum Schutz vor diesen Angriffen beschrieben. Die Sicherheitsanforderungen finden sich in der anschließenden Beschreibung des Grids (Kap. 2.3) als auch dem Design des Kommunikationskanals (Kap. 4.1) wieder.

Das in dieser Arbeit entwickelte Modell für Online-Steering betrachtet Online-Steering als eine Anwendung von verteiltem gemeinsamen Speicher. In Kap. 3.2 wird die Idee des verteilten gemeinsamen Speichers, grundlegende Konzepte und Begriffe und wesentliche Probleme erläutert. Auf diese Begriffe und Konzepte wird bei der Entwicklung des Steering-Modells wiederholt zurückgegriffen. In Kap. 3.2 folgt eine Darstellung des aktuellen Forschungsstandes des Gebietes des DSM.

2.1 Online Steering

In vielen wissenschaftlichen Bereichen werden heutzutage aufwendige Simulationen oder Berechnungen durchgeführt, oder es müssen riesige Datenmengen verarbeitet werden. Beispiele hierfür sind:

- die Hochenergiephysik, in der für die Auswertung von Messdaten aktueller Experimente, Millionen von Kollisionen rekonstruiert und ausgewertet werden müssen.
- die Klimaforschung, die anhand von Simulationen versucht, die Entwicklung des Klimas vorherzusagen.
- die Berechnung der räumlichen Struktur von Molekülen (molecular dynamics).
- die Simulation der Kinematik von Galaxien in der Astronomie.
- die Berechnung des Strömungsverhaltens von Flüssigkeiten (computational fluid dynamics).

Unabhängig von dem Anwendungsgebiet und der speziellen Berechnung kann der Ablauf einer Simulation typischerweise in folgende Phasen aufgeteilt werden:

- Erstellen der Konfiguration der Berechnung durch den Wissenschaftler
- Durchführen der Berechnung. Aufgrund der Komplexität der Berechnungen lassen sie sich in der Regel nur auf Großrechenanlagen ausführen und benötigen selbst auf diesen Anlagen Stunden oder Tage bis hin zu Monaten für einen Durchlauf. Der Wissenschaftler muss warten bis sein Job beendet ist.
- Die Ergebnisse der Berechnung werden schließend visualisiert und durch den Wissenschaftler interpretiert.

Allerdings kommt es häufig vor, dass der Wissenschaftler bei der Visualisierung feststellt, dass die Konfiguration noch nicht optimal gewesen ist, bzw. dass er eine andere Konfiguration ausprobieren möchte, oder dass die Konfiguration sogar fehlerhaft war und die Resultate nicht das gewünschte Ergebnis beinhalten, inhaltlich fehlerhaft oder nicht aussagekräftig sind. In solchen Fällen, muss der vollständige Arbeitszyklus wiederholt werden. Aufgrund der langen Berechnungszeit impliziert jeder Durchlauf eine signifikante Wartezeit und einen nicht zu vernachlässigenden Kostenfaktor.

In vielen Fällen könnte der Wissenschaftler schon in einem frühen Stadium der Berechnung feststellen, dass die Berechnung nicht zum gewünschten Ergebnis führt, wenn es ihm möglich wäre, die Zwischenergebnisse einzusehen. Dies würde es dem Wissenschaftler erlauben, fehlerhafte Jobs frühzeitig zu beenden, früher mit der Auswertung und Interpretation zu beginnen und evtl. neue Jobs früher zu konfigurieren. Dadurch würden die Produktionszyklen verkürzt und dadurch Zeit und Ressourcen gespart werden.

Wenn neben der Online-Visualisierung auch die Möglichkeit besteht, entsprechende Parameter während des laufenden Jobs zu verändern, könnte ein solcher Job evtl. einfach korrigiert werden. Darüber hinaus kann dann der Effekt

von bestimmten Parametern experimentell, online untersucht werden, anstatt an mehreren unterschiedlichen Jobs. Der Wissenschaftler kann dadurch auch ein intuitives Gespür für die Auswirkungen von Änderungen entwickeln. Eine weitere Anwendungsmöglichkeit von Online-Steering ist, die Lastverteilung bei parallelen Jobs online zu optimieren, um die Laufzeit zu verkürzen.

Bei Berechnungen, die auf dem lokalen Computer ausgeführt werden, kann die Anwendung einfach um eine graphische Komponente ergänzt werden, die Zwischenergebnisse anzeigt und die Möglichkeit zur Modifikation der Parameter bietet. Die Anwendungen, um die es aber in diesem Fall geht, werden aber auf speziellen Großrechnern, oft auf Clustern oder in Grids ausgeführt, was bedingt, dass der Benutzer keinen direkten Zugriff auf die Rechner hat, auf denen die Anwendung ausgeführt wird. Zusätzlich können Teile der Berechnung parallel durchgeführt werden und über mehrere Prozessoren oder, im Falle eines Grids, sogar über unterschiedliche Domains verteilt sein.

Um diese Hindernisse zu überwinden sind Online-Steering-Systeme entwickelt worden, die

- eine interaktiven Kommunikationskanal mit der Anwendung herstellen.
- Zwischenergebnisse aus der Anwendung auslesen und evtl. von verschiedenen Prozessoren zusammentragen um eine Online-Visualisierung zu ermöglichen.
- in eine Anwendung eingreifen um die weitere Ausführung zu beeinflussen (steuern) und z.B. Parameter zu verändern. Die interaktiven Eingriffe werden als Steering-Aktionen bezeichnet.

In dieser Arbeit werden Steering-Systeme und Steuerungswerkzeuge unterschieden. Steuerungswerkzeuge sind Programme, die die Daten auf dem Rechner des Nutzers visualisieren. Steering-Systeme umfassen alle Komponenten die für das Steering notwendig sind. Dazu gehören das Steuerungswerkzeug, spezielle Bibliotheken, die in die Anwendung eingebunden werden, Kommunikationsdienste und weitere Dienste, die für das Online-Steering verwendet werden. Einzelne Systeme, die das Online-Steering für verschiedene Umgebungen und Anwendungen ermöglichen, werden in Kap. 3.1 untersucht.

Eine Simulation über den Lake Erie [119] war die erste Simulation, die Interaktion mit langlaufenden Anwendungen ermöglichte. Im Laufe der Zeit wurden einige weitere Online-Steering-Systeme entwickelt [60, 21, 105, 70, 49, 22, 26, 181, 178, 145, 74, 149]. Andere in der Literatur verwendete Bezeichnungen für Online-Steering sind computational steering oder interactive steering.

Die Arbeitsweise typischer Online-Steering-Systeme [181, 178, 74, 149, 70, 105] ist, dass in der Anwendung sog. Sensoren und Aktoren eingefügt werden, die es erlauben, Daten zu lesen. Die Sensoren generieren bestimmte Ereignisse, z.B. „ein Wert wurde gesetzt". Dieser Ereignisstrom wird dann vom Steering-System verarbeitet und visualisiert. Initiiert der Benutzer eine sog. Steeringaktion, z.B. indem er einen Parameter ändert, wird die Steeringaktion zur Anwendung transportiert und durch einen sog. Aktor umgesetzt. Aufgrund der großen Nachrichtenmenge, die von der Anwendung dabei erzeugt wird, ist ein effektives Filtern überflüssiger Nachrichten wichtig, um eine gute Performance zu erzielen und das Netzwerk nicht schon allein durch das Steering zu überlasten.

Aufbauend auf den Methoden der Ereignisstrom-basierten Steering-Systeme wurden objektorientierte Steering-Systeme geschaffen z.B. MOSS (Mirror Object Steering System) [47, 48]. Hierbei wird zu jedem Datenobjekt der Anwendung ein entsprechendes Spiegelobjekt im Steering-System erzeugt, das in verschiedenen Prozessen repliziert werden kann und den aktuellen Zustand des Objektes repräsentiert. Dies führt zu einer besseren Kontrolle über das Steering selbst. In MOSS wurde noch Ereignisstrom-basiert kommuniziert, aber in [48] wird als interessanter, zukünftiger Forschungsgegenstand bereits ein Steering-Modell vorgeschlagen, das auf verteiltem gemeinsamen Speicher (DSM, distributed shared memory) basiert.

In Kap. 3.1 werden die bestehenden Steering-Systeme detailierter erläutert.

2.2 Sicherheit

Jede Kommunikation über unsichere Netzwerke, insbesondere dem Internet, ist möglicherweise Ziel von Angriffen durch Dritte. Sicherheit ist also eine notwendige Voraussetzung für alle verteilten Systeme (wie das Grid), die das Internet nutzen, auch für das Online-Steering. Schließlich möchte der Benutzer nicht, dass sein Job von jemand anderem gesteuert oder abgebrochen wird oder anderen Personen unautorisierter Zugriff auf irgendwelche Rechner gegeben wird. Eine ausführlichere Behandlung des Themas Sicherheit findet man z.B. in [171].

Sicherheit kann je nach Kontext etwas anderes beinhalten. Für das Online-Steering bedeutet sicher:

- Nur autorisierte Personen können Daten an den Job übermitteln oder Zwischenergebnisse des Jobs empfangen.
- Durch das Steering-System kann keine Person zusätzliche Rechte oder Möglichkeiten auf irgendwelchen Rechnern erhalten, ausgenommen die Möglichkeit von autorisierten Personen mit einem Job zu interagieren.

Daraus folgt:

- Die Authentizität muss sichergestellt werden. Der Absender einer Nachricht muss gegenüber dem Empfänger der Nachricht seine Identität nachweisen.
- Die Integrität der Daten muss geschützt werden. Das bedeutet, dass die Daten nicht während der Übertragung verändert oder ausgetauscht werden können. In der Regel wird die Integrität der Daten beim Empfang überprüft und die Nachricht verworfen, falls der Integritätstest scheitert.
- Die Vertraulichkeit der Daten bedeutet, das keine dritte Person vom Inhalt der Kommunikation Kenntnis erhält. Dies wird in der Regel durch Verschlüsselung realisiert.
- Es muss eine Zugriffskontrolle durchgeführt werden. Jeder Benutzer kann beim Zugriff auf einen Rechner nur die Funktionen nutzen, für die er autorisiert ist. Wenn keine anonymen Nutzer autorisiert sein dürfen, setzt eine Autorisierung immer eine korrekte Authentifizierung voraus.

Verschlüsselung

Zur Sicherung der Kommunikation werden meistens kryptographische Verfahren verwendet. Das grundlegende Prinzip besteht darin, dass der Absender einen *Klartext* M mithilfe eines Schlüssels E verschlüsselt. Der verschlüsselte Text wird *Chiffretext* genannt. Der Chiffretext wird an den Empfänger übertragen, der den Chiffretext mit Hilfe eines zu E passenden Schlüssels D entschlüsseln kann und so wieder den Klartext erhält. Sind D und E identisch spricht man von einem symmetrischen Verschlüsselungsverfahren, sind D und E verschieden und kann D nicht aus E rekonstruiert werden, spricht man von einem asymmetrischen Verfahren.

Bei symmetrischer Verschlüsselung kann jeder, der den Schlüssel besitzt, die Nachrichten mitlesen. Das bedeutet, dass der gemeinsame Schlüssel von Sender und Empfänger geheim sein muss. Symmetrische Verschlüsselungsverfahren sind z.B. DES [175] oder AES [127]. Bei asymmetrischer Verschlüsselung besitzt jeder Teilnehmer T ein Schlüsselpaar (D_T, E_T), wobei D_T geheim ist und E_T veröffentlicht wird. Soll eine Nachricht an T verschickt werden, wird die Nachricht mit dem öffentlichen Schlüssel E_T von T verschlüsselt. Da nur T den dazu passenden Schlüssel D_T besitzt, kann nur T die Nachricht lesen. Ein Verfahren, das asymmetrische Verschlüsselung verwendet, ist RSA [150].

Authentizität

Um sicherzugehen, dass der Absender einer Nachricht wirklich derjenige ist, für den er sich ausgibt, muss die Identität des Absenders überprüft werden können. Ansonsten könnte der Absender eine falsche Identität angeben. Andersherum möchte man z.B. beim Online-Banking auch sicherstellen, dass der Empfänger (die Bank) derjenige ist, der vom Absender als Empfänger vorgesehen ist. Z.B. sollen die Daten wirklich zur Bank übertragen werden und nicht zu Jemand, der sich fälschlicherweise als die Bank ausgibt, um an Kontodaten zu gelangen. Auch für die Authentifizierung werden kryptographische Verfahren verwendet. Eine gegenseitige Authentifizierung wird z.B. mithilfe des Drei-Wege-Handshakes durchgeführt. Beschrieben wird der 3-Wege-Handshake mit asymmetrischen Schlüsseln. Dabei wird vorausgesetzt, dass die öffentlichen Schlüssel jedes Schlüsselpaars sicher einer Identität zugeordnet werden können. Dies kann zum Beispiel mit X.509 Zertifikaten erreicht werden (siehe Kap. 2.2.1).

Zuerst sendet der Client eine Zufallszahl x, die mit dem öffentlichen Schlüssel des Servers verschlüsselt ist, und seine eigene Identität. Der Server kann die Zufallszahl entschlüsseln und sendet $x + 1$ und eine weitere Zufallszahl y, die mit dem öffentlichen Schlüssel des Clients verschlüsselt werden, zum Client. Damit ist der Server authentifiziert. Der Client sendet nun $y + 1$ an den Server zurück und ist damit ebenfalls authentifiziert.

Durch die Verwendung von Zufallszahlen, kann der Verbindungsaufbau nicht von Dritten wiederholt werden, indem sie einfach abgehörte Nachrichten erneut abspielen.

Anschließend wird ein Sitzungsschlüssels (session key) vom Server an der Client geschickt, der für die weitere Kommunikation verwendet wird.

Integrität

Der Integritätstest überprüft, ob eine Nachricht während des Transports verändert wurde. Dafür wird an jede Nachricht ein Nachrichtenintegritätcode (MIC, message integrity code) angehängt. Der MIC besteht aus einem Hashwert über der Nachricht, der mit dem Sitzungsschlüssel verschlüsselt wird. Wird die Nachricht verändert, stimmt der Hashwert nicht mehr. Dritte, die den Sitzungsschlüssel nicht kennen, können auch kein korrekt kodierten MIC Code an eine geänderte Nachricht anhängen. Gleichzeitig wird durch die Verwendung des Sitzungsschlüssels sichergestellt, das die Nachricht auch von dem authentifizierten Absender stammt.

Statt eines symmetrischen Sitzungsschlüssels, kann auch der private Schlüssel eines asymmetrischen Schlüsselpaares verwendet werden, um den Hashwert zu verschlüsseln. Der verschlüsselte Hashwert kann mit dem öffentlichen Schlüssel dekodiert und anschließend überprüft werden. Da niemand anderes den privaten Schlüssel besitzt, kann niemand Anderes einen gültigen MIC für eine veränderte oder falsche Nachricht erzeugen.

2.2.1 X.509 Zertifikate

Jeder Teilnehmer im Grid benötigt ein sog. X.509 Zertifikat um sich zu authentifizieren. Das X.509 Zertifikat enthält den öffentlichen Schlüssel eines asymmetrischen Schlüsselpaares. Bisher wurde immer davon ausgegangen, dass der öffentliche Schlüssel eindeutig einer Person zugeordnet werden kann. Um dies zu gewährleisten wird bei X.509 Zertifikaten die Authentizität des öffentliche Schlüssel von einer Zertifizierungsstelle (certification authority, CA) bestätigt. Die CA überprüft die Identität der Person, die den Schlüssel zertifizieren lässt und fügt einen Hashwert an das Zertifikat an. Der Hashwert wird mit dem privaten Schlüssel der CA verschlüsselt.

Jetzt kann jeder die Authentizität des zertifizierten, öffentlichen Schlüssels mit dem Zertifikats der CA überprüfen. Genauso kann eine CA ihr Zertifikat von einer übergeordneten Zertifizierungsstelle mit deren Zertifikat signieren lassen. Ein Wurzelzertifikat ist das Zertifikat, das von der CA an der Spitze der so entstehenden Hierarchie verwendet wird. Diese Struktur wird PKI (public key infrastructure) genannt. Der CA müssen dabei alle Beteiligten vertrauen.

Zur Authentifizierung braucht jeder Server somit nur das Wurzelzertifikate der CA zu kennen, um jedes Zertifikat zu überprüfen oder zu finden, das von dieser CA zertifiziert wurde. Ein Dienst, der Authentifizierung verlangt, braucht aber nicht die öffentlichen Schlüssel aller (häufig wechselnden) Benutzer zu kennen.

Ein Zertifikat ist kompromittiert, wenn der zugehörige private Schlüssel anderen Personen als dem Besitzer des Zertifikats bekannt geworden ist. Der Besitzer kann ein kompromittiertes Zertifikat zurückziehen. Die CA stellt aktuelle Listen mit zurückgezogenen Zertifikaten, sog. Certificate Revokation Lists (CRLs), zur Verfügung. Wenn während der Authentifizierung festgestellt wird, dass ein Zertifikat in der CRL vorkommt, wird es abgelehnt. Dadurch kann eine zeitnahe Sperrung kompromittierter Zertifikate erreicht werden. Zusätzlich haben Zertifikate in der Regel eine beschränkte Gültigkeitsdauer, so dass jeder Benutzer sein Zertifikat regelmäßig erneuern muss. Dadurch wird der Schaden durch unbemerkt kompromittierte und nicht zurückgezogene Zertifikate limitiert

und die CRLs wachsen nicht unbegrenzt.

2.2.2 GSSAPI

Im Laufe der Zeit wurden verschiedene Sicherheitssysteme implementiert, die Verschlüsselung, Integrität, Authentifizierung, etc. anbieten, wie SSL und Kerberos [128]. Jede dieser Systeme hat eine andere Programmierschnittstelle (API) und unterschiedliche Datenstrukturen. Wenn eine Anwendung verschiedene Sicherheitssysteme nutzen will, die unterschiedliche APIs verwenden, muss oft ein signifikanter Teil der Anwendung neu geschrieben werden. Dies kann z.B. notwendig werden, wenn die Anwendung in eine neue Umgebung migriert werden soll, in der ein anderes Sicherheitssystem verwendet wird. Um die Verwendung unterschiedlicher Sicherheitssysteme zu vereinfachen, werden typischerweise Standardschnittstellen definiert, so dass die Implementierung nicht verändert werden muss, sondern einfach eine andere Implementierung mit der Anwendung verlinkt wird.

GSSAPI [68, 170] ist die Abkürzung für Generic Security System Application Programming Interface. Sie bietet eine implementierungsunabhängige Schnittstelle zur Verwendung verschiedener Sicherheitssysteme wie SSL, Kerberos und der Globus Sicherheitsprotokolle. Bei der Schnittstelle wurden folgende Ideen umgesetzt:

- Die Schnittstelle soll unabhängig von dem verwendeten Transportkanal sein. Die GSSAPI Funktionen kommunizieren daher nicht selbst, sondern generieren Nachrichten (sog. Token) die an das aufrufende Programm zurückgegeben werden. Das Anwendungsprogramm ist dann dafür verantwortlich diese Token zu versenden. Das Anwendungsprogramm ist auch dafür zuständig, die Token wieder zu empfangen und der GSSAPI zu übergeben, die entweder ein weiteres Token zur Rücksendung oder die Daten im Klartext zurückgibt.

- Da die Sicherheitsmechanismen durch unterschiedliche Konzepte und Verfahren umgesetzt werden können, sind alle verwendeten Objekte *opaque*. Die Struktur der Objekte kann außerhalb der GSSAPI Implementierung nicht aufgelöst werden.

Um eine Verbindung aufzubauen und sich gegenüber einem anderen Rechner authentifizieren zu können, muss die GSSAPI überhaupt erst einmal die Dokumente haben, mit denen sie sich ausweisen soll. Dies kann z.B. ein X509 Zertifikat sein. Diese Objekte werden mit Credentials bezeichnet.

Mit dem initialisierten Credential kann dann für eine Kommunikation ein sog. Sicherheitskontext aufgebaut werden. Im Sicherheitskontext werden alle für diese Kommunikation relevanten Daten gespeichert, z.B.

- die verwendeten Verschlüsselungsprotokolle und Hashfunktionen;

- welche Sicherheitsstandards eingehalten werden, bzw. ob/wie der Integritätstest, die Verschlüsselung oder die Authentifizierung durchgeführt wird;

- evtl. Sitzungsschlüssel;

- momentaner Status der Authentifizierung und z.B. versendete Zufallszahl beim 3-Wege Handshake.

Um den Sicherheitskontext aufzubauen, werden solange Tokens zwischen den Prozessen ausgetauscht, bis die Sicherheitstest erfolgreich beendet wurden oder endgültig gescheitert sind.

Wurde ein Sicherheitskontext etabliert, kann über die entsprechende Verbindung mit den ausgehandelten Sicherheitseigenschaften kommuniziert werden. Dafür werden die zu übertragenden Daten der GSSAPI-Implementierung übergeben. Diese fügt die notwendigen Informationen, z.B. den MIC, hinzu und verschlüsselt die Daten gegebenenfalls. Die fertig verpackte Nachricht wird zurückgegeben und kann verschickt werden. Entsprechend wird die Nachricht beim Empfänger von der GSSAPI-Implementierung ausgepackt und die benötigten Überprüfungen, z.B. der Integritätstest, durchgeführt. Waren die Überprüfungen erfolgreich, wird die Nachricht im Klartext zurückgegeben.

2.3 Grids

In letzter Zeit ist der Begriff 'Grid' zu einem populären Schlagwort geworden. Grid ist zunächst ein Konzept [56, 53], wird aber auch für eine konkrete Infrastruktur, die das abstrakte Konzept implementiert, verwendet, wie z.B. dem LCG [96]. Dabei ist die Verwendung des Begriffes Grid sehr breit auf eine Reihe von Systemen angewendet worden.

Die Idee des Grids besteht darin, dass Ressourcen von unterschiedlichen Betreiben von einer Gruppe Anwendern gemeinsam genutzt werden. Für das Prinzip des Grids ist es zunächst unerheblich, welcher Art die Ressourcen sind. Verbreitete Ressourcentypen in Grids sind z.B. Rechenleistung (CPUs) oder Speicherplatz. Es können aber auch beliebige andere Ressourcen sein, wie Mikroskope, Teleskope oder spezielle Dienste. Dabei bleiben die Ressourcenbetreiber in der Verwaltung und dem Betrieb ihrer Ressourcen unabhängig voneinander.

Die Ressourcen stehen allen Mitgliedern einer Nutzungsgemeinschaft zur Verfügung. Die Regeln und Bedingungen, zu denen ein Nutzer auf eine Ressource zugreifen kann, werden von einer sogenannten *Virtuellen Organisation* (VO) definiert. Zur Verwendung eines Grids wird eine Middleware benötigt, die einem Nutzer zu den von der VO vorgegebenen Regeln den Zugriff auf eine Menge an Ressourcen ermöglicht, die die jeweilige VO unterstützen. Daraus folgt, dass zu jeder VO eine Infrastruktur gehört, so dass manche auch eine VO als ein Grid bezeichnen [104].

Ein entscheidender Punkt dabei ist, dass ein Mitglied einer VO nicht seinen persönlichen Zugang für jede einzelne Ressource des Grid beantragen muss, sondern sobald er als Mitglied der VO akzeptiert ist, automatisch Zugriff zu allen Ressourcen der VO erhält. In größeren VOs wäre es ein gewaltiger Organisationsaufwand, für jeden Nutzer auf jeder Ressource die persönlichen Zugangsberechtigungen zu verwalten. Stattdessen muss der Nutzer seine Identität und seine Zugehörigkeit und evtl. seine Funktion in einer VO verifizieren, um Zugriff zu erhalten.

Die Middleware stellt die Verbindung zwischen dem Benutzern und den Ressourcen her. Dazu gehört z.B., dass sie eine passende (freie) Ressource für den Nutzer auswählt, Zustandsinformationen über die verfügbaren Ressourcen sam-

melt, Dienste zur Überprüfung der Identität (Authentifizierung) und Zugriffsrechte (Autorisierung) eines Nutzers bereitstellt und eine einheitliche Schnittstelle besitzt, über die Aktionen der Ressource initiiert werden können. Eine Gridinfrastruktur besteht also aus verschiedenen Diensten und Protokollen, die die Kommunikation zwischen den Diensten ermöglichen. Die Gridarchitektur ist die Spezifikation der Schnittstellen und der Protokolle einer Infrastruktur, und die Middleware ist eine Implementierung einer Gridarchitektur. Existierende Middlewares sind beispielsweise Globus [51], gLite [64], Unicore [169], Legion [67], ICENI [57], MILAN [9] und WEDS [39]. Die Open Grid Service Architecture (OGSA) [50] ist eine Webservice basierte Gridarchitektur, die z.B. von Globus implementiert wird.

Im folgenden wird zunächst der Ausführungszyklus von Rechenaufträgen mit gLite und die dazu benötigten Dienste näher erklärt. Die Rechenaufträge werden Job oder Gridjob genannt. Der Ausführungszyklus und die dazugehörigen Dienste bilden eine wichtige Rahmenbedingung für das Online-Steering von Gridjobs. Anschließend wird auf die Autorisierung und Authentifizierung im Grid eingegangen, da die Sicherheitskonzepte der Kommunikation für das Online-Steering in die Grid-Sicherheitsarchitektur eingebunden werden müssen.

2.3.1 Ausführung eines Gridjobs in gLite

Der Zugang zum Grid erfolgt für einen Benutzer immer über eine Benutzerschnittstelle (user interface, abgekürzt UI). Damit wird ein Rechner bezeichnet, welcher die Programme und Bibliotheken installiert hat, die der Benutzer benötigt, um mit dem Grid zu interagieren.

Um einen Job unter gLite zu submittieren, muss zunächst eine Jobbeschreibungsdatei erstellt werden. Diese Datei enthält u.a. die Namen des zu startenden Programms oder Skriptes, eventuelle Parameter, Dateien, die mit dem Job mittransferiert werden müssen und die Dateinamen für die Dateien, in die die Standardausgabe umgeleitet wird. Von dem UI aus kann dieser Job ins Grid submittiert werden. Das geschieht mit Hilfe eines Submissionsprogramms `glite-wms-job-submit`, dem die Jobbeschreibungsdatei übergeben wird. Bei erfolgreicher Submission erhält der Benutzer einen Jobidentifier, der aus einer Zeichenfolge besteht, mit der der Job eindeutig spezifiziert werden kann.

Das Submissionsprogramm kontaktiert einen Ressource Broker (RB) dem es den Job übergibt. Der RB ist ein anderer Grid-Dienst, der in der Regel auf einem anderen Rechner läuft. Der RB sucht eine passendes Computing Element aus und leitet den Job an diese weiter. Jedes Computing Element muss Informationen zur momentanen Belegung, Kapazität, installierter Software, usw. durch Informations- und Monitoringdienste bereitstellen. Diese Dienste werden von RB verwendet um eine Entscheidung zu treffen.

Ein Computing Element (CE) ist meistens der Front-end Rechner einer Batchfarm, und führt den Job selbst nicht aus, sondern fügt ihn in eine Warteschlange ein, die von den Worker Nodes (WN), die zu der Batchfarm gehören, abgearbeitet wird.

Von seinem UI aus kann der Benutzer den Status seines Jobs mit Hilfe eines entsprechenden Programms überprüfen, der den Logging und Bookkeeping (LB) Dienst befragt. Der LB ist normalerweise mit dem Ressource Broker (RB) kombiniert und verfolgt die Ausführung der Jobs. Wurde der Job erfolgreich beendet, werden die Ausgabedateien gespeichert und können auf des UI geladen

werden.

2.3.2 Authentifizierung und Autorisierung

Bei der Benutzung von Grids kann eine einfache Operation bereits die Interaktion mit mehreren Rechnern beinhalten, die zu unterschiedlichen Verwaltungseinheiten gehören. Allein in dem in Abschnitt 2.3.1 beschriebenen Fall muss der Benutzer auf das UI, einen RB, ein CE und ein WN zugreifen. Außerdem findet der Zugriff auf manche Dienste zeitlich verzögert statt, z.B. wenn ein Job eine Weile warten muss, bevor er einem WN zugewiesen wird. Aber natürlich möchte der Benutzer nicht ständig ein Passwort eingeben, um sich zu authentifizieren, aber auch nicht die ganze Zeit bis zur Beendigung seines Jobs online sein, sondern sich lediglich einmal zu Beginn authentifizieren. Andere Dienste werden dann in seinem Auftrag aktiv. So kontaktiert zum Beispiel der RB in seinem Auftrag das CE. Zwei Paradigmen zum Design der Authentifizierung und Autorisierung sind daher einmaliges Anmelden (single sign on) und Delegation (delegation) [54], so dass ein Dienst im Namen des Nutzers aktiv werden kann.

Für die Authentifizierung und Autorisierung wurde die Grid-Sicherheits-Infrastruktur (grid security infrastructure) [54, 104], abgekürzt GSI, entwickelt, die auf X509 Zertifikaten basiert. Jeder Benutzer benötigt ein Zertifikat einer Zertifizierungsstelle, der alle Beteiligten vertrauen müssen. Bevor der Benutzer auf das Grid zugreift, erstellt er ein kurzlebiges Zertifikat, das Proxy (-zertifikat) genannt wird. Das Proxy wird durch das Zertifikat des Nutzers zertifiziert.

Gegenüber anderen Rechnern authentifiziert sich der Benutzer durch sein Proxy. Falls ein anderer Dienst im Auftrag des Benutzers handeln soll, wird das Proxy zu dem entsprechenden Dienst transportiert. Nun verfügt der Dienst über das Proxy des Benutzers und kann sich somit anderen Diensten gegenüber als der Benutzer ausgeben. Bei der Delegation muss auch der private Schlüssel des Proxies übertragen werden und darf nicht durch ein Passwort gesichert sein, bzw dass Passwort muss auch übertragen werden. Somit könnte das Proxyzertifikat in die Hände Dritter gelangen. Durch das kurzlebige Proxyzertifikat soll die Gefährdung durch ein kompromittiertes Proxyzertifikat gesenkt werden.

Anhand der verifizierten Identität, kann ein Benutzer zur Ausführung bestimmter Aktionen auf Gridressourcen autorisiert werden. Die Autorisierung kann auf der Grundlage der Mitgliedschaft in einer VO, der besonderen Rolle in einer VO oder individuell durchgeführt werden. Zur Durchführung können Listen der Mitglieder einer VO installiert sein, die regelmäßig aktualisiert werden, oder es kann ein besonderer Dienst (z.B. VO membership service, VOMS) befragt werden.

2.4 Verteilter gemeinsamer Speicher

In diesem Kapitel wird eine Einführung in den verteilten gemeinsamen Speicher (DSM, distributed shared memory) gegeben. Im Kapitel 3.2 wird der Stand der Forschung im Bereich der DSM detailierter beschrieben.

Verteilter gemeinsamer Speicher ist eine Abstraktion, die auf dem Modell des gemeinsamen Speichers beruht, wie er für Mehrprozessorsysteme existiert. Bei Mehrprozessorsystemen gibt es zwei Möglichkeiten den Speicher anzuordnen:

- Jeder Prozessor besitzt seinen eigenen Speicher, auf den nur er zugreifen kann. Die Synchronisation und der Datenaustausch zwischen den verschiedenen Prozessoren findet über Nachrichten statt, die die Prozessoren untereinander austauschen.

- Es existiert ein gemeinsamer Speicher auf den alle Prozesse zugreifen können. Die Kommunikation zwischen den verschiedenen Prozessoren wird durch den gemeinsam genutzten Speicher hergestellt, indem ein Prozessor in eine Speicherzelle etwas schreibt, was der andere anschließend liest.

Besitzt ein System keinen physikalischen gemeinsamen Speicher, sondern jeder Prozessor seinen eigenen lokalen Speicher, kann mittels Software die Illusion eines gemeinsamen Speichers erzeugt werden. Dafür stellen die verschiedene Prozessoren einen Teil ihres lokalen Speichers zur Verfügung, um Daten des virtuellen gemeinsamen Speichers abzulegen. Greift ein Prozessor auf eine Speicherstelle zu, muss er zuerst feststellen, wo dieser Wert gespeichert ist, und gegebenenfalls den Wert von einem anderen Prozessor anfordern. Dies dauert aber deutlich länger als ein einfacher lokaler Speicherzugriff. Da in diesem Fall der gemeinsam genutzte Speicher auf die lokalen Speicher der verschiedenen Prozessoren verteilt ist, wird das System verteilter gemeinsamer Speicher (distributed shared memory, DSM) genannt.

Da bei DSM der gemeinsame Speicher nur virtuell von einer Software modelliert wird, müssen es auch nicht zwangsläufig physikalisch unterschiedliche Prozessoren sein, die auf den DSM zugreifen und einen eigenen physikalischen Speicher besitzen. Vielmehr abstrahiert man in der Regel auch von den physikalischen Komponenten und spricht von unterschiedlichen Prozessen, die ja einen eigenen virtuellen Adressraum besitzen. Natürlich kann sich hinter jedem Prozess ein eigener Prozessor verbergen.

Da der Zugriff auf Daten, die im lokalen Speicher eines anderen Prozesses liegen, relativ zeitaufwendig ist, besitzt bei den meisten DSM-Systemen jeder Prozess eine Kopie der Daten des gemeinsamen Speichers in seinem lokalen Speicher. Dadurch können die Datenzugriffe auf den DSM deutlich beschleunigt werden. Allerdings kann dadurch die Konsistenz der Daten im DSM verletzt werden:

Angenommen Prozess p_1 überschreibt die Speicherstelle x. Jetzt haben alle anderen Prozesse einen ungültigen Wert für x in ihrer lokalen Kopie des DSM. Eine Leseoperation von x eines anderen Prozesses würde einen ungültigen Wert zurückgeben. Bei einer Schreiboperation müssen also alle anderen Prozesse benachrichtigt werden, dass sich der Wert einer Speicherstelle geändert hat.

Um die lokalen Kopien des DSM konsistent zu halten, gibt es zwei grundlegende Strategien:

- Invalidierungsstrategie: Bei einer Schreiboperation wird in allen anderen Prozessen der gespeicherte Wert dieser Speicherstelle als ungültig markiert. Bei der nächsten Leseoperation muss der aktuelle Wert von dem Prozess bezogen werden, der diese Speicherstelle zuletzt überschrieben hat.

- Aktualisierungsstrategie: Bei einer Schreiboperation wird der neue Wert in allen Kopien des DSM anderer Prozesse geändert.

Die Aufgabe eines DSM-Systems besteht also darin, die Illusion eines gemeinsamen Speichers zu erzeugen, obwohl dieser physikalisch nicht existiert.

Speicherzugriffe auf den DSM müssen vom DSM-System abgefangen und auf physikalische Speicherzellen umgelenkt werden, die den entsprechenden Wert enthalten. Außerdem muss das DSM-System dafür Sorge tragen, dass die verschiedenen Kopien des virtuellen gemeinsamen Speichers konsistent bleiben. Konsistent bedeutet dabei, dass es eine sequentielle Ordnung aller Speicherzugriffe gibt, insbesondere ist die Reihenfolge der Speicherzugriffe aus Sicht aller Prozessoren dieselbe. Eine Leseoperation gibt immer den Wert zurück, der bei der letzten Schreiboperation gesetzt wurde. Dieses Konsistenzmodell wird auch sequentielle Konsistenz genannt [103].

Allerdings hat sich herausgestellt, dass die sequentielle Konsistenz im Vergleich mit nachrichtenbasierten parallelen Programmen, zu einer schlechteren Performance führt. Daher wurden zunehmend abgeschwächte Konsistenzmodelle entwickelt und implementiert. Diese Konsistenzmodelle definieren einen Vertrag zwischen der Anwendung und dem Speichersystem, in der der Speicher konsistent erscheint, falls der Anwendungsprogrammierer bestimmte Richtlinien befolgt [2], z.B. genügend Synchronisationsoperationen einfügt. Es werden also in beschränken Maße Inkonsistenzen erlaubt. Wenn sich das Programm aber an die vereinbarten Regeln hält, verhält es sich als ob sequentiell konsistenter Speicher verwendet würde. In Kap. 3.2 werden verschiedene Konsistenzmodelle beschrieben.

Kapitel 3

Verwandte Arbeiten

In dieser Arbeit wird ein neues Modell für Online-Steering entwickelt, das auf verteiltem gemeinsamen Speicher basiert. Bisher gibt er keine Arbeiten, die diese zwei Themen verbinden, allerdings existieren in beiden Gebieten bereits zahlreiche Vorarbeiten. Zuerst werden andere Ansätze und Methoden des Online-Steerings und danach Arbeiten über verteiltem gemeinsamen Speicher vorgestellt.

Das Steering-System soll in einer Grid Umgebung verwendet werden. Bisher gibt es zwei andere Projekte [21, 88], die sich ebenfalls mit Steering im Grid beschäftigen. Die wesentliche Herausforderung bestehende Steering-Systeme im Grid zu verwenden, besteht darin, einen interaktive Kommunikation mit dem Gridjob zu ermöglichen. In Kapitel 3.3 werden daher neben Steering-Systemen im Grid auch andere interaktive Anwendungen beschrieben. Dabei wird besonders auf den Kommunikationskanal eingegangen.

Schließlich werden in Kap. 3.4 noch einige Arbeiten erwähnt, die einen Bezug zu einzelnen Punkten in dieser Arbeit haben, wie Monitoringsysteme, Netzwerkdateisysteme und die Identifikation von Zugriffmustern und die Optimierung von Datenzugriffen basierend auf dem Zugriffsmuster.

3.1 Steering Werkzeuge und Steering Modelle

Die Anfänge des Online-Steering haben sich aus der Visualisierung von Ergebnissen langlaufender, wissenschaftlicher Simulationen entwickelt. Die Visualisierung wurde nach dem vollständigen Abschluss der Simulation durchgeführt. Interaktion war nicht möglich. 1988 bemerkte Brooks in [24], dass der Benutzer gerne mit seiner Simulation zur Laufzeit interagieren würde. Für eine Simulation über den Eriesee [119] wurde eine Software entwickelt, die als erstes, wichtiges Steuerungswerkzeug gilt. Seit dieser Zeit wurden unzählige Steering-Systeme entwickelt [60, 134, 21, 88, 26, 47, 70, 178, 181, 149, 89, 49, 105]. Im folgenden werden die wichtigsten Modelle und Entwicklungen beschrieben. Anschließend wird eine kurze Übersicht über verschiedene Steering-Systeme gegeben und die Steering-Systeme den entsprechenden Konzepten zugeordnet.

3.1.1 Steering-Modelle

In diesem Abschnitt werden verschiedene Steering-Modelle beschrieben. Dazu zählen das Datenflussmodell (data flow model) [73], das Kontrollflussmodell (control flow model) [26], das Feedback-System-Modell [179] und das Stellvertreterobjekt-Modell (mirror object model) [47]. Andere Arbeiten, die eine Übersicht über Steering-Systeme enthalten, sind [123, 71].

Das Datenflussmodell für Online-Steering

Das Datenflussmodell [73] beschreibt den Arbeitsablauf von der Erstellung der Rohdaten bis zu Visualisierung als eine Folge von Schritten, die Transformationen auf den Daten durchführen. In der Praxis existierten für jeden Schritt eigene Werkzeuge, wodurch die Komplexität der Bedienung stark erhöht wurde. Die Steering-Systeme, die auf dem Datenflussmodell beruhen wie SCIRun [134, 185, 135, 122, 136], gViz [21, 20, 187] und Uintah [168], bieten eine so genannte Problem-Lösungs-Umgebung (problem solving environment, PSE), in der alle Schritte von einer gemeinsamen Oberfläche aus ausgeführt werden können. Jeder Schritt wird durch ein Modul repräsentiert. Zu Beginn der Simulation erstellt der Benutzer ein Datenfluss-Netzwerk, indem er die gewünschten Module auswählt. Jedes Modul kann Eingabe- und Ausgabeschnittstellen besitzen, die durch Datenleitungen verbunden werden. Die Datenleitungen spezifizieren, welche Eingabeschnittstellen mit welchen Ausgabeschnittstellen verbunden werden.

Sobald bei einem Modul an allen Eingängen Daten anliegen, wird das Modul ausgeführt und die Ausgabe berechnet. Die Ergebnisse werden an die nächsten Module weiter gereicht. Wird eine Konfiguration verändert, werden alle betroffenen Module neu ausgeführt. Ein häufig verwendetes Bild für Problem-Lösungs-Umgebungen ist das einer Werkbank, an der der Benutzer experimentiert.

Um existierende Anwendungen in eine solche Problem-Lösungs-Umgebung zu integrieren, muss sie in Module aufgeteilt werden, die der Modulspezifikation der entsprechenden PSE genügen. In der Praxis bedeutet dies in der Regel, dass die Anwendung komplett neu geschrieben werden muss. Interaktion ist nur auf der Ebene von ganzen Modulen möglich. Man kann die Ergebnisse bereits fertig ausgeführter Module ansehen, oder die Eingabe und Konfiguration von Modulen ändern, und das Modul erneut ausführen. Prinzipiell wird davon ausgegangen, dass während der Simulation der Benutzer durchgängig mit der Simulation verbunden ist. PSEs sind nicht für langlaufende, autonome Anwendungen entworfen worden, die echtzeit-fähiges, detailliertes Steering auf entfernten Rechnern unterstützen.

Online-Steering mit dem Client/Server Modell

Das Client/Server Modell ist zunächst ein Kommunikationsmodell. Anstatt Simulation und Visualisierung in einer Umgebung einzubetten, verfolgen viele Steering-Systeme wie z.B. die Steering API des RealityGrid Projekts [88, 138, 23, 141], OViD [145], MoSt [63] und CUMULVS [60, 97, 133] den Ansatz, die Visualisierung und die Simulation zu verbinden. In der Regel wird dabei die Simulation als Server betrachtet, die die zu visualisierenden Daten enthält. Der Benutzer kann mit dem Steering-Client auf die Simulation zugreifen, Daten visualisieren und auch steuernd eingreifen. In anderen Steering-Systemen, z.B. im Computational Steering Environment (CSE) [105, 106, 124] und VIPER [144]

ist die Simulation und die Visualisierung ein Client. Als Server fungiert eine zentrale Komponente des Steering-Systems.

Steering-Systeme, die nach dem Client/Server Modell arbeiten, bieten meistens eine API an, mit der die Anwendung instrumentiert wird, um Zugriff auf die Anwendung zu erhalten.

Das Kontrollflussmodell für Online-Steering

Das Kontrollflussmodell (control flow model) für Online-Steering wurde im VASE Projekt [26] entwickelt, um eine engere Verzahnung von Online-Steering mit der Anwendung zu erreichen als im Datenflussmodell. Die Aufteilung einer Anwendung in komplexe Module wurde als zu grob bewertet. Im Kontrollflussmodell wird eine abstrakte Kontrollstruktur des Programms erstellt, welche den Kontrollfluss in der Anwendung beschreiben soll. Die Anwendung wird mit Markierungen instrumentiert, die eine Zuordnung der Kontrollstrukturbeschreibung zu definierten Stellen im Programmcode ermöglicht. Die markierten Programmpunkte sind außerdem die Einsprungstellen für die Steering-Operationen. Von der abstrakten Strukturbeschreibung können Eingriffsmöglichkeiten abgeleitet werden, z.B. in welchen Bereichen welche Variablen geändert oder gelesen werden können, ohne die Konsistenz der Anwendung zu gefährden.

Allerdings ist der Aufwand, um eine Anwendung auf Steering vorzubereiten, sehr hoch. Es muss manuell eine detaillierte Strukturbeschreibung erstellt werden, und es müssen Markierungen innerhalb der Anwendung gesetzt werden. Außerdem müssen die Zugriffsregeln für jeden Bereich definiert werden. Teilweise kann diese Aufgabe automatisiert ausgeführt werden.

Bisher wurde das Kontrollflussmodell in VASE [26] und EPSN [49, 38] implementiert. In VASE wird das Kontrollflussmodell als Erweiterung des Datenflussmodells verstanden. In EPSN wird das Kontrollflussmodell mit dem Client/Server Modell kombiniert.

Das Ereignisstrommodell

Das Ereignisstrommodell (event stream model) lehnt sich an ähnlichen Modellen für das Debuggen an, wie sie beispielsweise in Dynascope [162, 163] implementiert und beschrieben sind. Es bietet eine abstrakte Sicht auf Online-Visualisierung und Steering und unterstützt sehr detailliertes Steering. Die Simulation wird als Quelle für atomare Ereignisse betrachtet, die vom Steering-System verarbeitet werden. So wird zum Beispiel jedesmal, wenn die Simulation einen Wert ändert, ein Ereignis generiert. Die Ereignisse durchlaufen eine Kette von verarbeitenden Komponenten, die auf ein Ereignis reagieren, indem sie eine Operation ausführen. Zum Beispiel werden noch auf dem Knoten, auf dem die Simulation läuft, möglichst viele Ereignisse ausgefiltert. Oder es werden auf einer anderen Komponente Ergebnisse zusammengesetzt oder Bedingungen überwacht, die bestimmte weitere Ereignisse auslösen. Schließlich erreichen die Ereignisse die Visualisierung, die eine Visualisierungsoperation auf den Ereignissen ausführt.

Bei der Interaktion werden wiederum Ereignisse vom Benutzer oder einem Steuerungsalgorithmus ausgelöst, die dann in der Simulation Aktionen auslösen. Das Ereignisstrom-Modell geht implizit von einem Client/Server-Kommunikationsmodell aus.

Ereignisstrom-Modelle erzeugen typischerweise einen hohen Netzwerkverkehr, erlauben aber eine enge Verzahnung zwischen Simulation und Benutzerschnittstelle. Des Ereignisstrom-Modell ist z.B. in Progress [178], Falcon [70, 69], Magellan [181, 182, 180] und Autopilot [149, 177] implementiert.

Das Rückkopplungsmodell

Vetter und Schwan interpretieren Ereignisstrom-basiertes Steering als ein Rückkopplungsmodell (feedback model) [179]. Die Simulation erzeugt einen Strom von Ereignissen. Das Steering-System oder der Benutzer reagiert auf die Ereignisse der Simulation mit Steering-Aktionen. Die veränderten Randbedingungen der Simulation drücken sich anschließend in dem Ereignisstrom der Simulation aus.

Das Stellvertreterobjekt-Modell

Das Stellvertreterobjekt-Modell (mirror object model) [47, 48] versucht das objektorientierte Datenmodell in Online-Steering zu integrieren. Für jedes Datenobjekt, das für das Steering-System sichtbar ist, wird ein Stellvertreterobjekt erzeugt, das dieses Objekt der lokalen Komponente des Steering-Systems gegenüber repräsentiert. Das Stellvertreterobjekt kann in verschiedenen Komponenten des Steering-Systems beliebig repliziert werden. Der Zugriff auf das entfernte Datenobjekt geschieht über den Zugriff auf das lokale Stellvertreterobjekt, wobei das Steering-System die Änderungen des Zustands eines Stellvertreterobjektes an alle Duplikate und an das Datenobjekt weiterleitet.

In [48] wird vorgeschlagen, ein Modell für Online-Steering, das auf verteiltem gemeinsamen Speicher basiert, zu entwickeln.

Das Stellvertreterobjekt Modell wurde in MOSS (Mirror Object Steering System) [47, 48] implementiert. Es erlaubt eine detaillierte Konfiguration auf Objekt-Basis. Anstatt Ereignisse an alle Komponenten zu versenden, werden Ereignisse gezielt an die Duplikate gesendet. Dadurch kann Netzwerkbandbreite gespart, und somit die Effizient gesteigert werden.

Ein anderes Projekt mit objektorientiertem Ansatz ist DIOS (Distributed Interactive Object Substrate) [126].

Agenten- und Transaktionsbasiertes Steering

Das Steering-System Pathfinder [74, 121, 100] implementiert ein Agenten- und transaktionsbasiertes Modell. Dabei werden alle Monitoring- und Steering-Aktionen von mobilen Agenten durchgeführt. Auch Ereignisse und Zustände von Prozessen werden von speziellen Agenten repräsentiert.

Die Anwendung wird als verteilte Datenbank betrachtet, die die Zustände der verschiedenen Prozesse speichert. Zustandsänderungen eines Prozesses werden als atomare Transaktionen gesehen, insbesondere sind Steering-Aktionen ebenfalls Transaktionen. Durch die Verwendung atomarer Transaktionen wird sequentielle Datenkonsistenz zugesichert. Pathfinder ist das einzige Steering-System, das das Problem der Datenkonsistenz direkt adressiert. Es wird allerdings kein allgemeines Konsistenzmodell für Online-Steering präsentiert. Es beschränkt sich auf die Beschreibung eines Mechanismus, der sequentielle Konsistenz garantiert und eines Algorithmus, der die Konsistenz der Daten überprüfen kann.

3.1.2 Steering-Systeme

In diesem Abschnitt werden bestehende Steering-Systeme kategorisiert (siehe Tab. 3.1). Dabei werden folgende Punkte untersucht:

- Zugrundeliegende Modelle: Welche der in Kap. 3.1.1 beschriebenen Modelle werden implementiert?
- Umgebung: Gibt die Art von Rechnern und Programmen an, für die dieses Steering-System konzipiert worden ist.
- Kommunikation: Die gesteuerte Anwendung und die Benutzerschnittstelle müssen Informationen austauschen. Benennt den verwendeten Kommunikationskanal oder das Kommunikationssystem.
- Besonderheiten: Beschreibung besonderer Merkmale oder Eigenschaften.
- Anwendungen: Zählt exemplarisch Bereiche auf, in denen das Steering-System verwendet worden ist, z.B. Hochenergiephysik (HEP), Molekulardynamik (MD), Dynamik von Flüssigkeiten (computational fluid dynamics), etc. Diese Liste erhebt keinen Anspruch auf Vollständigkeit.
- Visualisierung: Nennt Programme oder Visualisierungssysteme, die als Benutzerschnittstelle verwendet worden sind. Manche Steering-Systeme bieten ein spezielles Werkzeug für ihr System an. Datenflussbasierte Systeme verwenden meist Standardwerkzeuge zur Visualisierung wie AVS [174], VTK [159] oder den IRIS Explorer [183]. Cactus [4] und Discover [118] können über einen beliebigen Browser gesteuert werden.

Name	Zugrundeliegende Modelle	Umgebung	Kommunikation	Besonderheiten	Anwendungen	Visualisierung
CUMULVS [60, 133, 97]	Client/Server	Parallele Anwendungen	PVM [59]	Unterstützung verteilter Datenstrukturen, Checkpointing	Seismologie, Dynamik von Flüssigkeiten	AVS, VTK
SCIRun [134, 185, 135, 122, 136]	Datenflussmodell	Einzelrechner, SMP	gemeinsamer Speicher		MD, Bioelektrik, Medizin, Energie, Weitere	AVS
VASE [26]	Kontrollflussmodell, Datenflussmodell		TCP/IP	Computergestützte Instrumentierung	Mechanik	Spezielle Werzeuge
Falcon [70, 69]	Ereignisstrom-Modell	Parallelrechner	gemeinsamer Speicher, PVM [59], Sockets		MD	
Progress [178]	Ereignisstrom-Modell		Sockets		Astronomie	Spezielles Werkzeug
Magellan [181, 182, 180]	Ereignisstrom-Modell, Feedback-Modell		Sockets	Verwendet eigene Sprache für Steering-Befehle; basiert auf Progress	Wärmediffusion	
CSE [105, 106, 124]	Client/server/client, Ereignisstrom-Modell			Generische Benutzerschnittstelle basierend auf Parametrierbaren Graphischen Objekten (PGO)	Luftverschmutzung	PGO-Editor
Fortsetzung nächste Seite						

Name	Zugrundeliegende Modelle	Umgebung	Kommunikation	Besonderheiten	Anwendungen	Visualisierung
Pathfinder [74, 121, 100]	Agentenbasiert, Transaktionsbasiert	Parallele, verteilte Systeme	PVM [59], MPI [161] Sockets	Erhaltung und Überprüfung der Datenkonsistenz		
Autopilot [149, 177]	Ereignisstrom	verteilte Systeme, Grid	Nexus System [55], Globus	Wahl der Aktualisierungsstrategie teilweise automatisiert		Virtue [177]
gViz [21, 20, 187]	Datenflussmodell	Grid	Globus, Verwendung eines Vermittlers auf entfernter Site	Kommunikation ähnelt dem geschlossenen Szenario von RMOST (siehe Kap. 4.1)	Luftverschmutzung	IRIS Explorer
RealityGrid [88, 138, 23, 141]	Client/Server	Grid	Pipes, Sockets, SOAP über http, Globus		MD, Lattice Boltzmann, und andere	VTK Werkzeuge, Java Anwendung
Uintah [168]	Datenflussmodell	stark parallele Anwendungen	Nexus System [55]	basiert auf SCIRun	Brandsimulationen	Uintah PLU Schnittstelle
EPSN [49, 38]	Kontrollflussmodell, Client/server	Parallele Programme	CORBA		Dynamik von Flüssigkeiten	AVS/Express, VTK
MoSt [63]	Client/Server	Sequentielle Programme	Sockets	Dynamische Instrumentierung der Anwendung zur Laufzeit mit DynInst API [77]		Spezielle Werkzeuge
Fortsetzung nächste Seite						

Name	Zugrundeliegende Modelle	Umgebung	Kommunikation	Besonderheiten	Anwendungen	Visualisierung
OViD [145]	Client/Server	Parallelrechner und verteilte Systeme	CORBA		Dynamik von Flüssigkeiten	
Discover [126, 89, 118]	Objektmodell (DIOS)	Grid, Netzwerke	CORBA, Java RMI und http/https	Definition von Steering-Regeln		Webportal
GRASPARC [22]	Datenfluss Modell	Parallelrechner		Erstellt Baum von Momentaufnahmen der Anwendung	Kinetik von thermisch aktiven Prozessen	Spezielles Werkzeug
MOSS [47, 48]	Stellvertreterobjekt Modell	Vernetzte Rechner, Cluster	RMI/CORBA, PBIO/Data Exchange library [48]			
Cactus [4, 65]	Client/Server	Cluster, Grid	http/https	Cactus ist kein reines Steering-System	Astronomie	Jeder Browser
VISIT [46]	Client/server		TCP/IP, SSH, Dateien		Medizin, MD und Andere	AVS
VIPER [144]	Client/server/client	entfernte Parallelrechner	RPC		Dynamik von Flüssigkeiten	

Tabelle 3.1: Steering-Systeme

3.1.3 Fazit

Bisher gibt es bis kein Online-Steering-System, das Online-Steering als verteilten gemeinsamen Speicher betrachtet. Die auf diesem Modell aufbauenden Möglichkeiten der automatisierten Transportoptimierung, vereinfachten Integration in bestehende Anwendungen und Datenkonsistenzgarantien werden bisher von keinem der untersuchten Steering-Systeme umgesetzt.

Ferner existiert kein Online-Steering-System außer gViz [21], das eine sichere Verbindung zu einem Gridjob aufbauen kann. Bei gVis handelt es sich aber um eine PSE, so dass es nicht praktikabel ist gViz in bestehende Anwendungen zu integrieren. Andere Systeme wie RealityGrid [88], die den Anspruch erheben, Online-Steering im Grid zu ermöglichen, setzen voraus, dass der Gridjob Verbindungen aus dem Internet annehmen kann. Diese Annahme ist aber in vielen Produktionsgrids (z.B. dem LCG [96]) nicht erfüllt.

3.2 Verteilter gemeinsamer Speicher

Bei verteiltem gemeinsamen Speicher (DSM) haben Prozesse auf verschiedenen Rechnern Zugriff auf gemeinsam genutzte Speicherressourcen. Der gemeinsam genutzte Speicher ist physikalisch auf unterschiedliche Rechner verteilt. Mithilfe von Softwarebibliotheken und/oder Hardwareunterstützung wird ein virtueller gemeinsamer Speicher simuliert, auf den alle Prozesse zugreifen können.

Da Simulation und Visualisierung auf dieselben Daten zugreifen, bietet sich ein verteiltes gemeinsames Speichermodell für Online-Steering an. Es existieren bereits etliche DSM-Systeme, die verteilten gemeinsamen Speicher implementiert haben, aber keines der DSM-Systeme geht auf die speziellen Bedingungen beim Online-Steering ein.

In diesem Unterkapitel werden zunächst verschiedene Konsistenzmodelle vorgestellt. Anschließend werden andere Eigenschaften beschrieben, nach denen DSM-Systeme unterschieden werden können. Zuletzt werden existierende DSM-Systeme mit ihren Eigenschaften aufgelistet. Ein Vergleich und eine Systematisierung verschiedener Konsistenzmodelle wurde von Steinke und Nutt durchgeführt [167]. Andere Übersichten existieren von Nitzberg, Lo [131] und Shi [160]. Es existieren einige Ansätze für eine Formalisierung, in der die verschiedenen Konsistenzmodelle durch ein einheitliches System definiert werden [167, 12, 11].

3.2.1 Konsistenzmodelle

Eines der grundlegenden Merkmale eines DSM-Systems ist das verwendetet Konsistenzmodell. Es wurden bereits einige Konsistenzmodelle entwickelt, die in diesem Abschnitt kurz beschrieben werden. Im allegemeinen sind strenge Konsistenzmodelle einfacher zu verwenden, da ihr Verhalten dem Verhalten sequentieller Programme stärker ähnelt. Dafür sind lockere Konsistenzmodelle performanter [84].

Sequentielle Konsistenz

Die sequentielle Konsistenz (sequential consistency) [103] garantiert, dass alle Prozesse alle Speicherzugriffe in der selben Reihenfolge sehen. Die sequentielle

Konsistenz ist ein strenges Konsistenzmodell, das die Datenkonsistenz garantiert ohne Bedingungen an den Benutzer zu stellen. Ein System, welches dieses Modell implementiert, ist z.B. Ivy [107]. DSM Systeme die sequentielle Konsistenz implementieren, können für die Anwendung vollkommen transparent verwendet werden [98].

Linearisierbarkeit (linearizability) [75] ist eine noch strengere Eigenschaft als sequentielle Konsistenz. Linearisierbarkeit unterscheidet sich von sequentieller Konsistenz dadurch, dass sie für jede Operation ein Zeitintervall definiert, in der diese Operation ausgeführt werden muss. Wenn sich die Intervalle zweier Operationen nicht überlappen, darf deren Ausführungsreihenfolge nicht vertauscht werden.

Speicherkonsistenz

Die Speicherkonsistenz (cache consistency) [66] oder auch Kohärenz ist ein schwächeres Konsistenzmodell als die sequentielle Konsistenz, was zu einer besseren Performance führt. Ein System ist speicherkonsistent, wenn für jede Speicherstelle alle Prozesse die Schreibzugriffe auf diese Speicherstelle in derselben Reihenfolge sehen. Speicherzugriffe auf verschiedene Speicherstellen können von unterschiedlichen Prozessoren in verschiedener Reihenfolge gesehen werden. Im Folgenden wird mit $w(p, x, v)$ ein Schreibzugriff von Prozessor p auf Speicherstelle x notiert, wobei p den Wert v nach x schreibt.

Angenommen die Prozessoren p_1 und p_2 führen die folgenden Schreibzugriffe aus:

$$\begin{aligned} p_1 &: w(p_1, x, 1), w(p_1, y, 1) \\ p_2 &: w(p_2, y, 2), w(p_2, x, 2) \end{aligned} \tag{3.1}$$

dann könnten p_1 und p_2 folgende Reihenfolge sehen:

$$\begin{aligned} p_1 &: w(p_1, y, 1), w(p_1, x, 1), w(p_2, x, 2), w(p_2, y, 2) \\ p_2 &: w(p_1, x, 1), w(p_1, y, 1), w(p_2, y, 2), w(p_2, x, 2) \end{aligned} \tag{3.2}$$

Nicht möglich wäre:

$$\begin{aligned} p_1 &: w(p_1, y, 1), w(p_1, x, 1), w(p_2, x, 2), w(p_2, y, 2) \\ p_2 &: w(p_1, y, 2), w(p_1, x, 1), w(p_2, x, 2), w(p_2, y, 1) \end{aligned} \tag{3.3}$$

In diesem Fall würden p_1 und p_2 die Zugriff auf y in unterschiedlicher Reichenfolge sehen.

PRAM Konsistenz

Die PRAM Konsistenz (pipelined RAM consistency) [108] fordert, dass alle Schreiboperationen eines Prozessors von allen anderen Prozessoren in derselben Reihenfolge gesehen werden. Allerdings dürfen Schreiboperationen verschiedener Prozesse von unterschiedlichen Prozessoren in unterschiedlicher Reihenfolge gesehen werden. Angenommen p_1 und p_2 führen die in (3.2) beschriebenen Schreiboperationen aus. Dann könnten p_3 und p_4 folgende Zugriffe sehen

$$\begin{aligned} p_3 &: w(p_1, x, 1), w(p_2, y, 2), w(p_1, y, 1), w(p_2, x, 2) \\ p_4 &: w(p_2, y, 2), w(p_1, x, 1), w(p_2, x, 2), w(p_1, y, 1) \end{aligned} \tag{3.4}$$

Es ist nicht erlaubt, Schreiboperationen desselben Prozesses zu vertauschen.

Prozessor-Konsistenz

Für die Prozessor-Konsistenz (processor consistency) gibt es zwei Definitionen [66, 61], die nicht äquivalent sind [3]. Die verbreitetere Variante ist die Definition von Goodman [66], die auch in dieser Arbeit verwendet wird. Bei der Prozessor-Konsistenz sehen alle Prozessoren Schreibzugriffe auf dieselbe Speicherstelle in derselben Reihenfolge und Schreibzugriffe eines Prozessors werden von allen Prozessoren in derselben Reihenfolge gesehen.

Zum Beispiel könnten p_1 und p_2 folgende Reihenfolge der Zugriffe aus Beispiel (3.2) sehen:

$$\begin{aligned} p_1 &: w(p_1, x, 1), w(p_2, y, 2), w(p_1, y, 1), w(p_2, x, 2) \\ p_2 &: w(p_2, y, 2), w(p_1, x, 1), w(p_2, x, 2), w(p_1, y, 1) \end{aligned} \tag{3.5}$$

Es ist also möglich, dass p_1 und p_2 eine unterschiedliche Reihenfolge der Schreiboperationen sehen. Allerdings könnte p_2 nie $w(p_1, y, 1)$ vor $w(p_1, x, 1)$ sehen, weil die Reihenfolge, in der die Schreibzugriffe vom jeweiligen Prozessor ausgeführt werden, eingehalten werden muss. Die Reihenfolge (3.3) ist also nicht Prozessorkonsistent. Prozessor-Konsistenz beinhaltet Speicherkonsistenz und die PRAM Konsistenz.

Schwache Konsistenz

Bei der schwachen Konsistenz (weak consistency) [40] gibt es zusätzliche Synchronisationsoperationen. Ein Prozess wird ausgeführt, ohne dass irgendwelche Konsistenzgarantien gegeben werden, bis eine Synchronisationsoperation erreicht ist. Nach einer Synchronisationsoperation wird garantiert, dass alle bis dahin durchgeführten Änderungen global, d.h. für allen anderen Prozesse, sichtbar sind. Die Synchronisationsoperationen sind Prozessor-konsistent. Es liegt in der Verantwortung des Programmierers genügend Synchronisationsoperationen einzufügen damit das parallele Programm korrekt ausgeführt wird. Andererseits kann durch die schwächere Synchronisation eine bessere Performance erzielt werden, vor allem da der Kommunikationsaufwand gegenüber sequentiell konsistenten Systemen sinkt.

Freigabe-Konsistenz

Die Freigabe-Konsistenz (release consistency) [62] unterscheidet ebenfalls zwischen Datenzugriffen und Synchronisationsoperationen. Es existieren zwei Synchronisationsoperationen. Die Anforderung (acquire) und die Freigabe (release). Das DSM-System garantiert, dass Synchronisationsoperationen sequentiell konsistent sind. Bevor auf den gemeinsamen Speicher zugegriffen werden darf, muss eine Anforderung für den entsprechenden Speicherbereich ausgeführt werden. Operationen auf den Daten dürfen erst ausgeführt werden, wenn alle vorhergehenden Anforderungsoperationen abgeschlossen wurden.

Die eigentliche Synchronisation findet bei der Freigabeoperation statt. Die Freigabeoperation ist erst beendet, wenn alle Änderungen im gemeinsamen Speicher auf allen Kopien durchgeführt wurden. Es ist also sichergestellt, dass nach einer Anforderung alle Änderungen an der lokalen Kopie durchgeführt wurden,

die vor den vorhergehenden Freigaben gestartet wurden. Manche Implementierungen unterstützen nur exklusiven Schreibzugriff, andere [32, 80, 155] erlauben mehreren Prozessen gleichzeitig zu schreiben.

Die Freigabe-Konsistenz wurde zum Beispiel von Munin [32, 13, 14, 33] implementiert. Von dem Grundmuster der Freigabe-Konsistenz gibt es einige Variationen, die häufig einen eigenen Namen haben wie die faule Freigabe-Konsistenz [94], die Eintritt-Konsistenz [16] und die Gültigkeitsbereich-Konsistenz [83].

Die faule Freigabe-Konsistenz (lazy release consistency) [94] unterscheidet sich von der Freigabe-Konsistenz nur durch die interne Synchronisation. Anstatt die Modifikationen im Speicher bei der Freigabeoperation an allen Kopien durchzuführen, werden die Änderungen erst bei der nächsten Anforderungsoperation und nur bei dem anfordernden Prozess angewendet. Falls mehrere Kopien einer Speicherstelle existieren, kann dadurch die Performance verbessert werden, da weniger Nachrichten verschickt werden müssen. Die faule Freigabe-Konsistenz wurde in Treadmarks [93, 42] entwickelt und implementiert.

Die Eintritts-Konsistenz (entry consistency) [16] wurde im DSM-System Midway [16] entwickelt. Anstatt den Zugriff auf der Basis von einzelnen Speichereinheiten (Seiten, Variable) zu vergeben, wird jede Speicherstelle explizit mit einer Synchronisationsvariable verknüpft. Dadurch werden explizit Gruppen mit logisch zusammengehörigen Speicherstellen zusammengestellt. Die Synchronisationsoperationen erfolgen dann immer auf diese Synchronisationsvariable und wirken sich auf alle mit ihr verknüpften Speicherstellen aus. Somit wird die Synchronisation immer für eine Gruppe von Speicherstellen durchgeführt.

Die Gültigkeitsbereich-Konsistenz (scope consistency) [83] basiert auf der Eintritt-Konsistenz. Allerdings müssen die Daten nicht explizit einer Synchronisationsvariable zugeordnet werden, sondern werden automatisch Synchronisationsvariablen zugeordnet. Die Gültigkeitsbereichs-Konsistenz ist z.B. in JIAJIA [80] implementiert. Das gleiche Ziel wird in [78, 79] verfolgt, aber dort anders genannt (view consistency).

Kausaler Speicher

Bei Kausalem Speicher (causal memory) [90] werden Abhängigkeiten zwischen zwei Speicherzugriffsoperationen o_1, o_2 definiert. Eine Abhängigkeit von o_2 von o_1, bedeutet dass o_1 vor o_2 ausgeführt werden muss. Es gibt zwei Arten von Abhängigkeiten:

- Programmreihenfolge: Die Reihenfolge, in der ein Prozess Speicheroperationen ausführt, muss gewahrt bleiben.

- Zuvor-Geschrieben Abhängigkeit: Eine Speicherzugriffsoperation darf erst ausgeführt werden, wenn eine bestimmte Schreiboperation durchgeführt wurde. Wenn z.B. ein Prozess p_1 ein Zwischenergebnis lesen soll, das zuvor von Prozess p_2 berechnet wurde, darf p_1 die Leseoperation erst ausführen nachdem p_1 die Schreiboperation von p_2 gesehen hat.

Kausaler Speicher garantiert, dass alle Prozesse alle Speicheroperationen in einer Reihenfolge sehen, die konform zu den definierten Abhängigkeiten ist.

DAG Konsistenz

Bei der DAG Konsistenz (dag consistency) [18] wird die Anwendung durch asynchrone Prozeduraufrufe und Synchronisationsoperationen in verschiedene Threads aufgeteilt, wobei eine Synchronisationsoperation wartet bis alle von dieser Prozedur aufgerufenen Prozeduren beendet sind. Wenn eine Prozedur durch eine Synchronisationsoperation geteilt wird, ist jeder Teil ein eigener Thread. Die Threads bilden die Knotenmenge eines Graphen. Die Prozeduraufrufhierarchie ist durch gerichtete Kanten abgebildet. Außerdem existieren Kanten zu dem Thread von allen Threads, auf die in einer Synchronisationsoperation gewartet wird. Beim Start eines Threads erhalten die Threads eine Kopie von einem Hauptspeicher. Nach Beenden des Threads werden die Änderungen in den Hauptspeicher geschrieben. Das bedeutet, jeder Thread kommuniziert immer nur mit dem Hauptspeicher, die lokalen Kopien der konkurrierenden Threads werden nicht aktualisiert. Die DAG Konsistenz ist in Cilk [18] implementiert.

Örtliche Konsistenz

Die örtliche Konsistenz (location consistency) [58] verwendet ähnlich der Freigabe-Konsistenz Anforderungs- und Freigabeoperationen zur Synchronisation. Allerdings sind die Konsistenzgarantien der örtlichen Konsistenz andere. Die Örtliche Konsistenz definiert eine Menge möglicher Ergebnisse einer Leseoperation zu einer Speicherstelle x, aus der ein beliebiges Element ausgewählt werden kann. Zu dieser Menge gehören:

- Alle Schreiboperationen anderer Prozessoren seit dem letzten Freigabe-Anforderungs-Paar.
- Die letzte Schreiboperation des lesenden Prozessors seit dem letzten Freigabe-Anforderungs-Paar.
- Die letzte Schreiboperation vor der letzten Freigabe, wenn der lesende Prozessor danach nicht mehr nach x geschrieben hat.

Zum Beispiel ist folgende Ausführungsreihenfolge möglich:

$$\begin{aligned} &ac(p_1,x), w(p_1,x,1), rel(p_1,x), ac(p_2,x), w(p_2,x,2),\\ &w(p_2,x,3), r(p_1,x,3), r(p_1,x,2), r(p_1,x,1), r(p_1,x,2),\\ &r(p_1,x,1), r(p_1,x,3), r(p_1,x,2), r(p_1,x,3), r(p_1,x,1) \end{aligned} \tag{3.6}$$

Es ist also möglich, dass mehrere aufeinander folgende Leseoperationen alternierend den Wert zweier Schreiboperationen zurück liefern,

3.2.2 Unterscheidungsmerkmale von DSM-Systemen

Neben den verschiedenen Konsistenzmodellen unterscheiden sich Implementierungen anhand Granularität, verwendetem Protokoll und Besitzer der Daten.

Granularität

Die Granularität definiert die atomare Dateneinheit, die das DSM-System verwendet. Grundsätzlich können seiten- und objektbasierte Systeme unterschieden werden. Bei objektbasierten DSM-Systemen wird jedes Datenobjekt, unabhängig von seiner Größe, separat behandelt. Objektbasierte Systeme sind z.B.

Globe [95, 166], ORCA [7] und Midway [16]. Dagegen wird bei seitenbasierten Systemen der gesamte lokale Speicher in Seiten aufgeteilt. Der Zugriff und die Änderungen werden dann seitenweise behandelt. Beispiele für seitenbasierten Implementierungen sind Munin [32] und Ivy [107]. Eine Übergangsform sind Regionenbasierte Systeme, bei denen der gemeinsame Speicher in Regionen beliebiger Größe aufgeteilt wird, z.B. bei CRL [91].

Ein Vorteil seitenbasierter Systeme ist, dass sie über Ausnahmebehandlungsroutinen von Seitenzugriffsfehlern transparent für die Anwendung Speicherzugriffe verfolgen können. Allerdings funktioniert seitenbasierter DSM nur bei homogenen Systemen. Bei seitenbasierten Systemen tritt außerdem das Problem des sogenannten *false sharing* auf. Bei gleichzeitiger Nutzung zweier Variablen der selben Seite werden viele unnötige Synchronisationen durchgeführt, was zu bedeutenden Performanceeinbußen führen kann. Dies kann umgangen werden, indem man gleichzeitig mehreren Prozessen gestattet, eine Seite zu schreiben [33]. Eine andere Möglichkeit ist MultiView [86], das Variablen, die auf derselben physikalischen Seite plaziert sind, auf unterschiedlichen virtuellen Seiten anordnet.

Da bei objektbasierten Systemen die Datentypen bekannt sind, können Konversionen in ein anderes Format vorgenommen werden. Somit können objektbasierte DSM Systeme auch in heterogenen Systemen implementiert werden. Objektbasierte Systeme benötigen aber häufig spezielle Unterstützung, um den Datenzugriff zu ermöglichen z.B. dadurch, dass der Programmierer seinen Quellcode an das DSM-System anpasst oder durch einen speziellen Compiler. Allerdings erfordern etliche schwächere Konsistenzmodelle wie die Freigabe-Konsistenz und die schwache Konsistenz sowieso manuelle Instrumentierung durch den Programmierer.

Da die Wahl der richtigen Granularität Einfluss auf die Performance hat, wurden Techniken entwickelt, die die Granularität dynamisch an das Zugriffsmuster anpassen können [132, 85]. Ein Vergleich zweier Systeme mit unterschiedlicher Granularität wurde von Dwarkadas et al. [43] durchgeführt. Zhou et al. [190] untersuchen die Performance in Abhängigkeit von Granularität und Konsistenzmodell.

Protokoll

Wenn ein Prozess einen Wert in eine Speicherstelle geschrieben hat, müssen alle Kopien aktualisiert werden. Es gibt zwei Grundmuster für Protokolle, um sicherstellen [107], dass alle Kopien auf dem aktuellen Stand sind: Aktualisierungsprotokolle (write-update) und Invalidierungsprotokolle (invalidate). Beim Aktualisierungsprotokoll wird bei einer Schreiboperation der neue Wert an alle Kopien weitergeleitet und sofort angewendet. Beim Invalidierungsprotokoll werden bei der ersten Schreiboperation alle Kopien als ungültig markiert. Weitere Änderungen bleiben dann für die Kopien ohne Wirkung. Wenn ein Prozess auf eine ungültige Kopie zugreifen möchte, wird zuerst der aktuelle Wert angefordert. Welches Protokoll günstiger ist, hängt von der Verteilung der Kopien und dem Zugriffsmuster ab [45, 13, 184, 44]. Einige DSM-Systeme unterstützen daher beide Protokolle [32].

In [44] wird ein hybrides Protokoll für die faule Freigabe-Konsistenz vorgestellt. Angenommen der Prozess p_1 hat zuletzt eine Freigabeoperation (release) ausgeführt, und als nächstes führt Prozess p_2 eine Anforderung (aquire) aus.

Die Anforderung wird von p_2 an p_1 geschickt. An die Antwort von p_1 werden die Aktualisierungen von allen Speicherstellen angehängt, von denen p_1 und p_2 ein Kopie besitzen. Wurde eine Speicherstelle von p_1 oder von einem anderen Prozess zuvor geändert, von der p_2 keine Kopie besitzt, wird eine Invalidierung von dieser Speicherstelle angehängt. Führt nun p_2 eine Freigabeoperation aus und ein weiterer Prozess p_3 fordert den Speicherzugriff von p_2 an, kann p_2 nur für die Speicherstellen eine Aktualisierung weitergeben, für die es selbst eine Aktualisierung erhalten hat, oder die er selbst modifiziert hat. Für die anderen Speicherstellen hängt p_2 die Invalidierungen an die Antwort an.

Besitzer

Als Besitzer eines Datenobjektes wird der Prozess bezeichnet, der den aktuellen Wert eines Datenobjektes hat. Es können dabei *heimatlose* und *beheimatete* Implementierungen unterschieden werden. Bei beheimateten Systemen besitzt jedes Datenobjekt eine feste Heimat, also einen festen Prozess, der der Besitzer dieses Objektes ist. Bei heimatlosen Systemen ändert sich der Besitzer; in der Regel ist der Besitzer der Prozess, der zuletzt den Wert geschrieben hat. Laut einer Studie von Yu et al. [188] sind beheimatete Systeme etwas performanter. JIAJIA [80] ist ein Beispiel für ein beheimatet System, während Munin [32] ein heimatloses System ist.

3.2.3 Existierende DSM-Systeme

Es wurden bisher etliche DSM-Systeme implementiert. In der Tabelle 3.2 werden bestehende DSM-Systeme nach dem implementierten Konsistenzmodell, Protokoll, Granularität und Besitzer kategorisiert. Außerdem sind Besonderheiten des DSM-Systems vermerkt.

Name	Konsistenzmodell	Protokoll	Besitzer	Granularität	Besonderheiten
Ivy [107]	Sequentielle Konsistenz	Invalidierer	Heimatlos	Seitenbasiert	
Munin [32, 13, 14, 33]	Freigabe-Konsistenz	Beide - adaptiv, gleichzeitiges Schreiben mehrerer Prozesse,	Heimatlos	Seitenbasiert	
TreadMarks [93]	Faule Freigabe-Konsistenz	Invalidierer	Heimatlos	Seitenbasiert	
Midway [16]	Eintritt-Konsistenz	Invalidierer	Heimatlos	Objektbasiert	
CRegionLibrary (CRL)[91]	Freigabe-Konsistenz	Invalidierer	Beheimatet	Regionenb.	
Orca [7]	Sequentielle Konsistenz	Aktualisierer		Objektbasiert	Programmiersprache für DSM-Systeme
JIAJIA [80]	Gültigkeitsbereich-Konsistenz	Invalidierer, gleichzeitiges Schreiben mehrerer Prozesse	Beheimatet	Seitenbasiert	
Khazana [34]	Sequentiell Konsistenz		Heimatbasiert, erlaubt Wechsel des Heimatprozesses	Regionenb.	
SMG (Shared Memory for Grids) [155]	Eintritt-Konsistenz	Aktualisierer, gleichzeitiges Schreiben mehrerer Prozesse		Objekt basiert	Implementiert DSM für eine Grid Umgebung
Shasta [156]	Freigabe-Konsistenz	Invalidierer	Beheimatet	Regionenb., Datenbasiert	
Clounds [143]			Heimatlos	Objektbasiert	Betriebssystem
DSM in Java [189]	Faule Freigabe-Konsistenz	Aktualisierer	Heimatlos	Objektbasiert	Basiert auf TreadMarks
Fortsetzung nächste Seite					

Name	Konsistenzmodell	Protokoll	Besitzer	Granularität	Besonderheiten
Cilk [18]	DAG Konsistenz	Backer Algorithmus [18] (Invalidierer)	Beheimatet	Seitenbasiert	
MilliPage [86]	Sequentielle Konsistenz	Invalidierer	Heimatlos	MultiView [86]	Erweiterung für adaptive Granularität [132, 85]
Jackal [176]	Java-Synchronisation	Invalidierer	Beheimatet	Objektbasiert, Regionenb.	Optimierungen durch Compiler
Blizzard [157]	Sequentielle Konsistenz	Invalidierer		Regionenb.	Implementierung des Tempest Interface [148]. Verschiedene Versionen mit verschiedenen Zugriffskontrollmechanismen
DOSA [81]	Freigabe-Konsistenz	Invalidierer		Objektbasiert	Laufzeitinformationen ob Referenz oder nicht.
Cashmere [99]	Faule Freigabe-Konsistenz	Invalidierer	Beheimatet	Seitenbasiert	

Tabelle 3.2: Implementierungen von verteiltem, gemeinsamen Speicher

3.3 Interaktive Anwendungen im Grid

Die meisten Anwendungen, die in einem Grid ausgeführt werden, laufen autonom, d.h. ohne Benutzerinteraktion. Online-Steering-Systeme hingegen ermöglichen die Interaktion mit dem laufenden Gridjob. Bisher gibt es nur wenige interaktive Gridanwendungen. In diesem Abschnitt werden existierende interaktive Anwendungen und Projekte beschrieben. Von besonderem Interesse ist dabei, wie sie eine interaktive Verbindung herstellen.

CrossGrid Projekt

Das Ziel der Interaktion im CrossGrid Projekt [76, 29, 120, 140, 101] ist die Ausführung interaktiver Programme im Grid. Dabei wird allerdings die Verbindung vom Gridjob zum Benutzer aufgebaut, der nach der Jobsubmission online bleiben muss. Dagegen kann der Gridjob beim Online-Steering autonom laufen, bis der Benutzer die Verbindung initiiert.

Im CrossGrid Projekt wird der Job mit einem speziellen Submissionsprogramm gestartet. Bevor das Submissionsprogramm den Job an das Grid übergibt, startet es einen so genannten Schattenprozess für den Job. Der Schattenprozess ist ein Prozess, der entweder auf dem sog. Roaming Access Server oder dem Rechner des Benutzers läuft. Der jeweilige Rechner muss in einem Bereich von Ports eingehende Verbindungen vom Internet annehmen können. Danach wird der Job gestartet und verbindet sich mit seinem Schattenprozess. Wenn der Benutzer mit seinem Job interagieren will, verbindet er sich ebenfalls mit dem Schattenprozess. Die Site, auf der die Gridjobs ausgeführt werden, muss ausgehende Verbindungen von allen Worker Nodes gestatten. Die Verbindungen werden über die Grid-Sicherheits-Infrastruktur (GSI) gesichert. Einen Mechanismus (z.B. einen Namensdienst) zum späteren Auffinden des Jobs gibt es nicht. Die Initiative zum Verbindungsaufbau liegt bei dem Job.

Auf Jobseite werden die Standard Ein- und Ausgaben (stdin, stdout, stderr) auf Systemeebene abgefangen und umgelenkt. Allerdings ist die Kommunikation auch auf die Standardkanäle beschränkt.

glogin

Ein weiteres Werkzeug, das aus dem CrossGrid Projekt entstanden ist, ist glogin [153, 152, 154]. Ursprünglich öffnete glogin eine interaktive Konsole auf dem Computing Element. Erweiterungen für glogin leiten TCP Verbindungen oder XWindows-Verbindungen weiter und ermöglichen so Interaktion mit den Worker Nodes. Da glogin einige spezielle Eigenschaften von Globus [52] nutzt, funktioniert es nur in Grids, die Globus verwenden.

Um eine Konsole auf dem Computing Element zu öffnen, wird glogin mittels des fork-jobmanagers auf dem Computing Element gestartet. Globus benötigt ein Intervall von offenen Ports, das in einer Umgebungsvariable gespeichert ist. In diesem Intervall öffnet glogin einen Socket. Der Port, auf dem glogin lauscht, ist dem Benutzer allerdings bisher nicht bekannt. Um Globus zu veranlassen die Portnummer zurückzusenden, wird ein neuer Prozess abgezweigt, der den offenen Port erbt. Anschließend wird der Vaterprozess beendet. Dadurch nimmt Globus an, der Job sei beendet und sendet die Ausgabe zurück. Anschließend wird eine Socketverbindung zu dem Kindprozess hergestellt.

glogin kann eine ähnliche Funktion übernehmen, wie der in dieser Arbeit verwendete Verbindungsdienst (siehe Kap. 4.1.3). Allerdings kann glogin nur auf dem Rechenelement gestartet werden, welches bei großen Sites gut ausgelastet ist. Da auch ein Verbindungsdienst eine hohe Prozessorlast erzeugen kann (siehe Kap. 7.1), können ein oder mehrere Instanzen auf einem Rechenelement zu einer Überlastung des Rechenelements führen.

I-GASP

I-GASP ist eine Abkürzung für Interactive Grid Application Service Provider [10]. Dabei werden interaktive Anwendungen in einem Grid bereit gestellt, die dann von Kunden genutzt werden können. Das Konzept von I-GASP geht von Anwendungen aus, die ständige Interaktion benötigen. Im Gegensatz dazu stammt der Verbindungswunsch beim Online-Steering von dem Benutzer und nicht von der Anwendung. Um die Kommunikation zwischen dem Anwender und der Anwendung, die auf einem Knoten des Anbieters läuft, zu ermöglichen, wird in einer demilitarisierten Zone des Anbieters ein Verbindungsdienst (VD) eingerichtet, mit dem sich beide Partner verbinden. Der VD leitet die Informationen zwischen Anwender und Anwendung weiter. Letztendlich ähnelt diese Architektur der des Migration Desktop und des Roaming Servers aus dem CrossGrid Projekt, nur dass der VD vom Anbieter bereitgestellt wird, während der korrespondierende Schattenprozess beim CrossGrid Projekt lokal läuft. Die Verbindung von der Anwendung zum VD besteht, solange die Anwendung läuft.

RealityGrid Projekt

Im Rahmen des RealityGrid Projekte wurde ein Steering-System für das Online-Steering im Grid entwickelt [23] (siehe Abschnitt 3.1). Beim Steering im RealityGrid Projekt wird von einem Client/Server Modell ausgegangen. Es werden Bibliotheken bereitgestellt, die grundlegende Steering-Aktionen anbieten. Die Anwendung muss Daten mit Aufrufen an die Steering-Bibliothek registrieren und wird benachrichtigt, wenn Ereignisse auftreten, z.B. wenn der Benutzer eine Steering-Aktion ausgeführt hat. Die Anwendung muss dann auf die Ereignisse selbst reagieren und kann dazu teilweise Funktionen aus der Steering-Bibliothek nutzen. Es wird angenommen, dass die Anwendung eine zentrale Hauptschleife besitzt, in der regelmäßig eine `check` Methode aufgerufen wird. Innerhalb der `check` Methode wird die Kommunikation abgewickelt.

Um eine Kommunikation herzustellen, wird im RealityGrid Projekt wird davon ausgegangen, dass eine direkte Verbindung zum Gridjob aufgebaut werden kann. Es ist möglich, mittels einer direkten Globus-Verbindung, TCP, SOAP über HTTP oder Pipes zu kommunizieren.

gViz

Im gViz Projekt [21, 20, 187] (siehe auch Abschnitt 3.1) wurde der IRIS-Explorer [183] erweitert, so dass Module des IRIS-Explorer auf anderen Grid-Sites gestartet werden können. Um mit den Komponenten auf den entfernten Gridrechnern zu kommunizieren, muss auf einem Knoten der Site, der von beiden Seiten erreichbar ist, ein gVizProxy installiert werden, der die Kommunikation weiterleitet. Zur Kommunikation mit dem gVizProxy wird Globus Toolkit 2 [52]

verwendet. Das Verfahren zum Verbindungsaufbau gleicht dem geschlossenen Szenario (siehe Kap. 4.1.3). gViz verfolgt einen Ansatz, der auf dem Datenflussmodell basiert.

3.4 Andere

In diesem Kapitel werden noch einige Arbeiten vorgestellt, die ebenfalls mit dieser Arbeit verbunden sind oder verwandte Probleme behandeln, aber nicht zu einer, der in Kapitel 3.1, 3.2 und 3.3 untersuchten, Kategorien gehören. So haben z.B. verteilte Webobjekte [95, 35, 72] Gemeinsamkeiten mit der vorliegenden Arbeit in der verteilten Speicherung von Daten. Zur Optimierung des Steering-Systems können verschiedene Zugriffscharakteristiken unterschieden werden (siehe Kap. 4.5). Studien über Zugriffsmuster existieren beispielsweise für DSM und bestehende Steering-Ansätze, die evtl. übertragen werden können. Manche Monitoringsysteme erlauben die Überwachung von Jobspezifischen Daten und haben so eine Verwandtschaft mit dem Online-Steering, wobei allerdings die Richtung vom Steuerungswerkzeug zur Anwendung fehlt. Schließlich können auch Daten in Dateien gespeichert werden, die von dem Steering-System gelesen oder manipuliert werden sollen (siehe Kap. 5.4.3). Hier existiert eine Verbindung zu Netzwerkdateisystemen.

3.4.1 Verteilte Webobjekte

Bei Webobjekten handelt es sich um Objekte eines Webdokumentes oder eines Internetdienstes, die im Internet über verschiedene Rechner verteilt sind. Der Zustand der einzelnen Objekte muss maschinenübergreifend konsistent gehalten werden. Globe [95, 166] implementiert ein internetweites, objektorientiertes System für verteilte Anwendungen im Internet. In [72] und [35] geht es um die Performancevorteile durch Zwischenspeichern von Webseiten und daraus resultierende Konsistenzprobleme. Im Vergleich zum Online Steering ist die Anforderung an die Aktualität der Daten deutlich geringer. So ist eine Verzögerung von einigen Minuten bis hin zu ein paar Stunden bei Webobjekten noch akzeptabel.

3.4.2 Monitoringsysteme

Monitoringsysteme existieren für unterschiedliche Umgebungen. Grundsätzlich lassen sich Ressourcenmonitoringsysteme von Jobmonitoringsysteme unterscheiden. Ressourcenmonitoringsysteme zeigen Daten über die entsprechende Umgebung an, wie zum Beispiel die Verfügbarkeit von Diensten, deren Auslastung, etc. Jobmonitoringsysteme konzentrieren sich auf die Ausführungsumgebung eines Jobs. Sie messen z.B. die benötigte Prozessorzeit und Speicherausnutzung eines Jobs. Zwischenergebnisse, wie sie im Steering beobachtet werden können, werden nicht unterstützt. Evtl. lassen sich flexiblen Monitoringsystemen als Transportkanal verwenden, wenn sie eine API anbieten, mit deren Hilfe beliebige Daten übertragen werden können.

Monitoringsysteme habe eine eindirektionale Kommunikation von der Ressource oder dem Job zum Benutzer. Im Gegensatz zum Steering fehlt das interaktive Element, mit dem der Benutzer Modifikationen am Job vornehmen kann. Da Ressourcen sich zum Teil auf wesentlich größeren Zeitskalen ändern

wie Daten in einer Anwendung, erachten einige Systeme ein Aktualisierungsverzögerung von mehreren Minuten als akzeptabel. Steering benötigt dagegen eine kurze Antwortzeit.

Beispiele für Ressourcenmonitoringsysteme im Grid sind MDS 4 [158] und GridICE [5]. R-GMA [30, 186, 31, 37, 142] und MonALISA [82, 130, 129] bieten ein flexibles System für das Monitoring unterschiedlicher Daten im Grid und können daher beides unterstützen. Benötigt wird jeweils ein geeigneter Sensor, der die entsprechenden Daten schreibt. R-GMA bietet auch eine API mit deren Hilfe eine Anwendung Logginginformationen schreiben kann. AMon [125], Job Execution Monitor (JEM) [125], JobMon [165], PDM [87], OCM-G [8] und G-Monitor [139] sind Jobmonitoringsysteme für Gridumgebungen. Pablo [146, 147] ist ein Beispiel eines Monitoringsystems für parallele Anwendungen, die nicht im Grid laufen.

3.4.3 Zugriffsmuster

Die Performance von Programmen und von Datenzugriffen kann oft erhöht werden, wenn bekannt wäre, welche Zugriffe in der Zukunft erfolgen. In diesen Fällen könnten entsprechende Daten schon in Speicherbereiche (Caches) geladen werden, auf die schnell zugegriffen werden kann. Anhand vergangener Zugriffe kann man versuchen die nächsten Zugriffe vorherzusagen. Interessant ist dieses Vorgehen dann, wenn die Zugriffe langsam sind, z.B. weil die Daten in langsamen Speichern oder auf einem entfernten Rechner liegen.

In [117, 116] werden Zugriffsmuster auf Dateien durch drei Parameter klassifiziert:

- Sequenzielles Muster: Findet häufig nach einer Zugriffsoperation ein weiterer Zugriff auf die nachfolgenden Speichereinheiten statt? Oder ist der Abstand der Speicherstellen zweier aufeinander folgender Zugriffe häufig gleich?
- Größe einer Zugriffsoperation: Anzahl der Bytes, die bei einer Operation gelesen oder geschrieben werden.
- Lese/Schreibverhältnis: Verhältnis der Anzahl der Lese- und Schreiboperationen. Wird nur gelesen/geschrieben? Kommt beides gemischt vor?

Munin [32, 13, 14, 33] ist ein System, das verteilten gemeinsamen Speicher implementiert. In [14] werden folgende Zugriffsmuster unterschieden:

- Einmal schreiben: Daten, die einmal initialisiert, und die anschließend nur noch gelesen werden.
- Privat: Daten auf die nur ein Prozess zugreift.
- Häufig geschrieben: Daten, die häufig von verschiedenen Prozessen geschrieben werden.
- Ergebnis: Daten, die ein Ergebnis enthalten, das nach Berechnung nur noch gelesen wird.
- Synchronisation: Synchronisationsobjekte wie Sperren und Semaphore.

- Migrierend: Objekte auf die in Phasen zugegriffen wurde, wobei jede Phase eine Serie aus Zugriffen desselben Prozesses besteht.
- Produzent-Konsument: Ein Prozess schreibt die Daten, einer oder mehrere andere Prozesse lesen die Daten.
- Meist gelesen: Daten, die deutlich häufiger gelesen werden, als sie geschrieben werden.
- Unklassifiziert: Daten auf die keines der anderen Muster zutrifft.

Anhand von Zugriffsmustern kann versucht werden, vorherzusagen, welche Daten als nächstes angefordert werden. Die entsprechenden Daten können dann schon bevor sie angefordert werden, in einen schnell verfügbaren Zwischenspeicher zu laden (sog. Prefetching). Für DSM Systeme ist dieser Ansatz von Karlsson und Stenström [92], Bianchini et al. [17] und Lai und Falsafi [102] verwendet worden.

Ein anderer Ansatz wird von Dwarhadas et al. verfolgt. In [42] wird ein Compiler vorgestellt, welcher den Quellcode einer Anwendung analysiert und Hinweise zu Zugriffen generiert, die vom Laufzeitsystem verwendet werden können. Es werden Hinweise zu folgenden Zugriffsmustern gegeben:

- Lesen: Liest Teile der Seite.
- Schreiben: Schreibt Teile der Seite
- Lesen und Schreiben: Liest und schreibt Teile der Seite
- Alles Schreiben: Schreibt die komplette Seite, bevor sie gelesen wird.
- Alles Schreiben und Lesen: Überschreibt die komplette Seite. Allerdings dürfen neu geschriebene Teile schon gelesen werden, bevor die ganze Seite geschrieben wurde.

Je nachdem welches Zugriffsmuster auftritt, wird eine Seite schreibgeschützt, die Seite vorher kopiert, um später die Änderungen extrahieren zu können, oder die Seite vorher aktualisiert. Zum Beispiel braucht eine Seite, die sowieso komplett überschrieben wird, ohne gelesen zu werden, vorher nicht aktualisiert werden. Oder wenn die Seite nur gelesen oder komplett überschrieben wird, braucht keine Kopie erzeugt werden um Änderungen zu extrahieren.

Viele dieser Optimierungsansätze lassen sich auf das DSM-basierte Steering übertragen. In dieser Arbeit werden ebenfalls Möglichkeiten zur Performanceoptimierung in Abhängigkeit von der Zugriffsmustern und Netzwerkeigenschaften untersucht. Allerdings unterscheidet sich der Ansatz in dieser Arbeit von den in diesem Kapitel vorgestellten Optimierungen dadurch, dass eine kontinuierliche, parametrierbare Kostenfunktion verwendet wird.

3.4.4 Netzwerk-Dateisysteme

Das network file system (NFS) [137] oder deutsch Netzwerkdateisystem ermöglicht den Zugriff auf Dateien auf einem entfernten Rechner. Der Zugriff auf die entfernten Dateien geschieht transparent, d.h für die Anwendungsprogrammierung besteht kein Unterschied zwischen einer lokalen Datei und einer Datei auf

einem entfernten Rechner. Mit Hilfe von NFS wird ein logisches Dateisystem implementiert, das sich über mehrere physikalische Rechner verteilen kann.

Die Betrachtung von Netzwerk-Dateisystemen ist von Interesse, da in der verwendeten Anwendung viele Daten in Dateien abgelegt werden, auf die mit Hilfe des Steerings von entfernten Rechnern zugegriffen werden soll. Bei Netzwerkdateisystemen tritt ebenfalls das Konsistenzproblem der Dateien auf. NFS verwendet wechselseitigen Ausschluss, um die Konsistenz zu garantieren, wenn ein Prozess Daten schreiben möchte.

Ab NFS 4 [137] wird Authentifizierung und Autorisierung auf Basis der GSSAPI [68, 170] unterstützt.

Kapitel 4

Analyse und Modellierung

Online-Steering bietet die Möglichkeit, Daten einer autonomen Anwendung zur Laufzeit der Anwendung anzuzeigen und interaktiv in die Anwendung einzugreifen. Ein Steering-System muss also Daten einer laufenden Anwendung auslesen und benutzerinitiierte Aktionen ausführen können, die die weitere Ausführung der Anwendung beeinflussen. Dies impliziert eine Änderung des Zustands der Anwendung, was immer auch eine Änderung von Daten des Speichers bedeutet. Folglich kann Online-Steering darauf reduziert werden, Daten der Anwendung zu lesen und zu schreiben. Die erste Teilaufgabe eines Steering-Systems ist also Zugriff auf Anwendungsdaten zu erhalten.

Auf der Benutzerseite wiederum müssen die Daten visualisiert werden. Dafür können spezielle Steuerungswerkzeuge entwickelt werden, aber oft bestehen bereits Visualisierungswerkzeuge zur Anzeige von Ergebnissen, die um Online-Steering-Funktionalität erweitert werden können.

Damit es überhaupt zu einem Datenaustausch zwischen der Anwendung und dem Steuerungswerkzeug kommen kann, wird zweitens ein Kommunikationskanal zwischen der Anwendung und dem Steuerungswerkzeug benötigt.

Drittens greifen beim Online-Steering die Anwendung und das Steuerungswerkzeug parallel auf dieselben Datenobjekte zu und ändern diese möglicherweise. Dies kann zu Datenkonsistenz- und Integritätsproblemen führen. Ein Online-Steering-System muss die Konsistenz und Integrität der Daten bewahren.

Schließlich sollte das Steering-System in der Lage sein, komplexe Steering-Operationen ausführen zu können. Dies können z.B. Transformationen in andere Sichtweisen, Reduktionen der Daten, Filterung nach bestimmten Kriterien oder sogar automatische Evaluation von Zwischenergebnissen sein. Diese Berechnungen sollten möglichst nahe an der Datenquelle ausgeführt werden, um unnötige Kommunikation zu vermeiden.

Die drei Teilbereiche Interaktive Kommunikation, Datenkonsistenz und Datenzugriff bauen in einer Anbieter/Nutzer Beziehung hierarchisch aufeinander auf (siehe Abb. 4.1), in die als vierte Schicht die Auswertungskomponenten eingefügt werden, die natürlich auch den Zugriff auf die Daten benötigen, sich aber selbst nicht um die Konsistenz kümmern, sondern dafür auf der Datenkonsistenzschicht aufsetzen.

Die Grundlage des Steerings bildet eine Kommunikationsplattform, die den Datenaustausch zwischen der Benutzerschnittstelle und der Anwendung ermöglicht. Dies kann ein gemeinsamer Speicher, eine Socketverbindung oder eben ein

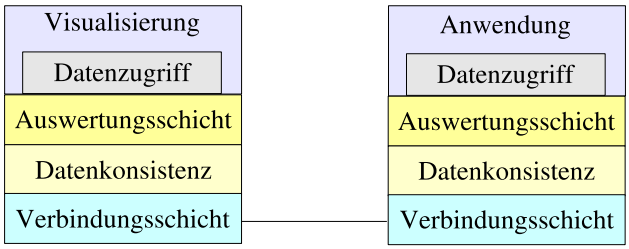


Abbildung 4.1: Schichtenmodell

komplexer, sicherer Kommunikationskanal im Grid sein.

Auf der Kommunikationsplattform setzt die Datenkonsistenzschicht auf. Sie stellt die Funktionalität bereit, die die Integrität der Daten gewährleistet. Dazu implementiert sie ein Konsistenzmodell, welches die Regeln definiert, in welcher Reihenfolge Datenzugriffe ausgeführt werden dürfen, und was der momentan gültige Wert eines Datenobjektes ist. Sie gewährleistet, dass die Werte der verschiedenen Kopien der Daten in dem Rahmen, der von dem Konsistenzmodell vorgegeben ist, übereinstimmen und übernimmt die Synchronisation der verschiedenen Prozesse. Es wird ein automatischer Mechanismus bereitgestellt, der auf Datenänderungen selbständig reagiert.

Die dritte Schicht ist die Auswertungsschicht. Die Ausführung komplexer Steering-Operationen wie die Optimierung, automatische Auswertung und Datenkompression, gehört in diese Schicht. Die in dieser Arbeit untersuchte Optimierung, bei der Zugriffsmuster analysiert und automatisch das Konsistenzprotokoll angepasst wird, um die Performance zu verbessern, ist in dieser Schicht angesiedelt.

Die oberste Schicht ist die Datenzugriffsschicht. Sie stellt Werkzeuge bereit, um auf die Daten der Anwendung zuzugreifen. Das Ziel ist eine einfache Integration des Steering-Systems in bestehende Anwendungen. Sie soll Datenzugriffe der Anwendung erfassen und die entsprechenden Konsistenzoperationen der unteren Schichten initiieren. Damit bildet sie die Schnittstelle zwischen der Anwendung oder Visualisierung und dem Steering-System. Des weiteren ist sie für die Serialisierung der speziellen Datentypen verantwortlich.

Einzelne der vier Schichten können transparent ausgetauscht werden. So kann z.B. die Kommunikationsschicht ausgetauscht werden, ohne das andere Schichten geändert werden müssen. Dadurch können Anwendungen und Steuerungswerkzeuge leicht in andere Umgebungen überführt werden. Je nachdem welchen Kanal oder welches Protokoll die Kommunikationsschicht verwendet, kann dasselbe Steering-System auf unterschiedlichen Architekturen und Umgebungen wie SMP, Cluster oder Grid, laufen. Eine neue Anwendung muss nur den Datenzugriff ermöglichen und gegebenfalls die Datenzugriffsschicht anpassen. Neue Steering-Operationen werden möglich, indem die Auswertungsschicht erweitert wird und andere Konsistenzbedingungen werden umgesetzt, indem man die Datenkonsistenzschicht ändert.

Im folgenden werden daher die folgenden Teilaufgaben für sich analysiert:

- Aufbau einer interaktiven Grid Verbindung
- Zugriff auf Anwendungsdaten.

- Erhaltung der Datenkonsistenz und Integrität
- Optimierung der Performance

4.1 Interaktive Kommunikation im Grid

Die Grundlage für Online-Steering ist ein Kommunikationskanal zwischen den verschiedenen Prozessen des Gridjobs und dem User Interface. Dabei werden von verschiedenen Seiten z.B. vom Benutzer, dem Systemadministrator und der technischen Umgebung Anforderungen an den Kommunikationskanal gestellt. Zuerst werden daher die Anforderungen untersucht. Anschließend werden verschiedene mögliche Szenarien identifiziert und ein Mechanismus zum Aufbau des Kommunikationskanals für jedes Szenario beschrieben. Zum Schluss werden Sicherheitsaspekte betrachtet.

4.1.1 Anforderungen an den Kommunikationskanal

Der Kommunikationskanal ist ein Dienst, der vom Benutzer zum Online-Steering verwendet wird. Das bedeutet, er muss den Kommunikationscharakteristiken des Online-Steerings gerecht werden. Dies führt zu folgenden Anforderungen:

- Die Kommunikation muss in beiden Richtungen möglich sein. Dadurch scheidet die Verwendung von eindirektionalen Kommunikationsstrukturen, wie sie z.B. von manchen Monitoringwerkzeugen zur Verfügung gestellt werden, aus.
- Da bei manchen Anwendungen beträchtliche Datenmengen transferiert werden (beim ATLAS Experiment [172, 173] bis zu mehreren GB), muss die Datenübertragung effizient sein. Insbesondere der Datendurchsatz sollte hoch sein, um große Datenmengen zu transferieren. Außerdem sollte die Antwortzeit kurz sein. Dies impliziert, dass der Verwaltungsaufwand gering sein sollte.
- Die Gridjobs können mehrerer Stunden oder Tage laufen. Ein dauerhafte Verbindung, die die gesamte Joblaufzeit bestehen bleibt, scheidet daher aus. Der Benutzer muss die Möglichkeit haben, sich zu jedem Zeitpunkt von beliebigen Rechnern zu seinem Job zu verbinden, die Verbindung zu trennen und sich später erneut zu verbinden. Die Verbindung muss deshalb vom Benutzer initiiert werden. Hierin unterscheidet sich der benötigte Kommunikationskanal von ähnlichen Gridverbindungen, die z.B. in I-GASP [10] oder dem CrossGrid Projekt [29] entwickelt wurden.
- Die Verbindung muss sicher sein. Das bedeutet zum ersten, dass es nur autorisierten Personen möglich sein darf, sich zu dem Gridjob zu verbinden. Zweitens darf der Benutzer keine zusätzlichen Rechte auf den Rechnern einer Site erhalten. Und drittens darf Dritten nicht der Zugriff auf Rechner oder Dienste der Site ermöglicht werden.
- Die Komplexität des Grids sollte vor dem Nutzer verborgen bleiben. Insbesondere soll als Zieladresse der eindeutige Jobname verwendet werden.

Da es um Online-Steering im Grid geht, müssen beim Entwurf des Kommunikationskanals die Eigenschaften des Grids, wie Inhomogenität und Sicherheitsvorkehrungen von Grid-Sites, berücksichtigt werden. In den meisten Fällen ist ein direkter Verbindungsaufbau nicht möglich. Folgende Punkte müssen berücksichtigt werden:

- Die zugrundeliegende Kommunikationstruktur des Grids ist das Internet. Das bedeutet, dass die zugrundeliegende Kommunikationsinfrastruktur als unsicher angesehen werden muss.
- Die Gridjobs werden mit Hilfe eines Ressource Brokers einem Computing Element (CE) zugeteilt. Der Rechner, der das Computing Element repräsentiert, führt die Anwendung aber nicht selbst aus, sondern reicht den Job an ein Worker Node (WN) weiter. Während der Benutzer noch die Möglichkeit hat, das Computing Element anhand der Rückmeldungen des Ressource Brokers herauszufinden, ist es nicht ohne weiteres möglich den Worker Node zu ermitteln, der den Job tatsächlich ausführt.
- Die meistens Sites schützen ihre Rechner durch Firewalls vor unberechtigten Zugriffen. Diese Firewalls verhindern, dass eine direkte Verbindung zum Worker Node und damit zu dem Gridjob aufgebaut wird. Eventuell werden sogar ausgehende Verbindungen blockiert. Nur einige privilegierte Ports für die Frontrechner, wie das Computing Element, können von außerhalb kontaktiert werden. Von der Existenz solcher privilegierten Ports kann ausgegangen werden, da sonst die Jobsubmission zu diesen Rechnern nicht möglich wäre.
- Möglicherweise liegen die Worker Nodes in einem privaten Netzwerk und können gar nicht direkt mit dem Internet kommunizieren. Lediglich die Frontknoten, z.B. das Computing Element, sind sowohl im privaten Netzwerk als auch im Internet.
- Das Grid ist heterogen. Jede Site kann ihre eigenen Sicherheitsphilosophien umsetzen. Die Konfiguration der Firewall, das Vorhandensein von privaten Netzwerken und das Vorhandensein und die Installation bestimmter Softwarepakete kann von Site zu Site unterschiedlich sein. Bei Submittierung eines Jobs kann daher keine Aussage über die Konfiguration der Site und des Rechners, der den Job schließlich ausführt, getroffen werden.

Daraus lassen sich direkt weitere Bedingungen für den Aufbau des Kommunikationskanals ableiten:

- Es muss ein **Namensdienst** existieren, der es dem Benutzer bzw. dem vom Benutzer verwendeten Programm erlaubt, die Kontaktinformationen auszulesen. Der Gridjob muss sich beim Namensdienst mit seinem eindeutigen Jobnamen registrieren und seine Kontaktinformationen angeben. Das Steuerungswerkzeug kann dann von dem Namensdienst die zum Jobnamen gehörenden Kontaktinformationen erfragen und eine Verbindung zu der ermittelten Adresse aufbauen.
- Bei privaten Netzwerken muss ein **Verbindungsdienst** existieren, der auf den Frontrechnern die Weiterleitung der Kommunikation zwischen privaten Netzwerk und Internet ermöglicht. Der Verbindungsdienst sollte von

beiden Seiten kontaktiert werden können oder auf Anfrage auch selbst einen Kontakt zu einem Ziel herstellen können. Sind Verbindungen zu beiden Kommunikationspartnern hergestellt, werden Daten, die von einem Ende verschickt werden, auf der anderen Verbindung zum Ziel weitergeleitet. Die dadurch implizierten Sicherheitsrisiken werden im Abschnitt 4.1.5 behandelt.

- Die Verbindung muss trotz Firewall aufgebaut werden können, ohne die Sicherheitsphilosophie der Grid Site zu verletzen. Zum Beispiel würde die Voraussetzung, dass ein Bereich an Ports für die Worker Nodes geöffnet wird, die Sicherheit der Sites gefährden. Auch in diesem Fall kann ein Verbindungsdienst verwendet werden.

- Der Mechanismus, der den Kommunikationskanal aufbaut, muss dynamisch an die Sitekonfiguration angepasst werden, auf der der Job läuft. Ziel ist es den Kommunikationskanal in so vielen Fällen wie möglich aufzubauen. Bisher gibt es keine interaktiven Systeme, die flexibel mehrere Methoden einsetzen.

Im Folgenden wird mit *Konnektor* diejenige Anwendung gemeint, die den Verbindungsaufbau initiiert, während der *Akzeptor* die Anwendung ist, die Verbindungswünsche entgegennimmt. Beim Online-Steering ist also der Gridjob der Akzeptor und das Steuerungswerkzeug der Konnektor. Aber der Kommunikationskanal im Grid ist auch in anderen Anwendungsfällen verwendbar, z.B. zwischen zwei Jobs.

4.1.2 Szenarioeinteilung

In diesem Abschnitt werden die möglichen Konfigurationen, die eine ähnliche Charakteristik haben, in verschiedene Szenarien eingeteilt. In den Fällen, die zum selben Szenario gehören, kann ein Kommunikationskanal mit demselben Mechanismus aufgebaut werden. Für die Einteilung in die Szenarien sind folgende Kriterien relevant. In jedem Fall wird davon ausgegangen, dass der Konnektor Verbindungen zu Rechnern im Internet aufbauen kann.

- Gestattet die Firewall jedem Worker Node Verbindungen in einem Bereich von Ports anzunehmen? Dabei dürfen diese Ports nicht von irgendwelchen permanenten Diensten belegt sein. Von dieser Bedingung unberührt bleibt die Existenz von privilegierten Diensten, die Verbindungen auf festen Port annehmen dürfen.

- Gestattet die Firewalls allen Worker Nodes Verbindungen nach außen aufzubauen?

- Liegen die Worker Nodes in einem privaten IP Netzwerk?

- Darf ein Frontend-Knoten (z.B. ein Computing Element) Verbindungen auf einem festen Port annehmen? Es wird davon ausgegangen, dass ein Frontend-Knoten zumindest eine Verbindung ins Internet aufbauen kann.

Die sich daraus ergebenden möglichen Konstellationen sind in Tab. 4.1 aufgelistet. Existiert ein privates Netzwerk, spielt die Frage, ob die Firewall eingehende

Verbindungen zu allen Worker Nodes zulässt, keine Rolle mehr, da dann die WNs schon aufgrund des privaten Netzes keine Verbindungen mehr annehmen können.

Fall	Eing. WN	Ausg. WN	Eing. CE	Priv. Netz	Szenario
1	ja	egal	egal	nein	offen
2	nein	ja	egal	nein	halboffen
3	nein	nein	ja	nein	geschlossen
4	nein	nein	nein	nein	privat
5	egal	ja	egal	ja	halboffen
6	egal	nein	ja	ja	geschlossen
7	egal	nein	nein	ja	privat

Tabelle 4.1: Mögliche Konfigurationen und das zugehörige Szenario. Das Szenario ist davon abhängig, ob die Firewall eingehende bzw. ausgehende Verbindungen erlaubt, und ob ein privates IP-Netzwerk existiert.

Anhand dieser Merkmale werden vier Szenarien unterschieden:

- Das **offene Szenario**, bei dem die Worker Nodes vom Internet aus kontaktiert werden können (Fall 1).
- Das **halboffene Szenario**, bei dem die Worker Nodes keine Verbindungen annehmen können, aber selbst Verbindungen zu anderen Rechnern im Internet aufbauen können (Fall 2 und 5).
- Das **geschlossene Szenario**, bei dem die Rechner des Worker Node keine Verbindung zu Rechnern im Internet aufbauen können, aber ein evtl. vorhandener Verbindungsdienst auf einem Frontendrechner erreichbar ist (Fall 3 und 6).
- Das **private Szenario**, bei dem weder die Worker Nodes (WN) noch ein Verbindungsdienst (VD) auf einem Frontend-Knoten der Site eine Verbindung annehmen können. Vom Frontend-Knoten kann der VD lediglich eine Verbindung nach außen aufbauen, während für die Worker Nodes auch keine ausgehenden Verbindungen möglich sind (Fall 4 und 7).

4.1.3 Verbindungsaufbau

In diesem Abschnitt wird beschrieben, wie in den verschiedenen Szenarien eine Verbindung aufgebaut werden kann. Wie das richtige Szenario automatisch bestimmt werden kann, und wie der benötigte Verbindungsdienst und Namensdienst lokalisiert werden können, wird in Kapitel 4.1.4 beschrieben.

Das offene Szenario

Bei dem offenen Szenario wird der Verbindungsaufbau nicht von Firewalls oder privaten Netzwerken behindert, lediglich der Zielrechner ist für den Konnektor unbekannt. Um eine Verbindung aufzubauen wird folgendermaßen vorgegangen:

1. Der Akzeptor ermittelt den Hostnamen des Zielrechners auf dem er läuft und öffnet einen Port auf dem er Verbindungen annimmt.

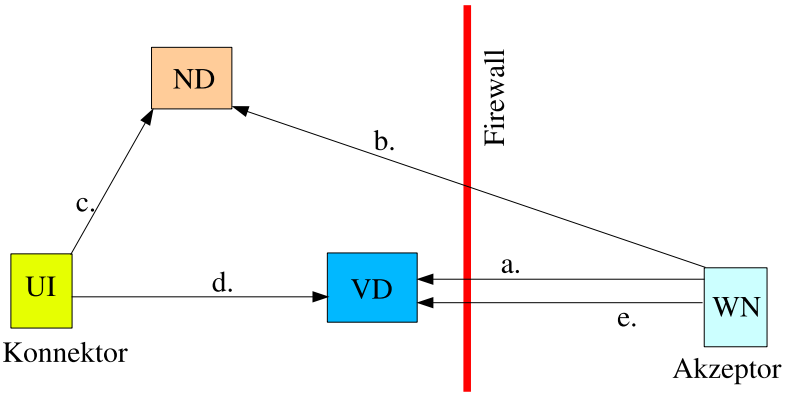


Abbildung 4.2: Verbindungsaufbau im halboffenen Szenario

2. Anschließend registriert sich der Akzeptor beim Namensdienst unter seinem Jobnamen und trägt als Kontaktinformation den Hostnamen und den Port ein, auf dem er Verbindungen annimmt.

3. Der Konnektor erfragt die Adresse des Akzeptors vom Namensdienst anhand des Jobnamens.

4. der Konnektor initiiert eine direkt Verbindung zum Akzeptor.

Das halboffene Szenario

Im halboffenen Szenario wird ein Verbindungsdienst (VD) benötigt, der über das Internet erreichbar ist. Der Verbindungsaufbau ist in Abbildung 4.2 dargestellt.

1. Der Akzeptor ermittelt ein passenden VD.

2. Der Akzeptor initiiert eine Verbindung A zum VD (a) und informiert den VD, dass er auf dieser Verbindung über Verbindungswünsche benachrichtigt werden möchte.

3. Danach registriert sich der Akzeptor beim Namensdienst (b) und trägt seine Adresse und die Adresse des VD als Kontaktinformation ein.

4. Der Konnektor erhält die Adresse des VD und des Akzeptors vom Namensdienst (c).

5. Anschließend verbindet sich der Konnektor zum VD (d) und meldet diesem seinen Wunsch an, zur Adresse des Akzeptors verbunden zu werden.

6. Der VD schickt dem Akzeptor eine Nachricht, dass ein Verbindungswunsch vorliegt. Daraufhin baut der Akzeptor eine zweite Verbindung B zum VD auf (e).

7. Der VD verknüpft nun die Verbindung B des Akzeptors mit der Verbindung des Konnektors. Alle Daten, die auf einer der beiden Verbindungen zum VD gesandt werden, werden vom VD auf der jeweils anderen Verbindung weitergeleitet.

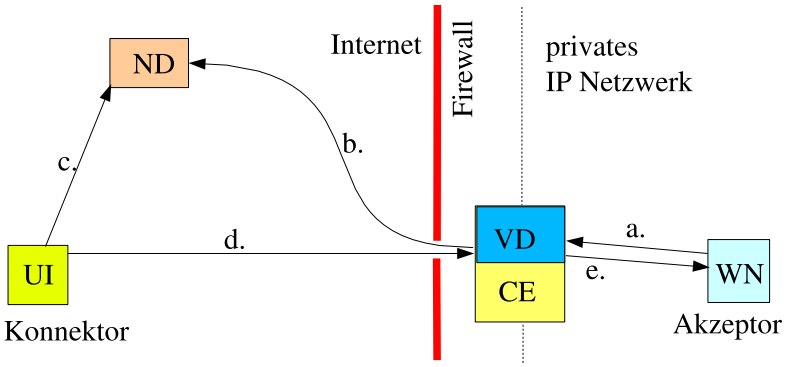


Abbildung 4.3: Verbindungsaufbau im geschlossenen Szenario

8. Wenn eine Verbindung abgebaut wird, schließt der VD auch die andere Verbindung. Die Verbindung A bleibt weiterhin bestehen und erlaubt dem Akzeptor weitere Datenkanäle aufzubauen.

Das geschlossene Szenario

Während im halboffenen Szenario der VD beliebig im Internet platziert sein kann, erfordert das geschlossene Szenario, dass ein VD so auf der Site installiert ist, dass er die Worker Nodes kontaktieren kann und auch von außerhalb der Site erreichbar ist. Jede Site besitzt mindestens ein Computing Element, welches auch von außen erreichbar ist, da sonst kein Job auf dieser Site ausgeführt werden könnte. Für den Verbindungsdienst muss ein Port in der Firewall geöffnet werden. Alternativen dazu werden später in diesem Abschnitt behandelt. Der Verbindungsaufbau ist in Abbildung 4.3 dargestellt.

1. Zuerst ermittelt der Akzeptor seine Adresse und die des VD, der auf dieser Site installiert ist.

2. Dann muss er sich beim ND mit seiner Adresse und der Adresse des verwendeten VD registrieren. Um dem ND zu kontaktieren, kann der ermittelte VD verwendet werden. Der Akzeptor verbindet sich also zum VD (a) und beantragt eine Weiterleitung zum ND. Der VD baut eine Verbindung zum ND auf (b) und leitet alle Daten vom Akzeptor zum ND weiter und umgekehrt. Diese Verbindung wird anschließend wieder abgebaut.

3. Der Konnektor erhält die Kontaktinformation vom ND (c) und baut eine Verbindung zum VD auf (d).

4. Der Konnektor fordert eine Weiterleitung zum Akzeptor beim VD an, woraufhin der VD eine Verbindung zum Akzeptor aufbaut (e).

5. Solange beide Verbindungen bestehen, leitet der VD alle Daten vom Konnektor zum Akzeptor weiter und umgekehrt. Wenn eine Verbindung geschlossen wird, beendet der VD die Verbindung mit der anderen Seite ebenfalls.

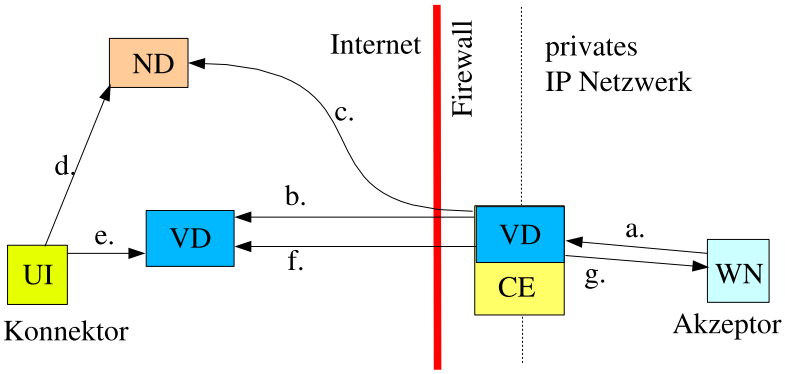


Abbildung 4.4: Verbindungsaufbau im privaten Szenario ohne privilegierten Port für den Verbindungsdienst

Das private Szenario

Wenn der VD keinen offenen Port hat, auf dem er Verbindungen annehmen kann, aber eine ausgehende Verbindung für den VD existiert, ohne dass die Worker Nodes ausgehende Verbindungen aufbauen können, kann man eine Variante wählen, die zwei VD verwendet, einen auf der Site (der lokale VD) und einen außerhalb der Site (der freie VD), und das Verfahren von halboffenen und geschlossenen Szenario kombinieren (siehe Abb. 4.4).

1. Der Akzeptor teilt dem lokalen VD seine Registrierungswunsch, seine Adresse und seinen Jobnamen mit (a).
2. Der lokale VD initiiert eine Verbindung A zu einem freien VD (b). Der lokale VD braucht nur eine Verbindung zu einem VD für alle Verbindungen der Site.
3. Der lokale VD registriert den Akzeptor beim ND (c) mit Jobname, Adresse des Zielrechners, Adresse des lokalen VD, Adresse des freien VD.
4. Der Konnektor holt die Verbindungsinformationen vom ND (d).
5. Analog zum halboffenen Szenario wird eine Kommunikation über den freien VD zum lokalen VD erstellt, indem zuerst der freie VD kontaktiert wird (e), der über Verbindung A eine neue Verbindung von dem lokalen VD anfordert (f).
6. Der lokale VD kontaktiert den Akzeptor (g). Die Informationen werden jeweils von beiden Verbindungsdiensten weitergeleitet. Zum Informationsaustausch existiert für jede logische Verbindung von einem Konnektor zu einem Akzeptor jeweils eine eigene Verbindung zwischen den Verbindungsdiensten.

Falls im geschlossenen oder privaten Szenario der VD die Worker Nodes nicht kontaktieren kann, aber die Worker Nodes sich zum VD verbinden können, lässt sich auch der Teil zwischen lokalem VD und Akzeptor durch einen Mechanismus ersetzen, der dem halboffenen Szenario gleicht.

4.1.4 Automatische Szenarioerkennung

Im Grid werden die Ressourcen von verschiedenen Betreibern bereit gestellt, die individuell unterschiedliche Sicherheitskonzepte verwenden. Daraus resultieren unterschiedliche Konfigurationen von Firewall und privaten Netzwerken auf den verschiedenen Sites eines Grids. Es kann nicht davon ausgegangen werden, dass alle Sites einen VD an einer einheitlichen Stelle installiert haben. Da zur Submissionszeit des Gridjobs die Zielsite im allgemeinen nicht bekannt ist, muss die Konfiguration dynamisch ermittelt werden und der Verbindungsaufbau sich an die Umgebung der Site anpassen. Um eine Verbindung aufbauen zu können, muss der Akzeptor vier Fakten ermitteln:

- Die Adresse des Rechners auf dem er läuft. Die Ermittlung der Rechneradresse stellt kein Problem dar und kann mit Standardfunktionen erledigt werden.
- Die Adresse des Verbindungsdienstes.
- Die Kontaktmöglichkeiten des Namensdienstes.
- Ein mögliches Szenario. Möglicherweise funktionieren mehrere Szenarien auf einer Site.

Der Verbindungsdienst

Um das Szenario zu ermitteln oder evtl. den Namensdienst zu kontaktieren, wird zu allererst ein VD benötigt. Es gibt verschiedene Möglichkeiten einen VD zu finden.

1. Der Administrator hat einen lokalen VD durch Umgebungsvariablen oder Konfigurationsdateien spezifiziert.
2. In einem lokalen Netz kann eine Anfrage per Broadcast versendet werden, auf die der VD sich meldet.
3. Es kann auf einem standardisierten Port auf dem Computing Element oder einem anderen Frontendrechner der Site angefragt werden, die dem WN bekannt sind.
4. Der Benutzer (oder das Submissionsprogramm) kann bei der Jobsubmission einen VD, den er kennt, spezifizieren. Dies kann ein permanenter Dienst sein oder bei Bedarf durch den Benutzer gestartet werden. Z.B. kann auf dem entsprechen Computing Element ein VD mittels Globus fork jobmanager gestartet werden. Dieser öffnet einen Port im Bereich, der durch die Umgebungvariable `GLOBUS_TCP_PORTRANGE` spezifiziert ist. Dieses Vorgehen erlaubt eine Implementierung, die keine Installation von Diensten durch einen Systemadministrator benötigt.
5. Es können sog. Service Locator Dienste befragt werden, die passende Dienste finden.

Da sich der Akzeptor in einem geschlossenen oder privaten Szenario befinden kann, versucht er zuerst herauszufinden, ob es einen lokalen VD gibt, den er

verwenden kann. Dabei wird die explizite Konfiguration dem Broadcast vorgezogen. Wurde bisher kein VD gefunden, kann der Akzeptor noch Standardports auf bekannten Frontend-Rechnern ausprobieren, um einen lokalen VD zu ermitteln. Dadurch, dass lokale Verbindungsdienste bevorzugt verwendet werden, kann der Site-Administrator die Interaktionsregeln konfigurieren und nachvollziehen, welche Verbindungen über den VD hergestellt wurden.

Wird kein lokaler VD gefunden, wird ein externer VD verwendet, der nicht auf der Site installiert ist. Dabei wird die explizite Benutzerkonfiguration vor automatischen Methoden berücksichtigt.

Bestimmung des Szenarios

Das erste Ziel der Szenariobestimmung ist natürlich, dass ein Szenario gefunden wird, das eine Kommunikation überhaupt erst ermöglicht. In vielen Fällen kann aber über mehr als ein Szenario kommuniziert werden. Z.B. wenn das offene Szenario möglich ist, können auch andere Szenarien verwendet werden, sofern ein VD gefunden wurde. Allerdings ist die Performance um so besser, je weniger Zwischenstationen verwendet werden. Zweitens muss im halboffenen Szenario eine Verbindung vom Akzeptor zum VD während der ganzen Laufzeit des Akzeptors bestehen bleiben, was im geschlossenen Szenario nicht der Fall ist. Aus diesem Grund ergibt sich als zweites Ziel, das einfachste Szenario zu finden, wobei das offene Szenario das Einfachste ist und das private das Komplexeste. Die Reihenfolge sieht dann folgendermaßen aus:

1. Das offene Szenario
2. Das geschlossene Szenario
3. Das halboffene Szenario
4. Das private Szenario

Wenn kein VD vom Akzeptor gefunden wurde, kommt nur noch das offene Szenario in Frage. Ist das offene Szenario nicht möglich, kann keine Verbindung aufgebaut werden. Bleiben also nur noch die Fälle, in denen ein VD gefunden wurde.

Kann der gefundene VD keine Verbindungen vom Internet annehmen, wird ein weiterer externer VD benötigt, der dem lokalen VD bekannt sein muss. Die Adresse des externen VD muss dem lokalen VD beim Start übergeben werden. Anhand des externen VD kann der lokale VD überprüfen, ob er sich in einer Umgebung befindet, die das private Szenario erfordert, indem der lokale VD versucht, sich über den externen VD zu sich selbst zu verbinden. Gelingt dies nicht, weiß der lokale VD, dass das private Szenario verwendet werden muss. Ansonsten kommen das geschlossene oder halboffene Szenario mit diesem VD in Frage.

Nachdem der Akzeptor einen VD gefunden hat, öffnet er einen Port und verbindet sich zum VD. Falls der VD keine Verbindungen vom Internet annehmen kann, teilt er dem Akzeptor die Adresse des externen VD mit, und dass er sich im privaten Szenario befindet. Wenn der VD nicht vom Internet erreichbar ist, muss er sich auf der selben Site befinden wie der Akzeptor. Somit kann davon ausgegangen werden, dass der Akzeptor auch keine Verbindungen annehmen kann.

Wenn der VD Verbindungen vom Internet annehmen kann, versucht der VD eine Verbindung zum Akzeptor aufzubauen. Gelingt dies nicht, aber der Akzeptor konnte eine Verbindung zum VD aufbauen, geht das halboffene Szenario. Konnte der VD eine Verbindung zum Akzeptor aufbauen, ist das geschlossene Szenario auf jeden Fall möglich, auch wenn das offene Szenario noch nicht vollkommen ausgeschlossen wurde.

Möglicherweise wurde das geschlossene Szenario gewählt, obwohl auch das offene funktioniert. Dies kann überprüft werden, indem entweder der Konnektor beim Verbindungsaufbau zuerst das offene Szenario probiert, aber auf das geschlossene Szenario zurückfällt, falls keine direkte Verbindung möglich ist. Dieser Test kann auch von einem VD durchgeführt werden, der garantiert nicht zu der lokalen Site gehört. Dies beschleunigt den Verbindungsaufbau.

Der Namensdienst

Um den Namensdienst zu finden, werden zwei Fälle unterschieden. Im ersten Fall wurde ein VD gefunden. Da standardmäßig die Registrierung über den VD geleitet wird, reicht es, wenn der VD einen Namensdienst kennt. Wenn kein VD gefunden wurde, kommt sowieso nur noch das offene Szenario in Frage, und dann muss der Akzeptor selbst einen ND finden. Im Prinzip kann dafür dieselbe Liste abgearbeitet werden wie für den VD.

4.1.5 Sicherheitsaspekte

Durch eine Gridverbindung erhält ein Benutzer interaktiven Zugriff auf einen Worker Node. Wenn sie allerdings missbraucht wird oder Unbefugten Zugriff gewährt, bedeutet sie eine Gefährdung. Ein weltweites Grid wie das LHC Computing Grid (LCG) [96], das das Internet als Kommunikationsinfrastruktur verwendet, kann auf eine Absicherung seiner Dienste und Infrastruktur nicht verzichten. Aus diesem Grund wurde die Grid-Sicherheits-Infrastruktur (GSI) entworfen (siehe Kap. 2). Ein wichtiger Aspekt der Gridkommunikation ist daher die Sicherheit sowohl aus Sicht des Diensteanbieters wie auch aus Sicht des Benutzers.

Um die nötigen Sicherheitsmechanismen zu entwickeln, wird zuerst definiert, was Sicherheit bedeutet, und die Angriffsmöglichkeiten auf die Gridverbindung analysiert. Dabei werden drei Gruppen von möglichen Angreifern identifiziert, die unterschiedliche Möglichkeiten haben: Der Benutzer, der Administrator und Dritte (andere Gridnutzer und Außenstehende). Bei der Analyse wird angenommen, dass die GSI Methoden bereitstellt, die für eine direkte Verbindung zwischen zwei Rechnern Authentifizierung, Autorisierung, Integrität und Vertraulichkeit sicherstellen.
Aus Sicht des Benutzers ist die Verbindung sicher, wenn

1. sichergestellt ist, dass nur er selbst, bzw. von ihm spezifizierte Personen sich zu seinem Job verbinden können. Es muss also eine Ende-zu-Ende Authentifizierung und Autorisierung stattfinden, wobei entweder nur der Benutzer selbst autorisiert ist, bzw. der Benutzer spezifizieren kann, welche Personen autorisiert sind.

2. die Integrität der Daten garantiert ist.

3. bei Bedarf die Vertraulichkeit der Daten garantiert ist.

Aus Sicht des Diensteanbieters ist eine Verbindung sicher, wenn

4. der Benutzer keine zusätzlichen Rechte auf den Rechnern erhält, auf die er legalen Zugriff hat.

5. der Benutzer nicht die Möglichkeit erhält, neben seinem Job noch zusätzlich andere Dienste oder andere Rechner von außen zu kontaktieren.

6. Dritte keinen Zugriff auf Rechner der Site erhalten können.

Die Autorisierung darf nicht gelockert oder umgangen werden können. Die aufgezählten Punkte sind Bestandteile der Autorisierung.

Sicherheit im offenen Szenario

Im offenen Szenario wird eine direkte Verbindung zwischen Konnektor und Akzeptor hergestellt. Mithilfe der GSI kann die Authentifizierung, Autorisierung, Integrität und Vertraulichkeit gewährleistet werden, wodurch die Verbindung aus Sicht des Benutzers sicher ist, und auch Außenstehende keine Verbindung zum Gridjob aufbauen können.

Der einzige zusätzliche Dienst, der im offenen Szenario verwendet wird, ist der Namensdienst. Auch die Kommunikation mit dem Namensdienst kann mit GSI abgesichert werden. Alle Einträge im Namensdienst können nur von demjenigen eingesehen werden, der auch autorisiert ist, auf den Job zuzugreifen. Dadurch wird sichergestellt, dass niemand Informationen über Jobs anderer erhält, und niemand Jobeinträge anderer überschreiben kann. Wenn zwei Benutzer denselben Jobnamen verwenden, kann jeder nur die Einträge auslesen, die zu seinem Job gehören.

Der einzige noch verbleibende Punkt ist, dass die Autorisierung des Benutzers nicht aufgeweicht werden darf. In einem offenen Szenario kann jeder Benutzer einen Job ausführen, der Verbindungen annimmt und sich anschließend zu seinem Job verbinden. Interaktivität an sich ist also keine zusätzliche Möglichkeit im offenen Szenario. Eine standardisierte, sichere Kommunikationsschnittstelle erhöht daher die Sicherheit gegenüber individuellen Akzeptoren, die möglicherweise einfacher gehackt werden können.

Sicherheit im geschlossene Szenario

Im geschlossenen Szenario wird die Kommunikation zwischen Konnektor und Akzeptor über den VD geleitet. Der VD kann auf der Site installiert sein, oder es kann ein externer VD verwendet werden. Aus Benutzersicht ist eine Ende-zu-Ende Authentifizierung, Autorisierung, Integrität und Vertraulichkeit notwendig. Eine Ende-zu Ende Absicherung mit GSI, die auf der Gridverbindung aufsetzt, erfüllt die Sicherheitsanforderungen des Benutzers.

Aus Sicht des Administrators ist Sicherheit im geschlossenen Szenario komplexer. Im geschlossenen Szenario gibt es einen zusätzlichen Dienst, der von außen erreichbar sein muss, um Verbindungen zu Akzeptoren hinter der Firewall herzustellen. Die Weiterleitung von Verbindungen muss also eingeschränkt werden können, da sonst jedermann den VD verwenden könnte, um die Firewall zu umgehen und Adressen hinter der Firewall zu kontaktieren. Aus diesem

Grund muss auch jede Verbindung zum VD authentifiziert, autorisiert und integer sein. Dadurch können nur noch diejenigen den VD verwenden, die autorisiert sind, Jobs auf der Site auszuführen. Außerdem können alle Verbindungen zurückverfolgt werden. Der VD kann von dem jeweiligen Zielport ebenfalls eine Identität und eine Liste (access control list, ACL) von Identitäten, die sich zu dem Job verbinden dürfen, verlangen. Der VD kann dann eine Verbindung nur zulassen, wenn die Identität des Konnektors in der ACL enthalten ist. Dadurch kann jeder Benutzer nur zu Jobs verbinden, für die er autorisiert ist, und keine Verbindungen zu anderen offenen Ports aufbauen.

Zusätzliche Sicherheit für den Administrator kann erreicht werden, wenn der Administrator die Bedingungen im VD noch weiter einschränken kann, indem er Regeln definiert, wann Verbindungen zulässig sind, ähnlich wie für Firewalls. Diese Regelungen können mögliche Kombinationen aus Absenderadresse/Identität und Zieladresse/Identität definieren. Z.B. könnte der Administrator die Möglichkeit, Verbindungen zu einem Job aufzubauen, auf dem Submittierer beschränken.

Der Benutzer erhält die Möglichkeit eine interaktive Verbindung zum Job aufzubauen. Dies ist auf jedem Fall eine zusätzliche, aber beabsichtigte Möglichkeit. Erhält er aber dadurch weitere Möglichkeiten Schaden anzurichten? Wer einen Gridjob mit beliebigem eigenem Code ausführen kann, kann letztendlich interaktiv keine zusätzlichen Aktionen durchführen. Die zusätzliche Gefährdung kann also eingeschränkt werden auf die Frage, ob Interaktivität an sich eine Gefahrenquelle für den Administrator darstellt. Diese Frage ist umstritten, aber grundsätzlich gibt es keine neuen Möglichkeiten für den Benutzer, und damit bestehen keine neuen Angriffsmöglichkeiten.

Im geschlossenen Szenario ist also sowohl eine Ende-zu-Ende Sicherung nötig, als auch eine Sicherung der Verbindungen zum VD. Es gibt zwei Möglichkeiten beide Bedingungen zu erfüllen:

- Die Sicherheitsmechanismen werden zweimal verwendet. Zum einen wird jede Verbindung zum VD abgesichert und zweitens eine Ende-zu-Ende Sicherung und Verschlüsselung durchführt.

- Anstatt einer zweiten Sicherung wird Delegation verwendet. Der Verbindungsdienst erhält ein delegiertes Proxy vom Konnektor und verbindet sich damit zum Akzeptor. Der Vorteil dieser Methode ist bessere mögliche Performance bei sehr schnellen Netzwerken (siehe Kap. 7). Allerdings muss der Benutzer dem VD vertrauen, keinen Schaden mit seiner Identität zu verursachen. So ist es beispielsweise dem Administrator oder Nutzer, unter dessen Kennung der VD läuft, möglich, die delegierten Identität zu stehlen und andere (schädliche) Jobs zu starten.

  Falls die Daten verschlüsselt sind, muss der VD die Daten umschlüsseln, also entschlüsseln und mit dem Schlüssel des Empfängers erneut verschlüsseln. Dadurch kann der VD die Daten lesen. Außerdem macht das Umschlüsseln den Performancevorteil wieder zunichte.

Sicherheit im halboffenen Szenario

In einem halboffenen Szenario ist eine Kommunikation immer möglich, auch wenn sie vom Administrator unerwünscht ist. Ein externer VD kann jederzeit

von jemandem eingerichtet werden, wodurch die Frage aus dem geschlossenen Szenario vermieden wird, ob Interaktivität an sich eine Gefährdung darstellt.

Im halboffenen Szenario braucht der VD keine Verbindungen initiieren. Ist der VD außerhalb der Site platziert, verhindert eine Firewall jeden Versuch einer Weiterleitung zu internen Ports. Ist der VD auf der Site eingerichtet, kann die Möglichkeit des VD, Verbindungen zu initiieren, abgeschaltet werden.

Die vom Benutzer benötigt eine Ende-zu-Ende Sicherung kann im halboffenen Szenario durch Delegation des Proxyzertifikates gewährleistet werden, oder dadurch dass eine zweite Sicherung an den Endpunkten stattfindet. Der Nachteil bei der Verwendung von Delegation ist, dass der Benutzer und der Administrator der Worker Node dem VD vertrauen muss, dass der VD die Autorisierung gewährleistet. Falls der Benutzer Vertraulichkeit benötigt, hat Delegation den zusätzlichen Nachteil, dass die Ent- und Verschlüsselung der Daten im VD den Durchsatz durch den VD stark senkt. Der Vorteil ist die bessere Performance, wenn die Kommunikation nicht verschlüsselt wird.

4.2 Konsistenz

Ein Online-Steering-System soll Methoden bereitstellen, die es erlauben, Daten aus einer laufenden Anwendung zu extrahieren, bzw. Daten in einer laufenden Anwendung zu manipulieren. Das bedeutet, dass mindestens ein externer Prozess auf dieselben Daten zugreift wie die Anwendung. Letztendlich führt jede Steering-Aktion zu einer Änderung des Zustandes der Anwendung, und jede Änderung des Zustands korrespondiert zu einer Änderung des Speicherinhaltes der Anwendung. Es existieren also mehrere unabhängige Prozesse, die auf denselben Speicher zugreifen wollen. Daher lässt sich Online-Steering als verteilter gemeinsamer Speicher (DSM) modellieren.

Ein wesentlicher Vorteil gegenüber Client/Server basierten Modellen liegt darin, dass Steering-Systeme, die auf einem DSM-Modell beruhen, intuitiver und einfacher zu benutzen sind. Dies ist analog dazu, dass ein gemeinsamer Speicher intuitiver zu programmieren ist wie Nachrichten-basierte parallele Programme.

Wenn ein Datenobjekt in einer Anwendung durch externe Prozesse modifiziert wird, muss eine Synchronisation mit der Anwendung durchgeführt werden, da ansonsten schwere Fehler in der Anwendung auftreten können. Man stelle sich vor, es wird eine komplexe Formel berechnet, die eine Variable x sowohl im Zähler als auch im Nenner enthält. Wird x geändert, während die Formel berechnet wird, so dass im Zähler ein anderer Wert für x verwendet wird als im Nenner, dann ist das Ergebnis wahrscheinlich falsch. Durch den Eingriff in die Anwendung wurde die Integrität der Daten zerstört. Es werden daher Regeln benötigt, die die Integrität der Daten sicher stellen. Diese Regeln wiederum definieren ein Konsistenzmodell.

In diesem Kapitel werden Bedingungen untersucht, die die Integrität der Anwendung sicherstellen. Anschließend werden passende Konsistenzmodelle definiert, doch zuerst wird der Formalismus eingeführt, der in diesem Kapitel verwendet wird.

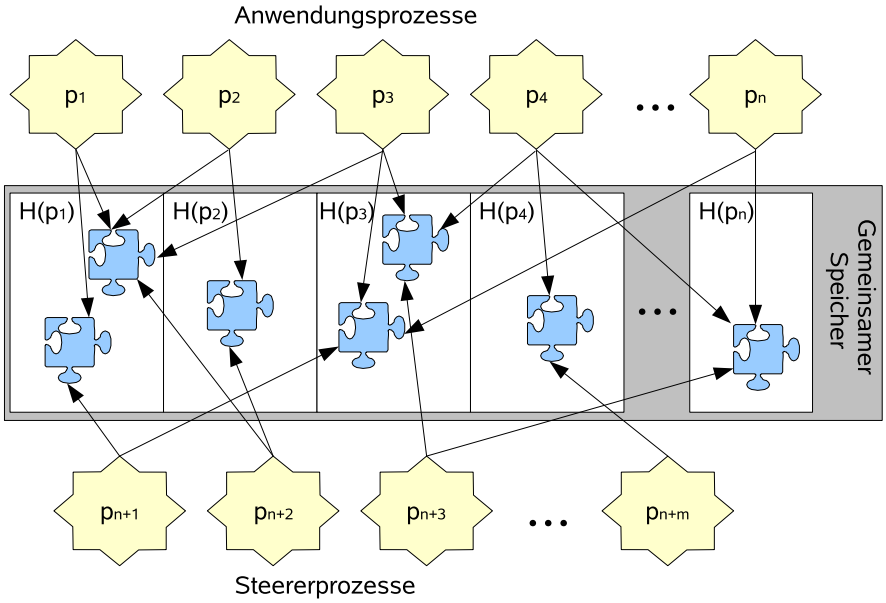


Abbildung 4.5: Modell des verteilten gemeinsamen Speichers

4.2.1 Formalismus zur Beschreibung des Modells

Beim Online-Steering greifen zwar sowohl die Anwendung als auch das Steering-System auf dieselben Daten zu, aber nicht unbedingt mit den gleichen Rechten und nach denselben Regeln. Die Anwendung besteht dabei aus n Prozessen $p_1, ..., p_n$ (siehe Abb. 4.5). Die anwendungsinterne Synchronisation der verschiedenen Anwendungsprozesse kann dabei durch ein beliebiges Verfahren umgesetzt werden, z.B. MPI [161] oder gemeinsamer Speicher. Neben den Anwendungsprozessen, gibt es m Steering-Prozesse $p_{n+1}, ..., p_{n+m}$. Dabei kann es sich um mehrere Prozesse eines Steuerungswerkzeugs handeln oder um mehrere Benutzer in einem kollaborativen Steering-System.

Jedes Datenobjekt o besitzt einen Heimatknoten $H(o) \in \{p_1, ..., p_n\}$, der einer der Anwendungsprozesse ist. Steering-Prozesse werden nicht als Heimatknoten verwendet, da ein Steuerungswerkzeug die Verbindung trennen kann, womit der Heimatknoten für die Anwendung nicht mehr erreichbar wäre. Außerdem initialisiert der Heimatknoten den Wert des Datenobjekts beim Anwendungsstart.

Jedem Anwendungsprozess p_i wird eine logische Uhr T_i zugeordnet, die den Fortschritt des Prozesses anzeigt. T_i wird an sog. *Synchronisationspunkten* inkrementiert. Eine *Epoche* ist das Intervall zwischen zwei aufeinander folgenden Synchronisationspunkten. Zwei Prozesse befinden sich in derselben Epoche, wenn ihre logischen Uhren denselben Wert haben. Jede Epoche wird durch die logische Zeit, die den Anfang der Epoche markiert, eindeutig identifiziert. Durch die Einteilung in Epochen lässt sich der Fortschritt der Prozesse miteinander vergleichen.

Die größte gemeinsame Zeit $T_{min} = \min(T_1, ..., T_n)$ ist die Zeit des Prozesses mit dem geringsten Fortschritt unter allen Anwendungsprozessen. Außerdem

ist $T_{max} = \max(T_1, ..., T_n)$ die Zeit des am weitesten fortgeschrittenen Anwendungsprozesses.

Es gibt drei Arten von Zugriffsoperationen: Die Leseoperation r, die Schreiboperation w und den Synchronisationspunkt s. Eine Lese- und Schreiboperation wird durch das Tupel $r_1 = r(p, x, v)$ bzw. $w_1 = w(p, x, v)$ spezifiziert. Dabei ist p der Prozess, der die Operation ausführt, x die Speicherstelle und v der Wert, der gelesen bzw. geschrieben wird. Wenn für die Benennung einer Operation der Wert ohne Bedeutung ist, kann das dritte Feld auch weggelassen werden. In diesem Fall bedeutet $r_1 = r(p, x)$ bzw. $w_1 = w(p, x)$, dass Prozess p von Speicherstelle x liest, bzw. auf x schreibt. Wenn auch die konkrete Speicherstelle irrelevant ist und daher nicht explizit definiert werden soll, kann auch das zweite Feld fehlen, so dass dann nur noch der Prozess angegeben wird, der die Operation ausführt. Z.B. bedeutet $w_1 = w(p)$, dass w_1 eine Schreiboperation von p ist. Synchronisationspunkte werden mit $s_1 = s(p)$ spezifiziert, wobei p den Prozess definiert, der seine logische Uhr inkrementiert.

Mit $R(p)$ wird die Menge alle Leseoperationen von Prozess p bezeichnet. $R(p, x)$ ist die Menge aller Leseoperationen von p auf eine bestimmte Speicherstelle x. Analog ist $W(p)$ die Menge aller Schreiboperationen von p und $S(p)$ die Menge aller Synchronisationspunkte von p sowie $W(p, x)$ die Menge der Schreiboperationen von p auf eine bestimmte Speicherstelle x.

Jeder Prozess p wird als eine Folge von Speicherzugriffen betrachtet:

$$(o_1, o_2, ...), o_i \in R(p) \cup W(p) \cup S(p). \tag{4.1}$$

Ein Prozess p *sieht* eine Schreiboperation $w_1 = w(q, x, v)$ ab der Stelle im Programmablauf, an der eine, an dieser Stelle ausgeführte, Leseoperation den von w_1 geschriebenen Wert v zurückgeben würde, wenn er nicht durch eine andere Schreiboperation überschrieben wird. Dies impliziert, dass wenn eine Schreiboperation sichtbar geworden ist, sie immer sichtbar bleibt. Eine Leseoperation wird immer den Wert, der von der *zuletzt sichtbar gewordenen* Schreiboperation geschrieben wurde, zurückgeben. Es können mehrere Schreiboperationen in derselben atomaren Operation (z.B. einem Synchronisationspunkt) sichtbar werden, aber nur eine davon kann tatsächlich von einer Leseoperation zurückgegeben werden. Die anderen Schreiboperationen sind quasi sofort überschrieben worden. Sie bleiben aber sichtbar.

Ein Prozess p *sieht* einen Synchronisationspunkt $s_1 \in S(q)$ eines anderen Prozesses q, wenn p die Auswirkungen sieht, die sich für ihn aus s_1 ergeben. Welche Konsequenzen ein Synchronisationspunkt für andere Prozesse hat, hängt vom jeweiligen Konsistenzmodell ab. Z.B. können Schreiboperationen anderer Prozesse sichtbar werden, wenn der Synchronisationspunkt sichtbar wird. Falls $p = q$ ist, sieht p den Synchronisationspunkt s_1 sobald er ihn ausführt. Eine Leseoperation liefert immer den zuletzt sichtbar gewordenen Wert zurück.

Die Programmordnung $<_{PO(p)}$ zweier Operationen $o_1, o_2 \in R(p) \cup W(p) \cup S(p)$ bezeichnet die Reihenfolge der Operationen o_1 und o_2, in der sie vom Prozess p aufgerufen werden. $o_1 <_{PO(p)} o_2$ bedeutet, dass p die Operation o_1 vor o_2 aufruft.

Die Sichtordnung $<_{SO(q)}$ zweier Operationen $o_1 \in W(p) \cup S(p)$, $o_2 \in W(p') \cup S(p')$ bezeichnet die Reihenfolge der Operationen o_1 und o_2, in der sie von einem Prozess q gesehen werden. $o_1 <_{SO(q)} o_2$ bedeutet, dass q die Operation o_1 sieht, bevor er o_2 sieht. Die Sichtordnung kann eine partielle Ordnung sein. Es kann vorkommen, dass eine Speicheroperation von einem Prozess nicht gesehen wird.

4.2.2 Integritätsbedingungen

Bei der Erstellung eines Anwendungsprogramms geht der Programmierer davon aus, dass an bestimmten Stellen des Programmablaufs die Werte einer Speicherstelle einen definierten Zustand haben. Darüber hinaus können innerhalb einer Anwendung zwischen einzelnen Datenobjekten der Anwendung semantische Zusammenhänge bestehen. Wenn dies nicht erfüllt ist, liefert die Anwendung unter Umständen kein korrektes Ergebnis. Um die Integrität der Daten einer Anwendung zu wahren, müssen die semantischen Beziehungen zwischen Datenobjekten berücksichtigt werden, und der Zustand der Daten darf die Korrektheit der Berechnung nicht beeinträchtigen.

Es wird davon ausgegangen, dass die Daten der Anwendung integer sind, wenn kein Steering durchgeführt wird. Wenn die Anwendung um Steering-Möglichkeiten erweitert wird, darf die Integrität der Daten nicht zerstört werden. Es bedeutet, dass die vom Steering-System angezeigten Daten in sich konsistent sein müssen, und die Integrität der Daten in der Anwendung gewahrt bleiben muss, wenn Daten in der Anwendung manipuliert werden. Daraus lassen sich zwei Integritätsbedingungen ableiten, die in diesem Abschnitt anschließend genauer erklärt werden. Die erste Bedingung ist die *Intra-Prozess Bedingung* und die zweite ist die *Inter-Prozess Bedingung.*

Die Intra-Prozess Bedingung

Die Intra-Prozess Bedingung verlangt, dass eine Anwendung während bestimmter Ausführungsintervalle keine Änderungen anderer Steering-Prozesse sehen darf. Ferner dürfen Änderungen, die die Anwendung durchgeführt hat, nur sichtbar werden, wenn die Daten in einem definierten, konsistenten Zustand sind.

Ein Beispiel, bei dem die Einhaltung dieser Bedingung wichtig ist, wurde bereits eingangs von Kap. 4.2 erwähnt. Angenommen es soll eine komplexe Formel berechnet werden, bei der eine Variable an mehreren Stellen der Formel vorkommt, dann sollte diese Variable während der Berechnung der Formel unverändert bleiben. Werden für dieselbe Variable an verschieden Stellen unterschiedliche Werte eingesetzt, kann das Ergebnis nicht korrekt sein. Evtl. können dadurch schwerwiegende Folgefehler ausgelöst werden. Es kann sogar zum Programmabsturz kommen, wenn dadurch z.B. der Nenner Null wird.

Der zweite Teil dieser Bedingung verlangt, dass Änderungen der Anwendung nur dann für andere Prozesse sichtbar werden, wenn sie in einem konsistenten, definierten Zustand sind. Angenommen es werden verschieden Eigenschaften eines Datenobjektes berechnet: Wenn das Objekt nach der Berechnung der ersten Eigenschaften bereits sichtbar ist, während andere Eigenschaften noch von einer anderen Eingabe stammen, ist das angezeigt Resultat möglicherweise inkonsistent und kann zu falschen Entscheidungen führen.

Bei der Berechnung der Kinematik von Galaxien beispielsweise, muss die Anziehungskraft paarweise zwischen allen Sternen berechnet werden. Der Summenvektor der Kräfte, die auf einen Stern wirken, kann in eine vollkommen falsche Richtung zeigen, solange nicht die Kräfte zu allen anderen Sternen vollständig berechnet wurden. Wenn dieser teilweise berechnete Kraftvektor dem Benutzer angezeigt wird, kann dies leicht zu Missverständnissen führen. Außerdem kann der Benutzer nie wissen, ob der angezeigte Vektor einen korrekten Eindruck wiederspiegelt. Falls das Steering-System neue Werte nur sieht, wenn die

Berechnung des Vektors abgeschlossen ist, kann dieses Problem vermieden werden.

Das ATLAS Experiment [1] der Hochenergiephysik ist ein Beispiel für eine Anwendung, für die die Anforderungen der Intra-Prozess Bedingung zutreffen (siehe auch Kap. 6). Beim ATLAS-Experiment werden Kollisionen von Teilchen erzeugt, wobei die einzelnen Ereignisse von einander unabhängig sind. Die Gridjobs bearbeiten eine große Anzahl von Daten solcher Ereignisse, um statistisch relevante Daten zu erhalten. Dabei können mehrere Gridjobs unabhängig voneinander verschiedene Ereignisse untersuchen, deren Ergebnisse anschließend in einer gemeinsamen Statistik zusammengefasst werden. Es ist dabei irrelevant, ob ein Prozess bereits deutlich mehr oder weniger Daten berechnet hat als ein anderer. Für die korrekte Berechnung ist es lediglich wichtig, dass neue Daten nur sichtbar werden, wenn ein Ereignis vollständig berechnet wurde.

Um die Intra-Prozess Bedingung zu erfüllen, dürfen Schreiboperationen der Anwendung erst an bestimmten wohldefinierten Punkten für andere Prozesse sichtbar werden. Ferner dürfen Schreiboperationen, die ein Steering-Prozess ausgeführt hat, nur in wohldefinierten *Synchronisationspunkten* der Anwendung für diese sichtbar werden.

Die Inter-Prozess Bedingung

Die Inter-Prozess Bedingung behandelt Datenabhängigkeiten zwischen Anwendungsprozessen bei parallelen Anwendungen oder zwischen Steering-Prozessen bei kollaborativem Steering. Insbesondere der unterschiedliche Fortschritt zweier Prozesse kann die Beziehungen zwischen Daten in verschiedenen Prozessen gefährden. Die Inter-Prozess Bedingung beinhaltet erstens, dass Schreiboperationen von Steering-Prozessen von allen Anwendungsprozessen im selben logischen Zeitpunkt gesehen werden müssen. Zweitens müssen die Werte aller angezeigten Datenobjekte aus derselben Epoche stammen.

Diese Bedingung trifft beispielsweise auf parallele Anwendungen zu, die die zeitliche Entwicklung eines Systems berechnen, wobei jeder Prozess einen Teil des gesamten Systems bearbeitet. Wird durch einen Benutzer ein Umweltparameter geändert, dann sollte dieser neue Wert bei allen Prozessen in derselben Epoche sichtbar werden.

Oder es soll ein verteilt gespeichertes Objekt angezeigt werden, von dem jeder Prozess einen Teil des Gesamtobjektes berechnet. In diesem Fall muss für alle Teile des Objektes der Wert zum Zeitpunkt T_{min} angezeigt werden, ansonsten wäre das Gesamtobjekt nicht mehr konsistent. Schreiboperationen von Prozessen die bereits weiter fortgeschritten sind, dürfen für das Steuerungswerkzeug nicht sichtbar sein.

Ein möglicher Anwendungsfall könnte eine Simulation zur Klimaentwicklung sein. Hierbei kann das zu untersuchende Gebiet in Teilbereiche aufgeteilt werden, die unterschiedlichen Prozessen zugewiesen werden. Bestimmte Parameter, wie z.B. die erwartete Erderwärmung, oder der prognostizierte Ausstoß an Treibhausgasen können global geändert werden. Dabei ist wichtig, dass diese Parameter an allen Prozessen zum selben Berechnungszeitpunkt geändert werden. Es sollte nicht sein, dass ein Prozess eine höhere Erderwärmung bei der Berechnung des Jahres 2050 übernimmt, während ein anderer Prozess es erst für die Berechnung des Jahres 2500 berücksichtigt. Auch wenn ein Benutzer sich eine Übersicht über das momentan berechnete Klima machen möchte, sollte er

ein Darstellung erhalten, die das Klima in allen Regionen zur selben Zeit zeigt.

Ein anderes mögliches Beispiel ist die Berechnungen zur Ausbreitung von giftigen Gasen nach Unfällen oder Bränden in der Umgebung, wobei sich die Stärke und Richtung von Winden ändern kann. Auch in diesem Fall sollten während der Berechnung eines Zeitschrittes die Parameter konstant bleiben, während sich Parameter, die äußere Einflüsse darstellen, zwischen zwei Zeitschritten verändern können.

4.2.3 Konsistenzmodelle

Um die Integrität der Daten beim Online-Steering sicherzustellen, werden Regeln benötigt, die bestimmen, wann welcher Prozess eine Speicheroperation sehen darf. Die möglichen Reihenfolgen von gegebenen Speicheroperationen, die ein Prozess sehen darf, definieren ein *Konsistenzmodell* [167]. Die Datenintegrität wird also durch die Verwendung eines passenden Konsistenzmodells gesichert. Ziel dieses Kapitels ist es, Konsistenzmodelle zu entwickeln, die den Integritätsanforderungen aus Kap. 4.2.2 gerecht werden.

Es gibt dabei nicht ein Konsistenzmodell, das für alle Anwendungen passend ist, sondern verschiedene Datenobjekte in einer Anwendung können unterschiedliche Konsistenzmodelle erfordern. Welches Konsistenzmodell passend ist, hängt davon ab, welche Integritätsbedingungen für ein Datenobjekt erfüllt werden müssen. Einige Datenobjekte erfordern beide Konsistenzbedingungen, für andere Datenobjekte muss nur die Intra-Prozess Bedingung erfüllt werden. Schließlich gibt es Datenobjekte, die asynchron behandelt werden können, weil sie keine der beiden Bedingungen benötigen. Für jeden dieser Fälle muss das passende Konsistenzmodell gefunden werden. Der Fall, dass nur die Inter-Prozess Bedingung erfüllt werden muss, wird nicht behandelt, da die Inter-Prozess Bedingung die Existenz von Epochen voraussetzt.

Das Konsistenzmodell für Datenobjekte ohne Integritätsbedingungen

Neben Datenobjekten wie Umgebungsparametern oder Zwischenergebnissen, die wahrscheinlich die Intra-Prozess Bedingung oder sogar beide Bedingungen erfordern, existieren möglicherweise Datenobjekte, für die weder die Intra-Prozess noch die Inter-Prozess Bedingung gelten. Beispielsweise können Prozessorauslastung oder andere Ressourcenmonitoringdaten in Entscheidungsprozesse mit einbezogen werden. Diese Daten besitzen keine der beiden Bedingungen und können asynchron ausgetauscht werden. Sie besitzen ein Produzenten-Konsumenten Zugriffsmuster, bei dem ein Prozess (der Produzent) die Daten schreibt, während ein anderer Prozess (der Konsument) ausschließlich lesend auf die Daten zugreift.

Für diese Datenobjekte können Aktualisierungsintervalle, wie sie durch die schwache Konsistenz oder die Zeitplankonsistenz impliziert werden, unpassend sein. Diese Objekte sind per Definition von anderen Daten unabhängig. Daher reicht für solche Daten die PRAM Konsistenz [108] aus, die sicherstellt, dass alle Schreiboperationen eines Prozesses p von allen Prozessen in derselben Reihenfolge gesehen werden.

Definition PRAM Konsistenz: Sei S ein DSM System mit einer Menge P von Prozessen, $p \in P$ und $w_1 \in W(p)$ und $w_2 \in W(p)$ zwei Schreiboperationen

des Prozesses p. Dann ist S PRAM-konsistent falls gilt:

$$\forall q \in P : w_1 <_{PO(p)} w_2 \Longrightarrow w_1 <_{SO(q)} w_2 \tag{4.2}$$

Das Konsistenzmodell für die Intra-Prozess Bedingung

Die Intra-Prozess Bedingung betrachtet nur die Integrität innerhalb eines Prozesses. Falls eine Kommunikation zwischen Anwendungsprozessen notwendig ist, um die Konsistenz zwischen den Anwendungsprozessen zu gewährleisten, muss diese von der Anwendung selbst durchgeführt werden. Ansonsten würde die Anwendung ohne Steering-System nicht korrekt ablaufen. Wie diese anwendungsinterne Konsistenzerhaltung durchgeführt wird, ist für das Steering-System unerheblich. Sie darf durch das Steering-System auch nicht verändert werden.

Einen ähnliche Betrachtungsweise kann auch für Steering-Prozesse verwendet werden, in dem verlangt wird, dass die Steering-Prozesse eines Steuerungswerkzeugs oder mehrere Steuerungswerkzeuge sich untereinander synchronisieren. Das Hauptinteresse der Konsistenz des Steering-Systems ist die Kommunikation zwischen Anwendungsprozessen und Steering-Prozessen.

Allerdings kann beim kollaborativen Steering die Synchronisation unter den Steuerungswerkzeugen auch in das Konsistenzmodell/Konsistenzprotokoll integriert werden.

Man kann also 3 Konsistenzbereiche unterscheiden:

- Die Anwendungsprozesse untereinander: Die Anwendung muss ein internes Konsistenzmodell besitzen, das das Steering nicht verändern darf, es aber auch nicht zu verändern braucht.
- Die Steering-Prozesse untereinander.
- Zwischen der Anwendung auf der einen und dem Steuerungswerkzeug auf der anderen Seite.

Die Intra-Prozess Bedingung erlaubt das Sichtbarwerden von Schreiboperationen eines Prozesses nur an bestimmten Synchronisationspunkten. Außerdem soll ein Prozess Schreiboperationen anderer Prozesse erst bei Synchronisationspunkten sehen. Dies erinnert stark an die schwache Konsistenz [40]. Bei der schwachen Konsistenz müssen Schreiboperationen bis spätestens zum nächsten Synchronisationspunkt sichtbar werden, für die Intra-Prozess Bedingung dagegen müssen Schreiboperationen genau bei Synchronisationspunkten sichtbar werden. Insofern kann ein Konsistenzmodell verwendet werden, das diesem Spezialfall der schwachen Konsistenz entspricht und daher *Spezielle Schwache Konsistenz* genannt wird.

Abgesehen von einer kleinen Toleranz nach Ausführung von Schreiboperationen, sollen alle Prozesse denselben Wert für dieselbe Variable lesen. Als ordnende Komponente sollen die Synchronisationspunkte verwendet werden. Wie bei der schwachen Konsistenz, müssen daher die Synchronisationspunkte von allen Prozessen in derselben Reihenfolge gesehen werden.

Konflikte bei 'gleichzeitigem' Schreiben eines Anwendungsprozesses und eines Steering-Prozesses können gelöst werden, indem dem Steering-Prozess eine höhere Priorität zugewiesen wird. Die Schreiboperation des Prozesses mit der geringeren Priorität wird 'verworfen'. Die Befehle des Benutzers haben dadurch Vorrang vor berechneten Werten der Anwendung. Wenn Konflikte zwischen

Anwendungsprozessen auftreten, müssen diese auch gelöst werden, wenn kein Steering verwendet wird. Diese Konflikte müssen daher von der Anwendung selbst gelöst werden. Ebenso muss ein Steuerungswerkzeug Konflikte zwischen Steering-Prozessen selbst lösen.

Das folgende Beispiel soll die Funktionsweise der Speziellen Schwachen Konsistenz (SSK) verdeutlichen. Eine graphische Illustration des Beispiels ist Abb. 4.6. In diesem Beispiel existieren ein Anwendungsprozess p und ein Steering-Prozess q. Die Prozesse führen folgende Sequenz an Zugriffsoperationen aus:

$$p : (s_2, s_4, w_2, s_5, w_4, s_8)$$
$$q : (w_1, s_1, s_3, s_6, w_3, s_7)$$

Dabei sind s_2, s_4, s_5, s_8 $inS(p)$, $s_1, s_3, s_6, s_7 \in S(q)$, $w_1, w_3 \in W(q, x)$ und $w_2, w_4 \in W(p, x)$.

Zuerst führt q die Schreiboperation $w_1 = w(q, x, 1)$ aus. Von da an sieht q den Wert 1 für x. Danach wird der Synchronisationspunkt s_1 von q ausgeführt. Ab s_1 kann w_1 für andere Prozesse sichtbar werden.

Schreiboperationen anderer Prozesse darf p ab dem folgenden eigenen Synchronisationspunkten sehen. Die Pfeile in der Abbildung sollen anzeigen, welcher Synchronisationspunkt des anderen Prozesses als nächstes in der Sichtordnung folgt. In diesem Beispiel sieht p also bei s_2 den Synchronisationspunkt s_1. Der Prozess p kann demnach w_1 ab s_2 sehen.

Analog gilt dasselbe für w_2. Der Prozess p kann w_2 sofort nach Ausführung sehen. Ab s_5 kann w_2 für andere Prozesse sichtbar werden. Ab dem ersten auf s_5 folgenden eigenen Synchronisationspunkt s_6, kann q auch w_2 sehen.

Die Schreiboperationen $w_3 = w(q, x, 3)$ und $w_4 = w(p, x, 4)$ finden „gleichzeitig“ statt und stehen in einem Konflikt zueinander. Gewünscht ist hier, dass sich die Schreiboperation w_3, die vom Steering-Prozess ausgeführt wird, durchsetzt. Am Ende sollen Leseoperationen beider Prozesse den Wert 3 zurückgeben.

Die Auswirkung dieses Konsistenzmodells ist, dass das Steuerungswerkzeug immer Daten aus der Anwendung zusammen mit Änderungen des Steuerungswerkzeugs anzeigt, die von der Anwendung noch nicht umgesetzt wurden. Dieses Konsistenzverhalten kann als Anzeige der Konfiguration interpretiert werden. Ein Effekt davon ist, dass die angezeigten Daten nicht immer zusammenpassen. So können schon Werte angezeigt werden, die die Anwendung noch gar nicht sieht. Als Folge sind die Auswirkungen der Änderungen noch nicht in den nächsten Zwischenergebnissen der Anwendung enthalten. Da der Benutzer seine Änderungen aber schon länger sieht, erwartet er, dass die Änderungen auch schon in den neuen Zwischenergebnissen enthalten ist.

Als Alternative kann deswegen auch ein Verhalten gewünscht werden, bei dem das Steuerungswerkzeug immer die Werte sieht, wie sie auch die Anwendung sieht. Eigene Änderungen bleiben für das Steuerungswerkzeug unsichtbar, bis sie von der Anwendung gesehen werden. Erst dann werden die neuen Werte auch im Steuerungswerkzeug sichtbar. Dadurch bilden die vom Steuerungswerkzeug gelesenen Daten immer ein in sich konsistentes Ganzes, was insbesondere wichtig sein kann, wenn das Steuerungswerkzeug selbst noch eine Transformation der Daten vornimmt. Auch werden so Änderungen zusammen mit ihren Auswirkungen sichtbar, was eine bessere Ursache-Wirkung-Koppelung erlaubt,

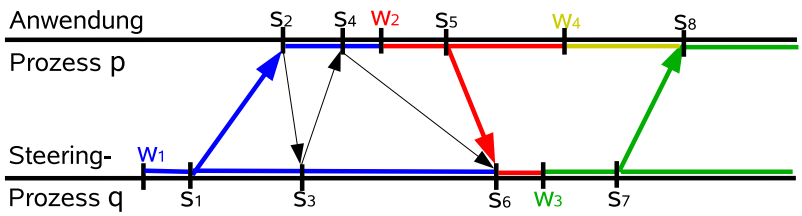


Abbildung 4.6: Beispiel für das Verhalten der Speziellen Schwachen Konsistenz

und der Benutzer sieht, wann seine Änderungen von der Anwendung übernommen werden.

Dieses andere Verhalten erfordert ein anderes Konsistenzmodell, das die Sichtbarkeit der Schreiboperationen des Steuerungswerkzeugs durch das Steuerungswerkzeug verzögert, bis auch die Anwendung die Änderung sieht. Dieses Konsistenzmodell wird daher *Verzögerte Schwache Konsistenz* genannt.

Bei Anwendungen mit mehreren Prozessen sehen nicht alle Prozesse eine Schreiboperation des Steuerungswerkzeugs zur selben Zeit. Es stellt sich die Frage, wann sieht die Anwendung eine Schreiboperation? Sobald ein Prozess die Schreiboperation sieht? Wenn alle Prozesse sie sehen? Wenn die Hälfte der Prozesse sie sieht?

Falls es ausreicht, dass ein Anwendungsprozess eine Schreiboperation eines Steering-Prozesses sieht, sieht auch das Steuerungswerkzeug den selbst geschriebenen Wert auf jeden Fall, sobald die ersten Ergebnisse sichtbar werden, die mit dem neuen Wert berechnet wurden. Andernfalls könnte ein Prozess schon mehrere Epochen mit dem geänderten Wert rechnen, bevor ein weiterer Prozess (oder alle anderen Prozesse) den neuen Wert ebenfalls sehen. Dies würde dann dazu führen, dass Ergebnisses angezeigt werden, die mit dem neuen Wert berechnet wurden, ohne dass das Steuerungswerkzeug selbst den von ihm geschriebenen Wert sieht. Dies ist aber nicht mit der ursprünglichen Absicht vereinbar, so dass es ausreicht, wenn ein Anwendungsprozess eine Schreiboperation eines Steering-Prozesses sieht, damit der Steering-Prozess die Schreiboperation auch sieht.

Evtl. auftretende Konflikte zwischen Schreiboperation von Steering-Prozessen und Anwendungsprozessen können wieder dadurch gelöst werden, dass die Steering-Prozesse gegenüber den Anwendungsprozessen priorisiert werden, so dass Eingaben des Benutzers Vorrang vor Schreiboperationen der Anwendung besitzen.

Da die Steering-Prozesse eigene Schreibzugriffe erst sehen, wenn die Anwendung sie sieht, wird quasi die Sichtordnung der Anwendung verwendet, um Schreibzugriffe zwischen Prozessen zu ordnen. Auf eine global einheitliche Reihenfolge der Synchronisationspunkte kann daher verzichtet werden.

Die Funktionsweise der VSK soll anhand eines Beispiels (siehe auch Abb. 4.7) illustriert werden. In diesem Beispiel existieren ein Anwendungsprozess p und ein Steering-Prozess q, die jeweils folgende Operationen ausführen:

$$p : (w_1, s_1, s_3, s_5, w_4, s_7)$$
$$q : (s_2, w_2, s_4, w_3, s_6, s_8)$$

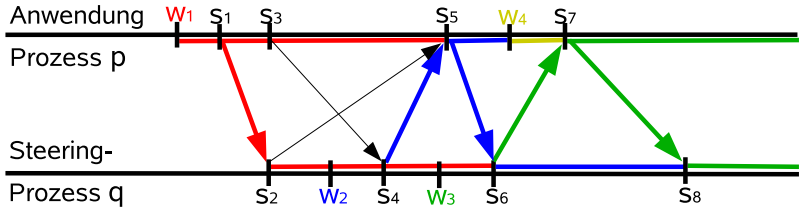


Abbildung 4.7: Beispiel für die Funktionsweise der Verzögerten Schwachen Konsistenz

Dabei sehen beide Prozesse die Synchronisationspunkte in etwas unterschiedlicher Reihenfolge:

$$p : s_1, s_3, s_2, s_4, s_5, s_6, s_7, s_8$$
$$q : s_1, s_2, s_3, s_4, s_5, s_8, s_7, s_8$$

Wenn p die Schreiboperation $w_1 = w(p, x, 1)$ ausführt, sieht p sofort den neu geschriebenen Wert, der ab dem nächsten Synchronisationspunkt s_1 in anderen Steering-Prozessen sichtbar werden darf. Der Prozess q sieht s_1 vor s_2, folglich sieht q ab s_2 die Schreiboperation w_1.

Anders verhält es sich, wenn q die Schreiboperation $w_2 = w(q, x, 2)$ ausführt. Auch nach w_2 würde eine Leseoperation in q immer noch den Wert 1 für x lesen. Ab dem auf w_2 folgenden Synchronisationspunkt s_4 dürfen Anwendungsprozesse w_2 sichtbar machen. Der Prozess p sieht s_4 vor s_5, also sieht p ab s_5 die Schreiboperation w_2. Ein Steering-Prozess sieht eine eigene Schreiboperation, sobald ein Anwendungsprozess die Schreiboperation sieht. Dabei darf auch ein Steering-Prozess Schreiboperationen nur in Synchronisationspunkten sichtbar machen. s_6 ist der erste Synchronisationspunkt von q, den q nach s_5 sieht. Also wird w_2 von q ab s_6 gesehen.

In s_7 gibt es einen Konflikt. Zum einen hat in der vergangenen Epoche p die Schreiboperation w_4 ausgeführt, zum anderen sieht p an dieser Stelle w_3. Da Steering-Prozesse priorisiert werden, sollen beide Prozesse den von q gelesenen Wert sehen.

Interessanterweise ist die sequentielle Konsistenz [103] zu stark für die Intra-Prozess Bedingung, da sie Wertänderungen an jeder Stelle in der Anwendung ermöglicht. In den meisten DSM-Systemen wird die Verwendung und Entwicklung lockerer Konsistenzmodelle durch die bessere Performance motiviert, die sie im Vergleich zu starken Konsistenzmodellen haben. Aber eigentlich möchte der Programmierer, dass sich das Programm verhält, als ob sequentielle Konsistenz [103] verwendet worden wäre. Beim Online-Steering wird die sequentielle Konsistenz nicht verwendet, da sie nicht das gewünschte Verhalten besitzt.

In den folgenden Abschnitten werden sowohl die Spezielle Schwache Konsistenz (SSK) als auch die Verzögerte Schwache Konsistenz (VSK) formal definiert.

Formale Definition der Forderungen der Speziellen Schwachen Konsistenz

In diesem Abschnitt werden die Forderungen der Speziellen Schwachen Konsistenz formal definiert. Im nächsten Abschnitt wird dann die Spezielle Schwache Konsistenz definiert.

Sichtbarkeit eigener Operationen: Wenn durch das Steering-System die Reihenfolge vertauscht wird, in der die Anwendung die eigenen Speicherzugriffe sieht, wird die Korrektheit der Anwendung zerstört.

Forderung 1. *Jeder Prozess p sieht seine eigenen Operationen in der Reihenfolge, in der er sie ausführt. D.h. seien $o_1, o_2 \in W(p) \cup S(p)$ dann muss gelten:*

$$o_1 <_{PO(p)} o_2 \Longrightarrow o_1 <_{SO(p)} o_2. \tag{4.3}$$

Beibehaltung der Reihenfolge von Schreiboperationen: Zweitens soll die Reihenfolge zweier Schreiboperationen eines Prozesses auf dasselbe Datenobjekt in der Sichtordnung anderer Prozesse nicht vertauscht werden.

Forderung 2. *Wenn ein Prozess p erst $w(p, x, 1)$ und dann $w(p, x, 2)$ schreibt, soll kein Prozess erst 2 sehen und später 1 lesen. Es muss also für $w_1, w_2 \in W(p, x)$ gelten:*

$$w_1 <_{PO(p)} w_2 \Longrightarrow w_2 \not<_{SO(q)} w_1. \tag{4.4}$$

Zu beachten ist, dass nicht alle Schreiboperationen für andere Prozesse überhaupt sichtbar werden müssen.

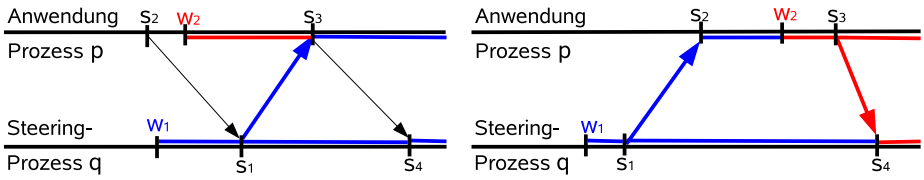
Ordnung des Synchronisationspunkte: Es muss eine eindeutige, globale Ordnung aller Synchronisationspunkte existieren. Folgendes Szenario darf nicht möglich sein: Sei p ein Anwendungsprozess und q ein Steering-Prozess, die folgende Operationen ausführen:

$$p : (s_2, w_2, s_3, r(p, x))$$
$$q : (w_1, s_1, s_4, r(q, x))$$

Dabei ist $s_2, s_3 \in S(p)$, $s_1, s_4 \in S(q)$, $w_1 \in W(q, x)$ und $w_2 \in W(p, x)$. Wenn p und q die Synchronisationspunkte s_1 und s_2 in unterschiedlicher Reihenfolge sehen könnten, könnte die Sichtreihenfolge folgendermaßen aussehen:

$$p : (s_2, w_2, w_1, s_1, s_3, s_4, r(p, x))$$
$$q : (w_1, s_1, s_2, w_2, s_3, s_4, r(q, x))$$

Die Sichtweise für beide Prozesse ist in Abb. 4.8 dargestellt. Als Ergebnis würde p den von w_1 geschriebenen Wert bei $r(p, x)$ lesen, und q würde bei $r(q, x)$ den von w_2 geschriebenen Wert lesen. Finden keine weiteren Schreiboperationen statt, würde der unterschiedliche Wert für den Rest der Laufzeit beider Programme erhalten bleiben. Um dies zu verhindern, ist die global einheitliche Sichtordnung der Synchronisationspunkte notwendig.



(a) Sichtweise von Prozess p (b) Sichtweise von Prozess q

Abbildung 4.8: Beispiel für die Notwendigkeit der globalen Ordnung der Synchronisationspunkte.

Forderung 3. *Alle Prozesse müssen alle Synchronisationspunkte in derselben Reihenfolge sehen. D.h. seien s_1, s_2 zwei Synchronisationspunkte und p, q zwei beliebige Prozesse, dann muss gelten:*

$$s_1 <_{SO(p)} s_2 \iff s_1 <_{SO(q)} s_2. \tag{4.5}$$

Da die Sichtordnung von Synchronisationspunkten für alle Prozesse dieselbe ist, braucht in diesem Konsistenzmodell der Prozess in der Sichtordnung von Synchronisationspunkten nicht mehr angegeben zu werden. Es gilt:

$$s_1 <_{SO(p)} s_2 \iff s_1 <_{SO(q)} s_2 \iff s_1 < s_2$$

Da eine globale Ordnung der Synchronisationspunkte existiert, kann jedem Synchronisationspunkt s_1 eine eindeutige Position $T_g(s_1)$ in dieser Ordnung zugewiesen werden. Dadurch wird eine globale Zeit T_g definiert. Jede Lese- und Schreiboperation $o \in W(p) \cup R(p)$ eines Prozesses p liegt in einer Epoche, die von zwei Synchronisationspunkten $s_1, s_2 \in S(p)$ mit $s_1 <_{PO(p)} o <_{PO(p)} s_2$ begrenzt wird. Damit lassen sich zwei Zeitfunktionen für Lese- und Schreiboperationen definieren, die jeder Operation die Zeit des Synchronisationspunktes zuordnen, der die Epoche beginnt bzw. der sie beendet.

$$T_b(o) = T_g(s_1) \tag{4.6}$$
$$T_e(o) = T_g(s_2). \tag{4.7}$$

Durch die global einheitliche Sichtordnung der Synchronisationspunkte ist auch eindeutig definiert, wann zwei Epochen nebenläufig ausgeführt werden. Seien E_1 und E_2 zwei Epochen. Wenn wenigstens einer der beiden Synchronisationspunkte, die E_1 begrenzen, zwischen den beiden Synchronisationspunkten von E_2 gesehen wird, sind E_1 und E_2 nebenläufig. Z.B. ist in Abb. 4.10 die Epoche zwischen s_1 und s_5 nebenläufig zu den Epochen zwischen s_0 und s_2, s_2 und s_3, s_3 und s_4 sowie s_4 und s_6.

Sichtbarkeit von Schreiboperationen von Steering-Prozessen für Anwendungsprozesse: Wenn ein Anwendungsprozess einen Wert ändert, sollte die Anwendung selbst dafür sorgen, dass die anderen Anwendungsprozesse diesen Wert auch zum richtigen Zeitpunkt sehen. Dies kann vorausgesetzt werden, da die Anwendung sonst nicht richtig funktionieren würde, wenn kein Steering

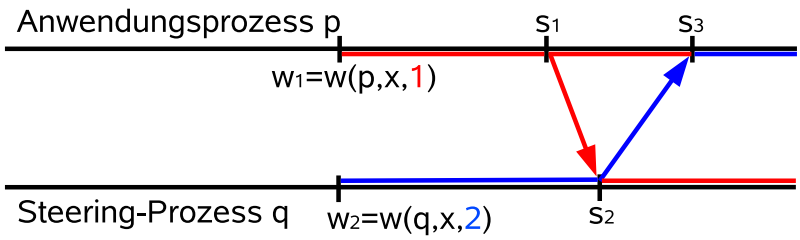


Abbildung 4.9: Fall 1 des Konflikts zwischen Schreiboperationen eines Anwendungsprozesses und eines Steering-Prozesses bei der Speziellen Schwachen Konsistenz

verwendet wird. Das Konsistenzmodell des Steering-Systems muss also nur definieren, wie ein Anwendungsprozess auf Änderungen von Steering-Prozessen reagiert.

Bevor eine Schreiboperation eines Steering-Prozesses q für andere Prozesse sichtbar werden kann, muss q einen Synchronisationspunkt $s_1 \in S(q)$ ausgeführt haben. Der Anwendungsprozess p darf Änderungen nur in Synchronisationspunkten sehen. Also muss auch die Anwendung einen Synchronisationspunkt $s_2 \in S(p)$ ausgeführt haben, bevor der geänderte Wert gelesen werden kann.

Forderung 4. *Für jede Speicherstelle x ist eine Schreiboperation $w_1 \in W(q,x)$ in einer Epoche eines Steering-Prozesses q für einen Anwendungsprozess p genau dann für eine Leseoperation $r_1 \in R(p)$ sichtbar, wenn gilt:*

$$\exists\, s_1 \in S(q) \text{ und } s_2 \in S(p), \text{ so dass } w_1 <_{PO(q)} s_1 < s_2 <_{PO(p)} r_1. \tag{4.8}$$

Eine äquivalente Formulierung dieser Bedingung ist: Für einen Anwendungsprozess p ist w_1 genau dann in einer Leseoperation $r_1 \in R(p)$ sichtbar, wenn

$$T_e(w_1) < T_b(r_1) \tag{4.9}$$

gilt, da dann der auf w_1 folgende Synchronisationspunkt vor dem Beginn der Epoche, die r_1 enthält, liegt.

Sichtbarkeit von Schreiboperationen von Anwendungsprozessen für Steering-Prozesse: Ändert ein Anwendungsprozess p einen Wert x, darf die Schreiboperation erst beim nächsten Synchronisationspunkt $s_1 \in S(p)$ für Steering-Prozesse sichtbar werden. Auch ein Steering-Prozess q darf Änderungen erst in einem eigenen Synchronisationspunkt $s_2 \in S(q)$ sehen. Bis hierher ist Forderung 5 analog zu Forderung 4. Wenn eine Schreiboperation sichtbar wird, muss also mindestens gelten:

$$T_e(w(p)) < T_b(r(q)).$$

Wenn allerdings ein Anwendungsprozess p und ein Steering-Prozess q in nebenläufigen Epochen beide auf x schreiben, kann es zu einem *Konflikt* kommen, der nicht dazu führen darf, dass am Ende beide Prozesse unterschiedliche Werte für dieselbe Speicherstelle lesen (siehe Abb. 4.9 und Abb. 4.10). Es lassen sich dabei zwei Fälle unterscheiden, je nachdem welcher Prozess die Epoche zuerst beendet, in der die Schreiboperation stattgefunden hat.

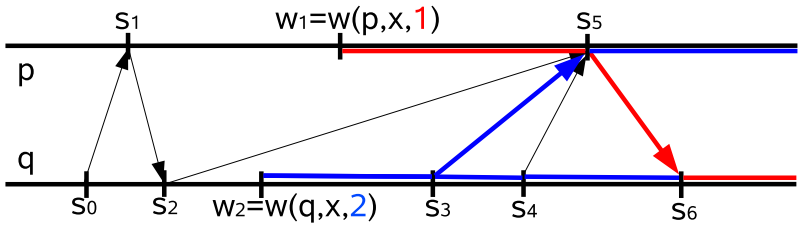


Abbildung 4.10: Fall 2 des Konflikts zwischen Schreiboperationen eines Anwendungsprozesses und eines Steering-Prozesses bei der Speziellen Schwachen Konsistenz

Der erste Fall ist in Abb. 4.9 dargestellt. Hier führt der Steering-Prozess $w_2 = w(q, x, 2)$ aus. Bis zum nächsten Synchronisationspunkt s_2 sieht q auch wie gewünscht den Wert 2. Der Anwendungsprozess sieht nach s_3 auch den Wert 2, was richtig ist. Aber wenn die Forderung 4 analog auf den Steering-Prozess übertragen werden würde, würde q nach s_2 den Wert 1 sehen, so dass p und q verschiedene Werte lesen. Gewünscht ist in diesem Fall, dass q weiterhin den Wert 2 liest. Dies bedeutet, dass q den Wert 1 verwerfen muss.

Der zweite Fall ist in Abb. 4.10 dargestellt. Auch hier führen der Steering-Prozess q und der Anwendungsprozess p jeweils eine Schreiboperation $w_1 \in W(p, x)$ und $w_2 \in W(q, x)$ in nebenläufigen Epochen aus. Nach $s_5 \in S(p)$ wird w_2 von p wie gewünscht gesehen. Aber wenn Forderung 4 analog auf den Steering-Prozess übertragen werden würde, sähe q nach s_6 den Wert, der von w_1 geschrieben wurde.

Nach den bisherigen Forderungen würde dieser Effekt in jeder Epoche auftreten, die zu der von s_1 und s_5 eingegrenzten Epoche nebenläufig ist. Um dies zu verhindern, muss der Steering-Prozess eine Schreiboperation eines Anwendungsprozesses verwerfen, wenn der Steering-Prozess in einer dazu nebenläufigen Epoche einen Wert geschrieben hat.

Forderung 5. *Sei Q die Menge alle Steering-Prozesse und sei $W_S(x)$ die Menge aller Schreiboperationen von Steering-Prozessen auf die Speicherstelle x. Ferner sei p ein Anwendungsprozess und $q \in Q$. Eine Schreiboperation $w_1 \in W(p, x)$ ist bei einer Leseoperation $r_1 \in R(q, x)$ sichtbar, wenn gilt:*

$$\begin{aligned} & T_e(w_1) < T_b(r_1) \\ & \text{und } \nexists\, w_s \in W_S(x) \text{ mit } T_e(w_s) \leq T_b(r_1) \\ & \qquad \text{und } T_b(w_1) < T_e(w_s) \text{ und } T_e(w_1) > T_b(w_s) \end{aligned} \tag{4.10}$$

Bei kollaborativem Steering wird eine Schreiboperation $w_1 \in W(q_1)$ eines Steering-Prozesses q_1 für einen anderen Steering-Prozess q_2 ebenfalls analog zu Forderung 4 sichtbar. Allerdings müssen noch Konflikte zwischen Steering-Prozessen berücksichtigt werden. Hierbei sollen die Steering-Prozesse die Reihenfolge der Schreiboperation gegenüber der Sichtordnung der Anwendungsprozesse nicht vertauschen.

Angenommen q_1 und q_2 führen jeweils eine Schreiboperation $w_1 \in W(q_1, x)$ und $w_2 \in W(q_2, x)$ auf dieselbe Speicherstelle aus. Anschließend führt q_1 den Synchronisationspunkt $s_1 \in S(q_1)$ und q_2 den Synchronisationspunkt $s_2 \in S(q_2)$

aus, wobei $s_1 < s_2$ gilt. In einem Synchronisationspunkt $s_p \in S(p)$ eines Anwendungsprozesses p mit $s_2 < s_p$ muss nun zuerst w_1 sichtbar werden und anschließend von w_2 überschrieben werden.

Daraus folgt, dass q_2 die Schreiboperation w_1 nicht sehen darf, obwohl bei s_2 gilt, dass $w_1 <_{PO(q_1)} s_1 < s_2$ ist. Ein Steering-Prozess q_2 muss also eine Schreiboperation w_1 eines anderen Steering-Prozesses q_1 verwerfen, wenn w_1 und w_2 in nebenläufigen Epochen ausgeführt wurden, und die Epoche, in der w_1 ausgeführt wurde, zuerst beendet wurde.

Forderung 6. *Seien q_1, q_2 zwei verschiedene Steering-Prozesse. Außerdem sei $w_1 \in W(q_1, x)$. Dann sieht q_2 die Schreiboperation w_1 bei einer Leseoperation $r_1 \in R(q_2, x)$ wenn*

$$\begin{aligned} &T_e(w_1) < T_b(r_1) \text{ und} \\ &\nexists\, w_2 \in W(q_2, x) \text{ mit } T_b(w_2) < T_e(w_1) < T_e(w_2) \leq T_b(r_1). \end{aligned} \tag{4.11}$$

Definition des Speziellen Schwachen Konsistenz

Aus den Forderungen der SSK lässt sich ableiten, welchen Wert eine Leseoperation zurückliefern muss. Jedes System ist speziell schwach konsistent, wenn es für jede Leseoperation den durch die SSK definierten Wert zurückgibt und die Synchronisationspunkte von allen Prozessen in derselben Reihenfolge gesehen werden. Bei der Definition der Rückgabewerte der Leseoperation wird unterschieden, ob die Leseoperation von einem Anwendungsprozess oder einem Steering-Prozess ausgeführt wird.

Rückgabewert der Leseoperation eines Anwendungsprozesses: Sei p ein Anwendungsprozess, der eine Leseoperation $r_1 \in R(p, x)$ auf die Speicherstelle x ausführt. Für den Rückgabewert der Leseoperation wird die Schreiboperation gesucht, die zuletzt sichtbar geworden ist. Dazu wird zuerst ein Kandidat unter den Schreiboperationen der Steering-Prozesse und dann ein Kandidat unter den Schreiboperationen der Anwendungsprozesse ermittelt. Anschließend wird bestimmt, welcher der beiden Kandidaten zuletzt sichtbar wurde.

Nach Forderung 4 kann bei r_1 eine Schreiboperation $w \in W(q, x)$ eines Steering-Prozesses q nur sichtbar sein, wenn $T_e(w) < T_b(r_1)$ gilt. Nach Forderung 3 muss von allen Schreiboperationen, für die Forderung 4 erfüllt ist, diejenige als letztes gesehen werden, für die $T_e(w)$ maximal ist. Sei $W_S(x)$ die Menge aller Schreiboperationen von Steering-Prozessen auf die Speicherstelle x. Gesucht wird dann ein

$$w \text{ mit } T_e(w) = max\{T_e(w) | w \in W_S(x) \text{ und } T_e(w) < T_b(r_1)\}. \tag{4.12}$$

Gibt es mehrere Schreiboperationen, für die $T_e(w)$ gleich ist, müssen diese aus derselben Epoche desselben Prozesses stammen. ObdA sei dies der Prozess q_s. Nach Forderung 2 muss von diesen Schreiboperationen diejenige gelesen werden, die in der Programmordnung als letztes kommt. Sei O eine Menge von Speicherzugriffsoperationen, für die eine Ordnung $<_{Ord}$ existiert. Dann sei das Infimum $inf_{Ord}O$ das letzte Element von O nach der Ordnung $<_{Ord}$. Sei Q die Menge aller Steering-Prozesse. Die zuletzt sichtbar gewordene Schreiboperation

w_s eines beliebigen Steering-Prozesses ist somit:

$$w_s = inf_{PO(q_s)}\{w \in W_S(x) \,|\, T_e(w) = max\{T_e(w') \,| \\ w' \in W_S(x) \text{ und } T_e(w') < T_b(r_1)\}\} \tag{4.13}$$

Neben den Steering-Prozessen gibt es außerdem noch die Anwendungsprozesse. Bei der SSK ist die Anwendung selbst dafür zuständig, wann ein Anwendungsprozess die Schreiboperation eines anderen Anwendungsprozesses sieht. Nach Forderung 1 sieht ein Anwendungsprozess eigene Operationen sofort. Somit gibt es für jeden Anwendungsprozess p eine, durch die Anwendung festgelegte, Ordnung $<_{AO(p)}$, in der p Speicherzugriffsoperationen von Anwendungsprozessen sieht. Diese Reihenfolge wird von der SSK übernommen.

Dies setzt voraus, dass das Konsistenzmodell der Anwendung es zulässt, dass Forderung 2 und 3 mit der Anwendungsordnung vereinbar sind. Das bedeutet:

- Die Anwendung muss eine Ausführung erlauben, in der für alle Prozesse die Synchronisationspunkte in der gleichen Reihenfolge gesehen werden können.
- Schreiboperationen desselben Anwendungsprozesses auf dieselbe Speicherstelle müssen von allen anderen Anwendungsprozessen in derselben Reihenfolge gesehen werden.

Sei $W_A(x)$ die Menge aller Schreiboperationen von Anwendungsprozessen auf die Speicherstelle x. Dann ist

$$w_a = inf_{AO(p)}\{w \in W_A(x) | w <_{AO(p)} r_1\} \tag{4.14}$$

die letzte, für p sichtbare Schreiboperation eines Anwendungsprozesses vor r_1.

Es muss jetzt noch entschieden werden, ob p zuerst w_a oder w_s sieht. Sei q der Prozess der w_s ausgeführt hat. Falls gilt

$$\exists\, s_1 \in S(q) \text{ und } s_2 \in S(p), \text{ so dass } w_s <_{PO(p)} s_1 < s_2 <_{AO(q)} w_a, \tag{4.15}$$

muss w_s bei s_2 für q sichtbar sein und w_a somit nach w_s sichtbar werden. Folglich wird r_1 den von w_a geschriebenen Wert lesen. Ansonsten wird w_a zuerst gesehen, und r_1 muss den von w_s geschriebenen Wert zurückgeben.

Rückgabewert der Leseoperation eines Steering-Prozesses: Bei den Steering-Prozessen müssen die Schreiboperationen von anderen Steering-Prozessen, von Anwendungsprozessen und dem lesenden Prozesses unterschieden werden.

Die Schreiboperationen des eigenen Prozesses müssen nach Forderung 1 in der Reihenfolge gesehen werden, in der sie vom Programm ausgeführt werden. Sei q_1 ein Steering-Prozess und $r_1 \in R(q_1, x)$ eine Leseoperation von q_1 auf die Speicherstelle x, dann ist

$$w_q = inf_{PO(q_1)}\{w \in W(q_1, x) | w <_{PO(q_1)} r_1\} \tag{4.16}$$

die letzte Schreiboperation von q_1 vor r_1.

Unter den Schreiboperationen von anderen Prozessen wird die Schreiboperation $w_s \in W_S(x) \setminus W(q_1, x)$ gesucht, für die gilt:

$$T_e(w_s) = max\{T_e(w) | w \in W_S(x) \setminus W(q_1, x) \text{ und } T_e(w) < T_b(r_1)\} \tag{4.17}$$

Wenn es mehrere Schreiboperationen gibt, für die diese Bedingung zutrifft, müssen diese aus derselben Epoche desselben Prozesses q_s stammen. In diesem Fall ist w_s die letzte Schreiboperation dieser Epoche:

$$w_s = inf_{PO(q_s)}\{w \in W_S(x) \setminus W(q_1, x) \,|\, T_e(w) = max\{T_e(w') \,| \\ w' \in W_S(x) \setminus W(q_1, x) \text{ und } T_e(w') < T_b(r_1)\}\} \tag{4.18}$$

Außerdem sei $w_a \in W_A(x)$ die letzte Schreiboperation eines Anwendungsprozesses, für den die Bedingung $T_e(w_a) < T_b(r_1)$ erfüllt ist. Falls es mehrere Schreiboperationen gibt, auf die das zutrifft, wurden allen von demselben Prozess p_a ausgeführt.

$$w_a = inf_{PO(p_a)}\{w \in W_A(x) \,|\, T_e(w) = max\{T_e(w') \,| \\ w' \in W_A(x) \text{ und } T_e(w') < T_b(r_1)\}\} \tag{4.19}$$

Die Schreiboperation w_a eines Anwendungsprozesses wird nach Forderung 5 genau dann verworfen, wenn ein Steering-Prozess in einer parallelen Epoche ebenfalls eine Schreiboperation w_2 ausgeführt hat. In diesem Fall gilt:

$$T_b(w_a) < T_e(w_2) < T_e(w_a) \text{ oder } T_b(w_2) < T_e(w_a) < T_e(w_2). \tag{4.20}$$

Falls allerdings $T_e(w_a) < T_e(w_s)$ oder $T_e(w_a) < T_e(w_q)$ ist, wird w_a nach Forderung 3 und 5 sowieso von w_s oder w_q überschrieben. In diesen Fällen würden natürlich auch alle Schreiboperationen von Anwendungsprozessen überschrieben werden, die vor w_a gesehen werden. Folglich kann w_a nur gelesen werden kann, wenn

$$T_e(w_s) < T_b(w_a) \text{ und } T_e(w_q) < T_b(w_a) \tag{4.21}$$

gilt.

Die Schreiboperation w_s eines anderen Steering-Prozesses kann nach Forderung 6 nur sichtbar sein wenn gilt:

$$T_e(w_s) < T_b(r_1) \text{ und nicht} \\ T_b(w_q) < T_e(w_s) < T_e(w_q) \leq T_b(r_1). \tag{4.22}$$

Falls also $T_e(w_q) > T_e(w_s)$ ist oder $T_b(w_a) > T_e(w_s)$ ist, wird w_s von w_q oder w_a überschrieben. Also kann w_s nur dann von r_1 gelesen werden, wenn

$$T_e(w_q) < T_e(w_s) \text{ und } T_b(w_a) < T_e(w_s) \tag{4.23}$$

ist.

Ansonsten wird r_1 den von w_q geschriebenen Wert zurückgeben. Die Entscheidung wird also danach gefällt, was das Maximum von

$$\{T_b(w_a), T_e(w_s), T_e(w_q)\} \tag{4.24}$$

ist. w_s und w_q werden unterschieden, da w_q auch aus einer noch nicht beendeten Epoche stammen darf, während w_s immer in einer abgeschlossenen Epoche ausgeführt worden sein muss.

Formalisierung der Forderungen der Verzögerten Schwachen Konsistenz

In diesem Abschnitt werden die Forderungen für die Verzögerte Schwache Konsistenz (VSK) formalisiert.

Sichtordnung eigener Operationen der Anwendungsprozesse: Wenn durch das Steering-System die Reihenfolge vertauscht wird, in der die Anwendung die eigenen Speicherzugriffe sieht, wird die Korrektheit der Anwendung zerstört. Die erste Forderung ist also, dass jeder Anwendungsprozess p seine eigenen Operationen in der Reihenfolge sieht, in der er sie ausführt. Im Unterschied zu Forderung 1 der SSK gilt diese Forderung bei der VSK nicht für Steering-Prozesse.

Forderung 1. *Sei p ein Anwendungsprozess und seien $o_1, o_2 \in W(p) \cup S(p)$. Es muss gelten:*

$$o_1 <_{PO(p)} o_2 \Longrightarrow o_1 <_{SO(p)} o_2. \tag{4.25}$$

Reihenfolge von Schreiboperationen eines Prozesses: Zweitens darf die Reihenfolge der Schreiboperationen eines Prozesses auch in der Sichtordnung anderer Prozesse nicht vertauscht werden.

Forderung 2. *Wenn ein Prozess p erst $w(p, x, 1)$ und dann $w(p, x, 2)$ schreibt, darf kein Prozess erst 2 sehen und später 1 lesen. Diese Forderung ist identisch mit Forderung 2 der SSK. Es muss also für $w_1, w_2 \in W(p, x)$ gelten:*

$$w_1 <_{PO(p)} w_2 \Longrightarrow w_2 \not<_{SO(q)} w_1. \tag{4.26}$$

Ordnung der Synchronisationspunkte: Alle Prozesse müssen die Synchronisationspunkte desselben Prozesses p in der Reihenfolge sehen, in der sie von p ausgeführt werden. Die VSK fordert keine eindeutige, totale Ordnung aller Synchronisationspunkte wie die SSK. Lediglich Synchronisationspunkte desselben Prozesses dürfen nicht vertauscht werden. Aber Synchronisationspunkte verschiedener Prozesse können von unterschiedlichen Prozessen in unterschiedlicher Reihenfolge gesehen werden. Synchronisationspunkte müssen sozusagen PRAM-konsistent sein.

Forderung 3. *Seien p, q zwei beliebige Prozesse und $s_1, s_2 \in S(p)$ zwei Synchronisationspunkte. Es muss gelten:*

$$s_1 <_{PO(p)} s_2 \Longleftrightarrow s_1 <_{SO(q)} s_2. \tag{4.27}$$

Zur Definition weiterer Forderungen ist wieder eine Zeitfunktion auf den Synchronisationspunkten hilfreich, die allerdings keine globale Funktion sein kann, sondern in die die Sichtordnung eines Prozesses eingeht: Sei p ein Prozess und s_1 ein Synchronisationspunkt eines beliebigen Prozesses. Dann bezeichnet $T_g(p_1, s_1)$ die Position von s_1 in der Sichtordnung des Prozesses p_1. Analog kann für jede Operation $o_1 \in W(p_2) \cup R(p_2)$ eines Prozesses p_2, die in einer Epoche liegt, die von den Synchronisationspunkten $s_1, s_2 \in S(p_2)$ mit $s_1 <_{PO(p_2}$ $o_1 <_{PO(p_2)} s_2$ begrenzt wird, der Wert $T_b(p_1, o_1)$ und $T_e(p_1, o_1)$ zugeordnet werden:

$$T_b(p_1, o_1) = T_g(p_1, s_1) \tag{4.28}$$

bezeichnet also die Position des Synchronisationspunktes, den p_2 aus Sicht von p_1 vor o_1 ausgeführt hat.

$$T_e(p_1, o_1) = T_g(p_1, s_2) \tag{4.29}$$

bezeichnet dementsprechend die Position des Synchronisationspunktes, den p_2 aus Sicht von p_1 nach o_1 ausgeführt hat.

Sichtbarkeit von Schreiboperationen von Steering-Prozessen für Anwendungsprozesse: Für die Anwendung ändert sich an der Sichtbarkeit von Schreiboperationen von Steering-Prozessen gegenüber der Forderung 4 der SSK im Prinzip nichts. Es muss beachtet werden, dass es keine globale Ordnung der Synchronisationspunkte mehr gibt, sondern die Sichtordnung des Anwendungsprozesses p verwendet werden muss.

Forderung 4. *Sei q ein Steering-Prozess und $w_1 \in W(q)$ eine Schreiboperation von q. Ferner sei p ein Anwendungsprozess. Dann ist w_1 für eine Leseoperation $r_1 \in R(q)$ sichtbar, wenn gilt:*

$$\exists\, s_1 \in S(q) \text{ und } s_2 \in S(p), \text{ so dass } w_1 <_{PO(q)} s_1 <_{SO(p)} s_2 <_{PO(p)} r_1. \quad (4.30)$$

Dazu äquivalent ist die Formulierung, dass w_1 für p bei r_1 sichtbar ist, wenn gilt:

$$T_e(p, w_1) < T_b(p, r_1). \quad (4.31)$$

Da Schreiboperationen von Steering-Prozessen immer bei einem Synchronisationspunkt in einem Anwendungsprozess sichtbar werden, kann jeder Schreiboperation w_1 eines Steering-Prozesses für jeden Anwendungsprozess p genau ein Synchronisationspunkt $s_w(p, w_1)$ zugeordnet werden, in dem w_1 für p sichtbar wird.

Sichtbarkeit von Schreiboperationen von Anwendungsprozessen für Steering-Prozesse: Bisher wurde noch nicht definiert, wann ein Steering-Prozess q die Schreiboperationen eines Anwendungsprozesses p sieht. Das Prinzip ist analog zu Forderung 4.

Forderung 5. *Sei p ein Anwendungsprozess und q ein Steering-Prozess. Ferner sei $w_1 \in W(p)$. Dann ist w_1 für q in $r_1 \in R(q)$ sichtbar, wenn gilt:*

$$\exists\, s_1 \in S(p) \text{ und } s_2 \in S(q), \text{ so dass } w_1 <_{PO(p)} s_1 <_{SO(q)} s_2 <_{PO(q)} r_1, \quad (4.32)$$

oder

$$T_e(q, w_1) < T_b(q, r_1). \quad (4.33)$$

Sichtbarkeit von Schreiboperationen von Steering-Prozessen für Steering-Prozesse: Ein Steering-Prozess sieht Schreiboperationen von Steering-Prozessen einschließlich eigener Schreiboperationen erst, nachdem sie auch von der Anwendung gesehen werden. Wie oben diskutiert, bedeutet dies, dass die Schreiboperation von mindestens einem Anwendungsprozess p gesehen wird. Damit auch Schreiboperationen eines Steering-Prozesses in einem Steering-Prozess q_2 nur an definierten Punkten sichtbar werden und keine Integritätsprobleme im Steuerungswerkzeug auslösen, sollen die verzögerten Schreiboperationen auch erst bei einem Synchronisationspunkt des Steering-Prozesses sichtbar werden.

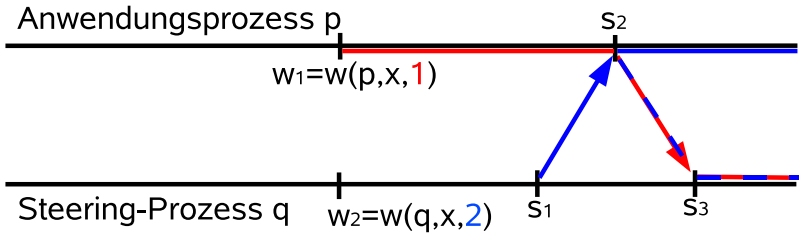


Abbildung 4.11: Konflikt zweier Schreiboperationen zwischen einem Anwendungsprozess und einem Steering-Prozess bei der Verzögerten Schwachen Konsistenz

Forderung 6. *Sei also $w_1 \in W(q_1)$ eine Schreiboperation eines Steering-Prozesses q_1. Dann ist w_1 für einen Steering-Prozess q_2 in $r_1 \in R(q_2)$ sichtbar, wenn gilt:*

$$\begin{aligned} &\exists\, s_1 \in S(q_1), s_3 \in S(q_2) \text{ und } \exists\, p \text{ und } s_2 \in S(p), \text{ so dass} \\ &w_1 <_{PO(q_1)} s_1 <_{SO(p)} s_2 <_{SO(q_2)} s_3 <_{PO(q_2)} r_1. \end{aligned} \tag{4.34}$$

Bei dieser Forderung darf $q_1 = q_2$ sein.

Sei P die Menge der Anwendungsprozesse und $r_1 \in R(q, x)$ eine Leseoperation eines Steering-Prozesses q. Dann ist eine Schreiboperation w_1 eines Steering-Prozesses q_1 sichtbar, wenn gilt:

$$\exists p \in P \text{ so dass } T_e(q, w_1) < T_b(q, r_1) \tag{4.35}$$

Konfliktlösung: Dadurch, dass Schreiboperationen von Steering-Prozessen bei Synchronisationspunkten sichtbar werden, kann es zu Konflikten kommen, wenn bei einem Synchronisationspunkt des Steering-Prozesses q auch noch eine Schreiboperation eines Anwendungsprozesses p sichtbar wird (siehe Abb. 4.11). Da hier die Steering-Prozesse priorisiert werden sollen, muss sichergestellt werden, dass in diesem Fall die Schreiboperation des Steering-Prozesses nach der Schreiboperation des Anwendungsprozesses gesehen wird.

Forderung 7. *Sei p ein Anwendungsprozess und q ein Steering-Prozess. Ferner sei $w_1 \in W(p)$ und $w_2 \in W(q)$. Dann muss gelten:*

$$\begin{aligned} &\text{Falls } \nexists\, s_1 \in S(q), \text{ so dass } w_2 <_{SO(q)} s_1 <_{SO(q)} w_1 \\ &\Longrightarrow w_1 <_{SO(q)} w_2. \end{aligned} \tag{4.36}$$

Beim kollaborativen Steering kann bei der VSK außerdem ein Konflikt auftreten, wenn zwei Steering-Prozesse auf dieselbe Speicherstelle x schreiben. Folgender Ablauf führt z.B. zu einem Konflikt (siehe auch Abb. 4.12): Seinen q_1, q_2 zwei Steering-Prozesse und p_1, p_2 zwei Anwendungsprozesse. Die Prozesse führen jeweils folgende Sequenz an Zugriffsoperationen aus:

$$\begin{aligned} q_1 &: (w_1 = w(q_1, x, 1), s_2, s_7, s_9) \\ q_2 &: (w_2 = w(q_2, x, 2), s_1, s_8, s_{10}) \\ p_1 &: (s_3, s_5) \\ p_2 &: (s_4, s_6) \end{aligned}$$

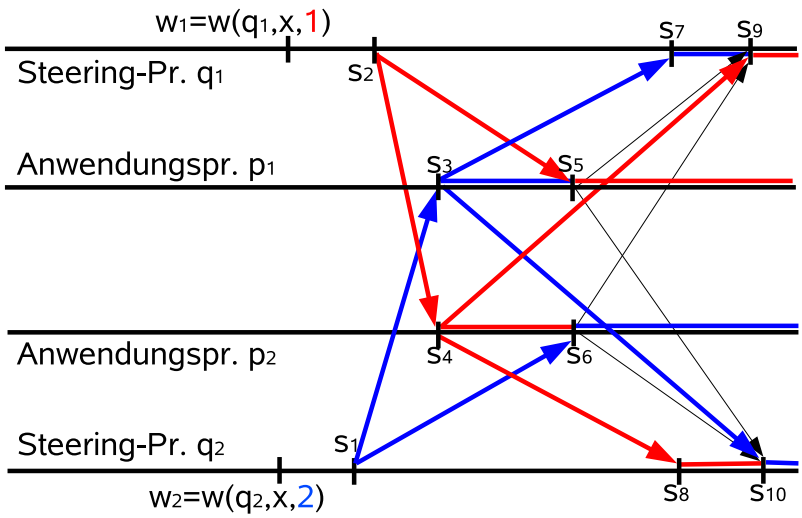


Abbildung 4.12: Konflikt zweier Schreiboperationen bei kollaborativem Steering

Dann wäre folgende Sichtordnung der Synchronisationspunkte möglich, was dazu führen würde, dass verschiedene Prozesse unterschiedliche Werte für dieselbe Variable lesen:

$$
\begin{aligned}
q_1 &: (s_1, s_2, s_3, s_7, s_4, s_5, s_6, s_8, s_9, s_{10}) \\
q_2 &: (s_1, s_2, s_4, s_8, s_3, s_5, s_6, s_7, s_9, s_{10}) \\
p_1 &: (s_1, s_3, s_2, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}) \\
p_2 &: (s_2, s_4, s_1, s_3, s_5, s_6, s_7, s_8, s_9, s_{10})
\end{aligned}
$$

Damit würde p_1 nach s_3 die Schreiboperation w_2 sehen. Anschließend würde dieser Wert bei s_5 von w_1 überschrieben, so dass p_1 nach dieser Sequenz für x den Wert 1 lesen würde. Für p_2 verhält es sich genau anders herum. Nach s_4 würde p_2 die Schreiboperation w_1 sehen, die allerdings in s_6 durch w_2 überschrieben würde.

Auch bei den Steering-Prozessen würden beide Prozesse am Ende unterschiedliche Werte für x sehen. Bei s_7 würde q_1 sehen, dass w_2 von p_1 in s_3 gesehen wird. Somit würde w_2 auch in q_1 sichtbar. Die eigene Schreiboperation hingegen würde erst in s_9 sichtbar, so dass q_1 am Ende der Sequenz den Wert 1 lesen würde. Andererseits würde q_2 am Ende der Sequenz den Wert 2 sehen.

Die vollständige Sichtordnung für die 4 Prozesse wäre also folgende:

$$
\begin{aligned}
q_1 &: (s_1, s_2, s_3, s_7, w_2, s_4, s_5, s_6, s_8, s_9, w_1, s_{10}) \\
q_2 &: (s_1, s_2, s_4, s_8, w_1, s_3, s_5, s_6, s_7, s_9, s_{10}, w_2) \\
p_1 &: (s_1, s_3, w_2, s_2, s_4, s_5, w_1, s_6, s_7, s_8, s_9, s_{10}) \\
p_2 &: (s_2, s_4, w_1, s_1, s_3, s_5, s_6, w_2, s_7, s_8, s_9, s_{10})
\end{aligned}
$$

Die Sichtordnung von w_1 und w_2 wäre also unterschiedlich. Es würde gelten:

$$w_2 <_{SO(q_1)} w_1$$
$$w_2 <_{SO(p_1)} w_1$$
$$w_1 <_{SO(q_2)} w_2$$
$$w_1 <_{SO(p_2)} w_2$$

Dieser Konflikt kann gelöst werden, indem gefordert wird, dass die Schreibzugriffe von Steering-Prozessen speicherkonsistent [66] (siehe auch Kap. 3.2.1) sein müssen. Analog zur SSK müssen Schreiboperationen von Steering-Prozessen auf dieselbe Speicherstelle x von Anwendungsprozessen in derselben Reihenfolge gesehen werden.

Forderung 8. *Seien q_1, q_2 zwei Steering-Prozesse und $w_1 \in W(q_1, x)$ sowie $w_2 \in W(q_2, x)$ zwei Schreiboperationen auf dieselbe Speicherstelle. Dann muss für jedes beliebiges Paar von Anwendungsprozessen p_1, p_2 gelten:*

$$w_1 <_{SO(p_1)} w_2 \iff w_1 <_{SO(p_2)} w_2 \tag{4.37}$$

Definition der Verzögerten Schwachen Konsistenz

Analog zur Definition der SSK wird von den Forderungen der Verzögerten Schwachen Konsistenz abgeleitet, welchen Wert eine Leseoperation zurückliefern muss. Jedes System, das für jede Schreiboperation, die für die VSK definierten Werte zurück liefert, ist verzögert schwach konsistent. Da für die Sichtreihenfolge bei den Steering-Prozessen andere Regeln gelten wie bei den Anwendungsprozessen, wird wieder unterschieden, ob die Leseoperation von einem Anwendungsprozess oder einem Steering-Prozess ausgeführt wird.

Rückgabewert der Leseoperation eines Anwendungsprozesses: Für die Anwendungsprozesse ist die Argumentation analog zur SSK. Es muss lediglich statt der globalen Sichtordnung der Synchronisationspunkte, die Sichtordnung des lesenden Prozesses verwendet werden.

Sei p ein Anwendungsprozess, der eine Leseoperation $r_1 \in R(p, x)$ auf die Speicherstelle x ausführt. Sei $W_S(x)$ die Menge der Schreiboperationen von Steering-Prozessen auf eine Speicherstelle x. Nach Forderung 8 müssen alle Prozesse Schreiboperationen, die von Steering-Prozessen auf x ausgeführt werden, in derselben Reihenfolge sehen. Diese Reihenfolge wird Datenordnung $<_{DO(x)}$ genannt. Wegen Forderung 3 und 4 werden durch $<_{DO(x)}$ gleichzeitig auch die Synchronisationspunkte, die auf eine Schreiboperation in $W_S(x)$ folgen, geordnet. Allerdings führt dies nicht zwangsläufig zu einer einheitlichen Sichtreihenfolge aller Synchronisationspunkte von Steering-Prozessen. Für zwei Schreiboperationen $w_1, w_2 \in W_S(x)$ muss gelten:

$$T_e(p, w_1) < T_e(p, w_2) \Longrightarrow w_1 <_{DO(x)} w_2 \tag{4.38}$$

Nach Forderung 4 sind bei r_1 alle Schreiboperationen $w(q)$ von Steering-Prozessen q sichtbar, für die $T_e(p, w(q, x)) < T_b(p, r_1)$ gilt. Daher muss für die letzte sichtbar gewordene Schreiboperation $w_s \in W_S(x)$ gelten:

$$w_s = inf_{DO(x)}\{w \in W_S(x) | T_e(p, w) < T_b(p, r_1)\}. \tag{4.39}$$

Neben den Steering-Prozessen gibt es außerdem noch die Anwendungsprozesse. Wie bei der SSK ist die Anwendung selbst dafür zuständig, wann ein Anwendungsprozess die Schreiboperation eines anderen Anwendungsprozesses sieht. Analog gibt es für jeden Anwendungsprozess p eine durch die Anwendung festgelegte Ordnung $<_{AO(p)}$, in der p die Speicherzugriffsoperationen von Anwendungsprozessen sieht. Diese Reihenfolge wird von der VSK übernommen.

Dies setzt voraus, dass das Konsistenzmodell der Anwendung es zulässt, dass Forderung 2 und 3 erfüllbar sind. Das bedeutet, das die Anwendungsordnung zwei Schreiboperationen oder Synchronisationspunkte eines Anwendungsprozesses nicht vertauscht.

Sei $W_A(x)$ die Menge aller Schreiboperationen von Anwendungsprozessen auf die Speicherstelle x. Dann ist somit

$$w_a = inf_{AO(p)}\{w \in W_A(x) | w <_{AO(p)} r_1\} \tag{4.40}$$

die letzte für p sichtbare Schreiboperation eines Anwendungsprozesses vor r_1.

Damit kann der Rückgabewert der Leseoperation definiert werden. Falls gilt

$$T_e(p, w_s) < T_e(p, w_a) \tag{4.41}$$

wird w_s zuerst gesehen, und folglich wird r_1 den von w_a geschriebenen Wert lesen. Ansonsten wird w_a zuerst gesehen, und r_1 muss den von w_s geschriebenen Wert zurückgeben.

Rückgabewert der Leseoperation eines Steering-Prozesses: Sei q ein Steering-Prozess, der eine Leseoperation $r_1 \in R(q, x)$ ausführt. Um den Rückgabewert von r_1 zu bestimmen, wird zuerst ein Kandidat aus W_A, der Menge der Schreiboperationen der Anwendungsprozesse, ermittelt. Wenn es mehrere Schreiboperationen gibt, für die T_e gleich ist, dann wurden sie vom selben Prozess p_a ausgeführt. Nach Forderung 5 und 3 muss für w_a gelten:

$$\begin{aligned} w_a = inf_{PO(p_a)}\{w \in W_A(x) \,|\, T_e(w) = max\{T_e(w') \,| \\ w' \in W_A(x) \text{ und } T_e(w') < T_b(r_1)\}\} \end{aligned} \tag{4.42}$$

Eine Schreiboperationen w_2 eines Steering-Prozesses q_1 sieht ein Steering-Prozess q_2 nach Forderung 6, wenn gilt:

$$\exists p \in P \text{ so dass } T_g(q_2, s_w(p, w)) < T_b(q_2, r_1) \tag{4.43}$$

Falls p die Schreiboperation w_2 sieht, muss p wegen Forderung 8 auch jede andere Schreiboperation $w_1 \in W_S(x)$ mit $w_1 <_{DO(x)} w_2$ sehen. Dadurch wird die Erfüllung von Forderung 8 auch für die Steering-Prozesse garantiert. Die letzte Schreiboperation $w_s \in W_S(x)$, die q_2 sieht, ist also:

$$w_s = inf_{DO(x)}\{w \in W_S(x) | \exists p \in P \text{ so dass } T_g(q_2, s_w(p, w)) < T_b(q_2, r_1)\} \tag{4.44}$$

Wegen Forderung 7 wird w_a durch w_s in der Sicht eines Steering-Prozesses q überschrieben, wenn w_a nicht in einer späteren Epoche als w_s für q sichtbar wird. Sei P die Menge aller Anwendungsprozesse. Eine Leseoperation $r_1 \in R(q, x)$ liest dann w_a, wenn gilt:

$$\exists p \in P, s_1 \in S(q) \text{ so dass } T_g(q, s(p, w_s)) < T_g(q, s_1) < T_e(q, w_a). \tag{4.45}$$

In diesem Fall ist $s(p, w_s)$ der Synchronisationspunkt, bei dem w_s für p sichtbar wird. Folglich wird w_s in s_1 für q sichtbar (Forderung 5) und anschließend von w_a überschrieben. Ansonsten wird r_1 den von w_s geschriebenen Wert zurück liefern.

Das Konsistenzmodell für beide Bedingungen

Bei einem Konsistenzmodell, das beide Integritätsbedingungen erfüllen soll, dürfen Schreiboperationen nur in Synchronisationspunkten sichtbar werden. Zusätzlich muss es sicherstellen, dass alle gelesenen Daten aus derselben Epoche stammen, und dass alle Schreiboperationen von Steering-Prozessen in allen Anwendungsprozessen zu Beginn derselben Epoche sichtbar werden.

Zu jedem gegebenen Zeitpunkt kann jede Epoche einer der folgenden drei Gruppen zugeordnet werden:

- Die *Vergangenheit* umfasst jene Epochen, die bereits von allen Anwendungsprozessen abgeschlossen worden sind. Für jede Epoche der Vergangenheit gilt $T < T_{min}$.

- Die *Zukunft* umfasst jene Epochen, die noch kein Anwendungsprozess erreicht hat. Für jede Epoche der Zukunft gilt $T > T_{max}$.

- Die *Gegenwart* umfasst die Epochen, die weder zur Vergangenheit noch zur Zukunft gehören. Die Epochen der Gegenwart liegen also alle im Intervall $T \in [T_{min}, T_{max}]$.

Jeder Schreiboperation $w_1 \in W(p)$ kann ein Zeitstempel $T(w_1)$ zugeordnet werden, in der der geschriebene Wert für andere Prozesse sichtbar wird. Für Anwendungsprozesse entspricht $T(w_1)$ dem Zeitstempel des auf $T(w_1)$ folgenden Synchronisationspunktes. Eine Schreiboperation $w_2 \in W(q)$ eines Steering-Prozesses q muss in der Zukunft stattfinden, damit alle Anwendungsprozesse w_2 in derselben Epoche anwenden können.

Eine Leseoperation $r_1 \in R(p, x)$ eines Anwendungsprozesses p liest den Wert der Schreiboperation mit dem größten Zeitstempel, der kleiner als $T(p)$ ist, wenn p in der aktuellen Epoche noch keine Schreiboperation $w_3 \in W(p, x)$ auf x ausgeführt hat. Ansonsten liefert r_1 den von w_3 geschriebenen Wert. Ein Steering-Prozess liest immer den letzten Wert aus der Vergangenheit. Dieses Konsistenzmodell wird *Zeitplankonsistenz* genannt.

Steering-Prozesse können also immer nur aus der Vergangenheit lesen und in der Zukunft schreiben. Dies hat den Nebeneffekt, dass Änderungen nicht immer sofort sichtbar sind, sondern nach einer Verzögerung, die von der Länge der Gegenwart abhängt.

Formal wird dieser Effekt dadurch erzeugt, dass die Ausführung von Lese- und Schreibzugriffen von Steering-Prozessen umsortiert werden. Es kann sein, dass Schreiboperationen, die in einem Steuerungswerkzeug vor einer Leseoperation vorkommen, erst gesehen werden, nachdem die Leseoperation ausgeführt wurde.

Falls es mehrere Schreiboperationen von unterschiedlichen Prozessen gibt, die auf dasselbe Datenobjekt schreiben und in derselben Epoche sichtbar werden, müssen die Schreiboperationen von Anwendungsprozessen grundsätzlich

vor den Schreiboperationen von Steering-Prozessen sichtbar werden. Schreiboperationen von Steering-Prozessen auf dieselbe Speicherstelle müssen von allen Prozessen in derselben Reihenfolge gesehen werden.

4.3 Protokolle

Im vorhergehenden Kapitel 4.2 wurden verschiedene Konsistenzmodelle für das Online-Steering entwickelt. In diesem Kapitel werden Protokolle entwickelt, die die Konsistenzmodelle implementieren. Vorgestellt werden Protokolle für die PRAM-Konsistenz, die Spezielle Schwache Konsistenz (SSK) und die Verzögerte Schwache Konsistenz (VSK). Für jedes dieser 3 Konsistenzmodelle wird jeweils ein Aktualisierungsprotokoll und ein Invalidierungsprotokoll entwickelt.

Da die Zeitplankonsistenz von einem Algorithmus in CUMULVS [133] abgeleitet wurde, existiert schon ein Protokoll und auch eine Implementierung in CUMULVS. Daher konzentriert sich die weitere Entwicklung in dieser Arbeit auf die Protokolle der Intra-Prozess Bedingung, für die noch keine Protokolle existieren.

In Fällen, in denen nur die Intra-Prozess Bedingung umgesetzt wird, sind die Anwendungsprozesse aus Sicht eines Steering-Prozesses von einander unabhängig. Dies schränkt auch die möglichen Datenabhängigkeiten zwischen den Anwendungsprozessen ein. Für die anwendungsinterne Synchronisation können folgende Fälle unterschieden werden:

- Die Daten liegen in einem (verteilten) gemeinsamen Speicher. Die Anwendung ist in diesem Fall für die Konsistenz des anwendungsinternen gemeinsamen Speichers verantwortlich. Es reicht, an den passenden Stellen durch einen Prozess auf den gemeinsamen Speicher zuzugreifen. Letztendlich handelt es sich dann eher um Threads.

- Jeder Prozess erhält eine eigene Kopie desselben Datenobjektes und diese Daten werden nach der Initialisierung nicht verändert. Allerdings soll das Steering-System diese Daten in allen Prozessen ändern können.

- Jeder Prozess erhält eine eigene Kopie desselben Datenobjektes und verschiedene Prozesse können unterschiedliche Werte für dasselbe Datenobjekt haben. In diesem Fall ist es aber nicht möglich einen Wert anzugeben, der den Wert dieses Datenobjektes in der Anwendung (allen Anwendungsprozessen) darstellt, sondern dieser Wert ist für jeden Prozess gesondert zu betrachten.

Dasselbe gilt auch für die Steering-Prozesse. Ferner wird angenommen, dass die Synchronisation der Anwendungsprozesse untereinander und der Steering-Prozesse untereinander den Anforderungen des jeweiligen Konsistenzmodells entsprechen. Das Steering-System braucht dann keine Kommunikation zwischen den Prozessen einer Prozessgruppe auszuführen. Es findet jeweils nur Kommunikation zwischen dem Steuerungswerkzeug und dem Anwendungsprozess statt. Die Kommunikation des Steuerungswerkzeugs mit einem Anwendungsprozess ist also unabhängig von der Kommunikation des Steuerungswerkzeugs mit einem anderen Anwendungsprozess. Das Steuerungswerkzeug kann für die Kommunikation mit jedem Anwendungsprozess dasselbe Protokoll verwenden. Bei

der Entwicklung der Protokolle für die Intra-Prozess Bedingung und für die PRAM-Konsistenz, wird also davon ausgegangen, dass es genau einen Anwendungsprozess und einen Steering-Prozess gibt.

Das Steuerungswerkzeug kann dann Daten für jeden Prozess einzeln anzeigen. Gibt es bestimmte Parameter, die in allen Prozessen vorkommen, macht es für die Konsistenz keinen Unterschied, ob dieser Wert bei den unterschiedlichen Prozessen mit zeitlicher Verzögerung geschrieben oder gelesen wird. Der Grund dafür ist, dass der Fortschritt der Anwendungsprozesse bei dieser Bedingung unterschiedlich sein kann und unabhängig von einander ist. Schreibt also das Steuerungswerkzeug auf ein Datenobjekt, das in mehreren Prozessen geändert werden soll, kann es einfach für jeden Anwendungsprozess eine unabhängige Schreiboperation ausführen. Sollen Daten aus mehreren Anwendungsprozessen aggregiert angezeigt werden, muss das Steuerungswerkzeug eine geeignete Aggregation aller Daten durchführen. Die Aggregationsmethode ist dabei abhängig von der Anwendung. Mögliche Aggregationsmethoden sind z.B. einen Mittelwert zu bilden oder die Werte aus allen Prozessen aufzusummieren. Die Aggregationsmethoden liegen aber außerhalb des Fokus des Konsistenzprotokolle. In dem beschrieben Schichtenmodell (siehe Abb. 4.1) könnte die Auswertungsschicht eine solche Aggregation durchführen.

Bei der Entwicklung der Protokolle und der Beweisführung wird vorrausgesetzt, dass der Kommunikationskanal die Reihenfolge der Nachrichten beibehält und zuverlässig ist, also keine Nachrichten verloren gehen. Diese Voraussetzungen werden z.B. durch eine TCP-Verbindung erfüllt. Die dritte Annahme ist, dass der verwendete lokale Speicher innerhalb eines Prozesses sequentiell konsistent ist.

Die Protokolle werden durch ihre Reaktionen auf relevante Ereignisse vollständig beschrieben. Zu diesen Ereignissen zählen Zugriffsoperationen auf den DSM, sowie das Empfangen von Nachrichten, die von einem anderen Prozess versandt worden sind. Bei Auftreten eines Ereignisses wird die zugehörige Protokollfunktion ausgeführt:

- `onRead`: Bei einer Leseoperation auf den DSM
- `onWrite`: Bei einer Schreiboperation auf den DSM
- `onSync`: Bei einem Synchronisationspunkt.
- `onRecv`*Nachrichtentyp*: Beim Empfang einer Nachricht des angegebenen Nachrichtentyps.

Grundsätzlich wird die Verwendung mehrerer Threads in einem Prozess unterstützt. Eine der Protokollfunktionen kann nur ausgeführt werden, wenn die vorhergehende Protokollfunktion abgeschlossen wurde. Dies kann bei der Protokollimplementierung durch eine gemeinsam genutzte Sperre erzwungen werden. Eine Ausnahme bilden die Funktionen, die blockieren, bis ein bestimmtes Ereignis eingetroffen ist (z.B. die Antwort empfangen wurde). In der Protokollbeschreibung wird der Befehl `wait` verwendet, wenn an dieser Stelle andere Empfangsfunktionen `onRecv...` bearbeitet werden dürfen. Allerdings dürfen keine anderen `onRead`, `onWrite` oder `onSync` Funktionen ausgeführt werden, wenn eine Prozedur durch `wait` blockiert.

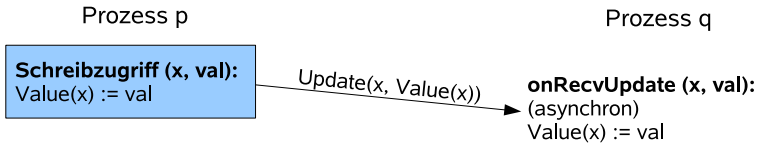


Abbildung 4.13: Funktionsprinzip des Aktualisierungsprotokolls für PRAM Konsistenz

4.3.1 Aktualisierungsprotokoll für die PRAM Konsistenz

Das Aktualisierungsprotokoll reagiert auf eine Schreiboperation, indem es den neu geschriebenen Wert zu den anderen Prozessen versendet. Wird die Aktualisierungsnachricht dort empfangen, wird sofort der Wert der entsprechenden Variable geändert, bevor die nächste Nachricht gelesen wird. Dieses Verfahren ist in Abb. 4.13 dargestellt. Listing 4.1 enthält die Beschreibung des Protokolls in Pseudocode. In jedem Prozess existiert eine lokale Kopie eines Datenobjektes im DSM. Der Wert dieser lokalen Kopie im Pseudocode wird mit `Value(x)` bezeichnet.

Listing 4.1: Aktualisierungsprotokoll für PRAM Konsistenz

```
1 onWrite(x, val):
2   set Value(x) := val
3   send Update (x, Value(x))
4
5 onRecvUpdate(x, val):
6   set Value(x) := val
7
8 onRead(x):
9   return Value(x)
```

Beweis der Korrektheit des Algorithmus: Eine Ausführung der Operation `onRead(x)` liefert den momentanen Wert der lokalen Kopie von x zurück, der entweder durch den letzten vorherigen Aufruf von `onWrite(x)` oder durch `onRecvUpdate(x)` gespeichert wurde.

Angenommen für zwei Schreiboperationen von p gilt: $s_1(p) <_{PO(p)} s_2(p)$. Dann gilt aufgrund der eindeutigen Reihenfolge der Protokollfunktionen auch $s_1(p) <_{SO(p)} s_2(p)$. Der andere Prozess q empfängt die Aktualisierungsnachrichten in der Reihenfolge, in der sie gesendet wurden. Da die erste Nachricht bearbeitet wird, bevor die zweite Nachricht empfangen wird, folgt:

$$onRecvUpdate(s_1(p)) <_{PO(q)} onRecvUpdate(s_2(p)). \tag{4.46}$$

$$\Longrightarrow \quad s_1(p) <_{SO(q)} s_2(p). \tag{4.47}$$

Somit ist das Protokoll PRAM-konsistent.

4.3.2 Invalidierungsprotokoll für die PRAM Konsistenz

Beim Invalidierungsprotokoll wird jedem Datenobjekt x in jedem Prozess p ein Validitätszustand $Status_p(x)$ zugeordnet, der drei mögliche Werte besitzt:

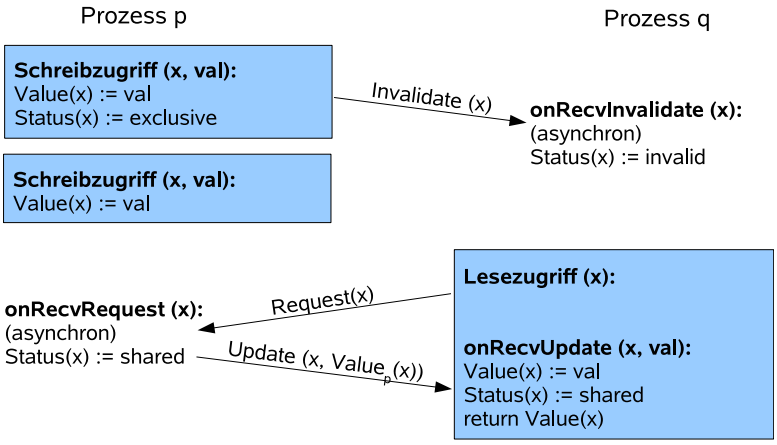


Abbildung 4.14: Funktionsprinzip des Invalidierungsprotokolls für PRAM Konsistenz

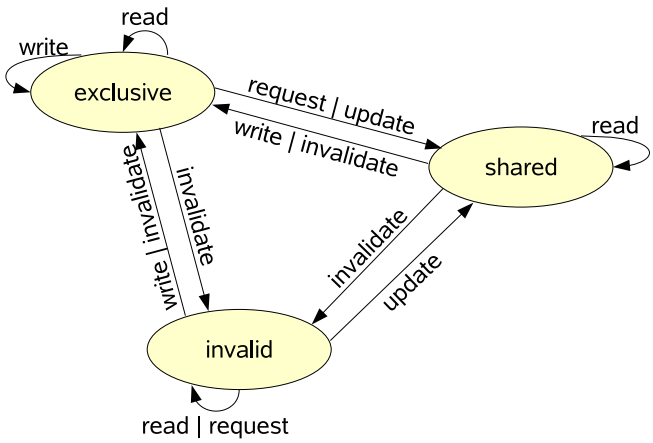


Abbildung 4.15: Zustandsübergangsdiagramm für das Invalidierungsprotokoll der PRAM-Konsistenz

- *invalid* (ungültig): Der lokal gespeicherte Wert von x ist ungültig. Dies impliziert, dass der andere Prozess einen gültigen Wert von x besitzt.
- *exclusive* (allein gültig): Der lokal gespeichert Wert von x ist gültig. Der Wert des anderen Prozesses wurde invalidiert.
- *shared* (beide gültig): Der lokal gespeicherte Wert ist gültig, aber auch der andere Prozess besitzt eine gültige Kopie von x.

Das Funktionsschema des Protokolls ist in Abb. 4.14 dargestellt. In Listing 4.14 ist der Pseudocode aufgelistet, der die Reaktionen des Protokolls auf mögliche Ereignisse zeigt. Das Zustandsübergangsdiagramm für das Protokoll ist in Abb. 4.15 abgebildet.

Listing 4.2: Invalidierungsprotokoll für PRAM Konsistenz

```
1  onWrite(x, val):
2    set Value(x) := val
3    if Status(x) != exclusive
4      set Status(x) := exclusive
5      send Invalidate(x)
6
7  onRead(x):
8    if Status(x) = invalid
9      send Request(x)
10     wait until Status(x) != invalid
11   return Value(x)
12
13 onRecvInvalidate(x):
14   set Status(x) := invalid
15
16 onRecvRequest(x):
17   set Status(x) := shared
18   send Update(x, Value(x))
19
20 onRecvUpdate(x, newValue(x)):
21   set Status(x) := shared
22   set Value(x) := newValue(x)
```

Wenn der Prozess p eine Schreiboperation für ein Datenobjekt x ausführt, wird der neu geschriebene Wert von x lokal gespeichert. Falls $Status_p(x)$ nicht schon *exclusive* ist, wird eine Invalidierungsnachricht (`send Invalidate`, Zeile 5) an Prozess q geschickt und $Status_p(x)$ auf *exclusive* gesetzt (Zeile 4). Empfängt q die `Invalidate`-Nachricht, wird $Status_q(x)$ auf *invalid* gesetzt (Zeile 14). Falls p jetzt weitere Schreiboperationen auf x ausführt, wird einfach der neue Wert von x gespeichert, aber keine neue Nachricht versendet, da $Status_p(x)$ für p schon *exclusive* ist.

Liest q den Wert von x, wird überprüft ob der lokale Wert gültig ist (Zeile 8). Falls er nicht gültig ist, wird eine Aktualisierungsanfrage (`send Request`, Zeile 9) nach p geschickt um den aktuellen Wert zu erhalten. Sobald p die Aktualisierungsanfrage erhält, sendet p den aktuellen Wert von x an q und setzt $Status_p(x)$ auf *shared* (Zeile 17 und 18). Empfängt q den aktuellen Wert, wird

der neue Wert für x lokal gespeichert (Zeile 22) und $Status_q(x)$ ebenfalls auf *shared* gesetzt (Zeile 21). Die Leseoperation von q wartet, bis die Aktualisierung empfangen wurde und gibt anschließend den lokal gespeicherten Wert für x zurück.

Falls q den Wert von x liest und $Status_q(x)$ ist nicht *invalid*, wird einfach sofort der lokal gespeicherte Wert zurück gegeben.

Beweis der Korrektheit des Algorithmus: Um zu zeigen, dass dieses Protokoll die PRAM-Konsistenz erfüllt, muss gezeigt werden, dass gilt (siehe Formel 4.2):

$$w_1 <_{PO(p)} w_2 \Longrightarrow w_1 <_{SO(q)} w_2$$

Angenommen für zwei Schreiboperationen $w_1 = w(p, x, 1)$ und $w_2 = w(p, y, 2)$ gilt: $w_1 <_{PO(p)} w_2$. Der Prozess p sieht die eigenen Schreibzugriffe trivialerweise in der Reihenfolge, in der sie im Programm vorkommen. Interessanter ist die Reihenfolge, die der andere Prozess q sieht.

Es wird gezeigt, dass wenn q w_2 sieht, q in diesem Moment auch schon w_1 gesehen haben muss. Da die Protokollfunktionen beim Empfang von Nachrichten und für die Schreiboperation nicht gleichzeitig ausgeführt werden dürfen, können w_1 und w_2 nicht gleichzeitig gesehen werden. Daraus folgt dann die Behauptung.

Dies lässt sich auch in Formeln ausdrücken: Falls gilt:

$$w_1 <_{PO(p)} w_2 \Longrightarrow \neg(w_2 <_{SO(q)} w_1) \tag{4.48}$$

folgt dann:

$$w_1 <_{PO(p)} w_2 \Longrightarrow w_1 <_{SO(q)} w_2$$

Angenommen eine Leseoperation $r(q, y, 2)$ liefert den von w_2 geschriebenen Wert zurück. Falls vor Beginn der Leseoperation $Status(q, x) = invalid$ war, würde eine folgende Leseoperation $r_2 = r(q, x)$ ebenfalls eine Validierung anfordern. Da w_1 von p vor w_2 ausgeführt wurde, kann die Anfrage keinen Wert zurückgeben, der vor w_1 geschrieben wurde.

Falls aber $Status(q, x) \neq invalid$ war, würde p eine Invalidierung versenden, wenn er w_1 ausführt. Diese Invalidierung wird versendet bevor w_2 ausgeführt wurde und damit bevor p eine Aktualisierung mit dem Wert von w_2 an q schicken konnte. Daraus folgt, dass q die Invalidierung von x erhält, bevor es den neuen Wert von y erhält, somit würde jede folgende Leseoperation von x den Wert von w_1 zurückgeben. Also gilt Formel 4.48 und damit gilt die Behauptung.

4.3.3 Aktualisierungsprotokoll für die Spezielle Schwache Konsistenz

Bei der Speziellen Schwachen Konsistenz wird ein neu geschriebener Wert erst beim nächsten Synchronisationspunkt versendet. Daher wird bei einem Schreibzugriff das entsprechende Datenobjekt x nur als geändert markiert, indem die Variable `Modified(x)` (Zeile 2) auf wahr gesetzt wird. Beim nächsten Synchronisationspunkt werden Aktualisierungen aller geänderten Objekte verschickt

(Zeile 19). Empfängt der andere Prozess q Aktualisierungen, werden sie zwischengespeichert (Zeile 38), bis der nächste Synchronisationspunkt erreicht ist. Im nächsten Synchronisationspunkt werden die Werte aus den Aktualisierungen übernommen (Zeile 16). Anschließend werden Aktualisierungen für die Objekte, die q geändert hat (Zeile 19), verschickt. Das Aktualisierungsprotokoll ist in Abb. 4.16 dargestellt und in Listing 4.3 beschrieben.

Listing 4.3: Aktualisierungsprotokoll für die Spezielle Schwache Konsistenz

```
1  onWrite(x, val):
2    set Modified(x) := true
3    set Value(x) := val
4
5  onRead(x):
6    return Value(x)
7
8  onSync:
9    set NeedSync := true
10   if not HaveSync
11     send RequestSync
12     wait until HaveSync
13   while bufferedUpdates not empty
14     dequeue (x, val) from bufferedUpdates
15     if not (this is Steerer and Modified(x))
16       set Value(x) := val
17       set Modified(x) := false
18   for all x with Modified(x) = true
19     send Update (x, Value(x))
20     set Modified(x) := false
21   set NeedSync := false
22   if SyncRequested
23     set SyncRequested:= false
24     set HaveSync := false
25     send Sync
26
27 onRecvRequestSync:
28   if NeedSync
29     set SyncRequested := true
30   else
31     set HaveSync := false
32     send Sync
33
34 onRecvSync:
35   set HaveSync := true
36
37 onRecvUpdate(x, val):
38   enqueue (x, val) to bufferedUpdates
```

Um die Priorisierung des Steering-Prozesses umzusetzen, werden im Steuerungswerkzeug Aktualisierungen nicht angewendet, wenn dasselbe Datenobjekt in der vergangenen Epoche vom Steuerungswerkzeug verändert wurde (Zeile

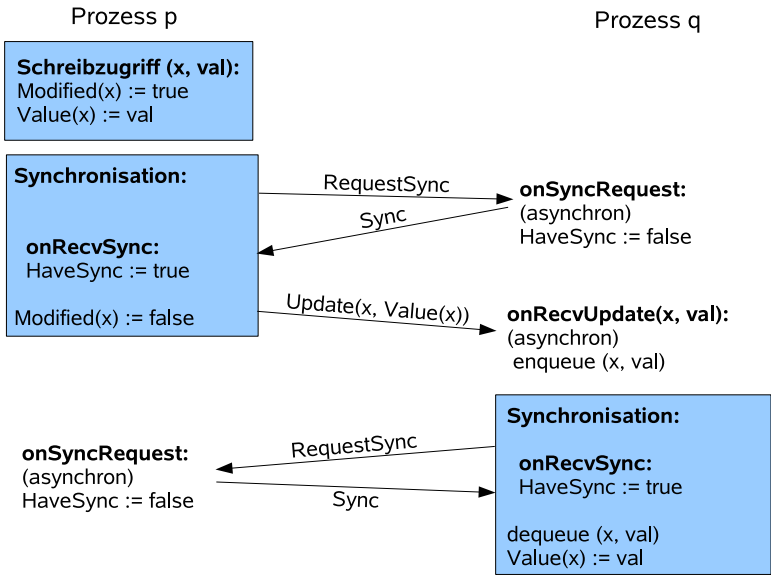


Abbildung 4.16: Funktionsprinzip des Aktualisierungsprotokolls für die Spezielle Schwache Konsistenz

15). Im Job werden von allen Datenobjekten, für die Aktualisierungen empfangen wurden, die Modifikationsmarkierungen gelöscht, da die Änderung ja bereits überschrieben wurde.

Um korrekte Werte zu lesen, müssen die lokalen Kopien korrekt initialisiert werden. Der Job wird dies zwangsweise tun, da er seine eigenen Daten vor dem ersten Lesezugriff initialisieren muss und als Heimatknoten der Daten damit auch den DSM initialisiert. Anders sieht es bei dem Steuerungswerkzeug aus, das sich später mit der Anwendung verbindet und die gültigen aktuellen Werte nicht kennt. Daher muss das Steuerungswerkzeug vor dem ersten Lesezugriff den aktuellen Wert anfordern, da später nur geänderte Werte versendet werden.

Um die sequentielle Ordnung der Synchronisationspunkte zu gewährleisten, wird ein Token-passing Mechanismus verwendet. Er kann mit dem Auffädeln von Perlen auf eine Schnur verglichen werden. Es existiert ein Synchronisationsobjekt (token, das Schnurende), dass immer nur im Besitz eines Prozesses sein kann. Dies wird durch die boolesche Variable `HaveSync` dargestellt, die nur in einem Prozess wahr sein darf. Die Synchronisationspunkte sind dann entsprechend die Perlen. Nur der Prozess, der zurzeit das Schnurende besitzt, kann eine Perle auffädeln. Ansonsten muss er warten, bis er das Schnurende vom anderen Prozess erhält (Zeile 12). Dazu sendet er eine Anforderung an den anderen Prozess (request Sync). Empfängt dieser die Anforderung (onRecvRequestSync), überprüft er, ob er gerade im Begriff ist, ein Synchronisationspunkt auszuführen. Ist dies der Fall, wird der Synchronisationspunkt normal zu Ende ausgeführt und anschließend das “Schnurende” übergeben (Zeile 25). Ansonsten wird des “Schnurende” sofort übergeben (Zeile 32). Zu Beginn besitzt der Anwendungsprozess das “Schnurende”, da ja das Steuerungswerkzeug evtl. gar nicht mit dem Job verbunden ist. Das Zustandsübergangsdiagramm für die Ordnung der

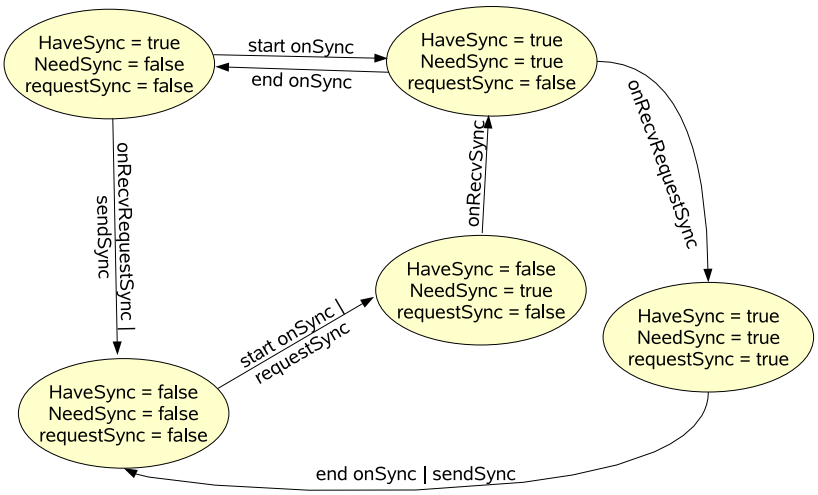


Abbildung 4.17: Zustandsübergangsdiagramm für die Ordnung der Synchronisationspunkte

Synchronisationspunkte ist in Abb. 4.17 dargestellt.

Beweis der Korrektheit des Protokolls: Zum Beweis für die Korrektheit muss gezeigt werden, dass alle Synchronisationspunkte von allen Prozessen in derselben Reihenfolge gesehen werden, und dass jede Leseoperation die durch die SSK definierten Werte zurück liefert.

Beweis für die sequentielle Ordnung der Synchronisationspunkte: Der Kern des Beweises ist zu zeigen, dass folgende Invariante immer erfüllt ist: Es kann höchstens in einem Prozess die Variable `HaveSync` wahr sein.

Ein Synchronisationspunkt kann nur ausgeführt werden, solange `HaveSync` wahr ist, und `HaveSync` kann während der Ausführung auch nicht abgegeben werden.

Bei Jobstart wird `HaveSync` im Job mit wahr initialisiert und beim Start des Steuerungswerkzeugs mit unwahr. Somit ist zu Beginn nur im Anwendungsprozess die Variable `HaveSync` wahr. Wenn ein Prozess eine Nachricht erhält, die ihm das Synchronisationsobjekt übergibt, dann müssen bereits alle Nachrichten vorhergehender Synchronisationspunkte empfangen und bearbeitet worden sein. Daraus folgt, dass ein Prozess alle vorhergehenden Synchronisationspunkte sehen muss, bevor er `HaveSync` auf wahr setzt. Also sehen beide Prozesse alle Synchronisationspunkte in der selben Reihenfolge.

Beweis für den korrekten Rückgabewert der Leseoperation: Nach der Definition der SSK hängt der Wert, den eine Leseoperation r_1 zurückgibt, davon ab, ob sie von einem Anwendungsprozess oder einem Steering-Prozess ausgeführt wurde. Es muss also gezeigt werden:

Für jede Leseoperation r_1, die vom Anwendungsprozess p ausgeführt wird, ist der Rückgabewert bei zwei Prozessen folgendermaßen definiert: Sei w_s die

letzte Schreiboperation des Steering-Prozesses q, die p sieht. Die Definition in Formel 4.13 vereinfacht sich, da nur noch ein Steering-Prozess existiert:

$$w_s = inf_{PO(q)}\{w \in W(q,x) | T_e(w) < T_b(r_1)\} \tag{4.49}$$

Außerdem sei w_a die letzte Schreiboperation von p vor r_1. Auch hier vereinfacht sich die Formel 4.14, da w_a bei nur einem Prozess die letzte Schreiboperation vor der Leseoperation ist:

$$w_a = inf_{PO(p)}\{w \in W(p,x) | w <_{PO(p)} r_1\} \tag{4.50}$$

dann muss r_1 den von w_a geschriebenen Wert lesen, falls gilt (siehe Formel 4.15):

$$T_e(w_s) < T_b(w_a) \tag{4.51}$$

Ansonsten muss r_1 den von w_s geschriebenen Wert zurückgeben.

Für jede Leseoperation r_1 des Steering-Prozesses q, ist der Rückgabewert bei zwei Prozessen folgendermaßen definiert: Sei w_a die letzte Schreiboperation des Anwendungsprozesses p, die q sieht. Da es nur einen Anwendungsprozess gibt, vereinfacht sich Formel 4.19 zu:

$$w_a = inf_{PO(p)}\{w \in W(p,x) | T_e(w) < T_b(r_1)\} \tag{4.52}$$

Außerdem sei w_s die letzte Schreiboperation von q vor r_1 (siehe Formel 4.16):

$$w_s = inf_{PO(q)}\{w \in W(q,x) | w <_{PO(q)} r_1\} \tag{4.53}$$

dann muss r_1 den von w_s geschriebenen Wert lesen, falls gilt (siehe Formel 4.24):

$$T_b(w_a) < T_e(w_s) \tag{4.54}$$

Ansonsten muss r_1 den von w_a geschriebenen Wert zurückgeben.

Dies lässt sich zusammenfassen: Seien p und q die beiden Prozesse und sei $r_1 \in W(p,x)$ eine Leseoperation. Dann sei w_p die letzte Schreiboperation von p vor r_1:

$$w_p = inf_{PO(p)}\{w \in W(p,x) | w <_{PO(p)} r_1\} \tag{4.55}$$

Ferner sei w_q definiert als:

$$w_q = inf_{PO(q)}\{w \in W(q,x) | T_e(w) < T_b(r_1)\} \tag{4.56}$$

Ist p ein Anwendungsprozess, dann gibt r_1 den von w_p geschriebenen Wert zurück, wenn gilt:

$$T_e(w_q) < T_b(w_p) \tag{4.57}$$

Ist p ein Steering-Prozess, dann gibt r_1 den von w_p geschriebenen Wert zurück, wenn gilt:

$$T_b(w_q) < T_e(w_p) \tag{4.58}$$

Ansonsten gibt r_1 den von w_q geschriebenen Wert zurück.

Um zu zeigen, dass jede Leseoperation den korrekten Wert zurückgibt, wird zuerst gezeigt, dass eine Leseoperation den richtigen Wert zurückgibt, wenn eine Invariante erfüllt ist. Anschließend wird eine vollständige Induktion über alle Speicheroperationen eines Prozesses, die die Invariante verändern können, ausgeführt und gezeigt, dass die Invariante nach jeder Speicheroperation erfüllt ist.

Die Invariante. Die Invariante ist: `Value(x)` enthält zu jeder Zeit den Wert, den eine zu dieser Zeit ausgeführte Leseoperation $r_1 \in R(p, x)$ nach der Definition der SSK zurückgeben müsste. Zu zeigen, dass eine Leseoperation den korrekten Wert liest, ist trivial, da `onRead` lediglich `Value(x)` zurückgibt.

Bleibt mit Hilfe der vollständigen Induktion über die Speicheroperationen von p, die die Invariante verändern können, zu zeigen, dass die Invariante erfüllt ist.

Der Induktionsanfang ist die korrekte Initialisierung. Beim Jobstart muss der Job seinen Speicher initialisieren. Dafür muss er eine Schreiboperation auf jede verwendete Speicherstelle x ausführen. Dabei wird `Value(x)` gesetzt. Somit ist auf Jobseite der Speicher korrekt initialisiert. Wenn der Steering-Prozess gestartet wird und sich mit dem Job verbindet, muss er zuerst die aktuellen Werte der Anwendung erfragen, um seinen Speicher zu initialisieren. Damit ist die Invariante erfüllt.

Beim Induktionsschritt wird gezeigt, dass wenn die Invariante erfüllt ist und p eine Leseoperation, Schreiboperation oder einen Synchronisationspunkt ausführt, danach die Invariante immer noch erfüllt ist. Dabei sei w_q die letzte Schreiboperation von q, die zum Zeitpunkt der Ausführung der Speicheroperation für p sichtbar ist.

- **Leseoperation:** `Value(x)` wird nicht verändert. Also bleibt die Invariante wahr.
- **Schreiboperation:** Wenn eine Schreiboperation $w_1 = w(p, x, v)$ ausgeführt wird, wird `Value(x)` auf den neu geschriebenen Wert v gesetzt (siehe Zeile 3). Außerdem gilt jetzt:

  $$w_1 = inf_{PO(p)} W(p, x) \tag{4.59}$$

  und

  $$T_b(w_1) > T_e(w_q), \tag{4.60}$$

  da w_q sonst noch nicht sichtbar sein dürfte (siehe Forderung 4 der SSK). Folglich müssen Leseoperationen von p von nun an v zurückgeben. Somit bleibt die Invariante wahr.
- **Synchronisationspunkt:** Sei s_1 der aktuell ausgeführte Synchronisationspunkt von p und sei $s_0 \in S(p)$ der zuletzt vor s_1 ausgeführte Synchronisationspunkt von p. Ferner sei w_q die letzte Schreiboperation von q, die für p sichtbar ist. Es können nun zwei Fälle unterschieden werden: q hat zwischen s_0 und s_1 einen Synchronisationspunkt ausgeführt oder q hat zwischen s_0 und s_1 keinen Synchronisationspunkt ausgeführt.

  Angenommen q hat zwischen s_0 und s_1 keinen Synchronisationspunkt ausgeführt: In diesem Fall kann es keine Schreiboperation $w_2 \in W(q)$ von q geben, mit $T_e(w_2) < T_g(s_1)$ und $w_q <_{PO(q)} w_2$. Somit bleibt der vorher sichtbare Wert gültig. Da q keinen Synchronisationspunkt ausgeführt hat, wurde auch kein `send Update` ausgeführt. Es wurden folglich von p auch keine Updates empfangen. Also bleibt `Value(x)` unverändert. In diesem Fall ist das Protokoll also korrekt.

Angenommen q hat zwischen s_0 und s_1 einen Synchronisationspunkt s_q ausgeführt: In diesem Fall muss gelten:

$$s_0 < s_q < s_1 \tag{4.61}$$

Aufgrund der unterschiedlichen Regeln für Steering-Prozesse und Anwendungsprozesse muss hier unterschieden werden, ob p ein Anwendungsprozess oder ein Steering-Prozess ist. Da z.B. `Modified(x)`, `Value(x)` oder `HaveSync` in jedem Prozess einmal vorkommen, wird folgende Schreibweise eingeführt, um deutlich zu machen, in welchem Prozess welche Variable gemeint ist: $Name_{Prozess}(x)$ bezeichnet die Variable `Name` im angegebenen Prozess. Z.b. ist $Value_p(x)$ die Variable `Value(x)` im Prozess p.

- Sei p zunächst der Anwendungsprozess. Folglich muss q ein Steering-Prozess sein. Falls q eine Schreiboperation w_1 in der von s_q abgeschlossenen Epoche ausgeführt hat, dann hat diese Schreiboperation $Modified_q(x)$ auf wahr gesetzt und den neu geschriebenen Wert in $Value_q(x)$ gespeichert (Zeile 2 und 3).

  In s_q wird für jede modifizierte Speicherstelle x der zuletzt geschriebene Wert in einer Update-Nachricht versendet (Zeile 19). O.b.d.A ist dies der durch w_1 geschriebene Wert. Für w_1 gilt dann also:

  $$w_1 = inf_{PO(q)}\{w \in W(q,x) | w <_{PO(q)} s_q\} \tag{4.62}$$

  und $T_e(w_1) = T_g(s_q)$.

  In s_q wird anschließend $Modified_q(x)$ auf unwahr gesetzt. Also kann $Modified_q(x)$ nur wahr sein, wenn x in der vergangenen Epoche verändert wurde. Es kann also keine Update-Nachricht für eine Speicherstelle versendet werden, wenn diese in der vergangenen Epoche nicht verändert wurde.

  Bevor p seinen Synchronisationspunkt ausführen kann, muss er alle Nachrichten von s_q erhalten haben, also auch die `Update`-Nachricht für x. Beim Empfang der `Update`-Nachricht wird diese in die Queue eingereiht. Die Bedingung in Zeile 15 ist immer wahr, da p nicht der Steering-Prozess ist. Also wird der neue Wert der `Update`-Nachricht auch in $Value_p(x)$ gesetzt.

  Dies entspricht der Definition der SSK, da für jede Schreiboperation w_p der Epoche zwischen s_0 und s_1 gilt:

  $$T_b(w_p) = T_g(s_0) < T_g(s_q) = T_e(w_1). \tag{4.63}$$

  Gab es mehrere Synchronisationspunkte von q zwischen s_0 und s_1, so wird durch die Verwendung der Queue sichergestellt, dass die Nachrichten der Synchronisationspunkte in der Reihenfolge bearbeitet werden, in der sie versendet wurden. Somit ist die Invariante am Ende des Synchronisationspunkts wahr.

- Sei p der Steering-Prozess und q der Anwendungsprozess. Sei $w_2 \in W(q,x)$ eine Schreiboperation von q, die in der Epoche, die von s_q beendet wurde, ausgeführt wurde. Dann muss bei Beginn des Synchronisationspunkts $Modified_q(x)$ wahr sein.

Die einzige Möglichkeit, dass $Modified_q(x)$ unwahr werden kann, bevor eine `Update`-Nachricht verschickt wird, ist, wenn in s_q eine `Update`-Nachricht für x vorliegt (Zeile 17). In diesem Fall muss eine Schreiboperation $w_1 \in W(p,x)$ existieren mit $T_e(w_1) > T_b(w_2)$. Dann aber muss p den von w_1 geschriebenen Wert lesen (siehe Formel 4.58). Daher braucht in diesem Fall keine `Update`-Nachricht in s_q verschickt zu werden. Wenn also die Möglichkeit besteht, dass p nach der Definition der SSK w_2 lesen könnte, wird eine `Update`-Nachricht in s_q verschickt.

Angenommen es liegt keine `Update`-Nachricht für x in s_q vor. Dann wird in s_q eine `Update`-Nachricht für x an p verschickt (Zeile 19). Diese Nachricht enthält den Wert der letzten Schreiboperation von q auf x. Sei w_2 die letzte Schreiboperation. Es gilt also:

$$w_2 = inf_{PO(q)}\{w \in W(q,x) | w <_{PO(q)} s_q\} \tag{4.64}$$

Alle `Update`-Nachrichten von s_q müssen von p empfangen worden sein, bevor $HaveSync_p$ wahr werden kann. Das bedeutet auch, dass die `Update`-Nachricht durch die `onRecvUpdate` Prozedur in die Queue von p eingestellt wurde.

Die Bedingung in Zeile 15 ist erfüllt, wenn keine Schreiboperation $w_3 \in W(p,x)$ existiert, die in der letzten Epoche von p ausgeführt wurde.

Ist die Bedingung in Zeile 15 nicht erfüllt, gilt:

$$\exists w_3 : T_e(w_3) = T_g(s_1) > T_g(s_q) > T_b(w_2). \tag{4.65}$$

Also muss eine folgende Leseoperation den von w_3 geschriebenen Wert lesen. Nach der Induktionsvoraussetzung muss dieser bereits in $Value_p(x)$ gesetzt sein, da p die Schreiboperation w_3 ab dem Zeitpunkt der Ausführung von w_3 sehen muss. Also darf $Value_p(x)$ in diesem Fall nicht überschrieben werden. In diesem Fall ist das Protokoll also korrekt.

Ist die Bedingung in Zeile 15 erfüllt, bedeutet das, dass für die letzte Schreiboperation $w_p \in W(p,x)$ gilt:

$$T_e(w_p) < T_b(w_2) \tag{4.66}$$

Wäre $T_b(w_2) < T_e(w_p) < T_g(s_1)$, dann müsste in s_q eine `Update`-Nachricht von w_p dafür gesorgt haben, dass in Zeile 17 $Modified_q(x)$ falsch wurde, und somit wäre die `Update`-Nachricht nie gesendet worden. Wäre $T_e(w_p) = T_g(s_1)$, dann wäre die Bedingung in Zeile 15 nicht erfüllt.

Folglich muss p in diesem Fall in einer folgenden Leseoperation den von w_2 geschriebenen Wert zurück geben (siehe Formel 4.58). Dies wird von dem Protokoll auch umgesetzt, da $Value_p(x)$ in Zeile 16 auf den entsprechenden Wert gesetzt wird. Damit ist auch in diesem Fall die Invariante wahr.

Also bleibt sowohl beim Anwendungsprozess als auch beim Steering-Prozess nach einem Synchronisationspunkt die Invariante wahr.

Da die Invariante in p nach jeder Speicheroperation erfüllt ist, wird p immer den von der SSK definierten Wert lesen. Folglich ist das Protokoll korrekt.

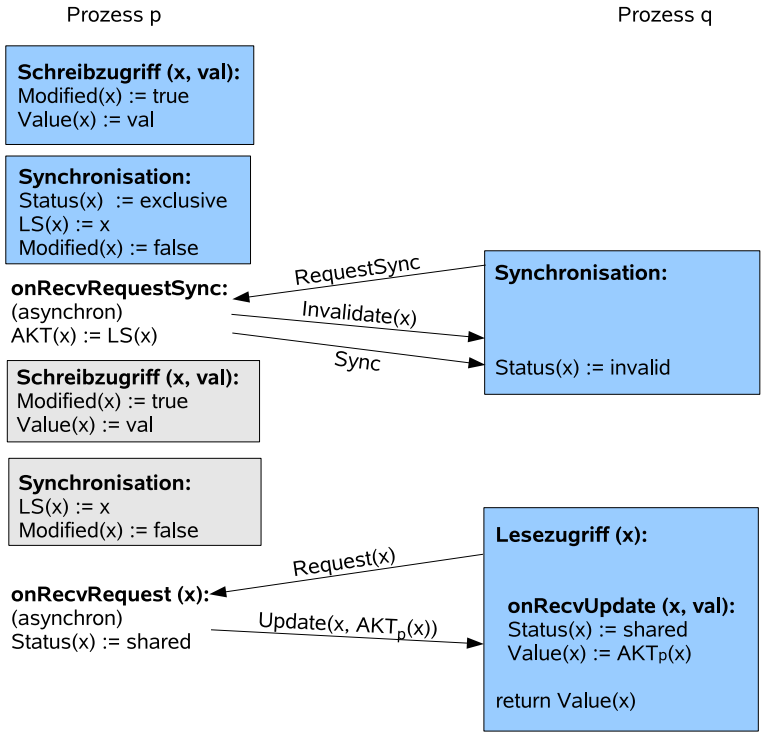


Abbildung 4.18: Funktionsprinzip des Invalidierungsprotokolls für die Spezielle Schwache Konsistenz

4.3.4 Invalidierungsprotokoll für die Spezielle Schwache Konsistenz

Das Invalidierungsprotokoll für die Spezielle Schwache Konsistenz verwendet denselben Mechanismus wie das Aktualisierungsprotokoll, um die sequentielle Ordnung der Synchronisationspunkte zu gewährleisten. Anstatt bei Schreiboperationen Aktualisierungen (Update-Nachrichten) zu versenden, werden Invalidierungen verschickt. Wenn ein Prozess q ein invalidiertes Datenobjekt x liest, muss also der entsprechende Wert erst vom anderen Prozess p angefordert werden. Allerdings kann p den Wert von x mittlerweile überschrieben haben, muss aber noch den Wert von x zurückgeben, den x zum Zeitpunkt des letzten Synchronisationspunkts $s(p)$ hatte, der vor einem Synchronisationspunkt von q liegt. Also muss jeder Prozess p den Wert $AKT_p(x)$ speichern, der für die Aktualisierung notwendig ist.

Der Wert von $AKT_p(x)$ stammt immer aus dem letzten selbst geschriebenen Wert, den x in einem Synchronisationspunkt von p hatte. Allerdings kann p weitere Synchronisationspunkte bearbeitet haben, nachdem q seinen letzten Synchronisationspunkt ausgeführt hat. Wenn jetzt q einen Wert von p anfordert, muss p einen Wert zurück liefern, der mehrere Synchronisationspunkte zurück liegt. Führt p anschließend einen Synchronisationspunkt aus, muss p den Wert von $AKT_p(x)$ aktualisieren. Der neue Wert von $AKT_p(x)$ muss auf den Wert

von x im letzten Synchronisationspunkt gesetzt werden.

Also muss jeder Prozess p zwei Werte nachhalten:

- Wert von x beim letzten Synchronisationspunkt: $LS_p(x)$
- Wert der bei Aktualisierungen zurückgeliefert werden muss: $AKT_p(x)$. Dies entspricht dem Wert, den x beim letzten Synchronisationspunkt von p vor dem letzten Synchronisationspunkt von q hatte.

Das Protokoll ist in Listing 4.4 beschrieben und in Abb. 4.18 illustriert. Initialisiert werden alle Datenobjekte im Anwendungsprozess durch den Job. Im Steuerungswerkzeug wird der Status (`Status(x)`) aller Datenobjekte anfangs auf *invalid* gesetzt. Außerdem führen beide Seiten mindestens einen Synchronisationspunkt aus, bevor sie lesend zugreifen. Der Job muss dabei seinen ersten Synchronisationspunkt ausführen, bevor das Steuerungswerkzeug seinen Synchronisationspunkt ausführt.

Es mag etwas seltsam anmuten, dass die `onWrite` Prozedur `Status(x)` auf *shared* setzt, wenn er bisher *invalid* war (Zeile 4). Dies liegt darin begründet, dass der schreibende Prozess p ab jetzt den neu geschriebenen Wert lesen soll, also nicht mehr *invalid* sein darf. Besitzt aber der andere Prozess q bisher einen gültigen Wert, wird dieser erst invalidiert, wenn die entsprechenden Synchronisationspunkte ausgeführt wurden. Im Moment besitzen dann also beide Prozesse einen gültigen Wert (der nicht gleich sein muss). Dies wird durch den Status *shared* ausgedrückt.

Der Status wird erst dann auf *exclusive* gesetzt, wenn die Invalidierung wirklich verschickt wird. Dazu muss zuerst ein eigener Synchronisationspunkt ausgeführt werden. Wenn dann q wieder das Synchronisationsobjekt anfordert, werden auch die Invalidierungen verschickt. Der andere Prozess q soll die Invalidierungen ja erst an diesem Synchronisationspunkt übernehmen.

Wenn allerdings q auch Schreiboperationen ausgeführt hat, weiß q nicht, dass p mittlerweile auch einen gültigen Wert hat. Daher muss p auch q darüber informieren, dass er eine Schreiboperation auf ein bisher ungültiges Objekt gemacht hat, indem er eine `WriteNotice` versendet (Zeile 5). Empfängt q die `WriteNotice` setzt er `Status(x)` ebenfalls auf *shared*.

Wenn bei einer Schreiboperation auf ein ungültiges Objekt zuerst eine Aktualisierung stattfinden würde, müsste die Schreiboperation blockierend warten, bis die angefragte Aktualisierung empfangen wurde. Allerdings würde der neue Wert gar nicht gelesen, sondern sofort überschrieben werden. Die Aktualisierung dient nur dazu den Status in dem anderen Prozess zu korrigieren. Demgegenüber ist die Schreiboperation des hier entwickelten Protokolls nicht-blockierend.

Listing 4.4: Invalidierungsprotokoll für die Spezielle Schwache Konsistenz

```
1  onWrite(x, val):
2    set Modified(x) := true
3    if Status(x) = invalid
4      set Status(x) := shared
5      send WriteNotice(x)
6    set Value(x) := val
7
8  onRead(x):
9    if Status(x) = invalid
10     send Request(x)
11     wait until Status = shared
12   return Value(x)
13
14 onSync:
15   set NeedSync := true
16   if not HaveSync
17     send RequestSync
18     Wait until HaveSync
19   while bufferedInvalidates not empty
20     dequeue x from bufferedInvalidates
21     if not (this is Steerer and Modified(x))
22       set Status(x) := invalid
23       set Modified(x) := false
24       set Updated(x) := false
25   for all x with Modified(x) = true
26     set Updated(x) := true
27     set LS(x) := Value(x)
28     set Modified(x) := false
29   setNeedSync := false
30   if SyncRequested
31     set SyncRequested := false
32     set HaveSync := false
33     for all x with Updated(x) = true
34       set AKT(x) := LS(x)
35       if Status(x) = shared
36         send Invalidate(x)
37         set Status(x) := exclusive
38       set Updated(x) := false
39     send Sync
40
41 onRecvRequestSync:
42   if NeedSync
43     set SyncRequested := true
44   else
45     set HaveSync := false
46     for all data objects x with Updated(x) = true
47       set AKT(x) := LS(x)
48       if (Status(x) == shared)
```

```
49          send Invalidate(x)
50          set Status(x) := exclusive
51        set Updated(x) := false
52      send Sync
53
54  onRecvSync:
55    set HaveSync := true
56
57  onRecvInvalidate(x):
58    enqueue x to bufferedInvalidates
59
60  onRecvUpdate(x, val):
61    set Value(x) := val
62    set Status(x) := shared
63
64  onRecvRequest(x):
65    set Status(x) := shared
66    send Update (x, AKT(x))
67
68  onRecvWriteNotice(x):
69    set Status(x) := shared
```

Korrektheitsbeweis: Das Protokoll ist korrekt, wenn für die Synchronisationspunkte eine global einheitliche Sichtordnung existiert und die Leseoperationen die durch die SSK definierten Werte zurückliefert. Die Ordnung der Synchronisationspunkte wird mit Hilfe desselben Mechanismus hergestellt wie beim Aktualisierungsprotokoll der SSK. Daher kann der Beweis für die sequentielle Ordnung der Synchronisationspunkte von dort übernommen werden. Es muss also nur noch die Übereinstimmung der Rückgabe von Leseoperationen gezeigt werden.

Wie beim Aktualisierungsprotokoll, ist die Rückgabe einer Leseoperation folgendermaßen definiert: Seien p und q die beiden Prozesse und sein $r_1 \in W(p,x)$ eine Leseoperation. Dann sei w_p die letzte Schreiboperation von p vor r_1:

$$w_p = inf_{PO(p)}\{w \in W(p,x) | w <_{PO(p)} r_1\} \tag{4.67}$$

Ferner sei w_q definiert als:

$$w_q = inf_{PO(q)}\{w \in W(q,x) | T_e(w) < T_b(r_1)\} \tag{4.68}$$

Ist p ein Anwendungsprozess, dann gibt r_1 den von w_p geschriebenen Wert zurück, wenn gilt:

$$T_e(w_q) < T_b(w_p) \tag{4.69}$$

Ist p ein Steering-Prozess, dann gibt r_1 den von w_p geschriebenen Wert zurück, wenn gilt:

$$T_b(w_q) < T_e(w_p) \tag{4.70}$$

Ansonsten gibt r_1 den von w_q geschriebenen Wert zurück.

Um die Korrektheit zu beweisen, wird gezeigt, dass eine Leseoperation den korrekten Wert zurück liefert, wenn eine Invariante zu Beginn der Leseoperation erfüllt ist. Anschließend wird gezeigt, dass diese Invariante für jeden Prozess nach jeder Speicheroperation und bearbeiteten Nachricht erfüllt ist. Dieser Beweis wird mit Hilfe der vollständigen Induktion geführt. Dabei wird jeweils nur ein Prozess betrachtet. Jeder Zugriff wird als ein Induktionsschritt betrachtet und gezeigt, dass eine folgende Leseoperation immer noch den korrekten Wert zurück geben würde.

Die Invariante: Entweder ist

- `Status(x)` *invalid*, und der zu lesende Wert wurde von dem anderen Prozess q geschrieben oder
- `Value(x)` enthält den durch die SSK definierten Wert, und `Status(x)` ist nicht *invalid*.

Eine Leseoperation r_1 gibt den in `Value(x)` gespeicherten Wert zurück, wenn `Status(x)` nicht *invalid* ist. Wenn die Invariante wahr ist, ist dies der von der SSK definiert Wert.

Ansonsten wird an q die Anforderung des aktuellen Wertes geschickt. q sendet $AKT_q(x)$ zurück. $AKT_q(x)$ wird dann in $Value_p(x)$ gespeichert und zurückgegeben.

$AKT_q(x)$ wird gesetzt, wenn q das Synchronisationsobjekt `HaveSync` an p verschickt. In diesem Fall wird $AKT_q(x)$ auf den Wert gesetzt, den x beim letzten Synchronisationspunkt $s_q \in S(q)$ von q hatte. Mit w_q wird die Schreiboperation von q benannt, die diesen Wert geschrieben hatte. Also ist

$$w_q = inf_{PO(q)}\{w \in W(q,x) | T_e(w) \leq T_g(s_q)\}. \tag{4.71}$$

Diese Anforderung des Synchronisationsobjekts (`requestSync`) kann p nur bei Beginn eines eigenen Synchronisationspunkts versendet haben (Zeile 17). Das bedeutet, dieser Synchronisationspunkt muss vor der aktuellen Leseoperation beendet worden sein. Somit muss gelten:

$$T_e(w_q) < T_b(r_1). \tag{4.72}$$

Wenn die Invariante wahr ist, muss der von p zu lesende Wert von q geschrieben worden sein. Es muss also die Bedingung von Formel 4.69 bzw. von Formel 4.70 erfüllt sein. Also gibt r_1 den von der SSK definierten Wert zurück.

Es bleibt zu zeigen, dass die Invariante nach jeder Speicheroperation erfüllt ist.

Induktionsanfang: Im Job wird jedes Datenobjekt x initialisiert, sowie `Status(x)` auf *exclusive* und `Modified(x)` auf wahr gesetzt. Beim Ausführen des ersten Synchronisationspunkts des Jobs hat er noch keine Aktualisierung empfangen und alle Datenobjekte sind auf *exclusive* gesetzt. Es werden auch keine Invalidierungen versendet. Es wird lediglich der Wert nach `LS(x)` übernommen. Somit ist die Invariante im Job erfüllt.

Im Steuerungswerkzeug wird `Status(x)` auf *invalid* gesetzt. Eine erste Leseoperation im Steuerungswerkzeug würde also zu allererst den Wert der letzten

Aktualisierung zurückgeben, was korrekt ist, da das Steuerungswerkzeug noch keine Schreiboperation ausgeführt hat.

Induktionsschritt: Sei p der Prozess, der die jeweilige Operation ausführt und q entsprechend der andere Prozess. Sei x das Datenobjekt auf das zugegriffen wird und sei w_p die letzte Schreiboperation von p auf x. Ferner sei w_q die letzte Schreiboperation von q für x, die bisher für p sichtbar ist und s_p der letzte Synchronisationspunkt von p.

Es wird gezeigt, dass wenn die Invariante erfüllt ist und eine Leseoperation, eine Schreiboperation oder ein Synchronisationspunkt ausgeführt wird oder ein Nachricht bearbeitet wird, anschließend die Invariante wieder erfüllt ist.

1. **Die Leseoperation:** Nach Voraussetzung wird die Leseoperation den korrekten Wert zurückgeben, da die Invariante erfüllt sein muss. Der Wert, den die Leseoperation zurückgibt, ist anschließend in `Value(x)` gespeichert. War `Status(x)` zuvor *invalid*, dann ist `Status(x)` bei Erhalt der `Update`-Nachricht auf *shared* gesetzt worden (Zeile 62). Also kann der Wert von `Status(x)` nicht *invalid* sein. Damit ist die Invariante erfüllt.

2. **Die Schreiboperation:** Wird eine Schreiboperation $w_1 = w(p, x, v)$ ausgeführt, ist $w_p <_{PO(p)} w_1$ und $T_b(w_1) > T_e(w_q)$. Daher muss nach der SSK, eine folgende Leseoperation den Wert v für x lesen. Da v in `Value(x)` gespeichert wird (Zeile 6) und `Status(x)` auf *shared* gesetzt wird, falls er vorher *invalid* war (Zeile 4), ist die Invariante nach der Schreiboperation wieder erfüllt.

3. **Der Synchronisationspunkt:** Bei Synchronisationspunkten können neue Schreiboperationen des anderen Prozesses sichtbar werden. Wurde bisher für x der Wert einer Schreiboperation von p gelesen, muss in diesem Fall der Status *invalid* werden. Es muss also gezeigt werden, dass genau die Datenobjekte invalidiert werden, für die in Zukunft der Wert einer Schreiboperation von q gelesen werden muss und für die der Status bisher nicht *invalid* war.

   Sei $s_1 \in S(p)$ der aktuell ausgeführte Synchronisationspunkt. Zunächst wird unterschieden, ob q während der letzten Epoche von p einen Synchronisationspunkt ausgeführt hat oder nicht.

   Falls q in der vergangen Epoche keinen Synchronisationspunkt ausgeführt hat, dürfen auch keine neuen Schreiboperationen sichtbar sein. Da p das Synchronisationsobjekt immer noch besitzt, sendet es keine `SyncRequest` Nachricht. Folglich hat q auch keine Invalidierungen versendet und es sind keine neuen Schreiboperationen sichtbar. Die Invariante ist also erfüllt.

   Falls q in der vergangenen Epoche einen Synchronisationspunkt ausgeführt hat, muss p bei Beginn der Ausführung des aktuellen Synchronisationspunkts s_1 eine `SyncRequest`-Nachricht versenden (Zeile 17), um das Synchronisationsobjekt zu erhalten. Bevor q das Synchronisationsobjekt in einer `Sync`-Nachricht versendet (Zeile 39 oder Zeile 52), werden `Invalidate`-Nachrichten für jedes Objekt x versendet, für das $Updated_q(x)$ wahr ist und bei dem $Status_q(x)$ auf *shared* gesetzt ist. Es muss also gezeigt werden, dass das die Objekte sind, für die Invalidierungen verschickt werden müssen.

Wenn q eine Schreiboperation $w_1 = w(q, x)$ ausführt, wird $Modified_q(x)$ auf wahr gesetzt (Zeile 2). Wenn diese Schreiboperation dann nicht beim nächsten Synchronisationspunkt invalidiert wird, wird dort $Updated_q(x)$ auf wahr gesetzt. Hat nach w_1 noch kein Synchronisationspunkt stattgefunden, ist $T_e(w_1) > T_g(s_1)$. Daher darf w_1 auch noch nicht gesehen werden. Die einzige Möglichkeit, wie $Updated_q(x)$ wieder unwahr werden kann bevor eine Invalidierung verschickt wird, ist, wenn in einem Synchronisationspunkt von q eine Invalidierung von p angewendet wurde (Zeile 24).

Wenn $Updated_q(x)$ aufgrund einer Invalidierung auf unwahr gesetzt wird, muss q der Anwendungsprozess sein, da sonst die Bedingung in Zeile 21 nicht erfüllt wäre. Außerdem muss dann eine Schreiboperation $w_0 = w(p, x, v_0)$ existieren mit $T_e(w_0) > T_b(w_1)$. Folglich muss w_1 verworfen werden. Damit ist gezeigt, dass $Updated_q(x)$ wahr ist, wenn w_1 für p in s_1 sichtbar werden soll.

Zweitens ist zu zeigen, dass $Status(x) = shared$ ist, wenn eine Aktualisierung versendet werden muss. Es gibt 3 Gründe, warum ein Objekt x, auf das eine Schreiboperation stattgefunden hat, nicht *shared* ist: Es wurde in einem früheren Synchronisationspunkt eine Invalidierung empfangen, es wurde bereits vorher eine `Invalidate`-Nachricht für x verschickt oder x war bereits vor der Schreiboperation *exclusive*:

(a) Es wurde in einem früheren Synchronisationspunkt eine Invalidierung empfangen: Dass findet nur statt, wenn die Schreiboperation verworfen werden muss. Die Begründung ist analog zu der Argumentation, wenn die Variable $Updated_q(x)$ auf unwahr gesetzt wird.

(b) $Status_q(x)$ war schon vor der Schreiboperation *exclusive*: Diesen Status kann x nur erhalten haben, wenn bereits eine Invalidierung verschickt wurde (Zeile 37 oder Zeile 50). Daraus folgt, dass in p der $Status_p(x) = invalid$ gesetzt wurde. Wenn dies durch eine Lese- oder Schreiboperation von p zwischenzeitlich geändert worden wäre, hätte p eine `Request-` oder `WriteNotice`-Nachricht verschickt (Zeile 10 bzw. Zeile 5). Diese hätte q vor der `SyncRequest`-Nachricht empfangen müssen. In diesem Fall hätte q den $Status_q(x)$ auf *shared* gesetzt (Zeile 65 bzw. Zeile 69). Also muss $Status_p(x)$ noch *invalid* sein und es braucht keine Invalidierungsnachricht verschickt zu werden. Die Invariante ist erfüllt.

(c) Es wurde bereits eine Invalidierungsnachricht für x verschickt, nachdem w_1 ausgeführt wurde. Folglich muss $Status_p(x) = invalid$ sein. Dies kann mit derselben Argumentation wie bei 3b gezeigt werden. Folglich braucht auch hier keine Invalidierung verschickt zu werden und die Invariante ist erfüllt.

Wenn also w_1 für p sichtbar werden muss, und $Status_p(x)$ nicht *invalid* ist, wird eine `Invalidate`-Nachricht verschickt.

Alle `Invalidate`-Nachrichten werden verschickt, bevor q das Synchronisationsobjekt verschickt. Also müssen alle `Invalidate`-Nachrichten von p empfangen worden sein, bevor p die Bearbeitung des Synchronisationspunkts beginnt.

Für jedes Datenobjekt x, für dass p eine Invalidierungsnachricht empfängt, muss eine Schreiboperation $w_1 \in W(q, x)$ von q existieren mit $T_e(w_1) > T_b(w_p)$. Wäre $T_e(w_1) < T_b(w_p)$ hätte die `Invalidate`-Nachricht schon bei einem früheren Synchronisationspunkt s_0 mit $T_g(s_0) \leq T_b(w_p)$ empfangen werden müssen. Ob die Invalidierungsnachricht angewendet wird, hängt davon ab, ob p ein Anwendungsprozess oder ein Steering-Prozess ist.

(a) Ist p ein Anwendungsprozess, ist die Bedingung aus Formel 4.69 erfüllt, und die Invalidierung muss angewendet werden. Nun gilt, dass p als nächstes eine Schreiboperation von q lesen muss und $Status_p(x)$ *invalid* ist (Zeile 22). Die Invariante ist damit für den Anwendungsprozess erfüllt.

(b) Ist p ein Steering-Prozess, darf $w_1 \in W(q, x)$ nur sichtbar werden, wenn $T_b(w_1) > T_e(w_p)$ ist. Da eine `Invalidate`-Nachricht empfangen wurde, ist sichergestellt, dass keine Schreiboperation $w_0 \in W(p, x)$ mit $T_e(w_0) > T_b(w_1)$ existiert, die vor dem letzten Synchronisationspunkt s_p ausgeführt wurde. Ansonsten hätte die `Invalidate`-Nachricht von w_0 die Variablen $Modified_q(x)$ und $Updated_q(x)$ auf unwahr gesetzt. Folglich wäre keine `Invalidate`-Nachricht für w_1 verschickt worden.

Es muss noch überprüft werden, ob p in der letzten Epoche eine Schreiboperation w_3 auf x ausgeführt hat. Wenn ja, dann ist

$$T_e(w_3) = T_g(s_1) > T_b(w_1). \tag{4.73}$$

Dann muss p weiterhin den von w_3 geschriebenen Wert lesen. In diesem Fall ist die Bedingung in Zeile 21 falsch, und die Invalidierung wird nicht ausgeführt. Da p auch schon vor dem aktuellen Synchronisationspunkt s_1 den von w_3 geschriebenen Wert lesen musste, ist die Invariante erfüllt.

Hat p keine Schreiboperation in der letzten Epoche ausgeführt, ist $T_e(w_1) > T_b(w_p)$. Also muss p ab jetzt den von w_1 geschriebenen Wert lesen. Die Bedingung in Zeile 21 ist in diesem Fall wahr, die Invalidierung wird angewendet. Somit ist auch in diesem Fall die Invariante erfüllt.

Damit wurde gezeigt, dass auch nach der Ausführung eines Synchronisationspunkts die Invariante erhalten bleibt.

4. **onRecvRequestSync:** In dieser Prozedur wird `Value(x)` nicht verändert und `Status(x)` wird auf *exclusive* gesetzt, falls er vorher *shared* war. Der ausführende Prozess liest denselben Wert wie vorher. Folglich bleibt die Invariante erhalten.

5. **onRecvSync:** Weder `Value(x)` noch `Status(x)` ändern sich. Der ausführende Prozess liest denselben Wert wie vorher. Folglich bleibt die Invariante erhalten.

6. **onRecvInvalidate:** Weder `Value(x)` noch `Status(x)` ändern sich. Der ausführende Prozess liest denselben Wert wie vorher. Folglich bleibt die Invariante erhalten.

7. **onRecvUpdate:** Diese Nachricht kann nur empfangen werden, wenn eine Leseoperation eine Aktualisierung angefordert hat. Daher muss zu Beginn der Leseoperation `Status(x)` *invalid* gewesen sein. Also ist der empfangene Wert, derjenige den die Leseoperation zurückgeben muss. Der neue Wert wird in `Value(x)` gespeichert und `Status(x)` auf *shared* gesetzt. Damit ist die Invariante erfüllt.

8. **onRecvRequest:** Zu zeigen ist hier, dass diese Nachricht von p nur empfangen werden kann, wenn `Status(x)` nicht *invalid* ist. In diesem Fall liest der ausführende Prozess denselben Wert wie vorher. Folglich bleibt die Invariante erhalten.

   Um zu zeigen, dass die `onRecvRequest`-Nachricht nur empfangen werden kann, wenn `Status(x)` nicht *invalid* ist, wird gezeigt, dass nie in beiden Prozessen 'gleichzeitig' `Status(x)` *invalid* ist. D.h. die Intervalle von p und q, in denen `Status(x)` *invalid* ist, sind sequentiell geordnet.

   `Status(x)` kann in p nur *invalid* werden, wenn p eine `Invalidate`-Nachricht von q empfangen hat (siehe Zeile 22). `Invalidate`-Nachrichten werden von q nur versendet, wenn p eine `RequestSync`-Nachricht versendet hat. Das bedeutet, p wartet am Beginn eines Synchronisationspunktes in Zeile 18. D.h. nach dem Versand der `RequestSync`-Nachricht werden keine weiteren Nachrichten von p versendet, bis p eine `Sync`-Nachricht erhält. Das bedeutet insbesondere, dass keine `Invalidate`-Nachrichten von p versendet werden können, bis die `Invalidate`-Nachrichten von q angekommen sind. Die `Sync`-Nachricht wird von q erst versendet, wenn q alle `Invalidate`-Nachrichten versendet hat (Zeile 39 und 52). Dabei setzt q `Status(x)` auf *exclusive* (Zeile 37 und 50).

   Invalidierungen sind also nur möglich, wenn das Synchronisationstoken seinen Besitzer wechselt. Dabei wird sichergestellt, dass der vorherige Besitzer `Status(x)` auf *exclusive* setzt. Da die Synchronisationspunkte sequentiell geordnet sind, müssen auch die Intervalle von p und q, in denen `Status(x)` *invalid* ist, sequentiell geordnet sein.

9. **onRecvWriteNotice:** Diese Nachricht kann nur empfangen werden, wenn `Status(x)` nicht *invalid* ist. Dies kann mit derselben Argumentation wie bei der `onRecvRequest`-Funktion gezeigt werden. Der ausführende Prozess liest denselben Wert wie vorher. Folglich bleibt die Invariante erhalten.

Da nach jeder Speicherzugriffsoperation und bearbeiteten Nachricht die Invariante erhalten bleibt, gilt die Behauptung, dass das vorgestellte Protokoll korrekt ist.

4.3.5 Aktualisierungsprotokoll für die Verzögerte Schwache Konsistenz

Bei der Verzögerten Schwachen Konsistenz (VSK) reagieren Job und Steuerungswerkzeug unterschiedlich auf Schreiboperationen. Wenn der Job schreibt, wird der neue Wert gespeichert und vom Job auch zurückgegeben. Beim nächsten Synchronisationspunkt wird der geänderte Wert in einer `Update`-Nachricht an das Steuerungswerkzeug verschickt. Um zu wissen welche Daten geändert

wurden, wird bei der Schreiboperation eine Modifikationsmarkierung gesetzt (`Modified(x)`). Das Protokoll ist in Abb. 4.19 und Listing 4.5 dargestellt.

Schreibt das Steuerungswerkzeug einen Wert, wird der neue Wert zunächst in `LW(x)` gespeichert (Zeile 7). `Value(x)`, der Wert den die Leseoperation zurückgibt, bleibt gleich. Das Steuerungswerkzeug sieht also nach einer Schreiboperation nicht direkt den geschriebenen Wert. Beim nächsten Synchronisationspunkt werden die `Update`-Nachrichten versendet (Zeile 21), sobald der Job den neuen Wert übernommen hat (Zeile 16), schickt er den neuen Wert zurück an das Steuerungswerkzeug (Zeile 17), der ihn dann ab dem nächsten eigenen Synchronisationspunkt auch sieht.

Um sicherzustellen, dass entweder keine oder alle `Update`-Nachrichten eines Synchronisationspunkts angewendet werden, schickt jeder Prozess am Ende eines Synchronisationspunkts eine `SyncEnd`-Nachricht (Zeile 24), die die Nummer des gerade bearbeiteten Synchronisationspunkts enthält. Empfängt ein Prozess eine `SyncEnd`-Nachricht (Zeile 29-30), weiß der Prozess, dass alle `Update`-Nachrichten, die vor dieser `SyncEnd`-Nachricht empfangen wurden, von diesem Synchronisationspunkt stammen.

Listing 4.5: Aktualisierungsprotokoll für die Verzögerte Schwache Konsistenz

```
1  onRead(x):
2    return Value(x)
3
4  onWrite(x, val):
5    if this is Job
6      set Value(x) := val
7    set LW(x) := val
8    set Modified(x) := true
9
10 onSync:
11   for each (x, val, num) in bufferedUpdates
12     if num < LastSync
13       dequeue (x, val, num) from bufferedUpdates
14       if this is Job
15         set Modified(x) := false
16         set Value(x) := val
17         send Update(x, Value(x))
18       if this is Steerer
19         set Value(x) := val
20   for all x with Modified(x) = true
21     send Update(x, LW(x))
22     set Modified(x) := false
23   set OwnNum := OwnNum + 1
24   send SyncEnd(OwnNum)
25
26 onRecvUpdate(x, val):
27   enqueue (x, val, LastSync) in bufferedUpdates
28
29 onRecvSyncEnd(SyncNum):
30   LastSync = SyncNum
```

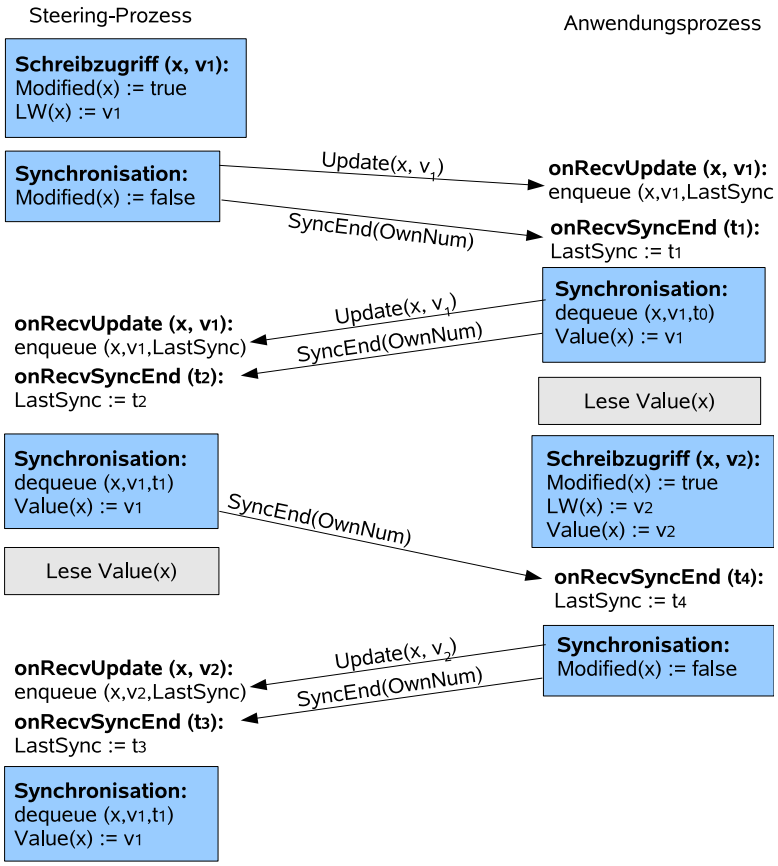


Abbildung 4.19: Funktionsprinzip des Aktualisierungsprotokolls für die Verzögerte Schwache Konsistenz

Beim Aktualisierungsprotokoll muss das Steuerungswerkzeug beim Verbinden mit dem Job seine lokalen Kopien initialisieren und daher die benötigten Werte anfordern, bevor erste Lesezugriffe stattfinden können.

Korrektheitsbeweis: Um die Korrektheit zu beweisen, muss gezeigt werden, dass

- die Synchronisationspunkte von anderen Prozessen in der Reihenfolge gesehen werden, in der sie vom jeweiligen Prozess ausgeführt werden.
- jede Leseoperation den von der Verzögerten Schwachen Konsistenz definierten Wert zurückgibt.

Da es sich bei diesem Protokoll um einen Spezialfall handelt, bei dem genau ein Anwendungsprozess und ein Steering-Prozess existiert, kann die Definition der Verzögerten Schwachen Konsistenz vereinfacht werden. Sei p der Anwendungsprozess und q der Steering-Prozess. Ferner sei r_1 eine Leseoperation von p. Der Rückgabewert von r_1 lässt sich dann folgendermaßen bestimmen:

Da es nur einen Anwendungsprozess gibt, reduziert sich die Anwendungsordnung auf die Programmordnung des Anwendungsprozesses. Sei w_p die letzte Schreiboperation von p:

$$w_p = inf_{PO(p)}\{w \in W(p,x) | w <_{PO(p)} r_1\}. \tag{4.74}$$

Es gibt außerdem nur einen Steering-Prozess. Damit muss nur die letzte sichtbare Schreiboperation des einzigen Steering-Prozesses gesucht werden. Sei w_q die letzte Schreiboperation von q die p sieht. Es gilt also:

$$w_q = inf_{PO(q)}\{w \in W(q,x) | T_e(p,w) < T_b(p,r_1)\} \tag{4.75}$$

Wenn $T_e(p,w_q) < T_b(p,w_p)$ ist, liefert r_1 den von w_p geschriebenen Wert zurück. Ansonsten gibt r_1 den von w_q geschriebenen Wert zurück.

Da es nur einen Anwendungsprozess gibt, ist eindeutig bestimmt, welcher Prozess Schreiboperationen des Steering-Prozesses q sehen muss, bevor q sie selbst sehen kann. Der Steering-Prozess sieht also die Schreiboperation, die der Anwendungsprozess als Letztes gesehen hat, bevor p den letzten Synchronisationspunkt ausgeführt hat, der von dem Steering-Prozess gesehen wird. Eine Leseoperation r_2 des Steering-Prozesses q gibt den Wert folgender Schreiboperation w_1 zurück:

$$\begin{aligned} w_1 = inf_{SO(p)}\{&w \in W(p,x) \cup W(q,x) | \\ &\exists\, s_1 \in S(p) \text{ mit } w <_{SO(p)} s_1 \text{ und } T_g(q,s_1) < T_b(q,r_2)\} \end{aligned} \tag{4.76}$$

PRAM-Konsistenz der Synchronisationspunkte Da die Nachrichten in der Reihenfolge ankommen, in der sie versendet werden und in der Reihenfolge bearbeitet werden, in der sie empfangen werden, werden die Synchronisationspunkte vom anderen Prozess in der Reihenfolge gesehen, in der sie ausgeführt werden. Daraus folgt, dass die Synchronisationspunkte in der Reihenfolge gesehen werden, in der sie ausgeführt werden.

Korrekter Rückgabewert der Leseoperation: Für diesen Beweis wird wieder eine Invariante definiert und gezeigt, dass eine Leseoperation den korrekten Wert zurück liefert, wenn die Invariante erfüllt ist. Anschließend wird mit vollständiger Induktion gezeigt, dass die Invariante nach jeder Speicherzugriffsoperation erfüllt ist.

Die Invariante: `Value(x)` enthält zu jeder Zeit den Wert der Schreiboperation, den eine Leseoperation zu dieser Zeit zurückgeben müsste.

Da eine Leseoperation einfach `Value(x)` zurück gibt (Zeile 2), liefert sie den korrekten Wert zurück.

Induktionsanfang: Der Job muss seine eigenen Daten initialisieren. Dazu muss er eine Schreiboperation auf die Daten ausführen. Danach enthält `Value(x)` diesen Wert. Somit ist die Invariante beim Job erfüllt.

Der Steering-Prozess fordert nach dem Verbinden zuerst alle Daten an, bevor die erste Leseoperation stattfindet. Damit ist die Invariante auch im Steuerungswerkzeug erfüllt.

Induktionsschritt: Es muss gezeigt werden, dass wenn die Invariante erfüllt ist und eine Leseoperation, eine Schreiboperation oder ein Synchronisationspunkt ausgeführt wird, anschließend die Invariante immer noch erfüllt ist. Sei p der ausführende Prozess und q entsprechend der andere Prozess.

1. **Die Leseoperation** ändert `Value(x)` nicht. Nach Induktionsvoraussetzung ist dies der richtige Wert. Die nächste Leseoperation muss denselben Wert zurück liefern. Daher bleibt die Invariante erfüllt.

2. **Die Schreiboperation** $w_1 \in W(p, x)$ speichert den neuen Wert in einer Variablen `LW(x)` (Zeile 7), die den Wert enthält, der beim nächsten Synchronisationspunkt verschickt werden soll. Außerdem wird `Modified(x)` auf wahr gesetzt (Zeile 8). Somit sind alle geänderten Variablen am nächsten Synchronisationspunkt markiert, und der zu aktualisierende Wert ist in `LW(x)` gespeichert. Ist p der Anwendungsprozess wird außerdem noch `Value(x)` mit dem neuen Wert überschrieben (Zeile 6). Da

   $$T_b(p, w_1) > T_e(p, w_q) \text{ und } w_p <_{PO(p)} w_1 \tag{4.77}$$

   ist, muss der neu geschriebene Wert im Anwendungsprozess gesehen werden. Also ist die Invariante korrekt.

3. **Der Synchronisationspunkt:** Sei s_1 der aktuelle Synchronisationspunkt von p. In s_1 können Schreiboperationen des anderen Prozesses q sichtbar werden.

   Bei einer Schreiboperation $w_1 \in W(q, x)$ wird `Modified(x)` auf wahr gesetzt und der neu geschriebene Wert in `LW(x)` gespeichert (Zeile 7-8). Beim Erreichen des nächsten Synchronisationspunkts $s_q \in S(q)$ nach w_1 ist also `Modified(x)` wahr und `LW(x)` enthält den zuletzt auf x geschriebenen Wert. Da nach jedem Synchronisationspunkt`Modified(x)` auf unwahr gesetzt ist (Zeile 22), ist `Modified(x)` genau dann wahr, wenn x in der vergangenen Epoche geschrieben wurde.

   Für das weitere Vorgehen muss unterschieden werden, ob p der Anwendungsprozess oder der Steering-Prozess ist.

   (a) Sei p der Anwendungsprozess. Folglich ist q der Steering-Prozess. Im Steering-Prozess wird für alle geänderten Datenobjekte eine `Update`-Nachricht versendet (Zeile 21). Wenn die `SyncEnd`-Nachricht von s_q von p empfangen wurde, müssen vorher alle `Update`-Nachrichten, die in s_q versendet wurden von p empfangen worden sein. Es gilt dann:

   $$T_e(p, w_1) > T_b(w_p) \tag{4.78}$$

   Folglich muss p von nun an den von w_p geschriebenen Wert sehen. Also setzt das Protokoll `Value(x)` auf den von w_1 geschriebenen Wert (Zeile 16). Da eine eigene Schreiboperation überschrieben würde, wird auch `Modified(x)` auf unwahr gesetzt (Zeile 15). Die Invariante ist erfüllt.

   (b) Sei p der Steering-Prozess. Folglich ist q der Anwendungsprozess. Empfängt p eine `Update`-Nachricht für x, dann enthält diese den zuletzt für q sichtbar gewordenen Wert für x zum Zeitpunkt des sendenden Synchronisationspunkts $s_q \in S(q)$. Die Nachricht wurde versendet, weil entweder $Modified_q(x)$ wahr war (Zeile 21) oder weil

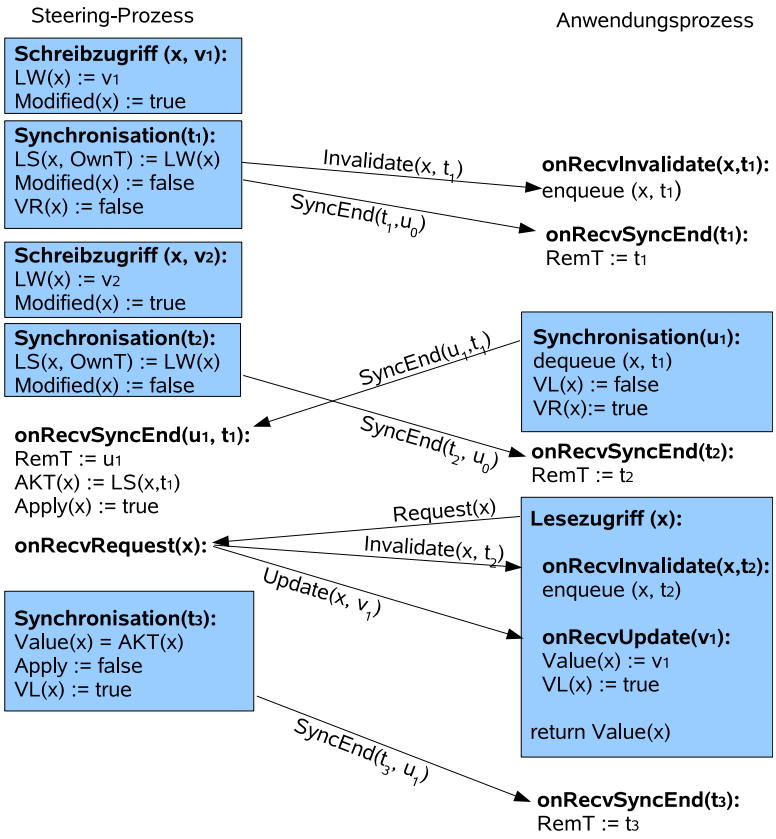


Abbildung 4.20: Funktionsprinzip des Invalidierungsprotokolls für die Verzögerte Schwache Konsistenz bei einer Schreiboperation des Steuerungswerkzeug.

eine Schreiboperation von p auf x in s_q für q sichtbar wurde (Zeile 17). Folglich setzt p `Value(x)` auf den von q geschriebenen Wert für x (Zeile 19). Außerdem gilt dann: Es gibt einen Synchronisationspunkt s_q mit $s_q <_{SO(p)} s_1$, und die Schreiboperation wurde von q vor s_q gesehen. Folglich enthält `Value(x)` genau den Wert, den p lesen muss. Also ist die Invariante erfüllt.

Damit wurde gezeigt, dass die Invariante nach jeder Speicheroperation erfüllt ist. Folglich ist das Protokoll korrekt.

4.3.6 Invalidierungsprotokoll für die Verzögerte Schwache Konsistenz

Beim Invalidierungsprotokoll rührt der meiste Aufwand aus der fehlenden sequentiellen Reihenfolge der Synchronisationspunkte. Dies gilt insbesondere wenn sich Nachrichten von Synchronisationspunkten beider Seiten überkreuzen. Auf der anderen Seite spart die fehlende sequentielle Synchronisation Wartezeit.

Beim Invalidierungsprotokoll für die Verzögerte Schwache Konsistenz werden

nicht die Zustände *exclusive*, *shared* und *invalid* verwendet. Da die Sichtreihenfolge der Speicherzugriffe für jeden Prozess deutlichere Unterschiede als bei der Speziellen Schwachen Konsistenz aufweist, ist es bei diesem Protokoll einfacher, den Status für jeden Prozess separat zu verwalten. Dafür werden zwei neue Variablen verwendet. In der Variable `VL(x)` wird gespeichert, ob `Value(x)` einen gültigen Wert enthält. In der Variable `VR(x)` wird gespeichert, ob der Wert des anderen Prozesses invalidiert werden müsste, wenn dieser Prozess eine Schreiboperation ausführt. In Listing 4.6 ist das Protokoll aufgeführt.

Da es keine einheitliche Reihenfolge der Synchronisationspunkte gibt, besitzt jeder Prozess seine eigene Uhr, d.h. zählt seine eigenen Synchronisationspunkte. Nach Beendigung eines Synchronisationspunkts wird eine `SyncEnd`-Nachricht versandt (Zeile 37), die die Nummer des beendeten Synchronisationspunkts, sowie die Nummer des Synchronisationspunkts des anderen Prozesses von der letzten empfangenen `SyncEnd`-Nachricht enthält. Die letztgenannte Nummer ist wichtig, damit der Steering-Prozess weiß, welche eigenen Synchronisationspunkte von dem Anwendungsprozess gesehen werden.

Es gibt also zwei zusätzliche Variablen:

- `OwnT`: Der Zeitstempel des eigenen Prozesses. Er zählt die Synchronisationspunkte, die von diesem Prozess ausgeführt werden (Zeile 18).
- `RemT`: Die Nummer des Synchronisationspunkts des anderen Prozesses. Der Wert wird der letzten empfangenen `SyncEnd` Nachricht entnommen (Zeile 40).

Bei Erhalt der `SyncEnd` Nachricht von p weiß der andere Prozess q somit den Zeitstempel des letzten Synchronisationspunkts von p und welche Nachrichten von p empfangen und bearbeitet worden sein müssen.

Zunächst wird der Fall untersucht, dass der Steering-Prozess q schreibt (siehe Abb. 4.20). Bei der Schreiboperation $w_1 \in W(q, x)$ wird `Modified(x)` auf wahr gesetzt (Zeile 15) und der neue Wert in der Variablen `LW(x)` gespeichert (Zeile 14). Dadurch kann bei Erreichen eines Synchronisationspunkts bestimmt werden, welche Objekte in der vergangenen Epoche verändert wurden. Außerdem ist der zuletzt geschriebene Wert für x in `LW(x)` gespeichert. Allerdings wird der neu geschriebene Wert noch nicht von q gesehen. Daher wird bei einer Schreiboperation weder `Value(x)` noch `VL(x)` verändert.

`Request`-Nachrichten fragen den zuletzt geschriebenen Wert für x beim zuletzt sichtbaren Synchronisationspunkt ab. Daher muss jeder Prozess diesen Wert nachhalten, auch wenn in der aktuellen Epoche bereits weitere Schreiboperationen stattgefunden haben. Da sich Nachrichten aus nebenläufig ausgeführten Synchronisationspunkten überkreuzen können, kann es sein, dass mehrere Werte aus unterschiedlichen Synchronisationspunkten nachgehalten werden müssen.

Beim Erreichen eines Synchronisationspunkts s_1 wird `(LW(x),` t_1`)` in `LS(x)` eingefügt (Zeile 33). Dabei sei t_1 der aktuelle Wert von `OwnT`. Falls `VR(x)` nicht unwahr ist, wird eine `Invalidate`-Nachricht versendet (Zeile 35), in der spezifiziert wird, welches Datenobjekt invalidiert wurde. Außerdem wird `VR(x)` auf unwahr gesetzt (Zeile 36).

Diese Invalidierung wird von p beim einem Synchronisationspunkt $s_p \in S(p)$ angewendet, $VL_p(x)$ wird auf unwahr gesetzt (Zeile 30). Nach Beendigung von s_p verschickt p eine `SyncEnd` Nachricht, in dem die Zeitstempel von s_p und t_1 enthalten sind.

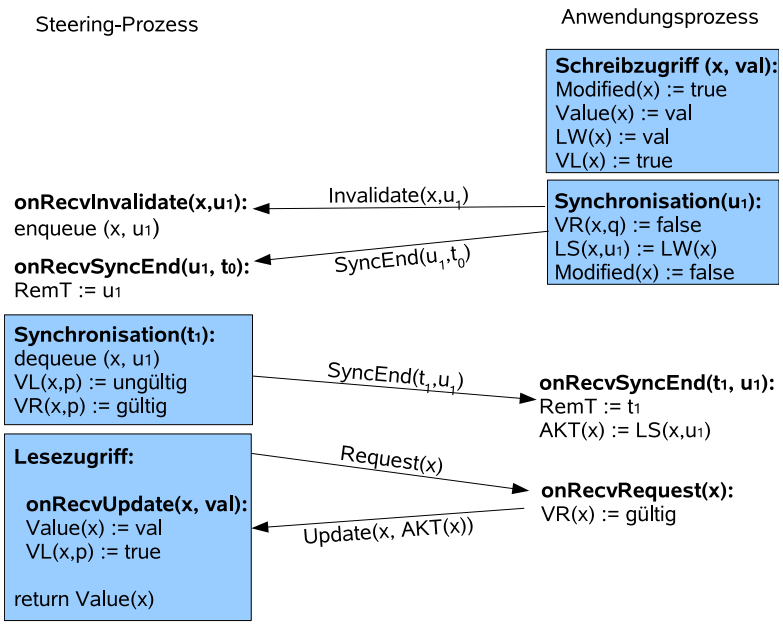


Abbildung 4.21: Funktionsprinzip des Invalidierungsprotokolls für die Verzögerte Schwache Konsistenz bei einer Schreiboperation der Anwendung.

Wenn q die `SyncEnd`-Nachricht von p empfängt, die die Kenntnisnahme von s_1 bestätigt, weiß q, dass p die Invalidierungen von s_1 sieht. Also kann q das Tupel `(val,t)` mit dem Zeitstempel $t = t_1$ in $LS(x)$ suchen und `AKT(x)` auf den dort gespeicherten Wert `val` setzten. Anschließend können alle Tupel (v, t) mit $t \leq t_1$ aus $LS(x)$ gelöscht werden (Zeile 42-46). Führt p eine Leseoperation auf x aus, ist der Wert, den q zurück senden muss, in `AKT(x)` gespeichert, während q selbst noch den vorherigen Wert liest. Erst im nächsten Synchronisationspunkt von q muss auch q *val* sehen. Daher wird in `onRecvSyncEnd` noch `Apply(x)` auf wahr gesetzt, damit q beim nächsten Synchronisationspunkt weiß, für welche Datenobjekte ein neuer Wert sichtbar werden muss.

Im nächsten Synchronisationspunkt, wird für alle Daten bei denen `Apply(x)` wahr ist, `Value(x)` auf `AKT(x)` gesetzt. Außerdem wird $VL_q(x)$ auf wahr gesetzt (Zeile 22-25).

Allerdings kann q weitere Schreiboperationen und Synchronisationspunkte ausgeführt haben, bevor q eine `Request`-Nachricht empfängt. Für diese Schreiboperationen wurde aber keine `Invalidate`Nachricht verschickt, da zu diesem Zeitpunkt $VR_q(x)$ noch unwahr war. Sei $w_2 \in W(q, x)$ solch eine Schreiboperation, für die in einem späteren Synchronisationspunkt $s_2 \in S(q)$ keine Invalidierung verschickt wurde. Ferner sei t_2 der Wert von `OwnT` bei s_2. In s_2 muss dann $(LW(x), t_2)$ in $LS(x)$ eingefügt worden sein. Empfängt q eine `Request`-Nachricht, wird mit der Antwort eine `Invalidate`-Nachricht mitgeschickt, die den Zeitpunkt der erneuten Invalidierung enthält (Zeile 54). Da die Leseoperation blockiert, bis die Antwort erhalten wurde (Zeile 4), kann der nächste Synchronisationspunkt von p nicht ausgeführt werden, bevor die Invalidierung empfangen wurde.

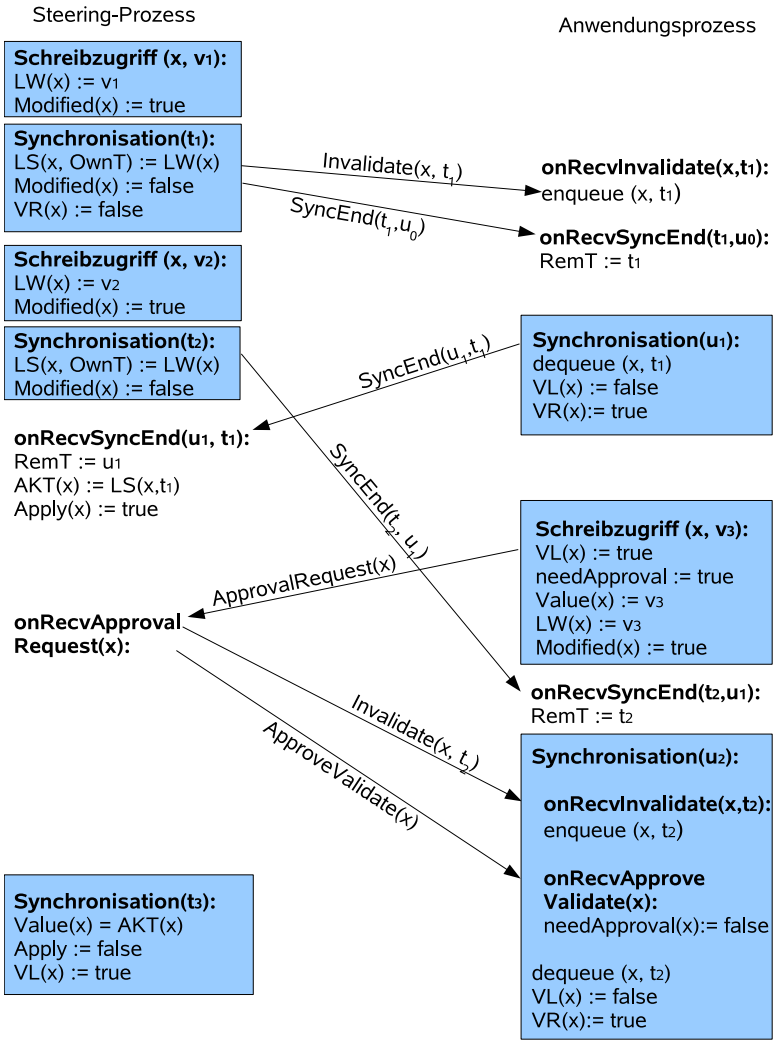


Abbildung 4.22: Funktionsprinzip des Invalidierungsprotokolls für die Verzögerte Schwache Konsistenz wenn die Anwendung einen invalidierten Wert überschreibt.

Nachdem der Fall untersucht wurde, dass der Steering-Prozess eine Schreiboperation ausführt, wird nun der Fall untersucht, dass der Anwendungsprozess p auf x schreibt (siehe Abb. 4.21). Im Gegensatz zum Steuerungswerkzeug ist der neu geschriebene Wert im Job sofort sichtbar und wird in `Value(x)` gespeichert (Zeile 13). Zusätzlich wird `Modified(x)` und $VL_p(x)$ auf wahr gesetzt (Zeile 14-15).

Falls beim nächsten Synchronisationspunkt $s_1 \in S(p)$ eine Invalidierung für x vorliegt, wird die Modifikationsmarkierung (`Modified(x)`) gelöscht (Zeile 29) und $VL_p(x)$ auf unwahr gesetzt (Zeile 30), was die Priorisierung des Steering-Prozesses bewirkt. Andernfalls wird $(LW(x), u_3)$ in $LS(x)$ eingefügt (Zeile 33) und eine Invalidierung verschickt, falls $VR_p(x)$ wahr ist (Zeile 35). Dabei sei u_3 der Zeitstempel des gerade ausgeführten Synchronisationspunkts. Außerdem wird $VR_p(x)$ auf unwahr gesetzt (Zeile 36), um zu verhindern das weitere Invalidierungen für ein ungültiges Objekt verschickt werden.

Empfängt p die `SyncEnd`-Nachricht, die anzeigt, dass q einen Synchronisationspunkt von p sieht, muss er nachhalten, welchen Wert q zur Zeit lesen würde. Dafür wird die Variable `AKT(x)` verwendet. Der Prozess p sucht das Tupel (val, u_3) aus $LS(x)$ und setzt `AKT(x)` auf val (Zeile 44). Anschließend werden alle (v, u) mit $u \leq u_3$ aus $LS(x)$ gelöscht. Fordert q nun einen aktualisierten Wert an, wird `AKT(x)` zurückgegeben. Gleichzeitig muss evtl. auch die nächste Invalidierung mitgeschickt werden und $VR_p(x)$ entsprechend gesetzt werden.

Ein Fall muss noch berücksichtigt werden. Wenn der Anwendungsprozess eine Schreiboperation auf ein Datenobjekt x ausführt und $VL_p(x)$ unwahr war, wäre es möglich, dass q in der Zwischenzeit eine weitere Schreiboperation auf x und einen darauf folgenden Synchronisationspunkt ausgeführt hat. In Folge dessen müsste p den Wert von $VL_p(x)$ im nächsten Synchronisationspunkt auf unwahr setzen. Da aber $VR_q(x)$ unwahr ist, hat q keine `Invalidate`-Nachricht versendet. Bei Leseoperationen wird dies bei Empfang der Anfrage nachgeholt, aber Schreiboperationen senden keine Anfrage (siehe Abb 4.20 und Abb 4.22).

Eine Lösung wäre, dass q, auch wenn $VR_q(x)$ unwahr ist, Invalidierungen sendet. Die andere Möglichkeit ist, dass p bei einer Schreiboperation eine `WriteNotice`-Nachricht an q sendet, falls $VL_p(x)$ unwahr ist. Wenn eine entsprechende Invalidierung existiert, verschickt q eine Invalidierungsnachricht. Auf jeden Fall wird eine `ApproveWrite`-Nachricht versendet. Der nächste Synchronisationspunkt von p muss mit der Ausführung warten, bis alle `ApproveWrite`-Nachrichten eingegangen sind (siehe Abb. 4.22).

Die zweite Lösung ist eine Variante davon, dass bei einer Schreiboperation auf ein ungültiges Objekt zuerst eine Aktualisierung durchgeführt wird. Der aktuelle Wert wird aber sowieso nur überschrieben. Es geht nur darum im anderen Prozess den korrekten Status zu setzen und evtl. eine neue Invalidierung für den nächsten Synchronisationspunkt zu erhalten. Daher braucht die Schreiboperation selbst auch nicht auf die Antwort zu warten, sondern erst beim nächsten Synchronisationspunkt. Dadurch kann die Wartezeit reduziert werden.

Die erste Lösung würde bedeuten, dass das Steuerungswerkzeug bei jeder Schreiboperation eine Invalidierung schicken würde, unabhängig vom Zustand des Datenobjektes. Im zweiten Fall könnten Wartezeiten auftreten, wenn die Anwendung Aktualisierungen durch das Steuerungswerkzeug einfach überschreibt, ohne sie vorher zu lesen. Da eine Verzögerung nur in wenigen Fällen vorkommt, wird in dieser Arbeit der zweite Ansatz weiter verfolgt.

Das gesamte Protokoll ist in Listing 4.6 gezeigt.

Listing 4.6: Invalidierungsprotokoll für die Verzögerte Schwache Konsistenz

```
onRead(x):
  if VL(x) = false
    send Request(x)
    wait until VL(x) = true
  return Value(x)

onWrite(x, val):
  if Job
    if VL(x) = false
      set VL(x) = true
      send WriteNotice(x)
      set needApproval := true
    set Value(x) := val
  set LW(x) := val
  set Modified(x) := true

onSync:
  OwnT := OwnT + 1
  if this is Job
    for all x
      wait until needApproval(x) = false
  for all x with Apply(x) = true
    set Apply(x) := false
    set Value(x) := AKT(x)
    set VL(x) := true
  for all (x,t) in BufInv with t <= RemT
    remove (x,t) from BufInv
    if not (this is Steerer and Modified(x) = true)
      set Modified(x) := false
    set VL(x) := false
    set VR(x) := true
  for all x with Modified(x) = true
    add (LW(x),OwnT) to LS(x)
    if VR(x) = true
      send Invalidate(x,OwnT)
      set VR(x) := false
  send SyncEnd(OwnT,RemT)

onRecvSyncEnd(SyncNum,DoneSyncs):
  set RemT := SyncNum
  for all x
    if exist (val,t) in LS(x) with t <= DoneSyncs
      find (val,t) with max t <= DoneSyncs
      set AKT(x) := val
      delete all (v,t) with t <= DoneSyncs from LS(x)
      if this is Steerer
        set Apply(x) := true

```

```
49 onRecvInvalidate(x,t):
50   add (x,t) to BufInv
51
52 onRecvRequest(x):
53   if exists (val,t) in LS(x)
54     send Invalidate(x,t)
55   else
56     set VR(x) := true
57   sendUpdate(x, AKT(x))
58
59 onRecvUpdate(x, val):
60   set Value(x) := val
61   set VL(x) := true
62
63 onRecvWriteNotice(x):
64   if exists (val,t) in LS(x)
65     send Invalidate(x,t)
66   else
67     set VR(x) := true
68   send ApproveWrite(x)
69
70 onRecvApproveWrite(x):
71   needApproval(x) := false
```

Zur Initialisierung setzt das Steuerungswerkzeug $VL_q(x)$ auf unwahr und $VR_q(x)$ auf wahr. Der Job muss zur eigenen Berechnung die lokalen Werte initialisieren. Mit diesen Werten wird auch `AKT(x)` initialisiert. Außerdem setzt der Job $VL_p(x)$ auf wahr und $VR_p(x)$ auf unwahr.

Korrektheitsbeweis: Analog zum Aktualisierungprotokoll kann gezeigt werden, dass die Synchronisationspunkte bei diesem Protokoll PRAM-konsistent sind. Es muss also nur noch gezeigt werden, dass jede Leseoperation den korrekten Wert zurück liefert. Wie beim Aktualisierungsprotokoll ist der Rückgabewert einer Leseoperation folgendermaßen bestimmt:

Sei w_p die letzte Schreiboperation des Anwendungsprozesses p:

$$w_p = inf_{PO(p)}\{w \in W(p,x) | w <_{PO(p)} r_1\}. \tag{4.79}$$

Sei w_q die letzte Schreiboperation des Steering-Prozesses q die p sieht. Es gilt also:

$$w_q = inf_{PO(q)}\{w \in W(q,x) | T_e(p,w) < T_b(p,r_1)\} \tag{4.80}$$

Wenn $T_e(p,w_q) < T_b(p,w_p)$ liefert r_1 den von w_p geschriebenen Wert zurück. Ansonsten gibt r_1 den von w_q geschriebenen Wert zurück.

Eine Leseoperation r_2 des Steering-Prozesses q gibt den Wert folgender Schreiboperation w_1 zurück:

$$\begin{aligned} w_1 = & inf_{SO(p)}\{w \in W(p,x) \cup W(q,x) | \\ & \exists\, s_1 \in S(p) \text{ mit } w <_{SO(o)} s_1 \text{ und } T_g(q,s_1) < T_b(q,r_2)\} \end{aligned} \tag{4.81}$$

Für den Beweis wird wieder eine Invariante definiert und gezeigt, dass wenn die Invariante zu jeder Zeit erfüllt ist, eine Leseoperation den korrekten Wert

zurück gibt. Anschließend wird mit vollständiger Induktion über alle Speicherzugriffe und Nachrichtenbearbeitungsroutinen eines Prozesse gezeigt, dass die Invariante nach jeder Operation erfüllt ist.

Die Invariante: Entweder ist

- `VL(x)` unwahr und der zu lesende Wert wurde von dem anderen Prozess q geschrieben oder
- `VL(x)` ist wahr und `Value(x)` enthält den durch die VSK definierten Wert.

Falls `VL(x)` wahr ist, wird `Value(x)` zurückgegeben (Zeile 5). Wenn die Invariante wahr ist, ist dies der korrekte Wert.

Falls der `VL(x)` unwahr ist, wird der neue Wert des jeweiligen anderen Prozesses angefordert (Zeile 3). Gezeigt wird, dass diese Leseoperation den angeforderten Wert sehen muss. Da Steering-Prozesse und Anwendungsprozesse sich unterschiedlich verhalten, muss eine Fallunterscheidung gemacht werden:

Falls x vom Anwendungsprozess p gelesen wird, kann $VL_p(x)$ nur unwahr sein, wenn eine `Invalidate`-Nachricht empfangen wurde (Zeile 30), nachdem der Anwendungsprozess diesen Wert das letzte Mal selbst geschrieben hat. Ansonsten hätte die Schreiboperation $VL_p(x)$ auf wahr gesetzt (Zeile 10). Daraus folgt, dass $T_e(p, w_q) > T_e(p, w_p) > T_b(p, w_p)$ ist. Somit ist der Wert, den der Steering-Prozess q mit $w_q \in W(q, x)$ geschrieben hat, derjenige den p lesen muss.

Die Invalidierung kann nur in einem Synchronisationspunkt von p erfolgt sein (Zeile 30), daher muss p eine `SyncEnd`-Nachricht verschickt haben. Da die Reihenfolge der Nachrichten beibehalten wird und die `SyncEnd`-Nachricht vor der `Request`-Nachricht verschickt wurde, hat q den durch w_q geschriebenen Wert in `AKT(x)` gespeichert (Zeile 43), bevor q die `Request`-Nachricht bearbeitet. Somit liefert die Leseoperation von p den korrekten Wert zurück.

Falls x vom Steering-Prozess q gelesen wird: Ist $VL_q(x)$ unwahr, muss p in einem Synchronisationspunkt $s_p \in S(p)$ eine `Invalidate`-Nachricht verschickt haben. Die Schreiboperation $w_p \in W(p, x)$ muss dann den Wert geschrieben haben, den p zu diesem Zeitpunkt lesen würde. Gäbe es eine Schreiboperation $w_q \in W(q, x)$ mit $w_p <_{SO(p)} w_q$, die q sehen müsste, dann müsste es auch einen Synchronisationspunkt $s_2 \in S(p)$ mit $s_p <_{PO(p)} s_2$ geben, so dass $T_g(q, s_2) < T_b(q, r_1)$ gilt. In diesem Fall aber hätte p in s_2 eine `SyncEnd`-Nachricht verschicken müssen, die q in einem Synchronisationspunkt vor r_1 empfangen hätte. Dabei wäre $VL_q(x)$ wahr geworden.

Somit wird richtigerweise der von w_p geschriebene Wert gelesen. Diesen Wert muss p bereits nach `AKT(x)` übernommen haben, da die `SyncEnd`-Nachricht verschickt wird, ehe die `Request`-Nachricht gesendet wird. Also wird der richtige Wert gelesen.

Wenn also die Invariante erfüllt ist, liefert die Leseoperation den korrekten Wert zurück.

Induktionsanfang: Der Anwendungsprozess muss zur Initialisierung seiner Daten eine Schreiboperation ausführen. Dabei wird `Value(x)` auf den richtigen Rückgabewert gesetzt und `VL(x)` wahr. Also ist die Invariante für den Anwendungsprozess erfüllt.

Der Steering-Prozess setzt `VL(x)` auf unwahr. Da zu Beginn der Anwendungsprozess die einzigen Schreiboperationen ausgeführt hat, muss der Steering-Prozess auch den Wert einer Schreiboperation des Anwendungsprozesses lesen. Folglich ist die Invariante auch in diesem Fall erfüllt.

Induktionsschritt: Jede Speicherzugriffsoperation ist wieder ein Induktionsschritt.

1. **Die Leseoperation:** Nach Induktionsvoraussetzung ist die Invariante wahr. Also muss diese Leseoperation den korrekten Wert zurück geben. Dieser Wert ist anschließend in `Value(x)` gespeichert und `VL(x)` ist wahr. Eine folgende Leseoperation muss denselben Wert lesen, die Invariante ist also erfüllt.

2. **Die Schreiboperation:** Falls die Schreiboperation $w_1 = w(p, x, v_1)$ vom Anwendungsprozess p durchgeführt wird, wird der neue Wert in `Value(x)` gespeichert und $VL_p(x)$ auf gültig gesetzt. Eine folgende Leseoperation würde also den neu geschriebenen Wert lesen. Sei $s_p \in S(p)$ der letzte Synchronisationspunkt von p vor w_1 und w_q die letzte sichtbare Schreiboperation von q, dann gilt:

$$T_b(p, w_1) = T_g(p, s_p) > T_e(p, w_q). \tag{4.82}$$

   Daher entspricht dies dem im Konsistenzmodell definierten Wert.

   Falls die Schreiboperation $w_2 \in W(q, x)$ vom Steering-Prozess durchgeführt wird, wird nur `LW(x)` und die `Modified(x)` auf wahr gesetzt. Eine folgende Leseoperation von q würde also noch denselben Wert lesen wie vor der Schreiboperation, wie es von der VSK verlangt wird. Die Invariante bleibt also erfüllt.

3. **Der Synchronisationspunkt:** Der Synchronisationspunkt besteht aus 5 Teilen:

   (a) Der Job wartet bis er für alle Datenobjekte, die der Anwendungsprozess überschrieben hat als `VL(x)` unwahr war, die `ApproveWrite`-Nachricht erhalten hat.

   (b) Im Steering-Prozess werden eigene Schreiboperationen sichtbargemacht (setzten von `Value(x)` für die eine `SyncEnd`-Nachricht die Kenntnisnahme bestätigt hat (`Apply`(x) ist wahr).

   (c) Invalidieren von lokalen Werten, aufgrund von Schreiboperationen des anderen Prozesses.

   (d) Versenden der Invalidierungen für Datenobjekte, deren Wert in der letzten Epoche geändert wurde.

   (e) Versenden der `SyncEnd` Nachricht.

   Zuerst wird gezeigt, dass eine Invalidierung vorliegt, wenn der ausführende Prozess p ab diesem Synchronisationspunkt s_0 eine Schreiboperation des anderen Prozesses q sehen soll. Eine Invalidierung ist nur dann nötig, wenn `VL(x)` bisher wahr war. Hierfür gibt es folgende Möglichkeiten:

(a) Eine Leseoperation von p: Diese muss abgeschlossen sein, bevor s_0 bearbeitet wurde. Folglich muss die Antwort von q vor Beginn von s_0 empfangen worden sein. Daher wurde eine `Invalidate`-Nachricht von q verschickt, falls noch eine Schreiboperation aus einer früheren Epoche vorhanden war, die eine Invalidierung auslösen sollte. Außerdem hat q `VR(x)` auf wahr gesetzt. Für Schreiboperationen in später abgeschlossenen Epochen von q wurde daher in dem entsprechenden Synchronisationspunkt eine Invalidierung versendet.

(b) Eine Leseoperation von q: Dass bedeutet, dass q zu diesem Zeitpunkt eine Schreiboperation von p gelesen hat. Daher kann es keine früheren Schreiboperationen von q geben, die eine Invalidierung auslösen könnten. Nachdem q die Leseoperation beendet hat, ist auch $VR_q(x)$ wahr. Daher müssen Schreiboperationen die in dieser oder späteren Epochen stattfinden eine `Invalidate`-Nachricht auslösen.

(c) Wenn p der Anwendungsprozess ist, kann eine eigene Schreiboperation`VL(x)` auf wahr setzen: Wenn `VL(x)` unwahr ist, sendet die Schreiboperation eine `WriteNotice`-Nachricht. Auf die Antwort wird bei Beginn des Synchronisationspunkts gewartet. Falls eine frühere Schreiboperation von q eine Invalidierung auslösen würde, wird sie vor der `WriteNotice`-Nachricht von q versendet. Außerdem wird $VR_q(x)$ wahr, so dass eine Schreiboperation in dieser oder einer nachfolgenden Epoche eine `Invalidate`-Nachricht auslöst.

(d) Wenn p der Steering-Prozess ist und `Apply(x)` wahr ist, wird in einem Synchronisationspunkt`VL(x)` wahr. Wenn `Apply(x)` wahr ist, muss q zuvor in einem Synchronisationspunkt $VL_q(x)$ auf unwahr gesetzt haben. Eine nachfolgende Schreiboperation löst daher eine `Invalidate`-Nachricht im nächsten Synchronisationspunkt von q aus.

Wenn also `VL(x)` unwahr werden muss, wird eine `Invalidate`-Nachricht empfangen.

Für den Anwendungsprozess gilt dann:

(a) Entweder existiert eine Schreiboperation $w_1 \in W(q, x)$ mit

$$T_e(p, w_1) > T_b(p, w_p)$$

und q hat eine `Invalidate`-Nachricht erhalten oder `VL(x)` war schon ungültig

(b) Oder wegen der Induktionsvoraussetzung enthält `Value(x)` den gültigen Wert und `VL(x)` ist wahr. Es wurde dann keine `Invalidate`-Nachricht empfangen.

Also ist die Invariante für den Anwendungsprozess nach einem Synchronisationspunkt erfüllt.

Bei einem Steering-Prozess können noch eigene Schreiboperationen in s_0 sichtbar werden. Bei einer Schreiboperation $w_1 = w(p, x, v_1)$ des Steering-Prozesses wird der neu geschriebene Wert für den Steering-Prozess nicht direkt sichtbar, sondern erst im nächsten Synchronisationspunkt nachdem der Steering-Prozess die Schreiboperation sieht.

Wenn ein Steering-Prozess p eine eigene Schreiboperation w_1 sichtbar macht, muss`Apply(x)` wahr sein. Das bedeutet, dass der Anwendungsprozess q eine `SyncEnd`-Nachricht verschickt hat, die den Zeitstempel des Synchronisationspunkts enthält, in dem q die Schreiboperation w_1 sieht. Da w_1 auf jeden Fall den bisherigen Wert in q überschreibt, ist w_1 zu diesem Zeitpunkt die zuletzt sichtbar gewordene Schreiboperation. Folglich muss auch p diesen Wert sehen.

Zusammenfassend gilt daher, entweder muss p

(a) denselben Wert lesen wie vorher und hat weder eine Invalidierung empfangen noch eine eigene Schreiboperation angewendet,

(b) hat eine Invalidierung empfangen und muss den Wert einer Schreiboperation von q sehen oder

(c) muss eine neue eigene Schreiboperation als letztes sehen und hat `VL(x)` auf wahr und `Value(x)` auf den Wert dieser Schreiboperation gesetzt.

Folglich ist auch in einem Steering-Prozess die Invariante nach einem Synchronisationspunkt wahr.

4. **onRecvSyncEnd:** In dieser Prozedur wird weder `Value(x)` noch `VL(x)` geändert. Eine folgende Leseoperation würde den selben Wert lesen wie vorher. Die Invariante bleibt wahr.

5. **onRecvInvalidate:** In dieser Prozedur wird weder `Value(x)` noch `VL(x)` geändert. Eine folgende Leseoperation würde den selben Wert lesen wie vorher. Die Invariante bleibt wahr.

6. **onRecvRequest:** In dieser Prozedur wird weder `Value(x)` noch `VL(x)` geändert. Eine folgende Leseoperation würde den selben Wert lesen wie vorher. Die Invariante bleibt wahr.

7. **onRecvUpdate:** Eine `Update`-Nachricht kann nur empfangen werden, wenn in einer Leseoperation `VL(x)` unwahr war und die Leseoperation eine `Request`-Nachricht verschickt hat. Die Leseoperation wartet bis die `Update`-Nachricht bearbeitet wurde. Nach Induktionsvoraussetzung muss sie den Wert eine Schreiboperation des anderen Prozesses zurückgeben. Dieser Wert ist in der empfangenen `Update`-Nachricht enthalten (siehe dazu Beweisführung für die Leseoperation) und wird in `Value(x)` gespeichert. Außerdem wird `VL(x)` auf wahr gesetzt. Also ist die Invariante erfüllt.

8. **onRecvWriteNotice:** In dieser Prozedur wird weder `Value(x)` noch `VL(x)` geändert. Eine folgende Leseoperation würde den selben Wert lesen wie vorher. Die Invariante bleibt wahr.

9. **onRecvApprovalValidate:** In dieser Prozedur wird weder `Value(x)` noch `VL(x)` geändert. Eine folgende Leseoperation würde den selben Wert lesen wie vorher. Die Invariante bleibt wahr.

Da nach jeder Speicheroperation eines Prozesses die Invariante erfüllt ist, ist der Beweis erbracht.

4.3.7 Hybride Protokolle

Die Idee von Hybriden Protokollen ist, dass beide Seiten unterschiedliche Strategien für die Bearbeitung von Lese- und Schreiboperationen verwenden können. So kann z.B. der Job Invalidierungen versenden, während das Steuerungswerkzeug Änderungen mittels Aktualisierungen verbreitet. Bei hybriden Protokollen ist es egal, ob der andere Prozess eine Invalidierungsstrategie oder eine Aktualisierungsstrategie verwendet, solange beide Seiten dasselbe Konsistenzmodell verwenden. Ein Prozess muss nicht einmal wissen, welches der beiden Strategien der andere Prozess verwendet.

Die Grundidee für die Implementierung ist, zwei Protokolle (ein Aktualisierungsprotokoll und ein Invalidierungsprotokoll)zu registrieren, die innerhalb eines Prozesses kooperieren. Der Vorteil ist, dass jeder Prozess unabhängig von anderen Prozessen entscheiden kann, ob er Invalidierungen versendet oder Aktualisierungen. Die vom Invalidierungsprotokoll versendeten Nachrichten werden weiterhin vom Invalidierungsprotokoll bearbeitet und entsprechend die Nachrichten des Aktualisierungsprotokolls vom Aktualisierungsprotokoll. Trotzdem sollen die Protokolle weiterhin korrekt funktionieren, d.h. den richtigen Wert lesen.

Beide Protokolle laufen gleichzeitig in beiden Prozessen. Ankommende Nachrichten können aufgrund des Nachrichtentyps jeweils einem Protokoll zugeordnet werden und werden von diesem bearbeitet. Jeder Prozess entscheidet sich bei jedem Datenobjekt ob er Invalidierungen oder Aktualisierungen verschickt und ruft bei einer Lese- oder Schreiboperationen jeweils die Protokollfunktion des gewünschten Protokolls auf. In Synchronisationspunkten werden die `onSync`-Methoden beider Protokolle ausgeführt. Dabei bearbeitet jedes Protokoll nur die Datenobjekte, die dieses Protokoll verwenden. Zwischen den `onSync`-Methoden beider Protokolle darf das Synchronisationstoken aber nicht abgegeben werden.

Damit das funktionieren kann, müssen das Aktualisierungsprotokoll als auch das Invalidierungsprotokoll sich die objektbezogenen Daten teilen. Dazu zählen z.B. `LW(x)`, `Status(x)`, `VL(x)`, `VR(x)`, `AKT(x)` und `Value(x)`. Das Invalidierungsprotokoll braucht dafür nicht abgeändert zu werden, aber das Aktualisierungsprotokoll muss berücksichtigen, dass der Wert invalidiert wurde und seinerseits die Statusvariablen richtig setzten, wenn er Aktualisierungen erhält oder versendet.

Das Modifizierte, hybride Aktualisierungsprotokoll für die VSK sieht dann folgendermaßen aus:

Listing 4.7: Hybrides Aktualisierungsprotokoll für die Verzögerte Schwache Konsistenz

```
1  onRead(x):
2    if VL(x) = false
3      send Request(x)
4      wait until VL(x) = true
5    return Value(x)
6
7  onWrite(x, val):
8    if Job
9      if VL(x) = false
10       set VL(x) = true
```

```
11        send WriteNotice(x)
12        needApproval := true
13      set Value(x) := val
14    set LW(x) := val
15    set Modified(x) := true
16
17  onSync:
18    OwnT ++
19    if Job
20      for all x
21        wait until needApproval(x) = false
22    for all x with Apply(x) = true
23      set Apply(x) := false
24      set Value(x) := AKT(x)
25      set VL(x) := true
26    for each (x, val, t) in bufferedUpdates
27      if t <= RemT
28        dequeue (x, val, num)
29        set VL(x) := true
30        set VR(x) := true
31        set Value(x) := val
32        if Job
33          set Modified(x) := false
34    for each x with Modified(x) = true
35      add (LW(x),OwnT) to LS(x)
36      send Update(x, LW(x))
37      set Modified(x) := false
38      set VR(x) := true
39    send SyncEnd(OwnNum)
40
41  onRecvSyncEnd(SyncNum,DoneSyncs):
42    set RemT := SyncNum
43    for all x
44      if exist (val,t) in LS(x) with t <= DoneSyncs
45        find (val,t) with max t <= DoneSyncs
46        set AKT(x) := val
47        delete all (v,t) with t <= DoneSyncs from LS(x)
48        if Steerer
49          set Apply(x) := true
50
51  onRecvUpdate(x, val):
52    enqueue (x, val, RemT)
53
54  onRecvRequest(x):
55    find (val,t) in LS(x) with max t
56      sendUpdate(x, val)
57    sendResponse(x, AKT(x))
58    set VR(x) := true
59
60  onRecvResponse(x, val):
```

```
61    set Value(x) := val
62    set VL(x) := true
63
64  onRecvWriteNotice(x):
65    find (val,t) in LS(x) with max t
66      sendUpdate(x, val)
67    VR(x) = valid
68    send ApproveWrite(x)
69
70  onRecvApproveWrite(x):
71    needApproval(x) := false
```

Insgesamt ähnelt die Struktur des hybriden Aktualisierungsprotokolls der Struktur des Invalidierungsprotokolls. Die Änderungen liegen im wesentlichen darin, dass anstatt der `Invalidate`-Nachrichten, `Update`-Nachrichten versendet werden. Zu beachten ist, dass es jetzt zwei Arten von Aktualisierungen gibt: Die `Update`-Nachrichten, die in Synchronisationspunkten angewendet werden und die `Response`-Nachrichten die als Antwort auf eine `Request`-Nachricht verschickt werden und sofort `Value(x)` setzten. Die `Response`-Nachricht entspricht der `Update`-Nachricht im Invalidierungsprotokoll. Die `Update`-Nachricht entspricht der `Update`-Nachricht des Aktualisierungsprotokolls.

Für die SSK und die PRAM-Konsistenz lassen sich hybride Protokolle nach dem gleichen Prinzip erstellen.

4.4 Datenzugriff

Ein Steering-System bildet die Vermittlungsschicht zwischen der Visualisierung oder dem Steuerungswerkzeug des Benutzers auf der einen Seite und der Anwendung auf dem entfernten Rechner, z.B. in einem Grid, auf der anderen Seite. Die Anwendung, wie auch die Visualisierung, können jahrelang gewachsene Programme sein. Damit dieser Datenaustausch stattfinden kann, muss das Steering-System mit der Anwendung und der Visualisierung interagieren. Die Verwendung eines Steuerungswerkzeugs mit einer neuen Anwendung bedeutet im wesentlichen, dass die Anwendung verändert werden muss, damit das Steering-System Zugriff auf die Daten erhält. Bei den meisten Steering-Systemen (z.B. [181, 178, 149, 49, 70, 145, 88, 60]) müssen Aufrufe von Funktionen der Steering-Bibliotheken in den Quellcode eingefügt werden. Diesen Vorgang nennt man *Instrumentierung*. Da ein hoher Instrumentierungsaufwand ein wesentliches Hindernis für die Anwendung von Steering-Systemen ist, sollte der Instrumentierungsaufwand gering gehalten werden. So gibt es Steering-Systeme die den Benutzer bei der Instrumentierung durch spezielle Präprozessoren und graphische Werkzeuge [26] unterstützen. Ein anderer Ansatz arbeitet mit automatischer Instrumentierung der ausführbaren Binärdatei [63].

Ziel der Zugriffsschicht ist es, Methoden und Werkzeuge zur Verfügung zu stellen, welche eine einfache Integration in neue Anwendungen erlauben. Dazu werden zunächst die nötigen Zugriffe klassifiziert (Kap. 4.4.1). Dann wird der Einfluss der Größe der verwalteten Dateneinheiten bestimmt (Kap. 4.4.2). Schließlich werden die Vor- und Nachteile verschiedener Integrationsmethoden analysiert (Kap. 4.4.3).

4.4.1 Zugriffsarten

Je nach Anwendung können unterschiedliche komplexe Steering-Operationen Verwendung finden. Viele komplexe Steering-Operationen sind speziell für eine bestimmte Anwendung zugeschnitten. Jeder Werkzeugkasten mit vorgefertigten Operationen kann aber nur eine begrenzte Zahl von Operationen abdecken. Glücklicherweise lassen sich alle Steering-Operationen in eine Folge von Basisoperationen aufteilen. Letztendlich bildet jede Visualisierung den Zustand der Anwendung ab und jede Steering-Aktion mündet in eine Veränderung des Zustands der Anwendung, ansonsten hat die Steering-Aktion keinen Einfluss auf die Berechnung. Der Zustand einer Anwendung wird durch den Inhalt des Speichers definiert, daher lassen sich zwei grundlegenden Operationen definieren:

- Die Leseoperation
- Die Schreiboperation

Die Lese- und Schreiboperationen werden immer auf *Datenobjekte* angewendet. Ein Datenobjekt bezeichnet eine logische Dateneinheit. Es kann auf unterschiedliche Weise und an verschiedenen Orten gespeichert sein oder repräsentiert werden. Z.B. kann es im Hauptspeicher liegen oder in einer Datei vorliegen. Es kann eine einzelne Variable sein, ein Feld von Variablen, eine Datei oder auch nur ein Teil einer Datei. Die Einteilung der Daten in Datenobjekte für das Steering-System wird vom Benutzer vorgenommen, da er die Bedeutung der Daten und deren Struktur kennt. Wenn z.B. ein großes zweidimensionales Feld existiert, kann das ganze Feld als ein Datenobjekt verwaltet werden; der Benutzer kann jedes Byte als einzelnes Datenobjekt definieren; jede Zeile kann ein Datenobjekt sein oder er kann jede beliebige andere Aufteilung vornehmen. Eine Datei kann als ein einzelnes großes Datenobjekt behandelt werden oder beliebig in Datensätze aufgeteilt werden, die als einzelne Datenobjekte verwaltet werden.

Dadurch, dass vom Steering-System nicht festgelegt wird wie ein Datenobjekt auszusehen hat, ist das Steering-System flexibel für alle Arten von Daten einsetzbar. Eine Aufgabe der Datenzugriffsschicht ist es, dem Steering-System eine einheitliche Schnittstelle zur Verfügung zu stellen, um auf Datenobjekte zuzugreifen ohne dass darunter liegende Schichten sich darum kümmern müssen, wie die Daten gespeichert sind oder wie die Daten repräsentiert werden.

Je nachdem wie und wo die Datenobjekte gespeichert sind, müssen unterschiedliche Zugriffsmethoden eingesetzt werden. Aber selbst für ein einziges Datenobjekt können mehrere Methoden existieren, um auf dieses Datenobjekt zuzugreifen, bzw. um Änderungen der Anwendung zu verfolgen und zu verarbeiten. Wenn ein Datenzugriff der Anwendung abgefangen und eine Funktion des Steering-Systems aufgerufen wird, kann das Steering-System direkt auf diesen Datenzugriff reagieren. Z.B. kann eine Funktion des Steering-Systems aufgerufen werden, nachdem eine Speicherstelle geändert wurde. Oder es kann eine Funktion des Steering-Systems aufgerufen werden, bevor eine Speicherstelle gelesen wird. Diese Aufrufe von Funktionen der Steering-Bibliothek geben dem Steering-System die Möglichkeit die Konsistenz zu überprüfen oder herzustellen. In diesem Fall wird der Zugriff als *aktiv* bezeichnet.

Greift die Anwendung dagegen auf ein Datenobjekt zu, ohne dass die Datenkonsistenzschicht informiert wird, kann sie nicht direkt reagieren. In diesem Fall werden die Zugriffe *passiv* genannt. Während passive Leseoperation

von dem Steering-System vollkommen unbemerkt bleiben, kann das Steering-System durch *Abtastung* des Datenobjektes auf Änderungen reagieren. Dabei kann man die *Wert-sensitive* und die *insensitive* Abtastung unterscheiden. Bei der Wert-sensitiven Abtastung können Änderungen eines Wertes durch das Steering-System bemerkt werden, indem der aktuelle Wert eines Datenobjektes mit dem Wert des Datenobjektes bei der letzten Abtastung verglichen wird. Hat sich der Wert verändert, hat eine Schreiboperation stattgefunden. Bei der insensitiven Abtastung wird bei jeder Abtastung eine Schreiboperation gemeldet. Dadurch muss der Wert nicht mehr gespeichert und verglichen werden, dafür werden möglicherweise unnötige Aktualisierungen versendet.

Der Vorteil der passiven Zugriffe ist, dass es in der Regel weniger Aufwand erfordert eine Anwendung zu instrumentieren als bei aktiven Zugriffen. Außerdem kann durch einen Akkumulierungseffekt die Anzahl der zu bearbeitenden Schreibzugriffe reduziert werden: Sind die Intervalle zwischen den Abtastungen groß genug, kann es auch vorkommen, dass mehrere Schreibzugriffe auf dasselbe Datenobjekt durchgeführt wurden, von denen nur die letzte vom Steering-System verarbeitet werden muss. Auf der anderen Seite wird auch Bearbeitungsaufwand benötigt, um Datenobjekte zu vergleichen oder zu verarbeiten wenn sie seit der letzten Abtastung nicht verändert worden sind.

Auf passive Lesezugriffe kann das Steering-System allerdings nicht reagieren. Dies verhindert die Verwendung von Invalidierungsprotokollen bei passiven Zugriffen. Aktualisierungsprotokolle hingegen müssen nur auf Schreibzugriffe reagieren. Bei schwachen Konsistenzmodellen dürfen Schreibzugriffe für andere Prozesse sowieso erst bei Synchronisationspunkten sichtbar werden. Dadurch können mit einer Abtastung bei jedem Synchronisationspunkt alle relevanten Schreibzugriffe rekonstruiert werden.

4.4.2 Granularität

Wie in Kap. 4.4.1 bereits erwähnt, werden die Datenobjekte durch den Benutzer definiert. Dabei kann der Benutzer die Daten sehr fein aufteilen, so dass jede Variable ein einzelnes Datenobjekt ist, oder aber große Datenobjekte verwenden, wie Arrays, ganze Dateien oder zusammengesetzte Strukturen. Diese Datenobjekte können mehrere MB oder sogar GB groß sein. Diese große Bandbreite beeinflusst natürlich auch die Effektivität des Steering-Systems.

In einigen Fällen ist es einfacher ein großes Objekt für das Steering zu definieren (z.B. eine Datei), als diese große Objekt in viele kleine aufzuteilen (z.B. einzelne Datensätze in einer Datei) und für jedes einzelnen (Teil-)Objekt eigene Zugriffmethoden zu definieren. In diesem Fall wird die Instrumentierung vereinfacht, wenn das Steering-System effizient mit großen Objekten umgehen kann. Werden allerdings immer nur ganze Objekte zwischen den Prozessen ausgetauscht und sind die Objekte zu groß, ändert sich bei den meisten Zugriffen nur ein Teil des Objekts, bzw. es wird eigentlich nur ein Teil benötigt, was dazu führt, dass viele Daten umsonst übertragen werden.

Werden die Objekte klein gewählt, ist der Verwaltungsaufwand für eine vergleichbare Datengröße höher als bei wenigen Objekten und der Instrumentierungsaufwand deutlich höher. Insbesondere bei der Verwendung von Invalidierungsprotokollen ist es günstiger einmal ein etwas größeres Datenobjekt anzufordern als nacheinander mehrere kleine Datenobjekte zu aktualisieren.

Bei DSM-Systemen existieren Studien, die zeigen, dass es von der Anwendung und deren Zugriffsverhalten abhängt, ob eine größere oder kleiner Granularität performanter ist [43]. Die Granularität definiert dabei die kleinste Dateneinheit, die von dem DSM-System verwaltet wird. In Ereignisstrom-basierten Steering-Systemen wird die Granularität in der Regel klein gewählt, weil es die Größe des zu transportierenden Datenvolumens reduziert und die Performance wesentlich durch das Übertragungsvolumen mitbestimmt wird. Dies führt dazu, dass die Optimierung vor allem darin besteht, die Daten feiner zu unterteilen, um das Datenvolumen zu reduzieren.

Da die Grid Jobs auf Rechnern mit unterschiedlicher Architektur laufen, kann z.B. bei der Binärdarstellung einer Ganzzahl die Anzahl oder Reihenfolge der Bytes unterschiedlich sein. Es muss also eine datentypabhängige Umwandlung der Datenobjekt gewährleistet werden. Dies bedeutet, dass der Zugriff mit der Granularität einzelner Basistypen möglich sein muss. Sie bildet also gewissermaßen die Mindestgröße der Granularität.

Für große Datenobjekte kann es auch sinnvoll sein, dass die Granularität kleiner als ein Datenobjekt ist (z.B. bei Dateien). Um Konvertierungsprobleme zu vermeiden, wird die serialisierte Darstellung des Datenobjektes in kleinere Einheiten aufgeteilt. Z.B. kann die serialisierte Darstellung eines großen Objekts in Blöcke gleicher Größe aufgeteilt werden.

Auf der anderen Seite könnte man auch kleinere Datenobjekte zu größeren Gruppen zusammenfassen. Dies bringt aber nur dann Vorteile für die Performance, wenn die kleineren Datenobjekte in einem gemeinsamen Kontext auftauchen, so dass sie häufig mit geringem Abstand gelesen oder geschrieben werden. Man kann natürlich versuchen automatisch zu ermitteln, welche Datenobjekte häufig in einem gemeinsamen Kontext stehen. Allerdings ist eine Gruppierung durch den Benutzer und die Aufteilung in Blöcke, falls das Objekt zu groß ist, einfacher.

Die Granularität ist also ein Objekt oder kleiner als ein Objekt. Die Datenzugriffsschicht muss also den Zugriff auf Objektebene vollziehen, die Objekte serialisieren/deserialisieren und Standardmethoden anbieten, um auf Teile serialisierter Objekte zuzugreifen. Da in allen unteren Schichten nur noch auf serialisierte Daten zugegriffen wird, benötigen sie keine Datentypinformationen. Dadurch ist die Implementierung der Steering-Mechanismen datentypunabhängig und somit anwendungsunabhängig. Um das Steering-System in neue Anwendungen zu integrieren, müssen lediglich der Datenzugriff auf die Daten und die passenden Serialisierungsmethoden bereitgestellt werden.

4.4.3 Integrationsmöglichkeiten

Das Ziel der Zugriffsschicht ist es, das Steering-System in die Anwendung oder die Visualisierung zu integrieren. Dazu muss es zum einen eine Möglichkeit geben, dass das Steering-System auf die entsprechenden Daten zugreifen kann. Das Steering-System muss z.B. bei aktiven Zugriffen und in Synchronisationspunkten benachrichtigt werden. Die häufigste Methode dafür ist die Quellcodeinstrumentierung. Diese Methode ist so aufwendig, dass es ein häufiger Grund ist, dass keine existierenden Steering-Systeme eingesetzt werden.

Methoden, die die manuelle Instrumentierung von Anwendungen reduzieren oder überflüssig machen, stellen eine enorme Erleichterung der Integration von Steering-Systemen in Anwendungen dar. Ziel dieses Abschnitts ist es, ver-

schieden Ansätze zur Instrumentierung zu untersuchen. Es sollen Werkzeuge zur Verfügung gestellt werden, die eine einfache Integration von Steering-Systemen in eine bestehende Anwendung oder ein bestehendes Visualisierungswerkzeug ermöglichen.

Quellcode-Instrumentierung

Bei der Quellcode-Instrumentierung werden in den Quellcode der Anwendung Bibliotheksaufrufe zum Steering-System eingefügt. Zuerst wird ein Wert, der zur Verfügung stehen soll, beim Steering-System registriert. Wenn ein aktiver Zugriff oder ein Synchronisationspunkt ausgeführt werden soll, muss an der entsprechenden Stelle ebenfalls ein Bibliotheksaufruf eingefügt werden. Aufgrund das hohen Aufwands bei jeden Zugriff einen Aufruf der Steering-Bibliothek einzufügen, wird häufig auf aktive Aufrufe verzichtet.

Eingriffe in die Speicherverwaltung

Eingriffe in die Speicherverwaltung sind eine in DSM-Systemen häufig verwendete Methode, um Speicherzugriffe abzufangen. Das kann auf unterschiedliche Weise umgesetzt werden. Einige DSM-Systeme arbeiten auf Betriebssystemebene und ersetzen Standardbibliotheken, setzen Bearbeitungsroutinen für geworfenen Seitenfehler oder überladen Speicherallokationsoperatoren. Dies erlaubt aktive Speicherzugriffe auf Daten im Hauptspeicher, ohne den Quellcode modifizieren zu müssen. Grundsätzlich können so Speicherzugriffe auf den gesamten virtuellen Speicherbereich verfolgt und entsprechende Steering-Methoden aufgerufen werden, ohne einen einzigen Befehl einfügen zu müssen. Allerdings kann eine Neukompilation der Anwendung notwendig sein.

Z.B. lassen sich die von der Anwendung geladenen Standardbibliotheken unter Linux ohne Probleme austauschen, indem man die `LD_PRELOAD` Umgebungsvariable setzt.

Der Nachteil dieses Verfahrens ist, dass die Typinformationen verloren gehen. Das bedeutet, dass es entweder nur auf homogenen Systemen verwendet werden kann oder man doch zusätzlich Typinformationen durch Instrumentierung bereitstellen muss.

POSIX Dateizugriffe umlenken

Ergebnisse, die in eine Datei geschrieben werden, verwenden meist dynamisch geladene Funktionen der C-Standardbibliotheken. Durch überladen der Standardbibliotheken für den Dateizugriff, kann jeder Zugriff an das Steering-System umgelenkt werden.

Dieses Vorgehen erzeugt aktive Lese- und Schreiboperationen, völlig transparent für die Anwendung. Die Anwendung muss nicht instrumentiert oder rekompiliert werden. Diese Methode kann daher sogar angewendet werden, wenn nur das ausführbare Maschinenprogramm vorliegt. Da Dateien normalerweise plattformunabhängig sein müssen, liegen die Daten in serialisierter Form vor und können entsprechend ausgetauscht werden.

Getter/Setter Methoden verändern

Getter/Setter Methoden verändern ist ein Ansatz, der dann funktioniert, wenn der Zugriff auf die Datenobjekte ausschließlich über die Getter/Setter Methoden erfolgt und die entsprechenden Methoden dynamisch ersetzt oder verändert werden können. So erfolgt beispielsweise in Python der Zugriff auf Attribute immer über `__getattr__` bzw. `__setattr__`. Es kann eine einfache Wrapperklasse entwickelt werden, mit der beliebige Pythonobjekte gekapselt werden können. Die Wrapperklasse ruft bei alle Methodenaufrufen die gekapselte Klasse und deren Methoden auf und verhält sich daher wie die gekapselte Klasse. Lediglich die `__getattr__` und `__setattr__` Methode wird überschrieben, um des Steering zu ermöglichen.

Eine andere Möglichkeit besteht darin, die virtuelle Funktionstabelle zu modifizieren und Aufrufe an das Steering-System einzuschieben ähnlich dem Vorgehen in [6]. Allerdings müssen dann die Getter/Setter Methoden virtuell sein.

Bei diesem Zugriffsansatz müssen die Daten angemeldet werden, um die Kapselung oder Modifikation der virtuellen Funktionstabelle zu ermöglichen. Dies geschieht in der Regel über einen einmaligen Bibliotheksaufruf. Danach sind die Zugriffe aktiv ohne weitere Änderungen des Quelltextes. Eine einmalige Anmeldung eines Wertes ist allerdings weiterhin nötig, bedeutet aber nur einen geringen Aufwand für die Instrumentierung.

Spezielle Compiler

Spezielle Compiler oder Präprozessoren können Datenzugriffe erkennen und entsprechende Anweisungen für das Steering einfügen. Dies erfordert erneutes kompilieren des gesamten Programms mit dem entsprechenden Compiler.

Welche der beschriebenen Ansätze am günstigsten ist, hängt von der Anwendung und dem Datenobjekt selbst ab. Die Datenzugriffsschicht kann mehrere Zugriffsarten anbieten, die gleichzeitig genutzt werden können. In der Regel wird eine Registrierung der Daten notwendig sein um Typinformationen zu erhalten und eine Auswahl von steuerbaren oder visualisierbaren Datenobjekten zu erstellen.

4.5 Optimierung

In dem Schichtenmodell für Online-Steering (siehe Abb. 4.1) ist die Datenauswertungsschicht für automatisierte Auswertungen und Optimierungen vorgesehen. Je nach Anwendung sind viele Optimierungsziele denkbar wie Minimierung des Übertragungsvolumens, Minimierung der Antwortzeit, Minimierung der Rechenzeit für das WN, Optimierung der Lastbalance bei parallelen Jobs oder automatisierte Optimierung von Anwendungsparametern. Welches Optimierungsziel verfolgt wird, ist anwendungsabhängig. So wird z.B. ein Programmierer, der ein Programm testet und Fehler sucht, eher einen höheren Rechenaufwand in Kauf nehmen, wenn er dafür eine kurze Antwortzeit hat, als jemand der rechenintensive Simulationen ausführt und hauptsächlich an den Endergebnissen interessiert ist.

Bei der automatisierte Optimierung können nur solche Parameter gesetzt werden, die das Konsistenzverhalten des Steering-Systems nicht ändern. Ob ein Update oder Invalidierungsprotokoll verwendet wird, spielt bei aktiven Zugriffen keine Rolle für die Konsistenzbedingungen. Untersuchungen bei DSM-Systemen [45] zeigen aber, dass abhängig vom Zugriffsmuster Performanceunterschiede bestehen.

Bei Steering-Systemen wie Magellan [181] kann die Performance verbessert werden, indem andere Sensoren gewählt werden, die andere Eigenschaften haben. Dies betrifft bei passiven Zugriffen den Abtastmodus zur Rekonstruktion von Schreibzugriffen an Synchronisationspunkten. In diesem Kapitel werden die folgenden Möglichkeiten untersucht:

- Automatisierte Wahl zwischen einem Aktualisierungsprotokoll und einem Invalidierungsprotokoll, falls die Speicherzugriffe aktiv sind.
- Wahl zwischen Wert-sensitiver und insensitiver Abtastung, falls sie Speicherzugriffe passiv sind.

4.5.1 Protokollwahl bei aktiven Zugriffen

Das Aktualisierungs- und das Invalidierungsprotokoll unterscheiden sich darin, wann und wie Aktualisierungen übermittelt werden. Daher kann es bei unterschiedlichen Folgen von Zugriffsoperationen zu Performanceunterschieden zwischen den einzelnen Protokollen kommen. Ziel ist es, anhand der bisherigen Zugriffsfolgen, das Protokoll zu wählen, welches die Performance optimiert.

Das Aktualisierungsprotokoll überträgt bei jedem Schreibzugriff den aktuellen Wert des entsprechenden Datenobjektes. Dies bedeutet, dass bei allen beteiligten Prozessen die lokalen Daten immer aktuell sind. Beim Lesen der Daten entstehen somit keine Verzögerung dadurch, dass der aktuelle Wert erst von einem entfernten Prozess angefordert und übertragen werden muss. Beim Aktualisierungsprotokoll werden die Aktualisierungen also von Schreibzugriffen ausgelöst. Der Nachteil ist, dass bei häufigen Schreibzugriffen und wenigen Leseoperationen viele Aktualisierungen übertragen werden, die nie gelesen werden. Insbesondere bei großen Datenobjekten kann viel unnötiger Datenverkehr entstehen. Übersteigt die Datenmenge die Netzwerkbandbreite, führt dies zu einer Überlastung des Netzwerks und damit zu einer Vergrößerung von Antwortzeiten.

Das Invalidierungsprotokoll überträgt eventuelle Aktualisierungen erst wenn sie gelesen werden. Dies führt zu einer Verlängerung der Antwortzeiten, da erst der aktuelle Wert eines Datenobjektes von einem entfernten Rechner angefordert und übertragen werden muss. Wird allerdings ein Wert häufiger geschrieben als er gelesen wird, kann das Datenvolumen gesenkt werden, da nach einer Leseoperation nur die erste Schreiboperation eine Invalidierungsnachricht verschickt. Folgende Schreiboperationen verursachen keinen weiteren Netzwerkverkehr bis zur nächsten Leseoperation. Beim Invalidierungsprotokoll werden die Aktualisierungen also von Lesezugriffen ausgelöst.

Formalisierung

Das Optimierungsziel ist eine möglichst geringe Antwortzeit zu erreichen, ohne dass das Datenvolumen die Netzwerkbandbreite übersteigt. Es muss ein Kompromiss zwischen der optimalen Antwortzeit und der verwendeten Netzwerk-

bandbreite gefunden werden. Um diese Anforderungen zu formalisieren, werden zuerst der benötigte Durchsatz D (zu versendendes Datenvolumen pro Zeiteinheit) und die Antwortzeit t in Abhängigkeit des Protokolls $P \in \{A, I\}$ für jedes Datenobjekt $o \in \Omega$ abgeschätzt, wobei A das Aktualisierungsprotokoll, I das Invalidierungsprotokoll und Ω die Menge der Datenobjekte kennzeichnen.

$$D(A, o) \approx N_w(o) \cdot S(o) \tag{4.83}$$
$$D(I, o) \approx N_r(o) \cdot (S(o) + Inv + Req) \tag{4.84}$$
$$t(A, o) \approx 0 \tag{4.85}$$
$$t(I, o) \approx t_r + (S(o) + Req)/B \tag{4.86}$$

Dabei bedeutet $N_w(o)$ die Frequenz der Schreiboperationen auf o, $N_r(o)$ die Frequenz der Leseoperationen, die anderer Prozesse auf o ausführen, die eine Aktualisierung auslösen, $S(o)$ die Größe von o, Inv die Größe einer Invalidierungsnachricht, Req die Größe einer Aktualisierungsanforderung, t_r die Round-Trip-Zeit des Netzwerks und B die Netzwerkbandbreite.

Sei $P(o)$ das für o verwendete Protokoll, dann können D und t folgendermaßen beschrieben werden:

$$D = \sum_{\forall o} D(P(o), o) \tag{4.87}$$
$$t = \frac{1}{\sum_{\forall o} N_r(o)} \sum_{\forall o} t(P(o), o) \cdot N_r(o) \tag{4.88}$$

Um das Datenvolumen und die Antwortzeit gegeneinander abwägen zu können, wird eine Kostenfunktion benötigt. Dabei wird die Antwortzeit mit c_t gewichtet und geht linear in die Kostenfunktion ein. Der benötigte Durchsatz wird erst kritisch, wenn die Bandbreite ausgelastet wird. Daher ist ein gewisser Durchsatz $c_B \cdot B$ kostenfrei. Wieviel kostenfrei ist, hängt von der Bandbreite ab. Ein Überschreiten der Bandbreite sollte aber verhindert werden. Daher geht der Durchsatz mit einem höheren Exponenten c_e in die Kostenfunktion ein. Durch die Konstante c_D wird der Durchsatz gewichtet. Somit erhält man folgende Kostenfunktion:

$$K = \frac{t}{c_t} + \left(\frac{max(0, D - c_B B)}{c_D} \right)^{c_e} \tag{4.89}$$

Die Protokolle $P(o)$ sollen für jedes o so gewählt werden, dass K minimal wird. Die Protokollwahl hängt dabei von den Netzwerkeigenschaften B und t_t, der Datengröße $S(o)$ und dem Datenzugriffsverhalten ab. Das Verhalten das Protokollwahlalgorithmus kann durch die Konstanten c_t, c_B, c_e und c_D konfiguriert werden.

Datenzugriffsverhalten

Da der Datentransfer beim Aktualisierungsprotokoll durch eine Schreiboperation ausgelöst wird und beim Invalidierungsprotokoll von einem Lesezugriff, gilt es abzuschätzen, wie oft das Invalidierungs- und das Aktualisierungsprotokoll in

der Zukunft eine Aktualisierung auslösen werden. Da die zukünftigen Speicheroperationen allerdings nicht bekannt sind, kann nur versucht werden, aus den bisherigen Zugriffen das zukünftige Verhalten zu extrapolieren.

Die relevanten Parameter, um eine Entscheidung für ein Protokoll zu fällen, sind für jedes Datenobjekt o und für jeden Prozess:

- die Frequenz $N_r(o)$ der Lesezugriffe anderer Prozesse, die einen anderen Wert lesen als die vorhergehende Leseoperation des selben Prozesses. Dies entspricht den Leseoperationen, die beim Invalidierungsprotokoll eine Aktualisierungsanfrage auslösen.
- die Frequenz der Schreibzugriffe $N_w(o)$.

Das Zugriffsverhalten $Z(o, p)$ für ein Datenobjekt o wird für jeden Prozess p separat beschrieben und besteht aus dem Tupel:

$$Z(o, p) = (N_r(o), N_w(o)) \tag{4.90}$$

Damit die automatisierte Protokollwahl effizient bleibt, sollten nicht bei jedem Zugriff zusätzliche Nachrichten versandt werden, die der Erfassung der Zugriffe dienen. Wenn beispielsweise beim Invalidierungsprotokoll eine Nachricht nach jeder Schreiboperation versandt wird, wird der Vorteil des Invalidierungsprotokolls geschwächt oder sogar aufgehoben. Der Vorteil des Invalidierungsprotokolls ist ja gerade, dass bei sukzessiven Schreiboperationen, die von dem anderen Prozess nicht gelesen werden, nur bei der ersten Schreiboperation eine Nachricht versendet wird.

Unglücklicherweise werden Informationen von anderen Prozessen benötigt um $N_r(o)$ bestimmen zu können. Verwendet der Prozess eine Invalidierungsstrategie, braucht er nur die Validierungsanfragen zu zählen um $N_r(o)$ zu ermitteln, verwendet er allerdings ein Aktualisierungsprotokoll, erhält er keine Informationen darüber, welche Aktualisierungen wirklich gelesen worden sind.

Um näherungsweise $N_r(o)$ bei Verwendung eines Aktualisierungsprotokolls zu ermitteln, wird der Anteil N_f der gelesenen Aktualisierungen von allen Aktualisierungen bestimmt. Dann gilt:

$$N_r(o) = N_w(o) \cdot N_f(o) \tag{4.91}$$

Falls ein Prozess p ein Aktualisierungsprotokoll verwendet, kann der andere Prozess q den Wert von N_f bestimmen. Ändert sich N_f um einen vorgegebenen Faktor, übermittelt q den aktuellen Wert von N_f an p.

Protokollwahlalgorithmus

Bisher wurden die Datenzugriffsmuster für einen einzelnen Prozess betrachtet. Zu einem Protokoll gehören jedoch alle beteiligten Prozesse. Außerdem können die Zugriffsmuster für dasselbe Datenobjekt in jedem Prozess unterschiedlich sein. Folglich müssen sich alle Prozesse paarweise auf ein gemeinsames Protokoll einigen. Dabei sollen die Protokolle so gewählt werden, dass die Kosten minimiert werden.

Eine interessante Alternative ist die Verwendung von hybriden Protokollen, bei dem Invalidierungs- und Aktualisierungsstrategien von verschiedenen Prozessen gleichzeitig genutzt werden können. Zum Beispiel kann eine Seite

Schreibzugriffe mit dem Aktualisierungsprotokoll behandelt, während die andere Seite eine Invalidierungsstrategie verwendet. Dabei kann jede Seite unabhängig entscheiden, ob sie Invalidierungen oder Aktualisierungen verschickt. Die Empfänger können sowohl Invalidierungen als auch mit Aktualisierungen korrekt verarbeiten, unabhängig davon ob sie selbst Invalidierungen oder Aktualisierungen versenden. Dies hat folgende Vorteile:

- Jede Seite kann das optimale Protokoll wählen, wodurch evtl. ein Ungleichgewicht bei der Lastverteilung ausgeglichen werden kann, um so die Ausführungszeit des Gesamtsystems zu verkürzen.
- Die Entscheidungsfindung benötigt keine zusätzliche Kommunikation, basiert nur auf den lokalen Zugriffsdaten und kann somit schneller durchgeführt werden.
- Der Komplexitätsgrad der Optimierungsalgorithmen sinkt, was zu einfacheren und schnelleren Algorithmen führen kann.

Eine wichtige Beobachtung, die sich aus der Kostenfunktion ergibt, ist, dass das Aktualisierungsprotokoll optimal ist, solange der benötigte Durchsatz einen bestimmten Anteil der Bandbreite nicht überschreitet. D.h. erst wenn der benötigte Durchsatz diesen Anteil überschreitet, führt die Verwendung eines Invalidierungsprotokolls zu einer besseren Performance. Überschreitet der benötigte Durchsatz die Bandbreite, führt der höhere Exponent c_e in der Kostenfunktion dazu, dass die Kosten mit einem Aktualisierungsprotokoll sehr schnell höher werden, als bei der Verwendung eines Invalidierungsprotokolls. Daraus lässt sich ein Algorithmus ableiten:

Listing 4.8: Protokollwahlalgorithmus zur Optimierung der Performance

```
1  Bestimme Netzwerkeigenschaften
2
3  FOR ALL o DO
4  BEGIN
5     Berechne D(A,o), D(I,o), t(A,o) und t(I,o)
6     P(o) := A
7  END
8
9  Sortiere alle o nach D(A, o) - D(I, o) absteigend
10
11 FOR ALL o DO
12 BEGIN
13    k = momentane Gesamtkosten
14    P(o) := I
15    IF neue Gesamtkosten > k THEN P(o) := A
16 END
```

Der Algorithmus 4.8 erstellt eine Liste, in der die Objekte danach sortiert sind, wieviele Durchsatz die Verwendung eines Invalidierungsprotokolls des Objektes spart. Gestartet wird mit einer Konfiguration in der alle Objekte das Aktualisierungsprotokoll verwenden. Dann wird bei dem Datenobjekt begonnen, bei dem die Verwendung des Invalidierungsprotokolls am Kostensparensten ist,

zu testen ob ein Wechsel zum Invalidierungsalgorithmus insgesamt eine Kostenersparnis bringt. Wenn ja, wird für dieses Objekt das Invalidierungsprotokoll verwendet.

Dieser Test wird für alle Objekte durchgeführt, auch wenn es sich bereits für ein anderes Objekt nicht gelohnt hat, es auf das Invalidierungsprotokoll umzustellen. Es besteht trotzdem die Möglichkeit, dass ein anderes Objekt, welches wesentlich kleiner ist, doch noch die Kosten senkt.

Dynamische Adaption des verwendeten Protokolls

Da sich das Zugriffverhalten während der Ausführung ändern kann oder die zur Verfügung stehende Netzwerkbandbreite schwanken kann, muss der Algorithmus regelmäßig ausgeführt werden, um die Protokolle auf die veränderte Umgebung anzupassen. Die Laufzeit der Initialisierung beträgt $O(|\Omega| \cdot log|\Omega|)$ aufgrund der Sortierung. Die Schleife wird $|\Omega|$ mal durchlaufen, wobei $O(1)$ Rechenzeit in der Schleife benötigt wird. Insgesamt erhält man also eine Laufzeit von $O(|\Omega| \cdot log|\Omega|)$.

In der Praxis, wird aber meist nur für einzelne Datenobjekte das Protokoll geändert. Wenn es daher vermieden werden kann, die Protokolle für alle Objekte reglemäßig neu zu berechnen, kann damit Rechenzeit gespart werden. Ziel dieses Kapitels ist es, einen dynamischen Algorithmus zu finden, der möglichst nur die Objekte berechnet, für die sich das Protokoll ändert.

Die Änderungen können nur auftreten, wenn sich einer der folgenden Werte ändert:

- Die verfügbare Bandbreite B, z.B. wenn andere Nutzer ebenfalls einen Teil der Gesamtbandbreite beanspruchen.
- Das Datenzugriffsverhalten auf ein Datenobjekt o und damit das Datenvolumen $D(o)$.
- Die Datengröße $S(o)$, was wiederum zu einer Veränderung von $D(o)$ führt.
- Die Round-Trip-Zeit.

$D(o)$ ändert sich also bei Datenzugriffen und die Netzwerkeigenschaften müssen regelmäßig überprüft werden. Wenn bei Algorithmus 4.8 darauf verzichtet wird die ganze Liste durchzugehen, sondern statt dessen die zweite Schleife abgebrochen wird, sobald ein Objekt erreicht wird, für das nicht das Invalidierungsprotokoll gewählt wird, kann dieser Algorithmus angepasst werden (siehe Alg. 4.9). Mit dieser Vereinfachung sind alle Objekte, für die das Invalidierungsprotokoll angewendet wird, am Anfang der Liste. Dahinter folgen alle Objekte, für die das Aktualisierungsprotokoll verwendet wird.

Listing 4.9: Adaptiver Protokollwahlalgorithmus

```
1 // Initialisierung
2 Bestimme Netzwerkeigenschaften.
3 FOR ALL o DO
4 BEGIN
5   Berechne D(A,o), D(I,o), t(A,o) und t(I,o)
6   P(o) := A
7 END
```

```
8  Sortiere alle o nach D(A, o) - D(I, o) absteigend.
9  p := Erstes Objekt der sortierten Liste
10
11 // Bei jedem Zugriff auf ein Datenobjekt o
12 BEGIN
13   Berechne D(P(o),o) und t(o)
14   Berechne neue Position von o
15   IF o = p AND NOT neue Position = POS(o) THEN
16     p := NEXT(p)
17   IF POS(o) > POS(p) THEN
18     P(o) := I
19   ELSE P(o) := A
20   Verschiebe o an neue Position
21 END
22
23 // In Synchronisationspunkten
24 LOOP
25   Bestimme Netzwerkeigenschaften
26   o := PREV(p)
27   IF Kosten geringer wenn P(o) := A  THEN
28     P(o) := A
29     p := o
30   ELSE IF Kosten geringer wenn P(p) := I
31     P(p) := I
32     p := NEXT(p)
33   ELSE BREAK
34 END
```

Der Algorithmus 4.9 wird zunächst ebenso initialisiert wie Algorithmus 4.8. Allerdings muss die Initialisierung nur einmal am Anfang des Programms durchgeführt werden. Zusätzlich wird ein Zeiger `p` auf das erste Element der Liste gesetzt. Als Invariante in dem Algorithmus gilt, dass `p` immer auf das vorderste Objekt der Liste zeigt, für das ein Aktualisierungsprotokoll verwendet wird.

Wenn ein Datenzugriff ausgeführt wird, muss der Algorithmus darauf reagieren. Er verschiebt das Objekt eventuell an die neue Position in der Liste und setzt das Protokoll entsprechend der neuen Position. Da der Datenaustausch bei synchronisierten Protokollen erst beim nächsten Synchronisationspunkt erfolgt, müssen die Auswirkungen auf andere Objekte noch nicht berechnet werden.

Beim nächsten Synchronisationpunkt beginnt der Algorithmus an der Position von `p` zu testen, ob Datenobjekte das Protokoll wechseln und `p` so umzusetzen, dass die Invariante weiter erfüllt bleibt. Die Schleife wird solange durchlaufen, bis weder das vor `p` stehende Objekt zum Aktualisierungsprotokoll wechselt, noch `p` zum Invalidierungsprotokoll.

Die Rechenzeit der Initialisierung von Algorithmus 4.9 ist immer noch in $O(|\Omega| \cdot log|\Omega|)$. Allerdings wird die Initialisierung nur einmal durchgeführt. Die Rechenzeit der häufig wiederholten Schleife liegt in $O(v)$, wobei v die Anzahl der Datenobjekte ist, für die in der Schleife das verwendetet Protokoll verändert wurde. In der Regel wird v allerdings deutlich kleiner als $|\Omega|$ sein. Dieser Vorteil hat allerdings den Preis, dass die einzelnen Datenzugriffe teurer werden. Die Neupositionierung eines Datenobjektes in der Liste kostet $O(log|\Omega|)$ Zeit. Somit

hängt der Vorteil dieses Algorithmus auch davon ab, wie lang das Intervall zwischen den Schleifendurchläufen ist und wie oft in einem Intervall Zugriffe stattfinden. Findet in jedem Durchlauf ein Zugriff auf jedes Datenobjekt statt, ist die Gesamtzeit wieder in $O(|\Omega| \cdot log|\Omega|)$

Bestimmung der Netzwerkeigenschaften

Der einzige offene Teil des Algorithmus ist die Bestimmung der Netzwerkeigenschaften. Dabei kann die Antwortzeit t einfach gemessen werden. Anders verhält es sich mit der Netzwerkbandbreite, die nicht so einfach direkt gemessen werden kann. Allerdings können Veränderungen der Antwortzeit verwendet werden um zu bestimmen, wann die Netzwerkbandbreite gesättigt ist. Dieses Verfahren ist auch adaptiv, wenn z.B. die Bandbreite mit anderen Nutzern geteilt werden muss und die verfügbare Bandbreite schwankt.

Zu Beginn wird eine niedrige Bandbreite B angenommen. B wird sukzessive vergrößert, bis eine längere Antwortzeit gemessen wird. Dies führt zu einer Reduktion von B.

Alternativ kann bei der Übertragung sehr großer Datenobjekte versucht werden, die Übertragungsdauer zu messen und daraus die Bandbreite abzuleiten.

4.5.2 Protokollwahl bei passiven Zugriffen

Werden passive Zugriffe verwendet, werden keine Bearbeitungsfunktionen des Steering-Systems aufgerufen. Folglich kann dass Steering-System die Zugriffe nicht zählen oder klassifizieren. An Synchronisationspunkten können Schreibzugriffe mittels Abtastung rekonstruiert werden, aber Lesezugriffe bleiben vollkommen verborgen. Das bedeutet, dass das Invalidierungsprotokoll nicht verwendet werden kann, da es Aktualisierungen nur bei Lesezugriffen durchführt, also bei Verwendung des Invalidierungsprotokolls nie eine Aktualisierung durchführen würde.

Da Aktualisierungen sowieso nur in Synchronisationspunkten angewendet werden, könnte man an jedem Synchronisationspunkt eine Leseoperation der Konsistenzschicht aufrufen. Dies wäre ein ähnlicher Ansatz wie die insensitive Abtastung bei passiven Schreibzugriffen. Allerdings führt dieser Ansatz dazu, dass an jedem Synchronisationspunkt alle Aktualisierungen übertragen werden müssen. Daher kann dann immer ein Aktualisierungsprotokoll verwendet werden, bei dem nicht noch zusätzlich Anfragen übertragen werden müssen und Wartezeiten anfallen.

Nur wenn ein Prozess p deutlich längere Intervalle zwischen Synchronisationspunkten hat als der andere Prozess q, kann es vorkommen, dass nicht alle Aktualisierungen von q auf p übertragen werden müssen. Es kann dann derselbe Optimierungsalgorithmus wie bei aktiven Zugriffen (siehe Abschnitt 4.5.1) verwendet werden.

4.5.3 Wahl der Zugriffsmethode bei passiven Zugriffen

Wie im Kapitel 4.4 beschrieben, gibt es bei passiven Zugriffen die Möglichkeit, Schreibzugriffe durch Abtastung zu rekonstruieren. Dabei gibt es zwei Abtastungsmethoden:

- Wert-sensitive Abtastung: Bei jedem Synchronisationspunkt, wird der aktuelle Wert mit dem Wert am letzten Synchronisationspunkt verglichen. Hat sich der Wert geändert, wird die entsprechende Funktion der Datenkonsistenzschicht aufgerufen. Der Vorteil dieser Methode ist, dass nur dann eine Funktionsaufruf statt findet, wenn wirklich ein Wert geändert wurde. Somit werden keine unnötigen Aktualisierungen versendet. Dafür muss eine Kopie jedes Datenobjektes erzeugt werden und bei jedem Synchronisationspunkt mit dem aktuellen Wert verglichen werden.
- Insensitive Abtastung: Bei jedem Synchronisationspunkt wird die entsprechende Funktion der Datenkonsistenzschicht aufgerufen. Der Nachteil ist, dass möglicherweise unnötige Aktualisierungen versandt werden. Dafür müssen Daten nicht gespeichert oder verglichen werden, was Speicherbedarf und Rechenzeit spart.

Offensichtlich muss zwischen niedrigerem Speicherbedarf und Rechenaufwand bei insensitiver Abtastung gegenüber geringerer Kommunikation bei Wert-sensitiver Abtastung abgewogen werden.

Formalisierung

Es gilt zwischen dem benötigtem Ressourcen in Form von Speicherbedarf M, Rechenzeit R und benötigtem Durchsatz D abzuwägen. Dazu werden zunächst die benötigten Ressourcen für ein Datenobjekt o in Abhängigkeit des verwendeten Zugriffsmethode pro Synchronisationspunkt abgeschätzt:

$$R(o, sensitiv) = c_S \cdot S(o) \tag{4.92}$$
$$R(o, insensitiv) = 0 \tag{4.93}$$
$$M(o, sensitiv) = S(o) \tag{4.94}$$
$$M(o, insensitiv) = 0 \tag{4.95}$$
$$D(o, sensitiv) = S(o) \cdot N_w(o) \tag{4.96}$$
$$D(o, insensitiv) = S(o) \tag{4.97}$$

Dabei ist $S(o)$ die Größe von o, $N_w(o) \in [0, 1]$ der Anteil der Epochen in denen o mindestens einmal geschrieben wurde und c_S ist eine Konstante.

Die Abwägung geschieht wieder mittels einer Kostenfunktion K in die drei Komponenten K_R für die Rechenzeit, K_M für den Speicherbedarf und K_D für das Datenvolumen einfließen und durch Konstanten gewichtet werden:

$$K = c_d \cdot K_D + c_M \cdot K_M + c_R \cdot K_R \tag{4.98}$$

Die Kosten für eine Ressourcenkomponente ergeben sich wiederum aus der Summe der Kosten für diese Komponente für alle Datenobjekte. Z.B.

$$K_D = \sum_{\forall o} K_D(o) \tag{4.99}$$

Analog werden auch K_M und K_R berechnet.

Die Kostenfunktionen für jede Komponente können natürlich unterschiedlich gestaltet werden. Zum Beispiel kann man für den Speicherbedarf eine dreistufige Funktion verwenden, je nachdem ob der Arbeitsspeicher ausreicht, Daten

noch auf der Festplatte ausgelagert werden können, oder die Kapazität der Festplatte überstiegen wird. Wird allerdings für alle drei Komponenten eine lineare Kostenfunktion angenommen, ergeben sich folgende Kosten für o:

$$K(o, insensitiv) = c_d S(o) \tag{4.100}$$

$$K(o, sensitiv) = (c_d N_w(o) + c_M + c_S c_R) \cdot S(o) \tag{4.101}$$

Das bedeutet, die Frage welcher Zugriffsmechanismus besser ist, lässt sich dann auf die Frage reduzieren, ob $N_w(o)$ größer oder kleiner als eine Konstante c ist:

$$K(o, insensitiv) > K(o, sensitiv) \tag{4.102}$$

$$\iff c_d S(o) > (c_d N_w(o) + c_M + c_S c_R) \cdot S(o) \tag{4.103}$$

$$\iff c_d > c_d N_w(o) + c_M + c_S c_R \tag{4.104}$$

$$\iff c_d - c_M - c_S c_R > c_d N_w(o) \tag{4.105}$$

$$\iff 1 - \frac{c_M + c_S c_R}{c_d} > N_w(o) \tag{4.106}$$

$$\iff c > N_w(o) \tag{4.107}$$

wobei $c = 1 - \frac{c_M + c_S c_R}{c_d}$ gilt.

Ist c größer als $N_w(o)$, sollte die sensitive Abtastung gewählt werden. Im anderen Fall sollte die die insensitive Abtastung gewählt werden. Wenn der benötigte Durchsatz im Verhältnis zu Rechenzeit und Speicherbedarf hoch gewichtet ist, nähert sich c dem Wert 1 an. Damit muss eine höherer Anteil an Schreiboperationen eine Aktualisierung erfordern, damit die insensitive Abtastung weniger Kosten verursacht. Wenn hingegen der benötigte Durchsatz gegenüber Rechenzeit und Speicherbedarf niedrig gewichtet ist, wird der Wert von c kleiner. Damit muss ein höherer Anteil an Schreiboperationen keine Aktualisierung erfordern, damit die sensitive Abtastung günstiger ist. Falls c einen negativen Wert hat. ist für die gewählten Gewichtungen immer die insensitive Abtastung günstiger.

Diese Bedingung kann für jedes Datenobjekt o entschieden werden sofern man $N_w(o)$ kennt. Solange man mit Wert-sensitiver Abtastung auf o zugreift, verfügt man über diese Information. Wenn allerdings insensitive Abtastung verwendet wird, weiß man nicht, ob in der vergangenen Epoche ein Schreibzugriff stattgefunden hat. Diese Information lässt sich nur gewinnen wenn man den aktuellen Wert mit dem Wert von o beim letzten Synchronisationspunkt vergleicht. Allerdings geht dadurch der Vorteil der insensitiven Abtastung verloren, und man kann dann sowieso Wert-sensitiv zugreifen.

Verwendet man andere Kostenfunktionen, die möglicherweise $N_w(o)$ eliminieren, hängt die Zuweisung nur noch von der Objektgröße ab und die Zugriffsmuster spielen keine Rolle mehr. Man hat daher nur die Möglichkeit Entscheidungen in eine Richtung zu automatisieren: Wenn Wert-sensitive Abtastung verwendet wird und insensitive Abtastung günstiger wäre, kann diese Entscheidung automatisiert getroffen werden.

4.6 Fazit

In diesem Kapitel wurden die Aspekte der Kommunikation, Datenkonsistenz, Optimierung und Datenzugriff beim Online-Steering im Grid betrachtet.

Bei der Kommunikation wurden die Möglichkeiten untersucht eine interaktive Kommunikation mit Gridjobs zu ermöglichen. Dabei sollten die Sicherheit der Sites und der Jobs nicht beeinträchtigt werden. Als Ergebnis wurde ein Konzept entwickelt, das einen Verbindungsaufbau zwei Rechnern im Grid ermöglicht, sowie ein Sicherheitskonzept erstellt. Beim Verbindungsaufbau werden verschiedene Konfigurationen von Firewalls und privaten IP-Netzwerken berücksichtigt. Insofern ist diese Lösung z.B. im LCG [96] anwendbar.

Der Schwerpunkt dieses Kapitels lag auf der Entwicklung eines neuen Modells für Online-Steering. Dabei wird Online-Steering als Zugriffe auf einen gemeinsamen Speicher mit der Anwendung betrachtet. Durch ein geeignetes Konsistenzmodell kann das Verhalten des Steering-Systems definiert werden. Eine der wichtigsten Anforderungen, die an ein Steering-System gestellt wird, ist die Datenkonsistenz.

Ein Vorteil dieses Ansatzes ist die Garantie von Konsistenzbedingungen, um die Integrität der Daten in der Anwendung zu erhalten. Dies kann die Integration des Steering-Systems in eine bestehende Anwendung vereinfachen. Außerdem eröffnet es Optimierungsmöglichkeiten, bei der verschiedene Implementierungen für dasselbe Konsistenzmodell transparent ausgetauscht werden können.

Um die Datenkonsistenz zu erhalten wurden zwei kritische Bedingungen formuliert. Je nach Anwendungsfall müssen die Inter-Prozess und die Intra-Prozess Bedingung oder nur die Intra-Prozess Bedingung eingehalten werden, um Datenkonsistenz zu gewährleisten. Es wurden drei neue Konsistenzmodelle vorgestellt:

- Die Spezielle Schwache Konsistenz (SSK)
- Die Verzögerte Schwache Konsistenz (VSK)
- Die Zeitplankonsistenz

Dabei erfüllt die Zeitplankonsistenz beide Bedingungen und die anderen beiden Konsistenzmodelle die Intra-Prozess Bedingung. Die SSK und die VSK unterschieden sich darin, dass sie ein unterschiedliches Verhalten des Steuerungswerkzeugs umsetzen. Insofern könnte man sich sicherlich zahlreiche andere Konsistenzmodelle vorstellen, die zwar eine oder beide Integritätsbedingungen einhalten, aber ein anderes Verhalten modellieren.

Z.B. könnte man Konsistenzmodelle entwickeln, bei denen die Synchronisationpunkte nicht global, sondern auf einige Datenobjekte beschränkt sind. Eine andere Möglichkeit wäre, Konsistenzmodelle zu entwickeln, die an der Freigabekonsistenz [62] angelehnt sind. Es werden dann allerdings detaillierte Informationen darüber benötigt, welches Datenobjekt von welcher Synchronisationsoperation betroffen ist. Dies macht die Integration in bestehende Anwendungen und Visualisierungswerkzeuge aufwendiger.

Die grundlegende Idee wurde möglichst allgemein formuliert. Diese Allgemeinheit eröffnete aber sehr viele neue Möglichkeiten und auch offenen Fragen. Dies machte es allerdings auch notwendig, sich zunehmend auf ein Kerngebiet zu fokussieren, um sich nicht in der Anzahl der Möglichkeiten zu verlieren. So ist die Beschreibung der Speziellen Schwachen Konsistenz und der Verzögerten Schwachen Konsistenz deutlich detaillierter und umfangreicher ausgefallen als bei der Zeitplankonsistenz. Dies kann zum einen darauf zurück geführt werden, dass die Zeitplankonsistenz auf einem Algorithmus von CUMULVS [133] basiert, während für die beiden anderen Konsistenzmodelle keine Vorlagen existierten.

Für die SSK und VSK wurde jeweils ein Invalidierungs- und ein Aktualisierungsprotokoll neu entwickelt und deren Korrektheit gezeigt. Insbesondere für parallele Anwendungen und kollaboratives Steering könnten sicher noch weitere Konsistenzmodelle und noch mehr Protokolle entwickelt werden, was aber den Rahmen dieser Arbeit sprengen würde.

Der Datenzugriff bildet die Schnittstelle zwischen Steering-System und Anwendung oder Visualisierung. Hierfür wurde analysiert, wie einem Steering-System die Möglichkeit gegeben werden kann, auf Speicherzugriffe der Anwendung zu reagieren. Wichtig sind in diesem Kapitel die Klassifikation von Zugriffsarten, da sie zu unterschiedlichen Bearbeitungsmöglichkeiten führen. Daraus folgt, welche Protokolle angewendet werden können und welche Optimierungsmöglichkeiten bestehen.

Die automatische Optimierung wurde für zwei Möglichkeiten untersucht:

- Die Wahl zwischen Invalidierungsprotokoll und Aktualisierungsprotokoll, falls die Zugriffe aktiv sind.
- Die Wahl zwischen Wert-sensitiver Abtastung und insensitiver Abtastung, falls die Zugriffe passiv sind.

Diese Untersuchungen basieren auf theoretischen Überlegungen und daraus resultierenden Erwartungen über das Verhalten der Protokolle. Insbesondere die Möglichkeit transparent zwischen den Protokollen zu wechseln, um die Performance zu optimieren, war ein Motivationsgrund, sich mit diesem Ansatz für Online-Steering zu beschäftigen. Die theoretische Analyse ergab, dass sich durch die Protokollwahl eine Optimierung der Performance erzielen lässt. Bei der Wahl der Abtastmethode führten die theoretischen Überlegungen zu dem Schluss, dass die Entscheidungsalgorithmen einen möglichen Nutzen egalisieren würden.

Kapitel 5

Implementierung

RMOST (Result Monitoring and Online Steering Tool) [112, 151, 109, 110] ist ein Online-Steering-System, das das Modell aus Kapitel 4 implementiert. In diesem Kapitel wird die Architektur von RMOST beschrieben und auf interessante Implementierungsdetails eingegangen. Eine Übersicht über die Architektur ist in Abbildung 5.1 zu sehen.

Durch das Modell wird eine Grobgliederung der Architektur in vier Teile vorgegeben, die jeweils einer Schicht im Schichtenmodell entsprechen (siehe Abb. 4.1). Jede dieser Schichten soll daher unabhängig von der Implementierung der anderen Schichten funktionieren. Die Implementierung der einzelnen Schichten wird in eigenen Unterkapiteln beschrieben. Für die Zusammenarbeit zwischen den einzelnen Schichten und um die Austauschbarkeit zu garantieren, sind gut definierte Schnittstellen zwischen den Schichten notwendig. Beim Design der Schnittstellen wurde berücksichtigt, dass die Anwendung autonom ausgeführt werden kann und interaktives Steering optional ist. Daher wurde ein Proxy-Mechanismus in den Schnittstellen verwendet. Höhere Schichten brauchen dann keine Kontrollstrukturen und Nebenläufigkeit implementieren, sondern ausschließlich die Funktionalität bei bestimmten Ereignissen. Nebenläufigkeit führt im Regelfall zu einer besseren Ausnutzung der Ressourcen, als bei blockierender Kommunikation.

Die einzelnen Schnittstellen werden jeweils in dem Unterkapitel der Schicht behandelt, die die Funktionalität zur Verfügung stellt.

5.1 Die Kommunikationsschicht

Dieses Kapitel beschreibt die Implementierung der Kommunikationsschicht. Zuerst wird auf die Definition der Schnittstelle eingegangen, danach wird die Architektur der Gridverbindung beschrieben. Abschließend wird auf die Instanzierung der Kommunikationsschicht eingegangen.

5.1.1 Die Schnittstelle

Beim Design der Kommunikationsschicht wurde berücksichtigt, dass die Gridverbindung zusätzlich in einen Produktionsjob eingebunden werden sollte, in

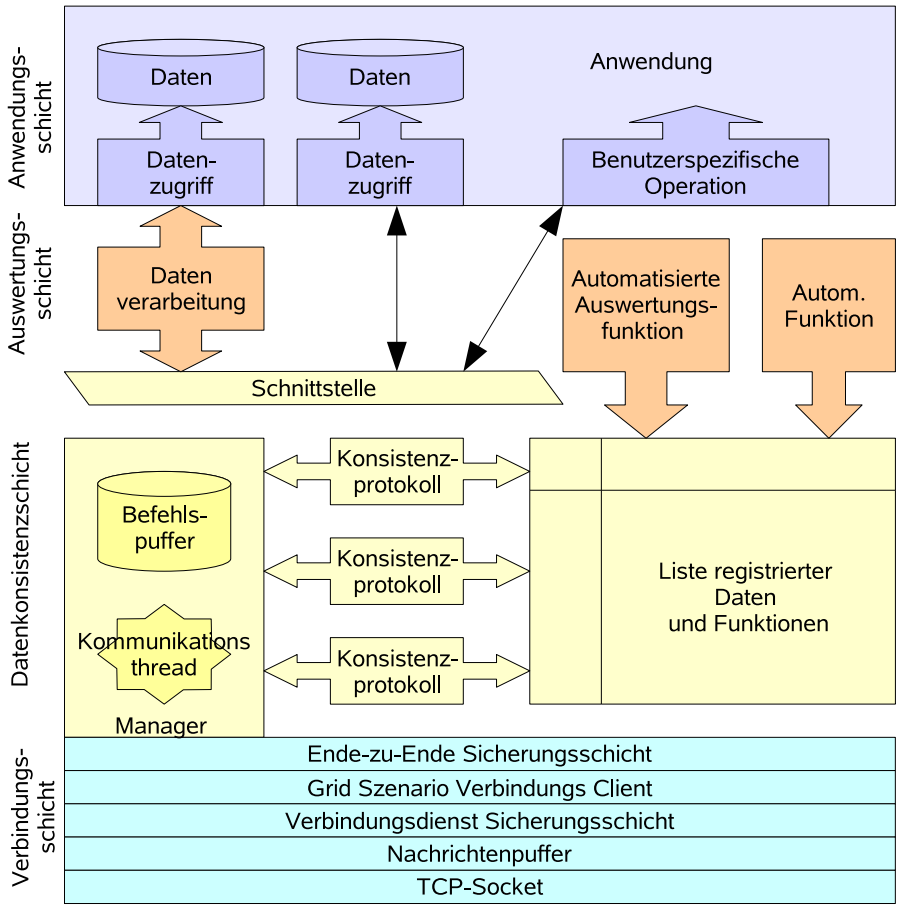


Abbildung 5.1: Architektur von RMOST

der die Kommunikation durch das Steering'nebenher' mitläuft. Deshalb wurden folgende Prinzipien beim Design der Schnittstelle angewendet:

- Nichtblockierende Methoden.
- Proxy-Mechanismus bei Ereignissen, die möglicherweise eine Nutzerreaktion auslösen.
- Leichtgewichtige Lösung (ohne eigene Threads)
- Flexible Adressdefinition, da die Schnittstelle nicht auf eine bestimmte Gridmiddleware beschränkt sein sollte. Außerdem sollte auch die Möglichkeit bestehen beliebige Kommunikationskanäle zu wählen.
- Möglichkeit mehrere Verbindungen zu verwalten.
- Die Daten einer Verbindung sollen in der Reihenfolge ankommen, in der sie gesendet wurden. Dies ist die intuitive Erwartung von Anwendern und kommt daher der Benutzbarkeit entgegen.

Die Schnittstelle besteht aus zwei Teilen. Auf der einen Seite stehen die Methoden, die von der Kommunikationsschicht angeboten werden und von den Benutzern der Kommunikation aufgerufen werden. Diese Schnittstelle ist durch die abstrakte Klasse `RM_IMessageLayer` definiert. Auf der anderen Seite der Schnittstelle stehen die Proxy-Methoden, die von Kommunikationsschicht aufgerufen werden können, falls ein Ereignis eintritt. Die Proxy Methoden werden durch die abstrakte Klasse `RM_IMessageUser` definiert.

Die Adresse wird in einer Zeichenkette übergeben. Dies ermöglicht es beliebige Adressarten (z.B. IP, URL, Jobidentifier) als Adressen zu verwenden. Das Format der Zeichenkette hängt von der jeweiligen Implementierung ab. Zum Beispiel ist das Format einer TCP Verbindung `<Hostname>:<Port>`, während es für die Gridverbindung der Jobname ist.

Um der Verbindungsschicht mitzuteilen, dass man eine Verbindung aufbauen möchte, wird die `connect()`-Methode aufgerufen. Die `connect()`-Methode erhält als Parameter die Adresse, zu der man sich verbinden möchte. Zurückgegeben wird eine ganze Zahl, die Verbindungs-ID genannt wird und die initiierte Verbindung spezifiziert. Ist der Verbindungsaufbau abgeschlossen oder fehlgeschlagen, wird die Proxy-Methode `onConnected()` aufgerufen. Bei komplexeren Verbindungstypen wird auch bei jedem Fortschritt `onConnected()` mit dem jeweiligen Status aufgerufen.

Um der Verbindungsschicht mitzuteilen, dass man Verbindungen annehmen möchte, wird die `listen()` Methode aufgerufen. Sie enthält ebenfalls eine Zeichenkette als Parameter, in der relevante Adressinformationen übergeben werden können. Zum Beispiel wird bei einer TCP-Verbindung eine Portnummer übergeben, oder bei einer Gridverbindung die Adresse eines Verbindungsdienstes. Der Rückgabewert ist eine ganze Zahl, die den Annahmewunsch spezifiziert. Ist eine Verbindungsanfrage vorhanden, wird die `onConnection()`-Methode vom `RM_IMessageUser` aufgerufen, welche die einzige Rückrufmethode ist, die einen Wert zurückgibt. Liefert `onConnection()` den Wert `true` zurück, wird die Verbindung angenommen, ansonsten wird sie abgelehnt. Der `onConnection()`-Methode wird eine Verbindungs-ID übergeben, die die neue Verbindung spezifiziert, falls sie angenommen wird.

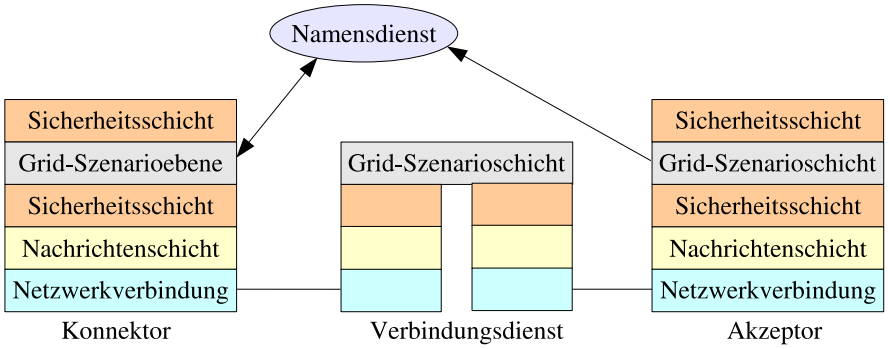


Abbildung 5.2: Die Architektur der Kommunikationsschicht

Besteht eine Verbindung, können Daten mit `sendMessage()` gesendet und mit `getMessage()` empfangen werden. Beide Methoden sind nicht-blockierend. Die Verbindung, auf der Daten gesendet bzw. empfangen werden, wird mit der Verbindungs-ID spezifiziert. Wurde die Nachricht vollständig versendet, wird die Proxy-Methode `onDataWritten()` aufgerufen. Ist eine Nachricht vollständig angekommen, wird die `onDataAvailable()`-Methode aufgerufen. Die Nachricht kann dann mit `getMessage()` abgerufen werden. Ein Aufruf von `getMessage()` liefert `NULL` zurück, wenn keine vollständige Nachricht vorliegt.

Die `check()`-Methode führt die internen Operationen aus, wie das Versenden von wartenden Nachrichten, Annehmen von neuen Verbindungen und Empfangen von Daten. Aus der `check()`-Methode werden die Proxy-Methoden aufgerufen falls man eine `RM_IMessageUser` Instanz mit `setCallbackFunction` registriert hat. Ist der `RM_IMessageLayer` Instanz keine `RM_IMessageUser` Instanz bekannt, wird keine Proxy-Methode aufgerufen.

Eine detailierte Beschreibung der Implementierung und Beispiele für die Verwendung der Kommunikationschicht sind in [109] enthalten.

5.1.2 Architektur

Aus dem Mechanismus zum Verbindungsaufbau (siehe Kap. 4.1) ist ersichtlich, dass der Kommunikationskanal aus mehreren Teilen besteht, für die ein entsprechendes Transportprotokoll verwendet werden kann. Diese einzelnen Teilverbindungen müssen jeweils authentifiziert, autorisiert und auf Integrität geprüft werden. Außerdem wird eine Ende-zu-Ende Authentifizierung, Autorisierung, Integrität und optionale Verschlüsselung benötigt, die auf der Ende-zu-Ende Kommunikation aufsetzt. Daraus resultiert eine Architektur (siehe Abb. 5.2), die aus mehreren Schichten besteht: Der Netzwerkverbindung, der Nachrichtenschicht, der ersten Sicherungsschicht, der Grid-Szenarioschicht und der Ende-zu-Ende Sicherheitsschicht. Abgesehen von der Netzwerkverbindung implementieren alle Schichten die `RM_IMessageLayer` Schnittstelle.

Wird eine Nachricht von einem Endpunkt der Kommunikation versendet, wird sie von der Ende-zu-Ende Sicherheitsschicht in ein Paket verpackt und an die Grid-Szenarioschicht weitergegeben. Die Grid-Szenarioschicht kennt den passenden Verbindungsdienst (VD) und reicht die Nachricht an die Sicherheitsschicht weiter. Hier wird die Nachricht wiederum mit den nötigen Sicherheitsin-

formationen verpackt und an die Nachrichtenschicht weitergegeben. Die Nachrichtenschicht speichert die Nachricht zwischen und sendet die Nachricht in einem oder mehreren Schritten über die Netzwerkschicht zum VD. Die Netzwerkschicht des VD empfängt die Nachricht Stück für Stück. Immer wenn neue Daten vorhanden sind, liest die Nachrichtenschicht des Verbindungsdienstes die empfangenen Daten. Wurde eine Nachricht vollständig empfangen, wird sie an die darüber liegende Sicherheitsschicht weitergegeben. Die Sicherheitsschicht führt die Sicherheitsüberprüfungen durch. War die Sicherheitsprüfung erfolgreich, wird die Nachricht an die Grid-Szenarioschicht des VD übergeben. Dieser ermittelt das nächste Ziel und übergibt die Nachricht der Sicherheitsschicht für die Verbindung zum Zielrechner. Hier durchläuft die Nachricht wieder die Sicherheits-. Nachrichten- und Netzwerkschicht von VD und Zielrechner. Bei erfolgreicher Sicherheitsprüfung der Teilverbindung wird die Nachricht von der unteren Sicherheitsschicht durch die Grid-Szenarioschicht zur Ende-zu-Ende Sicherheitsschicht weitergegeben. Hier werden die Ende-zu-Ende Sicherheitstest durchgeführt. Wenn sie erfolgreich abgeschlossen werden, kann die Nachricht von der Anwendung gelesen werden.

Die Netzwerkverbindung Die Gridverbindung setzt auf existierenden Verbindungsprotokollen auf. Für die Implementierung wurde als Basis das TCP Protokoll ausgewählt. Da das Internet als Kommunikationsinfrastruktur verwendet wird, ist TCP allgemein verfügbar, es ist verbindungsorientiert und zuverlässig. Außerdem garantiert es das gewünschte FIFO-Verhalten, bei dem die Daten in der Reihenfolge ankommen, in der sie versendet werden. Die Netzwerkverbindungsschicht wurde in der Klasse `RM_TCP_Connection` implementiert und verwaltet die offenen Verbindungen. Sie implementiert eine Teilmenge der Kommunikationsschnittstelle, da die Netzwerkverbindung noch nicht nachrichtenbasiert ist. Das Adressenformat ist `<Hostname>:<Port >`, zum Beispiel:

`myhost.mydomain.de:12345`.

Die Nachrichtenschicht Die Nachrichtenschicht setzt auf einer Netzwerkverbindung auf. Sie stellt sicher, dass immer nur ganze Nachrichten gesendet oder empfangen werden. Dafür werden zu sendende Nachrichten zwischengespeichert. Bei jedem Aufruf von `check()` wird versucht auf jeder Verbindung soviel Daten wie möglich zu senden und zu empfangen. Empfangene Daten werden zwischengespeichert bis die Nachricht vollständig ist, dann wird `onDataAvailable` des registrieten Benutzers aufgerufen. Um überprüfen zu können ob eine Nachricht vollständig ist, wird jeder Nachricht ein Header vorangestellt, der die Länge der Nachricht enthält. Die Nachrichtenschicht ist in der Klasse `RM_Tokenizer` implementiert. Das Adressformat ist dasselbe wie bei der zugrundeliegenden Netzwerkverbindung.

Die Sicherheitsschicht Die Sicherheitsschicht sorgt für Authentifizierung, Autorisierung und Integrität, optional kann die Kommunikation auch verschlüsselt werden. Die Implementierung verwendet die Generic Security System API (GSSAPI) [68, 170]. Dadurch kann die Implementierung mit jedem Sicherheitssystem gebaut werden, das eine GSSAPI anbietet und ist nicht auf ein bestimmtes System beschränkt. Sie wurde mit Globus TK2 [52] und Globus TK4 [51, 164] gebaut.

Die Sicherheitsschicht wird zwei mal verwendet: Einmal um die Teilverbindungen zum Verbindungsdienst zu sichern und zweitens für die Ende-zu-Ende Sicherheit. Die Klasse `RM_SecurityWrapper` implementiert die Sicherheitsschicht. Die Sicherheitsschicht verwendet das Adressformat der darunter liegenden Schicht. Sie implementiert gegenseitige Authentifizierung, einen Integritätstest, einen Replay-Check und optional Verschlüsselung. Die GSSAPI fügt der Nachricht Informationen hinzu, oder verschlüsselt sogar die gesamte Nachricht. Die Struktur einer Nachricht, die mit der GSSAPI verpackt wurde, bleibt allerdings für den Benutzer verborgen. Jedes Sicherheitssystem kann seine eigene Nachrichtenstruktur verwenden. Das verwendete Zertifikat wird mit der Methode `initializeCredentials()` bestimmt. Standardeinstellung ist die Verwendung der ersten gefundenen Identität. Alternativ kann auch ein Name angegeben werden, für den dann das Zertifikat gesucht wird.

Die Grid-Szenarioschicht In der Grid-Szenarioschicht wird das passende Szenario ermittelt und die Kommunikation entsprechend dem Szenario umgesetzt, z.B. der Job registriert, der Verbindungsdienst kontaktiert, etc. Sie ist in der Klasse `RMML_Routed` implementiert. Im Moment ist die Szenariowahl nicht automatisiert, sondern kann vom Administrator mittels der Umgebungsvariable `RM_CONNECTION_SCENARIO` festgelegt werden. Wird kein Szenario explizit konfiguriert, wird das halboffene Szenario verwendet. Der Verbindungsdienst kann ebenfalls mittels der Umgebungsvariable `RM_CONNECTION_SERVICE` festgelegt werden. Falls die Umgebungsvariable nicht gesetzt ist, kann ein VD in der `listen()` Methode angegeben werden. Der VD wird durch seine Adresse spezifiziert, die in dem Adressformat der Nachrichtenschicht unter `RMML_Routed` gegeben sein sollte. `RMML_Routed` selbst verwendet den Jobnamen als Adresse. Um den Namensdienst leicht austauschen zu können, wurde ein Plugin-Mechanismus implementiert. Im geschlossenen Szenario wird der Namensdienst nicht direkt kontaktiert, sondern eine Nachricht an den VD geschickt, der die Registrierung beim ND vornimmt.

Der Namensdienst Für den Namensdienst kann dynamisch ein Plugin geladen werden. Zurzeit existieren zwei Plugins.

- Für gLite wird als Namensdienst R-GMA [30] verwendet. R-GMA ist ein Monitoringsystem und wird ähnlich wie eine verteilte Datenbank verwendet. R-GMA besitzt verschiedene Tabellen, in die neue Zeilen eingetragen werden können. Für den Namensdienst wurde die Tabelle `userTable` verwendet. Eingetragen wird der Jobname und die Kontaktinformationen bestehend aus dem Adressstring für den VD und dem Adressstring für den Job, getrennt durch einen doppelten Doppelpunkt '::'. R-GMA besitzt seine eigene Kommunikationsstruktur.

- Für Testzwecke wurde ein einfacher eigener Namensdienst entwickelt, der allerdings nicht skaliert und daher nicht für den Produktionseinsatz geeignet ist.

Die Klasse`RM_INameService` enthält die Schnittstellendefinition die der Namensdienst bereit stellt.

Der Verbindungsdienst Der Verbindungsdienst verlangt eine Authentifizierung aller Verbindungen. Die zwei Teilverbindungen zum Akzeptor und zum Konnektor werden von dem VD statisch verknüpft. Alle Nachrichten die den Verbindungsdienst auf einer Verbindung erreichen, werden auf der jeweils anderen Verbindung weitergeleitet. Wird eine Teilverbindung abgebaut, wird auch die andere Teilverbindung geschlossen. Unterhalb der Grid-Szenarioschicht verwendet der Verbindungsdienst dieselbe TCP Schicht, Nachrichtenschicht und Sicherheitsschicht wie die anderen Komponenten auch. Die Grid-Szenario Schicht wurde durch eine Implementierung mit dem VD spezifischen Verhaltens ersetzt.

5.1.3 Instanzierung

Im Allgemeinen werden die verschiedenen Kommunikationsschichten von unten nach oben aufeinandergeschichtet, wobei jeweils der nächst höheren Schicht ein Zeiger auf die darunter liegende Schicht übergeben wird. Anschließend werden Schicht-spezifische Konfigurationen an der jeweiligen Schicht vorgenommen, wie z.B. in Listing 5.1 gezeigt wird.

Listing 5.1: Instanzierung einer Gridverbindung

```
1  RM_TCP_Connection   *tcp = NULL;
2  RM_Tokenizer        *tok = NULL;
3  RM_SecurityWrapper  *sec = NULL;
4  RMML_Routed         *rou = NULL;
5
6  try
7  {
8    tcp = new RM_TCP_Connection();
9    tok = new RM_Tokenizer(tcp);
10   sec = new RM_SecurityWrapper(tok);
11   sec->initializeCredentials(NULL);
12   rou = new RMML_Routed(sec);
13   rou->initializeJobID("EDG_WL_JOBID");
14   rou->initializePortRange("GLOBUS_TCP_PORT_RANGE");
15   rou->openPlugin("rmost_ns_defaultplugin.so");
16   sec = new RM_SecurityWrapper(rou, RM_SecurityWrapper::
         OnlySelf);
17   sec->initializeCredentials(NULL);
18 }
19 catch (RM_IException *e)
20 {
21   cout << "ERROR: " << e->getMessage()
22        << "\nat " << e->getPlace() << endl;
23   delete e;
24 }
```

Der Konstruktor der oberen Schichten setzt sich selbst als Empfänger der Aufrufe von Rückrufmethoden der jeweils darunter liegenden Schicht. Falls ein Fehler auftritt, wird eine Ausnahme geworfen, die vom Typ `RM_IException` abgeleitet ist und einen Nachrichtentext mit der Beschreibung des Fehlers enthält, sowie eine Beschreibung des Ortes an dem der Fehler aufgetreten ist.

Für die Sicherheitsschicht `RM_SecurityWrapper` müssen noch die verwendeten Credentials geladen werden. Dazu dient die `initializeCredentials()`-Methode, der eine Zeichenkette übergeben wird, zu dem die passenden Credentials geladen werden, falls sie vorhanden sind. Wird NULL übergeben, werden die ersten gefundenen Credentials verwendet. Dem Konstruktor kann eine Option übergeben werden, die angibt, ob alle authentifizierten Verbindungsanfragen akzeptiert werden (`RM_SecurityWrapper::AcceptAll`) oder ob nur Verbindungsanfragen angenommen werden, wenn die Credentials auf beiden Seiten gleich sind (`RM_SecurityWrapper::OnlySelf`).

Die Gridszenarioschicht `RMML_Routed` liest den Jobnamen und einen Portbereich, in dem der Listener geöffnet werden darf, aus Umgebungsvariablen aus (Zeile 13 und 14), deren Namen den entsprechenden Funktionen übergeben wird. Dies ist nur notwendig, wenn diese Instanz Verbindungen annehmen soll, ansonsten brauchen diese Werte nicht initialisiert zu werden.

In Zeile 15 wird das Plugin des verwendeten Namensdienstes geladen. Die an `openPlugin` übergebene Zeichenkette enthält den Namen der Bibliothek, in der die Funktionen zum Zugriff auf den Namensdienst implementiert sind. Diese Funktionen sind in `RM_INameService` definiert.

5.1.4 Verbindungsabbau

Wenn eine Anwendung die Verbindung abbauen möchte, ruft sie die `close()`-Methode der obersten Schicht auf. Dieser Aufruf wird von jeder Schicht jeweils an die darunter liegende Schichte weitergeleitet. Dabei werden evtl. noch im Puffer befindliche Nachrichten versendet. Die TCP-Schicht beendet die TCP-Verbindung. Wenn die TCP-Verbindung geschlossen wird, ruft die TCP-Verbindung die `onClose()`-Methode der darüber liegenden Schicht auf. Jetzt werden die Datenstrukturen abgebaut wie z.B. der Sicherheitskontext. Die oberste Schicht ruft die `onClose`-Methode der Anwendung auf.

Stellt eine Seite fest, dass die TCP-Verbindung abgebaut oder unterbrochen wurde, werden ebenfalls die `onClose`-Methode der höheren Schichten aufgerufen, die die Datenstrukturen abbauen, die zu dieser Verbindung gehören. Die oberste Schicht benachrichtigt wieder die Anwendung, dass die Verbindung geschlossen wurde.

Wird eine Verbindung zu einem Verbindungsdienst abgebaut, überprüft dieser, ob die Daten dieser Verbindung auf einer anderen Verbindung weitergeleitet wurden. Wenn ja, wird diese Verbindung ebenfalls abgebaut. So wird sichergestellt, dass die Verbindungen zwischen zwei Endpunkten vollständig abgebaut werden, auch wenn sie über einen Verbindungsdienst geleitet werden.

5.2 Die Datenkonsistenzschicht

Die Datenkonsistenzschicht implementiert die Datenkonsistenzmodelle, die für die Wahrung der Datenintegrität benötigt werden. Die Implementierung ist zunächst nur für sequentielle Jobs erfolgt, da ja auch nur Protokolle für sequentielle Jobs in Kap. 4.3 entwickelt wurden. Von den in Kapitel 4.2.3 beschriebenen Konsistenzmodellen wurden die PRAM Konsistenz [108], die Schwache Verzögerte Konsistenz (siehe Kap. 4.2.3) und die Spezielle Schwache Konsistenz (siehe Kap. 4.2.3) implementiert.

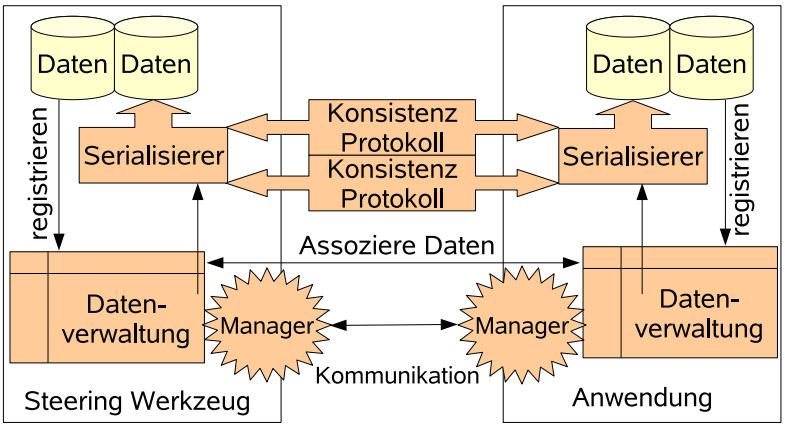


Abbildung 5.3: Funktion der Komponenten der Datenkonsistenzschicht

Für die Datenkonsistenzschicht wurde ein Framework entworfen, das die Verwendung von mehreren Konsistenzmodellen und Protokollimplementierungen erlaubt und leicht um weitere Protokolle erweitert werden kann. Die allgemeine Architektur des Frameworks und deren Schnittstellen sind in Kap. 5.2.1 beschrieben. Anschließend wird in Kap. 5.2.3 die Implementierung verschiedener Protokolle erläutert.

5.2.1 Architektur der Datenkonsistenzschicht

Die Wahrung der Datenkonsistenz lässt sich in folgende Teilaspekte aufteilen (siehe Abb. 5.3):

- Verwaltung der bekannten Datenobjekte.
- Bearbeitung eingehender Nachrichten.
- Reaktion auf Zugriffe durch ein Konsistenzprotokoll.

Die Datenkonsistenzschicht arbeitet auf einer serialisierten Darstellung der Daten. Die dafür benötigten Serialisierungsfunktionen werden von der Datenzugriffsschicht bereit gestellt. Ebenso ist das Auslösen von Aktionen der Datenkonsistenzschicht Aufgabe der Datenzugriffsschicht. Im Folgenden wird die Aufgabe und Funktionsweise jeder Komponente genauer erklärt. Wie Zugriffe der Anwendung abgefangen werden wird erst in Kap. 5.4 erklärt.

Die Datenverwaltung

Die Datenverwaltung enthält eine Liste mit allen Datenobjekten, auf die das Steering-System zugreifen kann, und den zugehörigen Informationen für das Steering-System. Wenn eine Anwendung Daten hat, auf die das Steering-System zugreifen soll, muss sie diese Daten bei der Datenverwaltung registrieren. Dabei müssen gleichzeitig die Serialisierungfunktionen mitgegeben werden. Außerdem kann die Anwendung ein bestimmtes Protokoll oder Konsistenzmodell wählen.

Jedem Datenobjekt wird ein eindeutiger Name zugeordnet, der Bindungsname, der der Zuordnung von Datenobjekten in verschiedenen Prozessen dient.

Haben zwei Prozesse Datenobjekte mit demselben Bindungsnamen registriert, werden diese als lokale Kopien desselben, gemeinsam genutzten Datenobjektes behandelt. Das bedeutet, dass Änderungen an diesem Datenobjekt in einem Prozess dazu führen, dass auch der Wert des Datenobjektes in dem anderen Prozess aktualisiert wird.

Die Datenverwaltung ist in der Klasse `RM_Registry` implementiert. Für jedes registrierte Datenobjekt enthält die Registrierung eine Instanz der Klasse `RM_VarEntry`, in der alle notwendigen Daten für die implementierten Protokolle für dieses Datenobjekt gespeichert sind.

Die Konsistenzprotokolle

Wenn von der Datenzugriffsschicht ein Datenzugriff gemeldet wird, wird für das entsprechende Objekt aus der Datenverwaltung das passende Konsistenzprotokoll herausgesucht und zur Behandlung des Datenzugriffs aufgerufen. Es hat die Aufgabe, die Konsistenzbedingungen bei Zugriffen zu überprüfen und gegebenenfalls Aktionen zur Wahrung der Konsistenz durchzuführen. Dazu kann es über den Manager Nachrichten an sein Pendant in anderen Prozessen verschicken. Außerdem bearbeitet das Konsistenzprotokoll eingegangene Nachrichten von anderen Prozessen.

Die Konsistenzmodelle werden durch entsprechende Konsistenzprotokolle umgesetzt. Die Datenkonsistenzschicht soll die gleichzeitige Verwendung verschiedener Konsistenzmodelle unterstützen. Für jedes Konsistenzmodell kann es mehrere Protokolle geben. Daher wurde eine Architektur entwickelt, die dynamisch um zusätzliche Protokolle erweitert werden kann. Jedes Protokoll ist in einer eigenen Klasse implementiert. Tritt ein relevantes Ereignis auf (z.B. Zugriffsoperationen oder Empfang von Nachrichten) werden entsprechende Methoden des Protokolls aufgerufen. Diese Methoden sind in der Schnittstelle `RM_IDataHandler` definiert, von der jedes Protokoll abgeleitet werden muss.

Der Manager

Der Manager wickelt die Kommunikation zwischen den Prozessen ab. Jedes Protokoll, das empfangene Nachrichten bearbeiten will, muss die Nachrichtentypen, die es bearbeitet, beim Manager registrieren. Auch der Nachrichtenversand der Protokolle erfolgt über den Manager.

Der Nachrichtenkopf besteht aus einem Nachrichtentypcode. Werden Nachrichten empfangen, entscheidet der Manager anhand des Nachrichtentyps von welchem Protokoll diese Nachricht bearbeitet wird und übergibt die Nachricht an das entsprechende Protokoll. Außerdem entscheidet der Manager, ob die Nachricht sofort nach dem Empfang asynchron bearbeitet wird oder ob sie erst beim nächsten Synchronisationspunkt bearbeitet werden soll. In diesem Fall wird die Nachricht vom Manager zwischengespeichert. Wurden alle Nachrichten empfangen, die in einer Epoche und anschließendem Synchronisationspunkt versendet wurden, werden im nächsten Synchronisationspunkt alle Nachrichten dieser Epoche an das entsprechende Protokoll übergeben und bearbeitet. Der Manager ist in der Klasse `RM_ConsistencyManager` implementiert.

Wie in Abbildung 5.1 gezeigt, erfolgt der Zugriff auf die gesamte Datenkonsistenzschicht über eine Schnittstellenklasse, die in `RM_IDataConsistencyLayer` implementiert ist.

5.2.2 Schnittstellen der Datenkonsistenzschicht

Die Datenkonsistenzschicht definiert mehrere Schnittstellen. Die wichtigste Schnittstelle ist die Klasse `RM_IDataConsistencyLayer`, die die Methoden definiert, über die auf die Datenkonsistenzschicht zugegriffen werden kann. Weitere verwendete Schnittestellen sind:

- `RM_IDataHandler` definiert die Methoden, die ein Konsistenzprotokoll implementieren muss. Allerdings ist die Verwendung nicht auf Konsistenzprotokolle beschränkt, sondern kann beispielsweise auch für automatisierte Auswertungen verwendet werden.

- `RM_IHandlerSupport` definiert zusätzliche Methoden, die von Konsistenzprotokollen zur direkten Kommunikation mit Instanzen desselben Protokolls in anderen Prozessen benötigt werden.

- `RM_IDataConsistencyUser` definiert eine Reihe von Methoden, die von der Datenkonsistenzschicht aufgerufen werden, wenn ein bestimmtes Ereignis eingetreten ist. Diese können benötigt werden, wenn die Anwendung oder Visualisierung auf bestimmte Ereignisse reagieren möchte. Zum Beispiel kann die graphische Anzeige aktualisiert werden oder eine Warnung ausgeben werden, dass die Verbindung beendet wurde.

- `RM_ISerializeMethod` und `RM_IDeserializeMethod` definieren die Schnittstellen der Serialisierungsmethoden für "kleine" Datenobjekte, während `RM_ISerializeFiles` und `RM_IDeserializeFiles` Serialisierungsmethoden für "große" Datenobjekte definieren. Sie werden ausführlich in Kapitel 5.4 behandelt.

Im verbleibenden Teil dieses Kapitels werden die wesentlichen Elemente der Schnittstellen beschrieben. Eine vollständige Auflistung der Schnittstellen findet sich in [109] und [110].

RM_IDataConsistencyLayer

Die Klasse `RM_IDataConsistencyLayer` kapselt die benötigten Funktionen der Datenkonsistenzschicht, die von den nächsthöheren Schichten benötigt werden. Die wichtigsten Funktionen dieser Klasse sind in Listing 5.2 aufgelistet.

Listing 5.2: Auszug aus RM_IDataConsistencyLayer: Schnittstelle der Datenkonsistenzschicht

```
1  virtual bool registerValue (RM_DataType dtype,
2                      const std::string name,
3                      RM_ISerializeMethod *Serialize,
4                      RM_IDeserializeMethod *Deserialize,
5                      RM_IDataHandler *handler = NULL)=0;
6
7  virtual bool registerStream(RM_DataType dtype,
8                      const std::string name,
9                      RM_ISerializeFiles *Serialize,
10                     RM_IDeserializeFiles *Deserialize,
11                     RM_IDataHandler *handler = NULL)=0;
```

```
12
13 virtual bool unregister(const std::string name)=0;
14
15 virtual void check() throw(RM_IException*)=0;
16
17 virtual bool initialize(RM_IMessageLayer *ML)=0;
18
19 virtual bool finalize()=0;
20
21 virtual void requestValue(const std::string name,
22                           RM_filesize_t offset = 0,
23                           RM_filesize_t length = -1,
24                           std::istream *param = NULL)=0;
25
26 virtual bool synchronize(const std::string name,
27                           RM_filesize_t offset = 0,
28                           RM_filesize_t length = -1,
29                           std::istream *param = 0)=0;
30
31 virtual void addHandler(RM_IDataHandler *handler)=0;
32
33 virtual RM_IHandlerSupport* getHandlerSupport()=0;
```

Um Datenobjekte der Datenkonsistenzschicht zugänglich zu machen, müssen sie registriert werden. Dazu stehen die beiden Methoden `registerValue` und `registerStream` bereit, wobei Erstere Daten registriert, deren Konsistenz immer als Ganzes behandelt wird, während Letztere für Daten verwendet wird, die in Teilen bearbeitet werden können. Beiden Methoden muss ein Datentypidentifier übergeben werden, sowie der Bindungsname und die entsprechenden Serialisierungsmethoden. Optional kann auch ein Protokoll spezifiziert werden, ansonsten wird das Standardprotokoll verwendet. Natürlich kann eine Variable auch wieder deregistriert werden. Dies geschieht durch die Methode `unregister`, der der Bindungsname des Datenobjektes übergeben wird.

Wird ein aktiver Datenzugriff ausgeführt, muss die entsprechende Methode der Datenkonsistenzschicht aufgerufen werden, die auf diesen Datenzugriff reagiert. Dies ist entweder `requestValue`, `synchronize` oder `check`. Dabei gehört `requestValue` zur Leseoperation, `synchronize` reagiert auf eine Schreiboperation und `check` gehört zum Synchronisationspunkt. Den Methoden `requestValue` und `synchronize` wird der Bindungsname, evtl. der Abschnitt des Datenobjektes auf den zugegriffen wurde und optional noch Parameter, die der Serialisierungsmethode übergeben werden, übergeben.

Neue Protokolle können mit `addHandler` hinzugefügt werden. Alle verwendeten Konsistenzprotokolle müssen vor ihrer Verwendung mit `addHandler` hinzugefügt werden. Möglicherweise fügt eine Implementierung einige Protokolle automatisch hinzu. In der Regel wird das neue Protokoll bei der Konstruktion einen Zeiger auf die zugehörige Instanz einer `RM_IHandlerSupport` Implementierung benötigen. Diese kann man mit `getHandlerSupport` bekommen.

Bevor die Datenkonsistenzschicht verwendet wird, muss sie initialisiert werden. Dazu dient die Methode `initialize`, der eine Kommunikationsschicht übergeben werden muss, die zum Datenaustausch verwendet wird. Bevor die

Datenkonsistenzschicht zerstört wird, muss wiederum die Methode `finalize` aufgerufen werden.

Natürlich gibt es etliche weitere Funktionen, die es erlauben zahlreiche Eigenschaften zu setzen und abzufragen. So können beispielsweise Protokolle und Serialisierungsmethoden abgefragt und geändert werden. Für eine detaillierte Beschreibung der Schnittstellen und der Implementierung sowie Beispielen zur Verwendung der Datenkonsistenzschicht sei auch auf [109] und [110] verwiesen.

RM_IDataConsistencyUser

In der Klasse `RM_IDataConsistencyUser` werden Funktionen definiert, die von der Datenkonsistenzschicht aufgerufen werden, wenn bestimmte Ereignisse auftreten. Diese Ereignisse sind:

- `onDataAvailable` wird aufgerufen, wenn ein neuer Wert für ein Datenobjekt empfangen wurde. Als Parameter wird der Name des Datenobjektes übergeben. Diese Methode muss von dem entsprechenden Protokoll aufgerufen werden. Sie erlaubt dem Benutzer der Datenkonsistenzschicht ein spezifisches Verhalten zu implementieren, z.B. die Anzeige zu aktualisieren.

- `onRequest` wird aufgerufen, wenn der andere Prozess einen Wert angefragt hat. Als Parameter wird der Name des Datenobjektes übergeben. Auch diese Methode muss von dem Protokoll aufgerufen werden.

- `onClose` wird aufgerufen wenn, die Verbindung getrennt wurde. Diese Methode wird von der Klasse `RM_ConsistencyManager` aufgerufen.

- `onNewConnection` wird vom `RM_ConsistencyManager` aufgerufen, wenn eine Verbindung hergestellt wurde.

- `onError` kann aufgerufen werden, wenn ein Fehler aufgetreten ist. Sie kann dazu verwendet werden anwendungspezifische Anzeigen von Fehlermeldungen zu implementieren. Als Parameter wird ein Zeiger auf ein Objekt vom Typ `RM_IException` übergeben.

- `onTableReceived` wird aufgerufen, wenn eine Liste der verfügbaren Datenobjekte empfangen wurde. Als Parameter wird eine Liste der registrierten Daten übergeben.

RM_IDataHandler

Die Klasse `RM_IDataHandler` definiert die Schnittstelle, die Konsistenzprotokolle implementieren müssen. Allerdings ist die Verwendung dieser Schnittstelle nicht auf Konsistenzprotokolle beschränkt. Sie kann auch verwendet werden um beispielsweise eine automatisierte Auswertung zu integrieren. Ein Auszug mit den wichtigsten Methoden der Klasse ist in Listing 5.3 aufgeführt.

Listing 5.3: Auszug aus RM_IDataHandler: Schnittstelle der Konsistenzprotokolle

```
1 virtual void onMessage (RM_messagetype_t code, std::
    istream *data)
```

```
2                   throw (RM_IException *) = 0;
3
4  virtual void check () throw (RM_IException *) = 0;
5
6  virtual void onWrite(std::string name,
7                       RM_filesize_t offset = 0,
8                       RM_filesize_t size = -1,
9                       std::istream *param = NULL) = 0;
10
11 virtual void onRead(std::string name,
12                      RM_filesize_t offset = 0,
13                      RM_filesize_t size = -1,
14                      std::istream *param = NULL) = 0;
15
16 virtual int getType() = 0;
```

Die Konsistenzprotokolle sollen auf Lese- und Schreibzugriffe reagieren. Dafür stehen die Methoden `onRead` vor dem Lesen bzw. `onWrite` nach dem Schreiben zur Verfügung. Als Parameter werden dieselben Werte übergeben wie den `requestValue` und `synchronize` Methoden der Schnittstelle der Datenkonsistenzschicht, der Klasse `RM_IDataConsistencyLayer`. Entsprechend wird in jedem Synchronisationspunkt die Methode `check` aufgerufen, die das Verhalten des Protokolls in einem Synchronisationspunkt implementiert.

Werden Nachrichten versendet, muss jede Nachricht mit einem Wert vom Typ `RM_messagetype_t` beginnen, in dem der Typ der Nachricht kodiert ist. Fehlt diese Information kann der empfangende Manager die Nachricht nicht dem korrekten Protokoll zuordnen. Wird eine Nachricht empfangen, wird von dem zugehörigen Protokoll die Methode `onMessage` aufgerufen. Diese erhält den Nachrichtencode und den Rest der Nachricht als Parameter und muss die Nachricht bearbeiten.

Damit zur Laufzeit ein Protokolltyp eindeutig identifiziert werden kann, wird jedem Protokoll eine eindeutige Nummer zugeordnet. Diese Identitätsnummer muss von der Methode `getType` zurückgeliefert werden.

RM_IHandlerSupport

Die Klasse `RM_IHandlerSupport` stellt ein paar Methoden bereit, die von einer Implementierung eines Konsistenzprotokolls benötigt werden. Die wichtigsten Methoden sind in Listing 5.4 aufgeführt.

Listing 5.4: Auszug aus RM_IHandlerSupport

```
1  virtual bool setMessageType (RM_MessageType code,
2                       RM_IDataHandler *handler,
3                       bool sync = true)=0;
4
5  virtual void sendMessage(std::stringstream *message,
6                       RM_IDataHandler *source,
7                       void *param) = 0;
```

Damit der Datenkonsistenzschicht bekannt ist, welches Protokoll welche Nachrichten bearbeitet, müssen die Protokolle die Nachrichtentypen, die sie be-

arbeiten, registrieren. Dies wird mit der Methode `setMessageType` getan, der der Nachrichtencode und ein Zeiger auf das Protokoll übergeben wird. Zusätzlich wird eine Information darüber benötigt, ob die Bearbeitung von Nachrichten diesen Typs asynchron oder nur an Synchronisationspunkten stattfinden soll.

Mit Hilfe der Methode `sendMessage` kann eine Nachricht versendet werden. Die Nachricht wird als `stringstream` übergeben, wobei am Anfang der Nachricht der Nachrichtencode stehen muss.

5.2.3 Die Protokolle

Die Datenkonsistenzschicht unterstützt eine beliebige Anzahl an Protokollen. Die Protokolle unterscheiden sich durch das Konsistenzmodell, das sie implementieren und der verwendeten Strategie. Es gibt dabei zwei grundlegende Strategien: Die Aktualisierungsstrategie und die Invalidierungsstrategie. Es wurden folgende Protokolle implementiert:

- Aktualisierungsprotokoll für die PRAM Konsistenz (siehe Kap. 4.3.1).
- Invalidierungsprotokoll für die PRAM Konsistenz (siehe Kap. 4.3.2).
- Aktualisierungsprotokoll für die Spezielle Schwache Konsistenz (siehe Kap. 4.3.3).
- Invalidierungsprotokoll für die Spezielle Schwache Konsistenz (siehe Kap. 4.3.4).
- Aktualisierungsprotokoll für die Verzögerte Schwache Konsistenz (siehe Kap. 4.3.5).
- Invalidierungsprotokoll für die Verzögerte Schwache Konsistenz (siehe Kap. 4.3.6).
- Invalidierungsprotokoll für die PRAM Konsistenz für die blockweise Bearbeitung von Dateien.
- Aktualisierungsprotokoll für die PRAM Konsistenz für die blockweise Bearbeitung von Dateien.
- Ein Protokoll, welches komplette Dateien überträgt.
- Ein Protokoll, welches Daten auf Anfrage überträgt.

Für die PRAM-Konsistenz, die SSK und sie VSK sind die Protokolle hybrid.

Das Aktualisierungsprotokoll für die PRAM Konsistenz

Das Aktualisierungsprotokoll für die PRAM Konsistenz verschickt bei jeder Schreiboperation sofort eine Nachricht mit dem neuen Wert. Es ist in der Klasse `RM_UpdatePram` implementiert. Eigentlich bräuchte ein reines Aktualisierungsprotokoll nur diesen einen Nachrichtentyp. Da es aber in Kombination mit einem Invalidierungsprotokoll benutzt werden können soll, muss es auch den Invalidierungsstatus verwalten und bei Leseoperationen evtl. den aktuellen Wert anfordern. Daher werden insgesamt 5 Nachrichtentypen verwendet:

- **Error:** Die Fehlernachricht wird versendet, wenn ein Fehler auftritt. Die Fehlernachrichten werden asynchron behandelt.
- **Update:** Die Aktualisierung wird bei einer Schreiboperation versendet und enthält den neu geschriebenen Wert.
- **Response:** Die Validierung wird als Antwort auf eine Validierungsanfrage gesendet.
- **Request:** Die Validierungsanfrage wird von der Leseoperation versendet, wenn der zu lesende Wert ungültig ist.

Diese Nachrichtentypen werden im Konstruktor registriert. Dafür wird der Befehl

```
setMessageType(RMMT_UpdatePram+1, this, false);
```

verwendet, der in diesem Beispiel die `Update`-Nachricht registriert. Die neu erzeugte Instanz soll die Nachrichten dieses Typs behandeln, also ist der zweite Parameter `this` und alle Nachrichten sollen sofort nach Empfang an die Protokollfunktion weitergegeben werden. Dies wird erreicht, indem die Synchronisation auf `false` gesetzt wird.

Wird eine Aktualisierung empfangen, wird nicht nur der neue Wert gesetzt, sondern auch der Status auf gültig gesetzt, so dass beim Umschalten auf ein Invalidierungsprotokoll der Invalidierungsstatus korrekt initialisiert ist, bzw. falls dieser Prozess ein Invalidierungsprotokoll verwendet, aber eine Aktualisierung von einem Aktualisierungsprotokoll erhält, der Status für das Invalidierungsprotokoll richtig gesetzt ist.

Das Invalidierungsprotokoll für die PRAM Konsistenz

Das Invalidierungsprotokoll unterscheidet sich von dem Aktualisierungsprotokoll dadurch, dass in der `onWrite` Methode keine Aktualisierung sondern eine Invalidierung verschickt wird, falls der Status nicht *exclusive* ist. Sie verwendet dieselben Nachrichtenliste wie das Aktualisierungsprotokoll, allerdings mit anderen Codes. Zusätzlich versendet sie noch Invalidierungsnachrichten. Das Invalidierungsprotokoll ist in der Klasse `RM_InvalidatePram` implementiert.

Das Aktualisierungsprotokoll für die Spezielle Schwache Konsistenz

Das Aktualisierungsprotokoll für die SSK verwendet ebenfalls `Error`-, `Update`-, `Response`- und `Request`-Nachrichten wie das PRAM-Protokoll. Allerdings haben diese Nachrichten einen anderen Code. Für die Aktualisierungs- und Invalidierungsnachrichten wird bei der Registrierung der dritte Parameter von `setMessageType` auf `true` gesetzt, da Invalidierungen und Aktualisierungen nur in Synchronisationspunkten angewendet werden sollen. Dadurch werden die Nachrichten vom `RM_ConsistencyManager` bis zum nächsten Synchronisationspunkt gepuffert und erst dann von dem Protokoll bearbeitet. Die Fehler-, Validierungsanfrage- und Validierungsnachricht sind weiterhin asynchron.

Es gibt zwei zusätzliche Nachrichtentypen zur Umsetzung der sequentiellen Ordnung der Synchronisationspunkte:

- `RMMT_SyncRequest` wird versendet, wenn ein Prozess einen Synchronisationspunkt ausführen möchte, aber das Synchronisationsobjekt nicht besitzt.

- `RMMT_SyncGrant` wird versendet, um das Synchronisationsobjekt an den anderen Prozess zu übergeben.

Das Synchronisationsobjekt ist eine statische Variable der Protokollklasse, von der auch das Invalidierungsprotokoll abgeleitet ist, so dass sich beide Protokolle der SSK dasselbe Synchronisationsobjekt teilen.

Implementiert ist das Protokoll in der Klasse `RM_UpdateWeak`.

Das Invalidierungsprotokoll für die Spezielle Schwache Konsistenz

Auch das Invalidierungsprotokoll verwendet wieder eine `Error`-, `Invalidate`-, `Request`- und `Response`-Nachricht. Zur Synchronisation verwendet es dieselben Nachrichtencodes wie das Aktualisierungsprotokoll. Das Protokoll, dass zuerst die `RMMT_SyncRequest` und `RMMT_SyncGrant`-Nachrichten registriert, übernimmt die Verwaltung des Synchronisationsobjekts. Außerdem wird noch eine `WriteNotice` -Nachricht registriert.

Das Aktualisierungsprotokoll für die Verzögerte Schwache Konsistenz

Das hybride Aktualisierungsprotokoll für die VSK ist in Kap. 4.3.7 beschrieben. Es ist in der Klasse `RM_UpdateDelayed` implementiert. Es verwendet folgende Nachrichten:

- **Error:** Die Fehlernachricht.

- **Update:** Die Aktualisierung wird in einem Synchronisationspunkt verschickt, wenn in der vergangenen Epoche eine Schreiboperation stattgefunden hat.

- **Request:** Die Validierungsanfrage wird verschickt, wenn ein invalidiertes Objekt gelesen wurde.

- **Response:** Beantwortet eine `Request`-Nachricht indem es den aktuellen Wert zurücksendet.

- **SyncEnd:** Sie wird nach Beendigung eines Synchronisationspunkts verschickt.

- **WriteNotice:** Sie wird verschickt, wenn der Job ein ungültiges Objekt überschreibt.

- **ApprovalWirte:** Sie bestätigt den Erhalt einer `WriteNotice`-Nachricht.

Die `Update`-Nachricht wird vom `RM_ConsistencyManager` bis zum nächsten Synchronisationspunkt gepuffert, während die `Response`-Nachricht sofort bearbeitet wird.

Das Invalidierungsprotokoll für die Verzögerte Schwache Konsistenz

Das Invalidierungsprotokoll ist in der Klasse `RM_InvalidateDelayed` implementiert, die von `RM_UpdateDelayed` abgeleitet ist. Für die `SyncEnd`-Nachricht wird derselbe Nachrichtencode wie bei dem Aktualisierungsprotokoll verwendet. Das Protokoll, dass zuerst registriert wird, übernimmt also den Versand und die Bearbeitung der `SyncEnd`-Nachricht.

Blockweise Konsistenzprotokolle für Dateien

Die blockweisen Konsistenzprotokolle sind für sehr große Datenobjekte gedacht, die als ein einzelnes Datenobjekt registriert werden. Das können z.B. große Dateien sein, wobei die ganze Datei als einzelnes Datenobjekt registriert wurde. Hierbei soll vermieden werden, dass bei einer Änderung eines kleinen Teils des Objekts jedesmal das gesamte Objekt übertragen werden muss. Das bedeutet, dass das Protokoll das Datenobjekt in Blöcke aufteilen und intern verwalten muss. Außerdem soll auch bei der Initialisierung nicht die ganze Datei komplett übertragen werden.

Bei den blockweisen Protokollen besitzen beide Prozesse eine lokale Kopie der Datei und einen Index aus gleich großen Blöcken, bei denen jeweils ein Bit angibt, ob der lokale Block gültig ist. Zu Beginn hat der Job eine vollständig gültige Kopie und das Steuerungswerkzeug eine vollständig ungültige Kopie. Wird ein Block gelesen, der ungültig ist, wird dieser Block von dem anderen Prozess angefordert.

Bei den Schreiboperationen unterscheiden sich die beiden Protokolle. Das Aktualisierungsprotokoll versendet die neu geschriebenen Blöcke sofort, das Invalidierungsprotokoll invalidiert die geschriebenen Blöcke.

Der Index ist in der Klasse `RM_BlockIndex` implementiert, die Protokolle in den Klassen `RM_BlockUpdate` und `RM_BlockInvalidate`.

Zuerst wurde versucht, statt spezielle Protokolle für die blockweise Behandlung von großen Datenobjekten zu erstellen, automatisch jeden Block als eigenes Objekt zu registrieren und dann die nicht-blockweise Protokolle zu verwenden. Dann muss allerdings die Verwaltung der Blöcke in die Serialisierungsmethoden ausgelagert werden. Dabei treten eine Reihe von Schwierigkeiten auf:

- Jeder Block würde versuchen dieselbe Datei erneut zu öffnen. Insbesondere wenn die Serialisierung der Dateien aufwendig ist (z.B. bei ROOT-Dateien in Athena) sollte dies vermieden werden.
- Bei den Standardprotokollen ist es nicht ohne weiteres möglich auf eine Anfrage mehrere Antwort-Nachrichten zu verschicken. Außerdem sollten die Protokolle nicht die Puffer mit mehreren hundert Megabytes Daten füllen, sondern möglichst den nächsten Block erst abschicken, wenn der vorhergehende Block vollständig übertragen wurde. Dies kann ohne eine Änderung des Protokolls nicht bewerkstelligt werden.
- Bei der Verwendung von Blöcken, müssen zusätzliche Daten ausgetauscht werden. Wenn dieser Datenaustausch von den Serialisierungsmethoden durchgeführt werden soll, muss diese Kommunikation im Datenteil von Nachrichten eines bestehenden Protokolls kodiert werden. Demgegenüber bietet die Schnittstelle der Protokolle wesentlich komfortablere Möglichkeiten zur Kommunikation.

- Eine Speicherzugriffsoperation kann mehrere Blöcke umfassen. Wann ist für jeden dieser Blöcke die Antwort angekommen? Dies lässt sich leichter verfolgen, wenn die Blöcke nicht völlig unabhängig von einander verwaltet werden.

5.3 Automatisierte Optimierung

In Kap.4.5 wurden verschiedene Möglichkeiten zur Optimierung untersucht. Von den untersuchten Ansätzen wurde ein Algorithmus zur automatisierten Protokollwahl in der Klasse `RM_ProtocolChooser` implementiert. Diese Klasse implementiert die Schnittestelle `RM_IDataConsistencyLayer` und wird als zusätzliche Schicht auf der Datenkonsistenzschicht aufgesetzt.

Die Optimierung basiert auf dem Algorithmus in Listing 4.8 aus Kap. 4.5.1. Es soll auf keinen Fall die Bandbreite überschritten werden. Daher wird für die Berechnung der Kostenfunktion ein Exponent $c_e = \infty$ gewählt. Damit spielt der Wert für die Konstanten c_D und c_t keine Rolle mehr, solange sie größer als 0 sind. Ferner wird $c_B = 1$ gesetzt, da die verfügbare Bandbreite voll ausgenutzt werden darf. Damit erhält man folgende vereinfachte Kostenfunktion:

$$K = \begin{cases} \infty & \text{falls} \quad D - B > 0 \\ 0 & \text{sonst} \end{cases} \tag{5.1}$$

Die verfügbare Netzwerkbandbreite B ist in der Variablen `m_Bandwidth` gespeichert. Ebenso ist ein fester Wert für die Größe der Invalidierungsnachricht und der Anfrage in `m_ReqInv` gespeichert. Beide Werte werden im Konstruktor gesetzt. Die Antwortzeit wird regelmäßig gemessen. Dafür wurde ein eigenes Protokoll `RM_ProtocolHandler` implementiert, das in Synchronisationspunkten eine Nachricht an den anderen Prozess sendet und die Zeit bis zum Erhalt der Antwort misst.

Für die Messung der Round-Trip-Zeit anhand von Nachrichten die sowieso verschickt werden, führt zu starken Schwankungen. Zum einen werden manche Nachrichten bis zu einem Synchronisationspunkt gepuffert und erst dort beantwortet. Diese Pufferung dauert unterschiedlich lang. Es könnte daher nur mit `Requests` bei Leseoperationen von invalidierten Objekten die Round-Trip-Zeit bestimmt werden oder bei `ApprovalRequests`. D.h. bei Aktualisierungsprotokollen müssen sowieso spezielle Nachrichten verschickt werden, um die Round-Trip-Zeit zu messen. Zum anderen müssten alle Protokolle umgerüstet werden. Daher wurde ein eigenes Protokoll, das mit speziellen Nachrichten die Round-Trip-Zeit misst, bevorzugt.

Die Lese- und Schreibzugriffe werden in jedem Prozess mitgezählt, wenn die `synchronize` bzw. die `requestValue` Methode der `RM_ProtocolChooser` Instanz aufgerufen wird. Um die Anfragen des anderen Protokolls mitzubekommen, wird eine doppelte Strategie verfolgt. Falls der andere Prozess ein Invalidierungsprotokoll verwendet, wird die `onRequest` Methode aufgerufen, die den Zähler inkrementiert. Ansonsten verschickt jeder Prozess die aktuelle Anzahl der Leseoperationen an den anderen Prozess, wenn sich die Anzahl um den Faktor 1.5 erhöht hat. Der andere Prozess berechnet daraus die durchschnittliche Anzahl von Leseoperation pro Zeiteinheit und schätzt auf dieser Basis die Anzahl

der Anfragen ab. Somit wird nicht bei jeder Leseoperation eine Nachricht gesendet, trotzdem wird die Anzahl der Leseoperationen innerhalb einer gewissen Fehlerrate aktuell gehalten.

5.4 Datenzugriff

Die Datenzugriffsschicht soll den darunter liegenden Schichten eine standardisierte Schnittstelle bieten, um auf Anwendungsdaten zuzugreifen. Außerdem soll sie Methoden bereitstellen, die eine einfache Integration in die Anwendung erlauben. Dabei wird jeweils eine Schnittstelle für kleine Datenobjekte, die als Ganzes behandelt werden und eine Schnittstelle für große Datenobjekte (z.B. Dateien), die blockweise behandelt werden (siehe 4.4.2), bereit gestellt.

5.4.1 Die Schnittstelle

Die Datenzugriffschicht hat eine fest definierte Schnittstelle zu den darunterliegenden Auswertung- und Datenkonsistenzschichten, die so auf einheitliche Weise auf alle Daten zugreifen. Dadurch müssen dieses Schichten nicht geändert werden, wenn neue Datentypen verwendet werden. Um einen neuen Datentyp für das Steering verfügbar zu machen, müssen entsprechende Serialisierungsmethoden bereitgestellt werden.

Die Schnittstelle besteht aus jeweils einer Serialisierungsmethode und einer Deserialisierungsmethode für beide unterstützte Granularitätsstufen. Jede dieser vier Methoden ist in einer eigenen abstrakten Klasse definiert (siehe Listing 5.5).

Listing 5.5: Schnittstelle der Datenzugriffsschicht

```
1  class RM_ISerializeMethod {
2  public:
3    virtual int serialize(std::ostream *os,
4                          std::istream *param)=0;
5  };
6
7  class RM_IDeserializeMethod {
8  public:
9    virtual void deserialize(std::istream *is)=0;
10 };
11
12 class RM_ISerializeFiles {
13 public:
14   virtual std::istream *serialize(std::istream *param)=0;
15 };
16
17 class RM_IDeserializeFiles {
18 public:
19   virtual void deserialize(std::istream *is,
20                            RM_filesize_t offset,
21                            RM_filesize_t size)=0;
22 };
```

Jede Methode wurde bewusst in einer eigenen Klasse definiert. Es gibt Objekte, die nur gelesen (oder geschrieben) werden können und daher auch nur die `serialize` (bzw. `deserialize`) Methode brauchen. Oder es existieren für einen Datentyp unterschiedliche `serialize` Methoden. Es soll aber immer dieselbe `deserialize` Methode verwendet werden. So werden z.B. beim ATLAS Experiment (siehe Kap. 6) ROOT-Dateien im Job mit einer anderen Methode geöffnet als in der Visualisierung.

Durch die Verwendung eigener Serialisierungsmethoden können auch komplexe Vorbereitungen oder Vorberechnungen gekapselt werden. Die Daten werden mit der `RM_ISerializeMethod::serialize()` Methode in einen eigenen `std::ostream` geschrieben, der bereits einen Nachrichtenkopf enthalten kann. Dieses Vorgehen hat den Vorteil, dass anschließend die Daten nicht mehr kopiert werden müssen, und daher Rechenzeit gespart wird. Als zweiten Parameter erhält die `serialize`-Methode einen `std::istream`, durch den Parameter übergeben werden können, die zu einer speziellen Vorberechnung notwendig sind. Serialisierungsmethoden welche keine Parameter benötigen, können diese Eingabe ignorieren. Die Parameter werden deshalb in serialisierter Form übergeben, da sie möglicherweise zuvor über das Netzwerk von einem anderen Rechner übermittelt wurden. Zu beachten ist, dass bei aktiven Zugriffen in der Regel keine Parameterübergabe möglich ist.

Das Gegenstück zu der `serialize()` Methode ist die `deserialize()` Methode der Klasse `RM_IDeserializeMethod`. Sie bekommt einen `std::istream` übergeben, deren Lesezeiger auf dem ersten Byte steht, dass von der Serialisierungsmethode geschrieben wurde. Die Aufgabe der Deserialisierungsmethode ist es, die Daten wieder in die lokale Kopie zu schreiben.

Das Interface für große Datenobjekte, die von der Datenkonsistenzschicht weiter aufgeteilt werden können, ist in den Klassen `RM_ISerializeFiles` und `RM_IDeserializeFiles` definiert. In diesem Fall wird das Datenobjekt nicht in einen Stream kopiert, sondern ein `std::istream` zurückgegeben, der die Daten enthält. Das Protokoll kann dann den Teil der Daten, den es übertragen muss, selektieren. Die Aufgabe der `serialize()`-Methode kann zum Beispiel darin bestehen, eine Datei als `std::fstream` zu öffnen und den Dateistrom zurückzugeben. Die Daten werden dann vom Konsistenzprotokoll stückweise in Nachrichten kopiert.

Die `RM_IDeserializeFile::deserialize()`-Methode hat neben den serialisierten Datenfragment zwei zusätzliche Parameter, die spezifizieren, welchen Teil der Daten sie enthalten. Es hat sich in der Praxis gezeigt, dass die Verwendung der allgemeinen `istream::seekp`-Methode nicht immer dazu führt, dass Datenfragmente an die korrekte Stelle geschrieben werden. Oft ist z.B. nur das Anhängen am Ende des Streams möglich. Daher wurde das Positionieren auch der `deserialize` Methode überlassen, die spezifischer auf die jeweilige Speichermethode eingehen kann. Dafür muss der `deserialize` Methode aber die Position und Länge des übertragenen Datenstücks übergeben werden.

5.4.2 Grundlegende Zugriffsunterstützung

Die grundlegenden Funktionen zur Zugriffsunterstützung sind in der Klasse `RM_RegisterBasicTypes` enthalten. Ein Ausschnitt dieser Klasse ist in Listing 5.6 dargestellt.

Listing 5.6: Die Klasse RM_RegisterBasicTypes

```
class RM_RegisterBasicTypes
      : public RM_DataConsistencyBase
{
public:
  virtual bool registerByte  (const std::string name,
                        char *value,
                        bool writeable = true,
                        bool readable  = true );

  virtual bool registerInt   (const std::string name,
                        __int32_t  *value,
                        bool writeable = true,
                        bool readable  = true );

  virtual bool registerFile(const std::string name,
                        RM_ISerializeFiles *Serialize,
                        RM_IDeserializeFiles *Deserialize,
                        RM_IDataHandler *handler = NULL);
  ...

  virtual bool registerFile(const std::string fileName);

  virtual void setDefaultMode(int mode);

  virtual int getDefaultMode();

  virtual void setMode(std::string name, int mode);

  virtual int getMode(std::string name);
};
```

Sie stellt Methoden bereit, die Datenobjekte von Basistypen registrieren und dafür Standardserializer erstellt. Derartige Methoden existieren für Bytes, 32-Bit und 64-Bit Ganzzahlen, Fließkommazahlen einfacher und doppelter Genauigkeit, Zeichenketten und Dateien. Ferner kann in dieser Klasse eine Abtastung von passiven Datenobjekten durchgeführt werden. Es gibt drei Abtastmodi:

- **Keine Abtastung:** Das Zugriffsverhalten wird von dieser Klasse nicht beeinflusst. Es werden keine Aufrufe von Zugriffsmethoden des Steering-Systems generiert.

- **Insensitive Abtastung:** Unabhängig davon, ob tatsächlich eine Schreiboperation stattgefunden hat, wird in jedem Synchronisationspunkt eine Schreiboperation ausgelöst.

- **Wert-sensitive Abtastung:** In jedem Synchronisationspunkt wird die serialisierte Form der Daten mit der serialisierten Form des Wertes am letzten Synchronisationspunkt verglichen. Falls sich die Daten geändert haben, wird eine Schreiboperation ausgelöst.

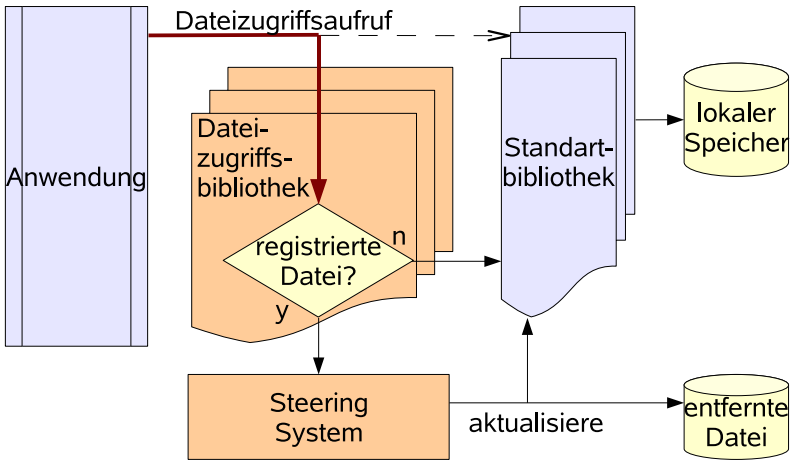


Abbildung 5.4: Funktionsweise der Umlenkung der Dateizugriffe

Das Standardabtastmodus ist “Keine Abtastung”, dass kann aber mit der Methode `setDefaultMode` geändert werden. Der Zugriff auf alle neu registrierten Datenobjekte erfolgt mit dem eingestellten Standardabtastmodus. Außerdem kann für spezielle Datenobjekte der Abtastmodus explizit bestimmt werden. Im allgemeinen wird der Zugriff auf die meisten Daten nach demselben Modus erfolgen. Es kann daher an zentraler Stelle mit einem Aufruf für alle Objekte angepasst werden.

5.4.3 Umlenkung von Dateizugriffen

Viele Anwendungen, die im Grid laufen, legen Ergebnisse oder Zwischenergebnisse in Dateien ab, die anschließend von Visualisierungswerkzeugen angezeigt werden können. Daten in Dateien sind bereits serialisiert, so dass die Notwendigkeit entfällt, dem Steering-System Typinformationen zur Verfügung zu stellen. Grundsätzlich kann diese Zugriffsmethode daher anwendungsübergreifend eingesetzt werden und bedarf keiner oder nur geringer Anpassung. Sie kann teilweise sogar eingesetzt werden, wenn nur der ausführbare Binärcode vorliegt. Dadurch ist sie auch für lange gewachsene Programme praktikabel. Ein Beispiel für die Verwendung ist in Kap. 6 beschrieben. Die Funktionsweise der Umlenkung der Dateizugriffe ist in Abb. 5.4 dargestellt.

Um die Dateizugriffe umzulenken, werden die Dateizugriffsfunktionen der Standardbibliothek von eigenen Implementierungen überschrieben, die zwar den gleichen Funktionskopf besitzen, aber den Aufruf an das Steering-System umleiten. Die Bibliothek, die diese Funktionen enthält, muss beim Start der Anwendung mittels `LD_PRELOAD` vor den Standardbibliotheken geladen werden. Hierbei macht man sich das Verhalten des dynamischen Linkers zunutze, der Aufrufe zu der zuerst geladenen passenden Funktion auflöst. Um selbst auf die Dateizugriffsfunktionen zugreifen zu können, wurden Wrapperfunktionen geschrieben, die einen Zeiger auf die nächste Funktion mit dem entsprechenden Namen anfordern und diese Funktion dann aufrufen (siehe Listing 5.7). Im Moment werden folgende Funktionen abgefangen: `open`, `open64`, `write`, `read`, `close`, `lseek`,

`fstat`und `fstat64`.

Listing 5.7: Wrapperfunktion um bei überladenen Dateizugriffsfunktionen auf die Standardfunktionen zugreifen zu können

```
1 int _open(const char *pathname, int flags, mode_t mode)
2 {
3   static int (*ptr)(const char *, int, mode_t) = 0;
4   if (ptr == 0)
5   {
6     ptr = (int(*)(const char *, int, mode_t)) dlsym(
          RTLD_NEXT, "open");
7   }
8   return (*ptr)(pathname, flags, mode);
9 }
```

Dateien, für die die Zugriffe an das Steering-System umgelenkt werden, müssen vorher bei der Zugriffsbibliothek registriert werden. Zugriffe auf nicht registrierte Dateien werden an die Standardbibliotheken weitergegeben. Dateien können registriert werden, indem die Funktion `addTracedFile()` aufgerufen wird, die als Parameter den Dateinamen und einen Zeiger auf eine Implementierung der Klasse `RM_IFile` enthält.

In der Klasse `RM_SyncFile` sind die umgelenkten Zugriffe auf eine Datei implementiert. Beim Öffnen der Datei wird eine lokale Kopie angelegt. Mit Hilfe eines blockweisen Konsistenzprotokolls der Datenkonsistenzschicht wird bei Schreibzugriffen und Lesezugriffen die Aktualisierung des Inhalts der Datei durchgeführt.

Um die Zugriffsbibliothek auch mit bestehenden binären Programmen verwenden zu können, die nicht neu kompiliert werden sollen, wurde eine weitere Option eingefügt, um Dateien über Umgebungsvariablen zu registrieren. In der Initialisierungsphase wird die Umgebungsvariable `RMOST _TARGET_JOB` ausgelesen. Enthält sie einen Jobnamen, erzeugt die Zugriffsbibliothek eine eigene Instanz der Datenkonsistenzschicht und verbindet sich zu dem angegebenen Job. Danach wird die Umgebungsvariable `RMOST _FILES` ausgelesen, welche durch Doppelpunkte separierte Dateinamen enthalten kann. Diese Dateien werden bei der Zugriffsbibliothek registriert.

5.5 Fazit

In diesem Kapitel wurde die Implementierung der Kommunikationsschicht, der Datenkonsistenzschicht, der automatischen Protokollwahl sowie der Zugriffsschicht vorgestellt.

Die Kommunikationsschicht baut einen interaktiven Kommunikationskanal zu einem Gridjob auf. Der verwendete Mechanismus adaptiert sich dynamisch an die Konfiguration der Gridsite. Die Sicherheit von Site und Job werden gewahrt. Die Kommunikation ist in den meisten Fällen möglich, ohne dass Sites ihre Sicherheitseinstellungen ändern müssen. Die meisten Sites im LCG [96] ermöglichen ausgehende Verbindungen. Mit Hilfe des Kommunikationskanals von RMOST ist es daher in den meisten Fällen möglich, eine interaktive Kommunikation zu dem Gridjob aufzubauen, ohne das die Sites zusätzliche Dienste installieren müssen.

Bei der Kommunikationsschicht lassen sich verschiedene Funktionen flexibel aufeinander schichten. So kann man eine reine TCP-Verbindung verwenden, eine einfache verschlüsselte Verbindung oder eben eine Gridverbindung. Diese Flexibilität war erwünscht, führte aber dazu, dass alle Schichten, die einen Header vor der Nachricht benötigen, die Daten einmal kopieren müssen. Dies betrifft die Nachrichtenschicht und die Sicherheitsschicht. Bei der Sicherheitsschicht verlangt die verwendete GSSAPI, dass die Daten in einer speziellen Datenstruktur übergeben werden, die durch die GSSAPI definiert wird. Mit einer enger verzahnten Schicht könnte sicherlich die Anzahl der Kopien verringert werden, was der Performance zugute käme, allerdings würde man dadurch auch die Flexibilität verringern.

Für den Namensdienst wurde R-GMA [30] verwendet, da es als Standardsystem im LCG [96] installiert ist. Anfragen an R-GMA benötigen aber eine bemerkbare Zeitspanne in der Größenordnung von 10 s. Die Verwendung eines anderen, schnelleren Namensdienstes könnte den Verbindungsaufbau deutlich beschleunigen. Für einen Einsatz in einem Produktionsystem müsste dieser Namensdienst entsprechend skalieren und überall verfügbar sein. Eine verwendbare Alternative gab es im LCG derzeit nicht.

Die Datenkonsistenzschicht bietet ein Framework mit verschiedenen Konsistenzprotokollen an. Mit Hilfe dieses Frameworks lassen sich neue Protokolle einfach und effektiv integrieren. Die Konsistenzschicht arbeitet mit serialisierten Daten. Sie ist daher unabhängig von der Art der Daten. Es wurden für kleine Datenobjekte jeweils Invalidierungs- und Aktualisierungsprotokolle für die PRAM-Konsistenz, SSK und VSK entwickelt.

Für die blockweise Behandlung großer Datenobjekte wurde zuerst versucht, dieselben Protokolle zu verwenden und die weitere Funktionalität in den Serialisierungsmethoden zu kapseln. Der Code für die Serialisierungsmethoden wurde allerdings sehr schnell unübersichtlich. Außerdem erwies sich die Schnittstelle der Serialisierungsmethoden, die nur aus dem Protokoll heraus aufgerufen werden, als zu eng. Letztendlich entwickelten sich verschiedene Nachrichtentypen, die in Anfrage- oder Aktualisierungsnachrichten kodiert wurden. Daran wurde klar, dass doch eigene Protokolle für die blockweise Behandlung der Daten nötig sind. Implementiert wurde allerdings nur die PRAM-Konsistenz.

Die Implementierung wurde für Steuerungswerkzeuge mit einem Steering-Prozess erstellt. Das Steuerungswerkzeug kann mehrere unabhängige Anwendungsprozesse gleichzeitig steuern. Dabei darf jedes Datenobjekt allerdings nur in einem Anwendungsprozess vorkommen (disjunkte Datenmengen der Anwendungsprozesse). Dies erlaubt die Untersuchung der Interaktion zwischen Job und Steuerungswerkzeug bei den vollständig neu entwickelten Konsistenzmodellen SSK und VSK. Bei beiden Modellen muss die Inter-Prozess Bedingung nicht berücksichtigt werden, d.h ein unterschiedlicher Fortschritt im Programmablauf braucht nicht beachtet werden.

Die automatische Protokollwahl wurde in einer eigenen Schicht implementiert, die auf die Datenkonsistenzschicht aufgesetzt werden kann. Durch ein eigenes Protokoll und die Schnittstelle der Datenkonsistenzschicht können so alle relevanten Aufrufe mitverfolgt werden. Die Parameter der Optimierung sind so gewählt, dass die verfügbare Bandbreite nie überschritten wird.

Die Datenzugriffsschicht ist eine weitere Schicht, die auf die Optimierung oder Datenkonsistenzschicht aufgesetzt werden kann. Für Standarddatentypen werden Serialisierungmethoden bereit gestellt. Sie bietet Abtastfunktionalität

für passive Datenobjekte.

Für Dateien wurde eine Bibliothek entwickelt, mit der Dateizugriffe abgefangen werden können. Dies erlaubt die transparente Umlenkung der Dateizugriffe an das Steering-System.

Die Gridverbindung bietet eine zuverlässige, sichere, effiziente Kommunikation mit dem Gridjob. Die Datenkonsistenzschicht bietet eine flexible, funktionsfähige Implementierung, an der die Eigenschaften der Konsistenzprotokolle und der Kommunikationskanals praktisch evaluiert werden können. Die Optimierung ist noch sehr einfach. Die Implementierung der Basisfunktionen der Datenzugriffsschicht ist funktionsfähig, enthält aber keine anspruchsvollen Teile. Zusätzlich existiert die Dateizugriffsbibliothek, die Dateizugriffe umlenkt. Hier wurde ein wirkungsvolles Werkzeug geschaffen. Für den universellen Einsatz müsste sie allerdings die kompletten POSIX-Schnittstelle unterstützen und sich nicht auf eine Untermenge der POSIX-Funktionen beschränken.

Kapitel 6

Anwendung beim ATLAS-Experiment

RMOST wurde in die Softwareumgebung des ATLAS-Experiments [1, 172, 173] integriert und unterstützt das Online-Steering von Gridjobs des ATLAS-Experiments. Die Software des ATLAS-Experiments ist sehr umfangreich und wird beständig weiter entwickelt. Eine vollständige Quellcode-Instrumentierung der Software ist für einen Einzelnen zu aufwendig. Daher musste die Integration des Steerings in die ATLAS Softwareumgebung ohne Quellcodeänderungen und Neukompilierung der bestehenden Programme ausgeführt werden.

In diesem Kapitel werden die Inhalte und Ziele des ATLAS-Experiments in Abschnitt 6.1 vorgestellt. Anschließend wird in Kap. 6.2 die verwendete Software im ATLAS-Experiment und die Integration von RMOST in die Experimentsoftware (siehe Kap. 6.3) der Gridjobs und die Visualisierungsumgebung (siehe Kap. 6.4) beschrieben.

6.1 Das ATLAS-Experiment

Mit dem Large Hadron Collider (LHC) [25] am CERN in Genf ist es möglich Protonen auf 7 TeV zu beschleunigen, so dass bei der Proton-Proton-Kollision eine Schwerpunktsenergie von 14 TeV erreicht werden kann. Außerdem besteht die Möglichkeit Bleikerne zu beschleunigen. Der LHC ist in dem 27 km langen Tunnel installiert, in dem vorher der inzwischen stillgelegte Large Electron Positron collider untergebracht war.

Der LHC wird für verschiedene Experimente genutzt. Mit den Experimenten sollen eine Vielzahl von Forschungsaufgaben erfüllt werden. Ein Ziel ist die Suche nach neuen Elementarteilchen. Insbesondere hofft man, das Higgs-Boson zu entdecken, das für die Masse von Teilchen verantwortlich ist.

Das ATLAS-Experiment ist eines der vier großen LHC-Experimente. Der 44 m lange und 7000 t schwere Detektor (siehe Abb. 6.1) des ATLAS-Experiments besteht aus mehreren Subsystemen, die es unter anderem erlauben, den Impuls der Teilchen und deren Energie zu bestimmen. Die Ziele des ATLAS-Experiments umfassen unter anderem die Suche nach einem (oder möglicherweise mehreren) Higgs-Teilchen, die Suche nach supersymmetrischen Teilchen und Untersuchungen von Zerfällen, in denen das Bottom-Quark vorkommt.

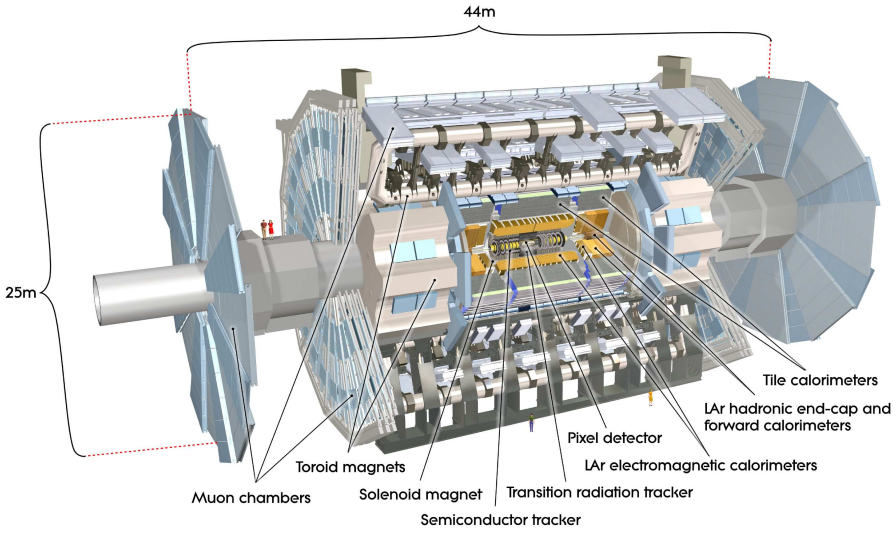


Abbildung 6.1: Der ATLAS Detektor (Quelle: [1])

Die gemessenen Rohdaten werden mit Hilfe von Computern ausgewertet. Allein im ATLAS-Experiment wird eine Datenmenge von 3,2 PB pro Jahr erwartet. Diese Daten sollen Physikern weltweit zur Verfügung stehen. Daher kann Speicherung und Auswertung nicht mehr lokal stattfinden, sondern es bedarf der Nutzung verteilter Ressourcen. Aus diesem Grund wurde das LHC Computing Grid (LCG) [96] eingerichtet, das zur Auswertung und Speicherung der Experimentdaten der LHC-Experimente dient. Es wird außerdem zur Simulation von Ereignissen in den Detektoren verwendet.

6.2 Die Softwareumgebung des ATLAS-Experiments

Die Softwareumgebung des ATLAS-Experiments besteht aus

- dem Athena Framework [41], das für die Auswertung der Daten genutzt wird und
- für die Visualisierung der Ergebnisse wird das ROOT Toolkit [28, 27] verwendet, das in der Hochenergiephysik verbreitet ist.

Im Folgenden werden kurz die für diese Arbeit wichtigen Teile von Athena und ROOT beschrieben.

6.2.1 Das Athena Framework

Das Athena Framework [41] wird im ATLAS-Experiment für die Auswertung benutzt. Dazu gehört, dass Simulationen durchgeführt werden, Kollisionen aus den einzelnen Messdaten rekonstruiert werden und Analysen durchgeführt werden. Jedes Auftreten einer Kollision zweier Teilchenpakete ist ein sog. Ereignis

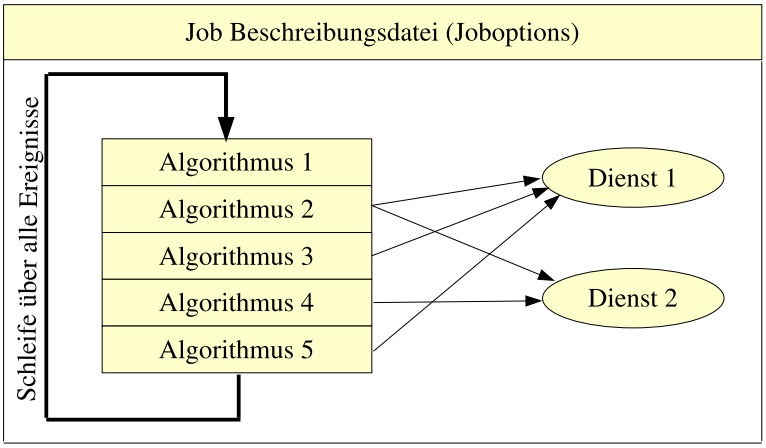


Abbildung 6.2: Aufbau eines Athena-Jobs

und kann unabhängig von anderen Ereignissen ausgewertet werden. Die Ergebnisse der Berechnung bestehen in der Regel aus Statistiken über Eigenschaften von Teilchen aus einer Vielzahl von Ereignissen und werden in sogenannten ROOT-Dateien gespeichert.

Athena hat einen modularen Aufbau und stellt dem Benutzer verschiedene Komponenten zur Verfügung, aus denen sich der Benutzer seine Berechnung zusammenstellen kann. Die Komposition und Konfiguration der einzelnen Komponenten geschieht in sog. Jobbeschreibungsdateien, die auch Joboptions genannt werden.

Der Benutzer kann auch eigene Komponenten erstellen und in seinen Berechnungen verwenden. Für die Implementierung neuer Komponenten muss die Schnittstelle des Komponententyps implementiert und in einer dynamisch ladbaren Bibliothek bereit gestellt werden. Über die Joboptions kann Athena angewiesen werden, die entsprechende Bibliothek zu laden und die neuen Komponenten zu instanzieren.

Es gibt unterschiedliche Arten von Athena-Komponenten. Die zwei wichtigsten Komponententypen für diese Arbeit sind die Athena-Algorithmen und Athena-Dienste (siehe 6.2).

Der Kern eines Athena-Jobs bildet eine Sequenz von Athena-Algorithmen, die nacheinander für jedes Ereignis ausgeführt werden. Jeder Algorithmus führt dabei eine Berechnung durch, auf deren Ergebnis der nächste Algorithmus aufbauen kann.

Athena-Dienste stellen bestimmte Funktionen für andere Komponenten bereit, auf den diese zugreifen können. Zum Beispiel gibt es Dienste zum persistenten Speichern von Daten, Dienste um die Ergebnisse einer Berechnung einem Histogramm hinzuzufügen, einen Dienst zur Ausgabe von Statusinformationen, Dienste zum Auffinden anderer Dienste und Weitere.

In Tab. 6.1 sind z.B. die Algorithmensequenz und die verwendeten Athena-Dienste eines sog. Monte-Carlo-Simulationsjob dargestellt. In diesem Job werden mit Hilfe eines Zufallsgenerators Daten einer simulierten Kollision erzeugt. Das Verhalten der entstandenen Teilchen wird simuliert, bis sie das Detektormaterial erreichen. Schließlich wird das simulierte Ereignis persistent gespeichert.

Algorithmen	Dienste
PythiaB	StoreGateSvc
BSignalFilter	DetDescrCnvSnc
CBNT_Truth	StatusCodeSvc
All	MessageSvc
History	DetectorStore
VisibleCharged	HistoryStore
Boson	PortPrepSvc
Lepton	AuditorSvc
Electron	AtRandomGenSvc
HeavyQuark	
Photon	
TauDaughters	
Tau	
AllDaughters	
VisibleChargedAncestors	
BosonAncestors	
AllLeptonAncestors	
HeavyQuarkAncestors	
PhotonAncestors	
AANTupleStream	

Tabelle 6.1: Sequenz der Athena-Algorithmen und die verwendeten Athena-Dienste eines typischen Monte-Carlo-Simulationsjobs mit Athena

In diesem Beispiel werden im Algorithmus `PythiaB` die initialen Daten erzeugt. Als Zufallsgenerator dient der Athena-Dienst `AtRandomGenSvc`. Die erzeugten Daten werden im `StoreGateSvc` abgelegt, aus dem nachfolgende Algorithmen die Daten auslesen können. Anschließend können die Ereignisdaten nach gewünschten Merkmalen gefiltert werden (`BSignalFilter`). Andere Algorithmen wie `CBNT_Truth` oder `AANTupleStream` dienen der persistenten Speicherung.

Neben den genannten Diensten werden noch weitere Athena-Dienste von mehreren Algorithmen verwendet. Der `MessageSvc` stellt z.B. eine Schnittstelle für Logging- und Debugging-Ausgaben bereit, oder der `DetDescrCnvSvc` liefert die detektorspezifischen Eigenschaften wie z.B. die Detektorgeometrie.

Wenn der Job beendet ist, kann der Benutzer mit weiteren Jobs simulieren, welche Signale der Detektor aus diesem Ereignis erzeugen würde und anschließend versuchen, aus diesen Signalen den Impuls und die Energie der Teilchen zu rekonstruieren. Dadurch kann er die wahren Zerfälle mit den rekonstruierten Zerfällen vergleichen und somit die Aussagekraft von rekonstruierten Ergebnissen überprüfen.

Die Jobbeschreibungsdatei ist ein Python-Skript, das weitere Skripte einbinden kann. Für häufig verwendete Standardketten von Algorithmen und Diensten (z.B. für die Rekonstruktion eines Ereignisses) existieren in der Regel Skripte, die an der entsprechenden Stelle eingebunden werden können. Folgender Auszug (Listing 6.1) aus den Joboptions weist Athena an, einen Algorithmus zu erstellen und am Ende der Algorithmenliste einzufügen. Außerdem werden ein paar Parameter des Algorithmus gesetzt:

Listing 6.1: Auszug aus einer Athena Jobbeschreibungsdatei

```
1 theApp.TopAlg += [ "AANTupleStream" ]
2 AANTupleStream = Algorithm( "AANTupleStream" )
3 AANTupleStream.OutputName = 'pythiaB.root'
```

In diesem Beispiel wird der Algorthmus `AANTupleStream` zur Algortihmensequenz hinzugefügt, der die simulierten Daten persistent speichert. Außerdem wird festgelegt, dass die Ausgabedatei `pythiaB.root` heißen soll.

6.2.2 Das ROOT Toolkit

Das ROOT Toolkit [27, 28], wird zur Visualisierung von Ereignisgrößen verwendet. ROOT besitzt einen C++ Interpreter mit einer Kommandozeile, von der aus auf eine Bibliothek von Klassen und Funktionen zugegriffen werden kann, die der Visualisierung und Weiterverwertung von berechneten Daten dienen. Die Daten können in ROOT nicht nur visualisiert werden, sondern es können auch beliebige weitere Berechnungen mit den Daten durchgeführt werden. Von der Kommandozeile können auch C++ Skripte (sog. Makros) ausgeführt werden oder neue Bibliotheken mit weiteren Klassen geladen werden.

ROOT verfügt über eine Schnittstelle, die es erlaubt, neue Klassen zu erstellen, die anschließend im ROOT Interpreter benutzt werden können.

6.3 Integration von RMOST in die Experimentsoftware

Bei der Auswertung des ATLAS-Experiments werden typischerweise viele sequentielle Jobs submittiert und die berechneten Statistiken anschließend addiert. Für das Aufaddieren der Statistiken ist es unerheblich, wieviele Ereignisse jeder einzelne Job berechnet hat. Daher braucht beim Steering von Athena-Jobs nur die Intra-Prozess Bedingung erfüllt werden.

Die Experimentsoftware Athena [41, 36] besteht aus einem, über mehrere Jahre entwickelten, Code. Die Software des ATLAS-Experimentes ist sehr umfangreich und wird beständig weiter entwickelt. Eine vollständige Quellcode-Instrumentierung der Software ist für einen Einzelnen sehr aufwendig. Allerdings können neue Module bereitgestellt werden, die sich über die Joboptions in den Job einfügen lassen.

Um das Steering von Athena-Gridjobs zu ermöglichen, wurden einige neue Athena-Komponenten entwickelt. Die Bibliothek, welche diese Komponenten enthält, kann mit dem Job mitgeschickt werden oder von einem Start-Skript auf das ausführende WN heruntergeladen werden. Der Benutzer kann grundlegende Steeringmöglichkeiten aktivieren, indem er seine Jobbeschreibungsdatei ändert und die entsprechenden Komponenten in seinen Job integriert.

Die entwickelten Komponenten `RM_Spy`, `RM_SteeringSvc`, `RM_Checker` und `RM_EvaluatorBase` werden in den kommenden Abschnitten detailierter vorgestellt.

RM_Spy

`RM_Spy` ist ein Athena-Algorithmus, der grundlegende Möglichkeiten bietet, um

Zwischenergebnisse zu visualisieren und den Job zu steuern. Im Einzelnen umfasst der Funktionsumfang von `RM_Spy`:

- Visualisierung von Zwischenergebnissen, die in ROOT-Dateien geschrieben werden. Um Athena dazu zu bringen, alle Puffer zu schreiben, wird ein Kindprozess abgezweigt, der sich anschließend beendet. Beim Beenden des Jobs im Kindprozess werden alle offenen Dateien geschrieben und geschlossen. Der Elternprozess wartet bis der Kindprozess beendet wurde und wird danach weiter ausgeführt.

- Anzeige der Anzahl der bereits berechneten Ereignisse.

- Beenden des Jobs.

- Anhalten des Jobs.

- Ausführung des Jobs bis zum nächsten Synchronisationspunkt. Am nächsten Synchronisationspunkt wird der Job angehalten. Synchronisationspunkte werden ausgeführt, wenn die `check` Methode das `RM_SteeringSvc` aufgerufen wird. Dies geschieht in `RM_Spy` und `RM_Checker`.

- Fortsetzen der Ausführung eines angehaltenen Jobs.

- Neustart des Jobs, ohne dass der Job dabei neu submittiert werden muss. Der Neustart wird mittels `exec` auf Athena mit den entsprechenden Argumenten ausgeführt.

- Die Jobkonfiguration in den Joboptions kann geändert werden. Die Änderungen werden nach einem anschließenden Neustart des Jobs wirksam, wobei der Job nicht neu submittiert wird.

- Es besteht die Option, bei Jobstart und Jobende eine Nachricht via Email und/oder über das Steuerungswerkzeug zu erhalten.

- Optional kann bei vorzeitigem Jobabbruch das Beenden um eine bestimmte Zeit verzögert werden. Während der Verzögerung kann mit dem Job interagiert werden, können eventuelle Problemursachen untersucht werden oder kann der Job neu gestartet werden. In diesem Fall erhält der Benutzer eine Benachrichtigung. Voraussetzung ist, dass es Athena nach Auftreten des Fehlers noch gelingt, die `finalize` Methode aufzurufen.

`RM_Spy` verwendet den `RM_SteeringSvc` und dessen Funktionalität für die Kommunikation und den Datenaustausch. Die Aufgabe des `RM_Spy` besteht darin, die Jobbeschreibungsdatei zu ermitteln und zu registrieren und Dateinamen von ROOT-Dateien, den Ereigniszähler und die Steuerungsvariable zu registrieren. Außerdem werden die Standardbenachrichtigungen von `RM_Spy` initialisiert. Außerdem enthält `RM_Spy` einen Synchronisationspunkt.

Die Funktionalität von `RM_Spy` kann genutzt werden, indem lediglich wenige Zeilen in den Joboptions hinzugefügt werden.

RM_SteeringSvc

`RM_SteeringSvc` ist ein Athena-Dienst, der die Steeringfunktionalität anderen Komponenten zugänglich macht. Er wird intern von den anderen drei Integrationskomponenten verwendet. Er enthält Methoden, mit deren Hilfe

- Daten für das Steering registriert werden können;
- Daten aktualisiert werden können;
- Benachrichtigungen verschickt werden können;
- der Job angehalten, beendet, fortgesetzt, neu gestartet oder bis zum nächsten Synchronisationspunkt bearbeitet werden kann oder
- ein Synchronisationspunkt ausgeführt werden kann.

`RM_SteeringSvc` kann dazu genutzt werden, interne Daten aus eigenen Komponenten dem Online-Steering-System zugänglich zu machen. Zur Verwendung des `RM_SteeringSvc` muss allerdings der Quellcode der entsprechenden Komponente geändert werden, um auf den Dienst zugreifen zu können.

RM_Checker

`RM_Checker` ist ein weiterer Athena-Algorithmus. Er fügt einen Synchronisationspunkt aus. Er kann daher genutzt werden, um ein Ereignis in kleineren Schritten nachzuverfolgen oder um die Aktualisierungsfrequenz zu erhöhen.

RM_EvaluatorBase

`RM_EvaluatorBase` ist eine Basisklasse für benutzerdefinierte, automatisierte Überwachung und Benachrichtigung. Diese Funktionen werden als Athena-Algorithmen über die Joboptions in einen Job eingebaut. Der Benutzer kann eigene, automatisierte Überwachungen erzeugen, indem er die `checkCondition` Methode überschreibt. Er kann in dieser Methode beliebige Auswertungen durchführen und eine Benachrichtigung auslösen, wenn die Bedingung erfüllt ist.

Die Synchronisation mit der Anwendung wird durch das Einfügen von `RM_Spy` oder `RM_Checker` Algorithmen an der entsprechenden Stelle in der Algorithmensequenz erreicht. Das System ist um weitere Funktionalitäten oder steuerbare Daten erweiterbar, indem eine weitere Komponente eingebunden wird, die diese Daten oder Aktionen registriert. Da die Zugriffe von der Anwendung passiv sind, wird ein Aktualisierungsprotokoll verwendet. Ein Invalidierungsprotokoll wird nicht verwendet, da das Steering-System nicht weiß, wann ein Prozess einen Wert liest.

Eine Ausnahme bilden die ROOT-Dateien, bei denen der Aufwand, bei jedem Ereignis eine Kopie zu erzeugen, zu groß wäre. ROOT-Dateien enthalten komplexe hierarchische Strukturen, die in einem Verzeichnis am Anfang der Datei abgelegt sind. Um dieses Verzeichnis zu speichern und alle gepufferten Daten zu schreiben, wird der Prozess geteilt und der Kind-Prozess beendet. Dabei speichert Athena alle bisherigen Ergebnisse. ROOT-Dateien werden nur auf Anforderung geöffnet. Es wird dann eine Kopie erzeugt, auf die blockweise asynchron zugegriffen werden kann. Es kann gleichzeitig mehrere Kopien derselben ROOT-Datei geben, die auf unterschiedlichem Stand sind.

Unter dem Namen der ROOT-Datei wird eine Zeichenkette registriert, die eine spezielle Serialisierungsmethode verwendet. Wird die ROOT-Datei vom Steuerungswerkzeug geöffnet, wird die Zeichenkette angefordert. Die Serialisierungsmethode erstellt eine Kopie der ROOT-Datei und registriert die Kopie im Steering-System. Anschließend schreibt die Serialisierungsmethode den Bindungsnamen der Kopie in die Antwort. Wenn das Steuerungswerkzeug die Antwort empfängt, kennt es den Bindungsnamen der neuen Kopie und registriert ebenfalls eine Datei mit diesem Namen. Alle weiteren Lese- und Schreiboperationen werden auf dieser Kopie ausgeführt. Das Nachladen von Blöcken der Kopie erfolgt mittels der PRAM-Konsistenz, da die Anwendung diese Kopie nicht mehr ändert und auch nicht aus ROOT-Dateien liest.

6.4 Integration von RMOST in die Visualisierungssoftware

Für die Visualisierung von Ergebnissen wird ROOT [27, 28] verwendet. Es ist gelungen, ROOT auch für die Online Visualisierung von Zwischenergebnissen zu verwenden, ohne den Quellcode von ROOT verändern zu müssen. Der Schlüssel hierzu war die Verwendung der Dateizugriffsbibliothek für die Umlenkung von Dateizugriffen.

Es wurden zwei neue Klassen für ROOT entwickelt, die dynamisch in ROOT geladen und verwendet werden können. Die Klasse `TResultMonitorBrowser` stellt eine graphische Benutzeroberfläche für die Visualisierung von Zwischenergebnissen bereit, während die Klasse `TResultMonitorData` den Zugriff auf die Daten eines Gridjobs via Kommandozeile ermöglicht.

Das Hauptfenster (siehe Abb. 6.3) der graphischen Benutzeroberfläche gibt eine Übersicht über alle geöffneten Gridjobs. Für jeden Job kann eine Detailansicht geöffnet werden, die eine Liste der verfügbaren Daten anzeigt (siehe Abb. 6.4). Bei Standarddatentypen wird der aktuelle Wert angezeigt. ROOT-Dateien können mit Hilfe der ROOT-Klasse `TBrowser` angezeigt werden (siehe Abb. 6.5). Die Dateizugriffe von `TBrowser` werden dabei zu der entfernten Datei umgelenkt.

Die graphische Benutzeroberfläche verwendet die Spezielle Schwache Konsistenz. Sie führt in einer Schleife Synchronisationspunkte für alle Protokolle aus, deren Detailansicht nicht geöffnet ist. Dies dient dazu, die Fortschrittsanzeige (Anzahl der berechneten Ereignisse) fortlaufend zu aktualisieren. Sobald die detailierte Ansicht eines Jobs geöffnet wird, werden keine automatischen Synchronisationspunkte mehr für das Protokoll zu dem entsprechenden Job ausgeführt, sondern nur noch wenn der Benutzer auf `synchronize` klickt.

Mit `TResultMonitorData` können ROOT-Dateien und Variablen bei dem Steering-System registriert werden. Über registrierte Daten können von ROOT aus Werte im Job geschrieben oder gelesen werden. Die Zugriffe auf (ROOT-) Dateien können mit der Dateizugriffsbibliothek abgefangen werden. Sie sind also aktiv. Daher können Dateien nach Registrierung von beliebigen anderen ROOT-Klassen verwendet und auf die entfernte Datei umgelenkt werden. Zugriffe auf Variablen sind passiv.

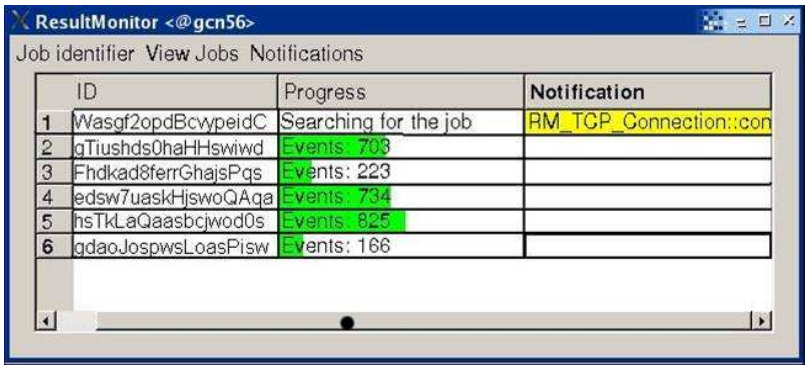


Abbildung 6.3: Das Hauptfenster der Visualisierung von RMOST

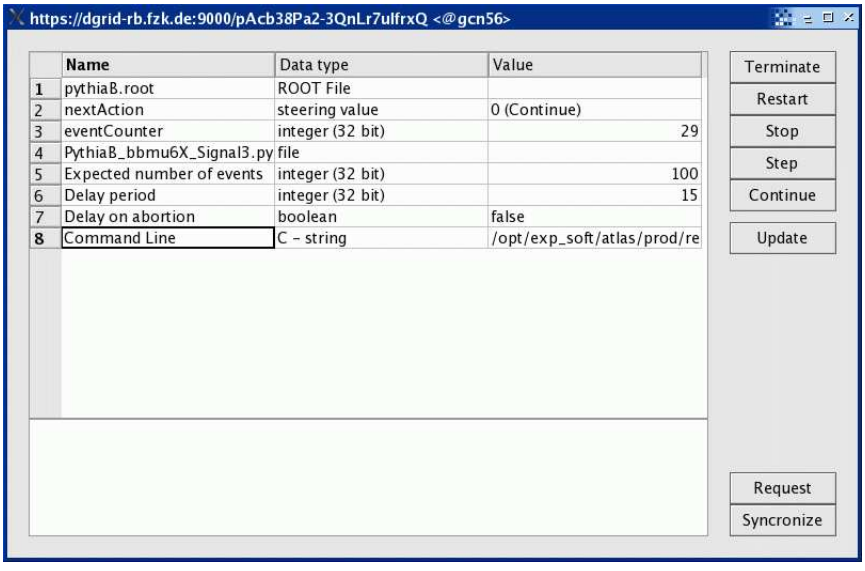


Abbildung 6.4: Die detailierte Ansicht eines Jobs in der Visualisierung von RMOST

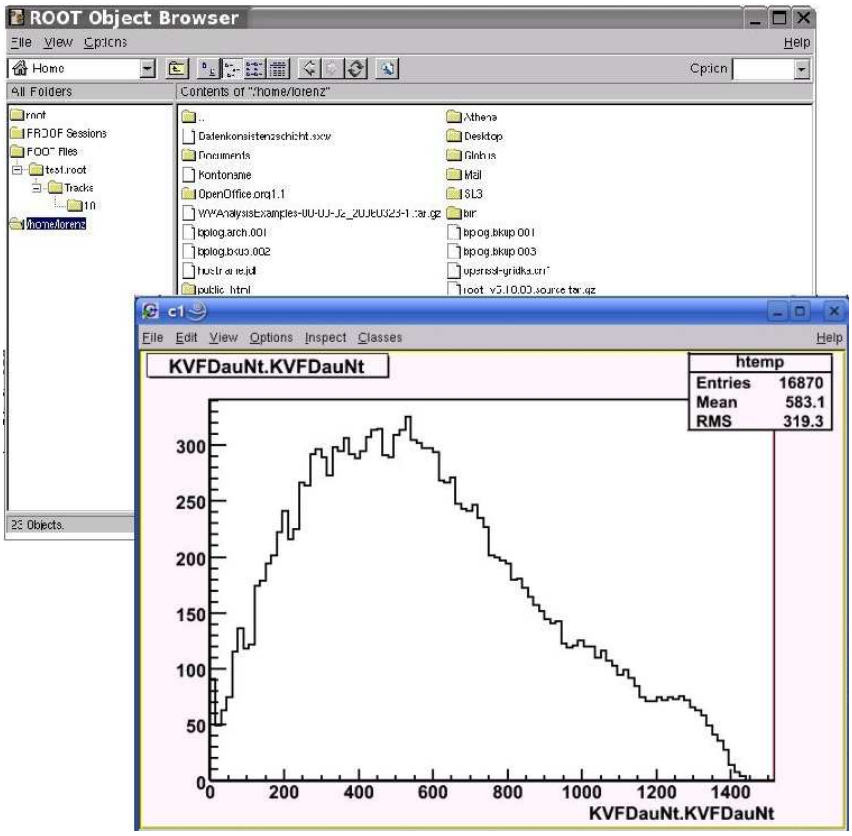


Abbildung 6.5: Der ROOT-Dateibrowser mit Histogramm

6.5 Fazit

In diesem Kapitel wurde das ATLAS-Experiment und die Integration von RMOST in die ATLAS-Software beschrieben. Durch die modulare Struktur der ATLAS-Software lassen sich Steuerungsfunktionen, die das Unterbrechen, Fortsetzen oder Abbrechen des Jobs ermöglichen leicht in die Software einfügen.

Für das Steering von Athena-Jobs muss die Intra-Prozess Bedingung erfüllt werden. Für die Integration in das Athena-Framework wurden zusätzliche Komponenten erstellt, die durch eine Änderung der Jobbeschreibungsdatei in den Job integriert werden können. Auch für das ROOT-Toolkit auf der Visualisierungsseite wurden zusätzliche Klassen erstellt, die den Zugriff auf die Daten des Jobs ermöglichen. Da die wichtigsten Daten in ROOT-Dateien gespeichert werden, können diese Daten durch die transparente Umlenkung von Dateizugriffen mit Hilfe der Offline-Visualisierungswerkzeuge von ROOT online visualisiert werden. Somit braucht die vorhandene Software auf Anwendungs- und Visualisierungsseite nicht verändert oder neu geschrieben zu werden, um Online-Steering zu ermöglichen.

Kapitel 7

Evaluation

Die entwickelten Ansätze wurden anhand der erstellten Implementierung getestet und evaluiert. Bei der Kommunikationsschicht lag der Fokus hierbei auf dem Einfluss der verschiedenen Szenarien und Sicherheitseinstellungen auf den Durchsatz und der Round-Trip-Zeit (siehe Kap. 7.1). Bei der Datenkonsistenzschicht wurde das Verhalten der verschiedenen Konsistenzprotokolle untersucht (siehe Kap. 7.2). Außerdem wurden Messungen zum entfernten Dateizugriff durchgeführt.

7.1 Performance der Grid Verbindung

Die in Kapitel 4.1 und 5.1 entwickelte und implementierte Kommunikationsschicht wurde einem Performancetest unterzogen. Dabei werden insbesondere die Auswirkungen der unterschiedlichen Szenarien und Sicherheitsstufen untersucht. Die wesentlichen Eigenschaften, die die Datenübertragung charakterisieren, sind der Durchsatz und die Round-Trip-Zeit der Verbindung. Für die Nutzung in einem Produktionsgrid in der Größenordnung des LCG, ist eine weitere wichtige Frage die Skalierbarbeit des Verbindungsdienstes. Schließlich wurde auch die Zeit gemessen, die für den Verbindungsaufbau notwendig ist.

7.1.1 Messung des Durchsatzes

Bei dem ersten Experiment waren der Konnektor und der Verbindungsdienst auf zwei verschiedenen Rechnern im selben Cluster installiert und über ein 1 Gb/s Ethernet Netzwerk miteinander verbunden. Beide liefen auf SuperMicro SuperServer 6013P-I Rechnern mit jeweils 2 Intel XEON CPUs, die mit 2,66 GHz getaktet waren. Der Akzeptor lief auf einem Dell GX400 mit einem Intel Pentium 4 Prozessor mit 1,7 GHz Taktrate. Der Akzeptor war mit dem Konnektor und dem Verbindungsdienst durch ein 100 Mb/s Netzwerk über 5 Hops verbunden.

Um den Durchsatz zu messen, wurden 10 MB Daten vom Konnektor zum Akzeptor gesendet. Diese Datenmenge kann von realen Anwendungen leicht erreicht werden. Beispielsweise können im ATLAS Experiment Ergebnisdateien mehrere GB groß werden. Die Daten wurden dabei in mehrere Nachrichten gleicher Größe aufgeteilt. Das Experiment wurde für Nachrichtengrößen zwischen 64

Byte und 256 kB durchgeführt. Wenn der Akzeptor die 10 MB Daten vollständig empfangen hat, wurde eine kurze Benachrichtigung zurückgesendet. Der Konnektor maß die Zeit t_{tp} zwischen dem Beginn der Datenübertragung und dem Empfang der Benachrichtigung, dass die Daten vollständig vom Akzeptor empfangen wurden. Folgende Faktoren fließen in t_{tp} ein:

- Die Rechenzeit t_w auf dem sendenden Rechner, um die Nachricht den Sicherheitsanforderungen entsprechend einzupacken. Dabei wird die Nachricht z.B. mit einem Nachrichten Integritäts Checkcode (MIC-Code) versehen oder verschlüsselt, falls Verschlüsselung verlangt wurde.
- Die Übertragungszeit t_t zum Akzeptor. Je nach Szenario ist dies die Übertragung vom Konnektor zu Akzeptor (offenes Szenario) oder von CS zum Akzeptor (halboffenes oder geschlossenes Szenario).
- Die Rechenzeit t_u des empfangenden Rechners, um die Nachricht auszupacken. Dabei wird z.B. die Integrität der Nachricht überprüft oder eine verschlüsselte Nachricht dekodiert. Falls der sendende und empfangene Rechner unterschiedlich schnell sind, können t_w und t_u unterschiedliche Werte haben.
- Die Rechenzeit t_{CS}, die der Verbindungsdienst verbraucht.
- Die Übertragungszeit t_{tCS} zwischen dem Konnektor und dem Verbindungsdienst.
- Die Zeit t_r, die die Rückantwort benötigt.

Da bei diesem Experiment der Konnektor und der VD zum selben Cluster gehörten und über denselben Switch mit der Außenwelt verbunden waren, ist die Übertragunszeit t_t vom VD zum Akzeptor dieselbe wie vom Konnektor zum Akzeptor. Das bedeutet, wenn in eine direkte Verbindung vom Konnektor zum Akzeptor ein VD eingefügt wird, t_t gleich bleibt. Es muss dann nur die Rechenzeit des Verbindungsdienstes t_{CS} und die Übertragungszeit vom Konnektor zum VD t_{tCS} hinzu addiert werden.

Da die einzelnen Schritte von verschiedenen Hardwarekomponenten (verschiedene CPUs, Netzwerkadapter) ausgeführt werden, können verschiedene Nachrichten gleichzeitig in den verschiedenen Komponenten bearbeitet werden. Dadurch arbeitet die Nachrichtenübertragung wie eine Pipeline. In einer Pipeline wird der Durchsatz durch die langsamste Pipelinestufe bestimmt. Die gemessene Zeit für den Durchsatz beträgt:

- Die Zeit, die die erste Nachricht braucht.
- plus $n-1$ mal die benötigte Zeit für eine Nachricht für die langsamste Pipelinestufe.
- plus die Zeit die die Rückantwort braucht:

$$\begin{aligned} t_{tp} &= t_w + t_t + t_u + t_{CS} + t_{tCS} \\ &\quad + (n-1)\max(t_w, t_t, t_u, t_{CS}, t_{tCS}) \\ &\quad + t_r \end{aligned} \tag{7.1}$$

	offen		halboffen		geschlossen	
	t_p	Std.abw.	t_p	Std.abw.	t_p	Std.abw.
TCP	0.959 s	6,780 ms				
1 Sicher.	0.959 s	2,879 ms	0.983 s	15,77 ms	0.972s	9,642 ms
2 Sicher.	1.483 s	1,366 ms	1.496 s	25,24 ms	1.516s	41,24 ms
Verschl.	7.180 s	28,77 ms	7.294 s	43,17 ms	7.234s	43,12 ms

Tabelle 7.1: Gemessene Zeit um 10 MB mit 8 kB großen Nachrichten über ein 100 Mb/s LAN zu übermitteln

Die gemessene Zeit für den Durchsatz wird also von der max-Klausel dominiert. Die ermittelten Zeiten für 8 kB große Nachrichten sind in Tabelle 7.1 dargestellt. Der daraus resultierende Durchsatz ist in Abbildung 7.2 illustriert.

Als Vergleich wurde der erreichte Durchsatz über die TCP Verbindung mit aufgesetzter Nachrichtenschicht verwendet. Der beste Durchsatz wurde bei 8 kB großen Nachrichten erreicht (siehe Abb. 7.1). Die Abhängigkeit des Durchsatzes von der Nachrichtengröße wird vor allem durch die Globus GSSAPI verursacht. Das offene Szenario mit einer Sicherheitsschicht entspricht einer TCP Verbindung, die mit der Globus GSSAPI gesichert wurde. Der Unterschied zwischen der TCP Verbindung und dem offenen Szenario wird nur durch die Globus GSSAPI verursacht. Wie in Abbildung 7.1 zu erkennen ist, hängt Globus deutlich stärker von der Nachrichtengröße ab als TCP. Natürlich wird dieses Muster bei der Verwendung von zwei Sicherheitsschichten im geschlossenen Szenario noch verstärkt.

Ein weiterer Grund, der auch den geringeren Durchsatz bei kleinen Nachrichten beeinflusst, ist, dass kleinere Nachrichten einen größeren Overhead haben.

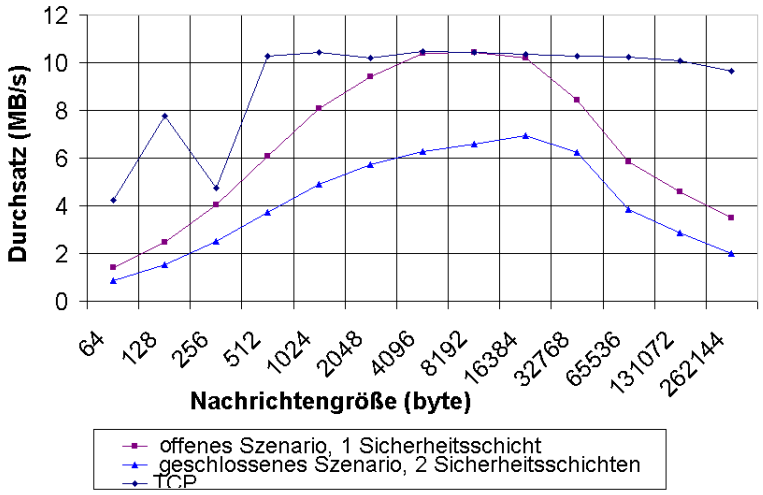


Abbildung 7.1: Durchsatz in Abhängigkeit von der Nachrichtengröße in einem 100 Mb/s LAN

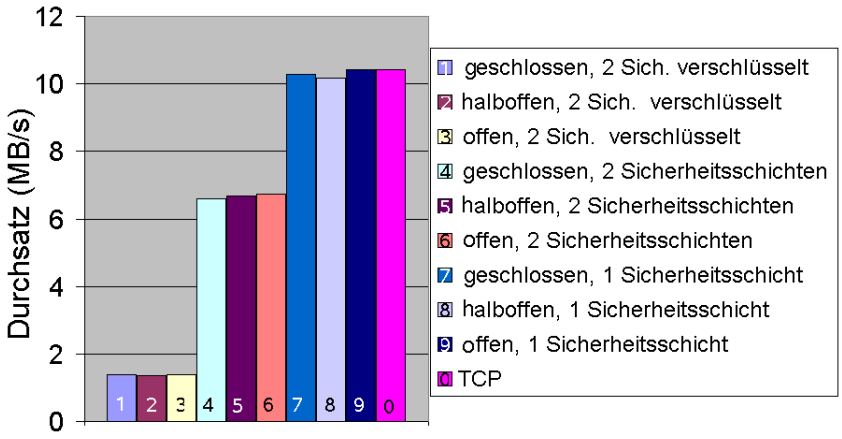


Abbildung 7.2: Durchsatz bei 8 kB großen Nachrichten in einem 100 Mb/s LAN

Der folgende Vergleich basiert auf dem Durchsatz bei 8 kB großen Nachrichten. Für die TCP Verbindung wurde ein Durchsatz von 10,6 MB/s erreicht, was mehr oder weniger der verfügbaren Netzwerkbandbreite entspricht. In diesem Fall waren nur t_t und t_r größer als Null. Fügt man der TCP-Verbindung eine einzelne Authentifizierungsschicht hinzu, ist im Schnitt kein Unterschied messbar. Die Rechenzeiten für t_w und t_u sowie die leicht erhöhte Zeit t_r für die Rückantwort, sind deutlich kleiner als die Standardabweichung. Der Durchsatz wird immer noch von der Übertragungszeit t_t bestimmt, der die max-Klausel dominiert und die ist in beiden Fällen gleich. Durch das Überlappen der einzelnen Schritte der Datensicherung und Übertragung, hat die erste Sicherheitsschicht keinen Einfluss auf den Durchsatz.

Das sieht anders aus, wenn eine zweite Sicherheitsschicht für die Ende-zu-Ende Authentifizierung verwendet wird. Der Durchsatz sinkt dann auf 6,7 MB/s. In diesem Fall ist die Rechenzeit für t_u und t_w auf der langsameren Maschine größer als t_t und dominiert die max-Klausel. Der Durchsatz wird jetzt nicht mehr durch die Netzwerkbandbreite limitiert, sondern durch die Rechenleistung des langsameren Rechners. Dies wird nochmals durch das zweite Experiment in Abschnitt 7.1.2 bestätigt.

Wenn zusätzlich noch die Verschlüsselung der Ende-zu-Ende Sicherheitsschicht aktiviert wird, erhöht sich der Rechenaufwand noch mehr und t_u und t_w werden noch größer. Dadurch sinkt der Durchsatz auf 1,4 MB/s im geschlossenen Szenario.

Das geschlossene und halboffene Szenario haben in allen drei Sicherheitsstufen jeweils denselben Durchsatz. Abweichungen werden durch Messfehler verursacht. Verglichen mit dem offenen Szenario ist die gemessene Zeit t_p nur geringfügig größer. Die erste Nachricht braucht die Zeit $t_{CS} + t_{tCS}$ länger und t_r wird um denselben Wert größer. Die max-Klausel wird aber weiterhin durch t_t bzw. t_u, t_w dominiert. Das heißt, dass der Durchsatz durch den Einsatz eines Verbindungsdienstes nicht verringert wird.

Szenario	t_p	Std.abw.	Durchsatz
TCP	0,0913 s	0,177 ms	109,5 MB/s
offenes Szenario, 1 Sicherheitsschicht	0,4212 s	0,538 ms	23,7 MB/s
offenes Szenario, 2 Sicherheitsschichten	0,7889 s	0,464 ms	12,7 MB/s
offenes Szenario, 2 Sicherh., verschl.	4,4754 s	1,561 ms	2,2 MB/s
halboffenes Sz., 1 Sicherheitsschicht	0,8392 s	26,19 ms	11,9 MB/s
halboffenes Sz., 2 Sicherheitsschichten	0,8646 s	26,41 ms	11,6 MB/s
halboffenes Sz., 2 Sicherh., verschlüsselt	4,5548 s	25,93 ms	2,2 MB/s
geschlossenes Sz., 1 Sicherheitsschicht	0,8308 s	17,90 ms	12,0 MB/s
geschlossenes Sz., 2 Sicherheitssch.	0,8559 s	13,86 ms	11,7 MB/s
geschlossenes Sz., 2 Sicherh. verschl.	4,4673 s	6,687 ms	2,2 MB/s

Tabelle 7.2: Gemessene Zeit um 10 MB mit 8 kB großen Nachrichten innerhalb eines Clusters über ein 1 Gb/s LAN zu übermitteln und daraus resultierender Durchsatz

7.1.2 Skalierbarkeit des Verbindungsdienstes

In einem zweiten Experiment wurde die Skalierbarkeit des Verbindungsdienstes untersucht. Das Ziel dieses Experimentes war es, herauszufinden in welchen Fällen der VD den Durchsatz begrenzt. Außerdem wurde in dem zweiten Experiment die Ergebnisse des Experiments aus Abschnitt 7.1.1 bestätigt.

In diesem Experiment liefen der Konnektor, der Akzeptor und der VD auf einem SuperMicro SuperServer und waren untereinander durch ein 1 Gb/s LAN verbunden. Das Ergebnis ist in Tabelle 7.2 dargestellt. Der Durchsatz ist in Abb. 7.3 illustriert.

In diesem Experiment war der Durchsatz einer TCP Verbindung mehr als 4 mal so groß wie der Durchsatz im offenen Szenario mit einer Sicherheitsschicht. Das bedeutet, dass in diesem Experiment schon mit der ersten Sicherheitsschicht

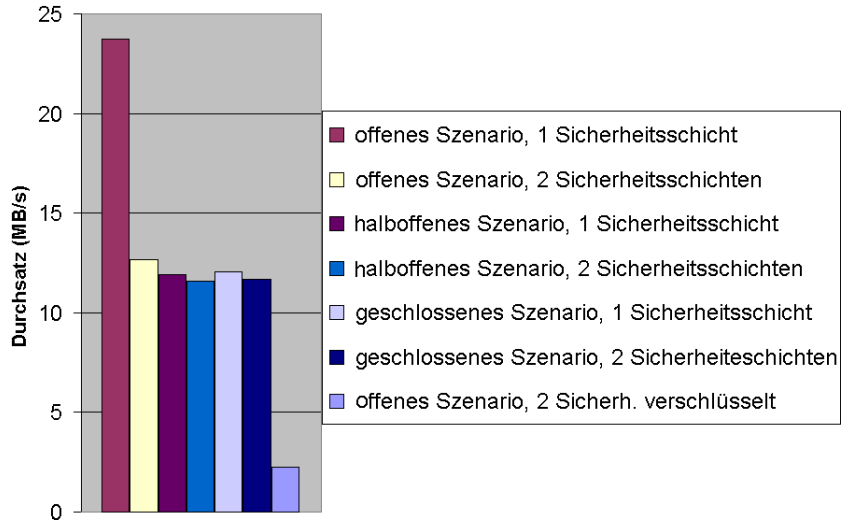


Abbildung 7.3: Durchsatz bei 8 kB großen Nachrichten in einem 1 Gb/s LAN

die Rechenzeit zum Ein- und Auspacken der Nachrichten den Durchsatz limitiert. Wenn zwei Sicherheitsschichten verwendet werden, verdoppelt sich t_p gegenüber dem entsprechenden Szenario mit einer Sicherheitsschicht beinahe, da sich t_u und t_w verdoppeln.

Das offene Szenario einer Sicherheitsschicht hat mehr als den doppelten Durchsatz des halboffenen Szenarios mit einer Sicherheitsschicht. Allerdings wird der Durchsatz im halboffenen Szenario durch die Verwendung einer zweiten Sicherheitsschicht nicht wesentlich gesenkt, während sich der Durchsatz im offenen Szenario beinahe halbiert, wenn eine zweite Sicherheitsschicht verwendet wird. Dieses Verhalten unterschiedet sich von den Ergebnissen des ersten Experiments, in dem die Durchsätze bei Verwendung gleicher Sicherheitsschichten fast gleich waren, aber deutliche Unterschiede bei Verwendung unterschiedlicher Sicherheitsschichten zu sehen waren. Verursacht wird dieses Verhalten durch den Verbindungsdienst, der eine empfangene Nachricht auspackt und anschließend wieder einpackt. Beides geschieht in einem Schritt, der sich nicht überlappt. Daraus folgt, dass $t_{CS} = t_w + t_u$ ist. Bei Verwendung einer Sicherheitsschicht wird in diesem Experiment also t_{CS} zum limitierenden Faktor, der sich so verhält, als ob zwei Sicherheitsschichten hintereinander ausgeführt werden. Wenn zwei Sicherheitsschichten verwendet werden, ändert sich daher der Durchsatz beim halboffenen und geschlossenen Szenario nicht wesentlich.

Vergleicht man die Werte aus dem Experiment, dass in in Abschnitt 7.1.1 beschrieben wurde, mit diesem Experiment, stellt man fest, dass vergleichbare Szenarien in dem zweiten Experiment einen deutlich höheren Durchsatz erreichten. In dem halboffenen Szenario beträgt t_p im zweiten Experiment nur 58,3% des Wertes wie im ersten Experiment. Dies kann auf die geringere Rechenleistung des Rechners zurückgeführt werden, der dort als Akzeptor verwendet wurde. Die Taktfrequenz des Akzeptors aus dem ersten Experiment beträgt 63,9% der Taktfrequenz aller Rechner, die in diesem Experiment verwendet wurden.

Für die Skalierung des VD ist der maximale Durchsatz, der durch ein VD möglich ist, entscheidend. Auf den SuperMicro Rechnern hat ein VD einen Durchsatz von bis zu 11,9 MB/s. Wenn also die Netzwerkbandbreite zum Internet 100 Mb/s beträgt, ist ein einzelner VD für die Site vollkommen ausreichend. Mehrere oder schnellere VD könnten auch nicht mehr Durchsatz erzeugen, da der Durchsatz durch die Netzwerkbandbreite begrenzt ist. Die Anzahl der Sites spielt für die Skalierung des Verbindungsdienstes keine Rolle, da die einzelnen Verbindungsdienste untereinander unabhängig sind.

Falls eine höhere Netzwerkbandbreite verfügbar ist, können Sites mehrere Verbindungsdienste installieren. Das erhöht zwar nicht den Durchsatz für einen einzelnen Job, aber mehrere Jobs könnten die gesamte Bandbreite nutzen. Wenn ein einzelner Job mehrere Streams über zwei verschieden Verbindungsdienste verwendet, kann der Job den doppelten Durchsatz nur erreichen, falls er nur eine Sicherheitsschicht verwendet. Bei zwei Sicherheitsschichten ergibt sich kein höherer Durchsatz durch die Verwendung mehrerer VD, da der Durchsatz bereits durch die Rechenleistung des Endpunktes der Verbindung beschränkt ist.

7.1.3 Durchsatz in einem realen Grid

Die bisherigen Messungen fanden alle in einem lokalen Netzwerk statt. Dies ermöglicht rekonstruierbare Ergebnisse in einer Umgebung mit stabilen Parametern. Inwiefern aber sind diese Ergebnisse auf ein reales Grid übertragbar?

Szenario	t_p	Std.abw.	Durchsatz
offenes Szenario, 1 Sicherheitsschicht	2,341 s	32,6 ms	4,272 MB/s
halboffenes Sz., 2 Sicherheitsschichten	2,315 s	30,9 ms	4,320 MB/s

Tabelle 7.3: Gemessene Zeit für die Übertragung von 10 MB mit 8 kB großen Nachrichten zwischen Gridsites in Siegen und Freiburg und daraus resultierendem Durchsatz

Um dies zu beantworten wurden Messungen mit einem Gridjob, der auf der Gridsite der Universität Freiburg lief, durchgeführt. Dabei wurden wieder 10 MB Daten von dem Gridjob in Freiburg zu dem User Interface in Siegen verschickt. Nachdem die Daten auf dem User Interface in Siegen empfangen wurden, wurde eine Bestätigungsnachricht zurück geschickt. Es wurde die Zeit vom Beginn des Sendens bis zum Empfang der Bestätigungsnachricht gemessen.

Dies wurde für das halboffene Szenario mit 2 Sicherheitsschichten und dem offenen Szenario mit einer Sicherheitsschicht durchgeführt. Die Ergebnisse sind in Tabelle 7.3 enthalten. Letztendlich sind die gemessenen Zeiten im Rahmen der Messgenauigkeit gleich. In beiden Fällen ist die verfügbare Bandbreite der limitierende Faktor.

Die unterschiedlichen Szenarien und der Anzahl der Sicherheitsschichten verursachen keinen nachweisbaren Unterschied beim Durchsatz. Allerdings ist der erreichte Durchsatz für das halboffene Szenario nur 35 % von dem Wert im ersten Experiment entfernt. Zwischen zwei Tier 2 Sites mit einer 10 GB/s Verbindung könnte genug Bandbreite verfügbar sein, dass bei 2 Sicherheitsschichten die Rechenzeit die Begrenzung wird.

7.1.4 Round-Trip-Zeit

Neben dem Durchsatz wurde die Round-Trip-Zeit getestet. Dafür wurde derselbe Aufbau wie in Abschnitt 7.1.1 verwendet. Um die Round-Trip-Zeit zu ermitteln, wurde ein 2 kB große Nachricht zwischen dem Konnektor und dem Akzeptor 1000 mal hin und her geschickt. Die Nachricht wurde dabei jeweils nach dem Empfangen wieder zurückgesendet. Die Round-Trip-Zeit t_{re} lässt sich

Szenario	t_{re}	Std.abw.
TCP	6,018 ms	3,212 μs
offenes Szenario, 1 Sicherheitsschicht	6,508 ms	4,911 μs
offenes Szenario, 2 Sicherheitsschichten	7,013 ms	11,47 μs
offenes Szenario, 2 Sicherh., verschl.	10,558 ms	78,16 μs
halboffenes Sz., 1 Sicherheitsschicht	7,525 ms	21,26 μs
halboffenes Sz., 2 Sicherheitsschichten	7,723 ms	166,2 μs
halboffenes Sz., 2 Sicherh., verschlüsselt	11,692 ms	335,2 μs
geschlossenes Sz., 1 Sicherheitsschicht	7,482 ms	154,7 μs
geschlossenes Sz., 2 Sicherheitssch.	8,060 ms	256,5 μs
geschlossenes Sz., 2 Sicherh. verschl.	11,798 ms	324,1 μs

Tabelle 7.4: Round-Trip-Zeit von 2kB großen Nachrichten in einem 100 Mb/s LAN über 5 hops

Komponente	
t_t	6,0 ms
$t_u + t_w$	0,25 ms
t_{CS}	0,25 ms
t_{tCS}	0,25 ms
Verschlüsselung	3,6 ms

Tabelle 7.5: Summanden der Round-Trip-Zeit von 2 kB großen Nachrichten in einem 100 Mb/s LAN über 5 hops

in folgende Summanden aufschlüsseln:

$$t_{re} = t_{w,1} + t_{u,1} + 2t_t + 2t_{CS} + 2t_{tCS} + t_{w,2} + t_{u,2} \tag{7.2}$$

Dabei ist $t_{w,x}$ die Rechenzeit von Rechner x um eine Nachricht einzupacken und $t_{u,x}$ die Rechenzeit von Rechner x um eine Nachricht auszupacken. Analog zu Abschnitt 7.1.1 ist t_t die Übertragungszeit zum Akzeptor, t_{CS} die Rechenzeit des Verbindungsdienstes und t_{tCS} die Übertragungszeit vom Konnektor zum VD. Da Globus ein symmetrischen Kryptographieverfahren verwendet, sind für einen Rechner x die Zeiten $t_{w,x}$ und $t_{u,x}$ gleich. Es gilt also: $t_{w,1} = t_{u,1}$ und $t_{w,2} = t_{u,2}$.

Die Messwerte für die Round-Trip-Zeit sind in Tabelle 7.4 dargestellt. Als Vergleichsgrundlage dient eine TCP-Verbindung mit der Nachrichtenschicht, die eine Round-Trip-Zeit von 6,0 ms hatte. Das offene Szenario mit einer Sicherheitsschicht entspricht der TCP Verbindung, auf die eine Sicherheitsschicht aufgesetzt wurde. Die Sicherheitsschicht verbraucht 0,5 ms zusätzliche Rechenzeit, so dass die Round-Trip-Zeit im offenen Szenario mit einer Sicherheitsschicht 6,5 ms beträgt. Eine zweite Sicherheitschicht im offenen Szenario mit zwei Sicherheitsschichten braucht nochmals 0,5 ms mehr. Diese Abstufung ist auch zwischen den Messwerten für eine und zwei Sicherheitsschichten im geschlossenen Szenario zu beobachten.

Vergleicht man die Round-Trip-Zeiten im offenen Szenario mit den Round-Trip-Zeiten des geschlossenen Szenarios, so stellt man fest, dass die Round-Trip-Zeit im geschlossenen Szenario ungefähr 1 ms größer ist als die Round-Trip-Zeit im offenen Szenario mit gleicher Anzahl Sicherheitsschichten. Die zusätzliche ms wird durch den VD verursacht und lässt sich aufteilen in $2t_{CS} + 2t_{tCS}$. Da der VD jede Nachricht einmal auspacken und neu einpacken muss, sollte $2t_{CS}$ ungefähr so viel Zeit verbrauchen wie eine Sicherheitsschicht. Folglich beträgt $2t_{CS} \approx 0.5$ ms. Die 0,5 ms zusätzliche Round-Trip-Zeit für eine Sicherheitsschicht enthalten jeweils zweimal das Einpacken und Entpacken, deshalb wird eine Sicherheitsschicht mit $2t_{CS}$ gleichgesetzt. Die verbleibenden 0.5 ms lassen sich auf die Übertragungszeit innerhalb des Clusters zwischen Konnektor und VD zurückführen.

Die Verschlüsselung verlängerte die Round-Trip-Zeit im offenen Szenario um 3,5 ms und im geschlossenen Szenario um 3,7 ms. Der Unterschied wird durch den Messfehler verursacht. Die einzelnen Bestandteile sind in Tabelle 7.5 nochmals aufgelistet.

Die in Tab. 7.4 und Tab. 7.5 angegeben Werte wurden in einem lokalen Netzwerk ermittelt. Um realistische Round-Trip-Zeiten zu erhalten und die Theorie

Szenario	t_{re}	Std.abw.
offenes Szenario, 1 Sicherheitsschicht	18,29 ms	0,71 ms
halboffenes Sz., 1 Sicherheitsschicht	19,57 ms	0,90 ms
halboffenes Sz., 2 Sicherheitsschichten	19,64 ms	0,34 ms

Tabelle 7.6: Round-Trip-Zeit zwischen Gridsites in Siegen und Freiburg

zu verifizieren wurde ein weiteres Experiment zwischen den Gridsites in Freiburg und Siegen durchgeführt (siehe Tabelle 7.6). Dabei wurde dieselbe Messungen durchgeführt. Die Verzögerung durch eine zusätzliche Sicherheitsschicht liegt innerhalb der Standardabweichung. Die Verzögerung durch den Verbindungsdienst ist gerade noch sichtbar. Die Einflüsse der Sicherheitsschicht und des Verbindungsdienstes auf die Round-Trip-Zeit sind im Grid angesichts einer Round-Trip-Zeit von ca. 19 ms vernachlässigbar.

7.1.5 Verbindungsaufbau

Als Namensdienst wurde R-GMA [30] verwendet. Die Zeit um von R-GMA einen Wert anzufragen beträgt 9 s und mehr. In vereinzelten Fällen hat es bis zu einer 1 Minute gedauert. Die Round-Trip-Zeit von R-GMA dominiert die Zeit zum Verbindungsaufbau. Startet man die Zeitmessung nachdem R-GMA abgefragt wurde, benötigt der Verbindungsaufbau im offenen Szenario mit einer Sicherheitsschicht 66 ms und mit zwei Sicherheitsschichten 163 ms. Eine einfache TCP-Verbindung wird in 3,8 ms aufgebaut. Diese Zeiten wurden ebenfalls in einem 100 Mb/s LAN über 5 hops gemessen.

7.2 Evaluation der Konsistenzprotokolle

In diesem Unterkapitel werden die Eigenschaften der implementierten Konsistenzprotokolle für die Spezielle Schwache Konsistenz und die Verzögerte Schwache Konsistenz (VSK) evaluiert. Als Vergleich wurde das Aktualisierungsprotokoll der PRAM Konsistenz verwendet. Da beim Aktualisierungsprotokoll der PRAM Konsistenz bei jeder Schreiboperation sofort eine Aktualisierung verschickt wird, besteht es im wesentlichen aus der Übertragungszeit der Daten.

Als Test wurden in einem Job und in einem Steuerungswerkzeug jeweils fünf Datenobjekte gleicher Länge registriert. Es wurden tausend Iterationen ausgeführt, die jeweils einen Synchronisationspunkt enthielten. Bei jeder Iteration wurde zufällig entschieden, ob in dieser Iteration alle fünf Werte gelesen oder geschrieben werden. Die Wahrscheinlichkeit für die Lese- und Schreiboperation konnte beim Start des Experimentes definiert werden. Es wurde die Zeit, bis die tausend Iterationen beendet waren, für beide Seiten separat gemessen. Falls die andere Seite ihre 1000 Iterationen noch nicht beendet hatte, wurden weiterhin Synchronisationspunkte ausgeführt, um synchronisierte eingehende Nachrichten zu bearbeiten. Allerdings wurden keine Lese- und Schreiboperationen mehr durchgeführt, wenn ein Prozess seine 1000 Iterationen beendet hatte.

Für die Experimente wurde dasselbe Netzwerk mit denselben Rechnern verwendet wie in Kap. 7.1.1. Das Steuerungswerkzeug lief auf dem SuperServer, der in Kap. 7.1.1 für den Konnektor verwendet wurde. Der Job lief auf dem

	Steerer		Job	
	Zeit	Std.abw.	Zeit	Std.abw.
SSK AktP.	117.34 s	0.27 s	117.25 s	0.31 s
SSK InvP.	39,71 s	3.25 s	61.84 s	3.25 s
VSK AktP.	24.16 s	0.13 s	21.24 s	0.04 s
VSK InvP.	46.71 s	0.55 s	31.54 s	0.70 s
PRAM K. AktP.	21.70 s	0.20 s	22.30 s	0.49 s

Tabelle 7.7: Gemessene Zeit für 1000 Iterationen bei fünf 10 Byte großen Datenobjekten und mit einer Wahrscheinlichkeit von 0.75 für die Lese- und Schreiboperation

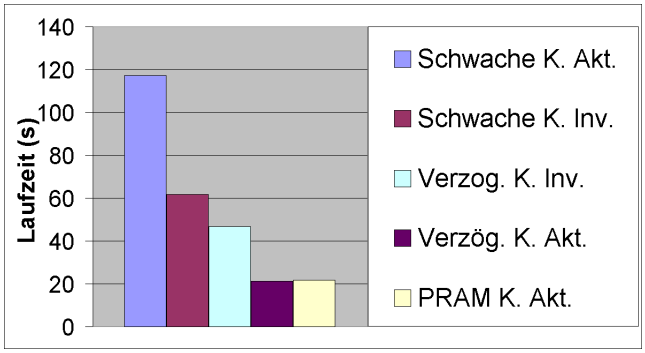


Abbildung 7.4: Laufzeit für 1000 Synchronisationspunkte bei fünf 10 Byte großen Datenobjekten.

Dell, der für den Akzeptor verwendet wurde. Zur Kommunikation wurde das offene Szenario mit 2 Sicherheitsschichten verwendet.

Zuerst wurde eine Datengröße von 10 Byte gewählt um den Fall zu simulieren, dass der Durchsatz deutlich unter der Netzwerkbandbreite liegt. Die Ergebnisse werden in Kap. 7.2.1 dargestellt. Anschließen werden in Kap. 7.2.2 die Messungen mit einer Datengröße von 100000 Byte ausgewertet.

7.2.1 Ergebnisse bei kleinen Datenmengen

Bei diesem Experiment wurde eine Lese- und Schreibwahrscheinlichkeit in jeder Iteration von 0.75 gewählt. Die Datengröße betrug 10 Byte. Nicht benötigte Protokolle wurden aus der Liste der verfügbaren Protokolle entfernt um überflüssige Synchronisationsmechanismen zu beseitigen. Das Ergebnis ist in Tabelle 7.7 in Abb. 7.4 dargestellt.

Auffällig ist hierbei zunächst, dass die Protokolle der Verzögerten Schwachen Konsistenz deutlich kürzere Laufzeiten hatten als die Protokolle der Speziellen Schwachen Konsistenz. Dies lässt sich darauf zurückführen, dass für die Spezielle Schwache Konsistenz eine sequentielle Ordnung der Synchronisationspunkte benötigt wird. Dies bedeutet, dass der Prozess blockiert, bis er das angeforderte Synchronisationsobjekt erhalten hat, bevor er einen Synchronisationspunkt bearbeiten kann. Zum anderen kann sich maximal ein Prozess in einem Synchronisationspunkt befinden. Der andere Prozess muss warten bis der Synchro-

nisationspunkt fertig bearbeitet wurde. Die gemessene Laufzeit setzt sich also aus der Laufzeit von ca. 2000 Synchronisationspunkten und 1000 Round-Trip-Zeiten zusammen. Entfernt man aus dem Quellcode der Protokolle der SSK am Anfang der Synchronisationspunkte die Bedingung, die den Besitz des Synchronisationsobjekts überprüft, erhält man für das (dann inkorrekte) Aktualisierungsprotokoll der Speziellen Schwachen Konsistenz die gleiche Laufzeit wie für das Aktualisierungsprotokoll der PRAM Konsistenz. Allerdings sind dann die Synchronisationspunkte nicht mehr sequentiell konsistent.

Die zweite interessante Beobachtung ist, dass bei der Speziellen Schwachen Konsistenz schon bei kleinen Datenmengen das Invalidierungsprotokoll schneller ist als das Aktualisierungsprotokoll. Bei der Verzögerten Schwachen Konsistenz, ist wie erwartet das Aktualisierungsprotokoll ca. 1.66 mal schneller als das Invalidierungsprotokoll, da die Wartezeit auf die Antwort einer `Request`-Nachricht deutlich höher ist, als die benötigte Zeit zum Versenden der Daten. Die Differenz zwischen dem Aktualisierungsprotokoll und dem Invalidierungsprotokoll entspricht der Zeit, die der Prozess beim Lesen auf eine Aktualisierung warten musste.

Bei der Speziellen Schwachen Konsistenz wird die meiste Zeit damit verbracht auf das Synchronisationsobjekt zu warten. Blockiert der Prozess p bei einer Leseoperation, weil er auf eine Aktualisierung wartet, kann der andere Prozess q in dieser Zeit mehrere Synchronisationspunkte bearbeiten, ohne zwischendurch auf das Synchronisationsobjekt zu warten. Die dadurch eingesparte Wartezeit auf das Synchronisationsobjekt beträgt ein Vielfaches der zusätzliche Wartezeit des lesenden Prozesses. Somit ist bei der Speziellen Schwachen Konsistenz das Invalidierungsprotokoll etwas schneller als das Aktualisierungsprotokoll.

7.2.2 Ergebnisse bei sehr großen Datenmengen

Um den Einfluss der Datengröße auf die Performance der Konsistenzprotokolle zu untersuchen, wurden dieselben Messungen wie in Kap. 7.2.1 nochmals durchgeführt, diesmal aber mit einer Datengröße von 100000 Byte. Die Schreib- und Lesewahrscheinlichkeiten blieben gleich. Die Messergebnisse sind in Tab. 7.8 angegeben, die Laufzeiten in Abb. 7.5 illustriert. Der Erwartungswert für das Volumen der Nutzdaten beträgt in diesem Experiment ca. 350 MB. Bei jeder Aktualisierung werden ca. 0.5 MB Daten verschickt. Die Laufzeiten werden folgendermaßen notiert $t_{Konsistenzmodell,Seite}$ wobei $Konsistenzmodell \in \{PRAM, VSK, SSK\}$ das für die Messung verwendete Konsistenzmodell beschreibt und $Seite \in \{Steerer, Job\}$ spezifiziert, ob die gemessene Zeit des Jobs oder des Steuerungswerkzeugs gemeint ist.

Auswertung Invalidierungsprotokolle

In diesem Experiment benötigte das Invalidierungsprotokoll der VSK weniger als ein Achtel der Zeit des Aktualisierungsprotokolls. Die Differenz der Laufzeit des Invalidierungsprotokolls der VSK bei 10 Byte und bei 100000 Byte ist gering. Weitere Messungen mit anderen Datengrößen zeigen, dass die Laufzeiten bei dem Invalidierungsprotokoll der VSK von der Datengröße praktisch nicht beeinflusst wird (siehe Abb. 7.6).

	Steerer		Job	
	Zeit	Std.abw.	Zeit	Std.abw.
SSK AktP.	412.21 s	6.31 s	412.25 s	6.35 s
SSK InvP.	44.46 s	5.77 s	68.51 s	11.07 s
VSK AktP.	395.82 s	50.97 s	235.50 s	29.90 s
VSK InvP.	45.70 s	1.25 s	28.72 s	1.18 s
PRAM K. AktP.	249.84 s	17.96 s	90.33 s	2.03 s

Tabelle 7.8: Gemessene Zeit für 1000 Iterationen bei fünf 100000 Byte großen Datenobjekten und mit einer Wahrscheinlichkeit von 0.75 für die Lese- und Schreiboperation

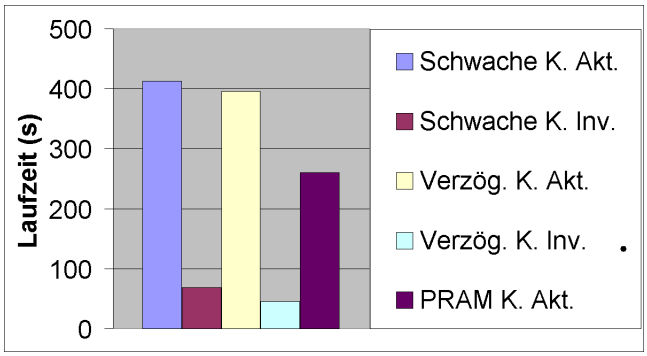


Abbildung 7.5: Laufzeit des langsameren Prozesses für 1000 Synchronisationspunkte bei fünf 100000 Byte großen Datenobjekten.

Dass die Datengröße die Laufzeit des Invalidierungsprotokolls der VSK nicht beeinflusst, liegt an folgendem Effekt: Angenommen Prozess p fordert in einer Leseoperation eine Aktualisierung von Prozess q an. Die `Request`-Nachricht, kann von q asynchron beantwortet werden. Dadurch wird der Anwendungsthread von q nicht blockiert. Auf der anderen Seite ist der Anwendungsthread von p blockiert, bis der angeforderte Wert empfangen wurde. p kann also keine Invalidierungen mehr an q schicken. Daher kann q eine Sequenz von Synchronisationspunkten ausführen, ohne in Leseoperationen auf Aktualisierungen warten zu müssen. Je länger die Datenübertragung dauert, desto länger ist die Sequenz. Als Folge werden bei größeren Datenobjekten seltener Aktualisierungen durchgeführt. Dadurch müssen weniger Daten verschickt werden, wodurch sich das Invalidierungsprotokoll automatisch an die verfügbare Netzwerkbandbreite anpasst. Also muss ein Prozess zwar länger auf eine Aktualisierung warten, dafür werden aber die Aktualisierungen auch seltener. Das Steuerungswerkzeug erhält bei regelmäßigen Änderungen eines Wertes die Aktualisierungen in größeren Abständen, je nachdem wie groß der verfügbare Durchsatz und die Netzwerkbandbreite ist.

Etwas anders sieht der Sachverhalt bei dem Invalidierungsprotokoll der SSK aus. Auch hier ist das Invalidierungsprotokoll ca. 6 mal schneller als das Aktualisierungsprotokoll, aber es ist ein leichter Anstieg der Ausführungszeit mit der Datenmenge zu verzeichnen. Allerdings wächst im selben Maße auch die Standardabweichung. Unabhängig von der Datengröße waren die schnellsten

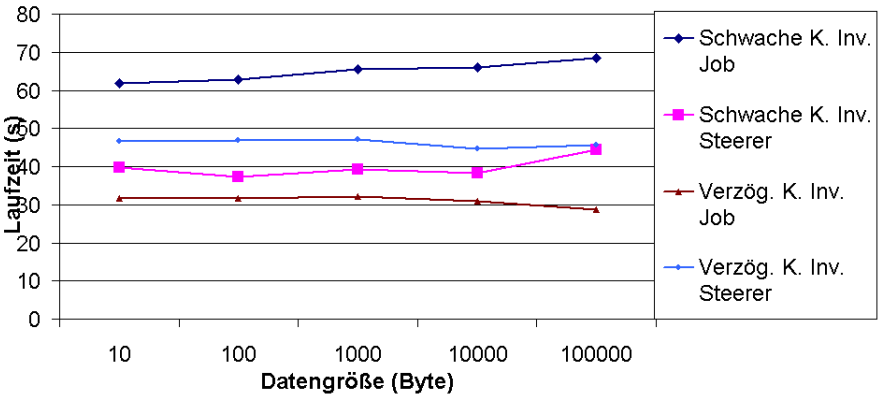


Abbildung 7.6: Laufzeit der Invalidierungsprotokolle in Abhängigkeit der Datengröße

Messungen ungefähr gleich. Durch den Synchronisationsalgorithmus hängt die Ausführungzeit der Protokolle der SSK von der Latenz ab. Im Allgemeinen verschlechtert sich die Latenz bei Zunahme der Netzauslastung. Als Resultat erhält man dadurch im Durchschnitt eine leicht steigende Laufzeit mit größer werdendem Fehler.

Analyse des Aktualisierungsprotokolls der PRAM Konsistenz

Bei den Aktualisierungsprotokollen bremst das Versenden der Datenmengen die Ausführung beider Prozesse. Bei der Verzögerten Schwachen Konsistenz braucht erstaunlicherweise das Steuerungswerkzeug, der auf dem schnelleren Rechner läuft, ca. 1.7 mal so lange wie der Job. Die lange Ausführungszeit des Steuerungswerkzeugs ist allein mit dem begrenzten verfügbaren Durchsatz nicht zu erklären. Beim Aktualisierungsprotokoll der PRAM-Konsistenz und der VSK benötigt der ca. Job 159 s weniger als das Steuerungswerkzeug.

Um die Ursache für das asymmetrische Verhalten zu finden, wurde ein ähnliches Experiment für die PRAM Konsistenz durchgeführt, bei dem aber nur einer der beiden Seiten Daten versendet hat. Hierfür wurde die PRAM Konsistenz gewählt, weil das Aktualisierungsprotokoll der PRAM Konsistenz für beide Seiten identisch ist. Bei das VSK verhalten sich die beiden Seiten unterschiedlich und lassen sich daher nicht direkt miteinder vergleichen. Bei der SSK wird die Laufzeit zu stark von dem Austausch des Synchronisationsobjekts dominiert. Das Aktualisierungsprotokoll der PRAM Konsistenz schickt bei jeder Schreiboperation die Daten direkt an den anderen Prozess. Das Ergebnis ist in Tab. 7.9 dargestellt. Die Laufzeiten werden mit $t_{uni,Seite}$ bezeichnet, wobei *uni* spezifiziert, dass es eine eindirektionale Messung ist und nicht beide Seiten senden. $Seite \in \{Steerer, Job\}$ definiert die Seite, die als einzige Daten schreibt

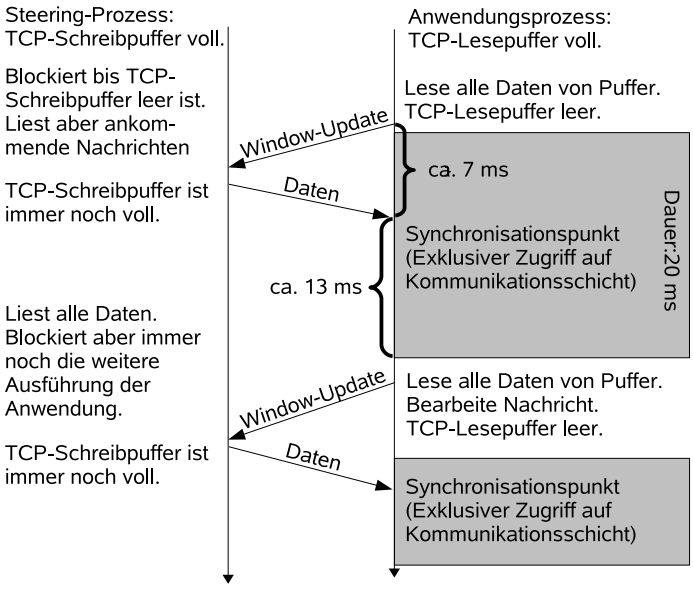


Abbildung 7.7: Ablauf des Mechanismus, der für die unterschiedlichen Laufzeiten je nach Transportrichtung beim Aktualisierungsprotokoll der PRAM-Konsistenz verantwortlich ist.

und für die die Laufzeit gemessen wurde.

Erstaunlicherweise dauert die Datenübertragung vom Steuerungswerkzeug zum Job doppelt so lange wie bei der umgekehrten Richtung. Es beginnt damit, dass der Rechner des Jobs langsamer ist als der Rechner des Steuerungswerkzeugs. Da das Steuerungswerkzeug die Daten schneller versenden kann als der Job, läuft der TCP-Empfangspuffer auf dem Rechner des Jobs voll (siehe auch Abb. 7.7). Die standardmäßige Größe des TCP-Empfangspuffers war dort 64 kB. Eine einzelne Aktualisierungsnachricht für ein fast 100 kB großes Datenobjekt kann damit nicht an einem Stück übermittelt werden. Der Sender schreibt also das erste Stück der ersten Nachricht und blockiert anschließend, weil der TCP-Empfangspuffer des Rechners des Jobs voll ist. Wenn der Job jetzt den TCP-Empfangspuffer ausliest, erhält er keine vollständige Nachricht. Da der Empfangspuffer wieder frei ist, versendet das TCP-Protokoll eine Window-Update Nachricht an den Sender, der daraufhin die nächsten Daten sendet. Die Zeit vom Versenden der Window-Update Nachricht bis die Daten am Job ankommen entspricht im wesentlichen der Round-Trip-Zeit, also ca. 7 ms (siehe Kap. 7.1.4).

Bevor allerdings die neuen Daten ankommen, stellt der Job fest, dass keine weiteren Daten empfangen wurden. Daher gibt der Kommunikationsthread die Kontrolle wieder ab. Der Anwendungsthread führt einen weiteren Synchronisationspunkt aus. Aus Synchronisationsgründen, darf ein Synchronisationspunkt nicht durch andere Speicheroperationen oder empfangene Nachrichten unterbrochen werden. Dies ist eine Voraussetzung für die Korrektheit der Konsistenzprotokolle. Eine Ausnahme hiervon ist, wenn zu Beginn des Synchronisationspunkts auf den Empfang bestimmter Nachrichten warten muss, was bei PRAM-

	Steerer	
	Zeit	Std.abw.
PRAM K., Job zum Steerer	75.03 s	0.59 s
PRAM K., Steerer zum Job	168.61 s	9.97 s
PRAM K., Steerer zum Job, Puffervergrößerung	98.18 s	22.37 s
PRAM K., St. z. J., Puffervergr., 1 Thread	81.02 s	1,70 s

Tabelle 7.9: Messungen zur Analyse des asymmetrischen Verhaltens der Aktualisierungsprotokolle. Alle Messungen wurden mit einer Datengröße von 100000 Byte durchgeführt. Es schrieb nur eine Seite Daten mit einer Wahrscheinlichkeit von 0.75.

Protokollen nie und bei VSK-Protokollen sehr selten vorkommt. Es können also keine eingehenden Nachrichten bearbeitet werden, solange der Synchronisationspunkt abgearbeitet wird und auch der TCP-Empfangspuffer wird in dieser Zeit nicht ausgelesen. Die Ausführung eines Synchronisationspunkts wurde mit 20 ms gemessen, wenn keine Schreiboperationen stattgefunden haben.

Wenn man die Differenz zwischen der Round-Trip-Zeit und der Ausführungszeit eines Synchronisationspunkts als Verzögerung berechnet, erhält man

$$20\,ms - 7\,ms = 13\,ms. \tag{7.3}$$

Wenn der Job pro empfangener Nachricht 2 Synchronisationspunkte ausführt, führt der Job 10 Synchronisationspunkte für jede Epoche des Steuerungswerkzeugs aus, in der das Steuerungswerkzeug die Daten geschrieben hat. Bei einer Schreibwahrscheinlichkeit von 75 % kann man ca. 750 solche Epochen erwarten. Dies ergibt 7500 zusätzliche Synchronisationspunkte des Jobs, was insgesamt zu einer Verzögerung t_V von

$$t_V = 750 \cdot 13\,ms = 97,5\,s \tag{7.4}$$

führen würde.

Vergrößert man den Empfangspuffer, wird die Laufzeit des Experimentes bei Datenübertragung vom Steuerungswerkzeug zum Job deutlich besser, da der Job dann mehr Daten zwischen zwei Synchronisationspunkten liest. Die besten Werte reichen an die Laufzeit bei umgekehrter Übertragungsrichtung heran, allerdings gibt es eine hohe Standardabweichung. Die Testanwendung führt zwischen zwei Synchronisationspunkten keine nennenswerte Berechnung durch. In den meisten Fällen bearbeitet lediglich der Kommunikationsthread eingegangene Daten. Da der Ablauf sowieso sequentiell ist, kann auch auf einen speziellen Kommunikationsthread verzichtet werden. Die Bearbeitung der Nachrichten wird dann auch vom Anwendungsthread zwischen zwei Synchronisationspunkten durchgeführt. Man erhält dann eine Laufzeit von 81,02 s mit nur geringer Standardabweichung.

Der Rechner des Steuerungswerkzeugs ist wesentlich schneller und hat standardmäßig eine Puffergröße von 128 kB. Dies führt zu kürzeren Ausführungszeiten des Synchronisationspunkts (ca. 12 ms) und dazu, dass eine ganze Nachricht in den Speicher passt. Nach der Bearbeitung einer Nachricht, muss außerdem keine Round-Trip-Zeit verstreichen bis die nächsten Daten im TCP-Puffer vorhanden sind, da der Puffer ja nicht voll war. Das Steuerungswerkzeug kann

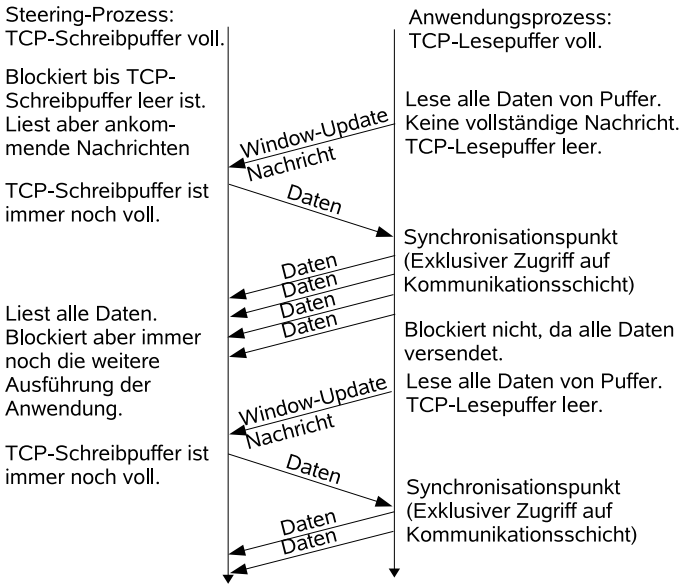


Abbildung 7.8: Ablauf des Mechanismus, der für die unterschiedlichen Laufzeiten der Prozesse beim Aktualisierungsprotokoll der VSK und PRAM-Konsistenz verantwortlich ist.

diese Nachricht schneller bearbeiten, als der Job sie senden kann. Nach Bearbeitung einer Nachricht liegen also schon Teile der nächsten Nachricht im TCP-Empfangspuffer, die wieder ausgelesen werden, ohne dass das Steuerungswerkzeug einen Synchronisationspunkt ausführt. Somit tritt dort dieses Problem kaum auf. Der Job kann mit maximaler Rate schreiben.

Wenn beim Steuerungswerkzeug keine Verzögerung durch den Job auftreten würde, hätte das Steuerungswerkzeug ungefähr dieselbe Laufzeit wie der Job. Wenn man jetzt die Verzögerung mit der Differenz

$$t_{uni,Steerer} - t_{uni,Job} = 93,58\,s \tag{7.5}$$

vergleicht, liegen beide sehr nahe beieinander. Wenn man für die Laufzeit des Steuerungswerkzeugs annimmt, dass sie sich aus der Laufzeit des Jobs und der berechneten Verzögerung zusammen setzt, erhält man:

$$t_{uni,Job} + t_V = 75,03\,s + 97,5\,s = 172,53\,s, \tag{7.6}$$

was innerhalb der Standardabweichung für $t_{uni,Steerer} = 168,61\,s \pm 9,97\,s$ liegt.

Damit lassen sich nun auch die Messungen erklären, wenn beide Seiten Daten senden. Der dafür verantwortliche Mechanismus ist in Abb. 7.8 dargestellt. Beim Job wird eine halbe Nachricht empfangen und dann ein Synchronisationspunkt ausgeführt. Alle in diesem Synchronisationspunkt versendeten Daten werden vom Anwendungsthread mit Sicherheitsinformationen versehen. Da das Einpacken der Nachrichten länger dauert als das Übertragen über das Netzwerk, sind am Ende des Synchronisationspunkts alle Nachrichten dieses Synchronisationspunkts versendet. Inzwischen ist der TCP-Lesepuffer voll. Wenn der Synchronisationspunkt beendet ist, wird wieder eine halbe Nachricht aus

	Steerer		Job	
	Zeit	Std.abw.	Zeit	Std.abw.
VSK ohne Flush	87.83 s	3.78 s	171.79 s	5.98 s
PRAM K. 256 kB Puffer	167.48 s	20.24 s	84.47 s	5.81 s
PRAM K. 1 MB Puffer	137.51 s	2.18 s	87.02 s	3.06 s
VSK 1 MB Puffer	198.24 s	23.78 s	173.82 s	5.72 s
Schwache K. 1 MB Puffer	321.54 s	14.69 s	332.94 s	14.77 s

Tabelle 7.10: Messungen zur Analyse des asymmetrischen Verhaltens der Aktualisierungsprotokolle. Alle Messungen wurden mit eine Datengröße von 100000 Byte und einer Lese- und Schreibwahrscheinlichkeit von 0.75 durchgeführt. Es haben jeweils beide Seiten gleichzeitig gesendet.

dem TCP-Lesepuffer ausgelesen, der dann leer ist. Danach wird wieder ein Synchronisationspunkt ausgeführt. Dies führt dazu, dass der Job nur wenige Synchronisationspunkte des Steuerungswerkzeugs bearbeitet hat, wenn der Job seine 1000 Iterationen beendet hat. Die gemessene Zeit für den Job ist also nur leicht höher als wenn nur der Job senden würde, da er kaum Zeit zum Auspacken von empfangenen Nachrichten verwendet hat.

Beim Steuerungswerkzeug dagegen spielt die `flush`-Operation nach jedem Synchronisationspunkt eine Rolle. Dieser Aufruf blockiert den Anwendungsthread bis alle Daten in den TCP-Schreibpuffer geschrieben werden konnten. Da der Job viel langsamer liest, läuft dieser nach wenigen Ereignissen voll. Beim Steuerungswerkzeug blockiert der Anwendungsthread daher die meiste Zeit, da er darauf wartet, dass der Empfänger die nächsten 64 kB liest. Während der `flush`-Operation kann der Kommunikationsthread ungehindert lesen. Daher hat das Steuerungswerkzeug ausreichend Zeit, um alle eingehenden Nachrichten zu lesen. Als Folge hat das Steuerungswerkzeug kaum eigene Synchronisationspunkte bearbeitet, wenn der Job seine 1000 Iterationen beendet hat. Anschließend sendet dann nur noch das Steuerungswerkzeug Daten. Insgesamt muss also beim Steuerungswerkzeug eine Zeit gemessen werden, die der Summe aus den Zeiten in beiden Transportrichtungen entspricht, wenn jeweils nur eine Seite sendet (siehe Tab. 7.9).

$$t_{PRAM,Steerer} \approx t_{uni,Steerer} + t_{uni,Job} \tag{7.7}$$

Gemessen wurde:

$$249,84\,s \approx 168,61\,s + 75,03\,s = 243,64\,s \tag{7.8}$$

Die Standardabweichung beträgt 17,96 s, somit ist das im Rahmen der Messgenauigkeit.

Vergrößert man die TCP-Empfangspuffer, sinkt die Laufzeit des Steuerungswerkzeugs, da der Job nun größere Datenblöcke zwischen eigenen Synchronisationspunkten liest. Trotzdem liest er weiterhin weniger Daten als er sendet, da der Kommunikationsthread zu häufig unterbrochen wird.

Ein wichtiger Faktor für diesen Effekt ist, dass die Anwendung als auch das Steuerungswerkzeug praktisch nichts anderes tun, als Synchronisationspunkte in einer Schleife auszuführen. Dadurch sind die Abschnitte zwischen zwei Synchronisationspunkten sehr kurz. Bei einer realen Anwendung ist mit großer Wahrscheinlichkeit der Abstand zwischen zwei Synchronisationspunkten größer, da

die Anwendung die neuen Daten ja erstmal berechnen muss. Außerdem wird in der Praxis das Steuerungswerkzeug in der Regel deutlich weniger Schreiboperationen ausführen als die Anwendung. Aus diesen Gründen wird bei realen Anwendungen das Steuerungswerkzeug diese kritische Menge an Schreiboperationen gar nicht erreichen. In diesem Experiment wurde die maximale Belastung des Protokolls simuliert.

Auswertung des Aktualisierungsprotokolle der Verzögerten Schwachen Konsistenz

Das Aktualisierungsprotokoll der VSK ist wesentlich aufwendiger, vor allem muss der Wert geänderter Objekte mehrfach kopiert werden. Beim Steuerungswerkzeug werden für eine Schreiboperation folgende Kopieroperationen durchgeführt:

- Nach der Schreiboperation wird der neue Wert in einen Zwischenpuffer kopiert.
- Danach wird der alte Wert aus einem Puffer wieder in die Variable kopiert.
- Die Aktualisierung wird vom Job empfangen und eine neue Aktualisierung zurückgesendet. Nach Empfang der Rückantwort, muss das Steuerungswerkzeug den Wert anwenden und somit wieder kopieren.

Der Rechenaufwand des Aktualisierungsprotokoll der VSK ist also größer als bei der PRAM Konsistenz, was zu einer längeren Laufzeit des Experimentes führt. Zwischen der Laufzeit des Jobs und des Steuerungswerkzeugs besteht wieder eine Differenz von ca. 160 s (siehe Tab. 7.8), genauso wie bei der PRAM Konsistenz. Der Grund ist hierbei derselbe wie beim Aktualisierungsprotokoll der PRAM Konsistenz. Bei beiden Protokollen wird fast dieselbe Datenmenge versendet. Bei der VSK gibt es zusätzlich nur die `SyncEnd`-Nachrichten, die im Verhältnis zu der Gesamtdatenmenge sehr klein ist. Daher ist die Verzögerung des Steuerungswerkzeugs im Wesentlichen gleich. Auch hier kann man beobachten, dass bei einer Vergrößerung des TCP-Empfangspuffers, sich die Ausführungszeiten beider Prozesse angleichen (siehe Tab. 7.10).

Auswertung des Aktualisierungsprotokolls der Speziellen Schwachen Konsistenz

Bei der Speziellen Schwachen Konsistenz findet keine Verzögerung durch unnötig ausgeführte Synchronisationspunkte statt, da die Synchronisationspunkte erst ausgeführt werden können, wenn der Prozess das Synchronisationsobjekt besitzt. Das Synchronisationsobjekt wird nach Beendigung des Synchronisationspunkts verschickt. Da die Daten in derselben Reihenfolge empfangen werden wie sie verschickt werden, müssen erst alle Daten angekommen sein, bevor das Synchronisationsobjekt empfangen wird. Sonst blockiert der Synchronisationspunkt und liest solange eingehende Daten bis er das Synchronisationsobjekt erhalten hat.

Bei der Speziellen Schwachen Konsistenz sendet immer höchstens ein Prozess, nie beide gleichzeitig (Es wird nur in Synchronisationspunkten gesendet). Wenn ein Prozess das Synchronisationsobjekt am Ende eines eigenen Synchronisationspunkts an den anderen Prozess abgibt, wird kurz darauf der nächste

	Steerer		Job	
	Zeit	Std.abw.	Zeit	Std.abw.
SSK AktP.	608,41 s	46.42 s	608.34 s	46.80 s
SSK InvP.	187.60 s	9.24 s	187.66 s	9.47 s
VSK AktP.	179.28 s	16.24 s	59.33 s	1.20 s
VSK InvP.	38.66 s	2.60 s	20.81 s	1.42 s
PRAM K. AktP.	129.35 s	5.70 s	31.82 s	0.97 s

Tabelle 7.11: Messungen zur Analyse des Verhaltens der Protokolle zwischen den Gridsites Freiburg und Siegen. Alle Messungen wurden mit eine Datengröße von 100000 Byte und einer Lese- und Schreibwahrscheinlichkeit von 0.75 durchgeführt.

Synchronisationspunkt begonnen und das Synchronisationsobjekt wieder angefordert. Dadurch wird quasi direkt nach dem Versand des Synchronisationsobjekts die Anforderung für das Synchronisationsobjekt versendet. Somit liegt am Ende eines Synchronisationsobjekts immer eine Anforderung vor und das Synchronisationsobjekt wechselt nach jedem Synchronisationspunkt den Besitzer. Folglich werden Synchronisationspunkte immer abwechselnd von beiden Prozessen ausgeführt. Dadurch haben beide Prozesse auch fast dieselbe Laufzeit.

7.2.3 Große Datenmengen im Grid

Nachdem das Verhalten der Protokolle in einer lokalen Umgebung einige interessante Effekte offenbart hat, stellt sich die Frage, inwiefern diese Ergebnisse in eine reale Gridumgebung übertragbar sind. Daher wurde das Experiment mit den 100000 Byte großen Datenobjekten im Grid widerholt. Dabei lief das Steuerungswerkzeug auf einem Rechner der Gridsite in Freiburg und der Job auf einem Rechner der Gridsite in Siegen. Die Messergebnisse sind in Tabelle 7.11 gezeigt.

Erstens ist festzustellen, dass die Protokolle der SSK deutlich länger brauchen als in der lokalen Umgebung (siehe Tab. 7.8). Bei den Protokollen der SSK liefert der Austausch des Synchronisationsobjekts den wesentlichen Beitrag zu der Laufzeit des Protokolls. Dieser Austausch hängt im wesentlichen von der Round-Trip-Zeit ab, die im Grid deutlich höher ist (siehe Kap. 7.1.4). Als Folge sind auch die Protokolle der SSK im Grid deutlich langsamer.

Die Protokolle der VSK und der PRAM Konsistenz sind dagegen in diesem Experiment schneller, obwohl die Bandbreite ca. ein Drittel geringer ist als im lokalen Experiment (siehe Kap. 7.1.3). Dies ist ein weiterer Beleg dafür, dass nicht die Netzwerkbandbreite die Performance bestimmt, sondern die Dauer der Berechnung der Nachrichten. Da schnellere Rechner beteiligt waren, sank die Berechnungsdauer und somit die Zeit für die Ausführung der Tests.

Bei den Aktualisierungsprotokollen sind wieder signifikante Differenzen zwischen den Laufzeiten des Jobs und des Steuerungswerkzeugs zu beobachten. Das ist auf die in Kap. 7.2.2 beschriebenen Effekte zurückzuführen.

		Hybrides Protokoll		Invalidierungsprotokoll	
	Wahrscheinl.	Zeit	Std.abw.	Zeit	Std.abw.
Steerer	0.01	25.95 s	1.60 s	23.67 s	2.06 s
Job	0.99	20.76 s	0.20 s	21.71 s	0.55 s
Steerer	0.05	43.00 s	3.01 s	35.06 s	1.77 s
Job	0.95	23.55 s	0.59 s	27.72 s	1.74 s
Steerer	0.10	60.33 s	4.68 s	41,31 s	1.73 s
Job	0.90	24.14 s	0.89 s	31,65 s	1.86 s

Tabelle 7.12: Messungen zur Analyse des asymmetrischen Verhaltens der hybriden Protokolle. Alle Messungen wurden mit eine Datengröße von 100000 Byte und der VSK durchgeführt. Das Steuerungswerkzeug (Steerer) verwendet das Invalidierungsprotokoll, der Job das Aktualisierungsprotokoll. Als Vergleich wurden Messwerte angegeben, wenn beide Seiten das Invalidierungsprotokoll verwenden. Die angegebenen Wahrscheinlichkeiten beziehen sich auf Lese- und Schreibwahrscheinlichkeiten.

		Hybrides Protokoll		Invalidierungsprotokoll	
	Wahrscheinl.	Zeit	Std.abw.	Zeit	Std.abw.
Steerer	0.01	118.70 s	2.39 s	51.94 s	1.58 s
Job	0.99	90.47 s	2.44 s	70.72 s	1.71 s

Tabelle 7.13: Messungen zur Analyse des asymmetrischen Verhaltens der Aktualisierungsprotokolle. Alle Messungen wurden mit eine Datengröße von 100000 Byte und der SSK durchgeführt. Das Steuerungswerkzeug (Steerer) verwendet das Invalidierungsprotokoll, der Job das Aktualisierungsprotokoll. Als Vergleich wurden Messwerte angegeben, wenn beide Seiten das Invalidierungsprotokoll verwenden. Die angegebenen Wahrscheinlichkeiten beziehen sich auf Lese- und Schreibwahrscheinlichkeiten.

7.2.4 Hybride Protokolle

Bei einer typischen Steering Situation wird der Gridjob viel häufiger schreiben und lesen als das Steuerungswerkzeug. Das Steuerungswerkzeug wird selten einen Wert ändern. Die Frage ist, ob sich bei einer solch asymmetrischen Lastverteilung die Balance durch ein kombiniertes Protokoll verbessern ließe, bei dem das Steuerungswerkzeug ein Aktualisierungsprotokoll und der Job ein Invalidierungsprotokoll verwendet. Dadurch würde das Maximum der Laufzeit beider Seiten sinken und somit die Gesamtlaufzeit beschleunigt.

Um dies zu testen, wurde die Lese- und Schreibwahrscheinlichkeit auf der Seite des Steuerungswerkzeugs und des Jobs variiert. Verwendet wurden immer 100000 Byte große Datenobjekte und derselbe Versuchsaufbau wie in Kap. 7.2.2.

Vergleichmessungen (siehe Tab. 7.12 und Tab 7.13) zeigen allerdings, dass hybride Protokolle für die getesteten Fälle nicht performanter sind als das Invalidierungsprotokoll (wenn die maximale Laufzeit aus beiden Prozessen verglichen wird). Das schließt natürlich nicht aus, dass Fälle existieren, in denen ein hybrides Protokoll eine kürzere Laufzeit besitzt, die aber nicht gefunden wurden. Insgesamt sind die Optimierungsmöglichkeiten durch hybride Protokolle eingeschränkt, da das Invalidierungsprotokoll, unabhängig von der Datenmenge, eine nahezu konstante Performance bietet.

Wird bei der SSK vom Steuerungswerkzeug das Aktualisierungsprotokoll verwendet, führt dies zu einem Wechseln des Synchronisationsobjektes nach jedem Synchronisationspunkt des Steuerungswerkzeugs, da der Job nie ein invalidiertes Datenobjekt besitzt, daher nie in der Leseoperation blockiert und folglich dem Steuerungswerkzeug keine Folge von Synchronisationspunkten ermöglicht. Daher sind für die SSK auch keine Verbesserungungen gegenüber dem Invalidierungsprotokoll durch hybride Protokolle möglich.

Bei der VSK wird durch ein hybrides Protokoll tatsächlich Rechenzeit vom Job zum Steuerungswerkzeug verlagert. Dies schlägt sich aber nicht in einer Reduktion der Gesamtlaufzeit wieder, da das Steuerungswerkzeug ohnehin der Prozess war, der eine längere Ausführungszeit hatte. Gilt es allerdings den Job zu entlasten, kann durch ein hybrides Protokoll durchaus Rechenzeit für das Steering verstärkt auf das Steuerungswerkzeug konzentriert werden. Falls die Auslastung der CPU des Jobs deutlich höher ist als die des Steuerungswerkzeugs (was bei dem Test nicht der Fall war), könnte evtl. auch eine Verkürzung der Gesamtlaufzeit erzielt werden.

7.2.5 Schlussfolgerung

Der erste Schluss, der aus den vorangegangenen Experimenten gezogen werden kann, ist, dass die VSK praktisch immer performanter als sie Schwache Konsistenz ist. Der Grund liegt darin, dass auf eine sequentielle Ordnung der Synchronisationspunkte verzichtet werden kann. Steigt die Round-Trip-Zeit, verschlechtert sich die Performance der SSK-Protokolle, während die Protokolle der VSK davon deutlich weniger beeinflusst werden.

Zweitens, das Invalidierungsprotokoll der VSK passt die Übertragungsfrequenz der Aktualisierungen effektiv an die verfügbare Netzwerkbandbreite an. Dies geschieht für den Benutzer transparent, es werden keine zusätzlichen Hinweise im Quellcode benötigt, um die Übertragung zu optimieren. Nur bei sehr kleinen Datenmengen kann ein Aktualisierungsprotokoll bei der VSK performanter sein.

Bei einer verfügbaren Netzwerkbandbreite von 100 Mb/s kann bei langsameren Rechnern, die Rechenleistung die Performance stark beeinflussen. Entgegen den Erwartungen, ist es im LCG nicht die verfügbaren Netzwerkbandbreite, die den Durchsatz bestimmt, sondern die Rechenleistung.

Insbesondere bei unterschiedlich schnellen Rechnern führt ein zu kleiner TCP-Empfangspuffer dazu, dass bei einer hohen Frequenz an Synchronisationspunkten und großen Datenmengen der langsamere Rechner ein Vielfaches an Synchronisationspunkten ausführen kann wie der schnellere Rechner. Dadurch liest der langsamere Rechner noch weniger, wodurch die TCP-Puffer voll laufen. Der schnellere Rechner blockiert beim Schreiben und erleidet so deutliche Verzögerungen. Es sollte daher darauf geachtet werden, dass die Empfangspuffer mindestens die Größe des größten Datenobjektes besitzen. Besser ist es, wenn der Empfangspuffer größer ist als die Daten, die in einer Epoche versendet werden. Sind die Datenobjekt zu groß, sollte man in Betracht ziehen das Datenobjekt als Stream zu registrieren und somit in Blöcken zu behandeln.

Der Nutzen von hybriden Protokollen konnte nicht validiert werden.

	Anzahl im Steerer		Anzahl im Job	
Bandbreite	Akt.P.	Inv.P.	Akt.P.	Inv.P.
100 kB/s	0	5	0	5
250 kB/s	0	5	0	5
500 kB/s	2	3	1	4
750 kB/s	3	2	3	2
1 MB/s	4	1	4	1
2 MB/s	5	0	5	0

Tabelle 7.14: Anzahl der Datenobjekte mit einer Größe von 20 kB, für die die Optimierung bei vorgegebener Bandbreite das Invalidierungsprotokoll und der Aktualisierungsprotokoll gewählt hat.

7.3 Evaluation der Optimierung

Eine der Annahmen für die Entwicklung der Optimierung in Kap. 4.5 war, dass die Netzwerkbandbreite die Kommunikation begrenzt. Da sich bei der Evaluation der Protokolle herausgestellt hat, dass die Rechenleitung und nicht die Netzwerkbandbreite das beschränkende Element ist, hat sich diese Annahme nicht erfüllt. Folglich führt eine Anpassung des benötigten Durchsatzes an die verfügbare Badbreite nicht zwangsläufig zu einer besseren Performance.

Die benötigte Rechenleistung steigt linear mit dem zu versendenden Datenvolumen an. Es gibt also keine Stufe, bis zu dem der Durchsatz kostenfrei ist, wie es in Kap. 4.5 angenommen wurde. Eine Verbesserung der Laufzeit des Programms würde also ausschließlich durch eine Reduzierung des zu versendenden Datenvolumens erreicht.

Trotzdem kann getestet werden, welche Protokolle die Optimierung bei einer verfügbaren Bandbreite auswählt. Dafür wurde der Parameter für die Bandbreite in der Berechnung fest definiert und jeweils nur die Round-Trip-Zeit gemessen, die aber im wesentlichen konstant war. Die Größe der Datenobjekte betrug 20 kB. Es wurden 1000 Iterationen durchgeführt und ausgegeben, wieviele Datenobjekte ein Invalidierungsprotokoll und wieviele ein Aktualisierungsprotokoll verwendeten. Für das Experiment wurden die Protokolle der VSK verwendet. Das Ergebnis ist in Tab. 7.14 dargestellt.

Offensichtlich werden bei geringerer Bandbreite mehr Invalidierungsprotokolle verwendet und bei größerer Bandbreite mehr Aktualisierungsprotokolle. Wie es bei der Bandbreite von 750 kB zu sehen ist, müssen die Entscheidungen nicht immer auf beiden Seiten gleich sein, sondern können in Grenzfällen bei einem Datenobjekt unterscheiden.

Bei den obigen Messungen wurde nicht die physische Netzwerkbandbreite reduziert, sondern nur der entsprechende Parameter in der Berechnung gesetzt. Um die Auswirkungen eines vollen Netzwerkes realitätsnah zu testen, wurden diese Messungen wiederholt, dabei wurde versucht die Verbindung zwischen beiden Rechnern durch TCP-Kommunikation auszulasten. Allerdings hat sich das weder auf die Laufzeit noch auf die Protokollwahl in sichtbarer Weise niedergeschlagen.

	Zeit	Std.abw.
Herunterladen, 1. Zugriff $t_n(DL)$	9.84 s	3.99 s
Herunterladen, Histogramm $t_a(DL)$	10.09 s	4.37 s
Blockweise, 1. Zugriff $t_n(BA)$	2.13 s	0.93 s
Blockweise, Histogramm $t_a(BA)$	13.49 s	6.25 s

Tabelle 7.15: Zugriffzeiten auf eine ROOT Datei mittels blockweisem Zugriff auf Teile der Datei im Vergleich zum Herunterladen der gesamten Datei.

7.4 Zugriffe auf entfernte Dateien in realer Anwendung

Die wesentlichen Ergebnisse aus Gridjobs des ATLAS Experiments werden in sog. ROOT-Dateien abgelegt. Um während der Laufzeit auf Zwischenergebnisse aus ROOT-Dateien zuzugreifen, wurde die Dateizugriffsbibliothek entwickelt. In diesem Unterkapitel wird ein Experiment beschrieben, bei dem eine ROOT Anwendung auf eine entfernte Datei über das Grid zugreift.

Versuchsaufbau

Als Job wurde ein gestoppter Grid Job verwendet, der simulierte Ereignisdaten (Monte-Carlo Daten) erzeugte und auf einem Knoten der Site UNI-SIEGEN-HEP lief. Dieser Job wurde immer nach 100 Ereignissen angehalten und erhielt immer dieselben Eingaben, so dass die Ergebnisdateien stets identisch waren. Die Ergebnisdatei hatte eine Größe von ca. 2.5 MB.

Auf die Zwischenergebnisse wurde einem UI am CERN (lxplus.cern.ch) über das Internet zugegriffen. Zum Zugriff wurde die Kommandozeilenkomponente von RMOST für ROOT verwendet. In einem ROOT-Skript wurde die Ergebnisdatei geöffnet und zuerst die Anzahl der Ereignisse aus der Datei ausgelesen. Anschließend iterierte das Skript über alle Einträge der Datei und erstellte ein Histogramm des maximalen transversalen Impulses jedes Ereignisses. Dies bedeutete, dass die komplette Datei gelesen werden musste. Zum Zugriff auf die Datei wurden jeweils zwei unterschiedliche Protokolle eingesetzt: Zuerst ein Protokoll, welches zuerst die gesamte Datei herunter lädt und dann lokal darauf zugreift. Als Zweites wurde ein Protokoll verwendet, welches blockweise auf die Datei zugreift und nur die angefragten Blöcke transferiert. Die Zeitmessung wurde gestartet, direkt bevor die Datei geöffnet wurde. Gemessen wurde die Zeit bis die Anzahl der Einträge $t_n(V)$ gelesen war, sowie bis die Auswertung $t_a(V)$ beendet war. Dabei ist $V \in \{DL, BA\}$ das Verfahren mit dem auf die Datei zugegriffen wurde. Der Download wird mit DL bezeichnet, der blockweise Zugriff mit BA. Die Messergebnisse sind in Tab. 7.15 aufgelistet.

Auswertung

Die ROOT-Datei mit Zwischenergebnissen einer laufenden Anwendung kann erst geöffnet werden, wenn Athena seinen Cache in die Datei geschrieben hat. Beim Öffnen muss Athena also dazu gebracht werden, die ROOT-Datei korrekt zu schreiben. Anschließend wird eine Kopie davon angelegt, auf die zugegriffen wird. Das Steuerungswerkzeug erhält eine entfernte Referenz zu der geöffneten

	Zeit
Herunterladen, 1. Zugriff $t_n(DL)$	3.05 s
Herunterladen, Histogramm $t_a(DL)$	3.27 s
Blockweise, 1. Zugriff $t_n(BA)$	0.88 s
Blockweise, Histogramm $t_a(BA)$	4.42 s

Tabelle 7.16: Niedrigste Messwerte für den Zugriff auf eine entfernte ROOT-Datei.

Kopie. Das Öffnen ist also ein komplexer Vorgang und benötigt im Durchschnitt etwa 2 s. In diesen 2 s sind die Zeiten für die Kommunikation, dem Schreiben der ROOT-Datei durch Athena, das Erstellen der Kopie im Job und das Anlegen der Strukturen im Steuerungswerkzeug enthalten. Dieser Prozess kann unter Umständen deutlich schneller ausgeführt werden, wenn z.B. die Kommunikation schneller ist.

Bei nachfolgenden Zugriffen erfolgt der Zugriff natürlich wesentlich schneller. Einmal transferierte Blöcke werden lokal gespeichert und brauchen danach nicht wieder angefordert werden. Werden alle Daten Stück für Stück gelesen, werden im Durchschnitt ca. 13.5 s benötigt.

Wird die ganze Datei heruntergeladen, werden im Durchschnitt schon fast 10 s für den ersten Zugriff benötigt. Allerdings kann danach sehr schnell auf die lokale Kopie zugegriffen werden.

Um zu berechnen ab wann es sich lohnt den Download zu verwenden anstatt den direkten Zugriff, wird davon ausgegangen, dass die Zeit zum Öffnen der Datei bei beiden Verfahren 2 s beträgt. Wenn man annimmt, dass die restliche Zeit zur Datenübertragung benötigt wird, kann man den Anteil f der Daten abschätzen, der gelesen werden muss, bis beide Verfahren dieselbe Zeit brauchen:

$$f = \frac{t_a(DL) - 2\,s}{t_a(BA) - 2\,s} \approx 0.7 \tag{7.9}$$

Solange also weniger als 70 % der Daten gelesen werden, ist der blockweise Zugriff schneller. Für komplexe Analysen auf den gesamten Daten ist der blockweise Zugriff ungünstiger.

Diese Berechnungen gehen vom Durchschnitt des gemessenen Wertes aus. Die Standardabweichung (Std.abw.) der Messwerte beträgt fast 50 % des Messwertes. Insgesamt schwankt die verfügbare Bandbreite stark, was sich auch auf die Messwerte auswirkt. Die niedrigsten Messwerte in jeder Kategorie betragen weniger als ein Drittel des Durchschnitts (siehe Tabelle 7.16). Diese Werte lassen sich aber nicht in eine Beziehung zueinander setzen und gehören zu unterschiedlichen Messungen, die zeitlich auseinander liegen.

7.5 Fazit

In diesem Kapitel wurden Experimente und deren Ergebnisse beschrieben, mit denen die Kommunikationschicht, die Datenkonsistenzschicht, die Optimierung und der Datenzugriff untersucht wurden.

Bei der Kommunikationschicht wurde der erzielte Durchsatz und die Round-Trip-Zeit mit unterschiedlichen Szenarien und Sicherheitseinstellungen in ver-

schiedenen Umgebungen ermittelt. Das Verhalten der Kommunikationsschicht für den Durchsatz kann mit einer Pipeline modelliert werden. Bei der Verwendung von Verschlüsselung oder zwei Sicherheitsschichten, kann die Rechenzeit für die Sicherheitstests zu dem limitierenden Faktor werden. Dies setzt aber voraus, dass die Netzwerkbandbreite groß genug ist. Allerdings ist die Rechenzeit für empfangene Nachrichten und gesendete Nachrichten nicht nebenläufig, während bei voll-duplex Netzwerken die volle Bandbreite in beiden Richtungen gleichzeitig zur Verfügung steht. Der höchste Durchsatz wurde bei einer Nachrichtengröße von 8 kB erreicht.

Die Round-Trip-Zeit steigt mit der Verwendung jeder Sicherheitsschicht, Verschlüsselung oder eines Verbindungsdienstes jeweils um einen konstanten Wert. Allerdings sind in einer reellen Gridumgebung diese Werte, gegenüber der Transferzeit über das Internet, vernachlässigbar.

Bei der Datenkonsistenzschicht wurde die Laufzeit beider Prozesse bei der Verwendung verschiedener Protokolle gemessen. Dabei stellte sich heraus, dass die Protokolle der VSK schneller sind als die Protokolle der SSK. Dies liegt im wesentlichen in der strengen Ordnung der Synchronisationspunkte der SSK begründet. Mit steigender Round-Trip-Zeit vergrößert sich der Unterschied noch mehr, da die Protokolle der SSK deutlich stärker von der Round-Trip-Zeit beeinflusst werden als die Protokolle der VSK.

Insgesamt hat sich herausgestellt, dass die Rechenzeit zum Ein- und Auspacken für die Laufzeit des Steering-Experimentes verantwortlich ist und nicht die Limitierung des Durchsatzes durch die Netzwerkbandbreite.

Des Weiteren war die Diskrepanz zwischen der Laufzeit des Steering-Prozesses und des Anwendungsprozesses bei den Aktualisierungsprotokollen der VSK und PRAM-Konsistenz auffällig. Der dafür verantwortliche Mechanismus wurde beschrieben.

Da die Netzwerkbandbreite nicht den Datenaustausch beim Steering limitiert und dadurch Verzögerungen in der Programmausführung verursacht, war die Grundannahme der Optimierung nicht gegeben. Eine effektive Optimierung müsste die Rechenzeit so weit wie möglich reduzieren. Eine praxisnahe Optimierung muss auf der Evaluation der Protokolle aufbauen, die erst nach der Analyse und Implementierung vorhanden war und somit beim Design der Optimierung noch nicht zur Verfügung stand.

Von der Zugriffsschicht wurde der transparente, blockweise Dateizugriff in einer realen Anwendung getestet. Ein Vergleich mit einem Download der kompletten Datei ergab, dass der Download sich lohnt, wenn mindestens 70 % der Datei vom Steuerungswerkzeug gelesen wird. Somit ist der blockweise Zugriff beim Online-Steering sinnvoll, wenn nur ein Teil der Gesamtdaten eines Berechnungsschrittes für die Visualisierung von Zwischenergebnissen gelesen werden muss.

Kapitel 8

Zusammenfassung und Ausblick

Diese Arbeit hat sich mit dem Online-Steering von Gridjobs beschäftigt. Das Online-Steering wurde in vier Teilaufgaben zerlegt: Die Kommunikation, die Datenkonsistenz, automatisierte Auswertungen und Optimierungen und der Datenzugriff.

Der Schwerpunkt dieser Arbeit liegt in der Teilaufgabe, die sich mit der Datenkonsistenz beschäftigt. Hierfür wurde ein neues Konzept vorgestellt, das Online-Steering als Zugriffe auf DSM betrachtet. Es wurden die Intra-Prozess und die Inter-Prozess Bedingung identifiziert, die Regeln für den Datenaustausch definieren, die die Integrität der Daten sichern sollen. Je nach Anwendung müssen dafür beide Bedingungen erfüllt werden oder es reicht die Erfüllung der Intra-Prozess Bedingung.

Ausgehend von den Integritätsbedingungen wurden die Spezielle Schwache Konsistenz und die Verzögerte Schwache Konsistenz definiert, die die Intra-Prozess Bedingung erfüllen. Ferner wurde, basierend auf einem CUMULVS-Algorithmus [133], die Zeitplan-Konsistenz definiert, die beide Bedingungen erfüllt. Zusätzlich kann es Daten geben, für die keine der beiden Bedingungen erfüllt sein muss. Diese Daten können mit der bereits existierenden PRAM-Konsistenz [108] bearbeitet werden.

Bei der Speziellen Schwachen Konsistenz und der Verzögerten Schwachen Konsistenz müssen jeweils nur ein Steering-Prozess und ein Anwendungsprozess miteinander kommunizieren. Für beide Konsistenzmodelle wurde je ein Aktualisierungsprotokoll und ein Invalidierungsprotokoll entworfen und implementiert. Zusätzlich wurden auch ein Aktualisierungsprotokoll und ein Invalidierungsprotokoll für die PRAM-Konsistenz implementiert. Anhand dieser Implementierungen wurde das Laufzeitverhalten der Protokolle untersucht.

Bei der Evaluation wurde festgestellt, dass die Protokolle der SSK, aufgrund der strengen Ordnung der Synchronisationspunkte, langsamer als die Protokolle der VSK sind. Dieser Unterschied verstärkt sich, wenn die Round-Trip-Zeit wächst. Bei der VSK und PRAM-Konsistenz traten deutliche Unterschiede bei der Laufzeit zwischen Steering-Prozess und Anwendungsprozess auf.

Um Online-Steering mit Gridjobs zu ermöglichen, muss mit dem Gridjob kommuniziert werden können. Es wurde in dieser Arbeit eine Kommunikations-

schicht entwickelt, die eine sichere Kommunikation zum Gridjob aufbauen kann. Dabei sollte das Sicherheitskonzept der Site aufrecht erhalten werden können. Es wurden daher mehrere Szenarien entwickelt, die je nach Konfiguration der Site zum Einsatz kommen. Das zu verwendende Szenario kann dabei dynamisch ermittelt werden.

Die Implementierung dieser Schicht enthält das offene, halboffene und geschlossene Szenario. Der Durchsatz und die Round-Trip-Zeit der Kommunikationsschicht bei verschiedenen Szenarien und Sicherheitsstufen wurde evaluiert. Bei der Bestimmung des Durchsatzes kann die Kommunikation mit einer Pipeline modelliert werden. Der Durchsatz wird von der Pipelinestufe mit dem geringsten Durchsatz bestimmt. Bei Netzwerken, wie sie zwischen den deutschen LCG-Sites bestehen, führt die Verwendung von Verschlüsselung zu einer Limitierung des Durchsatzes durch die Rechenzeit auf den sendenden und empfangenden Rechnern. Bei der Verwendung einer zweiten Sicherheitsschicht ohne Verschlüsselung, war bei eindirektionalem Datentransfer die Bandbreite limitierend. Wenn in beide Richtungen gleichzeitig gesendet wird, wird die Rechenzeit an den Endpunkten kritisch, da bei voll-duplex Netzwerken die Übermittlung von Daten mit maximaler Bandbreite in beiden Richtungen gleichzeitig erfolgen kann, während die Rechenzeit zum Ein- und Auspacken der Nachrichten sich addiert.

Zu der Round-Trip-Zeit fügt jede Komponente einen additiven Bestandteil bei. In Anbetracht der Round-Trip-Zeit zwischen zwei Sites sind die zusätzlichen Zeiten für einen Verbindungsdienst und die Sicherheitsschichten vernachlässigbar.

Die Optimierung und der Datenzugriff wurden in wesentlich geringerem Umfang behandelt als die Datenkonsistenz und die Kommunikation. Die Optimierungsidee basierte auf der Annahme, dass die Netzwerkbandbreite den Durchsatz limitiert. Bei der Evaluation der Protokolle wurde festgestellt, dass in etlichen Fällen der Rechenaufwand zu einer Limitierung des Durchsatzes führt. Für eine effektive Optimierung müsste dies miteinbezogen werden.

Bei dem Datenzugriff ging es darum, Datenzugriffe der Anwendung abzufangen und Methoden des Steering-Systems aufzurufen, so dass das Steering-System auf die Datenzugriffe reagieren kann. Hierbei wurden zunächst einige Klassifizierungen vorgenommen, die sich darauf auswirken, wie das Steering-System auf diese Datenzugriffe reagieren kann oder soll. So wurden aktive und passive Zugriffe unterschieden und die Granularität untersucht. Ob Datenzugriffe passiv oder aktiv sind, wirkt sich auf die Möglichkeiten aus, mit denen das Steering-System auf Zugriffe reagieren kann. Aktive Zugriffe erlauben eine sofortige Reaktion auf die Zugriffe, während passive Zugriffe durch Abtastung rekonstruiert werden müssen. Dadurch kann das Steering-System nicht sofort auf Zugriffe reagieren, sondern nur an den Punkten, an denen die Abtastung durch geführt wird. In der Regel sind dies die Synchronisationspunkte. Bei der Abtastung wurde die insensitive und die wert-sensitive Abtastung eingeführt. Danach wurden technische Möglichkeiten aufgezählt, wie Datenzugriffe abgefangen werden können. Implementiert wurde eine Bibliothek, die Dateizugriffe umlenkt.

Mit der Implementierung in RMOST können Jobs des ATLAS Experiments gesteuert werden. Es existiert also eine Integration in eine reale Anwendung.

Da es sich diese Arbeit in weiten Bereichen auf sequentielle Jobs beschränkt, bleibt die Frage offen, wie entsprechende Konsistenzmodelle, Protokolle und Im-

plementierungen bei kollaborativem Steering aussehen. DSM-Systeme skalieren nur beschränkt. Jenseits einer bestimmten Anzahl Prozesse verhindert die Synchronisation des DSM eine Beschleunigung durch eine Vergrößerung der Anzahl der parallelen Prozesse. Da der DSM allerdings nicht zur Synchronisation der Prozesse untereinander dient und vielleicht nicht-blockierende Protokolle existieren, ist vielleicht eine bessere Skalierung als bei bekannten DSM-Systemen möglich. Es könnte dann auch ein Vergleich der Performance der Zeitplankonsistenz mit derVerzögerten Schwachen Konsistenz und Speziellen Schwachen Konsistenz durchgeführt werden.

Das unterschiedliche Verhalten von Invalidierungs- und Aktualisierungsprotokollen im Bezug auf Latenz und Durchsatz, sowie die unterschiedliche Abhängigkeit der Protokolle von verschiedenen Parametern, legt die Entwicklung von transparenten Optimierungsprotokollen nahe. Aufbauend auf der Evaluation der Konsistenzprotokolle, könnten weitere Untersuchungen im Hinblick auf effizientere Optimierungsalgorithmen interessant sein.

Der dritte Bereich betrifft den Datenzugriff: In dieser Arbeit wurde eine Bibliothek für den transparenten Dateizugriff durch Überladen von Systemoperation beschrieben. Andere angeschnittene Möglichkeiten des Datenzugriffs wurden aber nicht weiter verfolgt, wie z.B. dynamische Klassenerweiterungen in Python und Überladen von C++ Operatoren. Neben diesen Ansätzen gibt es eine Vielzahl weiterer Zugriffsmöglichkeiten, die den Aufwand für manuelle Quellcodemodifikationen reduzieren oder sogar ganz ohne (manuelle) Änderungen am Quellcode einer Anwendung auskommen. Interessant in diesem Zusammenhang ist auch die aspektorientierte Programmierung.

Literaturverzeichnis

[1] Aad, G und others: *The ATLAS Experiment at the CERN Large Hadron Collider.* JINST 3 S08003, August 2008.

[2] Adve, Sarita V. und Mark D. Hill: *Weak Ordering—A New Definition.* In: *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA'90)*, Seiten 2–14, 1990.

[3] Ahamad, Mustaque, Rida A. Bazzi, Ranjit John, Prince Kohli und Gil Neiger: *The Power of Processor Consistency.* In: *ACM Symposium on Parallel Algorithms and Architectures*, Seiten 251–260, 1993.

[4] Allen, Gabrielle, Werner Benger, Thomas Dramlitsch, Tom Goodale, Hans-Christian Hege, Gerd Lanfermann, André Merzky, Thomas Radke, Edward Seidel und John Shalf: *Cactus Tools for Grid Applications.* Cluster Computing, 4(3):179–188, Juli 2001.

[5] Andreozzi, Sergio, Natascia De Bortoli, Sergio Fantinel, Antonia Ghiselli, Gian Luca Rubini und Gennaro Tortone Maria Cristina Vistoli: *GridICE: a Monitoring Service for Grid Systems.* In: *Terena Networking Conference 2005*, 2005.

[6] Arjomandi, Eshrat, William O'Farrell, Ivan Kalas, Gita Koblents, Frank Ch. Eigler und Guang R. Gao: *ABC++: concurrency by inheritance in C++.* IBM Systems Journal, 34(1):120–137, 1995.

[7] Bal, Henri E., M. Frans Kaashoek und Andrew S. Tanenbaum: *Orca: a language for parallel programming of distributed systems.* IEEE Transactions on Software Engineering, 18(3):190–205, 1992.

[8] Balis, Bartosz, Marian Bubak, Wlodzimierz Funika, Tomasz Szepieniec, Roland Wismüller und Marcin Radecki: *Monitoring Grid Applications with Grid-enabled OMIS-Monitor.* In: Rivera, F.F. und others (Herausgeber): *Proceedings of First European Across Grids Conference*, Band 2970 der Reihe *Lecture Notes in Computer Science*, Seiten 230–239, Santiago de Compostela, Spain, Februar 2003. Springer-Verlag.

[9] Baratloo, Arash, Partha Dasgupta, Vijay Karamcheti und Zvi M. Kedem: *Metacomputing with MILAN.* In: *Heterogeneous Computing Workshop*, Seiten 169–183, 1999.

[10] Basu, Sujoy, Vanish Talwar, Bikash Agarwalla und Raj Kumar: *Interactive Grid Architecture for Application Service Providers.* In: *International Conference on Web Services 2003 (ICWS '03)*, 2003.

[11] Bataller, Jordi und Josep M. Bernabéu Aubán: *Synchronized DSM Models.* In: *Euro-Par '97: Proceedings of the Third International Euro-Par Conference on Parallel Processing*, Seiten 468–475, London, UK, 1997. Springer-Verlag.

[12] Bataller, Jordi und Josep M. Bernabéu Aubán: *Adaptable Distributed Shared Memory: A Formal Definition.* In: *Euro-Par '98: Proceedings of the 4th International Euro-Par Conference on Parallel Processing*, Seiten 887–891, London, UK, 1998. Springer-Verlag.

[13] Bennett, John K., John B. Carter und Willy Zwaenepoel: *Adaptive Software Cache Management for Distributed Shared Memory Architectures.* In: *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA'90)*, Seiten 125–135, 1990.

[14] Bennett, John K., John B. Carter und Willy Zwaenepoel: *Munin: distributed shared memory based on type-specific memory coherence.* In: *PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, Seiten 168–176, New York, NY, USA, 1990. ACM Press.

[15] Berman, Fran, Geoffrey Fox und Anthony J. G. Hey (Herausgeber): *Grid Computing: Making the Global Infrastructure a Reality.* Wiley Series in Communications Networking & Distributed Systems. Wiley & Sons, März 2003.

[16] Bershad, Brian N. und Matthew J. Zekauskas: *Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors.* Technischer Bericht CMU-CS-91-170, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA (USA), 1991.

[17] Bianchini, Ricardo, Raquel Pinto und Claudio L. Amorim: *Data prefetching for software DSMs.* In: *ICS '98: Proceedings of the 12th international conference on Supercomputing*, Seiten 385–392, New York, NY, USA, 1998. ACM Press.

[18] Blumofe, Robert D., Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson und Keith H. Randall: *Dag-Consistent Distributed Shared Memory.* In: *Proceedings of the 10th International Parallel Processing Symposium (IPPS'96)*, Seiten 132–141, 1996.

[19] Borovac, Stefan, Torsten Harenberg, David Meder-Marouelli, Peter Mättig, Markus Mechtel, Ralph Müller-Pfefferkorn, Reinhard Neumann, Peter Buchholz, Daniel Lorenz, Christian Uebing, Wolfgang Walkowiak, Roland Wismüller, Peer Ueberholz, Anar Manafov, Robert Manteufel, Victor Penso, Carsten Preuss, Kilian Schwarz, Günter Duckeck, Johannes Elmsheuser, Tariq Mahmoud, Dorothee Schailee und Patrick

Fuhrmann: *Overview over the High Energy Physics Community Grid in Germany's D-Grid Initiative.* In: *Proceedings of the XI International Workshop on Advanced Computing and Analysis in Physics Research*, POS(ACAT25)12, Amsterdam, the Netherlands, April 2007.

[20] Brodlie, Ken, David Duce, Julian Gallop und Musbah Sagar: *gViz: Visualization and Computational Steering on the Grid.* In: *Proceedings of the UK e-Science All Hands Meeting 2004*, Seiten 54–60, September 2004.

[21] Brodlie, Ken, David Duce, Julian Gallop, Musbah Sagar, Jeremy Walton und Jason Wood: *Visualization in Grid Computing Environments.* In: *Proceedings of IEEE Visualization 2004*, Seiten 155–162, 2004.

[22] Brodlie, Ken, Andrew Poon, Helen Wright, Lesley Brankin, Greg Banecki und Alan Gay: *GRASPARC: a problem solving environment integrating computation and visualization.* In: *VIS '93: Proceedings of the 4th conference on Visualization '93*, Seiten 102–109, 1993.

[23] Brooke, J. M., P. V. Coveney, J. Harting, S. Jha, S. M. Pickles, R. L. Pinning und A. R. Porter: *Computational Steering in RealityGrid.* In: *Proceedings of the UK e-Science All Hands Meeting*, September 2003.

[24] Brooks, F. P.: *Grasping reality through illusion - interactive graphics serving science.* In: *CHI '88: Proceedings of the SIGCHI conference on Human factors in computing systems*, Seiten 1–11, New York, NY, USA, 1988. ACM Press.

[25] Brüning, Olivier Sim, Paul Collier, P Lebrun, Stephen Myers, Ranko Ostojic, John Poole und Paul Proudlock: *LHC Design Report, v 1 – The LHC Main Ring.* [CERN-2004-003-V-1], ISBN: 789290832249, 2004.

[26] Brunner, J. D., D. J. Jablonowski, B. Bliss und R. B. Haber: *VASE: the visualization and application steering environment.* In: *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, Seiten 560–569, 1993.

[27] Brun, R., F. Rademakers, P. Canal, I. Antcheva und D. Buskulic: *ROOT User Guide.* ftp://root.cern.ch/root/doc/Users_Guide_5_12.pdf.

[28] Brun, Rene und Fons Rademakers: *ROOT - An Object Oriented Data Analysis Framework.* In: *Proceedings of AIHENP'96 Workshop*, Nummer A 389 in *Nuclear Instruments and Methods in Physics research (1997)*, Seiten 81–86, September 1996.

[29] Bubak, Marian, Maciej Malawski und Katarzyna Zając: *Grid Architecture for Interactive Applications.* In: *Parallel Processing and Applied Mathematics*, Band 3019/2004 der Reihe *LNCS*, Seiten 812–820. Springer, April 2004.

[30] Byrom, Rob, Brian A. Coghlan, Andrew W. Cooke, Roney Cordenonsi, Linda Cornwall, Martin Craig, Abdeslem Djaoui, Alastair Duncan, Steve Fisher, Alasdair J. G. Gray, Steve Hicks, Stuart Kenny, Jason Leake, Oliver Lyttleton, James Magowan, Robin Middleton, Werner Nutt, David O'Callaghan, Norbert Podhorszki, Paul Taylor, John Walk und Antony J. Wilson: *Fault Tolerance in the R-GMA Information and Monitoring System.* In: Sloot, Peter M. A., Alfons G. Hoekstra, Thierry Priol, Alexander Reinefeld und Marian Bubak (Herausgeber): *European Grid Conference*, Band 3470 der Reihe *Lecture Notes in Computer Science*, Seiten 751–760. Springer, 2005.

[31] Byrom, Rob und others: *R-GMA: A Relational Grid Information and Monitoring System.* In: *Proc. 2nd Grid Workshop*, Cracow, Dezember 2002.

[32] Carter, John B.: *Design of the Munin distributed shared memory system.* Journal of Parallel and Distributed Computing, 29(2):219–227, 1995.

[33] Carter, John B., John K. Bennett und Willy Zwaenepoel: *Techniques for Reducing Consistency-Related Communication in Distributed Shared-Memory Systems.* ACM Transactions on Computer Systems, 13(3):205–243, 1995.

[34] Carter, John B., Anand Ranganathan und Sai Susarla: *Khazana: An Infrastructure for Building Distributed Services.* In: *International Conference on Distributed Computing Systems*, Seiten 562–571. IEEE, Mai 1998.

[35] Caughey, S. J., D. B. Ingham und M. C. Little: *Flexible open caching for the Web.* Computer Networks and ISDN Systems, 29(8–13):1007–1017, 1997.

[36] CERN - European Laboratory for Particle Physics, http://atlas.web.cern.ch/Atlas/GROUPS/SOFTWARE/OO/z0_obsolete/architecture/General/Tech.Doc/Manual/AthenaDeveloperGuide.pdf: *Athena Developer Guide.*

[37] Cooke, Andrew und others: *Relational Grid Monitoring Architecture (R-GMA).* In: *UK e-Science All Hands Conference (AHM2003)*, September 2003.

[38] Coulard, Olivier, Michael Dussere und Aurelien Esnard: *Toward a Computational Steering Environment based on CORBA.* In: *Parallel Computing: Software Technology, Algorithms, Architectures & Applications*, Band 13 der Reihe *Advances in Parallel Computing*, Seiten 151–158. Elsevier, 2004.

[39] Coveney, Peter V., Jamie Vicary, Jonathan Chin und Matt Harvey: *WEDS: a Web services-based environment for distributed simulation.* Philosophical Transactions of The Royal Society A, 363(1833):1807–1816, August 2005.

[40] Dubois, M., C. Scheurich und F. Briggs: *Memory access buffering in multiprocessors.* In: *ISCA '86: Proceedings of the 13th annual international symposium on Computer architecture*, Seiten 434–442, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[41] Duckeck (ed.), G. und others: *ATLAS Computing Technical Design Report.* [ATLAS TDR–017], [CERN-LHCC-2005-022], Juni 2005.

[42] Dwarkadas, Sandhya, Alan L. Cox und Willy Zwaenepoel: *An integrated compile-time/run-time software distributed shared memory system.* In: *ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, Seiten 186–197, New York, NY, USA, 1996. ACM Press.

[43] Dwarkadas, Sandhya, Kourosh Gharachorloo, Leonidas Kontothanassis, Daniel J. Scales, Michael L. Scott und Robert Stets: *Comparative Evaluation of Fine- and Coarse-Grain Approaches for Software Distributed Shared Memory.* In: *HPCA '99: Proceedings of the 5th International Symposium on High Performance Computer Architecture*, Seite 260, Washington, DC, USA, 1999. IEEE Computer Society.

[44] Dwarkadas, Sandhya, Peter Keleher, Alan L. Cox und Willy Zwaenepoel: *Evaluation of release consistent software distributed shared memory on emerging network technology.* In: *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, Seiten 144–155, New York, NY, USA, 1993. ACM.

[45] Eggers, S. J. und R. H. Katz: *A characterization of sharing in parallel programs and its application to coherency protocol evaluation.* In: *ISCA '88: Proceedings of the 15th Annual International Symposium on Computer architecture*, Seiten 373–382, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[46] Eickermann, Thomas, Wolfgang Frings und Anke Häming: *VISIT - a Visualization Interface Toolkit.* Research Centre Juelich, Central Institute for Applied Mathematics, Juelich, Germany, 2002.

[47] Eisenhauer, Greg S. und Karsten Schwan: *An object-based infrastructure for program monitoring and steering.* In: *SPDT '98: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, Seiten 10–20, New York, NY, USA, 1998. ACM Press.

[48] Eisenhauer, Greg Stephen: *An object infrastructure for high-performance interactive applications.* Doktorarbeit, Georgia Institute of Technology, Mai 1998. Director-Karsten Schwan.

[49] Esnard, Aurelien, Michael Dussere und Olivier Coulard: *A Time-Coherent Model for the Steering of Parallel Simulations.* In: *Proceedings of Euro-Par 2004*, Band 3149 der Reihe *LNCS*, Seiten 90–97. Springer, 2004.

[50] Foster, I., C. Kesselman, J. Nick und S. Tuecke: *The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration*, 2002.

[51] Foster, Ian: *Globus Toolkit Version 4: Software for Service-Oriented Systems*. In: *IFIP International Conference on Network and Parallel Computing*, Nummer 3779 in *LNCS*, Seiten 2–13. Springer-Verlag, 2006.

[52] Foster, Ian und Carl Kesselman: *Globus: A Metacomputing Infrastructure Toolkit*. International Journal of Supercomputer Applications, 11(2):115–128, 1997.

[53] Foster, Ian und Carl Kesselman (Herausgeber): *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., November 1998.

[54] Foster, Ian, Carl Kesselman, Gene Tsudik und Steven Tuecke: *A security architecture for computational Grids*. In: *CCS '98: Proceedings of the 5th ACM Conference on Computer and Communications Security*, Seiten 83–92, New York, NY, USA, 1998. ACM.

[55] Foster, Ian, Carl Kesselman und Steven Tuecke: *The Nexus Approach to Integrating Multithreading and Communication*. Journal of Parallel and Distributed Computing, 37(1):70–82, 1996.

[56] Foster, Ian, Carl Kesselman und Steven Tuecke: *The Anatomy of the Grid: Enabling Scalable Virtual Organizations*. Int. J. High Perform. Comput. Appl., 15(3):200–222, 2001.

[57] Furmento, Nathalie, William Lee, Anthony Mayer, Steven Newhouse und John Darlington: *ICENI: an open Grid service architecture implemented with Jini*. In: *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Seiten 1–10, Los Alamitos, CA, USA, Nov. 2002. IEEE Computer Society Press.

[58] Gao, Guang R. und Vivek Sarkar: *Location Consistency-A New Memory Model and Cache Consistency Protocol*. IEEE Transactions on Computers, 49(8):798–813, 2000.

[59] Geist, Al, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek und Vaidy Sunderam: *PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA, 1994.

[60] Geist, G. A., J. A. Kohl und P. M. Papadopoulos: *CUMULVS: Providing Fault-Tolerance, Visualization and Steering of Parallel Applications*. International Journal of High Performance Computing Applications, 11(3):224–236, August 1997.

[61] Gharachorloo, Kourosh, Daniel Lenoski, James Laudon, Phillip B. Gibbons, Anoop Gupta und John L. Hennessy: *Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors*. In: *25 Years ISCA: Retrospectives and Reprints*, Seiten 376–387, 1998.

[62] Gharachorloo, Kourosh, Daniel Lenoski, James Laudon, Phillip B. Gibbons, Anoop Gupta und John L. Hennessy: *Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors*. In: *25 Years ISCA: Retrospectives and Reprints*, Seiten 376–387, 1998.

[63] Glasner, Christian, Roland Hügl, Bernhard Reitinger, Dieter Kranzlmüller und Jens Volkert: *The Monitoring and Steering Environment.* In: *Proceedings of ICCS 2001*, Band 2074 der Reihe *LNCS*, Seiten 781–790. Springer, 2001.

[64] gLite webpage: http://glite.web.cern.ch/glite.

[65] Goodale, Tom, Gabrielle Allen, Gerd Lanfermann, J. Massóa, Thomas Radke, Edward Seidel und John Shalf: *The Cactus Framework and Toolkit: Design and Applications.* In: *Proceedings of VECPAVector and Parallel Processing R'2002, 5th International Conference.* Springer, 2003.

[66] Goodman, J. R.: *Cache Consistency and Sequential Consistency.* Technischer Bericht 61, IEEE Scalable Coherent Interface Working Group, März 1989.

[67] Grimshaw, Andrew S., Wm. A. Wulf und CORPORATE The Legion Team: *The Legion vision of a worldwide virtual computer.* Communications of the ACM, 40(1):39–45, 1997.

[68] *RFC 2743: GSSAPI version 2 update 1.*

[69] Gu, Weiming, Greg Eisenhauer, Eileen Kraemer, Karsten Schwan, J. Stasko, Jeffrey S. Vetter und N. Mallavarupu: *Falcon: on-line monitoring and steering of large-scale parallel programs.* In: *FRONTIERS '95: Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation (Frontiers'95)*, Seite 422, Washington, DC, USA, 1995. IEEE Computer Society.

[70] Gu, Weiming, Greg Eisenhauer, Karsten Schwan und Jeffrey S. Vetter: *Falcon: On-line Monitoring for Steering Parallel Programs.* Concurrency: Practice and Experience, 6(2), April-June 1998.

[71] Gu, Weiming, Jeffrey Vetter und Karsten Schwan: *An Annotated Bibliography of Interactive Program Steering.* ACM SIGPLAN Notices, 29(9):140–148, September 1994.

[72] Gwertzman, James und Margo Seltzer: *The case for geographical push-caching.* In: *Proceedings of the HotOS '95 Workshop*, 1995.

[73] Haber, Robert B. und David A. McNabb: *Visualization idioms: A conceptual model for scientific visualization systems.* In: Nielson, G.M. und B. Shriver (Herausgeber): *Visualization in Scientific Computing.* IEEE Computer Society Press, 1990.

[74] Hart, Delbert und Eileen Kraemer: *Consistency Considerations in the Interactive Steering of Computations.* International Journal of Parallel and Distributed Systems and Networks, 2(3):171–179, 1999.

[75] Herlihy, Maurice P. und Jeannette M. Wing: *Linearizability: a correctness condition for concurrent objects.* ACM Transactions on Programming Languages and Systems, 12(3):463–492, 1990.

[76] Hervás, Antonio, Enol Fernández, Elisa Heymann, Miquel A. Senar, Marco Sottilaro, Stefano Beco, Alvaro Fernández, Miroslaw Kupczyk, Pavel Wolniewicz und Norbert Meyer: *Interactivity support in the CrossGrid Project.* http://grid.ifca.unican.es/-crossgrid/Archivos/InteractivityChapter_Final.pdf.

[77] Hollingsworth, J. K., B. P. Miller und J. Cargille: *Dynamic program instrumentation for scalable performance tools.* In: *Proceedings of Scalable High-Performance Computing Conference 1994*, Seiten 841–850, Mai 1994.

[78] Huang, Z., S. Cranefield, M. Purvis und C. Sun: *View-Based Consistency and Its Implementation.* In: *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, Seite 74, Washington, DC, USA, 2001. IEEE Computer Society.

[79] Huang, Z., C. Sun, M. Purvis und S. Cranefield: *View-based consistency and false sharing effect in distributed shared memory.* SIGOPS Operating Systems Review, 35(2):51–60, 2001.

[80] Hu, Weihu, Weisong Shi und Z. Tang: *JIAJIA: An SVM System Based on A New Cache Coherence Protocol.* In: *Proceedings of High-Performance Computing and Networking Europe 1999 (HPCN'99)*, Seiten 463–472, 1999.

[81] Hu, Y. Charlie, Weimin Yu, Alan L. Cox, Dan S. Wallach und Willy Zwaenepoel: *Run-Time Support for Distributed Sharing in Typed Languages.* In: *LCR '00: Selected Papers from the 5th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, Seiten 192–206, London, UK, 2000. Springer-Verlag.

[82] I.C.Legrand, H.B.Newman, R.Voicu, C.Cirstoiu, C.Grigoras, M.Toarta und C. Dobre: *MonALISA: An Agent based, Dynamic Service System to Monitor, Control and Optimize Grid based Applications.* In: *CHEP 2004*, September 2004.

[83] Iftode, L., J. P. Singh und K. Li: *Scope Consistency: A Bridge between Release Consistency and Entry Consistency.* In: *Proc. of the 8th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA'96)*, Seiten 277–287, 1996.

[84] Iosevich, Vadim und Assaf Schuster: *A comparison of sequential consistency with home-based lazy release consistency for software distributed shared memory.* In: *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, Seiten 306–315, New York, NY, USA, 2004. ACM Press.

[85] Itzkovitz, Ayal, Nitzan Niv und Assaf Schuster: *Dynamic adaptation of sharing granularity in DSM systems.* Journal of Systems and Software, 55(1):19–32, 2000.

[86] Itzkovitz, Ayal und Assaf Schuster: *MultiView and Millipage - fine-grain sharing in page-based DSMs.* In: *OSDI '99: Proceedings of the third*

symposium on Operating systems design and implementation, Seiten 215–228, Berkeley, CA, USA, 1999. USENIX Association.

[87] James, Michael Rogers, James E. Lumpp Jr und Jame Griffioen: *PDM: Programmable Monitoring For Distributed Applications.* In: *13th International Conference on Parallel and Distributed Computing Systems*, August 2000.

[88] Jha, S., S. Pickles und A. Porter: *A Computational Steering API for Scientific Grid Applications: Design, Implementation and Lessons.* In: *Workshop on Grid Application Programming Interfaces*, September 2004.

[89] Jiang, Lin, Hua Liu, Manish Parashar und Deborah Silver: *Rule-Based Visualization in a Computational Steering Collaboratory.* In: *ICCS 2004*, Band 3038 der Reihe *LNCS*, Seiten 58–65. Springer, 2004.

[90] John, Ranjit und Mustaque Ahamad: *Evaluation of Causal Distributed Shared Memory for Data-race-free Programs.* Technischer Bericht GIT-CC-94-34, Georgia Institute of Technology, 1994.

[91] Johnson, Kirk L., M. Frans Kaashoek und Deborah A. Wallach: *CRL: High-Performance All-Software Distributed Shared Memory.* ACM Operating Systems Review, SIGOPS, 29(5):213–226, 1995.

[92] Karlsson, Magnus und Per Stenström: *Effectiveness of Dynamic Prefetching in Multiple-Writer Distributed Virtual Shared-Memory Systems.* Journal of Parallel and Distributed Computing, 43(2):79–93, 1997.

[93] Keleher, P., S. Dwarkadas, A. L. Cox und W. Zwaenepoel: *TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems.* In: *Proceedings of the Winter 1994 USENIX Conference*, Seiten 115–131, 1994.

[94] Keleher, Pete, Alan L. Cox und Willy Zwaenepoel: *Lazy Release Consistency for Software Distributed Shared Memory.* In: *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA'92)*, Seiten 13–21, 1992.

[95] Kermarrec, A. M., I. Kuz, M. Van Steen und A. S. Tanenbaum: *A Framework for Consistent, Replicated Web Objects.* In: *ICDCS '98: Proceedings of the The 18th International Conference on Distributed Computing Systems*, Seite 276, Washington, DC, USA, 1998. IEEE Computer Society.

[96] Knobloch, Jürgen: *LHC Computing Grid Technical Design Report.* Technischer Bericht LCG-TDR-001, CERN, Geneva, Switzerland, Juni 2005.

[97] Kohl, J. A. und P. M. Papadopoulos: *Efficient and Flexible Fault Tolerance and Migration of Scientific Simulations Using CUMULVS.* In: *2nd SIGMETRICS Symposium on Parallel and Distributed Tools*, August 1998.

[98] Kohn, Emil-Dan und Assaf Schuster: *A Transparent Software Distributed Shared Memory.* In: Kosch, H. und L. Böszörmenyi (Herausgeber): *Proceedings of Euro-Par 2003*, Nummer 2790 in *LNCS*, Seiten 1180–1189. Springer-Verlag, 2003.

[99] Kontothanassis, Leonidas, Galen Hunt, Robert Stets, Nikolaos Hardavellas, MichałCierniak, Srinivasan Parthasarathy, Jr. Wagner Meira, Sandhya Dwarkadas und Michael Scott: *VM-based shared memory on low-latency, remote-memory-access networks.* SIGARCH Comput. Archit. News, 25(2):157–169, 1997.

[100] Kraemer, Eileen, Delbert Hart und Gruia-Catalin Roman: *Balancing Consistency and Lag in Transaction-Based Computational Steering.* In: *HICSS '98: Proceedings of the Thirty-First Annual Hawaii International Conference on System Sciences-Volume 7*, Seite 137, Washington, DC, USA, 1998. IEEE Computer Society.

[101] Kunze, M. und The CrossGrid Collaboration: *The CrossGrid project.* Nuclear Instruments and Methods in Physics Research A, 502, April 2003.

[102] Lai, An-Chow und Babak Falsafi: *Memory sharing predictor: the key to a speculative coherent DSM.* In: *ISCA '99: Proceedings of the 26th annual international symposium on Computer architecture*, Seiten 172–183, Washington, DC, USA, 1999. IEEE Computer Society.

[103] Lamport, Leslie: *How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs.* IEEE Transactions on Computers, C-28(9):690–691, September 1979.

[104] Lang, Bo, Ian Foster, Frank Siebenlist, Rachana Ananthakrishnan und Tim Freeman: *A Multipolicy Authorization Framework for Grid Security.* In: *NCA '06: Proceedings of the Fifth IEEE International Symposium on Network Computing and Applications*, Seiten 269–272, Washington, DC, USA, 2006. IEEE Computer Society.

[105] Liere, Robert van, Jurriaan D. Mulder und Jarke J. van Wijk: *Computational Steering.* Future Generation Computer Systems, 12(5), April 1997.

[106] Liere, Robert van und Jarke J. van Wijk: *CSE: A Modular Architecture for Computational Steering.* In: *Proceedings EG Workshop on Visualization in Scientific Computing.* Eurographics and Springer, 1996.

[107] Li, Kai und Paul Hudak: *Memory coherence in shared virtual memory systems.* ACM Transactions on Computer Systems, 7(4):321–359, 1989.

[108] Lipton, R. J. und J. S. Sandberg: *PRAM: A scaleable shared memory.* Technischer Bericht CS-TR-180-88, Princeton University, September 1988.

[109] Lorenz, Daniel: *RMOST Developer Guide.* Universität Siegen, http://www.hep.physik.uni-siegen.de/grid/rmost/doc/DeveloperGuide.pdf.

[110] Lorenz, Daniel: *RMOST User Guide.* Universität Siegen, http://www.hep.physik.uni-siegen.de/grid/rmost/doc/UserGuide.pdf.

[111] Lorenz, Daniel, Stefan Borovac, Peter Buchholz, Henrik Eichenhardt, Torsten Harenberg, Peter Mättig, Markus Mechtel, Ralph Müller-Pefferkorn, Reinhard Neumann, Kendall Reeves, Christian Uebing, Wolfgang Walkowiak, Thomas William und Roland Wismüller: *Job Monitoring and Steering in D-Grid's High Energy Physics Community Grid.* Future Generation Computing Systems, 25(3):308–314, 2009.

[112] Lorenz, Daniel, Peter Buchholz, Christian Uebing, Wolfgang Walkowiak und Roland Wismüller: *Online Steering of HEP Grid Applications.* In: *Proceedings of the Cracow Grid Workshop '06*, Seiten 191–198, Cracow, Poland, Juli 2007. Academic Computer Centre CYFRONET AGH.

[113] Lorenz, Daniel, Peter Buchholz, Christian Uebing, Wolfgang Walkowiak und Roland Wismüller: *RMOST: A Shared Memory Model for Online Steering.* In: Bubak, Marian, Geert Dick van Albada, Jack Dongarra und Peter M.A. Sloot (Herausgeber): *Computational Science – ICCS 2008*, Band 5103 der Reihe *LNCS*, Seiten 223–232. Springer, 2008.

[114] Lorenz, Daniel, Peter Buchholz, Christian Uebing, Wolfgang Walkowiak und Roland Wismüller: *Secure connections for computational steering of Grid jobs.* In: *Proceedings of the 16th Euromicro International Conference on Parallel, Distributed and network-based Processing*, Seiten 209–217, Toulouse, France, Februar 2008. IEEE.

[115] Lorenz, Daniel, Peter Buchholz, Christian Uebing, Wolfgang Walkowiak und Roland Wismüller: *Steering of Sequential Jobs with a Distributed Shared Memory Based Model for Online Steering.* Future Generation Computing Systems, 2009.

[116] Madhyastha, Tara M. und Daniel A. Reed: *Intelligent, Adaptive File System Policy Selection.* In: *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, Seiten 172–179. IEEE Computer Society Press, 1996.

[117] Madhyastha, Tara M. und Daniel A. Reed: *Exploiting Global Input/Output Access Pattern Classification.* In: *Proceedings of SC97: High Performance Networking and Computing*, San Jose, 1997. ACM Press.

[118] Mann, Vijay, Vincent Matossian, Rajeev Muralidhar und Manish Parashar: *DISCOVER: An environment for Web-based interaction and steering of high-performance scientific applications.* Concurrency and Computation: Practice and Experience, 2001.

[119] Marshall, Robert, Jill Kempf, Scott Dyer und Chieh-Cheng Yen: *Visualization methods and simulation steering for a 3D turbulence model of Lake Erie.* SIGGRAPH Comput. Graph., 24(2):89–97, 1990.

[120] Meyer, Norbert und others: *Deliverable D3.7, WP3 Final Report.* CrossGrid, 2005.

[121] Miller, David W., Jinhua Guo, Eileen Kraemer und Yin Xiong: *On-the-fly calculation and verification of consistent steering transactions.* In: *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, Seiten 1–17, 2001.

[122] Miller, Michelle, Charles D. Hansen, Steven G. Parker und Christopher R. Johnson: *Simulation Steering with SCIRun in a Distributed Environment.* In: *Proceedings of the The Seventh IEEE International Symposium on High Performance Distributed Computing*, Seite 364, 1998.

[123] Mulder, Jurriaan D., Jarke J. van Wijk und Robert van Liere: *A survey of computational steering environments.* Future Generation Computer Systems, 15(1):119–129, 1999.

[124] Mulder, Jurriaan Derk: *Computational Steering with Parametrized Geometric Objects.* Doktorarbeit, Universiteit van Amsterdam, Juni 1998.

[125] Müller-Pfefferkorn, Ralph, Reinhard Neumann, Stefan Borovac, Torsten Harenberg, Matthias Hüsken, Peter Mättig, Markus Mechtel, David Meder-Marouelli, Peer Ueberholz, Peter Buchholz, Daniel Lorenz, Christian Uebing, Wolfgang Walkowiak und Roland Wismüller: *User-Centric Monitoring and Steering of the Execution of Large Job Sets.* In: *Proceedings of the German e-Science Conference 2007*, Baden-Baden, Germany, Mai 2007.

[126] Muralidhar, Rajeev und Manish Parashar: *A Distributed Object Infrastructure for Interaction and Steering.* In: *Proceedings of the 7th Euro-Par Conference*, Band 2150 der Reihe *LNCS*, Seiten 67–74. Springer, 2001.

[127] National Institute of Standards and Technology, FIPS PUB 197: *Advanced Encryption Standard (AES)*, November 2001.

[128] Neuman, C., T. Yu, S. Hartman und K. Raeburn: *RFC 4120: The Kerberos Network Authentication System*, Juli 2005.

[129] Newman, Harvey B., Iosif C. Legrand und Julian J. Bunn: *A Distributed Agent-based Architecture for Dynamic Services.* In: *CHEP 2001*, September 2001.

[130] Newman, H.B., I.C. Legrand, P.Galvez, R. Voicu und C. Cirstoiu: *MonALISA: A Distributed Monitoring Service Architecture.* In: *CHEP 2003*, March 2003.

[131] Nitzberg, Bill und Virginia Lo: *Distributed Shared Memory: A Survey of Issues and Algorithms.* Computer, 24(8):52–60, 1991.

[132] Niv, Nitzan und Assaf Schuster: *Transparent adaptation of sharing granularity in MultiView-based DSM systems.* Software Practice and Experience, 31(15):1439–1459, 2001.

[133] Papadopoulos, P. M., J. A. Kohl und B. D. Semeraro: *CUMULVS: Extending a Generic Steering and Visualization Middleware for Application Fault-Tolerance.* In: *Proceedings of the 31st Hawaii International Conference on System Sciences (HICSS-31)*, Januar 1998.

[134] Parker, S.G., M. Miller, C.D. Hansen, C.R. Johnson und P.-P. Sloan: *An Integrated Problem Solving Environment: The SCIRun Computational Steering System.* In: El-Rewini, H. (Herausgeber): *31st Hawaii International Conference on System Sciences (HICSS-31)*, Band VII, Seiten 147–156. IEEE, Januar 1998.

[135] Parker, Steven G., Christopher R. Johnson und David Beazley: *Computational Steering Software Systems and Strategies.* IEEE Computational Science & Engineering, 4(4):50–59, Oktober 1997.

[136] Parker, Steven Gregory: *The SCIRun Problem Solving Environment and Computational Steering Software System.* Doktorarbeit, University of Utah, 1999.

[137] Pawlowski, Brian, Spencer Shepler, Carl Beame, Brent Callaghan, Michael Eisler, David Noveck, David Robinson und Robert Thurlow: *The NFS Version 4 Protocol.* Proceedings of the 2nd international system administration and networking conference (SANE2000), Seite 94, 2000.

[138] Pickles, S. M., R. Haines, R. L. Pinning und A. R. Porter: *Practical Tools for Computational Steering.* In: *Proceedings of the UK e-Science All Hands Meeting*, September 2004.

[139] Placek, Martin und Rajkumar Buyya: *G-Monitor: A Web Portal for Monitoring and Steering Application Execution on Global Grids.* In: *CLADE '03: Proceedings of the International Workshop on Challenges of Large Applications in Distributed Environments*, Seite 10, Washington, DC, USA, 2003. IEEE Computer Society.

[140] Plociennik, Marcin, Bartek Palak und Robert Pajak: *Developer Manual - Migrating Desktop and Roaming Access Server.* CrossGrid, 2005.

[141] Porter, Andrew: *The RealityGrid Steering Grid Service.* SVE Group, Manchester Computing, April 2004.

[142] *R-GMA User Guide for C++ Programmers.* http://www.r-gma.org.

[143] Ramachandran, Umakishore und M. Yousef A. Khalidi: *An implementation of distributed shared memory.* Software - Practice and Experience, 21(5):443–464, 1991.

[144] Rathmayer, Sabine: *A Tool for On-line Visualization and Interactive Steering of Parallel HPC Applications.* In: *IPPS '97: Proceedings of the 11th International Symposium on Parallel Processing*, Seiten 181–186, Washington, DC, USA, 1997. IEEE Computer Society.

[145] Rathmayer, Sabine: *Visualization and Computational Steering in Heterogeneous Computing Environments.* In: *Proceedings of Euro-Par 2000*, Band 1900 der Reihe *LNCS*, Seiten 47–56. Springer, 2000.

[146] Reed, D. A., R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz und L. F. Tavera: *Scalable Performance Analysis: The Pablo Performance Analysis Environment*. In: *Proceedings of Scalable Parallel Libraries Conference*, Seiten 104–113. IEEE Computer Society, 1993.

[147] Reed, Daniel A.: *Experimental Analysis of Parallel Systems: Techniques and Open Problems*. In: *Computer Performance Evaluation*, Seiten 25–51, 1994.

[148] Reinhardt, S. K., J. R. Larus und D. A. Wood: *Tempest and Typhoon: User-Level Shared Memory*. In: *Proceedings of the 21th Annual International Symposium on Computer Architecture (ISCA'94)*, Seiten 325–337, 1994.

[149] Ribler, Randy L., Huseyin Simitci und Daniel A. Reed: *The Autopilot performance-directed adaptive control system*. Future Generation Computer Systems, 18(1):175–187, 2001.

[150] Rivest, Ronald L., Adi Shamir und Leonard M. Adleman: *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*. Communications of the ACM, 21(2):120–126, Februar 1978.

[151] RMOST webpage: http://www.hep.physik.uni-siegen.de/grid/rmost.

[152] Rosmanith, Herbert und Dieter Kranzlmüller: *glogin - A Multifunctional, Interactive Tunnel into the Grid*. In: *5th IEEE/ACM International Workshop on Grid Computing*, November 2004.

[153] Rosmanith, Herbert und Jens Volkert: *glogin - Interactive Connectivity for the Grid*. In: Juhasz, Z., P. Kacsuk und D. Kranzlmüller (Herausgeber): *Proceedings of DAPSYS 2004, 5th Austrian-Hungarian Workshop on Distributed and Parallel Systems*, Seiten 3–11. Kluwer Academic Publishers, September 2004.

[154] Rosmanith, Herbert und Jens Volkert: *Traffic forwarding with GSH/GLOGIN*. In: *Proceedings of EUROMICRO-PDP*, Februar 2005.

[155] Ryan, J. P. und B. A. Coghlan: *Distributed Shared Memory in a Grid Environment*. In: Joubert, G.R., W.E. Nagel, F.J. Peters, O. Plata, P. Tirado und E. Zapata (Herausgeber): *Parallel Computing: Current and Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005*, Band 33 der Reihe *NIC*, Seiten 129–136, Oktober 2006.

[156] Scales, Daniel J., Kourosh Gharachorloo und Chandramohan A. Thekkath: *Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory*. In: *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOSVII)*, Seiten 174–185, 1996.

[157] Schoinas, Ioannis, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus und David A. Wood: *Fine-grain access control for distributed shared memory*. SIGOPS Operating Systems Review, 28(5):297–306, 1994.

[158] Schopf, Jennifer M., Mike D'Arcy, Neill Miller, Laura Pearlman, Ian Foster und Carl Kesselman: *Monitoring and Discovery in a Web Services Framework: Functionality and Performance of the Globus Toolkit's MDS4*. Technischer Bericht Tech Report ANL/MCS-P1248-0405, Argonne National Laboratory, April 2005.

[159] Schroeder, William J. und Kenneth M. Martin: *The Visualization Toolkit*. In: Johnson, Christopher und Charles Hansen (Herausgeber): *Visualization Handbook*, Seiten 593–614. Academic Press, Inc., Orlando, FL, USA, 2004.

[160] Shi, Weisong: *Heterogeneous Distributed Shared Memory on Wide Area Network*. IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, Januar 2001.

[161] Snir, Marc, Steve Otto, Steven Huss-Lederman, David Walker und Jack Dongarra: *MPI: The Complete Reference*. MIT Press, Cambridge, MA, 1996.

[162] Sosič, Rok: *Dynascope: a tool for program directing*. In: *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, Seiten 12–21, New York, NY, USA, 1992. ACM Press.

[163] Sosič, Rok: *The Dynascope Directing Server: Design and Implementation*. Computing Systems, 8(2):107–134, 1995.

[164] Sotomayor, Borja und Lisa Childers: *Globus Toolkit 4 - Programming Java Services*. Morgan Kaufmann, November 2005.

[165] Steenberg, Conrad D., Shih-Chieh Hsu, Elliot Lipeles und Frank Würthwein: *JobMon: A Secure, Scalable, Interactive Grid Job Monitor*. In: *Proccedings of CHEP '06*, Februar 2006.

[166] Steen, Marteen van, Philip Homburg und Andrew S. Tanenbaum: *The Architectural Design of Globe: A Wide-Area Distributed System*. Technischer Bericht IR-422, Vrije Universiteit Amsterdam, Department of Mathematics and Computer Science, Amsterdam, Netherlands, 1997.

[167] Steinke, Robert C. und Gary J. Nutt: *A unified theory of shared memory consistency*. Journal of the ACM, 51(5):800–849, 2004.

[168] St. Germain, J. Davison de, John McCorquodale, Steven G. Parker und Christopher R. Johnson: *Uintah: A Massively Parallel Problem Solving Environment*. In: *Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC-9 '00)*, Seite 33, 2000.

[169] Streit, A., D. Erwin, Th. Lippert, D. Mallmann, R. Menday, M. Rambadt, M. Riedel, M. Romberg, B. Schuller und Ph. Wieder: *UNICORE - From Project Results to Production Grids*. Grid Computing: The New Frontiers of High Performance Processing, Advances in Parallel Computing, Seiten 357–376, 2005.

[170] Sun Mircosystems, Inc.: *Solaris Security for Developers Guide*, 2005.

[171] Tanenbaum, Andrew S.: *Computernetzwerke*. Pearson Studium, 2003.

[172] The ATLAS Collaboration: *ATLAS TDR 14 Detector and physics performance, volume 1*. Technischer Bericht LHCC 99-14, CERN, 1999.

[173] The ATLAS Collaboration: *ATLAS TDR 15 Detector and physics performance, volume 2*. Technischer Bericht LHCC 99-15, CERN, 1999.

[174] Upson, Craig, Jr. Thomas Faulhaber, David Kamins, David H. Laidlaw, David Schlegel, Jefrey Vroom, Robert Gurwitz und Andries van Dam: *The Application Visualization System: A Computational Environment for Scientific Visualization*. IEEE Computer Graphics and Applications, 9(4):30–42, 1989.

[175] U.S. Department of Commerce/National Institute of Standards and Technology, FIPS PUB 46-3: *Data Encryption Standart (DES)*.

[176] Veldema, R., R. F. H. Hofman, R. A. F. Bhoedjang, C. J. H. Jacobs und H. E. Bal: *Source-level global optimizations for fine-grain distributed shared memory systems*. In: *PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, Seiten 83–92, New York, NY, USA, 2001. ACM Press.

[177] Vetter, Jeffrey S. und Daniel A. Reed: *Real-Time Performance Monitoring, Adaptive Control, and Interactive Steering of Computational Grids*. International Journal of High Performance Computing Applications, 14(4):357–366, 2000.

[178] Vetter, Jeffrey S. und Karsten Schwan: *Progress: A Toolkit for Interactive Program Steering*. In: *Proceedings of the 24th International Conference on Parallel Processing*, Seiten II:139–142, Oconomowoc, WI, 1995.

[179] Vetter, Jeffrey S. und Karsten Schwan: *Models for Computational Steering*. Technischer Bericht GIT-CC-95-39, College of Computing, Georgia Institute of Technology, Februar 1996.

[180] Vetter, Jeffrey S. und Karsten Schwan: *Techniques for Delayed Binding of Monitoring Mechanisms to Application-Specific Instrumentation Points*. In: *Proceedings of International Conference on Parallel Processing (ICPP)*, Seiten 477–484, 1998.

[181] Vetter, Jeffrey S. und Karsten Schwan: *High Performance Computational Steering of Physical Simulations*. In: *Proceedings of the 11th International Symposium on Parallel Processing*, Seiten 128–134. IEEE, 1999.

[182] Vetter, Jeffrey S. und Karsten Schwan: *Optimizations for Language-Directed Computational Steering*. In: *Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing*, Seiten 486–491. IEEE, 1999.

[183] Walton, Jeremy: *NAG's IRIS Explorer.* In: Johnson, Christopher und Charles Hansen (Herausgeber): *Visualization Handbook*, Seiten 633–654. Academic Press, Inc., Orlando, FL, USA, 2004.

[184] Weber, Wolf-Dietrich und Anoop Gupta: *Analysis of cache invalidation patterns in multiprocessors.* In: *ASPLOS-III: Proceedings of the third international conference on Architectural support for programming languages and operating systems*, Seiten 243–256, New York, NY, USA, 1989. ACM Press.

[185] Weinstein, D.M., S.G. Parker, J. Simpson, K. Zimmerman und G. Jones: *Visualization in the SCIRun Problem-Solving Environment.* In: Hansen, C.D. und C.R. Johnson (Herausgeber): *The Visualization Handbook*, Seiten 615–632. Elsevier, 2005.

[186] Wilson, A. J. und others: *Information and Monitoring Services within a Grid Environment.* In: *CHEP 2004*, September 2004.

[187] Wood, Jason, Ken Brodlie und Jeremy Walton: *gViz: Visualization and Steering for the Grid.* In: *Proceedings of the UK e-Science All Hands Meeting 2003*, September 2003.

[188] Yu, Byung-Hyun, Stephen Cranefield, Zhiyi Huang und Martin Purvis: *Homeless and Home-based Lazy Release Consistency Protocols on Distributed Shared Memory.* In: Estivill-Castro, Vladimir (Herausgeber): *Twenty-Seventh Australasian Computer Science Conference (ACSC2004)*, Band 26 der Reihe *CRPIT*, Seiten 117–123, Dunedin, New Zealand, 2004. ACS.

[189] Yu, Weimin und Alan L. Cox: *Java/DSM: A Platform for Heterogeneous Computing.* Concurrency - Practice and Experience, 9(11):1213–1224, 1997.

[190] Zhou, Yuanyuan, Liviu Iftode, Jaswinder Pal Sing, Kai Li, Brian R. Toonen, Ioannis Schoinas, Mark D. Hill und David A. Wood: *Relaxed consistency and coherence granularity in DSM systems: a performance evaluation.* In: *PPOPP '97: Proceedings of the sixth ACM SIGPLAN symposium on Principles and practice of parallel programming*, Seiten 193–205, New York, NY, USA, 1997. ACM Press.