Transaktionales Methoden-Caching im Applikationsserver-Bereich

Daniel Pfeifer
6. August 2004

Technischer Bericht 2004-13

Universität Karlsruhe (TH)
Fakultät für Informatik
Institut für Programmstrukturen und Datenorganisation
Lehrstuhl Professor Lockemann

Zusammenfassung

Applikationsserver haben etwa durch CORBA und EJB große Bedeutung gewonnen, denn auf ihnen fußen unternehmenskritische Anwendungen. Allgemein ermöglichen Applikationsserver den *entfernten Zugriff auf Geschäftsfunktionalität mittels einer objektorientierten Dienstschnittstelle*. Aus der Sicht einer Klientenanwendung, die ein derartiges System nutzt, gibt es aber ein großes Problem: Die *Leistung des Servers in Bezug auf Skalierbarkeit und Antwortzeiten* ist häufig unbefriedigend.

Dieser Bericht versucht durch eine neue Cache-Strategie Abhilfe zu leisten. Er zeigt, wie man *im Zusammenhang mit klientenbasierten Anwendungstransaktionen das Caching von Methodenresultaten auf der Klientenseite* eines Applikationsserver-Systems ermöglicht. Das Cache-Verfahren ist dabei transparent und konsistent, so dass sich der Anwendungscode weder auf Klienten- noch auf Serverseite durch die Einführung des Caches verändert.

Häufig sollen im Code eines Klientenprogramms eine Menge von Dienstmethodenaufrufen in ACID-Transaktionen gekapselt werden – man spricht von klientenbasierten Anwendungstransaktionen. Die Ausführung der Dienstmethoden ist auf der Applikationsserver-Seite mit einem Transaktionsverwalter und transaktionalen Ressourcen gekoppelt. Das Caching von Methodenresultaten muss daher die *transaktionale Konsistenz* entsprechender ACID-Transaktionen bewahren.

Zur Realisierung des Cache-Verfahrens wird im Grundsatz ein Transaktionsverwalter, der dem Applikationsserver unterstellt ist, um die Verarbeitung von so genannten *Methodenoperationen* erweitert. Letztere spiegeln den Zugriff auf vorgehaltene Methodenresultate wider, die während einer Transaktion beim Klienten aus dem Cache gelesen wurden. Der Cache arbeitet in diesem Zusammenhang intertransaktional, das heißt, er hält Ergebnisse aus einer Transaktion für zukünftige Transaktionen vor.

Die Hauptschwierigkeit beim transaktionalen, methodenbasierten Caching ist die *Zusicherung von Serialisierbarkeit und Rücksetzbarkeit*, also den bedeutendsten Eigenschaften von ACID-Transaktionen. Dazu wird ein etablierter Formalismus zur Transaktionstheorie um die Behandlung von Methodenoperationen erweitert. Der Bericht entwickelt eine formale Semantik für Transaktionen mit Methodenoperationen, die aus existierenden Transaktionsmodellen resultiert. Daraus entstehen drei Serialisierbarkeitsprotokolle und ein Rücksetzbarkeitsprotokoll, deren Korrektheit nachgewiesen wird. Darüber hinaus liefert der Bericht konkrete Implementierungsvorschläge für die Protokolle und eine dafür notwendige Architekturerweiterung eines Applikationssservers. Die Protokollimplementierungen werden am Ende analytisch auf Eigenschaften wie die Cache-Trefferrate und die Transaktionsabbruchrate hin untersucht.

Erste experimentelle Ergebnisse zeigen hervorragende Leistungssteigerungen für ein System, das einen entsprechenden Cache einsetzt. Die Experimente und Resultate bilden aber nicht Teil dieses Berichts.

Inhaltsverzeichnis

1 Einführung **1**
1.1 Idee . . . 1
1.2 Aufbau des Berichts . . . 3

2 Stand der Forschung beim transaktionalen Caching **3**
2.1 Einführung . . . 3
2.2 Protokolle . . . 5
2.2.1 Cachendes Zwei-Phasen-Sperrprotokoll . . . 6
2.2.2 Rückruf-Lese-Sperrprotokoll . . . 6
2.2.3 Optimistic Concurrency Control Protokoll . . . 7
2.3 Allgemeine Protokolleigenschaften . . . 8

3 Transaktionales Methoden-basiertes Caching **9**
3.1 Anwendungstransaktionen im Applikationssserver-Bereich . . . 9
3.2 Architektur eines Applikationsservers mit transaktionalem Methoden-Cache . . . 13

4 Serialisierbarkeitstheorie **19**
4.1 Einführung . . . 19
4.1.1 Idee . . . 20
4.1.2 Aufbau des Formalismus . . . 21
4.2 Methoden-Cache-Transaktionen und -Historien . . . 22
4.3 rw- und Mehr-Versionen-Historien . . . 25
4.4 Einbettung von rw- in Mehr-Versionen-Historien . . . 27
4.5 Einbettung von Methoden-Cache- in Mehr-Versionen-Historien . . . 30

5 Entwurfsansätze zum transaktionalen Methoden-Caching für EJB-basierte Applikationsserver **35**
5.1 Typische Klassenstruktur eines EJB-basierten Applikationsservers . . . 35
5.2 Erweiterung um Methoden-Cache und m-Planer . . . 39
5.3 Details beim Methoden-Cache . . . 43
5.4 Details beim m-Planer . . . 47
5.5 Speicherverwaltung bei vielen Klienten . . . 50

6 Eigenschaften zur Fehlererholung (Recovery) **50**
6.1 Formalismus . . . 50
6.2 Implementierung . . . 54
6.2.1 Verfahren . . . 54
6.2.2 Umsetzung . . . 55

7 Serialisierbarkeitsprotokolle **56**
7.1 Sperrprotokoll . . . 57
7.1.1 Idee . . . 57
7.1.2 Formalismus . . . 58
7.1.3 Implementierung . . . 62
7.2 Aktualitätsbasiertes Zeitstempelprotokoll . . . 64
7.2.1 Idee . . . 64
7.2.2 Basisformalismus . . . 68
7.2.3 Integration mit Sperrprotokoll-basiertem rw-Planer . . . 71
7.2.4 Implementierung . . . 73
7.3 Passungsbasiertes Zeitstempelprotokoll . . . 75
7.3.1 Idee . . . 75
7.3.2 Formalismus . . . 78
7.3.3 Implementierung . . . 81

8 Resultierende Protokolleigenschaften **90**
8.1 Bezug zum Stand der Forschung . . . 90
8.2 Analytische Abschätzung von Protokolleigenschaften . . . 92
8.2.1 Cache-Trefferrate . . . 93
8.2.2 Abbruchwahrscheinlichkeit durch den m-Planer . . . 93

1 Einführung

1.1 Idee

Die Nachfrage nach Applikationserver-Technologien und damit zusammenhängenen Produkten auf dem IT-Mark ist extrem hoch, denn sie werden in vielen Bereichen, wie etwa dem elektronischen Handel (E-Commerce) oder der Unternehmensressourcenplanung (Enterprise Resource Planning, ERP), genutzt. Aus der Sicht einer Klientenanwendung, die auf derartige Systeme zugreift, gab und gibt es aber nach wie vor ein großes Problem: *Die Leistung in Bezug auf Skalierbarkeit und Antwortzeiten* für die daraus resultierenden modernen Klient-Server-Systeme ist häufig unbefriedigend.

Üblicherweise stellt ein Applikationssserver Anwendungsfunktionalität in Form einer objektorientierte Schnittstelle für die entfernte Klienten bereit – die so genannte *Dienstschnittstelle*. Für die Definition der Schnittstelle wird dabei statische Typsicherheit vorausgesetzt. Die Struktur der Dienstschnittstelle ist also im Wesentlichen durch eine Menge von Klassen gegegeben, die aus Methoden und eventuell Attributen bestehen.

Damit ein entfernter Klient die Dienstschnittstelle nutzen kann, wird ein Protokoll für entfernte Methodenaufrufe benötigt. Beispiele hierfür bilden CORBA-IIOP ([34]), Java RMI ([31]) und .NET Remoting ([21]). Die bekanntesten Beispiele für Applikationserver-Technologien sind Teile der Common Request Broker Architecture (CORBA, [34]), der Enterprise Java Beans-Standard von Sun (EJB, [29]) und das .NET Framework von Microsoft (.NET, [20]).

Dieser Bericht greift die Ergebnisse aus [25, 26, 24] auf. Dort werden Resultate von Methodenaufrufen aus der Dienstschnittstelle in einem Cache vorgehalten, um sie für spätere Methodenaufrufe mit dem gleichem Aufrufkontext – also der gleichen Argumentliste – wiederzuverwenden. Enthält der Cache ein entsprechendes Methodenresultat, kann es direkt an den Aufrufer zurückgegeben werden, ohne dass die Methode im Applikationsserver ausgeführt werden muss. Dieser Cache-Ansatz orientiert sich also *nicht* an den Zuständen von Server-seitigen Objekten, sondern ausschließlich an Resultaten von Methoden aus der Dienstschnittstelle. Das Verfahren wird deshalb hier als *methodenbasiertes Caching oder kurz Methoden-Caching* bezeichnet.

Bei einem Cache-Treffer entfällt die potentiell aufwändige Ausführung eines entfernten Methodenaufrufs beim Server, und dies reduziert die Antwortzeit. Wenn viele Klienten gleichzeitig auf den Applikationsserver zugreifen, wird dieser unter Einsatz eines Methoden-Caches bei guten Trefferraten weniger belastet, was die Skalierbarkeit des Gesamtsystems verbessert.

Anwendungstransaktionen sind von einem Applikationsserver unterstützte ACID-Transaktion, die potentiell verteilt sein können und Anwendungscode von Dienstmethoden umspannen. Wenn die Grenzen (also der Beginn und das Ende) einer Anwendungstransaktion im Klientencode festgelegt werden, spricht man genauer von *klientenbasierten Anwendungstransaktionen*. Klientenbasierte Anwendungstransaktionen können mehrere Dienstmethodenaufrufe umspannen kann von allen oben genannten Applikationsserver-Technologien unterstützt.

Die Beiträge [25, 26, 24] haben Methoden-Caching bereits eingehend untersucht, *aber*

ohne die Berücksichtigung von klientenbasierten Anwendungstransaktionen. Der vorliegende Bericht beantwortet hingegen die Frage, *wie methodenbasiertes Caching mit klientenbasierten Anwendungstransaktionen zusammenspielt*. Man gelangt dadurch zum so genannten *transaktionalen methodenbasierten Caching*.

Der Konsistenzaspekt beim Methoden-Caching lässt sich dabei auf die ACID-Eigenschaften von Anwendungstransaktionen zurückführen, also unter anderem auf Atomizität, Dauerhaftigkeit und Isolation. Sofern der Methoden-Cache auch im Kontext von Transaktionen alle Aufrufe von Schreibmethoden an den Applikationsserver delegiert, bleiben die Eigenschaften Atomizität und Dauerhaftigkeit automatisch erhalten.

Die Isolationseigenschaft schließt insbesondere die *Serialisierbarkeit* von Transaktionen ein, und letztere lässt sich beim methodenbasierten Caching, wie es in [25, 26, 24] vorgestellt wurde, nicht ohne weiteres zusichern: Einerseits können die in einer Transaktion T_1 verwendeten, gecachten Methodenresultate *veraltet* sein, etwa weil ein Methodenresultat nicht rechtzeitig invalidiert wurde. Ein verwendetes Methodenresulat kann für T_1 aber auch *zu aktuell* sein! Dieser Fall tritt ein, wenn eine andere Transaktion T_2 ein vorhandenes gecachtes Methodenresultat durch einen aktuelleren Wert ersetzt. Wenn ein Ressourcenverwalter T_2 nach T_1 in seiner Serialisierungsordnung einplant, T_1 aber das durch T_2 bereitgestellte, gecachte Methodenresultat liest, kann es sein, dass T_1 nicht serialisierbar ist. Ohne bereits an dieser Stelle auf die Details des Beispiels einzugehen, soll betont werden, dass es großer Sorgfalt bedarf, um Serialisierbarkeit im Zusammenhang mit transaktionalem methodenbasiertem Caching zu garantieren.

Damit ein entsprechendes Cache-Verfahren transparent für den Klientencode bleibt, muss die Implementierung der Applikationsserver-Schnittstelle für Anwendungstransaktionen eventuell erweitert bzw. verändert werden. Dies resultiert, wie man später sehen wird, aus den Maßnahmen, die man beim Klient und Server für die Durchführung von Serialisierbarkeitsprotokollen treffen muss.

Für die Effizienz von transaktionalem methodenbasierten Caching benötigt man eine gute Trefferrate. Daher ist bietet es sich an, ein *intertransaktionales* Cache-Verfahren einzusetzen, das heißt, gecachte Methodenresultate, die aus einer Transaktion stammen, sind auch für andere Transaktionen zugänglich (sofern dies die Serialierbarkeit zulässt). Würde man dies nicht tun, wäre die Trefferrate voraussehbar gering und der Methoden-Cache somit nutzlos.

Natürlich ist zu beachten, dass man beim Verwenden vorgehaltener Methodenresultate entfernte (synchrone) Serverzugriffe möglichst vermeidet. Dies garantiert niedrige Antwortzeiten im Falle eines Cache-Treffers und reduziert die Last beim Applikationsserver.

Der vorliegende Bericht enthält keine Evaluation des entwickelten Cache-Verfahrens, allerdings sind ähnliche Leistungsverbesserungen zu erwarten, wie sie [25] experimentell für das nicht-transaktionale Methoden-Caching nachweist. Zumindest untersucht der Bericht einige wichtige Protokolleigenschaften, wie die Cache-Trefferrate und die zu erwartende Transaktionsabbruchrate analytisch.

1.2 Aufbau des Berichts

Weil der Bericht ein transaktionales Cache-Verfahren entwickelt, stellt Kapitel 2 existierende Protokolle vor, die sich mit transaktionalem Caching beschäftigen.

Danach folgen in Kapitel 3 allgemeine Überlegungen zur Architektur eines Applikationsserversystems, das transaktionales Methoden-Caching unterstützt. Dazu beschreibt das Kapitel zunächst die typische Verarbeitung von Anwendungstransaktionen durch Applikationssserver und im Besonderen bei EJB.

In Kapitel 4 geht es darum, die Probleme bei der Serialisierbarkeit entsprechender Anwendungstransaktionen im Detail zu untersuchen. Dazu wird ein klassischer Formalismus der Transaktionstheorie um eine neue Art von Operationen, den Methodenoperationen, erweitert. Diese repräsentieren Zugriffe auf vorgehaltene Methodenresultate als Teil einer Folge von Operationen verschiedener Transaktionen. Die Operationsfolgen werden Methoden-Cache-Historien genannt und müssen ein geeignetes Serialisierbarkeitskriterium erfüllen. Kapitel 4 entwickelt dieses Kriterium und definiert den Serialisierbarkeitsgraphen für Methoden-Cache-Historien, der genau dann azyklisch ist, wenn eine zugehörige Historie serialisierbar ist.

Kapitel 5 konkretisiert die Überlegungen zur Architektur eines Applikationsssservers, der transaktionales methodenbasiertes Caching unterstützt. Zunächst führt das Kapitel in Details der Transaktionsverarbeitung bei modernen Applikationsservern ein. Auf dieser Basis werden Erweiterungsansätze für das Caching mit Hilfe von UML-Klassendiagrammen diskutiert. Der entstehende Entwurf umfasst aber noch keine Serialisierbarkeitsprotokolle, sondern präsentiert lediglich die Infrastruktur, auf der entsprechende Protokollimplementierungen aufsetzen können.

Damit ist sind die Voraussetzungen vorhanden, um in Kapitel 7 Serialisierbarkeitsprotokolle für Methoden-Cache-Historien theoretisch und praktisch zu entwickeln. Es werden insgesamt drei Protokolle vorgestellt. Zunächst wird jeweils die Idee eines Protokolls behandelt, danach folgt eine formale Charakterisierung und Korrektheitsbetrachtung und abschließend wird eine in der Praxis anwendbare Implementierung des Protokolls erörtert.

Daneben handelt Kapitel 6 die so genannten Fehlererholungseigenschaften (Recovery-Eigenschaften) von Methoden-Cache-Historien ab. Diese stellen die Konsistenz im Zusammenhang mit Systemfehlern und abgebrochenen Transaktionen sicher.

Der Bericht schließt mit einer Analyse wichtiger Eigenschaften der vorgestellten Protokolle (Kapitel 8). Dazu zählen ein Vergleich mit Protokolltypen aus der Literatur sowie eine analytische Abschätzung der Cache-Trefferrate und der Transaktionsabbruchrate beim transaktionalen Methoden-Caching.

2 Stand der Forschung beim transaktionalen Caching

2.1 Einführung

Transaktionale (oder genauer intertransaktionale) Cache-Protokolle entstanden im Zusammenhang mit Klient-Server-Datenbanksystemen. Die Idee dabei ist, Seiten (Pages),

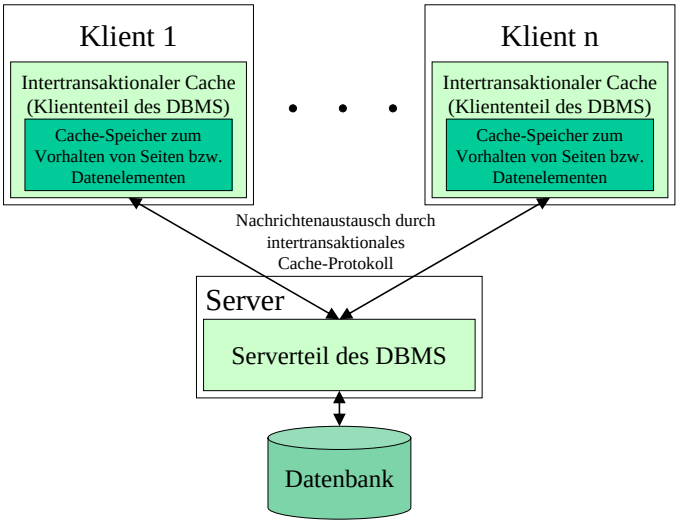

Abbildung 1: Interaktionsschema für einen intertransaktionalen Cache eines Klient-Server-Datenbankverwaltungssystems

also physische Ausschnitte einer Datenbank, zum schnellen Zugriff bei einem Klienten vorzuhalten. Mittels einer Programmbibliothek, die ein Teil des Datenbankmanagementsystems bildet, führt der Klient dann lokal Datenbankoperationen auf einer vorgehaltenen Seite im Rahmen von ACID-Transaktionen aus. Der Server speichert dabei *immer* die letztlich gültige und stabile Version einer Seite. Bevor also die Transaktion eines Klienten abschließt, müssen entsprechende Änderungen auf vorgehaltenen Seiten zum Server gelangt sein.

Die Aufgabe des Cache-Protokolls ist es, (auch) für Zugriffe auf gecachte Seiten transaktionale Konsistenzgarantien, also insbesondere Serialisierbarkeit und Rücksetzbarkeit ([4]), durchzusetzen. Der spezifischere Begriff *intertransaktionales Caching* weist darauf hin, dass Seiten, die im Rahmen einer Transaktionen beim Klienten eingelagert wurden, für nachfolgende Transaktionen wiederverwendet werden können. Abbildung 1 stellt eine entsprechende Referenzarchitektur schematisch dar.

Grundsätzlich repräsentieren intertransaktionale Cache-Verfahren eine Form der Datenbankreplikation (siehe etwa [4]). Ein wichtiger Unterschied zu allgemeinen Replikationsansätzen ist die Tatsache, dass der zu replizierende Datenbankausschnitt beim Caching nicht statisch (etwa durch eine feste Anfrage) festgelegt wird, sondern dynamisch anhand von Zugriffsoperationen beim Klienten entsteht.

Der Hauptunterscheid zu vielen üblichen Replikationsansätzen wurde bereits oben erwähnt: Beim intertransaktionalen Caching besitzt der (einzige) Server die primäre und letztlich gültige Kopie einer Seite oder allgemeiner eines Datenelements. Im Kontext der Literaturbeiträge zur Datenbankreplikation entspricht dies dem so genannten *Primärkopie-Ansatz* (Primary Copy bzw. Mastercopy, [28]).

Allgemeinere Replikationsverfahren unterstellen aber nicht die notwendig die Exi-

stenz einer Primärkopie, sondern behandeln mehrere replizierte Kopien (einer replizierten Datenbank) gleichberechtigt ([4]). Entsprechende transaktionale Replikationsforderungen müssen dann speziell auf diese Anforderung abgestimmt sein. Der vorliegende Bericht entwickelt jedoch ein intertransaktionales Cache-Verfahren und hat wenig mit allgemeineren Replikationsverfahren gemein hat. Letztere werden daher nicht ausführlicher betrachtet.

2.2 Protokolle

Bei Protokollen zum intertransaktionalen Caching geht es natürlich darum, die Leistung des Datenbanksystems insgesamt zu verbessern. Wichtige Voraussetzungen hierfür sind eine gut Cache-Trefferrate, eine geringe Transaktionsabbruchsrate und ein effizienter Nachrichtenaustausch zwischen Klient und Server. Seit Anfang der neunziger Jahre entstand zu diesem Thema eine Art Wettstreit der Forscher um immer leistungsfähigere Protokolle.

Mit dem wohl aktuellsten Beitrag [39] in diesem Bereich existieren grob gesehen neun bedeutende Protokolle, von denen einige noch jeweils zwei bis drei Varianten besitzen. Eine komplette Diskussion und Characterisierung dieser Protokolle würde den Rahmen dieses Beitrags sprengen, zumal die Beiträge [10] und [14] zusammen bereits eine exzellente Übersicht vermitteln.[1] Um eine Idee über entsprechende Verfahren zu vermitteln, werden hier nur drei besonders wichtige Protokolle behandelt:

- Das cachende Zwei-Phasen-Sperrprotokoll (Caching Two Phase Locking, C2PL) ist eines der frühesten intertransaktionalen Cache-Protokolle ([18]) und repräsentiert eine einfache Erweiterung des (strikten) Zwei-Phasen-Sperrprotokolls.

- Das Rückruf-Lese-Sperrprotokoll (Call Back Read Locking, CBR) ist nach [39] das zur Zeit führende und das in der Praxis am meisten verwendete intertransaktionale Cache-Protokoll. Es verbindet gute Systemleistung mit niedrigen Abbruchraten für Transaktionen und ist darüber hinaus leicht zu implementieren. [37] führte dieses Protokoll im Kontext von Klient-Server-Datenbanksystemen ein.

- Das Optimistic Concurrency Control Protokoll (OCC) ist das derzeit beste optimistische (intertransaktionale) Cache-Protokoll. Es hat, grob gesprochen, etwas bessere Leistungseigenschaften als CBR, führt aber bei hoher Transaktionslast zu deutliche höheren Transaktionsabbruchraten. [1] und [14] stellen OCC bzw. eine Variante davon ausführlich vor.

  Abgesehen von seiner allgemeinen Bedeutung ist OCC sehr ähnlich zu einigen transaktionalen Cache-Protokollen, die im Rahmen dieses Beitrags entwickelt werden.

[1] Nur das Protokoll „ADCC“ aus [39] sowie [23] werden darin nicht abgedeckt.

2.2.1 Cachendes Zwei-Phasen-Sperrprotokoll

Das cachende Zwei-Phasen-Sperrprotokoll (C2PL) ist eine Erweiterung des Zwei-Phasen-Sperrprotokolls, bei dem der Klient die in einer Transaktion zu bearbeitenden Seiten zusammen mit entsprechenden Schreib- oder Lesesperren vom Server anfordert. Für Transaktionsoperationen greift der Klient lokal auf die angeforderten Seiten zu und schickt Änderungen beim Abschluss (Commit) der Transaktion zum Server zurück. Erst beim Ende der Transaktion gibt der Klient die angeforderten Sperren wieder frei – die einmal eingelagerten Seiten hält er jedoch weiter lokal vor.

Falls der Klient in einer nachfolgenden Transaktion eine der bereits eingelagerten Seiten benötigt, muss er diese, nicht mehr vom Server anfordern. Es genügt dann eine Sperranforderungen für die Seite und eventuell einen Hash-Wert zum Server zu schicken, wobei der Hash-Wert den Inhalt der lokal vorgehaltenen Seite charakterisiert. Der Server belegt die bei ihm vorliegende Primärkopie der Seite sobald wie möglich mit der angeforderten Sperre. Desweiteren vergleicht er den Hash-Wert der Primärkopie mit dem Hash-Wert, den der Klient gesendet hat. Falls sich die Werte nicht unterscheiden muss er die Sperranforderungen für den Klienten lediglich bestätigen. Andernfalls muss er auch die aktuellste Version der Seite (also den Inhalt der Primärkopie) mit versenden.

Damit der Server etwa eine Schreibsperre für einer Seite nicht mehrmals gleichzeitig vergibt, benötigt er ein Verzeichnis, in der er einträgt, welche Klententransaktion gerade welche Seitensperren besitzt. Ansonsten ist mit dem Verständnis des gewöhnlichen Zwei-Phasen-Sperrprotokolls auch die Korrektheit des cachenden Zwei-Phasen-Sperrprotokolls (in Bezug auf Serialisierbarkeit) unmittelbar einsichtig.

Allgemein können bei allen Sperrprotokollen Deadlocks auftreten. Da der Server mittels des erwähnten Verzeichnisses weiß, welche Transaktionen welche Sperren halten kann er Deadlocks feststellen und bei einem Klienten einen entsprechenden Transaktionsabbruch veranlassen.

2.2.2 Rückruf-Lese-Sperrprotokoll

Während das cachende Zwei-Phasen-Sperrprotokoll im Vergleich zur nicht cachenden Variante die zu erwartende mittlere Größe der Klient-Server-Nachrichten reduziert, so verringert es nicht deren Anzahl. Das Rückruf-Lese-Sperrprotokoll (CBR) versucht diesen Missstand zu beseitigen, indem es Lesesperren auf Seiten auch nach der Beendigung einer entsprechenden Transaktion beim Klienten beibehält. Dahinter steckt die Überlegung, dass der gleiche Klient in naher Zukunft sehr wahrscheinlich wieder die bereits eingelagert Seite liest und somit ohnehin die Lesesperre benötigt.

Falls ein anderer Klient eine Schreibsperre auf einer Seite benötigt, muss der Server alle von anderen Klienten vorgehaltenen Lesesperren für die Seite zunächst „zurückrufen“ (Call Back). Hierzu kontaktiert der Server alle Klienten, die solche Lesesperren besitzen. Ein entsprechender Klient bestätigt die Sperrfreigabe unmittelbar, sofern er die Seite nicht in einer laufenden Transaktion liest. Ansonsten teilt der Klient dem Server mit, dass sich die Lesesperrfreigabe bis zum Ende der laufenden Transaktion verzögert.

Erst am Ende der Transaktion gibt der Klient die Lesesperre frei und teilt dies dem

Server (wiederum) mit. In jedem Fall löscht der Klient dabei auch die vorgehaltene Seite. Zur Bestätigung der Anforderung einer Schreibsperre, wartet der Server seinerseits bis alle betroffenen Klienten ihre Lesesperren auf der entsprechenden Seite freigegeben haben.

Auch bei CBR kann es zu Deadlocks kommen, die der Server wie bei C2PL bestimmt und auflöst. Eine Variante des Rückruf-Lese-Sperrprotokolls ist das Rückruf-Schreib-Lese-Sperrprotokoll (Call Back All Locking, CBA). Bei letzterem behält ein Klient nach einem Transaktionsende auch Schreibsperren auf vorgehaltenen Seiten bei.

2.2.3 Optimistic Concurrency Control Protokoll

Das Optimistic Concurrency Control Protokoll (OCC) ist eine Verbesserung des so genannten Cache Locks Protokolls aus [38]. Es wendet den klassischen, optimistischen Mechanismus der Rückwärtsvalidierung an (Backward Validation, [19]): Zunächst führt das Protokoll alle Schreib- und Leseoperationen einer Transaktion auf lokal beim Klienten vorgehaltenen Seiten durch, ohne diesbezüglich Sperren beim Server anzufordern. Der Klient holt sich vom Server grundsätzlich nur fehlende Seiten aber niemals Sperren! Dies hält die Anzahl der Nachrichten, die zwischen Server und Klient austauschen, gering.

Zum Abschluss einer Transaktion sendet der Klient Informationen über alle gelesenen und geschriebenen Seiten bzw. Datenelemente zum Server. Dort findet eine Rückwärtsvalidierung statt, das heißt, der Server prüft, ob zwischenzeitlich ein entsprechendes Datenelement von einer anderen (bereits abgeschlossenen) Transaktion verändert wurde. Im letzteren Fall bricht der Server die zu validierende Transaktion ab – ansonsten führt er sie zum Abschluss (Commit). Die Validierung ist eine unteilbare Operation, das heißt, zu einem Zeitpunkt kann jeweils nur eine Transaktion validiert werden.

Soweit wie eben beschrieben, würden einmal beim Klienten eingelagerte Seiten niemals aktualisiert werden. Als Folge würden immer mehr Transaktionen bei der Validierung abgebrochen werden, weil Schreiboperationen zwischenzeitlich abgeschlossener Transaktionen viele der vorgehaltenen Seiten ungültig machen. Um dies zu vermeiden, sendet der Server Invalidierungsnachrichten bezüglich veränderter Seiten an alle betroffenen Klienten. Der Server nutzt dazu (wie bei C2PL und CBR) ein Verzeichnis, das Aussagen darüber liefert, welcher Klient welche Seiten vorhält.

Um die Anzahl versendeter Nachrichten gering zu halten schickt der Server Invalidierungsnachrichten nicht einzeln, sondern packt sie zu Nachrichten hinzu, die sowieso zum jeweiligen Klienten gelangen müssen (Piggy Backing). Ein Beispiel für eine letztere Nachricht ist eine versendete Seite, die ein Klient angefordert hat, weil sie nicht in seinem Cache vorliegt. Durch die Invalidierungsnachrichten bleibt der Klienten-Cache immer auf einem relativ aktuellen Stand und dies vermindert die Wahrscheinlichkeit für einen Transaktionsabbruch während der Validierung.

Wenn eine noch laufende Transaktion bei einem Klienten ein Datenelement gelesen hat, für das gerade eine Invalidierungsnachricht eintrifft, dann bricht der Klient die Transaktion sofort ab (Early Abort). Ohnehin würde die Transaktion spätestens bei der

Validierung scheitern, und daher ist es sinnvoll, sie ummittelbar beim Klienten zu beenden.

OCC birgt noch einige Optimierungen – unter anderem können statt Invalidierungsnachrichten auch direkt Änderungsdeltas zur Aktualiserung einer Seite an einen Klienten verschickt werden. (Ähnliche Maßnahmen existieren auch für andere Protokolle.) Weitere Verbesserungen reduzieren den potentiellen Aufwand zum Rückgängigmachen bzw. zum Neustart einer einmal abgebrochenen Transaktion. Im Kontext dieses Berichts sind diese Details aber unerheblich.

2.3 Allgemeine Protokolleigenschaften

Zur Klassifikation transaktionaler Cache-Protokolle auf Basis ihrer Struktur liefert [10] den wichtigsten Beitrag. Franklin und Kollegen stellen in dem Papier eine Taxonomy auf in der sie die sieben bis 1997 bekannten Protokolle und ihre Varianten sinnvoll einordnen können.

Als erstes und wichtigstes Kriterium unterscheidet die Taxonomie *vermeidende* Protokolle (Avoidance Based) und *detektierende* Protokolle (Detection Based). Vermeidende Protokolle garantieren, dass der Inhalt eine Klienten-Caches immer gültig ist, das heißt, der Inhalt einer gecachten Seite entspricht dem Inhalt der entsprechenden Primärkopie auf dem Server. Detektierende Protokolle lassen hingegen ungültige Cache-Inhalte zu. Ob ein Zugriff auf eine entsprechende Seite gültig ist, muss dann im Verlauf einer zugehörigen Transaktion festgestellt werden.

In einem zweiten Schritt klassifiziert [10] detektierende Protokolle daher nach dem Zeitpunkt, an dem die Gültigkeit eines Seitenzugriffs spätestens überprüft wird. Möglich sind eine synchrone Prüfung direkt vor dem Seitenzugriff (On Initial Access, Sychronous), eine asynchrone Prüfung direkt vor dem Zugriff (On Initial Access, Asynchronous) oder eine Prüfung beim Transaktionsende (Deferred Until Commit).

Offenbar zählen sowohl C2PL als auch OCC zu den detektierenden Protokollen, denn beide lassen potentiell ungültige Cache-Inhalte zu (wenn diese auch bei OCC durch Invalidierungsnachrichten nicht lang währen). Bei C2PL wird die Gültigkeit einer Seite bei synchron und direkt vor dem Zugriff, also mit der Anforderung einer entsprechenden Sperre, abgesichert. Bei OCC kann sich die Gültigkeitsprüfung hingegen bis zum Transaktionende (genauer bis zur Validierung) verschieben.

Auf der Seite der vermeidenden Protokolle differenzieren Franklin und Kollegen die Verfahren danach, wann sie die Absicht eines Schreibzugriffs beim Server anmelden. Dies kann einerseits sofort vor der Durchführung der Schreiboperation geschehen oder andererseits beim Transaktionsende. Da sich die Primärkopie einer Seite erst beim Abschluss einer Transaktion verändert, sind beide Ansätze vermeidend, obwohl der letztere als optimistisch gewertet werden kann. Die Kategorisierung nach pessimistischen und optimistischen Protokollen liegt also orthogonal zur Taxonomie von [10].

CBR und CBA zählen zu den vermeidenden Protokollen, da sie nach Konstruktion immer für gültige Cache-Inhalte sorgen. Beide fordern Schreibanforderungen sofort bei der Ausführung der entsprechenden Schreiboperation an. Dies führt ja gerade zu dem oben erläuterten Rückruf (Call back).

Es gibt auch zwei vermeidende Protokolle die Schreibanforderungen an das Transaktionsende verschieben, nämlich O2PL ([7]) und Notify Locks [37]. Sie werden hier aus Platzgründen nicht weiter behandelt.

In [14] kritisiert Gruber das Unterscheidungskriterium „vermeidend versus detektierend“ aus [10] als nicht aussagekräftig genug. Interessant ist dabei vor allem die Untergliederung detektierender Protokolle in reaktiv und passiv: Reaktive Protokolle kümmern sich, wenn auch nicht sofort, darum, dass die Inhalte von Klienten-Caches möglichst aktuell sind. Ein Beispiel hierfür ist OCC mit seinen Invalidierungsnachrichten. Passive Protokolle tun dies hingegen nicht – darunter fällt etwa C2PL.

Eine wichtige Eigenschaft, die *alle* bekannten transaktionalen Cache-Protokollen teilen, ist die Tatsache, dass sie *Operationen von abzuschließenden Transaktionen nur auf der aktuellsten Version eines Datenelements erlauben.* Die aktuellste Version bezieht sich dabei auf den Zustand der Primärkopie, wie er von der zeitlich letzten abgeschlossenen Transaktion erzeugt wurde, die das Datenelement geschrieben hat. Obwohl dies aus zum Durchsetzen der Serialisierbarkeit keineswegs notwendig ist, brechen also alle Protokolle solche Transaktionen ab, die auf eine veraltete Version eines Datenelements zugreifen.

Die eben erörterten Verfahren haben Gemeinsamkeiten mit den transaktionalen Protokollen, die im Folgenden entwickelt werden. Schon jetzt lassen sich aber auch wichtige Unterschiede ausmachen: Während bei den obigen Cache-Protokollen das Vorhalten von Attributwerten oder genauer Seiten im Vordergrund steht, konzentriert sich dieser Bericht auf das Caching von Methodenresultaten.

Im Gegensatz zum vorliegenden Beitrag müssen die oben behandelten Protokolle auch mit Schreibzugriffen auf Cache-Inhalten umgehen. Weiter unterstellen diese Verfahren alle, dass sie direkt in ein Klient-Server-Datenbanksystem integriert werden können. Insbesondere kann ihre Implementierung tief in ein Datenbankverwaltungssystem eingreifen. Diese Annahme ist in Bezug auf das transaktionale Methoden-Caching im Applikationsserver-Bereich nicht haltbar. Dort hat man physisch und organisatorisch getrennte Systeme zur Datenhaltung bzw. zur Verarbeitung von Dienstmethodenaufrufen. Diese wirkt sich auf die Konstruktion entsprechender Transaktionsprotokolle aus und stellt eine zusätzliche Schwierigkeit dar.

Allgemein fällt bei den untersuchten Arbeiten auf, dass sie nur in geringem Maße Formalismen zur Verifikation von Protokolle anführen. Dies wäre aber ein entscheidender Vorteil bei der Entwicklung komplexer Verfahren, deren Korrektheit nicht unmittelbar einsichtig ist.

3 Transaktionales Methoden-basiertes Caching

3.1 Anwendungstransaktionen im Applikationssserver-Bereich

Wenn von einem Klienten eines Applikationsservers aus Dienstmethoden des Servers aufgerufen werden, so gibt es häufig die Anforderung, die Methoden in einem transaktionalen Kontext auszuführen. Die Transaktionen sollen dabei die bekannten ACID-Eigenschaften besitzen.

Um dies zu ermöglichen, delegiert der Applikationsserver die Ausführung einer entsprechenden Transaktion an eine oder mehrere so genannte *transaktionale Ressourcenverwalter* (oder kurz Ressourcenverwalter). Ressourcenverwalter sind Subsysteme des Applikationsservers, die ihrerseits ACID-Transaktionen unterstützen. Bekannte Beispiele hierfür sind etwa (transaktionale) Datenbankverwaltungssysteme.

Falls der Applikationsserver dabei Transaktionen ausführen soll, die sich auf mehrere (unabhängige) Ressourcenverwalter beziehen, spricht man von *verteilten Transaktionen*. Um in diesem Fall die ACID-Eigenschaften zuzusichern, muss der Applikationsserver die Arbeit der verschiedenen Ressourcenverwalter koordinieren. Er übernimmt dazu die Rolle eines *Transaktionsmonitors* ([5]) und schließt eine verteilte Transaktion üblicherweise über das Zwei-Phasen-Commit-Protokoll ab ([13]).[2]

Meist wird hierzu der so genannte X/Open-Standard für verteilte Transaktionen eingesetzt ([35]). Der Standard spezifiziert eine Reihe von Schnittstellen zwischen den beteiligten Systemen, nämlich den Klienten oder Anwendungen, dem Transaktionsverwalter und den Ressourcenverwaltern. Neben dem X/Open-Standard gibt es noch weitere wichtige Spezifikationen für verteilte Transaktionen, wie etwa der CORBA Transaktionsdienst ([33]).

Allgemein werden von einem Applikationsserver unterstützte Transaktionen, die potentiell verteilt sein können, im Folgenden als *Anwendungstransaktionen* bezeichnet. Sie unterscheiden sich wesentlich von den unterliegenden Transaktionen der Ressourcenverwalter, *da sie Anwendungscode von Dienstmethoden umspannen*. Eine Anwendungstransaktion erstreckt sich daher meist auf die Ausführung einer oder mehrerer Dienstmethoden.

Abbildung 2 zeigt das Zusammenspiel der Architekturkomponenten bei der Verarbeitung von Anwendungstransaktionen. Der Klient bzw. die Anwendung ruft Dienstmethoden auf und startet bzw. beendet darüber hinaus eventuell Anwendungstransaktionen, in die die Dienstmethodenaufrufe eingebettet sind. Dazu bietet der Applikationsserver neben der eigentlichen Dienstschnittstelle noch eigens eine Schnittstelle zur Handhabung von Anwendungstransaktionen an. Üblicherweise beinhaltet eine derartige Schnittstelle Methoden zum Beginnen (`begin()`) und Beenden von Anwendungstransaktionen (`commit()`, `rollback()`). Das Java-Interface `UserTransaction` aus dem JTA-Standard (siehe nächster Abschnitt) liefert hierfür ein gutes Beispiel.

Der Applikationsserver integriert einen Transaktionsverwalter, der die verteilten Transaktionen verarbeitet und über eine X/Open-Schnittstelle auf entsprechende lokale Transaktionen bei den Ressourcenverwaltern abbildet. Die Implementierungen der Dienstmethoden sorgen, etwa über Schreib- und Lesezugriffe, für Zustandsveränderungen auf den Ressourcen. Dies spielt sich in den Grenzen einer verteilten Transaktion ab.

Normalerweise besitzt der Applikationsserver kaum eigene Mechanismen zur Transaktionsverarbeitung, die etwa die Eigenschaften Dauerhaftigkeit und Atomizität garantieren, sondern er verlässt sich diesbezüglich auf die Ressourcenverwalter. Des-

[2] Genau genommen können dabei zwar die Eigenschaften Atomizität, Konsistenz und Dauerhaftigkeit zugesichert werden, aber nicht notwendig die Isolation.

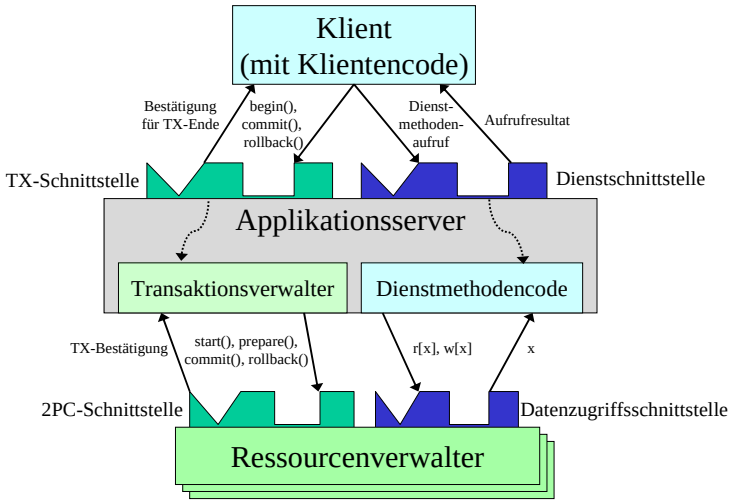

Abbildung 2: Verarbeitung verteilter Transaktionen im Applikationsserver-Bereich

halb kann er auch die Aufgaben zur Bereitstellung dieser Eigenschaften nicht selbst übernehmen. Beispielsweise muss der Zustand von Objekten, auf die sich Dienstmethodenaufrufe aus Anwendungstransaktionen beziehen, „im Wesentlichen“ über die entsprechenden Ressourcenverwalter (wie etwa Datenbankverwaltungssysteme) gespeichert werden. Kämen die Objektzustände etwa (ausschließlich) auf der Halde des Applikationsserver-Prozesses vor, dann könnte das System weder die Dauerhaftigkeit noch die Atomizität zusichern. Bei der Atomizität denke man etwa an den Abbruch einer Anwendungstransaktion, bei der veränderte Objektzustände wieder zurückgesetzt werden müssen (Recovery). Bei gewöhnlichen Laufzeitobjekten auf der Halde wäre der ursprüngliche Zustand verloren und nicht wieder herstellbar.

In Bezug auf das Setzen von Grenzen einer Anwendungstransaktion, also die Demarkation, lassen sich drei Ansätze unterscheiden:

- Applikationsserver-basierte Transaktionsgrenzen: Hier wird der Beginn und das Ende einer Anwendungstransaktion vom Applikationsserver automatisch festgelegt. Die Schnittstelle für Anwendungstransaktionen wird dann nicht benötigt und darf im Klientencode nicht angesprochen werden. Wie der Applikationsserver Transaktionsgrenzen setzt, leitet sich aus seiner Konfiguration ab.

- Dienstmethoden-basierte Transaktionsgrenzen: Hierbei werden die Transaktionsgrenzen *innerhalb* einer Dienstmethode, also durch deren Implementierungscode, gesetzt. Weder ein Klient noch der Applikationsserver haben dann Einfluss auf die Transaktionsgrenzen. Üblicherweise umspannt eine Transaktion dann *nicht* mehrere Dienstmethoden.

- Klientenbasierte Transaktionsgrenzen: In diesem Fall werden die Transaktionsgrenzen explizit im Programmcode eines Klienten bestimmt. Der Klient verwendet

hierzu die oben erwähnte Transaktionsschnittstelle des Applikationsservers.

Alle drei Typen von Anwendungstransaktionen kommen in realen Applikationsserver-Technologien wie etwa EJB und .NET vor.

Im Kontext von Enterprise Java Beans (EJB) fasst das Java Transaction API (JTA, [32]) eine Menge von Schnittstellen zusammen, die die wichtigsten Teilnehmer bei der Bearbeitung verteilter Transaktionen betrifft:

- Die Benutzertransaktionsschnittstelle (`UserTransaction` Interface) dient zum Starten und Beenden von Transaktionen aus Java-Anwendungen heraus. In Bezug auf Applikationsserver kann es sich dabei sowohl um Klientencode als auch um Implementierungscode für Dienstmethoden handeln.

- Die Transaktionsverwalterschnittstelle (`TransactionManager` und `Transaction` Interface) ermöglicht die Kontrolle eines X/Open-konformen Transaktionsverwalters aus einem Java-System heraus. Ein EJB-basierter Applikationsserver kann damit beispielsweise einen Verwalter für verteilte Transaktionen als separate Komponente integrieren und steuern.

- Die XA-Ressourcenschnittstelle (`XAResource` und `Xid` Interface) erlaubt schließlich die Durchführung von Transaktionen über X/Open-konforme Ressourcenverwalter mit Hilfe des Zwei-Phasen-Commit-Protokolls. Die Schnittstelle wird üblicherweise von einem (in Java programmierten) Transaktionsmonitor genutzt. `Xid`-Instanzen repräsentieren hierzu Identifikatoren von Transaktionen, die in einer XA-Ressource verarbeitet werden sollen. Der Transaktionsmonitor erzeugt für jede verteilte Transaktion eine eigene `Xid` und gibt diese über die `XAResource`-Schnittstelle an beteiligte Ressourcenverwalter weiter. Die Ressourcenverwalter ordnen dem Identifikator ihrerseits eine lokale Ressourcentransaktion eindeutig zu.

Zur Demarkation von Anwendungstransaktion, sind bei EJB alle oben genannten Alternativen zum Setzen von Transaktionsgrenzen verwirklicht. Sie können pro Dienstmethode konfiguriert werden.

Für Applikationsserver-basierte Transaktionsgrenzen (Container Managed Transactions) bietet EJB sechs Optionen an. Beispielsweise startet der Applikationsserver mit der Option `RequiresNew` automatisch eine neue Anwendungstransaktion beim Aufruf einer Dienstmethode und beendet die Transaktion zum Ende der Methodenausführung. Die Option `Requires` fordert hingegen lediglich, dass ein Dienstmethodenaufruf in eine (möglicherweise bereits bestehende) Transaktion eingebunden ist.

Zur Demarkation von Dienstmethoden-basierten Transaktionen (Bean Managed Transactions) und Klienten-basieren Transaktionen nutzt man als Anwendungsprogrammierer das Java-Interface `UserTransaction` des JTA-Standards. Abbildung 3 zeigt Beispielcode für eine klientenbasierte Transaktion. Eine Instanz der `UserTransaction`-Klasse erhält der Klient dabei vom Applikationsserver über JNDI.

```
1  ...
2  Context ctx = new InitialContext();
3  // Fordere eine Anwendungstransaktion an
4  UserTransaction utx =
5    (UserTransaction)ctx.lookup("java:comp/UserTransaction");
6
7  // Starte die Anwendungstransaktion
8  utx.begin();
9  // Führe Dienstmethoden (aus EJBs) aus ...
10 List subscribers = mouse.getSubscriberList();
11 if (!subscribers.contains(dan)) {
12   dan.addSubscription(mouse);
13 }
14 // Beende die Anwendungstransaktion
15 utx.commit();
16 ...
```

Abbildung 3: Beispielcode für klientenbasierte Anwendungstransaktionsgrenzen bei EJB

3.2 Architektur eines Applikationsservers mit transaktionalem Methoden-Cache

Die Sicherstellung der Serialisierbarkeit von Anwendungstransaktionen eine der größten Herausforderungen im Zusammenhang mit transaktionalem Methoden-Caching. Vor der Architektur eines Appplikationsserversystems mit transaktionalem Methoden-Cache sollen deshalb die Systemkomponenten, welche Serialisierbarkeit sicherstellen, auf abstrakter Ebene beleuchtet werden.

Üblicherweise ist innerhalb eines Ressourcenverwalters ein (eigener) Transaktionsverwalter für die Abarbeitung von Transaktionen verantwortlich. Alle anderen Systemteile des Ressourcenverwalters sind letzlich dem Transaktionsverwalter unterstellt. Vereinfacht dargestellt, bietet der Transaktionsverwalter gemäß [4] eine Schnittstelle nach außen mit den folgenden Operationen an:

- Eine Leseoperation liest Datenelemente aus der verwalteten Ressource im Kontext einer Transaktion und gibt den Wert an einen Klienten des Ressourcenverwalters weiter.[3] Eine entsprechende Operation kann durch den Ausdruck $r_i^j[x]$ formalisiert werden. i bezeichnet die Nummer oder Kennung der Transaktion T_i und x das gelesene Datenelement. r zeigt an, dass es sich um eine Leseoperation handelt. Der Index j gruppiert schließlich eine Menge von Leseoperationen aus der Transaktion T_i. Die Bedeutung von j wird später genauer erläutert.

- Eine Schreiboperation verändert ein Datenelement x im Rahmen einer Transaktion T_i und wird mit $w_i[x]$ notiert. Der geschriebene Wert selbst ist im Rahmen der

[3] Eine Ressource wie etwa eine Datenbank kann viele Datenelemente enthalten. Ein Datenelement entspricht dann beispielsweise einem Satz aus der Datenbank.

folgenden Betrachtungen unerheblich und wird deshalb nicht aufgeschrieben. (Es zählt nur, dass x durch die Transaktion T_i verändert wird.)

- c_i beschreibt das Abschließen (Commit) der Transaktion T_i. Alle Veränderungen der Transaktion werden in der Ressource wirksam und werden dauerhaft gespeichert. Die veränderten Werte können von nachfolgenden Transaktionen gelesen und auch wieder verändert werden.

- a_i beschreibt den Abbruch (Abort) einer Transaktion. Alle Veränderungen der Transaktion bleiben in der Ressource unwirksam bzw. werden zurückgenommen und bleiben/werden *nicht* dauerhaft gespeichert. Die durch Transaktion T_i veränderten Werte können von anderen Transaktionen (die nicht selbst abbrechen) nicht gelesen werden. Stattdessen lesen nachfolgende Transaktionen die Werte der Datenelemente, so wie sie vor der Ausführung von i bestanden.

Damit ein Klient eines Ressourcenverwalters weiß, dass $w_i[x]$, c_i bzw. a_i erfolgreich waren, erhält er entsprechende Bestätigungen ($w_i[x]^{ack}$, c_i^{ack}, a_i^{ack}). vom Transaktionsverwalter. Falls eine Operation nicht erfolgreich war, wird die zugehörige Transaktion abgebrochen und der Transaktionsverwalter sendet ein a_i^{ack} zum jeweiligen Klienten.[4]

Der Transaktionsverwalter gibt die bei ihm ankommenden Operationen an den so genannten *Planer* (Scheduler) weiter, der die Operationen so ordnet, dass alle Transaktionen, die jemals abschließen bzw. abgeschlossen haben, serialisierbar sind. Dazu hat der Planer die Möglichkeit, eine ankommende Operation zurückzuweisen (und somit die entsprechende Transaktion abzubrechen), die Operation zu bestätigen, die Verarbeitung der Operation zu verzögern oder die Operation an den so genannten Datenverwalter weiterzugeben. Der Datenverwalter schließlich führt Schreib- und Leseoperationen auf den Datenelementen einer Ressource aus. Das Verfahren, dass der Planer verwendet, um Operationen zu bearbeiten, wird als *Serialisierbarkeitsprotokoll* oder einfach Protokoll bezeichnet.

Um die Serialisierbarkeit von Transaktionen sicherzustellen, muss der Planer vor allem auf die Ordnung von Operationen achten, die zueinander *in Konflikt* stehen. Zwischen zwei (ankommenden oder bereits verarbeiteten) Operationen besteht genau dann ein Konflikt, wenn beide aus verschiedenen Transaktionen stammen, auf das gleiche Datenelement zugreifen und wenigstens eine von ihnen eine Schreiboperation ist. Mit anderen Worten: Für die beiden Operationen muss $w_i[x]$, $w_j[x]$ oder $w_i[x]$, $r_j[x]$ mit $i \neq j$ gelten.

Im Zusammenhang mit dem transaktionalen Methoden-Caching reicht es aber nicht aus, lediglich Operationen der Form $w_i[x]$ und $r_i[x]$ für die Serialisierbarkeit zu berücksichtigen. Bezüglich klientenbasierten Anwendungstransaktionen muss man auch ausdrücken können, dass innerhalb einer Transaktion T_i ein Resultat aus dem Methoden-Cache gelesen wurde. Der Planer muss dann sicherstellen, dass der Wert des Methodenresultats zu den anderen Werten, die in der Anwendungstransaktion verarbeitet wurden, „passt". „Passen" bedeutet hier vereinfachend, dass die Transaktion mit Methoden-

[4] Im Applikationsserver-Bereich wäre gemäß Abbildung 2 der Applikationsserver selbst ein entsprechender Klient.

Cache genauso verläuft, als wäre sie ohne Verwendung des Methoden-Cache durchgeführt worden. Die Werte, die der Klient innerhalb der Anwendungstransaktion vom Methoden-Cache erhält, sollen daher die gleichen sein, wie die, die er verarbeitet hätte, wenn die entsprechenden Methodenaufrufe zum Applikationsserver delegiert worden wären.

Da es dem Planer obliegt, die Serialisierbarkeit von Anwendungstransaktionen zu garantieren, sollte dieser auch Informationen darüber erhalten, welche Methodenresultate innerhalb einer Transaktion aus dem Methoden-Cache gelesen wurden. Daher wird die Schnittstelle des Transaktionsverwalters und des Planers so erweitert, dass beide so genannte Methodenoperationen oder m-Operationen verarbeiten können. Methodenoperationen repräsentieren also die Verwendung eines im Cache vorgehaltenen Methodenresultats durch einen Dienstmethodenaufruf im Klienten. Sofern das Protokoll des Planers darauf ausgelegt ist, kann er dann dafür sorgen, dass auch Transaktionen mit Methodenoperationen serialisierbar sind.

Eine Methodenoperationen wird durch den Ausdruck $m_i^{k,l}$ formalisiert, wobei i die Nummer der Transaktion bezeichnet, in der das entsprechende, gecachte Methodenresultat verwendet wird. k bezeichnet die Nummer jener Transaktion, innerhalb derer das Methodenresultat berechnet und in den Cache eingelagert wurde. Die Transaktion T_k hat dabei den Methodenaufruf auf dem Server ausgeführt und eine Menge von Leseoperationen beim Ressourcenverwalter getätigt. $m_i^{k,l}$ bezieht sich gerade auf diese Menge von Leseoperationen aus der Transaktion T_k. l zeigt an, um welche Operationen der Form $r_k^l[x]$, es sich dabei handelt. Dies erklärt den bereits erwähnten hochgestellten Index l für eine Leseoperation $r_k^l[x]$. Wenn sich eine m-Operation $m_i^{k,l}$ auf mehrere Leseoperationen bezieht, hat man in der Transaktion T_k also etwa Operationen der Form $r_k^l[x]$, $r_k^l[y]$ etc.

Zustandsverändernde Methodenaufrufe müssen *immer* zum Applikationsserver delegiert werden. Das Vorhalten entsprechender Methodenresultate macht also keinen Sinn. Wenn, wie bei transaktionalen Dienstmethoden üblich, der Zustand der Serverseitigen Dienstobjekte über einen Ressourcenverwalter (also etwa in einer Datenbank) gespeichert wird, lassen sich lesende und schreibende (also zustandsverändernde) Methodenaufrufe leicht unterscheiden: Lesende Methodenaufrufe erzeugen lediglich Operationen der Form $r_k^l[x]$ beim Transaktionsverwalter; schreibende Methodenaufrufe produzieren hingegen auch Operationen der Form $w_i[x]$. Ein entsprechend erweiterter Transaktionsverwalter kann diese Information dem Methoden-Cache zukommen lassen.

Der komplette Zustand eines Applikationsservers wird also über die Ressourcenverwalter (in Form von transaktionalen Ressourcen) repräsentiert. Alle Zustandsveränderungen erreichen die Ressourcenverwalter bzw. den Transaktionsverwalter und können so an einer zentralen Stelle für das Konsistenzprotokoll des Methoden-Caches beobachtet werden.

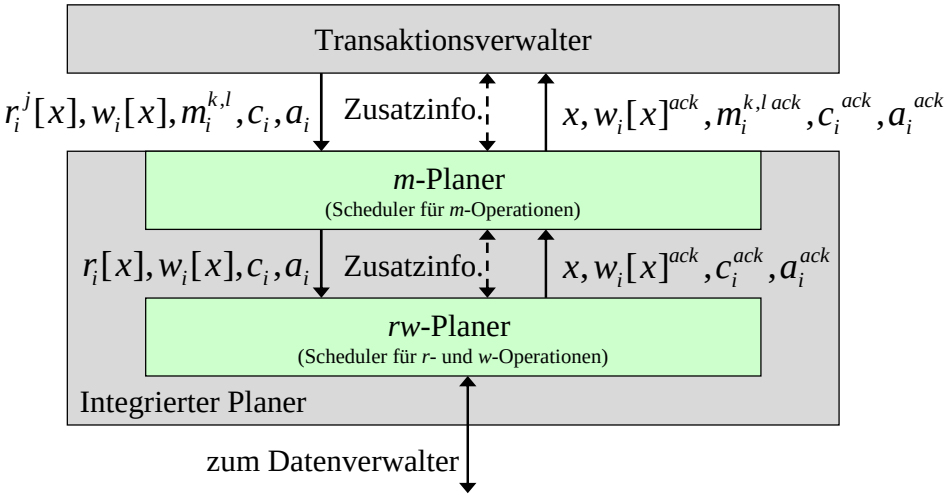

Abbildung 4: Abstrakte Architektur für die Integrationen von m-Planer und rw-Planer

In diesem Bericht wird davon ausgegangen, dass die Zustände und Zustandsveränderungen, von denen Dienstmethoden abhängen, dem Methoden-Cache im Rahmen der Verarbeitung einer Anwendungstransaktion mitgeteilt werden können.

Wie bereits beschrieben, hat der Planer unter anderem die Aufgabe, zueinander in Konflikt stehende Lese- und Schreiboperationen so zu ordnen, dass Serialisierbarkeit erreicht wird. Mit der Erweiterung der Schnittstelle des Planers muss dieser nun auch Methodenoperationen berücksichtigen und gemäß einem noch zu erarbeitenden Serialisierbarkeitskriterium ordnen. Eine Methodenoperation kann, wie man in Abschnitt 4 noch sehen wird, zu bestimmten Schreiboperationen in Konflikt stehen. Ein Planer, der sowohl Methodenoperationen als auch Schreib- und Leseoperationen berücksichtigt, wird im Folgenden als *integrierter Planer* bezeichnet.

Die Anpassung eines Planers (und Transaktionsverwalters) zur Berücksichtigung von Methodenoperationen ist vor allem in der Praxis eine schwierige oder gar unmögliche Aufgabe, denn beide Komponenten sind meist fest in einen Ressourcenverwalter (wie etwa ein Datenbankverwaltungssystem) eingebettet und für derartige Eingriffe unzugänglich. Eine gangbare Alternative ist die *Zerlegung eines integrierten Planers* in einen so genannten m*-Planer* und einen rw*-Planer*. Während der rw-Planer sich wie gewohnt um die Ordnung von konfliktbehafteten Lese- und Schreiboperation kümmert, behandelt der m-Planer ausschließlich Konflikte zwischen Methoden- und Schreiboperationen. Der m-Planer kann dann eventuell sogar außerhalb des eigentlichen Ressourcenverwalters implementiert werden und als eine (transparente) Schicht zwischen Applikationsserver und Ressourcenverwalter liegen.

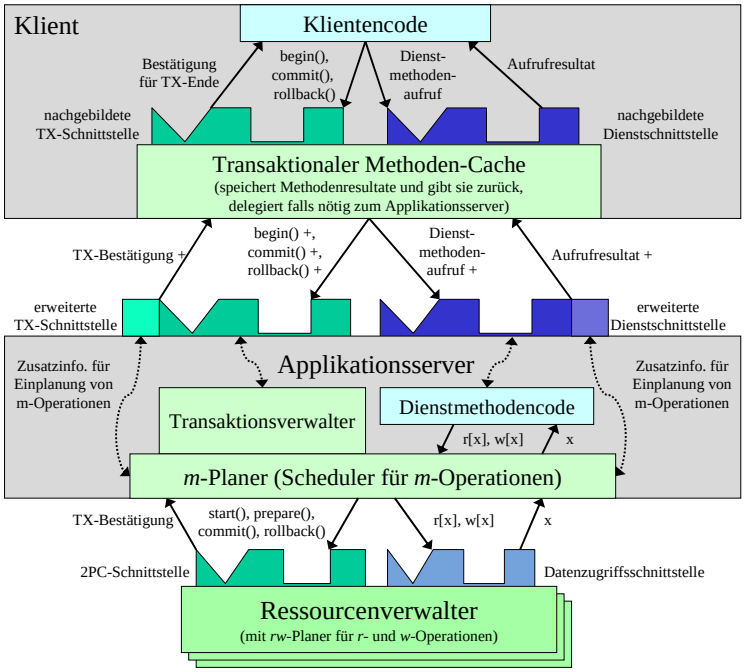

Abbildung 5: Architektur für transaktionales methodenbasiertes Caching mit externem m-Planer

Diese Art der Zerlegung von integrierten Planern führt implizit auch zu einer Zerlegung des Protokolls, welches Serialisierbarkeit gewährleistet. Allerdings ist nicht automatisch garantiert, dass die kooperierenden Planer auch tatsächlich serialisierbare Methoden-Cache-Historien produzieren. Dies muss unter Berücksichtigung der speziellen Protokolle des m-Planers und des rw-Planers explizit bewiesen werden.

Neben den erwähnten praktischen Aspekten erleichtert die erläuterte Zerlegung von integrierten Planern sogar die theoretische Entwicklung von Protokollen: Man kann nämlich bereits existierende Protokolle für rw-Planer, verwenden und muss lediglich zeigen, dass sie zusammen mit einem (neu entwickelten) m-Planer-Protokoll wie gefordert zusammenarbeiten.

Die Idee der Zerlegung von Planern wurde bereits in [4] (genauer in [4] Abschnitt 4.5) für rw-Planer behandelt. Die vorliegende Arbeit übernimmt also das in [4] vorgeschlagene, allgemeine Prinzip.

Abbildung 4 fasst die in den vorigen Abschnitten getroffenen Aussagen zu Planern schematisch zusammen. Sie zeigt die Zerlegung eines integrierten Planers in m- und rw-Planer, wobei lediglich der m-Planer die erweiterte Schnittstelle zur Verarbeitung von Transaktionsoperationen besitzt. Es ist zu beachten, dass der m-Planer selbstständig Transaktionen abbrechen kann, falls diese nicht serialisierbar sind. Daher kann es etwa vorkommen, dass der m-Planer trotz einer eingehenden Operation c_i ein a_i an den rw-Planer weiterreicht (und ein a_i^{ack} an den Aufrufer zurückgibt). Neben den Operationen

und Operationsbestätigungen können die Planer eventuell noch Zusatzinformationen austauschen, die für die Kooperation notwendig sind. Ein Beispiel sind Zeitstempel für Transaktionen, sofern beide Planer ein Zeitstempelprotokoll anwenden.

Man beachte: Für eine erfolgreiche Kooperation zwischen m-Planer und rw-Planer ist es unerlässlich, *dass der m-Planer sämtliche Transaktionsoperationen des rw-Planers beobachtet*. Bezüglich Abbildung 4 umfasst dies die Operationen $r_i[x]$, $w_i[x]$, c_i und a_i. Würde der m-Planer nicht alle solche Operationen beobachten, so könnte er zum Beispiel manche Konflikte zwischen m- und w-Operationen nicht erkennen und keine Serialisierbarkeit garantieren.

Nachdem nun die Architektur eines Transaktionsverwalters und integrierten Planers auf abstrakter Ebene diskutiert wurde, wendet sich der Rest dieses Abschnitts der Gesamtarchitektur eines Applikationsserversystems mit transaktionalem Methoden-Cache zu.

Abbildung 5 zeigt, wie ein entsprechendes System realisiert werden kann, wenn der m-Planer, wie oben bereits angesprochen, als separate Schicht außerhalb des bzw. der Ressourcenverwalter implementiert ist. Es handelt sich um eine Erweiterung der Architektur aus Abbildung 2, die den Methoden-Cache transparent auf der Klientenseite einfügt. Neben der Dienstschnittstelle bildet der Methoden-Cache auch die Schnittstelle für Anwendungstransaktionen nach und behandelt die Operationen zum Beginnen, Abschließen bzw. Abbrechen einer Anwendungstransaktion.

Je nach eingesetztem Serialisierbarkeitsprotokoll, werden die Operationen vor der Delegation zur Transaktionsschnittstelle des Applikationsservers eventuell mit Zusatzinformationen angereichert. Beim `commit()`-Aufruf könnte dies etwa eine Liste der innerhalb der abzuschließenden Transaktion verwendeten Methodenresultate aus dem Methoden-Cache sein. Die Transaktionsschnittstelle des Applikationsservers muss die zusätzlichen Informationen aufnehmen können und deshalb eventuell selbst erweitert werden. Die Zusatzinformationen, wie etwa die erwähnte Liste von Methodenresultaten, werden an den m-Planer weitergereicht, damit dieser die Serialisierbarkeit in Bezug auf Methodenoperationen sicherstellen kann. Die Resultate von Methoden aus der Transaktionsschnittstelle können auf ähnliche Weise und aus ähnlichen Gründen erweitert sein.

Moderne Applikationsserver wie etwa JBoss ([17]) ermöglichen darüber hinaus eine Erweiterung der Dienstmethodenschnittstelle, so dass bei jedem Dienstmethodenaufruf in transparenter Weise Zusatzinformationen zwischen Klient und Applikationsserver ausgetauscht werden können.

Es soll betont werden, dass die dargestellte Architektur in der Tat mit existierenden Applikationsserver-Produkten realisierbar ist. Dies wurde im Rahmen einer prototypischen Implementierung verifiziert.

Die Architektur aus Abbildung 6 unterstellt alternativ, dass der m-Planer direkt in den Ressourcenverwaltern integriert ist. Der integrierte Planer wird mit den für das Serialisierbarkeitsprotokoll notwendigen Zusatzinformationen direkt vom verteilten Transaktionsmanager des Applikationsservers versorgt. Ansonsten ähneln sich die Architekturen aus Abbildung 6 und 5.

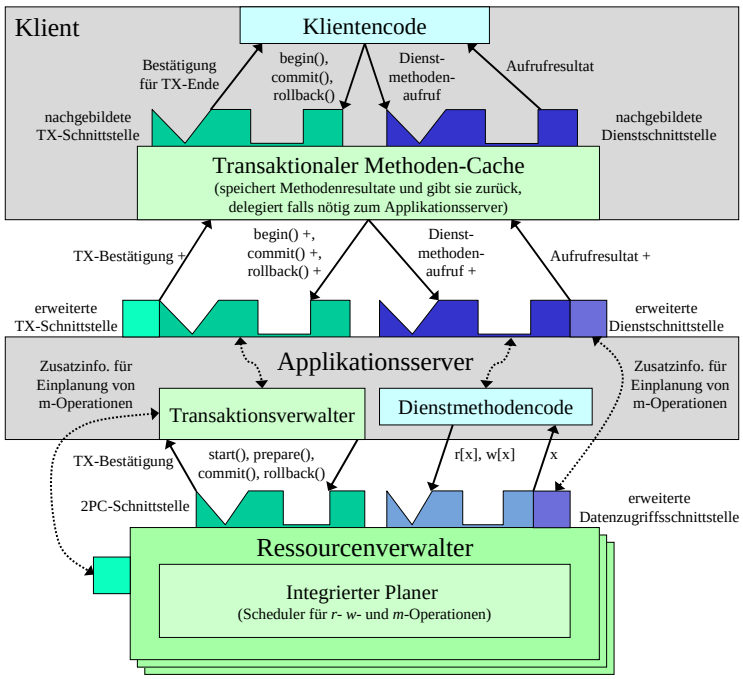

Abbildung 6: Architektur für transaktionales methodenbasiertes Caching mit m-Planer im Ressourcenverwalter

Bezüglich der weiter oben geforderten Beobachtbarkeit von Transaktionsoperationen durch den m-Planer ist vor allem die Architektur aus Abbildung 5 kritisch. Hier könnte es vorkommen, dass außer dem Applikationsserver auch andere Prozesse auf einen Ressourcenverwalter zugreifen und somit Transaktionen ohne Kenntnis des m-Planers ausführen. Für ein solches Szenario sollte man fordern, dass der Ressourcenverwalter den m-Planer aktiv über entsprechende Operationen benachrichtigt. Konventionelle Datenbankverwaltungssysteme bieten dafür allerdings keine geeignete Schnittstelle an. Ein möglicher Ausweg ist ein Proxydienst, der allen Klientenzugriffen auf das Datenbankverwaltungssystem vorgeschaltet ist und die beim Datenbankverwaltungssystems ein- sowie ausgehenden Nachrichten geeignet untersucht.

4 Serialisierbarkeitstheorie

4.1 Einführung

Im Folgenden wird eine Theorie für serialisierbare Transaktionshistorien entwickelt, die neben gewöhnlichen Schreib- und Leseoperationen auf Datenelementen auch Aufrufe bezüglich gecachter Methodenresultate berüchsichtigt. Hierzu wird der Begriff der Transaktion und der Historie, wie er aus [4, 36] bekannt ist, um Methodenoperationen der Form $m_i^{k,l}$ erweitert. Für die dadurch entstehenden *Methoden-Cache-Historien* leitet

das Kapitel einen Serialisierbarkeitsbegriff und ein Serialisierbarkeitstheorem her.

4.1.1 Idee

Wie bereits in Abschnitt 3.2 erwähnt, repräsentiert eine Methodenoperation $m_i^{k,l}$ den Zugriff auf ein Resultat aus dem Methoden-Cache. Genauer bezieht sich $m_i^{k,l}$ etwa auf eine Leseoperation der Form $r_k^l[x]$, die irgendwann zuvor der Applikationsserver (bzw. ein unterstellter Ressourcenverwalter) ausgeführt hat. Durch eine Schreiboperation $w_j[x]$, die nach $r_k^l[x]$ stattfindet, kann aber der Wert von x, auf den sich $m_i^{k,l}$ bezieht, bereits verändert worden sein. Sofern das entsprechende Resultat noch im Methoden-Cache vorkommt, kann $m_i^{k,l}$ dabei sogar nach $w_j[x]$ in einer Methoden-Cache-Historie stehen.

Mit Hilfe der Mehr-Versionen-Transaktionstheorie aus [3, 4, 36] lassen sich solche Ereignisse gut durch verschiedene *Versionen* von x charakterisieren. Im obigen Beispiel liest dann etwa $r_k^l[x]$ eine Version x_h von x, also $r_k^l[x_h]$. $w_j[x]$ schreibt hingegen eine neue Version x_j, also $w_j[x_j]$. Weiter kann man $m_i^{k,l}$ als einen Lesezugriff auf x_k interpretieren und in einer Mehr-Versionen-Historie durch $r_i[x_k]$ *ersetzen*, denn $m_i^{k,l}$ spiegelt ja gerade das Ergebnis einer Berechnung wider, bei der x_k gelesen wurde.

Ein Ziel dieses Kapitels ist es, ein Serialisierbarkeitskriterium für Methoden-Cache-Historien zu entwickeln. Aus der Beobachtung von eben folgt die Idee, dieses Ziel mit Hilfe der Mehr-Versionen-Transaktionstheorie zu erreichen. Allgemein bildet man dazu $m_i^{k,l}$ auf eine (oder eventuell mehrere) r-Operationen ab, die geeignete Versionen von Datenelementen lesen. Dies verleiht der Methodenoperation $m_i^{k,l}$ implizit eine *Semantik*.

Mit Schreib- und Leseoperationen der Form $r_i^l[x]$ bzw. $w_j[x]$ kann man, wie im Beispiel angedeutet, analog vorgehen. Weiter unten wird dazu die Abbildung MV', definiert, die komplette Methoden-Cache-Historien in Mehr-Versionen-Historien übersetzt. MV' soll die Menge der Konflikte, die zwischen Operationen aus der Methoden-Cache-Historie bestehen, (im Wesentlichen) im Bild erhalten. Auf diese Weise leitet sich ein Serialisierbarkeitskriteriums für eine Methoden-Cache-Historie aus der Serialisierbarkeit des zugehörigen Bildes unter MV' ab.

Durch eine reine Einbettungsabbildung würden aber bestimmte Konflikte in der zugeordneten Mehr-Versionen-Historien verloren gehen. Wenn MV' beispielsweise zwei Operationen $w_i[x] < w_j[x]$ lediglich auf $w_i[x_i] < w_j[x_j]$ abbilden würden, ginge der ursprüngliche Konflikt zwischen den beiden Operationen verloren.[5]

Als „Trick“ fügt MV' in diesem Fall noch eine künstliche Leseoperation ein, die in der ursprünglichen Methoden-Cache-Historie gar nicht vorkommt. Für das Beispiel von eben erhält man als Abbildungsergebnis (im Wesentlichen) $w_i[x_i] < r_j[x_i] < w_j[x_j]$.

Weiter unten wird gezeigt, dass eine Methoden-Cache-Historie und ihr Bild unter MV' (bis auf Transitivität) den gleichen Serialierbarkeitsgraphen besitzen. Daher kann man MV' in der Tat zur Definition des gewünschten Serialisierbarkeitskriteriums heranziehen: Eine Methoden-Cache-Historie definiert man genau dann als serialisierbar, *wenn ihr Bild unter MV', einen azyklischen Mehr-Versionen-Serialisierbarkeitsgraphen besitzt*.

[5] In Kapitel 5.2 von [36] beschreiben die Autoren ein derartige Einbettungsabbildung. Die Bildhistorie wird dort als „Monoversion Schedule“ bezeichnet.

Obwohl sich der Formalismus für Methoden-Cache-Historien wesentlich auf die Mehr-Versionen-Theorie abstützt, leitet diese Kapitel ein Serialisierbarkeitskriterium für besondere Ein-Versionen-Historien, nämlich den Methoden-Cache-Historien her. Warum sollte man stattdessen nicht direkt auf die Mehr-Versionen-Theorie, also ohne Verwendung des Ein-Versionen-Ansatzes, zurückgreifen? Dagegen sprechen drei Argumente:

- Zunächst ist es im Zusammenhang mit transaktionalem Methoden-Caching interessant, die allgemeinen theoretischen Eigenschaften für entsprechende Historien herauszuarbeiten. Hierfür liefert der gewählte Ein-Versionen-Formalismus einen intuitiven Ausgangspunkt, denn er resultiert direkt aus dem Architekturmodell für das Cache-Verfahren.

- Für die Zerlegung eines integrierten Planers in m-Planer und rw-Planer bieten Methodenoperationen einen guten Abstraktionsansatz. Das Protokoll für den m-Planer kann dann mit möglichst geringem Wissen über die interne Funktionsweise des rw-Planerprotokolls (wie etwa der Behandlung von Versionen) entwickelt und implementiert werden. Ohnehin ist der nachfolgende Formalismus direkt einsetzbar, wenn der rw-Planer ein Ein-Versionen-Protokoll nutzt.

4.1.2 Aufbau des Formalismus

Der im Folgenden behandelte Formalismus orientiert sich stark an [4] – einem Standardwerk zur Transaktionstheorie für Datenbanken. Durch den Rückgriff auf die dort etablierte Notation sollen dem Leser große Eingewöhnungseffekte beim Lesen erspart bleiben. Es ermöglicht aber auch eine leichte Abgrenzung der hier vorgestellten Ergebnisse zu bereits bekanntem Wissen.[6]

Damit der vorliegende Bericht in sich abgeschlossen ist, muss allerdings ein Repertoire von Begriffen, die bereits aus der Literatur bekannt sind, (nochmals) formal definiert werden. Dies umfasst:

- gewöhnliche Ein-Versionen-Transaktionen und -Historien (die hier als rw-Transaktionen bzw. rw-Historien bezeichnet werden),

- Serialisierbarkeitsgraphen für rw-Historien und

- Mehr-Versionen-Historien und entsprechende Mehr-Versionen-Serialisierbarkeitsgraphen.

Um andererseits unnötig viele, einführende Definitionen zu vermeiden, wurde versucht, die Menge derartiger Begriffe möglichst klein zu halten. Beispielsweise formalisiert der Bericht weder die Serialisierbarkeit von rw-Historien noch die von Mehr-Versionen-Historien. Glücklicherweise konnte zur Herleitung der Resultate für das

[6] Alternativ behandelt [36] kompakt und umfassend wichtige Transaktionsmodelle und -formalismen aus moderner Sicht.

Methoden-Caching darauf verzichtet werden – lediglich am Ende nimmt das Kapitel kurz auf die Serialisierbarkeit von rw-Historien Bezug.

Der Rest des Kapitels ist wie folgt aufgebaut: In Abschnitt 4.2 werden Methoden-Cache-Transaktionen und entsprechende Historien vorgestellt, die sich von gewöhnlichen Ein-Versionen-Transaktionen bzw. -Historien hauptsächlich durch Operationen der Form $m_i^{k,l}$ unterscheiden (Definition 1 und 2). In der Tat kann man gewöhnliche Transaktionen und Historien als spezielle Methode-Cache-Transaktionen bzw. -Historien auffassen. (Dies wird in Definition 5 aus Abschnitt 4.3 vermittelt.)

Wie bereits angedeutet, ist es ein Ziel dieses Kapitels, einen Serialisierbarkeitsbegriff für Methoden-Cache-Historien zu entwickeln und ein entsprechendes Serialisierbarkeitstheorem bezüglich eines Serialisierbarkeitsgraphen aufzustellen.

Auf dem Weg dorthin werden in Abschnitt 4.4 rw-Historien auf *äquivalente* Mehr-Versionen-Historien abgebildet. Äquvialenz bedeutet hierbei, dass einerseits die Operationen und die Ordnung von Ursprung und Bild im Wesentlichen gleich sind. Andererseits sind aber auch der Serialisierbarkeitsgraph der Ein-Versionen-Historie und der Mehr-Versionen-Serialisierbarkeitsgraph des Bildes bis auf Transitivität identisch. Dies ist die Aussage von Satz 1.

Mit Hilfe der zuvor eingeführten Abbildung MV (aus Definition 10) kann man die Ein-Versionen-Serialisierbarkeitstheorie als einen Spezialfall der Mehr-Versionen-Serialisierbarkeitstheorie auffassen. Um Satz 1 zu formulieren und zu beweisen, müssen zunächst Mehr-Versionen-Historien und Serialisierbarkeitsgraphen für rw- und Mehr-Versionen-Historien eingeführt werden (Definition 7, 6 und 9 aus Abschnitt 4.3).

In Abschnitt 4.5 erweitert Definition 11 dann die Abbildung von rw-Historien mittels MV auf die Abbildung von Methoden-Cache-Historien mittels MV'. Dadurch lässt sich der oben erwähnte Serialisierbarkeitsbegriff für Methoden-Cache-Historien entwickeln: Eine Methoden-Cache-Historie ist genau dann serialisierbar, wenn ihr Bild unter MV' einen azyklischen Mehr-Versionen-Serialisierbarkeitsgraphen besitzt (Definition 12).

Definition 13 bestimmt die direkte Struktur des Serialisierbarkeitsgraphen $MCSG$ einer Methoden-Cache-Historie. Mit Satz 2 und Korollar 2 wird dann gezeigt, dass eine Methoden-Cache-Historie in der Tat genau dann serialisierbar ist, wenn der zugehörige Serialisierbarkeitsgraph $MCSG$ azyklisch ist. Beweisen kann man dies mit Hilfe der Abbildung MV' und dem Mehr-Versionen-Serialisierbarkeitsgraphen für Mehr-Versionen-Historien. Es gilt insbesondere: Der Serialisierbarkeitsgraph $MCSG$ einer Methoden-Cache-Historie $(H^{MC}, <)$ ist (für eine vorgegebene Versionsordnung) bis auf Transitivität identisch zum Mehr-Versionen-Serialisierbarkeitsgraphen von $MV'((H^{MC}, <)$.

4.2 Methoden-Cache-Transaktionen und -Historien

Eine Methoden-Cache-Transaktion ist ähnlich aufgebaut wie eine gewöhnliche Ein-Versionen-Transaktionen – allerdings darf sie neben den bekannten r- und w-Operationen auch Operationen der Form $m_i^{k,l}$ enthalten. Letztere kennzeichnen den Zugriff auf ein gecachtes Resultat eines Lesemethodenaufrufs, wobei das Resultat eventuell aus eine vorausgehenden Transaktion stammt. Bezüglich $m_i^{k,l}$ bestimmt i die Transaktion $(T_i, <_i)$ in der die Operation stattfindet und k, l verweist im Wesentlichen auf das

gecachte Methodenresultat. Dieses wurde in der Transaktion $(T_k, <_k)$ mit Hilfe einer Menge von r-Operationen (aus T_k) berechnet. Zur Bestimmung der entsprechenden r-Operationen sind diese mit dem hochgestellten Index l in T_k gekennzeichnet. Damit ist klar, von welchen Leseoperationen der Form $r_k^l[x]$ die Methodenoperation $m_i^{k,l}$ abhängt. Zur Verdeutlichung diese Sachverhalts greift das nachfolgende Beispiel bereits der Definition von Methoden-Cache-Historien vor.

Beispiel 1. *Der folgende Ausdruck beschreibt die Methode-Cache-Historie* $(H_1, <)$*:*

$$(H_1, <) = r_1^4[y] r_1^4[x] c_1 w_2[x] c_2 m_3^{1,4} w_3[x] c_3.^7$$

Die Operation $m_3^{1,4}$ *aus Transaktion* T_3 *bezieht sich auf die Leseoperationen* $r_1^4[y]$ *und* $r_1^4[x]$ *aus Transaktion* T_1*.*

Mit dem intuitiven Verständnis von Methoden-Cache-Historien können diese nun formal eingeführt werden.

Definition 1. *Methoden-Cache-Transaktion: Eine* Methoden-Cache-Transaktion $(T_i, <_i)$ *ist eine Menge von Operationen* T_i *mit einer partiellen Ordnungsrelation* $<_i$*, wobei*

- $T_i \subseteq \{w_i[x], r_i^j[x], m_i^{k,l} |\ x$ *ist ein Datenelement und* $j, k, l \in \mathbb{N} \setminus \{0\}\} \cup \{a_i, c_i\}$*,*
- $a_i \in T_i \Leftrightarrow c_i \notin T_i$*,*
- $\forall p \in T_i : p \notin \{a_i, c_i\} \Rightarrow (p <_i a_i \vee p <_i c_i)$*,*
- $\forall r_i^j[x], w_i[x] \in T_i : r_i^j[x] <_i w_i[x] \Leftrightarrow \neg(w_i[x] <_i r_i^j[x])$*.*

Definition 2. *Methoden-Cache-Historie: Sei* $\{(T_1, <_1), \ldots, (T_n, <_n)\}$ *eine Menge von Transaktionen. Eine* Methoden-Cache-Historie $(H^{MC}, <)$ *ist die Vereinigung der Operationen der* T_i *mit einer partiellen Ordnungsrelation* $<$*, wobei* $H^{MC} = \bigcup_{i=1}^n T_i$ *mit der Bedingung*

$$\forall m_i^{k,l} \in H^{MC} : k \in \{1, \ldots, n\} \wedge \forall r_k^l[x] \in H^{MC} : r_k^l[x] < m_i^{k,l}$$

gilt und die Ordnungsrelationen $<_i$ *in* $<$ *enthalten sind, also* $< \subseteq \bigcup_{i=1}^n <_i$*.*[8]

Methoden-Cache-Transaktionen bzw. -Historien sind also ähnlich aufgebaut wie die Ein-Versionen-Transaktionen bzw. -Historien in [4]. Der wichtigste Unterschied sind die neue Art von Operationen, die *Methodenoperationen* der Form $m_i^{k,l}$, die sich über den hochgestellten Index k, l auf null oder mehr vorausgehende r-Operationen aus der Transaktion $(T_k, <_k)$ beziehen.

Im Gegensatz zum Transaktionsbegriff aus [4] kann es in *einer* Methoden-Cache-Transaktion $(T_i, <_i)$ mehrere r-Operationen auf dem gleichen Datenelement x geben, die sich durch die hochgestellten Indizes in der Menge der Transaktionsoperationen T_i

[7] Die Ordnungsbeziehung $<$ wird hier durch nebeneinanderstehende Operationen ausgedrückt. Für zwei Operationen p, q bedeutet pq also eigentlich $p < q$.

[8] In der letzten Formel der Definition werden $<$ bzw. $<_i$ als Mengen aufgefasst (auf denen $\subseteq$ und $\cup$ operieren).

unterscheiden. Hierbei handelt es sich (auch ohne Berücksichtigung der Methodenoperationen) formal um ein Erweiterung des Transaktionsbegriffs.

Häufig gibt es in einer Transaktion $(T_k, <_k)$ neben einer Operation $r_k^l[x]$ *keine* weitere Operation $r_k^o[x]$ mit $l \neq o$. Sofern es in einer entsprechenden Methoden-Cache-Historie keine Operationen $m_i^{k,l}$ gibt, kann man den Index l der Einfachheit halber weglassen, denn $r_k^l[x]$ ist dann in T_k auch ohne den Index l eindeutig. *Wir schreiben deshalb in vielen der nachfolgenden Beispiele, Definitionen und Beweise zur Vereinfachung häufig $r_k[x]$ anstatt $r_k^l[x]$, falls der Index l nicht weiter von Bedeutung ist.*

Die bisherige Definition für Methoden-Cache-Historien ist noch nicht ganz befriedigend, da (im Gegensatz zu Ein-Versionen-Historien in [4]) zueinander in Konflikt stehende Operationen nicht notwendig geordnet sein müssen. Im Folgenden wird zunächst der Konfliktbegriff definiert und darauf aufbauend werden „wohl geordnete Methoden-Cache-Historien" entwickelt. Letztere ordnen, wie man es erwartet, alle Paare von konfliktbehafteten Operationen.

Wegen den Methodenoperationen erweitert der vorgestellte Konfliktbegriff die klassischen rw- und ww-Konflikte um mw-Konflikte und setzt die Definition von Methoden-Cache-Historien von oben voraus.

Definition 3. *Konflikte in Methoden-Cache-Historien: Sei $(H^{MC}, <)$ eine Methoden-Cache-Historie. Für eine Operation $p \in H^{MC}$ sei $d(p)$ die Menge der Datenelemente, auf die sie zugreift, definiert durch:*

$$d(p) := \begin{cases} \{\ x\ \} & \text{falls } p = r_i^j[x] \vee p = w_i[x] \\ \{\ x \mid \exists r_k^l[x] \in H^{MC}\ \} & \text{falls } p = m_i^{k,l} \end{cases}.$$

Weiter sei $a(p)$ der Zugriffstyp von p, also $a(p) = r$, $a(p) = w$ oder $a(p) = m$ (entsprechend p) und $T(p)$ die Transaktion, zu der p gehört.

Ein Konflikt $u \nparallel v$ *zwischen zwei Operationen $u, v \in H^{MC}$ existiert genau dann, wenn gilt:*

$$d(u) \cap d(v) \neq \emptyset \wedge ((a(u) = w \wedge a(v) = m) \vee (a(v) = w \wedge a(u) = m) \vee$$

$$(T(u) \neq T(v) \wedge ((a(u) = w \wedge a(v) = w) \vee (a(u) = w \wedge a(v) = r) \vee (a(v) = w \wedge a(u) = r)))).$$

Offensichtlich ist die Konfliktrelation $\nparallel$ symmetrisch.

Die klassischen rw- bzw. ww-Konflikte gibt es auch in Methoden-Cache-Historien. Daneben steht eine Methodenoperationen $m_i^{k,l}$ mit den Schreiboperationen in Konflikt, die auf die gleichen Datenelemente zugreifen wie die Leseoperationen, auf die sich $m_i^{k,l}$ bezieht. Man beachte, dass hierbei die m- und die w-Operation auch aus der selben Transaktion stammen können. Beispielsweise existiert in der Methoden-Cache-Historie $(H_2, <) = r_1^2[x] w_1[x] m_1^{1,2} c_1$ der Konflikt $w_1[x] \nparallel m_1^{1,2}$.

Beispiel 2. *In $(H_1, <)$ aus Beispiel 1 gibt es folgende Konflikte:*

$$r_1^4[x] \nparallel w_2[x], r_1^4[x] \nparallel w_3[x], w_2[x] \nparallel w_3[x], m_3^{1,4} \nparallel w_2[x] \text{ und } m_3^{1,4} \nparallel w_3[x].$$

Definition 4. *Wohl geordnete Methoden-Cache-Historien: Ein Methoden-Cache-Historie* $(H^{MC}, <)$ *ist* wohl geordnet *genau dann, wenn gilt:*

$$\forall p, q \in H^{MC} : p \nparallel q \Rightarrow p < q \vee q < p.$$

In wohl geordneten Methoden-Cache-Historien sind also alle Paare von Operationen, die in Konflikt zueinander stehen, mittels $<$ geordnet. Beispielsweise ist die Historie $(H_1, <)$ trivialerweise wohl geordnet (da sie total geordnet ist).

4.3 rw- und Mehr-Versionen-Historien

Wie bereits angekündigt, werden nun rw-Transaktionen bzw. -Historien als Spezialfälle von Methoden-Cache-Transaktionen bzw. Methoden-Cache-Historien eingeführt. Die rw-Varianten entsprechen im Wesentlichen den klassischen Ein-Versionen-Transaktionen bzw. -Historien aus [4].[9]

Definition 5. rw*-Transaktion und* rw*-Historie: Eine* rw-Transaktion $(T_i, <_i)$ *ist eine Methoden-Cache-Transaktion, die keine Methodenoperationen enthält, genauer muss also gelten:*

$$\forall p \in T_i : a(p) \neq m.$$

Eine rw-Historie $(H, <) = (H^{MC}, <)$ *ist eine wohl geordnete Methoden-Cache-Historie, die ausschließlich aus* rw*-Transaktionen besteht, das heißt* $\forall p \in H : a(p) \neq m$.

Der Vollständigkeit halber und damit der Formalismus in sich abgeschlossen ist, werden im Folgenden Serialisierbarkeitsgraphen für rw-Historien definiert sowie Mehr-Versionen-Historien, Versionsordnungen und Mehr-Versionen-Serialisierbarkeitsgraphen. Die Definitionen und die verwendete Notation orientieren sich bis auf eine noch näher zu erläuternden Ausnahme meist an [4].[10]

Definition 6. *Serialisierbarkeitsgraphen für* rw*-Historien: Seien* $\{(T_1, <_1), \ldots, (T_n, <_n)\}$ *die* rw*-Transaktionen zur* rw*-Historie* $(H, <)$*. Der* Serialisierbarkeitsgraph $SG \subseteq \{T_1, \ldots, T_n\}^2$ *zu* $(H, <)$ *ist dann durch den folgenden Ausdruck gegeben:*

$$(T_i, T_j) \in SG :\Leftrightarrow c_i \in T_i \wedge c_j \in T_j \wedge \exists p \in T_i : \exists q \in T_j : p \nparallel q \wedge p < q.$$

Statt $(T_i, T_j) \in SG$ *schreibt man auch* $T_i \rightarrow T_j$.

Definition 7. *Mehr-Versionen-Historie: Ein Tupel* $(H^{MV}, <)$ *ist genau dann eine* Mehr-Versionen-Historie*, wenn eine Menge von* rw*-Transaktionen* $\{(T_1, <_1), \ldots, (T_n, <_n)\}$ *existiert, so dass* $H^{MV} = \{\ h(p) \mid p \in \bigcup_{i=1}^{n} T_i)\ \}$ *gilt und* $<$ *eine partielle Ordnungsrelation ist. Die Funktion* h *muss dabei folgende Eigenschaften besitzen:*

$$\forall c_i, a_i, w_i[x] \in \cup_{i=1}^{n} T_i : h(c_i) = c_i \wedge h(a_i) = a_i \wedge h(w_i[x]) = w_i[x_i], \text{ und}$$

$$\forall r_i^l[x] \in \cup_{k=1}^{n} T_k : \exists j \in \mathbb{N} : h(r_i^l[x]) = r_i[x_j].$$

Ferner muss die Ordnungsrelation $<$ *die folgenden Bedingungen erfüllen:*

[9] Das Präfix „rw“ wurde hier lediglich eingeführt, um den Unterschied zu Methoden-Cache-Transaktionen bzw. -Historien zu betonen.

[10] Für den Leser, der weniger Erfahrung im Bereich Transaktionstheorie besitzt, seien in diesen Zusammenhang Kapitel 2 und Abschnitt 5.2 von [4] empfohlen.

- $\forall i \in \{1, \ldots, n\} : \forall p, q \in T_i : p <_i q \Rightarrow h(p) < h(q)$,
- $\forall r_j^l[x] \in \bigcup_{k=1}^n T_k : h(r_j^l[x]) = r_j[x_i] \Rightarrow (i = 0 \vee \exists w_i[x_i] \in H^{MV} : w_i[x_i] < r_j[x_i])$,
- $\forall r_j^l[x] \in \bigcup_{k=1}^n T_k : (h(r_j^l[x]) = r_j[x_i] \wedge i \neq j \wedge c_j \in H^{MV}) \Rightarrow c_i \in H^{MV}$.

Für ein Datenelement x bezeichnet man x_i mit $i \in \mathbb{N}$ als Version *von x.*

In Mehr-Versionen-Historien wird unterstellt, dass für etwaige Leseoperationen auf einem Datenelement x, die vor irgendeiner Schreiboperation auf x stattfinden, bereits eine Version x_0 von x existiert.

Im Gegensatz zu der entsprechenden Definition von Mehr-Versionen-Historien in [4] wird hier nicht verlangt, dass eine Transaktion die Version eines Datenelements liest, die sie selbst zuvor geschrieben hat. Genauer wurde die folgende Bedingung aus der obigen Definition herausgenommen:

$$\forall w_i[x], r_i^l[x] \in \bigcup_{k=1}^{n} T_k : w_i[x] < r_i^l[x] \Rightarrow h(r_i^l[x]) = r_i[x_i].$$

Die Bedingung ist im Zusammenhang mit der weiter unten folgenden Definition von serialisierbaren Methoden-Cache-Historien zu stark einschränkend. Man kann jedoch leicht zeigen, dass sie immer erfüllt ist, wenn der Serialisierbarkeitsgraph einer Mehr-Versionen-Histore (entsprechend Definition 9 von unten) azyklisch ist. Dies wird durch die erste Disjunktionsklausel in Definition 9 erzwungen.

Desweiteren ist zu beachten, dass bei der Abbildung von Operationen der Form $r_j^l[x]$ mittels h der Index l „verloren geht". Dadurch kann es vorkommen, dass etwa zwei Operationen $r_j^l[x]$, $r_j^k[x]$ mit $l \neq k$ auf die gleiche Operation $r_j[x_m]$ abgebildet werden. Dies führt zu einer Vereinfachung des Mehr-Versionen-Formalismus, stellt aber für die Korrektheit der nachfolgenden Betrachtungen zur Serialisierbarkeit keine Gefahr da.[11]

Definition 8. *Versionsordnung: Es sei D die Menge der Datenelemente, auf die die Operationen einer Mehr-Versionenhistorie $(H^{MV}, <)$ zugreifen, also $D = \{x \mid \exists r_i[x_j] \in H^{MV} \vee \exists w_i[x_i] \in H^{MV}\}$. Eine* Verionsordnung $\ll$ *ist eine Ordnungsrelation auf den Versionen der Elemente aus D, die alle Versionen zu einem $x \in D$ total ordnet mit x_0 als dem kleinsten Element, genauer:*

$$\forall x \in D : \forall i, j \in \mathbb{N} \setminus \{0\} : x_0 \ll x_i \wedge (i \neq j \Rightarrow x_i \ll x_j \vee x_j \ll x_i).$$

Definition 9. *Serialisierbarkeitsgraph für Mehr-Versionen-Historien: Seien $\{(T_1, <_1), \ldots, (T_n, <_n)\}$ die rw-Transaktionen zur Mehr-Versionen-Historie $(H^{MV}, <)$ und $\ll$ eine Versionsordnung zu $(H^{MV}, <)$. Der Mehr-Versionen-Serialisierbarkeitsgraph $MVSG \subseteq \{T_1, \ldots, T_n\}^2$ zu $(H^{MV}, <)$ ist dann durch den folgenden Ausdruck gegeben:*

$$(T_i, T_j) \in MVSG :\Leftrightarrow c_i \in T_i \wedge c_j \in T_j \wedge \exists r_k[x_l], w_m[x_m] \in H^{MV} :$$

[11] Man dazu beachte, dass es in einer serialisierbaren rw-Transaktion $(T_j, <_j)$ höchstens *zwei* Werte für ein Datenelement x geben kann, das durch eine Menge von Operationen $M \subseteq \{r_j^l[x] \mid l \in \mathbb{N}\}$ gelesen wird – nämlich den Wert von x *vor* einem $w_j[x]$ und *nach* $w_j[x]$ (sofern $w_j[x]$ in T_j vorhanden ist). In Mehr-Versionen-Historien kann das Verhalten entsprechender Mehr-Versionen-Leseoperationen leicht mit Hilfe zweier Versionen für x dargestellt werden (wie etwa x_m und x_j mit $m \neq j$).

$$(i = j = k = m \wedge i \neq l \wedge w_i[x_i] < r_i[x_l]) \vee (i \neq j \wedge m = i = l \wedge k = j) \vee$$
$$(i \neq j \wedge m = i \wedge l = j \wedge x_m \ll x_l) \vee (i \neq j \wedge k = i \wedge m = j \wedge x_l \ll x_m).$$

Statt $(T_i, T_j) \in MVSG$ schreibt man wiederum $T_i \rightarrow T_j$. Falls eine der letzten beiden Disjunktionsbedingungen im obigen Ausdruck erfüllt ist, nennt man $T_i \rightarrow T_j$ eine Versionsordnungskante. *Zwischen den Operationen $r_k[x_l]$, $w_m[x_m]$ besteht dann ein so genannter* Versionskonflikt.

4.4 Einbettung von rw- in Mehr-Versionen-Historien

Nun ist das Rüstzeug beisammen, um zu zeigen, wie man rw-Historien als besondere Mehr-Versionen-Historien interpretieren kann. Dabei sollen insbesondere die Operationen einer rw-Historie „im Wesentlichen" erhalten bleiben und es sollen *genau* die Konflikte, die bereits in der rw-Historie existieren, in der entsprechenden Mehr-Versionen-Historie vorkommen.

Hierzu wird im Folgenden die Abbildung MV definiert, die eine rw-Historie $(H, <)$ auf eine Mehr-Versionen-Historie abbildet. MV spaltet sich auf in die Abbildung mv für die Operationen aus H und in eine Abbildung für die Ordnungsrelation $<$. Die ursprünglichen Operationen aus H bleiben durch mv im Wesentlichen erhalten: Für Operationen der Form $r_i^j[x]$ wird ein Versionsindex bezüglich x berechnet, so dass $r_i[x_k] \in mv(r_i^j[x])$ gilt.[12] Für $w_i[x]$-Operationen erzeugt mv neben $w_i[x_i]$ unter Umständen noch eine zusätzliche Operation $r_i[x_k] \in mv(w_i[x])$.

Die zusätzliche r-Operation hilft, genau jene Konflikte nachzubilden, die in der rw-Historie zwischen Schreiboperationen $w_i[x]$, $w_j[x]$ (mit $i \neq j$) existieren. Diese Konflikte würden sonst auf den Bildoperationen $w_i[x_i]$ und $w_j[x_j]$ „verloren gehen" – sie werden deshalb in der Mehr-Versionen-Historie durch Versionskonflikte nachgebildet.[13]

MV behält die Ordnung $<$ auf der entstehenden Mehr-Versionen-Historie bei – dort wird sie mittels $<_{mv}$ auf den Bildern $mv(p)$ der Operationen $p \in H$ notiert. Falls $r_i[x_k] \in mv(w_i[x])$ gilt, erhält die Operation $r_i[x_k]$ die gleichen Ordnungsbeziehungen wie $w_i[x_i]$ bezüglich $<_{mv}$.

Definition 10. *Sei $(H, <)$ eine rw-Historie mit den rw-Transaktionen $\{(T_1, <_1), \ldots, (T_n, <_n)\}$. Die zu $(H, <)$* äquivalente *Mehr-Versionen-Historie $MV((H, <)) = (\cup_{p \in H} mv(p), <_{mv})$ ist dann wie folgt definiert:*

$$mv(c_i) := \{c_i\}, mv(a_i) := \{a_i\},$$

$$mv(w_i[x]) := \begin{cases} \{w_i[x_i]\} & \textit{falls } \exists r_i[x] \in H : r_i^k[x] < w_i[x] \\ \{r_i[x_j], w_i[x_i]\} \textit{ mit } j = Q(w, i, x) & \textit{sonst} \end{cases}$$

$$mv(r_i^k[x]) := \begin{cases} \{r_i[x_i]\} & \textit{falls } \exists w_i[x] \in H : w_i[x] < r_i^k[x] \\ \{r_i[x_j]\} \textit{ mit } j = Q(r, i, x) & \textit{sonst} \end{cases} \quad \textit{und}$$

[12] Der Index j wird bezüglich $r_i^j[x]$ einfach weggelassen – dies stellt aber keine bedeutende Veränderung der Historie in Bezug auf ihre Konflikte dar. (Siehe hierzu auch die Kommentare von weiter oben.)

[13] Es sei daran erinnert, dass in Mehr-Versionen-Historien „gewöhnliche" Scheib-Lese-Konflikte nur bezüglich gleicher Versionen von Datenelementen entstehen können.

$\forall p, q \in \cup_{h \in H} mv(h) : p <_{mv} q :\Leftrightarrow \exists u, v \in H : (u < v \wedge p \in mv(u) \wedge q \in mv(v))$, *wobei*

$$Q(p,i,x) = \begin{cases} l \quad \textit{falls} \quad \begin{array}{c} \exists_1 w_l[x] \in H : w_l[x] < p_i[x] \wedge c_l \in H \wedge \\ (\forall w_m[x] \in H : w_m[x] < p_i[x] \wedge c_m \in H \Rightarrow \\ (w_m[x] < w_l[x] \vee m = l)) \end{array} \\ 0 \quad \textit{sonst} \end{cases}.$$

$Q(p,i,x)$ bestimmt für Mehr-Versionen-Operationen der Form $r_i[x_j]$ den Versionsindex j zu x. Die Funktion liefert den Transaktionsindex l der letzten Schreiboperation $w_l[x]$, die der Operation $p_i[x] \in H$ (direkt) vorausgeht oder 0, falls es keinen solchen Vorgänger gibt. Desweiteren muss T_l abgeschlossen sein.

Es leicht einzusehen, dass die Abbildung MV wohl geformt ist, das heißt, dass sie Historien erzeugt, die den Bedingungen aus Definition 7 genügen. Insbesondere ist die letzte Bedingung aus Definition 7 erfüllt, da bei der Berechnung des Index $l = Q(p,i,x)$ auch $c_l \in H$ für ein entsprechendes $w_l[x]$ gefordert wird.

Beispiel 3. *Die zur rw-Historie*

$$(H_2, <) = r_1[x]c_1 w_2[x]c_2 r_3[x]w_3[x]c_3$$

äquivalente Mehr-Versionen-Historie ist

$$(\cup_{p \in H_2} mv(p), <_{mv}) = r_1[x_0]c_1 \begin{array}{c} \nearrow r_2[x_0] \searrow \\ \searrow w_2[x_2] \nearrow \end{array} c_2 r_3[x_2] w_3[x] c_3.^{14}$$

Der folgende Satz präzisiert die Eigenschaft von MV, die Konfliktbeziehungen einer rw-Historie $(H, <)$ ins Bild zu übernehmen.

Satz 1. *Der transitive Abschluss des Serialisierbarkeitsgraphen SG zu einer rw-Historie $(H, <)$ ist identisch zum transitiven Abschluss des Mehr-Versionen-Serialisierbarkeitsgraphen $MVSG$ von $MV((H, <))$ für eine Versionsordnung $\ll$, die die folgende Bedingung erfüllt:*

$$\forall w_i[x], w_j[x] \in H : (w_i[x] < w_j[x] \vee i = 0) \Rightarrow x_i \ll x_j.$$

Für den Beweis des Satzes ist die angegebene Einschränkung für die Versionsordnung bzw. deren so getroffene Wahl zwingend erforderlich.

Beweis. Offensichtlich gilt $\forall i \in \{1, \ldots, n\} : c_i \in T_i \Leftrightarrow c_i \in mv(T_i)$. Die Abgeschlossenheit von Transaktionen braucht also bezüglich der Bedingung für Kanten in den jeweiligen Serialisierbarkeitsgraphen nicht weiter betrachtet zu werden.

Zuerst wird gezeigt: Gibt es eine Kante $T_i \rightarrow T_j$ im Serialisierbarkeitsgraphen SG von $(H, <)$, dann existiert eine entsprechende (transitive) Kante im Mehr-Versionen-Serialisierbarkeitsgraphen $MVSG$ zu $MV((H, <))$ (Behauptung I).

Sei also $T_i \rightarrow T_j \in SG$. Dann gibt es ein x mit $p_i[x] < q_j[x]$, $p_i[x] \not\parallel q_j[x]$ (und somit $i \neq j$).

[14] Die Ordnungsbeziehung $<_{mv}$ ist hier teilweise mittels Pfeilen dargestellt.

Falls $p = r, q = w$, hat man $r_i[x_s] <_{mv} w_j[x_j]$ in $MV((H, <))$ (für ein s). Damit gilt notwendig $w_s[x] < w_j[x]$, also $x_s \ll x_j$, und somit erhält man die Versionsordnungskante $T_i \rightarrow T_j \in MVSG$.

Falls $p = w, q = r$, hat man $w_i[x_i] <_{mv} r_j[x_s]$ in $MV((H, <))$ (für ein s), wobei für $w_s[x_s]$ gilt: Entweder ist $i = s$, und man ist fertig oder $w_i[x] < w_s[x]$ und somit $w_i[x_i] <_{mv} w_s[x_s]$ auf Grund der Wahl von s für $r_j[x_s]$ mittels $Q(r, i, x) = s$ in Definition 10. Mit $c_s \in mv(T_s)$ hat man dann also $T_s \rightarrow T_j \in MVSG$. Wie man sehen wird, impliziert aber der nachfolgend behandelte Fall die Kante $T_i \rightarrow T_s \in MVSG$ wegen $w_i[x] < w_s[x]$, und somit gelangt man zur transitiven Kante $T_i \rightarrow T_s \rightarrow T_j$.

Es bleibt also der Fall $p = w, q = w$ zu betrachten: Sei $w_i[x] = w_{k_1}[x] < \ldots < w_{k_n}[x] = w_j[x]$ die Teilfolge *aller* Schreiboperation bezüglich x in H zwischen $w_i[x]$ und $w_j[x]$ mit $n \geq 2$, wobei jeweils $c_{k_o} \in T_o$ mit $o \in \{1, \ldots, n\}$ gelte. Behauptung: Es gibt einen Pfad $T_{k_1} \rightarrow T_{k_n}$ in $MVSG$.

Der Beweis erfolgt durch vollständige Induktion über n. Induktionsanfang ($n = 2$): In diesem Fall ist $w_{k_1}[x]$ die direkt vorausgehende Schreiboperation von $w_{k_2}[x]$ bezüglich x mit $c_{k_1} \in T_{k_1}$. Dann hat man $mv(w_{k_2}[x]) = \{w_{k_2}[x_{k_2}], r_{k_2}[x_{k_1}]\}$ wegen $Q(w, k_2, x)$ und natürlich $w_{k_1}[x_{k_1}] <_{mv} r_{k_2}[x_{k_1}]$, also gilt $T_{k_1} \rightarrow T_{k_2} \in MVSG$. Induktionsschluss ($n-1 \curvearrowright n$): Die Argumentation erfolgt analog zum Induktionsanfang – man ersetze lediglich k_1 durch k_{n-1} und k_2 durch k_n.

Insgesamt ist sind somit alle Fälle bezüglich der Behauptung I abgehandelt.

Es bleibt zu zeigen: Gibt es eine Kante $T_i \rightarrow T_j$ im Mehr-Versionen-Serialisierbarkeitsgraphen $MVSG$ von $MV((H, <))$, dann existiert eine entsprechende (transitive) Kante im Serialisierbarkeitsgraphen SG zu $(H, <)$ (Behauptung II).

Sei also $T_i \rightarrow T_j \in MVSG$. Es kann sich entweder um eine Versionsordnungskante oder um eine Kante wegen $w_i[x_i] <_{mv} r_j[x_i]$ handeln und damit gilt auch $i \neq j$. Insbesondere kann der Fall $w_i[x_i] <_{mv} r_i[x_l]$ mit $i \neq l$ (aus der ersten Disjunktionklausel von Definition 9) wegen der Definition von mv ausgeschlossen werden.

Zum Fall $w_i[x_i] <_{mv} r_j[x_i]$: Auf Grund der Definition von mv gilt dann entweder $w_i[x] < r_j[x]$ (da $mv(r_j[x]) = \{r_j[x_k]\}$ für ein k) oder $w_i[x] < w_j[x]$ (da $mv(w_j[x]) = \{r_j[x_k], w_j[x_j]\}$ für ein k), also ist $T_i \rightarrow T_j \in SG$.

Falls es sich bei $T_i \rightarrow T_j \in MVSG$ um eine Versionsordnungskante handelt, gibt es zwei Unterfälle.

Fall 1: Man hat $w_i[x_i], r_k[x_j]$ in $MV((H, <))$ (für ein k) mit $x_i \ll x_j$. Weil mit $w_i[x_i]$ auch $i \neq 0$ gilt und weil $\ll$ eine Versionsordnung ist, kann j nicht 0 sein, und somit existiert auch ein $w_j[x_j]$. Wegen der Voraussetzung zur Verionsordnung im Satz folgt mit $x_i \ll x_j$ sogar $w_i[x] < w_j[x]$ und damit insbesondere $T_i \rightarrow T_j \in SG$.

Fall 2: Man hat $r_i[x_k], w_j[x_j]$ in $MV((H, <))$ (für ein k) mit $x_k \ll x_j$. In diesem Zusammenhang gibt es wiederum zwei Unterfälle, nämlich $r_i[x_k] <_{mv} w_j[x_j]$ und $w_j[x_j] <_{mv} r_i[x_k]$. (Die beiden Operationen sind sicher vergleichbar mittels $<_{mv}$, denn ihre Urbilder bezüglich mv sind konfliktbehaftete Operationen $p[x] < q[x]$ und die Ordnung von $<$ wird in $<_{mv}$ beibehalten.)

Sei zuerst $r_i[x_k] <_{mv} w_j[x_j]$. Dann ist $r_i[x_k] \in mv(r_i[x])$ oder $r_i[x_k] \in mv(w_i[x])$, und man hat $r_i[x] < w_j[x]$ bzw. $w_i[x] < w_j[x]$ also $T_i \rightarrow T_j \in SG$.

Sei nun $w_j[x_j] <_{mv} r_i[x_k]$. Wegen der Definition der Funktion Q kann k somit nicht

0 sein. Wegen $x_k \ll x_j$ gilt also weiter $w_k[x] < w_j[x]$. Falls $r_i[x_k] \in mv(r_i[x])$, erhält man $w_k[x] < w_j[x] < r_i[x]$ und deshalb $Q(r, i, x) \neq k$. Dies ist ein Widerspruch zu $r_i[x_k] \in mv(r_i[x])$. Abschließend ist noch der Fall $r_i[x_k] \in mv(w_i[x])$ zu behandeln. Man erhält dann $w_k[x] < w_j[x] < w_i[x]$ und deshalb $Q(w, i, x) \neq k$. Dies steht im Widerspruch zu $r_i[x_k] \in mv(w_i[x])$.

Insgesamt sind somit alle Fälle zum Vorkommen von Kanten $T_i \rightarrow T_j$ in MVSG abgehandelt und Behauptung II folgt. □

Mit der Definition der Serialisierbarkeit von Ein-Versionen-Historien aus [4] und dem zugehörigen Serialisierbarkeitstheorem folgt das nächste Korollar unmittelbar aus Satz 1.[15]

Korollar 1. *Eine rw-Historie $(H, <)$ ist genau dann serialisierbar, wenn die äquivalente Mehr-Versionen-Historie $MV((H, <))$ einen azyklischen Mehr-Versionen-Serialisierbarkeitsgraphen besitzt für die Versionsordnung $\ll$, die die folgende Bedingung erfüllt:*

$$\forall w_i[x], w_j[x] \in H : (w_i[x] < w_j[x] \vee i = 0) \Rightarrow x_i \ll x_j.$$

4.5 Einbettung von Methoden-Cache- in Mehr-Versionen-Historien

Der nächste Schritt erweitert die Abbildung MV von rw-Historien auf wohl geordnete Methoden-Cache-Historien. Dabei werden Methodenoperationen auf Leseoperationen der entsprechenden Mehr-Versionen-Historien abgebildet, und dies verleiht den Methodenoperationen indirekt eine *Semantik*. Das Konzept von Methoden-Operationen wird also auf das Lesen einer bestimmten Menge von Versionen von Datenelementen reduziert. Eine Operation $m_i^{k,l}$ liest dabei genau die Werte bzw. Versionen von Datenelementen, die bereits in den Operationen der Form $r_k^l[x]$ in $(H^{MC}, <)$ gelesen wurden.

Da man in Ein-Versionen-Historien (bzw. rw-Historien) keine Versionen von Datenelementen hat, kann man diesen Sachverhalt in solchen Historien nicht explizit mit r-Operationen ausdrücken. Dies rechtfertigt gerade die Einführung von Methoden-Operationen in einem Ein-Versionen-Formalismus zur Untersuchung von transaktionalem Methoden-Caching. Mit dem Übergang zu Mehr-Versionen-Historien erhält man aber doch die nötige Ausdruckskraft, um Methoden-Operationen durch Leseoperationen zu beschreiben. Aus dieser Motivation heraus greift die vorliegende Arbeit auf den Mehr-Versionen-Formalismus zurück und führt eine konflikterhaltende Abbildung von Methoden-Cache-Historien zu Mehr-Versionen-Historien ein (nämlich MV').

Definition 11. *Semantik von Methoden-Cache-Transaktionen: Sei $(H^{MC}, <)$ eine wohl geordnete Methoden-Cache-Historie mit den Transaktionen $\{(T_1, <_1), \ldots, (T_k, <_k)\}$. Die zu $(H^{MC}, <)$* äquivalente *Mehr-Versionen-Historie $MV'((H^{MC}, <)) =$ $(\cup_{p \in H^{MC}} mv'(p), <_{mv'})$ ist dann wie folgt definiert:*

$$mv'(p) := mv(p) \textit{ falls } a(p) = r \textit{ oder } a(p) = w \textit{ (mit } mv \textit{ aus Definition 10) und}$$

[15] Wie bereits gesagt, sei auf [4] verwiesen für die Definition von Ein-Versionen-Serialisierbarkeit bzw. für das entsprechende Serialisierbarkeitstheorem.

$$mv'(m_i^{k,l}) := \{r_i[x_h] \mid \exists r_k^l[x] \in H^{MC} : r_k[x_h] \in mv(r_k^l[x])\},$$

$$\forall p, q \in \cup_{h \in H^{MC}} mv'(h) : p <_{mv'} q :\Leftrightarrow \exists u, v \in H^{MC} : (u < v \wedge p \in mv'(u) \wedge q \in mv'(v)) \vee$$

$$(\exists m_i^{k,l}, r_k^l[x], r_k^l[y] \in H^{MC} : r_k^l[x] < r_k^l[y] \wedge$$

$$\exists r_i[x_s], r_i[y_t] \in mv'(m_i^{k,l}) : p = r_i[x_s] \wedge q = r_i[y_t]).$$

Wie bereits oben erwähnt, bildet mv' ein $m_i^{k,l}$ auf eine Menge von Leseoperationen ab, die dieselben Versionen von Datenelementen lesen, wie die Bildelemente der Operationen $r_k^l[x]$ (unter mv). Die Ordnungsrelation $<_{mv'}$ wird, wie schon bei MV, praktisch unverändert von den Ursprungsoperationen auf die Bildelemente von mv' übertragen. Die Definition von $<_{mv'}$ ist allerdings komplizierter als die von $<_{mv}$ aus Definition 10, denn für die Elemente von $\{r_i[x_s], r_i[y_t], \ldots\}$ aus dem Bild eines $m_i^{k,l}$ unter mv' sollen auch die Ordnungsbeziehungen der entsprechenden $r_k^l[x]$, $r_k^l[y]$ usw. in $<_{mv'}$ übernommen werden.

Beispiel 4. *Die Anwendung von MV' auf die Methoden-Cache-Historie $(H_1, <)$ aus Beispiel 1 liefert die die folgende Mehr-Versionen-Historie:*

$$(\cup_{p \in H_1} mv'(p), <_{mv'}) = r_1[y_0] r_1[x_0] c_1 \begin{array}{c} \nearrow r_2[x_0] \searrow \\ \searrow w_2[x_2] \nearrow \end{array} c_2 r_3[y_0] r_3[x_0] \begin{array}{c} \nearrow r_3[x_2] \searrow \\ \searrow w_3[x_3] \nearrow \end{array} c_3.$$

Mit Hilfe von MV' kann man nun ein nahe liegendes Serialisierbarkeitskriterium für Methoden-Cache-Historien festlegen. Dazu überprüft man einfach, ob der Mehr-Versionen-Serialisierbarkeitsgraph des Bildes von MV' azyklisch ist. Dies ist durch das Serialisierbarkeitstheorem für Mehr-Versionen-Historien in [4] gerechtfertigt: Es besagt, dass eine Mehr-Versionen-Historie genau dann Eins-serialisierbar (1-SR) ist, wenn eine Versionsordnung existiert, für die der entsprechende Mehr-Versionen-Serialisierbarkeitsgraph azyklisch ist.[16] Der Serialisierbarkeitsbegriff für Methoden-Cache-Historien ist im Vergleich noch stärker einschränkend als die Eins-Serialisierbarkeit, da die Versionsordnung nicht beliebig sein kann, sondern durch die Abbildung MV' bereits vorgegeben ist.

Definition 12. *Serialisierbarkeit von (wohl geordneten) Methoden-Cache-Historien: Eine wohl geordnete Methoden-Cache-Historie $(H^{MC}, <)$ ist genau dann* MC-serialisierbar *(oder einfach* serialisierbar*), wenn die äquivalente Mehr-Versionen-Historie $MV'((H^{MC}, <))$ einen azyklischen Mehr-Versionen-Serialisierbarkeitsgraphen besitzt für die Versionsordnung $\ll$, die die folgende Bedingung erfüllt:*

$$\forall w_i[x], w_j[x] \in H^{MC} : (w_i[x] < w_j[x] \vee i = 0) \Rightarrow x_i \ll x_j.$$

[16] Eins-Serialisierbarkeit ist in diesem Zusammenhang das „gewünschte" Serialisierbarkeitskriterium. Es orientiert sich stark an der Aussagekraft der Serialisierbarkeit für Ein-Versionen-Historien. Mehr dazu findet man in Abschnitt 5.2 von [4].

Beispiel 5. *Der Mehr-Versionen-Serialisierbarkeitsgraph $MVSG$ zu $MV'((H_1, <))$ (mit $(H_1, <)$ aus Beispiel 1) ist azyklisch und umfasst die folgenden Kanten:*

$$T_1 \to T_2, T_2 \to T_3, T_1 \to T_3, T_3 \to T_2.$$

Die Versionskante $T_1 \to T_2$ entsteht durch $r_1[x_0]$, $w_2[x_2]$ und $x_0 \ll x_2$. Ähnliches gilt für die Kante $T_1 \to T_3$ mit $r_1[x_0]$, $w_3[x_3]$ und $x_0 \ll x_3$. Die Kante $T_2 \to T_3$ kommt wegen dem Konflikt zwischen $w_2[x_2]$ und $w_3[x_2]$ zu Stande. Schließlich gibt es noch die Versionskante $T_3 \to T_2$ wegen $w_2[x_2]$, $r_3[x_0]$ und $x_0 \ll x_3$.

Ähnlich wie für rw-Historien und Mehr-Versionen-Historien wird nun die Konstruktion von Serialisierbarkeitsgraphen für Methoden-Cache-Historien definiert. Ein entsprechender Serialisierbarkeitsgraph ist dabei (natürlich) so strukturiert, dass er genau dann azyklisch ist, wenn die zugehörige Methoden-Cache-Historie serialisierbar ist. Dies bestätigen Satz 2 und Korollar 2 weiter unten. Das Serialisierbarkeitstheorem für Methoden-Cache-Historien repräsentiert das wichtigste Ergebnis dieses Abschnitts, denn darauf aufbauend können Serialierbarkeitsprotokolle für das transaktionale Methoden-Caching entwickelt bzw. verifiziert werden.

Definition 13. *Serialisierbarkeitsgraph für (wohl geordnete) Methoden-Cache-Historien: Seien $\{(T_1, <_1), \ldots, (T_n, <_n)\}$ die Transaktionen zur wohl geordneten Methoden-Cache-Historie $(H^{MC}, <)$. Der* Serialisierbarkeitsgraph $MCSG \subseteq \{T_1, \ldots, T_n\}^2$ *zu $(H^{MC}, <)$ ist dann durch den folgenden Ausdruck gegeben:*

$$(T_i, T_j) \in MCSG :\Leftrightarrow c_i \in T_i \wedge c_j \in T_j \wedge$$

$$((\exists p \in T_i : \exists q \in T_j : a(p) \neq m \wedge a(q) \neq m \wedge p \nparallel q \wedge p < q) \vee$$
$$(\exists m_i^{k,l}, w_j[x], r_k^l[x] \in H^{MC} : r_k^l[x] < w_j[x] \wedge (i \neq j \vee w_j[x] < m_i^{k,l}) \vee$$
$$(i \neq j \wedge \exists w_i[x], m_j^{k,l}, r_k^l[x] \in H^{MC} : w_i[x] < r_k^l[x]))$$

Statt $(T_i, T_j) \in MCSG$ schreibt man wiederum $T_i \to T_j$.

Offenbar gilt für eine wohl geordnete Methoden-Cache-Historie $(H^{MC}, <)$: $SG \subseteq MCSG$ (mit SG aus Definition 6), denn die erste Disjunktionsklausel in Definition 13 ist identisch zur Bedingung für Serialisierbarkeitsgraphen von rw-Historien in Definition 6 und jede rw-Historie ist eine wohl geordnete Methoden-Cache-Historie.

Die zweite und dritte Disjunktionsklausel der obigen Definition erzeugen Graphenkanten bezüglich Methodenoperationen. Zu einer Methodenoperation $m_i^{k,l}$ müssen hierbei die Operationen $r_k^l[x] \in H^{MC}$ betrachtet werden, auf die sich die Methodenoperation bezieht. Es werden dann Kanten zwischen T_i und allen T_j gezogen, für die ein entsprechendes $w_j[x]$ existiert. Falls dabei ein $r_k^l[x]$ vor dem entsprechenden $w_j[x]$ liegt, verläuft die Kante von T_i nach T_j (zweite Disjunktionsklausel). Ansonsten zeigt die Kante in die andere Richtung (dritte Disjunktionsklausel).

Man beachte, dass nur durch die zweite Disjunktionsklausel reflexive Kanten in $MCSG$ entstehen können. Beispielsweise führt die Historie $(H_2, <) = r_1^1[x] w_1[x] m_1^{1,1} c_1$ zur Kante $T_1 \to T_1 \in MCSG$.

Beispiel 6. *Der Serialisierbarkeitsgraph für die Methoden-Cache-Historie $(H_1, <)$ besteht aus folgenden Kanten:*

$$T_1 \to T_2, T_2 \to T_3, T_1 \to T_3, T_3 \to T_2.$$

Dies sind offenbar die gleichen Kanten wie in Beispiel 5. Die Kanten $T_1 \to T_2$, $T_2 \to T_3$ und $T_1 \to T_3$ sind gewöhnliche Konfliktkanten (erste Disjunktionsklausel in Definition 13). Die Kante $T_3 \to T_2$ kommt aber wegen $m_3^{1,1}$, $r_1^1[x]$ und $w_3[x]$ zu Stande (zweite Diskunktionsklausel in Definition 13).

Satz 2. *Der transitive Abschluss des Serialisierbarkeitsgraphen $MCSG$ zu einer wohl geordneten Methoden-Cache-Historie $(H^{MC}, <)$ ist identisch zum transitiven Abschluss des Mehr-Versionen-Serialisierbarkeitsgraphen $MVSG$ von $MV'((H^{MC}, <))$ für eine Versionsordnung $\ll$, die die folgende Bedingung erfüllt:*

$$\forall w_i[x], w_j[x] \in H^{MC} : (w_i[x] < w_j[x] \vee i = 0) \Rightarrow x_i \ll x_j.$$

Beweis. Offensichtlich gilt $\forall i \in \{1, \ldots, n\} : c_i \in T_i \Leftrightarrow c_i \in mv(T_i)$. Die Abgeschlossenheit von Transaktionen braucht also bezüglich der Bedingung für Kanten in den jeweiligen Serialisierbarkeitsgraphen nicht weiter betrachtet zu werden.

Zuerst wird gezeigt: Gibt es eine Kante $T_i \to T_j$ im Serialisierbarkeitsgraphen $MCSG$ von $(H^{MC}, <)$, dann existiert eine entsprechende (transitive) Kante im Mehr-Versionen-Serialisierbarkeitsgraphen $MVSG$ zu $MV'((H^{MC}, <))$ (Behauptung I).

Falls $T_i \to T_j \in SG$ von $(H^{MC}, <)$ dann ist $T_i \to T_j$ auch im transitiven Abschluss von $MVSG$ (siehe hierzu Satz 1).

Sei also $T_i \to T_j \in MCSG \setminus SG$. Dann kam $T_i \to T_j$ durch die zweite oder dritte Disjunktionsklausel in Definition 13 zu Stande und es existieren entweder Operationen $m_i^{k,l}$, $w_j[x]$, $r_k^l[x]$ mit $r_k^l[x] < w_j[x]$ oder $w_i[x]$, $m_j^{k,l}$, $r_k^l[x]$ mit $w_i[x] < r_k^l[x]$.

Im ersten Fall hat man für die Bilder der Operationen bezüglich mv': $mv'(r_k^l[x]) = \{r_k[x_s]\}$, $w_j[x_j] \in mv'(w_j[x])$ sowie $r_i[x_s] \in mv'(m_i^{k,l})$ (für ein s). Mit $r_k[x_s] <_{mv'} w_j[x_j]$, ergibt sich notwendig $s = 0 \vee w_s[x_s] <_{mv'} w_j[x_j]$ und somit die Versionsordnung $x_s \ll x_j$. Falls dabei $i \neq j$ gilt, erhält man für $r_i[x_s]$ und $w_j[x_j]$ die Versionskante $T_i \to T_j$ in $MVSG$ (letzte Disjunktionklausel in Definition 9). Hat man anderseits $i = j$, so folgt mit der zweiten Disjunktionsklausel von Definition 13 auch $w_j[x_j] = w_i[x_i] <_{mv'} r_i[x_s]$. Da $i \neq s$ erhält man $(T_i \to T_j = T_i) \in MVSG$ durch die erste Disjunktionsklausel von Definition 9.

Gibt es für $T_i \to T_j \in MCSG$ die Operationen $w_i[x]$, $m_j^{k,l}$, $r_k^l[x]$ mit $w_i[x] < r_k^l[x]$ (dritte Disjunktionsklausel von Definition 13), so verhalten sich die entsprechenden Elemente der Bilder bezüglich mv' wie folgt: $w_i[x_i] <_{mv'} r_k[x_s] <_{mv'} r_j[x_s]$ (für ein s). Falls $i = s$ gilt, ist man fertig. Ansonsten kann man per Induktion wie im Beweis von Satz 1 schließen, dass $T_i \to T_s \in MVSG$ gilt mit $w_s[x_s] \in T_s$ und somit auch $T_i \to T_j \in MVSG$. (Man beachte, dass s wegen der Definition von Q und wegen $w_i[x]$ sicher größer 0 ist.)

Es bleibt zu zeigen: Gibt es eine Kante $T_i \to T_j$ im Mehr-Versionen-Serialisierbarkeitsgraphen $MVSG$ von $MV'((H^{MC}, <))$, dann existiert eine entsprechende (transitive) Kante im Serialisierbarkeitsgraphen $MCSG$ zu $(H^{MC}, <)$ (Behauptung II).

Sei $T_i \rightarrow T_j \in MVSG$. In Satz 1 wurden bereits alle Kanten untersucht, die durch die Abbildung MV entstanden. Es genügt also Kanten zu untersuchen, die *zusätzlich* durch MV' (als Erweiterung von MV) eingebracht werden. Gemäß Definition 11 muss eine solche Kante durch die Abbildung einer Operation $m_n^{k,l}$ entstanden sein. Seien etwa $r_n[x_s] \in mv'(m_n^{k,l})$ und $w_t[x_t]$ die Operationen, die $T_i \rightarrow T_j \in MVSG$ produzieren. Dann gibt es nach Definition 9 vier mögliche Fälle zu unterschieden: $i = j = n = t, i \neq s, w_i[x_i] <_{mv'} r_i[x_s]$ oder $i \neq j, i = s = t, j = n$ oder $i \neq j, i = t, j = s, x_t \ll x_s$ oder $i \neq j, n = i, t = j, x_s \ll x_t$.

Im ersten Fall hat man $r_k^l[x] < w_i[x] < m_i^{k,l}$ oder $w_i[x] < w_o[x] < r_k^l[x] < m_i^{k,l}$, denn sonst wäre $i = s$. Für $r_k^l[x] < w_i[x] < m_i^{k,l}$ ergibt sich die Kante $T_i \rightarrow T_i \in MCSG$ mit $i = j$ aus der zweiten Disjunktionsklausel von Definition 13. Für $w_i[x] < w_o[x] < r_k^l[x] < m_i^{k,l}$ hat man $T_i \rightarrow T_o \rightarrow T_i \in MCSG$. Somit gilt die Behauptung für diesen Fall.

Im zweiten Fall erhält man durch Einsetzen: $w_i[x_i] <_{mv'} r_j[x_i] \in mv'(m_j^{k,l})$ mit $w_i[x_i] \in mv'(w_i[x])$. Also existiert in der Historie H^{MC} auch ein $r_k^l[x]$ mit $w_i[x] < r_k^l[x]$. Denn falls andererseits $r_k^l[x] < w_i[x]$ gelten würde, wäre $mv'(r_k^l[x]) = \{r_k[x_g]\}$ für ein $g \neq i$. Damit erhielte man $r_j[x_g] \in mv'(m_j^{k,l})$ anstatt $r_j[x_i] \in mv'(m_j^{k,l})$ (Widerspruch). Insgesamt folgt somit $T_i \rightarrow T_j \in MCSG$ aus der letzten Disjunktionsklausel von Definition 13.

Zum Fall $i \neq j, i = t, j = s, x_t \ll x_s$: Hier folgt direkt $w_i[x] < w_j[x]$ wegen der Voraussetzung für $\ll$ im zu beweisenden Satz. (Man bedenke, dass $t = i$ nicht 0 sein kann.)

Beim letzten Fall erhält man durch Einsetzen die Situation $x_s \ll x_j$, $r_i[x_s] \in mv'(m_i^{k,l})$ und $w_j[x_j] \in mv'(w_j[x])$ mit $w_j[x] \in H^{MC}$. Wegen $m_i^{k,l}$ existiert darüber hinaus ein $r_k^l[x] \in H^{MC}$ mit $r_k^l[x] < m_i^{k,l}$. Falls $r_k^l[x] < w_j[x]$ ergibt sich $T_i \rightarrow T_j$ durch die zweite Disjunktionsklausel von Definition 13. Ansonsten hat man $w_j[x] < r_k^l[x]$. Falls $r_k^l[x]$ von T_j liest ergibt sich unter MV': $w_j[x_j] <_{mv'} r_k[x_j] <_{mv'} r_i[x_j] \in mv'(m_i^{k,l})$ und somit $j = s$ im Widerspruch zu $x_s \ll x_j$. Falls andererseits $r_k^l[x]$ x in einem T_o (ungleich T_j) liest, also $w_j[x] < w_o[x] < r_k^l[x]$ vorliegt, gelangt man unter MV' zu $w_j[x_j] <_{mv'} w_o[x_o] <_{mv'} r_k[x_o] <_{mv'} r_i[x_o] \in mv'(m_i^{k,l})$. Daraus folgt $s = o$ und weiter $x_j \ll x_s$ (wegen $w_j[x_j] < w_o[x_o]$) im Widerspruch zur Annahme.

Insgesamt sind somit alle möglichen Fälle untersucht und Behauptung II folgt. □

Das nachfolgende Korollar folgt unmittelbar aus Definition 12 und 13 sowie Satz 2. Es repräsentiert das wichtigste Ergebnis dieses Abschnitts.

Korollar 2. *Eine wohl geordnete Methoden-Cache-Historie $(H^{MC}, <)$ ist genau dann MC-serialisierbar, wenn der Methoden-Cache-Serialisierbarkeitsgraph $MCSG$ zu $(H^{MC}, <)$ azyklisch ist.*

5 Entwurfsansätze zum transaktionalen Methoden-Caching für EJB-basierte Applikationsserver

Dieses Kapitel konkretisiert die Aussagen zur Architektur eines transaktionalen Methoden-Caches aus Abschnitt 3.2. Zunächst wird dazu ein UML-Entwurf von Klassen zur Verarbeitung von klientenseitigen Anwendungstransaktionen diskutiert, wie er in ähnlicher Weise bei herkömmlichen EJB-basierten Applikationsservern vorkommt. Darauf aufbauend folgen objektorientierte Entwurfsansätze zur Erweiterung dieser Struktur, die transaktionales methodenbasiertes Caching ermöglichen. Die vorgestellten Ansätze sind jedoch nicht vollständig sondern bilden ein Grundgerüst zur Beschreibung von Protokollimplementierungen für m-Planer in Kapitel 7.

Auf Basis der nachfolgenden Entwürfe wurde ein Prototyp entwickelt, der für die Evaluation der vorgeschlagenen Konzepte diente. Die entsprechende Implementierung erweitert den EJB-Applikationsserver JBoss ([17]).

5.1 Typische Klassenstruktur eines EJB-basierten Applikationsservers

Abbildung 7 zeigt ein konzeptionell stark vereinfachtes UML-Klassendiagramm eines EJB-Applikationsservers zur Verarbeitung von klientenseitigen Anwendungstransaktionen. Der Aufbau ist an die Struktur von JBoss angelehnt, lässt jedoch sehr viele Details außer acht. Er skizziert vor allem Klassen in Bezug auf den Datenfluss bei der Verarbeitung von Klententransaktionen.

Als Beispiel wird unterstellt, dass eine Klasse `ServiceMethodImplementation` vorliegt, die eine (bestimmte) Enterprise Java Bean repräsentiert.[17] Die Klasse enthält die Dienstmethode `someServiceMethod()`, deren Implementierung, wie angedeutet, über eine JDBC-Datenbankverbindung ([30]) Datenelemente aus einer relationalen Datenbank liest. Das unterliegende Datenbankverwaltungssystem ist dabei XA-konform und der JDBC-Treiber für das System unterstützt die XA-Anbindung für Java. Dies äußert sich dadurch, dass im JDBC-Treiber Unterklassen von `XAResource` und `XAConnection` vorkommen.

Die Abbildung unterscheidet neben den JDBC-Treiber-Klassen vier weitere Mengen von Klassen bzw. Schnittstellen (angedeutet durch gestrichelte Rechtecke):

- *Nicht eingerahmte Klassen* stammen aus der Java-Standard-Bibliothek bzw. der EJB-Spezifikation und gehören daher nicht zur Applikationsserver-Implementierung selbst. Die Klassen `UserTransaction`, `Xid` und `XAResource` wurden bereits in Abschnitt 3.1 angesprochen. Die Methoden `start()` und `end()` von `XAResource` dienen dazu, eine Servertransaktionen mit einer Datenbanktransaktion zu verknüpfen. Hierfür ruft die Servertransaktion (Klasse `ServerTransaction`) die Methoden `start()` und `end()` jeweils mit der `Xid`-Instanz auf, durch die die Servertransaktion identifiziert wird. Die übrigen Methoden dienen gemäß Abschnitt 3.1 zur Durchführung eines Zwei-Phasen-Commit-Protokolls.

[17] Sie ist also nicht Teil des Applikationsserver-Rahmenwerks selbst.

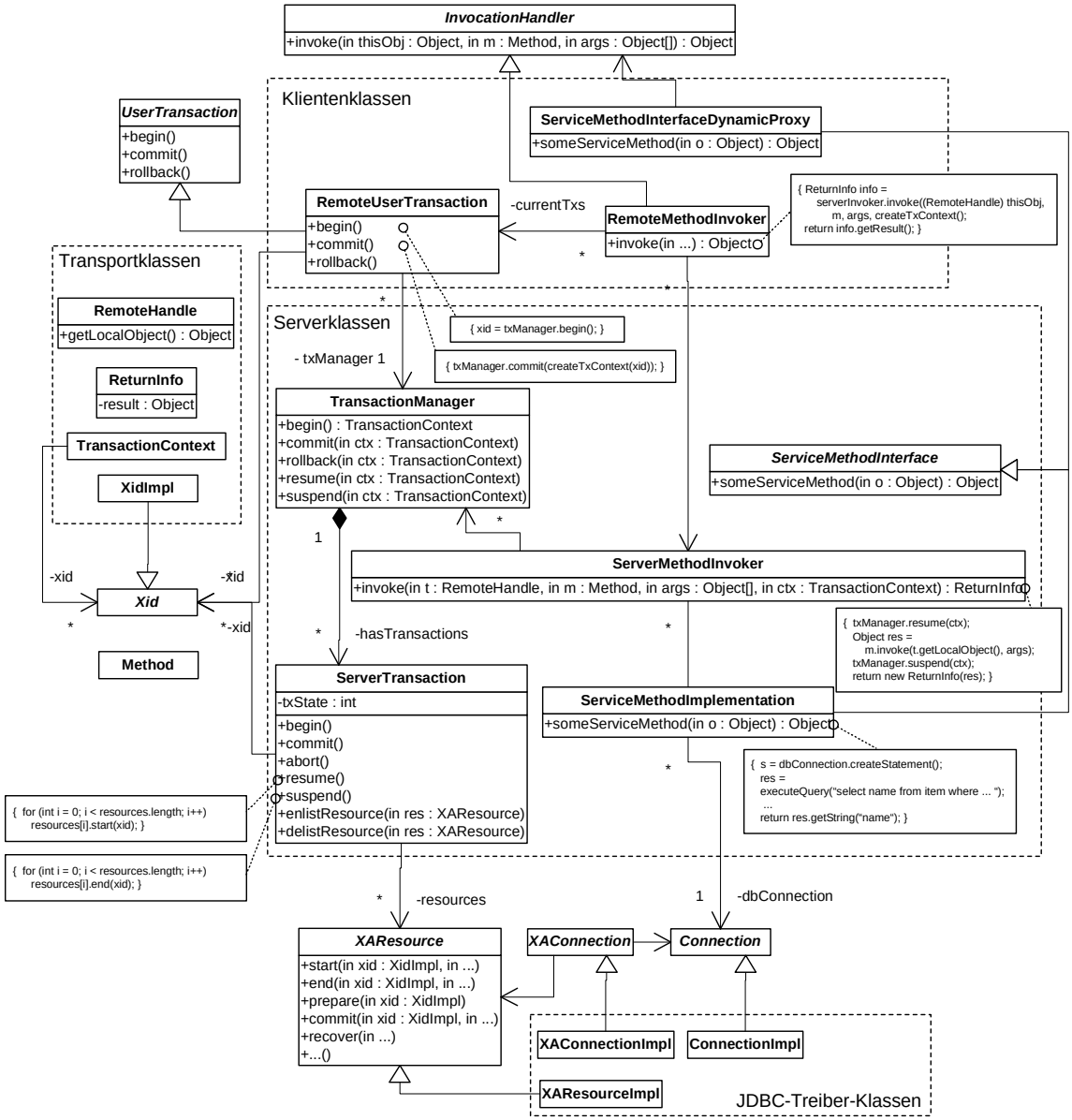

Abbildung 7: Vereinfachte UML-Darstellung einer EJB-basierten Applikationsserver-Implementierung, die Klententransaktionen unterstützt

Als weitere Standardklassen bzw. -schnittstellen kommen `Connection` und `XAConnection` in Abbildung 7 vor. `Connection` (genauer `java.sql.Connection`) erlaubt SQL-Anfragen an das relationale Datenbanksystem zu senden. Ein entsprechender Zugriff kommt häufig in der Implementierung von EJB-Methoden vor (im Beispiel durch `ServiceMethodImplementation.someServiceMethod()`).

`XAConnection` (genauer `javax.sql.XAConnection`) stellt eine Beziehung zwischen einer JDBC-Verbindung (eine Instanz von `Connection`) und einer `XAResource`-Instanz her. So können für eine JDBC-Verbindung, die gerade in einer EJB-Methodenausführung verwendet wird, über die XA-Schnittstelle entsprechende Datenbanktransaktionen gestartet, fortgeführt bzw. beendet werden. Der Transaktionsverwalter (Klasse `TransactionManager`) stellt dabei sicher, dass für die EJB-Methodenausführung die „richtige" Datenbanktransaktion fortgeführt wird. Die Zuordnung von Benutzertransaktionen, Servertransaktionen und Datenbanktransaktionen (oder allgemeiner Ressourcentransaktionen) erfolgt im System durchgängig über `Xid`-Instanzen.

Die Standardklasse `Method` (genauer `java.lang.reflect.Method`) dient zur generischen Repräsentation bzw. Ausführung von Methoden über das Java-eigene Metaobjekt-Protokoll.

`InvocationHandler` (genauer `java.lang.reflect.InvocationHandler`) unterstützt die Umsetzung des Enfwurfsmusters „Dynamischer Proxy" ([8]). Auf der Klientenseite liegt die Dienstschnittstelle zu einer EJB also als dynamische Proxyklasse vor. Beispielsweise wurde in der Abbildung das Java-Interface `ServiceMethodInterface` als Dienstschnittstelle gewählt und `ServiceMethodInterfaceDynamicProxy` wurde zur Darstellung der zugehörigen dynamischen Proxyklasse eingesetzt. In Wirklichkeit ist die Proxyklasse selbst aber namenlos – sie wird zu Systemlaufzeit erzeugt.

- Die *Klientenklassen* sind für den klientenseitigen Zugriff auf den Applikationsserver notwendig. Wie bereits erläutert, ermöglicht `ServiceMethodInterfaceDynamicProxy` den Aufruf von (entfernten) Dienstmethoden vom Klientencode aus. Die Struktur der enthaltenen Methoden hängt von der entsprechenden Schnittstelle ab; letztere ist in der Abbildung durch das Java-Interface `ServiceMethodInterface` repräsentiert.

  `RemoteUserTransaction` implementiert Benutzertransaktionen auf der Klientenseite. Entsprechende Instanzen fungieren als Statthalter für Servertransaktionen; die Eins-Zu-Eins-Zuordnung zwischen Benutzertransaktionen und Servertransaktionen erfolgt dabei, durch `Xid`-Objekte. `RemoteUserTransaction` delegiert die Aufrufe zur Demarkation von Benutzertransaktionen als entfernte Methodenaufrufe zum Server. Dort werden sie vom Transaktionsmanager (ein Singleton der Klasse `TransactionManager`) verarbeitet.

  `RemoteMethodInvoker` ist für die Weiterleitung von EJB-Methodenaufrufen verantwortlich. Letztere erhalten durch den dynamischen Proxy-Ansatz eine generische Struktur und sind mittels der Argumente `m`, `this`, `args` der `invoke()`-Methode eindeutig beschrieben. Vor der Übermittlung der entsprechenden Werte an den Server fügt `RemoteMethodInvoker` noch den Transaktionskontext in einen entfernten Methodenaufruf ein. Der Transaktionskontext enthält insbesondere die `Xid` zur Identifikation der Transaktion, innerhalb derer der Dienstmethodenaufruf beim Klienten stattfindet. Die `Xid` kann über eine Assoziation zur „aktuellen" Be-

nutzertransaktion ermittelt werden. Die Details zum Finden der aktuellen Benutzertransaktion wurden der Einfachheit halber in Abbildung 7 ausgelassen. Auf der Klientenseite wird das `this`-Objekt für einen (entfernten) Dienstmethodenaufruf übrigens nicht direkt referenziert – es wird lediglich durch einen Identifikator (eine Instanz der Klasse `RemoteHandle`) gekennzeichnet. Der Identifikator wird nach der Übermittlung auf der Serverseite durch das echte Dienstobjekt (also ein EJB-Objekt) ersetzt.

- Die *Transportklassen* werden zur Datenübermittlung bei entfernten Methodenaufrufen zwischen Klient und Server genutzt. Die Klasse `TransactionContext` enthält dabei alle Information zur Verarbeitung von Benutzertransaktionen auf der Serverseite. Im vorliegenden Fall unfasst dies lediglich die `Xid` – zur Realisierung eines Protokolls für transaktionales Methoden-Caching muss `TransactionContext` aber erweitert und mit zusätzlichen Informationen versehen werden.

  `ReturnInfo` liefert das Resultat eines Dienstmethodenaufrufs an den Klienten zurück. Das eigentliche Aufrufresultat ist im Attribut `result` enthalten. Für das transaktionale Methoden-Caching können hier ebenfalls Zusatzinformationen untergebracht werde, die nach der Ausführung einer Dienstmethode vom Server zum Klient zu übermitteln sind.

  `XidImpl` ist lediglich eine Implementierung des Java-Interfaces `Xid`. (Die genaue Implementierung einer `Xid` lassen der Java- bzw. der XA-Standard also offen.) Die Funktion der Klasse `RemoteHandle` wurde bereits behandelt.

- Von den *Serverklassen* zeigt Abbildung 7 nur jene, die für die Transaktionsverarbeitung im Applikationsserver besonders interessant sind. Die Klasse `TransactionManager` bearbeitet potentiell verteilte Anwendungstransaktionen auf der Serverseite. Neben den Methoden `begin()`, `commit()` und `rollback()` enthält sie auch Methoden zum Unterbrechen bzw. Fortführen von Anwendungstransaktionen (`suspend()` und `resume()`).

  Die beiden Methoden werden von einer Instanz der Klasse `ServerMethodInvoker` genutzt, wenn diese einen Dienstmethodenaufruf über `invoke()` durchführt. Zur Identifikation der entsprechenden Transaktionen wird `resume()` und `suspend()` jener Transaktionskontext übergeben, den die `ServerMethodInvoker`-Instanz vom Klienten erhalten hat. `ServerMethodInvoker.invoke()` bestimmt (über das übermittelte `RemoteHandle`) auch das lokale EJB-Objekt. Über die Klasse `java.lang.reflect.Method` kommt es dann zum eigentlichen Dienstmethodenaufruf. `ServerMethodInvoker.invoke()` verpackt das berechnete Resultat in einem `ReturnInfo`-Objekt und schickt dieses an den Klienten zurück.

  Das Singleton `TransactionManager` kennt alle laufenden Server-seitigen Transaktion über die Assoziation `hasTransactions`. Mit Hilfe der `Xid` einer Demarkationsmethode wie etwa `TransactionManager.commit()` kann der Transaktionsmanager ein zugehöriges `ServerTransaction`-Objekt bestimmen und einen

entsprechenden Methodenaufruf an die Servertransaktion delegieren. Die letztere informiert ihrerseits alle beteiligten XA-Ressourcen über das Unterbrechen, Fortführen oder Beenden zugeordneter Ressourcentransaktionen. Dazu verwendet die Servertransaktion die `Xid`, durch die sie sich selbst identifiziert. Entsprechende XA-Ressourcen werden mit der Methode `enlistRessources()` registriert. Wie im Beispiel angedeutet, repräsentieren die XA-Ressourcen meist (JDBC-)Verbindungen zu XA-fähigen Datenbanken. Ist einer Servertransaktion mehr als eine XA-Ressource zugeordnet, so handelt es sich offenbar um eine verteilte Transaktion. Es obliegt dann der Servertransaktion, durch entsprechende Aufrufe von `XARessource.prepare()` ein Zwei-Phasen-Commit-Protokoll für die beteiligten XA-Ressourcen durchzuführen.

5.2 Erweiterung um Methoden-Cache und m-Planer

Abbildung 8 zeigt vereinfacht die wichtigsten Erweiterungen der Basisinfrastruktur des letzten Abschnitts, so dass transaktionales Methoden-Caching ermöglicht wird. Es tauchen dabei Veränderungen an sehr vielen Klassen auf, die man schon aus Abbildung 7 kennt. Der Übersicht halber wurde auf Anmerkungen zu Methodenimplementierungen zum Großteil verzichtet. Die folgenden Paragraphen erläutern aber entsprechende Details.

Als wichtigster Unterschied zu Abbildung 7 gibt es nun zwei weitere (durch gestrichelte Rechtecke markierte) Mengen von Klassen: Auf der Klientenseite sorgen die *Methoden-Cache-Klassen* für das konsistente Vorhalten von Methodenresultaten. Auf der Serverseite stellen die *m-Planer-Klassen* die Serialisierbarkeit von Transaktionen im Zusammenhang mit dem Einsatz des Methoden-Caches sicher.

Entsprechend der Architektur aus Abbildung 5 in Abschnitt 3.2 befindet sich der m-Planer zwischen den Transaktionsverwalterklassen des Applikationsservers und den JDBC-Treiber-Klassen. Letztere sind im Gegensatz zu Abbildung 7 hier nicht illustriert. Der m-Planer präsentiert sich gegenüber dem Applikationsserver im Wesentlichen als eine XA-Ressource und eine Datenbankverbindung und wirkt wie ein delegierender XA-fähiger JDBC-Treiber. Die entsprechenden Klassen `MXAResource` und `MConnection` erfüllen daher das Entwurfsmuster „Dekorierer“ ([11]). Sie geben einerseits entsprechende Methodenaufrufe an einen unterliegenden „echten“ JDBC-Treiber weiter. Darüber hinaus sorgen sie aber noch für die konsistente Einplanung von m-Operationen, indem sie entsprechende Methoden aus der Klasse `MTransaction` aufrufen. `MTransaction` ist die Repräsentation einer Anwendungstransaktion innerhalb des m-Planers – Serialisierbarkeitsprotokolle für Methoden-Cache-Transaktionen manifestieren sich, wie noch deutlich wird, zum Großteil in dieser Klasse. Details hierzu folgen in den nächsten Abschnitten und in Kapitel 7.

Wie bereits in Abschnitt 3.2 erwähnt, benötigt das System für das Einplanen von Methodenoperationen sowie das Invalidieren von Methodenresultaten im Cache noch Zusatzinformationen, die zwischen dem Methoden-Cache (auf der Klientenseite) und dem m-Planer ausgetauscht werden müssen. Als Basis für den Transport dieser Informationen dienen die bereits bekannten Transportklassen `ReturnInfo` und

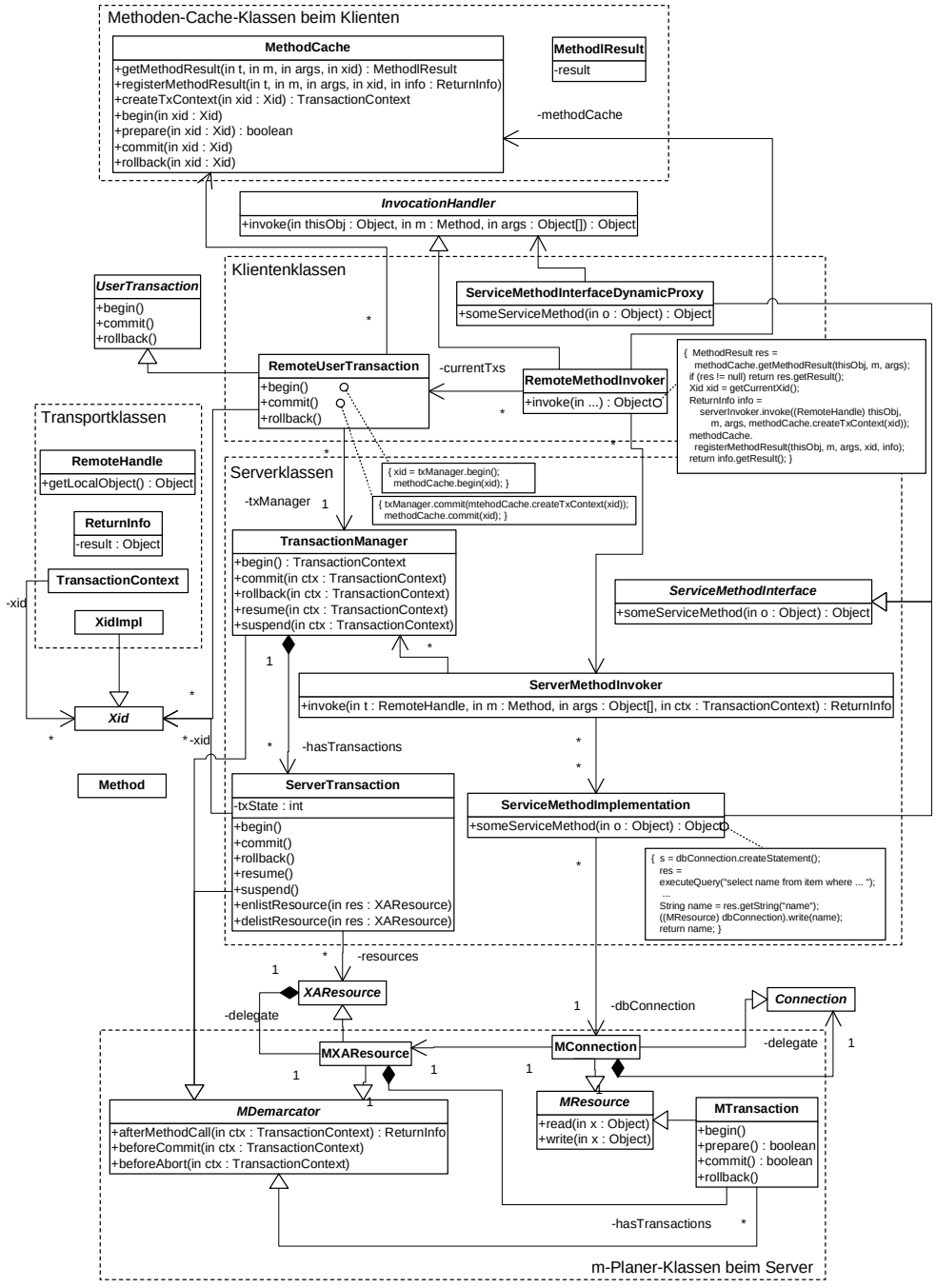

Abbildung 8: Erweiterung einer EJB-basierten Applikationsserver-Implementierung zur Unterstützung von Methoden-Caching in Klententransaktionen

TransactionContext.

Um dem m-Planer die Zusatzinformationen zukommen zu lassen, implementiert die Klasse MXAResource neben XAResource auch noch das Interface MDemarcator. Die Methoden von MDemarcator erlauben die Übergabe von Transaktionskontexten vor dem Beenden einer Transaktion (durch beforeCommit() und beforeAbort()) und nach der Ausführung von EJB-Methoden (durch afterMethodCall()). afterMethodCall() liefert als Resultat auch gleich das ReturnInfo-Objekt, welches nach einer EJB-Methodenausführung zum Klienten gelangen soll. Es enthält bereits alle Zusatzinformationen die der m-Planer im Rahmen des Methodenaufrufs dem Methoden-Cache zukommen lassen möchte. Lediglich das eigentliche Resultat des Methodenaufrufs ist noch zu ergänzen (durch Setzen des Attributs ReturnInfo.result).

Die Klasse ServerTransaction ruft die MDemarcator-Methoden aus der Klasse MXAResource auf, sofern eine entsprechende XA-Ressource bei einer Servertransaktion registriert ist. Im Gegensatz zu Abbildung 7 implementiert ServerTransaction nun ebenfalls das MDemarcator-Interface. Die Implementierung einer entsprechenden Methode soll am Beispiel ServerTransaction.beforeCommit() verdeutlicht werden:[18]

```
1  class ServerTransaction {
2    ...
3    void beforeCommit(TransactionContext ctx) {
4      for (int i = 0; i < resources.length; i++) {
5        if (resources[i] instanceof MDemarcator) {
6          // Delegation des Aufrufs an beteiligte MXAResource-Instanzen:
7          ((MDemarcator) resources[i]).beforeCommit(ctx); } } }
8        }
9      }
10   }
11 }
```

Die MDemarcator-Methoden aus ServerTransaction werden ihrerseits vom Transaktionsverwalter verwendet, der ebenfalls MDemarcator implementiert. Der Transaktionsverwalter ruft seine eigenen Implementierungen von beforeAbort() und beforeCommit() zu Beginn von commit() bzw. abort() auf. Die before-Methoden delegieren ihre Aufgabe dabei lediglich an die entsprechende Servertransaktion. Im Gegensatz dazu wird TransactionManager.afterMethodCall() von Instanzen der Klasse ServerMethodInvoker angesprochen. Der entsprechend angepasste Pseudocode der Methode ServerMethodInvoker.invoke() gestaltet sich, wie folgt:

```
1  class ServerMethodInvoker {
2    ...
3    ReturnInfo invoke(Object thisObj, Method m, Object[] args,
4                      TransactionContext ctx) {
5      txManager.resume(ctx);
6      Object res = m.invoke(t.getLocalObject(), args);
7      ReturnInfo info = txManager.afterMethodCall(ctx);
```

[18] Idealerweise sollte man die Klasse ServerTransaction aus Abbildung 7 zu diesem Zweck in einer Unterklasse erweitern. Der Einfachheit halber vernachlässigt die vorliegende Darstellung solche Details. Ähnliches gilt auch für die Klassen TransactionManager, ServerMethodInvoker und RemoteMethodInvoker, wie man weiter unten sieht.

```
 8       info.result = res;
 9       txManager.suspend(ctx);
10       return info;
11     }
12   }
```

Für die Einplanung von m-Operationen in Datenbanktransaktionen muss der m-Planer die Datenelemente kennen, die eine EJB-Methode liest oder schreibt (siehe hierzu auch Abbildung 4). Dem unterliegenden Datenbankverwaltungssystem werden im Zuge der Anfragebearbeitung über JDBC natürlich alle solche Datenelemente (wie etwa Tabelleneinträge) bekannt. Für die vorgestellte Implementierung des m-Planers trifft dies aber nicht zu, denn er delegiert SQL-Anfragen lediglich, ohne die Details von deren Ausführung zu berücksichtigen. Eine ausgereifte Implementierung des m-Planers sollte daher am besten mit dem Datenbankverwaltungssystem integriert sein, so dass er die Liste der während einer SQL-Anfrage verwendeten Datenelemente vom Datenbankverwaltungssystem erhält.

Eine entsprechende Integrationsschnittstelle ist bei existierenden Datenbankverwaltungssystemen leider nicht vorhanden.[19] Als prototypische Lösungsalternative, gibt ein Anwendungsprogrammierer Datenelementzugriffe explizit innerhalb des Codes von entsprechenden EJB-Methoden an. Dies geschieht mit Hilfe der Methoden `MResource.read()` und `MResource.write()`, die von der Klasse `MConnection` implementiert werden. Datenelemente können beliebige Java-Objekte sein, sofern sie sich als Schlüssel von Hash-Tabellen eignen. Sinnvoll sind etwa Java-Repräsentationen von Schlüsselwerten aus Tabellenzeilen, die in einer JDBC-Anfrage gelesen oder geschrieben werden. Letztlich muss also der Programmierer der EJB-Methoden geeignete Objekte bestimmen.

In Abbildung 8 wurde die EJB-Methode `someServiceMethod()` beispielsweise so angepasst, dass der Aufruf `((MResource) dbConnection).write(name)` das durch die Zeichenkette `name` identifizierte Datenelement beim m-Planer als „geschrieben“ vermerkt.

Bei den Klientenklassen in Abbildung 8 wurden auch `RemoteUserTransaction` und `RemoteMethodInvoker` angepasst. Die Implementierungen von `begin()`, `commit()` und `rollback()` aus `RemoteUserTransaction` informieren nun den Methoden-Cache über die entsprechenden Ereignisse. Dazu stehen in der Klasse `MethodCache` die Methoden `begin()`, `commit()` und `rollback()` bereit. Die Methode `RemoteMethodInvoker.invoke()` versucht Methodenresultate aus dem Methoden-Cache zu holen, bevor sie einen zugehörigen EJB-Methodenaufruf an den Applikationsserver sendet. Nachdem der Aufruf beim Server erfolgt ist, registriert `invoke()` das Resultat beim Methoden-Cache. Dazu übergibt die Methode alle Informationen zum getätigten Aufruf an `registerMethodResult()`. Zu beachten ist auch, dass der Methoden-Cache den Transaktionskontext für den EJB-Methodenaufruf mittels `createTxContext()` erzeugt. Auf diese Weise gelangen entsprechende Informationen vom Methoden-Cache schließlich bis zum m-Planer.

[19] Man bedenke, dass etwa eine Analyse der Log-Datei hierfür nicht ausreicht, weil das Datenbankverwaltungssystems dort üblicherweise nur Datenänderungen (nicht aber Lesezugriffe) aufzeichnet.

5.3 Details beim Methoden-Cache

Dieser Abschnitt vertieft die Entwurfsdetails der bereits angesprochenen Methoden-Cache-Klassen aus Abbildung 8. Abbildung 9 zeigt ein detaillierteres UML-Klassendiagramm des Methoden-Caches (auf der Klientenseite) mit Java-Pseudocode für viele der angegebenen Methoden. Neben `MethodCache` und `MethodResult` umfasst der Cache demnach noch einige weitere Klassen.

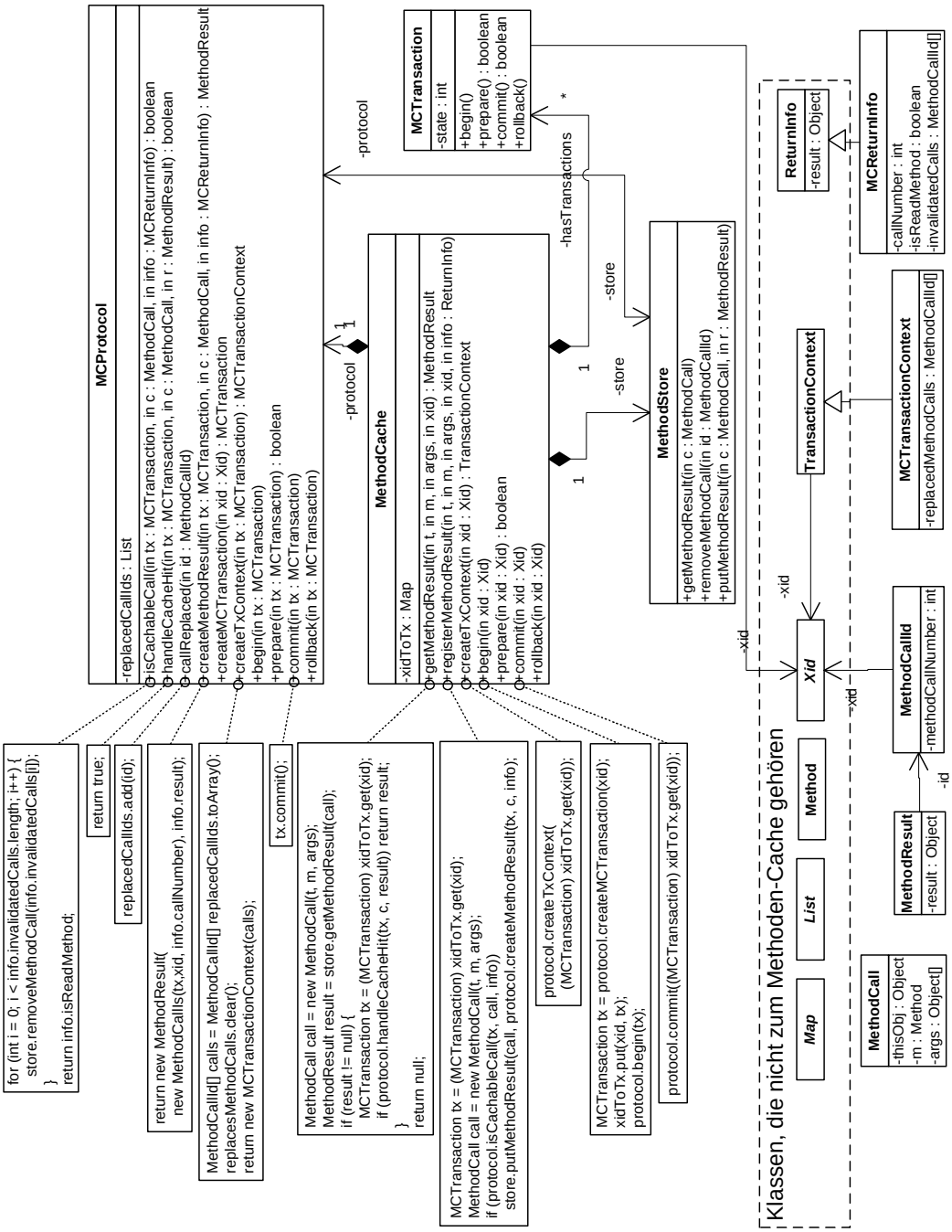

Abbildung 9: UML-Klassendiagramm für den Methoden-Cache

Allgemein sei vorwegschickt, dass der vorliegende Entwurf darauf ausgelegt ist, für Serialisierbarkeits- bzw. Fehlererholungsprotokolle erweitert zu werden. Die folgenden Paragraphen werden dies verdeutlichen.

Wie bereits im vorigen Abschnitt angesprochen, wurden die Transportklassen `TransactionContext` und `ReturnInfo` erweitert, um Zusatzinformationen zwischen dem Methoden-Cache und dem m-Planer auszutauschen.

Mit dem Attribut `MCTransactionContext.replacedMethodCalls` wird dem m-Planer eine Liste von Methodenaufrufen mitgeteilt, deren zugeordnete Methodenresultate seit dem letzten entfernten EJB-Methodenaufruf aus dem Cache-Speicher des Klienten verdrängt wurden. Der m-Planer erhält jedoch nicht die verdrängten Methodenaufrufe selbst (bestehend aus `this`-Objekt, Methode und Argumentliste) sondern lediglich Identifikatoren für diese Aufrufe. Entsprechende Instanzen des Typs `MethodCallId` bestimmen einen EJB-Methodenaufruf eindeutig durch die `Xid` der Transaktion, innerhalb derer der Aufruf stattfand sowie eine Nummer für die Stelligkeit des Aufrufs innerhalb der Transaktion. Die Nummer (Attribut `MethodCallId.methodCallNumber`) wird beim m-Planer für tatsächlich ausgeführte (also nicht gecachte) EJB-Methodenaufrufe vergeben. Sie ist innerhalb einer Transaktion eindeutig, da alle Aufrufe der Transaktion sequentiell geordnet sein müssen.[20]

Die Klasse `MCReturnInfo` enthält Zusatzinformationen, die nach einem entfernten EJB-Methodenaufruf vom m-Planer zum Methoden-Cache gelangen sollen. Zum einen erhält der Methoden-Cache durch das Attribut `callNumber` die im vorigen Paragraphen angesprochene Nummer des gerade durchgeführten Aufrufs.

Zum anderen zeigt das Attribut `MCReturnInfo.isReadMethod` dem Methoden-Cache an, ob der Methodenaufruf nur Leseoperationen auf den unterliegenden XA-Ressourcen durchgeführt hat oder nicht. Nur im ersten Fall darf er das Methodenresultat (möglicherweise) einlagern.

`MCReturnInfo.invalidatedCalls` ist eine Liste von Methodenaufruf-Identifikatoren für Methodenresultate, die vom m-Planer für ungültig erklärt wurden, aber beim Klienten noch gecacht sind. Der häufigste Grund hierfür sind Schreiboperationen auf den XA-Ressourcen, die mit Leseoperationen aus der Berechnung der gecachten Methodenresultate in Konflikt stehen.

Die Attribute `isReadMethod` und `invalidatedCalls` nutzt der Methoden-Cache in der Methode `MCProtocol.isCachableCall()`, wie der Pseudocode in Abbildung 9 zeigt. `isCachableCall()` iteriert über die empfangene `invalidatedCalls`-Reihung und entfernt die entsprechenden Einträge aus dem Cache-Speicher.

Die Klasse `MethodCache`, die bereits im vorigen Abschnitt angesprochen wurde, dient als Fassade (Entwurfsmuster „Fassade“, [11]) für die Klientenklassen, die mit dem Methoden-Cache interagieren. `MethodCache` ist ein Singleton und aggregiert die Klassen `MethodStore`, `MCTransaction` und `MCProtocol`.

Eine `MethodStore`-Instanz repräsentiert den Cache-Speicher selbst. Die dort angebotenen Methoden dienen zum Finden, Einlagern und Löschen von Methodenresulta-

[20] Bei herkömmlichen Spezifikationen für Anwendungstransaktionen finden diese in genau einem Prozessfaden (Thread) statt.

ten. Intern (und in der Abbildung nicht gezeigt) nutzt die Klasse Hash-Tabellen um Methodenaufrufe (repräsentiert durch `MethodCall`- bzw. `MethodCallId`-Objekte) mit Methodenresultaten zu assoziieren. `MethodStore` verwendet eine LRU-Strategie zum Verdrängen gespeicherter Einträge. Falls ein Element verdrängt werden muss, benachrichtigt die Klasse das zugeordnete `MCProtocol`-Objekt darüber mittels der Methode `MCProtocol.callReplaced()`. `callReplaced()` sichert den Identifikator des verdrängten Methodenaufrufs in einer Liste (Attribut `MCProtocol.replacedCallIds`) damit dieser später zum m-Planer propagiert werden kann. (Nachdem letzteres geschehen ist, wird der Eintrag aus der Liste entfernt.)

`MCTransaction`-Objekte stellen Anwendungstransaktionen im Kontext des Methoden-Caches dar. Die entsprechende Klasse kann für die Zwecke eines Serialisierbarkeits- bzw. Fehlererholungsprotokolls erweitert werden und dann transaktionsspezifische Informationen aufnehmen. Beispiele hierfür findet man in Kapitel 6. `MCTransaction` fungiert als eine Basisklasse für solche Erweiterungen und verwaltet davon abgesehen lediglich den Transaktionszustand über das Attribut `state`. Die im Wesentlichen trivialen Implementierungen von `begin()`, `prepare()`, `commit()` und `rollback()` prüfen und verändern diesen Zustand.

`MCProtocol` übernimmt spezifische Aufgaben zur konsistenten Verwaltung des Methoden-Caches, die sich je nach verwendetem Serialisierbarkeitsprotokoll unterscheiden. Die Implementierungen der in Kapitel 7 behandelten Serialisierbarkeitsprotokolle erweitern daher `MCProtocol` und überschreiben dort vorhandene Methoden. Ähnlich wie `MCTansaction` dient `MCProtocol` also Basisklasse, allerdings erfüllt sie auch schon wichtige Aufgaben für das Caching. Man hätte `MCProtocol` und `MethodCache` auch in einer einzigen Klasse zusammenfassen können. Die vorgenommene Trennung hat vor allem Vorteil, dass sich Protokolle zur Laufzeit austauschen lassen – die `MethodCache`-Fassade bleibt davon unberührt.

Zu den Methoden aus `MCProtocol`: Die Funktion von `isCachableCall()` und `callReplaced()` wurde bereits weiter oben behandelt. `handleCacheHit()` steuert die Reaktion des Methoden-Caches bei einem Cache-Treffer: Falls die Methode `true` liefert, wird das gefundene Resultat tatsächlich ausgeliefert, ansonsten gelangt ein entsprechender EJB-Methodenaufruf doch zum Applikationsserver. Die Basisimplementierung von `handleCacheHit()` ist trivial und liefert immer `true`.

Die Fabrikmethode `createMethodResult()` erzeugt geeignete Objekte zur Speicherung von Methodenresultaten. Falls ein Protokoll zusätzliche Informationen in Methodenresultaten speichern muss, genügt es `createMethodResult()` zu überschreiben und eine Unterklasse von `MethodResult` zu bilden. Ähnliches gilt für die Fabrikmethoden `createMCTransaction()` und `createTxContext()`. Die vorliegende Implementierung von `createTxContext()` fügt Identifikatoren verdrängter Methodenaufrufe in einen Transaktionskontext ein, so dass diese zum m-Planer gelangen.

Die Basisimplementierungen der Demarkationsmethoden, wie etwa `MCProtocol.commit()`, delegieren lediglich die entsprechende Operation an die zuständige `MCTransaction`-Instanz. Auch in diesem Zusammenhang kann man in Unterklassen von `MCProtocol` Anpassungen vornehmen.

5.4 Details beim m-Planer

Abbildung 10 illustriert die Klassen des m-Planers mit Pseudocode für viele wichtige Methoden.

Wie bereits in Abschnitt 5.2 beschrieben, repräsentieren Instanzen der Klasse `MTransaction` Anwendungstransaktionen aus der Sicht des m-Planers. Sie sichern die konsistente Bearbeitung von m-Operationen zu.

Fast alle der in `MTransaction` aufgeführten Methoden werden von `MXAResource`-Objekten aufgerufen. `MXAResource` nutzt die interne Tabelle `xidsToMTransactions`, um `Xids` eingehender Methodenaufrufe entsprechenden `MTransaction`-Instanzen zuzuordnen. (Es handelt sich hierbei um eine Eins-Zu-Eins-Zuordnung.) Aufrufe aus der `XAResource`-Schnittstelle leitet `MXAResource` sowohl an das Delegat (über die Assoziation `delegate`) als auch an die entsprechende `MTransaction`-Instanz weiter. `MXAResource.begin()` erzeugt darüber hinaus ein neues `MTransaction`-Objekt und legt es in der `xidsToMTransactions`-Tabelle ab (mit dem `Xid`-Argument der Methode als Schlüssel). Aufrufe aus der `MDemarcator`-Schnittstelle delegiert `MXAResource` ausschließlich an das betreffende `MTransaction`-Objekt. Die Implementierungsteile von `MXAResource` fehlen in Abbildung 10 zu Gunsten der Übersichtlichkeit.

Als Basisklasse für Serialisierbarkeitprotokolle hat die Klasse `MTransaction` drei wichtige Aufgaben:

- Sie bestimmt Listen zu invalidierender gecachter Methodenaufrufe und propagiert die Identifikatoren dieser Aufrufe über `MCReturnInfo`-Objekte an den Methoden-Cache.
- `MTransaction` stellt fest, ob ein verarbeiteter EJB-Methodenaufruf auschließlich Datenelemente (auf der zugeordneten XA-Ressource) liest oder nicht. Dem entsprechend setzt die Klasse das Attribut `isReadMethod` in `MCReturnInfo`-Objekten. Zusätzlich vergibt sie für EJB-Methodenaufrufe eine (eindeutige) Nummer.
- Die Klasse nimmt Identifikatoren verdrängter Methodenaufrufe aus Transaktionskontexten entgegen und entfernt damit zusammenhängende Einträge aus den Datenstrukturen des m-Planers.

Noch völlig unabhängig von einem konkreten Serialisierbarkeitsprotokoll soll ein gecachter Methodenaufruf beim Klienten invalidiert werden, wenn das entsprechende Resultat durch eine Zustandsveränderung bei der XA-Ressource ungültig wird. Dies ist der Fall, wenn *nach der Ausführung des gecachten Methodenaufrufs* bei der XA-Ressource ein Datenelement geschrieben wird, welches der gecachte Methodenaufruf selbst bei seiner ursprünglichen Ausführung gelesen hat.[21]

[21] Man erinnere sich, dass das System gelesene und geschriebene Datenelemente mit Hilfe der Methoden `MResource.read()` und `MResource.write()` bestimmen kann.

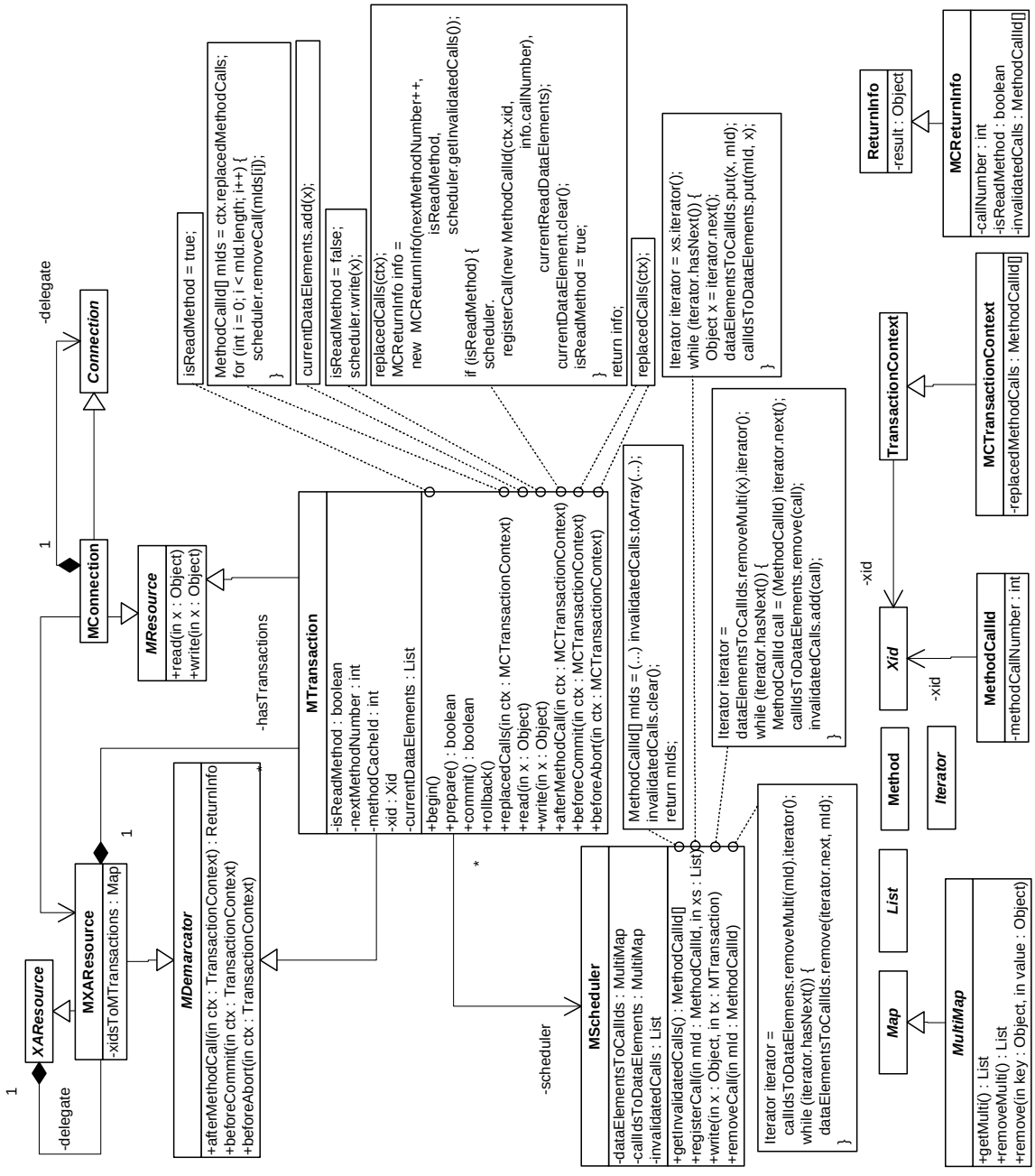

Abbildung 10: UML-Klassendiagramm für den m-Planer

Um derartige Konflikte festzustellen, muss der m-Planer zu jedem gecachten Methodenaufruf die Liste der dabei gelesenen Datenelemente speichern. Dies geschieht mit Hilfe der Hash-Tabellen `dataElementsToCallIds` und `callIdsDataElements` der Klasse `MScheduler`. Die Tabellen assoziieren entsprechend gelesene Datenelemente (des Typs `Object`) mit `MethodCallId`-Objekten. Da es m-Zu-n-Beziehungen zwischen Datenelementen und gecachten Methodenaufrufen geben kann, implementieren die Hash-Tabellen das Interface `MultiMap`. So kann man einem Tabellenschlüssel mehrere Werte (also eine Liste von Werten) zuordnen. In Abbildung 10 bietet `MultiMap` Methoden zur Bearbeitung von Wertlisten an.

Die Klasse `MScheduler` behandelt Informationen, die für *alle* Transaktionen gemeinsam von Bedeutung sind. Daher verweisen alle `MTransaction`-Objekte über die Assoziation `scheduler` auf das gleiche `MScheduler`-Objekt. (Es handelt sich dabei also um eine Singleton.) Mit der Methode `MScheduler.registerCall()` kann eine Transaktion eine Lesemethode und die darin gelesenen Datenelemente für die Tabellen des `MScheduler`-Singletons registrieren. `MScheduler.removeCall()` erlaubt hingegen, Einträge aus den Hash-Tabellen zu entfernen. `MScheduler.write()` löscht alle Lesemethoden aus den Hash-Tabellen, die das übergebene Datenelement gelesen haben. Die gelöschten `MethodCallId`-Objekte werden aber zunächst in der Liste `MScheduler.invalidatedCalls` gesichert. `MScheduler.getInvalidatedCalls()` liest schließlich die `invalidatedCalls`-Liste aus und leert sie. Nach einem Aufruf von `getInvalidatedCalls()` schickt der m-Planer die entsprechenden `MethodCallId`-Objekte zur Invalidierung an den Klienten.

In der Klasse `MTransaction` zeigt das Attribut `isReadMethod` an, ob eine ausgeführte EJB-Methode schreibt oder liest. Immer zum Beginn einer etwaigen EJB-Methodenausführung, also am Ende der Methoden `begin()` und am Ende von `afterMethodCall()`, unterstellt das System, dass die EJB-Methode tatsächlich nur liest. Ist dies nicht der Fall, wird während der EJB-Methodenausführung, `MTransaction.write()` aufgerufen und `isReadMethod` auf `false` gesetzt. Weiter invalidiert `write()` durch `scheduler.write(x)` gecachte Lesemethoden, die `x` zuvor gelesen haben. Im Falle einer Lesemethode müssen alle gelesenen Datenelemente für die Tabelleneinträge beim `MScheduler` gesammelt werden. Dies erreicht `read()` mit Hilfe der Liste `currentDataElements`.

Am Ende einer EJB-Methodenausführung erledigt `MTransaction.afterMethodCall()` die wichtigste Arbeit: Zunächst sorgt die Methode dafür, dass verdrängte Methodenaufrufe aus den `MScheduler`-Tabellen entfernt werden. Dann konstruiert sie das Rückgabeobjekt für den Klienten mit neuer Methodennummer, passendem Lesemethodenattribut sowie der aktuellen Liste invalidierter Methodenaufrufe. War es wirklich ein Lesemethodenaufruf, so wird dieser registriert. Abschließend setzt die Methode noch die Attribute `isReadMethod` und `currentDataElements` im `MTransaction`-Objekt zurück.

5.5 Speicherverwaltung bei vielen Klienten

Zur Vereinfachung der Darstellung wurde bisher unterstellt, dass es nur einen Methoden-Cache und damit nur einen Klienten gibt. Der vorgestellte Entwurf kann aber problemlos zur Behandlung vieler Klienten angepasst werden. Hierzu vergibt der m-Planer für jeden Methoden-Cache einen eindeutigen Identifikator. Um einem Klienten, die für ihn relevante Liste von invalidierten Methodenaufrufen zu kommen zu lassen, führt man den Identifikator sowohl in Transaktionskontexten als auch in Server-seitigen `MethodCallId`-Objekten mit.

Offenbar erzeugt der vorgestellte Ansatz für jedes gecachte Methodenresultat nicht nur Einträge beim Klienten sondern auch beim Server (nämlich in den Tabellen der Klasse `MScheduler`). Ein entsprechendes System ist deshalb nicht auf eine beliebig große Zahl von Klienten ausgerichtet, da andernfalls der Serverspeicher knapp wird. Genauer liegt der benötigte Serverspeicher in $O(n \cdot k)$ mit der n als der Anzahl der Klienten und k als der Speichergröße eines (klientenseitigen) Methoden-Caches.

Um Speicherproblemen von vornherein aus dem Weg zu gehen, sollten die Tabellen `MScheduler.dataElementsToCallIds` und `MScheduler.callIdsDataElements` daher mit einer Verdrängungsstrategie, etwa LRU, ausgerüstet sein. Verdrängte Einträge legt der m-Planer in der `invalidatedCalls`-Liste ab – sie sind bei den entsprechenden Klienten wie gewöhnliche, invalidierte Methodenaufrufe zu behandeln.

Falls ein Klient längere Zeit keine EJB-Methodenaufrufe durchführt, kann es vorkommen, dass die `invalidatedCalls`-Liste für diesen Klienten beim Server sehr groß wird. Überalterte Einträge müssen dann sogar aus dieser Liste entfernt werden, noch bevor sie zur Invalidierung zum entsprechenden Klienten propagiert wurden. Im Zusammenhang mit den in Kapitel 7 vorkommenden Serialisierbarkeitsprotokollen ist diese Vorgehensweise aber unproblematisch – dies geht aus den dort vorgeschlagenen Protokollimplementierungen hervor.

Mit einer Verdrängungsstrategie für die `MScheduler`-Tabellen begrenzt man implizit auch die Menge der gültigen Methoden-Cache-Einträge bei den Klienten. Es gilt dann $\sum_{i=1}^{n} k_i \leq s$, wobei n die Anzahl der Klienten repräsentiert, k_i die Anzahl der gültigen Methoden-Cache-Einträge von Klient i und s die maximale Anzahl von Einträgen in den `MScheduler`-Tabellen.

Alle übrigen Speicherprobleme beim vorgestellten Entwurf sind nicht von konzeptioneller Natur. Sie können leicht durch technische Verfeinerungen der präsentierten Klassen eliminiert werden.

6 Eigenschaften zur Fehlererholung (Recovery)

6.1 Formalismus

Rücksetzbarkeit (Recoverability, RC), Kaskadenfreiheit (Avoidance of Cascading Aborts, ACA) und Striktheit (Strictness, ST) sind klassische Eigenschaften von Historien und haben eine große Bedeutung für die Fehlererholung (Recovery) eines Transaktionssystems. Ihre intuitive und praktische Bedeutung ist ausführlich in der Literatur zur Transakti-

onstheorie diskutiert und wird hier als bekannt vorausgesetzt (siehe etwa Abschnitt 2.4 in [4]).

Dieser Abschnitt dehnt die genannten Fehlererholungseigenschaften durch entsprechende Definitionen auf (wohl geordnete) Methoden-Cache-Historien aus. Darauf aufbauend können wichtige Aussagen zu Methoden-Cache-Historien getroffen werden, die diese Eigenschaften erfüllen. Zunächst wird die Liest-Von-Relation für (wohl geordnete) Methoden-Cache-Historien definiert.

Definition 14. *Liest-Von-Relation: Es sei $(H^{MC}, <)$ eine (wohl geordnete) Methoden-Cache-Historie mit den Transaktionen $\{(T_1, <_1), \ldots, (T_n, <_n)\}$. Eine Transaktion T_i* liest (das Datenelement) *x* von *T_j* vermittels der Operation *p, wenn gilt:*

$$\exists r_h^k[x], w_j[x] \in H^{MC} : w_j[x] < r_h^k[x] \wedge (h = i \vee m_i^{h,k} \in H^{MC}) \wedge \neg(a_j < r_h^k[x]) \wedge$$

$$\forall w_o[x] \in H^{MC} : w_j[x] < w_o[x] < r_h^k[x] \Rightarrow a_o < r_h^k[x].$$

Dabei ist $p = r_h^k[x]$, falls in der obigen Bedingung $h = i$ zutrifft und $p = m_i^{h,k}$ sonst. Man schreibt $reads(T_i, x, T_j, p)$ genau dann, wenn die obige Eigenschaft für T_i, x, T_j und die entsprechende Operation p erfüllt ist. $reads$ bildet eine vierstellige-stellige Relation, die so genannte Liest-Von-Relation.

Beispiel 7. *Für die Methoden-Cache-Historie $(H, <) = w_2[x]r_1^1[y]r_1^1[x]c_1c_2m_3^{1,1}w_3[x]c_3$ ergibt sich die Liest-Von-Relation $reads = \{(T_1, x, T_2, r_1^1[x]), (T_3, x, T_2, m_3^{1,1})\}$.*

Mit Hilfe der Liest-Von-Relation kann man typische Fehlererholungseigenschaften auch für Methoden-Cache-Historien formulieren.

Definition 15. *Rücksetzbarkeit, Kaskadenfreiheit und Striktheit von Methoden-Cache-Historien: Eine (wohl geordnete) Methoden-Cache-Historie $(H^{MC}, <)$ mit den Transaktionen $\{(T_1, <_1), \ldots, (T_n, <_n)\}$ und mit den Datenelementen D ist* rücksetzbar *bzw.* kaskadenfrei *bzw.* strikt*, wenn sie die (jeweils) folgende Eigenschaft erfüllt:*[22]

- *Rücksetzbar:*

$$\forall i, j \in \{1, \ldots, n\} : \forall x \in D : \forall p \in H^{MC} : i \neq j \wedge reads(T_i, x, T_j, p)) \wedge c_i \in H^{MC} \Rightarrow$$

$$c_j < c_i$$

- *Kaskadenfrei:*

$$\forall i, j \in \{1, \ldots, n\} : \forall x \in D : \forall p \in H^{MC} : i \neq j \wedge reads(T_i, x, T_j, p) \Rightarrow c_j < p$$

- *Strikt: $(H^{MC}, <)$ ist kaskadenfrei und*

$$\forall w_i[x], w_j[x] \in H^{MC} : (i \neq j \wedge w_j[x] < w_i[x]) \Rightarrow a_j < w_i[x] \vee c_j < w_i[x]$$

[22] Die hier verwendeten Funktionen d und T entstammen Definition 3.

Für die eben definierten Fehlererholungseigenschaften gilt offenbar die übliche und für rw-Historien ohnehin bekannte Inklusionaussage „strikt $\subset$ kaskadenfrei $\subset$ rücksetzbar“ (siehe etwa Abschnitt 2.4 in [4]). Auf den (trivialen) Beweis sei hier verzichtet.

Beispiel 8. *Es werden nun die Methoden-Cache-Historien $(H_1, <), \ldots, (H_4, <)$ mit Methodenoperationen vorgestellt, so dass gilt: $(H_1, <)$ ist nicht rücksetzbar, $(H_2, <)$ ist rücksetzbar, aber nicht kaskadenfrei, $(H_3, <)$ ist kaskadenfrei, aber nicht strikt, $(H_4, <)$ ist strikt.*

$$(H_1, <) = w_1[x]w_1[y]w_2[y]r_2^1[x]m_3^{2,1}c_3c_1c_2,$$

$$(H_2, <) = w_1[x]w_1[y]w_2[y]r_2^1[x]m_3^{2,1}c_1c_3c_2,$$

$$(H_3, <) = w_1[x]w_1[y]w_2[y]c_1r_2^1[x]m_3^{2,1}c_3c_2,$$

$$(H_4, <) = w_1[x]w_1[y]c_1w_2[y]r_2^1[x]m_3^{2,1}c_3c_2,$$

Man betrachte außerdem die folgende Methoden-Cache-Historie $(H_5, <)$:

$$(H_5, <) = r_1^1[x]r_2[x]w_3[x]m_2^{1,1}c_3c_2c_1.$$

$(H_5, <)$ ist strikt, obwohl zwischen $w_3[x]$ und $m_2^{1,1}$ kein c_3 liegt! Hier ist ausschließlich *die Liest-Von-Relation aus der Kaskadenfreiheit ausschlaggebend und in dieser kommt $reads(T_2, x, T_3, m_2^{1,1})$ nicht vor, weil $r_1^1[x]$* nicht *von T_3 liest.*

Die in Abschnitt 3.2 angesprochene Aufgabenteilung von m-Planer und rw-Planer drückt sich aus formaler Sicht in den Anteilen der Methoden-Cache-Historien aus, welche der jeweilige Planer produziert. Um ein angestrebtes Kooperationsveralten der Planer formal abzusichern, sollte man deshalb Bedingungen an den Historienanteil jedes Planers stellen können. Für die Arbeit des rw-Planers ist dies leicht möglich, indem man alle m-Operationen aus einer Methoden-Cache-Historie „herausfiltert“. Genau diese Aufgabe leistet die im Folgenden definierte rw-Projektion.

Definition 16. *rw-Projektion: Es sei $(H^{MC}, <)$ eine (wohl geordnete) Methoden-Cache-Historie. Die Abbildung $RW : (H^{MC}, <) \mapsto (\cup_{p \in H^{MC}} rw(p), <_{rw})$ heißt* rw-Projektion*, wobei gilt:*

$$rw(p) := \begin{cases} \emptyset & \textit{falls } a(p) = m \\ \{p\} & \textit{sonst} \end{cases},^{23}$$

$$\forall p, q \in \cup_{h \in H^{MC}} rw(h) : p <_{rw} q :\Leftrightarrow \exists u, v \in H^{MC} : (u < v \wedge p \in rw(u) \wedge q \in rw(v)).$$

Beispiel 9. *Die rw-Projektion zu $(H_1, <) = r_1^1[y]r_1^1[x]c_1w_2[x]c_2m_3^{1,1}w_3[x]c_3$ aus Beispiel 1 ist $RW((H_1, <)) = r_1^1[y]r_1^1[x]c_1w_2[x]c_2w_3[x]c_3$.*

Der folgende Satz garantiert Kaskadenfreiheit für Methoden-Cache-Historien und trennt dazu Anforderungen für den m- und den rw-Planer. Formal geschieht dies einerseits dadurch, dass der Satz für die rw-Projektion einer Methoden-Cache-Historie,

[23] Die hier verwendete Funktion a entstammt Definition 3.

also letztlich für den Historienanteil des rw-Planers, Kaskadenfreiheit fordert. Andererseits zielt eine weitere Voraussetzung auf Methoden-Operationen aus der kompletten Methoden-Cache-Historie ab. Zusammen führen die beiden Voraussetzungen zum gewünschten Resultat.

Diese Art der Aufspaltung von Bedingungen kehrt bei den Serialisierbarkeitsprotokollen aus Kapitel 7 mehrmals wieder. Zur Implementierung eines Protokolls müssen in erster Linie die Voraussetzungen zu Methoden-Operationen beachtet werden. Die Forderungen an die rw-Projektion einer Historie soll ja bereits ein konventioneller, transaktionaler Ressourcenverwalter sicherstellen.

Satz 3. *Es sei $(H^{MC}, <)$ eine (wohl geordnete) Methoden-Cache-Historie mit den Transaktionen $\{(T_1, <_1), \ldots, (T_n, <_n)\}$ und $(H, <') = RW((H^{MC}, <))$ deren Bild unter der rw-Projektion. Dann gilt: $(H^{MC}, <)$ ist genau dann* kaskadenfrei, *wenn $(H, <')$ kaskadenfrei ist und wenn gilt:*

$$\forall i \in \{1, \ldots, n\} : \forall x \in D : reads(T_i, x, T_i, r_i^l[x]) \Rightarrow \forall m_j^{i,l} \in H^{MC} : c_i < m_j^{i,l}.$$

Beweis. „⇒": $(H^{MC}, <)$ erfülle die entsprechende Fehlererholungseigenschaft. Durch das Entfernen von Methodenoperationen mittels RW entsteht eine Teilmenge der ursprünglichen Liest-Von-Relation. Daher bleibt die Eigenschaften Kaskadenfreiheit für $(H, <')$ erhalten.

„⇐": Es gelte $(H, <') = RW((H^{MC}, <))$ mit $(H, <')$ kaskadenfrei. Bezüglich $(H^{MC}, <)$ müssen lediglich die hinzugekommenen Methodenoperationen geprüft werden. Es sei $m_i^{k,l}$ eine derartige Operation und $w_j[x]$ die Operation, von der $m_i^{k,l}$ liest (also $reads(T_i, x, T_j, m_i^{k,l})$). Dann existiert aber gemäß Definition 14 ein $r_k^l[x]$ mit $reads(T_k, x, T_j, r_k^l[x])$. Wegen Definition 2 gilt insbesondere $r_k^l[x] < m_i^{k,l}$. Falls $k \neq j$ folgt $c_j < r_k^l[x]$, weil $(H, <')$ kaskadenfrei ist und weiter $c_j < m_i^{k,l}$. Andernsfalls hat man mit $k = j$ auch $reads(T_j, x, T_j, r_j^l[x])$ und es folgt $c_j < m_i^{k,l}$ auf Grund der Voraussetzung des Satzes. Mithin ist $(H^{MC}, <)$ kaskadenfrei. □

Die Bedingung im vorausgehenden Satz ist notwendig, wie man am folgenden Beispiel erkennt.

Beispiel 10. *Die Methoden-Cache-Historie $(H_6, <) = w_1[x]r_1^1[x]m_2^{1,1}c_1c_2$ ist offenbar nicht kaskadenfrei wegen $reads(T_2, x, T_1, m_2^{1,1})$ und $m_2^{1,1} < c_1$. Allerdings ist $RW((H_6, <)) = w_1[x]r_1^1[x]c_1c_2$ kaskadenfrei. Hier ist die Bedingung zur Liest-Von-Relation aus Satz 3 verletzt, denn es gilt $reads(T_1, x, T_1, r_1^1[x])$.*

Wenn $(H, <') = RW((H^{MC}, <))$ rücksetzbar ist, dann gilt dies nicht notwendig auch für $(H^{MC}, <)$, wie das folgende Beispiel zeigt.

Beispiel 11. *Es sei $(H_7, <) = w_1[x]r_2^1[x]m_3^{2,1}c_3c_1c_2$ eine Methoden-Cache-Historie. $(H_7, <)$ ist offenbar nicht rücksetzbar, denn man hat $reads(T_3, x, T_1, m_3^{2,1})$ und $c_3 < c_1$. Trotzdem ist $RW((H_7, <)) = w_1[x]r_2^1[x]c_3c_1c_2$ rücksetzbar, denn $reads(T_3, x, T_1, m_3^{2,1})$ gilt dort natürlich nicht.*

Lemma 1. *Es sei $(H^{MC}, <)$ eine (wohl geordnete) Methoden-Cache-Historie und $(H, <') = RW((H^{MC}, <))$ deren Bild unter der rw-Projektion. Dann gilt: $(H^{MC}, <)$ ist genau dann* strikt, *wenn $(H^{MC}, <)$ kaskadenfrei und $(H, <')$ strikt ist.*

Beweis. „⇒": Sei $(H^{MC}, <)$ strikt. Durch RW werden lediglich Operationen entfernt. Da dabei die a_i- und c_i-Operationen (und die Ordnung) erhalten bleiben, ist $(H, <')$ notwendig strikt.

„⇐": Sei $(H^{MC}, <)$ kaskadenfrei und $(H, <')$ strikt. Bezüglich $(H^{MC}, <)$ müssen lediglich die hinzugekommenen Methodenoperationen geprüft werden. Diese wirken sich aber auf die Bedingung zu w-Operationen aus der Definition von Striktheit nicht aus. Somit folgt trivialerweise die Behauptung. □

6.2 Implementierung

Mit Hilfe des Entwurfs eines transaktionalen Methoden-Caches aus Kapitel 5 und mittels Satz 3 sowie Lemma 1 wird nun ein einfaches Protokoll vorgestellt, welches Striktheit in Bezug auf Methoden-Cache-Transaktionen garantiert. Die folgenden Paragraphen beschreiben die Idee des Protokolls zunächst abstrakt und liefern dann Implementierungsdetails auf Basis des UML-Diagramms aus Abbildung 9.

6.2.1 Verfahren

Als entscheidende Annahme für das Protokoll wird vorausgesetzt, dass ein rw-Planer bereits strikte rw-Historien produziert. Dies ist bei konventionellen Datenbankverwaltungssystemen praktisch immer der Fall. Lemma 1 folgend muss man dann lediglich die Kaskadenfreiheit von Methoden-Cache-Historien zusichern. Dazu genügt es, die entsprechende Bedingung aus Satz 3 im Protokoll umzusetzen.

Solange eine Transaktion T_i nur liest, also keine w-Operationen erzeugt, kann $reads(T_i, x, T_i, r_i^l[x])$ aus der Bedingung von Satz 3 natürlich nicht wahr werden. Das Protokoll kann dann Methodenresultate, die innerhalb von T_i berechnet werden im Cache-Speicher (eines Klienten) einlagern und sofort für andere Methoden-Cache-Transaktionen zum Zugriff freigeben.

Sobald T_i allerdings ein Datenelement x schreibt, ist Vorsicht geboten: Ein Methodenresultat mr, das danach von T_i berechnet und gecacht wird, könnte x mittels einer Operation $r_i^l[x]$ lesen und die Beziehung $reads(T_i, x, T_i, r_i^l[x])$ herstellen. Nach Satz 3 muss aber für diesen Fall garantiert werden, dass das gecachte Methodenresultat mr erst von anderen Transaktionen verwendet wird, nachdem c_i erfolgt ist.

Um diese Bedingung auf einfache Weise sicherzustellen, genügt es, *alle* von T_i berechneten und gecachten Methodenresultate, die auf *die erste* Schreiboperation in T_i folgen, zunächst für den Zugriff durch andere Transaktionen zu sperren. Nach dem Abschluss von T_i darf der Methoden-Cache die Methodenresultate freigeben.

Das Verfahren betrachtet die von T_i gelesenen und geschriebenen Datenelemente eigentlich gar nicht. Es berücksichtigt nur, ob T_i überhaupt bereits eine Schreiboperation ausgeführt hat. Auf Basis der Architektur von Kapitel 5 kann das Protokoll daher

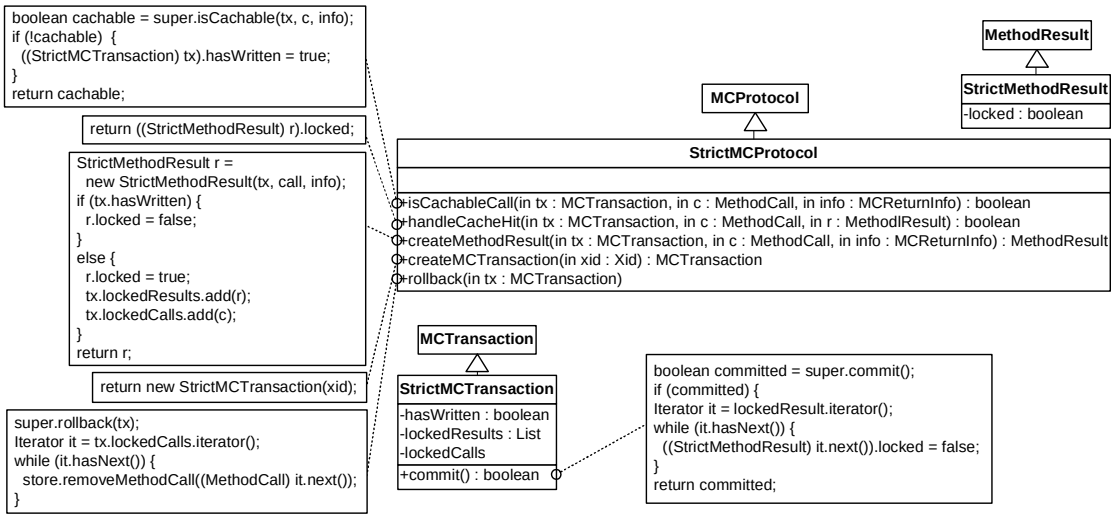

Abbildung 11: Erweiterung der Methoden-Cache-Klassen aus Abbildung 9 zur Garantie von Striktheit

komplett auf der Klientenseite implementiert werden. Der m-Planer muss dazu nicht einbezogen bzw. verändert werden.

6.2.2 Umsetzung

Abbildung 11 zeigt eine Erweiterung der Methoden-Cache-Klassen aus Abbildung 9, die Striktheit durch das eben skizzierte Protokoll zusichert. Hierzu wurden Unterklassen zu den bereits bekannten Klassen `MCProtocol`, `MCTransaction` und `MethodResult` gebildet und entsprechende Methoden überschrieben.

Das Attribut `locked` in der Klasse `StrictMethodResult` verhindert, dass ein vom Protokoll gesperrtes Methodenresultat von anderen Transaktionen verwendet werden kann. Dementsprechend wertet die Methode `StrictMCProtocol.handleCacheHit()` gerade dieses Attribut im Falle eines Cache-Treffers aus.

In dem Attribut `hasWritten` aus `StrictMCTransaction` vermerkt die Methode `StrictMCProtocol.isCachableCall()`, ob eine Transaktion eine Schreiboperation getätigt hat. Dazu genügt es der Methode, die `isCachableCall()`-Implementierung aus der Oberklasse zu konsultieren. Falls eine Transaktion geschrieben hat, sind Methodenresultate, die danach innerhalb der Transaktion gecacht werden sollen, zu sperren. Zur Sperrfreigabe beim Abschluss der Transaktion werden solche Methodenresultate in der Liste `StrictMCTransaction.lockedResults` gesichert. Bricht die Transaktion ab, werden die gesperrten Resultate aber aus dem Cache-Speicher entfernt. Um die entsprechenden Tabelleneinträge aus dem Cache-Speicher (Klasse `MethodStore`) löschen zu können, hinterlegt das Protokoll auch die zugeordneten Methodenaufrufe in der Liste `StrictMCTransaction.lockedCalls`.

In `StrictMCProtocol` erzeugt `createMethodResult()` ein sperrbares Methodenresultat und initialisiert die Sperre geeignet. Bei gesetzter Sperre hinterlegt das Protokoll das Methodenresultat und den Methodenaufruf in den bereits angesprochenen Listen von `StrictMCTransaction`. `StrictMCProtocol.rollback()` sorgt mittels der erwähnten Liste `StrictMCTransaction.lockedCalls` für die Bereinigung des Cache-Speichers. `StrictMCTransaction.commit()` gibt hingegen alle Sperren frei, welche eine Transaktion auf Methodenresultaten hält.

7 Serialisierbarkeitsprotokolle

In diesem Kapitel werden ausschließlich *optimistische* Serialisierbarkeitsprotokolle für m-Planer entwickelt. Die Idee dabei ist, dass während einer Transaktionsausführung beim Klienten wann immer möglich Methodenresultate *ohne Prüfung beim Server* aus dem Methoden-Cache gelesen werden dürfen. Ob ein solcher Zugriff eine gegebene Protokollbedingung erfüllt, stellt sich fühestens beim nächsten entfernten Dienstmethodenaufruf der Transaktion und spätestens beim Beenden der Transaktion heraus.

Der Ansatz ist optimistisch im Sinne von [19], weil die Korrektheit (genauer die Serialisierbarkeit) eines eine Zugriffs auf den Methoden-Cache erst im Nachhinein getestet wird, und zwar dann, wenn die entsprechende Transaktionen ohnehin einen entfernten Aufruf beim Applikationsserver durchführen muss. Die Alternative – ein pesssimistischer Ansatz – wäre wenig sinnvoll, da er bei einem Cache-Treffer *sofort* einen teuren Serverzugriff erzwingen würde. Letzteren möchte man aber gerade durch das transaktionale Methoden-Caching zu Gunsten von Leistungsverbesserungen vermeiden.

Neben den nachfolgend diskutierten Protokollen kann man aus Definition 13 auch sofort ein so genanntes *Serialisierbarkeitstestprotokoll* (SGT-Protokoll) ableiten. Entsprechende Verfahren sind aus der Literatur, etwa aus [4] Abschnitt 4.3. und [15], hinreichend gut bekannt und könnten für den Spezialfall von Methoden-Cache-Serialisierbarkeitsgraphen auf einfache Weise angepasst werden. Im Wesentlichen stellen SGT-Protokolle Zyklen in einem verallgemeinerten Serialisierbarkeitsgraphen durch geeignete Datenstrukturen explizit fest und brechen dann entsprechende Transaktionen ab.

Leider sind SGT-Protokolle mit den Anforderungen dieses Berichts kaum verträglich, da sie eine praxistaugliche Zerlegung des integrierten Planers in rw- und m-Planer *nicht* unterstützen. Dies liegt daran, dass zur Zyklenbestimmung der Serialisierbarkeitsgraph letztlich als Ganzes betrachtet werden muss. rw- und m-Planer müssten daher ihre Datenstrukturen zur Graphendarstellung austauschen. Die rw-Planer von kommerziellen Datenbankverwaltungssystemen sind aber nicht auf eine solche Interaktion ausgelegt. Ohnehin kommen bei konventionellen Datenbankverwaltungssystemen SGT-Protokolle nicht zum Einsatz.

7.1 Sperrprotokoll

7.1.1 Idee

Die Idee für das nun vorgestellte Protokoll ist angelehnt an ein klassisches optimistisches Zwei-Phasen-Sperrprotokoll, wie es etwa in [4] Abschnitt 4.4 oder [19] behandelt wird.

Als wichtigste Eigenschaft bricht das Protokoll eine Transaktion T_i ab, die ein gecachtes Methodenresultat liest, welches bereits von einer anderen Transaktionen T_j invalidiert wurde. Eine typische Historie für diesen Fall zeigt das nächste Beispiel.

Beispiel 12. *Die folgende, serielle Methoden-Cache-Historie ist nicht serialisierbar, obwohl ihre rw-Projektion serialisierbar ist.*

$$(H_1, <) = r_1^1[x]c_1w_2[x]c_2r_3[x]m_3^{1,1}c_3.$$

Der zugehörige Serialisierbarkeitsgraph $MCSG$ ist zyklisch und umfasst die Kanten $T_1 \to T_2$ (wegen $r_1^1[x] < w_2[x]$), $T_2 \to T_3$ (wegen $w_2[x] < r_3[x]$) und $T_3 \to T_2$ (wegen $r_1^1[x] < w_2[x] < m_3^{1,1}$).

Formal schließt der weiter unten definierte Begriff „aktuell“ gerade Graphenkanten wie $T_3 \to T_2$ aus Beispiel 12 aus. Für die Serialisierbarkeit der vom einem entsprechenden m-Planer und rw-Planer erzeugten Methoden-Cache-Historien reicht dies aber leider noch nicht aus.

Deshalb fordert man für produzierte Methoden-Cache-Historien zusätzlich noch „quasi-rigoros“. Vereinfachend besagt diese Eigenschaft, dass für zwei konfliktbehaftete Operationen $p_i < q_j$ mit $T_i \neq T_j$ alle Operationen von T_i mit Ausnahme von c_i vor der Operation q_j stattfinden müssen. Der Begriff ist von der klassischen Fehlererholungseigenschaft „rigoros“ abgeleitet, schwächt diese aber entscheidend ab: Gegenüber quasi-rigorosen Historien muss bei rigorosen Historien sogar die Operation c_i noch vor q_j stattfinden. „Quasi-rigoros“ wird etwa von dem in der Praxis wichtigen strikten Zwei-Phasen-Sperrprotokoll erfüllt – „rigoros“ hingegen nicht. Wie sich leicht zeigen lässt, gilt: Alle quasi-rigorosen rw-Historien sind serialisierbar.

Glücklicherweise garantieren die Eigenschaften „aktuell“ und „quasi-rigoros“ bereits die Serialisierbarkeit von Methoden-Cache-Historien, die ein integrierte Planer generiert. Dabei kann „aktuell“ bereits der m-Planer allein zusichern. Wie steht es aber mit „quasi-rigoros“? Es stellt sich heraus, dass eine Methoden-Cache-Historie nicht automatisch quasi-rigoros ist, wenn dies für ihre rw-Projektion zutrifft. Also muss sich an quasi-rigorosen Methoden-Cache-Historien auch der m-Planer beteiligen.

Hierfür isoliert der unten definierte Begriff „m-sperrend“ weitestgehend jene Aspekte der Eigenschaft „quasi-rigoros“, die ein m-Planer zusichern muss, damit der integrierte Planer insgesamt doch quasi-rigorose Methoden-Cache-Historien erzeugt. Die Notwendigkeit und die Idee von „m-sperrend“ verdeutlicht das nächste Beispiel.

Beispiel 13. *Die folgende, aktuelle Methoden-Cache-Historie ist nicht serialisierbar, obwohl ihre rw-Projektion quasi-rigoros und serialisierbar ist.*

$$(H_2, <) = r_1[x]w_2[x]c_2r_3^1[x]c_3m_1^{3,1}c_1.$$

Der zugehörige Serialisierbarkeitsgraph $MCSG$ ist zyklisch und enthält die Kanten $T_1 \rightarrow T_2$ (wegen $r_1[x] < w_2[x]$), $T_2 \rightarrow T_3$ (wegen $w_2[x] < r_3^1[x]$) und $T_2 \rightarrow T_1$ (wegen $w_2[x] < r_3^1[x] < m_1^{3,1}$).

Das Problem an der Methoden-Cache-Historie aus Beispiel 13 ist, dass sich $m_1^{3,1}$ nach $w_2[x]$ ereignet, obwohl es vorher mit $r_1[x] < w_2[x]$ schon zum Konflikt zwischen T_1 und T_2 kam. „m-sperrend“ schließt gerade solche Ereignisse aus.

Wie sich zeigt, kann man einen m-Planer, der „kaskadenfrei“, „aktuell“ und „m-sperrend“ sicherstellt, mit einem rw-Planer integrieren, der das strikte Zwei-Phasen-Sperrprotokoll anwendet, und erreicht somit Serialisierbarkeit.

Die Konstruktion eines entsprechenden m-Planers ist überraschend einfach und gestaltet sich sehr ähnlich, wie ein klassischer Planer für das optimistische Zwei-Phasen-Sperr-Protokoll. Abschnitt 7.1.3 liefert dazu einen Implementierungsvorschlag auf Basis des Entwurfs aus Abschnitt 5.4.

7.1.2 Formalismus

Das Protokoll dieses Abschnitts basiert auf einer Integration von m-Planer und rw-Planer mit Hilfe der Eigenschaft „quasi-rigoros“. Dazu müssen beide Planer Aspekte dieser Eigenschaft zusichern. „quasi-rigoros“ schwächt die Fehlererholungseigenschaft „rigoros“ ab, welche in der Transaktionsverwaltungsliteratur etabliert ist ([6, 36]).

Definition 17. *Quasi-rigorose Methoden-Cache-Historie: Eine Methoden-Cache-Historie $(H^{MC}, <)$ mit den Transaktionen $\{(T_1, <_1), \ldots, (T_n, <_n)\}$ ist genau dann* quasi-rigoros, *wenn gilt:*

$$\forall p, q \in H^{MC} : \forall i, j \in \{1, \ldots, n\} : (i \neq j \wedge p \in T_i \wedge q \in T_j \wedge p < q \wedge p \nparallel q) \Rightarrow$$

$$(a_i \in H^{MC} \vee a_j \in H^{MC} \vee \forall u \in T_i \setminus \{c_i\} : u < q).$$

Zum Vergleich mit „quasi-rigoros“, wird nun der etablierte Begriff „rigoros“ für rw-Historien definiert.

Definition 18. *Rigorose rw-Historie: Eine rw-Historie $(H, <)$ ist* rigoros *genau dann, wenn gilt: $(H, <)$ ist strikt und*

$$\forall w_i[x], r_j[x] \in H : (i \neq j \wedge r_j[x] < w_i[x]) \Rightarrow a_j < w_i[x] \vee c_j < w_i[x].$$

Gemäß dem folgenden Lemma ist „quasi-rigoros“ in der Tat eine Abschwächung von „rigoros“.

Lemma 2. *Jede rigorose rw-Historie ist quasi-rigoros.*

Beweis. Es sei $(H, <)$ eine rigorose rw-Historie und p, q Operationen aus H mit $p < q$, $p \nparallel q$ sowie $p \in T_i$, $q \in T_j$, $i \neq j$. Mithin gilt, weil $(H, <)$ rigoros ist: $p < c_i < q$ oder $p < a_i < q$. Es folgt sofort $\forall u \in T_i \setminus \{c_i\} : u < q$ und somit die Behauptung. □

Das nächste Beispiel belegt, dass die Fehlererholungseigenschaften aus Kapitel 6 nicht aus der Eigenschaft „quasi-rigoros“ folgen.

Beispiel 14. *Die folgende Methoden-Cache-Historie ist quasi-rigoros, aber nicht rücksetzbar:* $(H_3, <) = w_1[x]r_2[x]c_2c_1$. *Die Methoden-Cache-Historie* $(H_4, <) = r_1[x]w_2[x]r_1[y]c_2c_1$ *ist strikt, aber nicht quasi-rigoros.*

Wie man sehen wird, ist der Begriff „quasi-rigoros“ einerseits stark genug, um damit ein Serialisiebarkeitsprotokoll zu entwickeln. Andererseits ist er aber so schwach, dass er von vielen konventionellen rw-Planern unterstützt wird. Dies gilt insbesondere in Bezug auf das strikte Zwei-Phasen-Sperrprotokoll:

Satz 4. *Jede rw-Historie, die durch das strikte Zwei-Phasen-Sperrprotokoll erzeugt wird, ist quasi-rigoros.*

Beweis. Die Beweisargumentation unterstellt die Kenntnis des strikten Zwei-Phasen-Sperrprotokolls, welches hier der Einfachheit halber weder definiert noch formalisiert ist. Für eine genaue Beschreibung des Protokolls sei etwa auf Abschnitt 3.5 aus [4] verwiesen.

Es seien $p < q, p \nparallel q$ zwei Operationen mit $p \in T_i, q \in T_j$, die ein rw-Planer bearbeitet, der das strikte Zwei-Phasen-Sperrprotokoll befolgt. Vor der Bearbeitung von p wird eine Sperre l gesetzt, so dass q nicht bearbeitet kann, bis die Sperre freigegeben wird. Gemäß dem Zwei-Phasen-Sperrprotokoll wird l aber frühestens freigegeben, sobald sicher ist, dass der rw-Planer zu T_i keine Schreib- oder Leseoperationen mehr erhält. Das ist erst der Fall, wenn der Klient zu T_i eine Abschlussaufforderung (`commit()`-Aufruf) an den Planer sendet. Zu diesem Zeitpunkt wurden bereits alle Schreib- und Leseoperationen bezüglich T_i abgearbeitet, so dass die Bedingung $\forall u \in T_i \setminus \{c_i\} : u < q$ tatsächlich eingehalten wird. Falls der Planer T_i abbricht (Abort), muss die Bedingung ohnehin nicht eingehalten werden. □

Man beachte, dass das strikte Zwei-Phasen-Protokoll zwar quasi-rigoros, aber nicht notwendig *rigoros* ist. Dies liegt daran, dass der rw-Planer Lesesperren von T_i bereits freigeben kann, bevor die Abschlussaufforderung (`commit()`-Aufruf) von T_i durch den Planer bearbeitet wurde. Die entsprechende Sperrfreigabe kann also zwischen dem Eingang der Abschlussaufforderung für T_i und deren beendeter Bearbeitung (formalisiert durch c_i bzw. a_i) erfolgen.

Definition 19. *Aktuelle Methoden-Cache-Historie: Eine (wohl geordnete) Methoden-Cache-Historie* $(H^{MC}, <)$ *ist genau dann* aktuell, *wenn gilt:*

$$\forall m_i^{k,l} \in H^{MC} : (\exists r_k^l[x], w_j[x] \in H^{MC} : r_k^l[x] < w_j[x] < m_i^{k,l}) \Rightarrow a_i \in H^{MC} \vee a_j \in H^{MC}.$$

Der Begriff „aktuell“ verhindert, dass eine Transaktion Methodenresultate liest, die bereits durch eine Schreiboperation invalidiert wurden. Wenn eine Transaktion dies doch tut, muss sie abgebrochen werden.

Wie bereits in Abschnitt 7.1.1 erwähnt, sorgen „aktuell“ und „quasi-rigoros“ für die Serialisierbarkeit von Methoden-Cache-Historien. Dies belegt der folgende Satz.

Satz 5. *Jede aktuelle, quasi-rigorose (wohl geordnete) Methoden-Cache-Histore ist MC-serialisierbar.*

Beweis. Es sei $(H^{MC}, <)$ eine aktuelle, quasi-rigorose (wohl geordnete) Methoden-Cache-Historie und $MCSG$ der zugehörige Serialisierbarkeitsgraph. Angenommen $(H^{MC}, <)$ ist nicht serialisierbar und somit $MCSG$ zyklisch. Weiter sei dann $T_{k_1} \rightarrow T_{k_2} \rightarrow \ldots \rightarrow T_{k_l} \rightarrow T_{k_1}$ ein entsprechender Zyklus in $MCSG$.

Zunächst wird gezeigt, dass für jede Kante $T_i \rightarrow T_j \in MCSG$ gilt: Es existieren zwei Operationen $p \in T_i$, $q \in T_j$ mit $p < q \wedge p \nparallel q$ und man hat $i \neq j$.

Falls $T_i \rightarrow T_j$ aus der ersten Disjunktionsklausel aus Definition 13 stammt, trifft dies trivialerweise zu. Falls die dritte Disjunktionsklausel aus Definition 13 erfüllt ist, hat man $w_i[x] < r_k^l[x] < m_j^{k,l}$ mit $i \neq j$ und die Behauptung gilt ebenfalls. Bezüglich der zweiten Disjunktionsklausel liegen die Operationen $r_k^l[x] < w_j[x]$ und $m_i^{k,l}$ vor. Im Fall $r_k^l[x] < m_i^{k,l} < w_j[x]$ muss gemäß Definition 13 auch $i \neq j$ gelten, und es ergibt sich wiederum die Behauptung. Andererseits widerspricht $r_k^l[x] < w_j[x] < m_i^{k,l}$ der Voraussetzung, dass $(H^{MC}, <)$ aktuell ist, denn T_i und T_j sind abgeschlossen. Also gilt die obige Aussage zu den Operationen p und q.

Weil $T_i \rightarrow T_j \in MCSG$ gilt, sind T_i und T_j abgeschlossen und da $(H^{MC}, <)$ quasi-rigoros ist, folgt mit der eben gezeigten Aussage: $\forall u \in T_i \setminus \{c_i\} : u < q$.

Es seien nun p_{k_s}, $q_{k_{s \bmod l+1}}$ die jeweils zueinander in Konflikt stehenden Operationen aus T_{k_s}, $T_{k_{s \bmod l+1}}$, die also $T_{k_s} \rightarrow T_{k_{s \bmod l+1}} \in MCSG$ erzeugen. Wendet man die Aussage des vorigen Abschnitts an (also $\forall u \in T_{k_s} \setminus \{c_{k_s}\} : u < q_{k_{s \bmod l+1}}$), erhält man dann per Induktion die Ordnung $p_{k_1} < q_{k_2} < q_{k_3} < \ldots < q_{k_l} < q_{k_1} < q_{k_2}$ im Widerspruch zur Azyklität der Ordnungsrelation $<$. □

Lemma 3. *Jede quasi-rigorose rw-Historie ist serialisierbar.*

Beweis. Die Aussage folgt direkt aus Satz 5, denn rw-Historien sind trivialerweise aktuell. □

Beispiel 12 hat bereits dargelegt, dass „aktuell“ für die Serialisierbarkeit von Methoden-Cache-Historien nicht ausreicht (nicht einmal, wenn eine entsprechende rw-Projektion quasi-rigoros ist). Umgekehrt führt die Eigenschaft „quasi-rigoros“ alleine aber auch nicht zum Ziel. Dies erkennt man am folgenden Beispiel.

Beispiel 15. *Nicht jede quasi-rigorose Methoden-Cache-Historie ist MC-serialisierbar oder aktuell:* $(H_5, <) = r_1^1[x]c_1w_2[x]c_2r_3[x]m_3^{1,1}c_3$.

Der Begriff „m-sperrend“ stellt wichtige Eigenschaften heraus, für die der m-Planer zuständig ist, um zu quasi-rigorosen Methoden-Cache-Historien zu gelangen. „m-sperrend“ folgt sogar aus „quasi-rigoros“, wie weiter unten klar wird.

Definition 20. *m-sperrende Methoden-Cache-Historie: Eine Methoden-Cache-Historie $(H^{MC}, <)$ mit den Transaktionen $\{(T_1, <_1), \ldots, (T_n, <_n)\}$ ist* m-sperrend *genau dann, wenn gilt:*

$$(\forall p, q \in H^{MC} : \forall i, j \in \{1, \ldots, n\} : (i \neq j \wedge p \in T_i \wedge q \in T_j \wedge p < q \wedge p \nparallel q) \Rightarrow$$

$$(a_i \in H^{MC} \vee a_j \in H^{MC} \vee \forall m_i^{k,l} \in H^{MC} : m_i^{k,l} < q)) \wedge$$
$$(\forall m_i^{k,l}, w_j[x] \in H^{MC} : (i \neq j \wedge m_i^{k,l} < w_j[x] \wedge m_i^{k,l} \nparallel w_j[x]) \Rightarrow$$
$$(a_i \in H^{MC} \vee a_j \in H^{MC} \vee \forall u \in T_i \setminus \{c_i\} : u < w_j[x])).$$

Die folgenden beiden Beispiele verdeutlichen, dass „m-sperrend“ und „aktuell“ voneinander unabhängige Begriffe sind.

Beispiel 16. *Nicht jede aktuelle Methoden-Cache-Historie ist m-sperrend:* $(H_6, <) = w_1[x] r_2^1[x] m_1^{2,1} c_2 c_1$. *Nicht jede m-sperrende Methoden-Cache-Historie ist aktuell:* $(H_7, <) = r_1^1[x] w_1[x] m_2^{1,1} c_2 c_1$.

Lemma 4. *Jede quasi-rigorose (wohl geordnete) Methoden-Cache-Historie ist m-sperrend.*

Beweis. Die Aussage ist bei Vergleich der entsprechenden Definitionen offensichtlich. □

Der folgenden Satz vereint bereits alle nötigen Voraussetzungen für die Serialisierbarkeit des angestrebten Sperrprotokolls. Vom rw-Planer wird dazu lediglich erwartet, dass er quasi-rigorose rw-Historien erzeugt. Der m-Planer muss hingegen für die Eigenschaften „kaskadenfrei“, „aktuell“ und „m-sperrend“ sorgen. Wie man Kaskadenfreiheit umsetzt, wurde bereits in Kapitel 6 behandelt und braucht für die Implementierung des Sperrprotokolls im m-Planer nicht weiter berücksichtigt zu werden.

Satz 6. *Eine kaskadenfreie, aktuelle, m-sperrende (wohl geordnete) Methoden-Cache-Historie $(H^{MC}, <)$ ist genau dann quasi-rigoros, wenn die zugehörige rw-Projektion $RW((H^{MC}, <))$ quasi-rigoros ist.*

Beweis. Es sei $(H^{MC}, <)$ eine m-sperrende (wohl geordnete) Methoden-Cache-Historie und $(H, <') = RW((H^{MC}, <))$ die zugehörige rw-Projektion.

„$\Rightarrow$“: Angenommen $(H^{MC}, <)$ ist quasi-rigoros. Da RW lediglich Operationen entfernt, können in $(H, <')$ keine Konflikte bestehen, die es nicht auch in $(H^{MC}, <)$ gibt. $(H, <')$ ist somit (trivialerweise) quasi-rigoros.

„$\Leftarrow$“: Angenommen $(H, <')$ ist quasi-rigoros. Für $p, q \in H^{MC}$ gelte die Voraussetzung $p < q$, $p \nparallel q$ und $p \in T_i$, $q \in T_j$ gemäß Definition 17, wobei $i \neq j$.

Sei zuerst q *keine* Methodenoperation. Weiter sei $c_i, c_j \in H^{MC}$ und u eine beliebige Operation aus T_i. Falls u und p keine Methodenoperationen sind, gilt auch bezüglich $(H, <')$ die Voraussetzung $p <' q$, $p \nparallel q$ sowie $c_i, c_j \in H$. Damit folgt $u <' q$ und weiter $u < q$. Falls p eine Methodenoperation ist, muss q eine Schreiboperation sein. Mit der zweiten Konjunktionsklausel von Definition 20 gilt dann $u < q$. Falls u eine Methodenoperation ist, aber p nicht, folgt $u < q$ wegen der ersten Konjunktionsklausel von Definition 20.

Sei nun q eine Methodenoperation. Damit muss es sich bei p um eine Schreiboperation handeln. Es liegt also die Situation $p = w_i[x] < q = m_j^{k,l}$ vor, und somit existiert ein $r_k^l[x] < m_j^{k,l}$. Falls $r_k^l[x] < w_i[x] < m_j^{k,l}$ gilt, folgt $a_i \in H^{MC} \vee a_j \in H^{MC}$ denn sonst

wäre $(H^{MC}, <)$ nicht aktuell. Hat man andererseits $w_i[x] < r_k^l[x]$ und $c_i, c_j \in H^{MC}$ (bzw. $c_i, c_j \in H$), so kann man $i \neq k$ und $i = k$ unterscheiden.

Falls $i \neq k$ gilt, stehen $w_i[x]$ und $r_k^l[x]$ zueinander in Konflikt, und man erhält $u <' r_k^l[x]$ für alle Operationen u aus T_i, die keine Methodenoperationen sind, da $(H, <')$ quasi-rigoros ist. Also gilt sogar $u < r_k^l[x] < m_j^{k,l}$ für entsprechende Operationen. Falls u eine Methodenoperation ist, folgt aber auch $u < m_j^{k,l}$ wegen der ersten Konjunktionsklausel von Definition 20.

Zum Fall $i = k$: Es gilt dann notwendig $reads(T_j, x, T_i, m_j^{k,l})$ und somit $c_i < m_j^{k,l}$, da $(H^{MC}, <)$ kaskadenfrei ist. Also folgt auch hier die Behauptung. □

Beispiel 17. *In Satz 6 ist sowohl die Voraussetzung „kaskadenfrei" als auch „aktuell" notwendig, wie man and den folgenden Methoden-Cache-Historien sieht.*

$$(H_8, <) = w_1[x]r_1^1[x]r_1^1[y]m_2^{1,1}w_1[y]c_2c_1$$

ist zwar aktuell und m-sperrend, aber nicht kaskadenfrei, nicht quasi-rigoros und auch nicht serialisierbar. Allerdings ist $RW((H_8, <))$ quasi-rigoros (und serialisierbar).

$(H_9, <) = r_1^1[x]w_1[x]m_2^{1,1}r_1[y]c_1c_2$ ist zwar kaskadenfrei und m-sperrend, aber nicht aktuell und nicht quasi-rigoros. $RW((H_9, <))$ ist hingegen quasi-rigoros.

Aus Satz 5 ergibt sich in Verbindung mit Satz 6 die entscheidende Aussage zur Serialisierbarkeit:

Korollar 3. *Eine kaskadenfreie, m-sperrende, aktuelle (wohl geordnete) Methoden-Cache-Historie $(H^{MC}, <)$ ist MC-serialisierbar, wenn $RW((H^{MC}, <))$ quasi-rigoros ist.*

Zusammen mit Satz 4 kann man das Korollar noch deutlicher für rw-Planer mit einem Zwei-Phasen-Sperrprotokoll formulieren.

Korollar 4. *Eine kaskadenfreie, m-sperrende, aktuelle (wohl geordnete) Methoden-Cache-Historie $(H^{MC}, <)$ ist MC-serialisierbar, wenn $RW((H^{MC}, <))$ eine Historie ist, die durch das strikte Zwei-Phasen-Sperrprotokoll produziert wurde.*

7.1.3 Implementierung

Die im Folgenden diskutierte Implementierung eines optimistischen Sperrprotokolls unterstellt, dass der rw-Planer eines Ressourcenverwalters ein quasi-rigoroses und kaskadenfreies Protokoll (wie etwa das strikte Zwei-Phasen-Sperrprotokoll) anwendet. Gemäß Korollar 3 genügt es dann, dass der m-Planer für die Implementierung die Eigenschaften „m-sperrend" und „aktuell" garantiert. Man bedenke, dass kaskadenfreie Methoden-Cache-Historien nach Abschnitt 6.2 bereits der Methoden-Cache beim Klienten sicherstellt.

Die nächsten Paragraphen skizzieren einen überraschend einfachen Algorithmus, der dem Verfahren eines optimistischen Zwei-Phasen-Sperrprotokolls ähnelt (siehe hierzu etwa [4] Abschnitt 4.4). Anschließend wird noch eine konkrete Implementierung auf Basis des UML-Entwurfes aus Abschnitt 5.4 vorgestellt.

Verfahren Der m-Planer geht wie folgt vor: Für Operationsfolgen der Form $r_k^l[x]r_k^l[y]\ldots$, die zu einen zu einem cachenden Methodenaufruf gehören, merkt er sich die Beziehung zwischen dem Tupel (k,l) zur Identifikation des Methodenaufrufs und den dabei gelesenen Datenelementen $x, y, \ldots$ in einer Relation V. V enthält also Einträge der Form $(\{x, y, \ldots\}, (k,l))$. Tritt eine Operation der Form $w_h[z]$ auf, dann entfernt der m-Planer alle Einträge aus V, die z im ersten Tupelement enthalten.

Auf der Klientenseite merkt sich der Methoden-Cache für eine Transaktion T_i alle Tupel (k,l), die von T_i gelesene, gecachte Methodenresultate identifizieren in einer (initial leeren) Liste L_i. Ein Tupel $(k,l) \in L_i$ bedeutet also, das T_i auf der Klientenseite die Operation $m_i^{k,l}$ ausgeführt hat. Sobald T_i abschließen soll, schickt der Methoden-Cache L_i zum m-Planer. Dort findet dann eine *Verifikation* statt.

Zur Verifikation prüft der m-Planer für jedes Tupel (k,l) aus L_i, ob es einen Eintrag $(\{\ldots\}, (k,l))$ in V gibt. Wenn dies wenigstens für ein Element aus der Liste L_i *nicht* der Fall ist, dann wird T_i abgebrochen. Der m-Planer sendet dann also ein a_i an die beteiligten Ressourcenverwalter. Ansonsten gibt er ein c_i an die Ressourcenverwalter weiter.

Korrektheit Das beschriebene Verfahren erzeugt aktuelle Methoden-Cache-Historien: Bei einer Operationsordnung der Form $r_k^l[x] < w_j[x] < m_i^{k,l}$ wird zunächst $(\{x, \ldots\}, (k,l))$ in V eingetragen, aber dann durch $w_j[x]$ wieder entfernt. Wenn T_i versucht abzuschließen, existiert zu $m_i^{k,l}$ bzw. (k,l) kein Eintrag in V und T_i bricht ab.

Als nächstes wird untersucht, ob die erste Konjunktionsklausel von Definition 20 erfüllt ist. Es seien also T_i, T_j Transaktionen mit $i \neq j \wedge p \in T_i \wedge q \in T_j \wedge p < q \wedge p \nparallel q$ gegeben. Falls es sich bei p, q um eine Schreib- und eine Leseoperation handelt, verzögert das quasi-rigorose Protokoll des rw-Planers die Ausführung von q mindestens bis *nach* der letzten Operation aus $T_i \setminus \{c_i\}$. Die Klausel ist dann also erfüllt.

Angenommen p ist eine m-Operation, q eine Schreiboperation, und es gibt ein $m_i^{k,l}$ mit $q > m_i^{k,l}$. Dann findet die Verifikation von T_i nach q statt. Sie wird aber scheitern wegen p. Hat man andererseits $p = w_i[x]$ und $q = m_j^{k,l}$, dann muss eine der beiden Transaktionen abbrechen oder es gibt ein $r_k^l[x]$ mit $w_i[x] < r_k^l[x] < m_j^{k,l}$, denn sonst wäre die erzeugte Historie nicht aktuell. Falls $i = k$ gilt, folgt $w_i[x] < r_i^l[x] < c_i < m_j^{k,l}$, weil der Methoden-Cache Kaskadenfreiheit sicherstellt. Für $i \neq k$ gilt wegen der Kaskadenfreiheit des rw-Planers $w_i[x] < c_i < r_k^l[x] < m_j^{k,l}$, und somit ist die Klausel erfüllt.

Zur zweiten Klausel von von Definition 20: Angenommen man hat $m_i^{k,l} < w_j[x] < u$ mit $u \in T_i$. Dann findet die Verifikation von T_i nach $w_j[x]$ statt und scheitert.

Umsetzung Abbildungen 12 und 13 zeigen UML-Klassendiagramme für mögliche Umsetzungen des Sperrprotolls beim Methoden-Cache bzw. beim m-Planer.

Im Methoden-Cache aus Abbildung 12 nimmt die Liste `cacheHitCalls` in der Klasse `LockMCTransaction` die Identifikatoren aller neuen Methodenaufrufe auf, deren Resultate aus dem Cache gelesen wurden. `cacheHitCalls` entspricht also der Liste L_i von

oben.

Die überschriebene Version von `handleCacheHit()` in der Klasse `LockMCProtocol` befüllt die Liste `cacheHitCalls` geeignet. `createTxContext()` erzeugt Instanzen des Typs `LockMCTransactionContext` und versorgt sie mit den neuesten Einträgen aus der `cacheHitCalls`-Liste. Die Einträge gelangen dann über einen Dienstmethodenaufruf zum m-Planer und können daher aus der `cacheHitCalls`-Liste entfernt werden. `createMCTransaction()` kreiert nun natürlich Transaktionen des Typs `LockMCTransaction`.

Auf der Seite des m-Planers verarbeiten die Methoden `afterMethodCall()` und `beforeCommit()` aus der Klasse `LockMTransaction` die Einträge, die der Methoden-Cache mittels der Liste `LockMCTransactionContext.newCacheHitCalls` zum Server sendet. `beforeCommit()` sichert die empfangene Liste lediglich zur Weiterverarbeitung für einen nachfolgenden `prepare()`-Aufruf.

`afterMethodCall()` testet hingegen direkt, ob die `MethodCallId`-Einträge aus der `newCacheHitCalls`-Liste sowie der `cacheHitCalls`-Liste noch gültig sind. Dazu verwendet `afterMethodCall()` die Methode `LockMTransaction.checkCacheHitCalls()`. Die letztere kopiert zunächst die Einträge aus der Liste `newCacheHitCalls` nach `cacheHitCalls`. Dann ruft sie für jedes Element aus `cacheHitCalls` die Methode `LockMScheduler.isValidCall()` auf und prüft so, ob ein entsprechendes `MethodCallId`-Objekt noch in den Tabellen des m-Planers vorhanden ist. Ist dies nicht der Fall, so hat der m-Planer die entsprechenden Einträge durch den Aufruf von `MScheduler.write()` aus Abbildung 10 bereits gelöscht. Gemäß dem oben skizzierten Verfahren hat also bereits eine damit zusammenhängende Operation $w_i[x]$ stattgefunden.

Da der m-Planer die aktuelle Transaktion auch später keines Falls mehr verifizieren kann, bricht er sie unmittelbar ab. Dies ist eine Optimierung gegenüber dem oben geschilderten Verfahren, die einen weiteren (unnötigen) Ressourcenverbrauch der abzubrechenden Transaktion verhindert. Man spricht in diesem Zusammenhang von einem so genannten „frühen Abbruch“ (Early Abort, [14]).

Zur Verifikation einer Transaktionsinstanz ruft die Methode `LockMTransaction.prepare()` ebenfalls die Methode `checkCacheHitCalls()` auf. Wenn `prepare()` daraufhin den Wert `false` liefert, kommt es zum Transaktionsabbruch. Die Liste `cacheHitCalls` entspricht übrigens der Liste L_i *auf der Seite des* m*-Planers*.

7.2 Aktualitätsbasiertes Zeitstempelprotokoll

7.2.1 Idee

Das in diesem Abschnitt vorgestellte Zeitstempelprotokoll für Methoden-Cache-Historien gründet auf der Idee, Transaktionen vermittels einer Zeitstempelfunktion $ts : T_i \mapsto t, t \in \mathbb{R}$ total zu ordnen. Wenn man die Ordnung von zueinander in Konflikt stehenden Operationen an der durch ts induzierten Ordnung „orientiert“, kann man Serialisierbarkeit erreichen. In [4] findet man dazu in Abschnitt 4.2 die so genannte

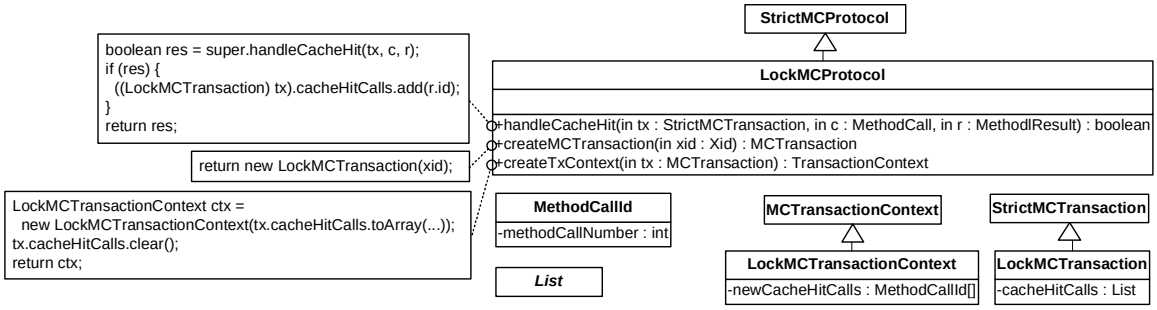

Abbildung 12: Erweiterung der Methoden-Cache-Klassen aus Abbildung 9 und 11 zur Umsetzung des Sperrprotokolls

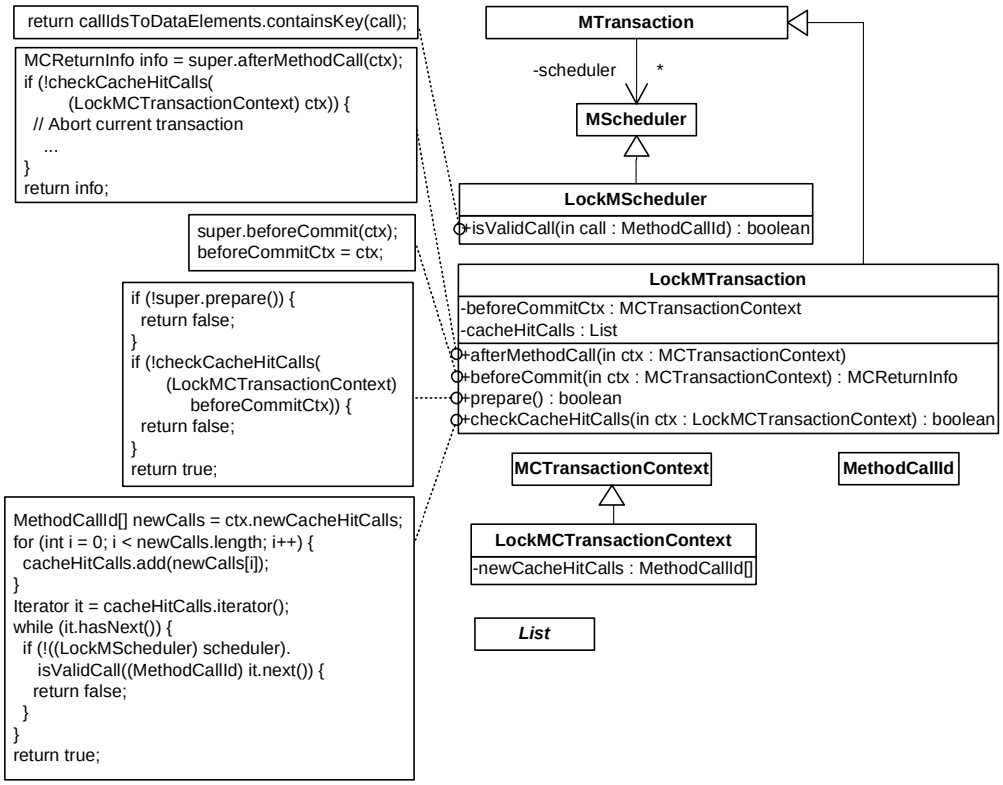

Abbildung 13: Erweiterung der m-Planer-Klassen aus Abbildung 10 zur Umsetzung des Sperrprotokolls

Zeitstempelregel, die die grundsätzliche Funktionsweise eines Zeitstempelprotokolls *für rw-Historien* beschreibt.

Zeitstempelregel: Wenn p_i und q_j Operationen sind, die zueinander in Konflikt stehen, dann bearbeite p_i vor q_j, falls $ts(T_i) < ts(T_j)$.

Für rw-Historien, die diese Regel befolgen, gilt bezüglich Kanten $T_i \to T_j$ im Serialisierbarkeitsgraphen SG dann offenbar immer $ts(T_i) < ts(T_j)$. Es kann also keine Zyklen $T_i \to \ldots \to T_i$ in SG geben, denn sonst käme der Widerspruch $ts(T_i) < \ldots < ts(T_i)$ zu Stande.

Die Zeitstempelregel kann auch im Zusammenhang mit Methodenoperationen und Methoden-Cache-Historien in angepasster Weise übernommen werden. Problematisch sind hier vor allem solche Kanten im entsprechenden Serialisierbarkeitgraphen $MCSG$, die durch Operationen der Art $r_k^l[x] < w_i[x] < m_j^{k,l}$ auftreten (mit $c_i \in T_i, c_j \in T_j$). Man erhält dann nämlich gemäß Definition 13 eine Kante $T_j \to T_i \in MCSG$, die der Ordnung $w_i[x] < m_j^{k,l}$ bzw. $ts(T_i) < ts(T_j)$ entgegensteht. Eine derartige Kante wird im Folgenden *Rückwärtskante* genannt.

Eine Rückwärtskante „passt“ nicht zur Zeitstempelregel, die, wie oben angedeutet, für die Serialisierbarkeit den Zusammenhang $p_i < q_j \wedge p_i \nparallel q_j \Rightarrow T_i \to T_j \in MCSG$ voraussetzt (sofern $c_i \in T_i$, $c_j \in T_j$ gilt). Das nächste Beispiel illustriert diesen Sachverhalt.

Beispiel 18. *Die folgende Methoden-Cache-Historie ist nicht serialisierbar, obwohl die Zeitstempelregel für die Zeitstempelfunktion $ts(T_i) = i$ erfüllt ist:*

$$(H_1, <) = r_1^1[x]c_1w_2[x]c_2r_3[x]m_3^{1,1}c_3.$$

Der zugehörige Serialisierbarkeitsgraph $MCSG$ ist zyklisch und enthält die Kanten $T_1 \to T_2$ (wegen $r_1^1[x] < w_2[x]$), $T_2 \to T_3$ (wegen $w_2[x] < r_3[x]$) und $T_3 \to T_2$ (wegen $r_1^1[x] < w_2[x] < m_3^{1,1}$).

Wenn man nun neben der Zeitstempelregel auch sicherstellt, dass Operationsfolgen wie in Beispiel 18 nicht auftreten, erreicht man Serialisierbarkeit. Diese Forderung wird im nächsten Abschnitt durch *Z-aktuelle* Methoden-Cache-Historien formalisiert. Auch die Eigenschaft, dass eine Methoden-Cache-Historie die Zeitstempelregel einhält, wird definiert und als *Z-geordnet* bezeichnet (mit „Z“ wie Zeitstempel). Danach lässt sich die Aussage beweisen, dass Z-aktuelle, Z-geordnete Methoden-Cache-Historien serialisierbar sind.

Offenbar verhält es sich mit dem obigen Problem im Hinblick auf unerwünschte Operationsfolgen von Historien ähnlich wie in Abschnitt 7.1. Deshalb lehnt sich der Begriff „Z-aktuell“ an den Begriff „aktuell“ aus Definition 19 an. Auch die Historie aus Beispiel 18 wurde bereits bei der Einführung des Sperrprotokolls angeführt (siehe Beispiel 12). Im Gegensatz zum Begriff „aktuell“ ergibt sich aber nun aus der Wahl der Zeitstempelfunktion ts ein neuer Freiheitsgrad. Ein Protokoll kann ihn eventuell zur Vermeidung von Transaktionsabbrüchen vorteilhaft ausnutzen.

Im Hinblick auf die Integration eines m-Planers und eines rw-Planers muss der m-Planer beim Zeitstempelprotokoll für die Einhaltung der Eigenschaft „Z-aktuell“ sorgen. Der rw-Planer garantiert hingegen, dass die rw-Projektion der Methoden-Cache-Historie Z-geordnet ist.

Leider reicht dies noch nicht aus, denn, wenn die rw-Projektion Z-geordnet ist, muss die entsprechende Methoden-Cache-Historie nicht notwendig Z-geordnet sein. In der Methoden-Cache-Historie können nämlich m-Operationen existieren, die nicht der Zeitstempelregel folgen. Der rw-Planer kann dies aber nicht bemerken, da er m-Operationen nicht betrachtet. Also muss der m-Planer garantieren, dass *alle* m-*Operationen* Z-geordnet sind. Dies wird im nächsten Abschnitt mit Hilfe des Begriffs „m-Z-geordnet“ formalisiert.

Insgesamt lässt sich die folgende Aussage treffen: Wenn eine Methoden-Cache-Historie bezüglich einer Zeitstempelfunktion Z-aktuell und m-Z-geordnet ist und darüber hinaus ihre rw-Projektion Z-geordnet ist, dann ist die Methoden-Cache-Historie serialisierbar.

Dass der rw-Planer Z-geordnete rw-Historien produziert, ist natürlich trivial, wenn er selbst ein Zeitstempelprotokoll verwendet. Die Zeitstempelfunktion des integrierten Planers (und somit des m-Planers) ist dann gerade die des rw-Planers. Was aber, wenn der rw-Planer ein anderes Protokoll anwendet? Zumindest für den Fall eines rw-Planers mit striktem Zwei-Phasen-Sperrprotokoll gibt es dazu eine wichtige Aussage (die man bereits in [4] Abschnitt 4.5 findet): Wählt man als Zeitstempelordnung die Commit-Ordnung (also die Reihenfolge der c_i-Operationen) des rw-Planers, dann ist die Historie, die der rw-Planer erzeugt, Z-geordnet.

Die Implementierungsbeschreibung für das Zeitstempelprotokoll des m-Planers in Abschnitt 7.2.4 berücksichtigt diese Aussage. Wie man darin sehen wird, lässt sich ein m-Planer, der „Z-aktuell“ und „m-Z-geordnet“ zusichert, sehr ähnlich implementieren wie der m-Planer für das Sperrprotokoll aus dem vorigen Abschnitt.

Die Umsetzung von „m-Z-geordnet“ ist dabei trivial, erfordert aber viel theoretische Vorarbeit. Ein m-Planer hat bei der Aufstellung einer Ordnung von Operationen einen Spielraum, der hauptsächlich durch die Operationsreihenfolge innerhalb einer Transaktion sowie durch die Abarbeitungsreihenfolge von Operationen im rw-Planer beschränkt ist. Der vorhandene Spielraum lässt sich nutzen, um „m-Z-geordnet“ über eine geeignete Umordnung von Operationen sozusagen automatisch zu garantieren.

Dazu betrachtet Abschnitt 7.2.3 Umordnungen von Methoden-Cache-Historien, wie sie aus Sicht des m-Planers stattfinden können, ohne dabei eine bereits gegebene Ordnung des rw-Planers für konfliktbehaftete Operationen zu verletzen. Zunächst werden zueinander äquivalente Methoden-Cache-Historien definiert – sie enthalten eine identische Menge von Operationen und belassen für Konflikte die Reihenfolge beteiligter Operationen. Zueinander äquivalente Methoden-Cache-Historien haben natürlich einen identischen Serialisierbarkeitsgraphen, da sich dessen Struktur ja gerade aus den Konfliktordnung von Operationen (bezüglich abschließender Transaktionen) ableitet (siehe Definition 13).

Satz 10 definiert zu einer Menge von Methoden-Cache-Transaktionen mit vorgegebener rw-Projektion $(H, <')$ eine passende Methoden-Cache-Historie, die implizit m-Z-geordnet ist. Die rw-Projektion der entstehenden Methoden-Cache-Historie $(H^{MC}, <)$ ist dabei äquivalent zu $(H, <')$. $(H^{MC}, <)$ repräsentiert daher eine gültige Sicht des m-Planers auf die Verarbeitungsreihenfolge aller eingangs gegebenen Operationen.

Aus Satz 10 lässt sich für die Implementierung eines Protokolls ableiten, wie der

m-Planer die Reihenfolge von Operationen „wahrnehmen“ darf bzw. muss, so dass „m-Z-geordnet“ von vornherein erfüllt ist. Als überraschende Konsequenz braucht dafür in der Implementierung gar nichts getan zu werden.

7.2.2 Basisformalismus

Für Zeitstempel-basierte Protokolle wird als wichtigstes Element eine Zeitstempelfunktion ts benötigt:

Definition 21. *Zeitstempelfunktion: Es sei $(H^{MC}, <)$ eine (wohl geordnete) Methoden-Cache-Historie mit den Transaktionen $\{(T_1, <_1), \ldots, (T_n, <_n)\}$. Eine Funktion $ts : \{T_1, \ldots, T_n\} \to \mathbb{R}$ heißt* Zeitstempelfunktion oder Z-Funktion *genau dann, wenn gilt: $\forall i, j \in \{1, \ldots, n\} : ts(T_i) = ts(T_j) \Rightarrow i = j$. ts induziert eine Totalordnung $<_{ts}$ auf der Menge $\{T_1, \ldots, T_n\}$ mit $T_i <_{ts} T_j :\Leftrightarrow ts(T_i) < ts(T_j)$.*

Die Ordnung konfliktbehafteter Operationen soll sich an einer Zeitstempelfunktion orientieren. Dies sichert die folgende Definition ab.

Definition 22. *Z-geordnete Methoden-Cache-Historie: Eine (wohl geordnete) Methoden-Cache-Historie $(H^{MC}, <)$ mit den Transaktionen $\{(T_1, <_1), \ldots, (T_n, <_n)\}$ ist* Z-geordnet bezüglich der Z-Funktion ts *genau dann, wenn gilt:*

$$\forall p, q \in H^{MC} : \forall i, j \in \{1, \ldots, n\} : (p \in T_i \wedge q \in T_j \wedge p \nparallel q \wedge ts(T_i) < ts(T_j)) \Rightarrow$$

$$(a_i \in H^{MC} \vee a_j \in H^{MC} \vee p < q).$$

Wie bereit oben angedeutet, reicht „Z-geordnet“ für die Serialisierbarkeit von Methoden-Cache-Historien nicht aus. „Z-aktuell“ vermeidet ähnlich wie aktuell aus Definition 19, dass eine m-Operation einen veralteten Wert für ein Datenelement liest.

Definition 23. *Z-aktuelle Methoden-Cache-Historie: Eine (wohl geordnete) Methoden-Cache-Historie $(H^{MC}, <)$ mit den Transaktionen $\{(T_1, <_1), \ldots, (T_n, <_n)\}$ ist* Z-aktuell bezüglich der Z-Funktion ts *genau dann, wenn gilt:*

$$\forall i, j \in \{1, \ldots, n\} : \forall w_i[x], m_j^{k,l} \in H^{MC} : (w_i[x] \nparallel m_j^{k,l} \wedge ts(T_i) \leq ts(T_j)) \Rightarrow$$

$$(a_i \in H^{MC} \vee a_j \in H^{MC} \vee \forall r_k^l[x] \in H^{MC} : w_i[x] < r_k^l[x]).$$

Zusammen sind „Z-geordnet“ und „Z-aktuell“ hinreichend für die Serialisierbarkeit.

Satz 7. *Wenn eine Z-Funktion ts existiert, bezüglich derer eine (wohl geordnete) Methoden-Cache-Historie $(H^{MC}, <)$ Z-aktuell und Z-geordnet ist, dann ist $(H^{MC}, <)$ MC-serialisierbar.*

Beweis. Es sei $(H^{MC}, <)$ eine Z-aktuelle, Z-geordnete (wohl geordnete) Methoden-Cache-Historie bezüglich der Z-Funktion ts. Angenommen $(H^{MC}, <)$ ist nicht MC-serialisierbar. Dann gibt es einen Zyklus $T_{k_1} \to T_{k_2} \to \ldots \to T_{k_l} \to T_{k_1}$ im Serialisierbarkeitsgraphen $MCSG$ zu $(H^{MC}, <)$.

Zunächst wird nun gezeigt: $ts(T_i) < ts(T_j)$ gilt für jede Kante $T_i \rightarrow T_j$ im Serialisierbarkeitsgraphen $MCSG$. (Man beachte dabei stets, dass T_i und T_j abgeschlossene Transaktionen sind (also $c_i, c_j \in H^{MC}$).

Falls $T_i \rightarrow T_j$ aus der ersten Disjunktionsklausel von Definition 13 stammt, gibt es zwei Operationen $p \in T_i$, $q \in T_j$ mit $p < q \wedge p \nparallel q$, $i \neq j$. Es folgt $ts(T_i) < ts(T_j)$, weil $(H^{MC}, <)$ Z-geordnet ist. Angenommen dies wäre nicht so: $ts(T_j) = ts(T_i)$ ist ausgeschlossen durch $i \neq j$. Mit $ts(T_j) < ts(T_i)$ erhielte man aber $q < p$ wegen Definition 22 (Widerspruch).

Falls $T_i \rightarrow T_j$ aus der dritten Disjunktionsklausel von Definition 13 stammt, hat man entsprechende Operationen $w_i[x], r_k^l[x], m_j^{k,l}$ mit $w_i[x] < r_k^l[x] < m_j^{k,l}$, $i \neq j$. Es ergibt sich $ts(T_i) < ts(T_j)$, da $(H^{MC}, <)$ Z-geordnet ist.

Aus der zweiten Disjunktionsklausel von Definition 13 erhält man $r_k^l[x] < w_j[x]$. Falls $w_j[x] < m_i^{k,l}$ gilt, dann folgt $ts(T_j) \leq ts(T_i)$, da $(H^{MC}, <)$ Z-geordnet ist. Weil $(H^{MC}, <)$ auch Z-aktuell ist, ergibt sich daraus $w_j[x] < r_k^l[x]$ im Widerspruch zur Voraussetzung $r_k^l[x] < w_j[x]$. Hat man andererseits $r_k^l[x] < m_i^{k,l} < w_j[x]$, dann muss gemäß der zweiten Disjunktionsklausel aus Definition 13 $i \neq j$ gelten. Somit folgt wiederum $ts(T_i) < ts(T_j)$, da $(H^{MC}, <)$ Z-geordnet ist.

Insgesamt folgt also die Behauptung $T_i \rightarrow T_j \Rightarrow ts(T_i) < ts(T_j)$ für alle $T_i \rightarrow T_j \in MCSG$. Für den Zyklus $T_{k_1} \rightarrow T_{k_2} \rightarrow \ldots \rightarrow T_{k_l} \rightarrow T_{k_1}$ erhält man so die Zeitstempelordnung $ts(T_{k_1}) < ts(T_{k_2}) < \ldots < ts(T_{k_l}) < ts(T_{k_1})$ im Widerspruch zur Azyklität von $<$. □

Nun geht es um die Aufgabenteilung zwischen m-Planer und rw-Planer. Es wird unterstellt, dass der rw-Planer für die rw-Projektion von Methoden-Cache-Historien bereits „Z-geordnet“ garantiert. Damit die Methoden-Cache-Historie insgesamt Z-geordnet ist, muss der m-Planer noch für „m-Z-geordnet“ und sogar für „Z-aktuell“ sorgen. Zunächst zur Definition von „m-Z-geordnet“:

Definition 24. *m-Z-geordnete Methoden-Cache-Historie: Eine (wohl geordnete) Methoden-Cache-Historie $(H^{MC}, <)$ mit den Transaktionen $\{(T_1, <_1), \ldots, (T_n, <_n)\}$ ist* m-Z-geordnet bezüglich der Z-Funktion ts *genau dann, wenn gilt:*

$$\forall i, j \in \{1, \ldots, n\} : \forall m_i^{k,l}, w_j[x] \in H^{MC} : (m_i^{k,l} \nparallel w_j[x] \wedge ts(T_i) < ts(T_j)) \Rightarrow$$

$$(a_i \in H^{MC} \vee a_j \in H^{MC} \vee m_i^{k,l} < w_j[x]).$$

Lemma 5. *Jede bezüglich einer Z-Funktion ts Z-geordnete (wohl geordnete) Methoden-Cache-Historie ist m-Z-geordnet bezüglich ts.*

Beweis. Die Behauptung ergibt sich direkt durch Einsetzen von $m_i^{k,l}$ und $w_j[x]$ aus Definition 24 in Definition 22. □

Wie das folgende Beispiel belegt, kommt man mit einer Z-geordneten rw-Projektion in der Tat nicht aus – auch nicht, wenn die Methoden-Cache-Historie Z-aktuell ist.

Beispiel 19. *Nicht jede bezüglich einer Z-Funktion ts Z-aktuelle (wohl geordnete) Methoden-Cache-Historie $(H^{MC}, <)$, deren rw-Projektion $RW((H^{MC}, <))$ Z-geordnet ist bezüglich ts, ist selbst Z-geordnet bezüglich ts. Hierzu betrachte man die folgende Historie in Bezug auf die Z-Funktion $ts(T_i) = i$:*

$$(H_{10}, <) = r_1[x]w_2[x]r_2^1[x]c_2m_1^{2,1}c_1.$$

Mit der vorgeschlagenen Aufgabenteilung von m- und rw-Planer kann man wirklich Z-geordnete Methoden-Cache-Historien erzeugen, wie der folgende Satz bestätigt.

Satz 8. *Eine bezüglich einer Z-Funktion ts Z-aktuelle, m-Z-geordnete (wohl geordnete) Methoden-Cache-Historie $(H^{MC}, <)$ ist genau dann Z-geordnet bezüglich ts, wenn $RW((H^{MC}, <))$ Z-geordnet ist bezüglich ts.*

Beweis. Es sei $(H^{MC}, <)$ eine Z-aktuelle, m-Z-geordnete (wohl geordnete) Methoden-Cache-Historie bezüglich der Z-Funktion ts und $(H, <') = RW((H^{MC}, <))$ die zugehörige rw-Projektion.

„$\Rightarrow$“: Angenommen $(H^{MC}, <)$ ist Z-geordnet. Da RW lediglich Operationen entfernt, können in $(H, <')$ keine Konflikte bestehen, die es nicht auch in $(H^{MC}, <)$ gibt. $(H, <')$ ist somit (trivialerweise) Z-geordnet.

„$\Leftarrow$“: Angenommen $(H, <') = RW((H^{MC}, <))$ ist Z-geordnet. Für $(H^{MC}, <)$ genügt es dann Operationen $w_i[x], m_j^{k,l}$ mit $w_i[x] \nparallel m_j^{k,l}$ und $i \neq j$ zu prüfen. Falls $ts(T_j) < ts(T_i)$ gilt, folgt die Behauptung sofort, weil $(H^{MC}, <)$ m-Z-geordnet ist. Falls andererseits $ts(T_i) < ts(T_j)$ zutrifft, folgt $a_i \in H^{MC} \vee a_j \in H^{MC}$ oder $w_i[x] < r_k^l[x]$, weil $(H^{MC}, <)$ Z-aktuell ist. Mit $a_i \in H^{MC} \vee a_j \in H^{MC}$ ist die Eigenschaft „Z-geordnet“ trivialerweise erfüllt. Aus $w_i[x] < r_k^l[x]$ folgt andererseits $w_i[x] < r_k^l[x] < m_j^{k,l}$ und somit die Behauptung. □

Das nächste Beispiel zeigt, dass man die Forderung „Z-aktuell“ im vorangegangenen Satz nicht fallen lassen kann.

Beispiel 20. *Nicht jede bezüglich einer Z-Funktion ts m-Z-geordnete (wohl geordnete) Methoden-Cache-Historien $(H^{MC}, <)$, deren rw-Projektion $RW((H^{MC}, <))$ Z-geordnet ist bezüglich ts, ist selbst Z-geordnet bezüglich ts. Hierzu betrachte man die folgende Historie in Bezug auf die Z-Funktion $ts(T_i) = i$:*

$$(H_{11}, <) = r_1^1[x]c_1m_3^{1,1}w_2[x]c_2r_3[y]c_3.$$

Aus Satz 7 und Satz 8 ergibt sich direkt das nachstehende Korollar. Es bringt die wichtigsten Begriffe dieses Abschnitts zusammen und fixiert die Aufgaben des m-Planers für die Realisierung eines Zeitstempelprotokolls.

Korollar 5. *Eine bezüglich einer Z-Funktion ts Z-aktuelle, m-Z-geordnete (wohl geordnete) Methoden-Cache-Historie $(H^{MC}, <)$ ist MC-serialisierbar, wenn $RW((H^{MC}, <))$ bezüglich ts Z-geordnet ist.*

7.2.3 Integration mit Sperrprotokoll-basiertem rw-Planer

Kann man einen m-Planer, der ein Zeitstempelprotokoll verfolgt, mit einem rw-Planer kombinieren, der ein andersartiges Protokoll einsetzt? Zumindest für das strikte Zwei-Phasen-Sperrprotokoll ist dies möglich, sofern man eine geeignete Zeitstempelfunktion einsetzt. Der folgende Satz definiert eine entsprechende Zeitstempelfunktion und zeigt, dass sie zu Z-geordneten rw-Historien führt.[24]

Satz 9. *Es sei $(H, <)$ eine rw-Historie, die durch das strikte Zwei-Phasen-Sperrprotokoll erzeugt wurde und die Transaktionen $\{(T_1, <_1), \ldots, (T_n, <_n)\}$ besitzt. Dann ist $(H, <)$ Z-geordnet bezüglich einer Zeitstempelfunktion ts, die einer Transaktion $(T_i, <_i)$ mit $i \in \{1, \ldots, n\}$ zu Beginn der Operation $c_i \in H$ den jeweils jüngsten Zeitstempel zuordnet. ts heißt* Commit-Zeitstempelfunktion.

Beweis. Seien etwa $p \in T_i$, $q \in T_j$ mit $p \not\parallel q$ und $p < q$ sowie $c_i, c_j \in H$. Zum Ausführungsbeginn von c_i hält das strikte Zwei-Phasen-Sperrprotokoll noch eine Lese- bzw. eine Schreibsperre bezüglich p, so dass q noch nicht bearbeitet werden kann. Der Transaktionsverwalter erzeugt den nächsten Zeitstempel $ts(T_i)$, bevor er die Sperre bezüglich p freigibt. Insbesondere kann auch die Ausführung von c_j vor der Vergabe von $ts(T_i)$ noch nicht beginnen. Es folgt $ts(T_i) < ts(T_j)$ und mithin ist $(H, <)$ Z-geordnet. □

Der nächste Satz ist sehr bedeutsam für eine effiziente Implementierung des aktualitätsbasierten Zeitstempelprotokolls, aber leider nicht leicht erfassbar. Dem Satz gemäß kann ein m-Planer einerseits die Operationen einer Methoden-Cache-Historie als so geordnet betrachten, dass die Eigenschaft „m-Z-geordnet“ automatisch erfüllt ist, und andererseits dabei die Teilordnung von konfliktbehafteten Operationen, wie sie der rw-Planer erzeugt, erhalten. Voraussetzung hierfür ist allerdings, dass die Operationsordnung, die der rw-Planer generiert, Z-geordnet ist. Wie man in Abschnitt 7.2.4 sehen wird, muss als Konsequenz die Eigenschaft „m-Z-geordnet“ bei einer entsprechenden Implementierung des Zeitstempelprotokolls nicht explizit zugesichert werden.

Zunächst wird aber der Begriff der Äquivalenz von Methoden-Cache-Historien definiert. Er ist nötig, um die Aussage des Satzes zu formulieren.

Definition 25. *Äquivalente Methoden-Cache-Historien: Zwei (wohl geordnete) Methoden-Cache-Historien $(H_1^{MC}, <_1)$ und $(H_2^{MC}, <_2)$ sind* äquivalent, *geschrieben $(H_1^{MC}, <_1) \equiv (H_2^{MC}, <_2)$, genau dann, wenn gilt:*

$$H_1^{MC} = H_2^{MC} \wedge \{(p, q) \mid p <_1 q \wedge p \not\parallel q\} = \{(p, q) \mid p <_2 q \wedge p \not\parallel q\}.$$[25]

Wie bereits erwähnt haben zueinander äquivalente Methoden-Cache-Historien den gleichen Methoden-Cache-Serialisierbarkeitsgraphen. Dies folgt sofort aus Definition 13, denn sie leitet Graphenkanten ausschließlich aus der Konfliktordnung von Operationen (bezüglich abschließender Transaktionen) ab.

[24] Eine äquivalente Aussage findet man bereits in [4] Abschnitt 4.5.

[25] Die Definition stimmt nicht ganz mit einer ähnlichen Definition aus [4] Abschnitt 2.2 überein. Dort werden für den Äquivalenzbegriff lediglich die Ordnungen von konfliktbehafteten Operationen aus *abgeschlossenen* Transaktionen betrachtet.

Satz 10. *Es sei $(H^{MC}, <'')$ eine (nicht notwendig wohl geordnete) Methoden-Cache-Historie mit den Transaktionen $\{(T_1, <_1), \ldots, (T_n, <_n)\}$ und mit der wohl geordneten und bezüglich einer Zeitstempelfunktion ts Z-geordneten rw-Projektion $(H, <') = RW((H^{MC}, <''))$. Die Transaktionen $(T_i, <_i)$ seien intern „wohl geordnet", das heißt es gelte:*

$$\forall i \in \{1, \ldots, n\} : \forall p, q \in T_i : p \not\parallel q \Rightarrow p < q \vee q < p.$$

Weiter sei die Ordnung $<''$ zu H^{MC} gegeben durch

$$<'' = (<' \cup \bigcup_{i=1}^{n} <_i) \cup \{(r_k^l[x], m_i^{k,l}) \mid r_k^l[x], m_i^{k,l} \in H^{MC}\}.$$

Dann kann man eine wohl geordnete und bezüglich ts m-Z-geordnete Methoden-Cache-Historie $(H^{MC}, <)$ definieren, deren rw-Projektion äquivalent ist zu $(H, <')$, also $RW((H^{MC}, <)) \equiv (H, <')$ und die die Operationsordnungen $<_i$ für die Transaktionen T_i (mit $i \in \{1, \ldots, n\}$) beibehält. Die Ordnungsrelation $<$ ist dabei definiert durch $<:= (\prec)^*$.[26] mit

$$\prec:= (\bigcup_{i=1}^{n} <_i) \cup \{(p, q) \mid p <' q \wedge p \not\parallel q\} \cup \{(r_k^l[x], m_i^{k,l}) \mid r_k^l[x], m_i^{k,l} \in H^{MC}\} \cup$$

$$\{(m_i^{k,l}, w_j[x]) \mid ts(T_i) < ts(T_j) \wedge m_i^{k,l} \not\parallel w_j[x]\} \cup$$

$$\{(w_j[x], m_i^{k,l}) \mid ts(T_j) < ts(T_i) \wedge m_i^{k,l} \not\parallel w_j[x]\}.$$

Im obigen Satz kann man sich $(H, <')$ als die vom rw-Planer vorgegebene Ordnung von rw-Operationen vorstellen. $(H^{MC}, <'')$ ist eine Methoden-Cache-Historie, die die Ordnung des rw-Planers und die interne Ordnung der Transaktionen berücksichtigt, aber ansonsten alle Spielräume zur Komplettierung der Ordnung für den m-Planer offen lässt. Der m-Planer sollte also eine wohl geordnete Methoden-Cache-Historie mit den Operationen aus H^{MC} erzeugen, deren rw-Projektion äquivalent ist zu $(H, <')$. Der Satz beschreibt durch die Ordnung $\prec$ bzw. deren transitiven Abschluss $<$, wie man zu einer solchen Methoden-Cache-Historie gelangt, die überdies m-Z-geordnet ist.

Beweis von Satz 10. Offenbar ist $(H^{MC}, <)$ wohl geordnet, denn alle zueinander in Konflikt stehenden Operationen in H^{MC} sind zueinander geordnet. Für entsprechende Operationen aus der rw-Projektion von $(H^{MC}, <)$ folgt dies aus der Voraussetzung des Satzes. Ebenso gilt dies für entsprechende m- und w-Operationen aus der gleichen Transaktion T_i. Es bleiben lediglich konfliktbehaftete Paare aus m- und w-Operationen aus unterschiedlichen Transaktionen zu betrachten, aber diese werden durch die Definition von $<$ geordnet.

$(H^{MC}, <)$ ist m-Z-geordnet und behält die Operationsordnungen $<_i$ bei. Beides ergibt sich sofort aus der Definition von $<$.

$RW((H^{MC}, <))$ ist äquivalent zu $(H, <')$: Wegen des Ausdrucks $\{(p, q) \mid p <' q \wedge p \not\parallel q\}$ in der Definition von $\prec$ kommen alle betreffenden Operationspaare in $<$ vor.

[26] Das hochgestellte Symbol $*$ bezeichnet den transitiven Abschluss einer Relation.

Es bleibt zu zeigen, dass $<$ tatsächlich eine (partielle) Ordnungsrelation bildet und somit keine zwei Operationspaare $p < q$ und $q < p$ *zusammen* vorkommen. Hierzu genügt es, zwei (ähnliche) Fälle für Operationen aus H^{MC} auszuschließen, nämlich $w_j[x] < m_i^{k,l} \wedge w_j[x] \nparallel m_i^{k,l} \wedge ts(T_i) < ts(T_j)$ sowie $m_i^{k,l} < w_j[x] \wedge m_i^{k,l} \nparallel w_j[x] \wedge ts(T_j) < ts(T_i)$ jeweils mit $i \neq j$, $i, j \in \{1, \ldots, n\}$.

Beim ersten Fall muss eine Operation $p_i \in T_i$ existieren, so dass $w_j[x] < p_i < m_i^{k,l}$ gilt. (Ansonsten könnte die Beziehung $w_j[x] < m_i^{k,l}$ mit $ts(T_i) < ts(T_j)$ gemäß der Definition von $<$ nicht bestehen.)

Durch Induktion über die Länge des Ordnungspfades $(w_j[x] < p_i) = (w_j[x] \prec \ldots \prec p_i)$ (mit $i \neq j$) wird nun gezeigt: $ts(T_j) < ts(T_i)$.

Zum Induktionsanfang, also $w_j[x] <^1 p_i$: Gemäß der Definition von $\prec$ muss $w_j[x] \nparallel p_i$ und weiter $ts(T_j) < ts(T_i)$ gelten, denn falls p_i eine Lese- oder Schreiboperation ist, folgt dies weil $(H, <')$ Z-geordnet ist. Ansonsten ist p_i eine Methoden-Operation und die Behauptung folgt direkt aus der Definition von $\prec$.

Induktionsschritt: Man hat $w_j[x] < q_k \prec p_i$ mit $ts(T_j) < ts(T_k)$ als Induktionsannahme. Für den Fall $i = k$ ist man bereits fertig. Falls $i \neq k$: Gemäß der Definition von $\prec$ muss $q_k \nparallel p_i$ gelten. Falls es sich bei q_k und p_i um eine Lese- und eine Schreiboperation handelt, gilt $q_k <' p_i$ und somit $ts(T_k) < ts(T_i)$, weil $(H, <')$ Z-geordnet ist. Ansonsten ist eine der beiden Operationen eine Methoden-Operation und $ts(T_k) < ts(T_i)$ ergibt sich sofort aus der Definition von $\prec$. Insgesamt erhält man $ts(T_j) < ts(T_k) < ts(T_i)$ im Widerspruch zur Voraussetzung für den untersuchten ersten Fall.

Die Argumentation für den zweiten Fall ist sehr ähnlich (und wird deshalb nicht weiter ausgeführt). □

Man kann sich auch umgekehrt die Frage stellen, wie ein rw-Planer, der ein Zeitstempelprotokoll einsetzt, mit einem m-Planer zusammenarbeitet, der das Sperrprotokoll aus Abschnitt 7.1 verwendet. Leider stellt ein striktes Zeitstempelprotokoll für einen rw-Planer aber nicht einmal die für das Sperrprotokoll des m-Planers bedeutsame Eigenschaft „quasi-rigoros“ sicher. Das folgende Beispiel illustriert dies.

Beispiel 21. *Nicht jede rw-Historie, die durch ein striktes Zeitstempelprotokoll erzeugt wird, für das $p < q \wedge p \nparallel q \Rightarrow ts(T(p)) < ts(T(q))$ gilt, ist quasi-rigoros. Man betrachte etwa $(H_{12}, <) = r_1[x]w_2[x]r_1[y]c_1c_2$ mit $ts(T_1) < ts(T_2)$.*

7.2.4 Implementierung

Dieser Abschnitt stellt eine Implementierung des aktualitätsbasierten Zeitstempelprotokolls vor, die auf einem rw-Planer mit striktem Zwei-Phasen-Sperrprotokoll basiert. Als Zeitstempelfunktion wählt man die Commit-Zeitstempelfunktion aus Satz 9.

Interessanterweise benötigt das resultierende Verfahren keine Zeitstempel im eigentlichen Sinne (etwa als Identifikatoren) – die Zeitstempel leiten sich als abstrakte Größen implizit aus der Reihenfolge der Commit-Operationen ab. Wie bereits erwähnt, ähnelt das Verfahren demjenigen aus Abschnitt 7.1.3, bringt aber eine entscheidende Verbesserung mit sich.

Verfahren Der m-Planer geht wie folgt vor: Für Operationsfolgen der Form $r_k^l[x]r_k^l[y]\ldots$, die zu einen zu einem cachenden Methodenaufruf gehören, merkt er sich die Beziehung zwischen dem Tupel (k,l) zur Identifikation des Methodenaufrufs und den dabei gelesenen Datenelementen $x, y, \ldots$ in einer Relation V. V enthält also Einträge der Form $(\{x, y, \ldots\}, (k,l))$. Tritt eine Operation der Form $w_h[z]$ auf, dann trägt der m-Planer für das Datenelement z alle Einträge aus V, die z im ersten Tupelelement enthalten, in eine (initial leere) Liste K_h ein. K_h enthält also Elemente der Form $(\{z, \ldots\}, (s,t))$ und gehört zur Transaktion T_k.

Auf der Klientenseite merkt sich der Methoden-Cache für eine Transaktion T_i alle Tupel (k,l), die von T_i von gelesene, gecachte Methodenresultate identifizieren in einer (initial leeren) Liste L_i. Ein Tupel $(k,l) \in L_i$ bedeutet also, dass T_i auf der Klientenseite die Operation $m_i^{k,l}$ ausgeführt hat. Sobald T_i abschließen soll, schickt der Methoden-Cache L_i zum m-Planer. Dort findet dann eine Verifikation statt.

Zur Verifikation prüft der m-Planer für jedes Tupel (k,l) aus L_i, ob es einen Eintrag $(\{\ldots\}, (k,l))$ in V gibt. Wenn dies wenigstens für ein Element aus der Liste L_i *nicht* der Fall ist, dann wird T_i abgebrochen. Der m-Planer sendet dann also ein a_i an die beteiligten Ressourcenverwalter. Andernfalls nutzt er die Liste K_i (die er bezüglich T_i erstellt hat) und entfernt für jedes Element aus K_i den entsprechenden Eintrag aus V. Danach gibt er ein c_i an die Ressourcenverwalter weiter.

Die Verifikation findet als unteilbarer Arbeitsschritt statt, das heißt, solange der m-Planer T_i verifiziert, darf er keine andere Transaktion verifizieren. Darüberhinaus sichern der Methoden-Cache und der m-Planer die Invalidierung von gecachten-Methodenresultaten entsprechend Abschnitt 5 ab.

Im Vergleich mit dem Sperrprotokollimplementierung aus Abschnitt 7.1 liegt die Verbesserung des Verfahrens offenbar darin, dass sich das Löschen von Einträgen aus V wegen einer Schreiboperation $w_i[x]$ bis zum Abschuss von T_i verzögert. Dadurch verbessert sich die Chance zum Abschluss für solche Transaktionen, die auf entsprechende Einträge in V zuzugreifen versuchen. (Sie müssen lediglich vor T_i verifiziert werden.) Beispielsweise wäre die Historie $r_1^1[x]c_1m_2^{1,1}w_3[x]c_2c_3$ durch die Sperrprotokollimplementierung nicht möglich – der m-Planer müsste T_2 während der Validierung abbrechen. Das eben beschriebene Verfahren lässt die Historie hingegen zu.

Korrektheit Das beschriebene Verfahren erzeugt Z-aktuelle Methoden-Cache-Historien für eine Commit-Zeitstempelfunktion: Sei zunächst $w_i[x] \not\Vdash m_j^{k,l} \wedge ts(T_i) < ts(T_j)$ und $r_k^l[x] < w_i[x]$. Die Commit-Zeitstempelfunktion impliziert, dass die Bearbeitung von c_i vor c_j beginnt, und somit $(\{x, \ldots\}, (k,l))$ aus V entfernt wurde, bevor T_j mit der Verifikation anfängt. (Man erinnere sich, dass die Verifikation ein unteilbarer Arbeitsschritt ist.) Im Widerspruch zur Annahme kann T_j dann nicht verifiziert werden, also sichert das Verfahren für diesen Fall „Z-aktuell“ ab. Man berücksichtige auch, dass die Verifikation von T_i wegen der Unteilbarkeitsforderung sicher vor der Verifikation von T_j abgeschlossen wird.

Falls $w_i[x] \not\Vdash m_j^{k,l} \wedge ts(T_i) = ts(T_j)$ und $r_k^l[x] < w_i[x]$ gilt, hat man die Situation $r_k^l[x] < w_i[x] < m_i^{k,l}$. In diesem Fall invalidiert spätestens $w_i[x]$ gemäß Abschnitt 5

das durch $r_k^l[x]$ berechnete Methodenresultat beim Cache oder entfernt einen entsprechenden Eintrag $(\{x, y, \ldots\}, (k, l))$ aus V (oder tut beides). Im Widerspruch zur Annahme kann also $m_i^{k,l}$ entweder nicht stattfinden, da kein entsprechender Wert mehr im Methoden-Cache vorliegt, oder T_i kann nicht verifiziert werden. Insbesondere bestätigt dies die Eigenschaft „Z-aktuell".

Wegen Satz 10 ist die Forderung „m-Z-geordnet" automatisch erfüllt.

Umsetzung Wie das Verfahren selbst, so unterscheidet sich auch dessen Umsetzung wenig von dem Vorschlag aus Abschnitt 7.1.3. Abbildung 14 zeigt die Änderungen, die sich für den UML-Entwurf des m-Planers ergeben.

Die Klasse `TStamp1MTransaction` bildet eine Unterklasse von `LockMTransaction` aus Abbildung 13. Das Attribut `invalidateAtCommit` entspricht dabei der Liste K_i von weiter oben. Es speichert Identifikatoren von Methodenaufrufen (`MethodCallId`-Objekte), damit entsprechende Einträge beim Abschluss der Transaktion aus den Tabellen des m-Planers entfernt werden können. Das Entfernen selbst erledigt die veränderte Implementierung von `prepare()`: Am Ende der Methode findet dazu für alle Elemente der `invalidateAtCommit`-Liste der Methodenaufruf `scheduler.removeCall(...)` statt. Die weiter oben gestellte Forderung zur Unteilbarkeit der Verifikation wird in `prepare()` durch `synchronized(scheduler) { ... }` erreicht. Ansonsten ist `prepare()` analog zu seiner überschriebenen Variante aus der Klasse `LockMTransaction` aufgebaut.

Die Methode `write()` in `TStamp1MScheduler` löscht, im Gegensatz zu ihrer überschriebenen Variante, Methodenaufrufe, die von einem Datenelement `x` abhängen, nicht aus den Tabellen des m-Planers, sondern sichert die Methodenaufrufe in der betreffenden `invalidateAtCommit`-Liste. Der Aufruf `invalidatedCalls.add(call)` stellt aber nach wie vor sicher, dass entsprechende Methodenresultate beim Methoden-Cache invalidiert werden (siehe dazu Abschnitt 5.4).

7.3 Passungsbasiertes Zeitstempelprotokoll

7.3.1 Idee

Den Protokollen aus den vorangegangenen beiden Abschnitten war gemeinsam, dass sie in Methoden-Cache-Historien für die Aktualität von m-Operationen sorgten. Dies wurde durch die Eigenschaft „aktuell" aus Definition 19 bzw. durch „Z-aktuell" aus Definition 23 erreicht. Das Protokoll dieses Abschnitts unterscheidet sich in diesem Punkt von seinen Vorgängern, weil es das Lesen von veralteten, gecachten Methodenresultaten in einem gewissen Rahmen erlaubt. Der Name „passungsbasiertes Zeitstempelprotokoll" weist darauf hin, dass das Protokoll einerseits wieder von der Zeitstempelregel Gebrauch macht. Andererseits bedeutet „passungsbasiert", dass es Transaktionen geben kann, die der Zeitstempelregel widersprechen, solange sie, gemäß einer noch zu bestimmenden Passungsregel, keine Zyklen im Serialisierbarkeitsgraphen hervorrufen.

Zum Verständnis der Problematik sei hier eine leicht veränderte Variante von Bei-

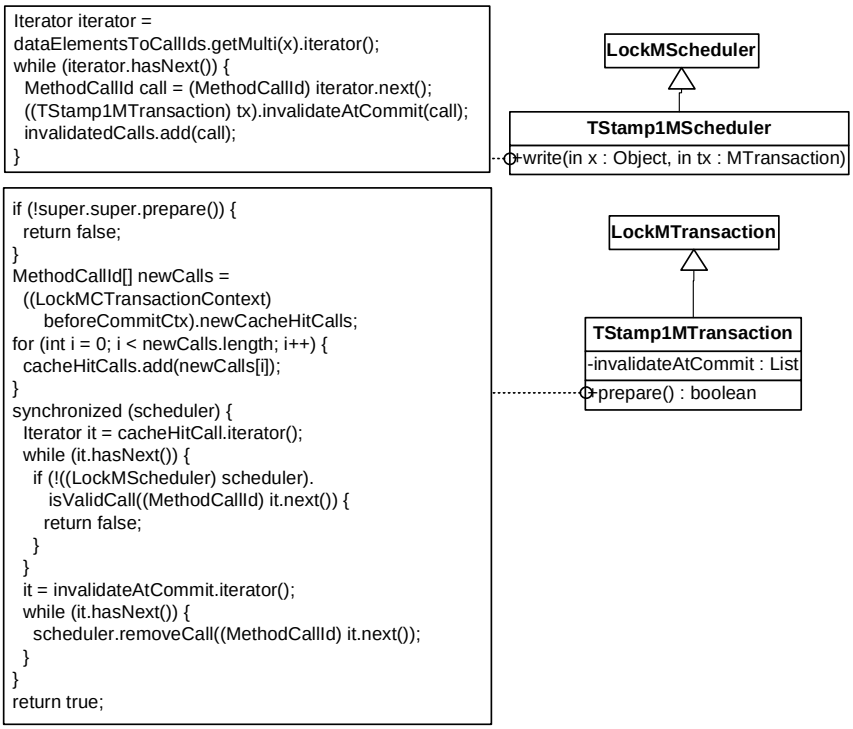

Abbildung 14: Erweiterung der m-Planer-Klassen `LockMScheduler` und `LockMTransaction` aus Abbildung 13 zur Umsetzung des aktualitätsbasierten Zeitstempelprotokolls

spiel 18 angeführt.[27]

Beispiel 22. *Die folgende Methoden-Cache-Historie ist nicht serialisierbar, obwohl die Zeitstempelregel für die Zeitstempelfunktion $ts(T_i) = i$ erfüllt ist:*

$$(H_1, <) = r_1^1[x] c_1 w_2[x] c_2 m_3^{1,1} r_3[x] c_3.$$

Der zugehörige Serialisierbarkeitsgraph $MCSG$ ist zyklisch und enthält die Kanten $T_1 \to T_2$ (wegen $r_1^1[x] < w_2[x]$), $T_2 \to T_3$ (wegen $w_2[x] < r_3[x]$) und $T_3 \to T_2$ (wegen $r_1^1[x] < w_2[x] < m_3^{1,1}$).

Im obigen Beispiel widerspricht die Kante $T_3 \to T_2 \in MCSG$ der Zeitstempelregel und wird in Abschnitt 7.2 durch „Z-aktuell“ ausgeschlossen. Was aber, wenn man die Kante zulässt? Dann muss man im Gegenzug die Kante $T_2 \to T_3$ verbieten, die wegen $w_2[x] < r_3[x]$ existiert.

Aus der Sicht eines m-Planers ist interessanterweise bereits *direkt nach einer entsprechenden Verarbeitung von $m_3^{1,1}$* klar, dass man Konfliktkanten von T_2 nach T_3 ausschließen

[27] Dabei wurden lediglich die Operationen $r_3[x]$ und $m_3^{1,1}$ vertauscht.

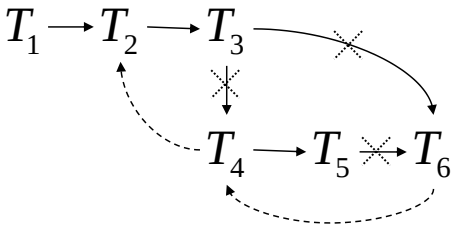

Abbildung 15: Ein Serialisierbarkeitsgraph, der die Idee des passungsbasierten Zeitstempelprotokolls illustriert

muss. Um potentielle Zyklen im Serialisierbarkeitsgraphen zu vermeiden, sollte T_3 dann nur noch eingehende Konfliktkanten von Transaktionen *mit einem kleineren Zeitstempel als* T_2 erhalten dürfen. Die Operation $r_3[x]$ verletzt diese Forderung (denn sie führt zu $T_2 \rightarrow T_3$), und daran könnte der m-Planer erkennen, dass er T_3 abbrechen muss.

Abbildung 7.2 verdeutlicht die Überlegungen an einem etwas allgemeineren Fall. Zur Illustration wird nur ein entsprechender Serialisierbarkeitsgraph betrachtet – die zu Grunde liegende Methoden-Cache-Historie kann man hierbei vernachlässigen.

Als Zeitstempelfunktion kommt wieder $ts(T_i) = i$ zum Einsatz. Kanten, die die Zeitstempelregel befolgen sind durchgezogen; Rückwärtskanten, also Kanten mit $T_j \rightarrow T_i, ts(T_i) \leq ts(T_j)$ sind gestrichelt dargestellt. Offenbar enthält der Graph Zyklen. Lässt man die beiden Rückwärtskanten $T_4 \rightarrow T_2$ und $T_6 \rightarrow T_4$ zu, so kann man die Zyklen entfernen, in dem man die mit einem Kreuzchen markierten Kanten ausschließt.

Die zu entfernenden Kanten widersprechen der (verallgemeinerten) Forderung von oben: Wegen $T_6 \rightarrow T_4$ sollte man etwa für T_6 nur Eingangskanten $T_i \rightarrow T_6$ mit $i < 4$ zulassen. Deshalb ist $T_5 \rightarrow T_6$ markiert. Analoge Betrachtungen gelten für die markierte Kante $T_3 \rightarrow T_4$, die wegen $T_4 \rightarrow T_2$ zum Zyklus führt.

Die Kante $T_3 \rightarrow T_6$ erfüllt aber bereits die Forderung $T_i \rightarrow T_6$ mit $i > 4$. Warum entsteht dann trotzdem ein Zyklus, sofern man sie beibehält? Offenbar ist die Forderung $i > 4$ zu schwach, um den Zyklus zu vermeiden. Der Grund ist, dass man über die Rückwärtskante $T_4 \rightarrow T_2$ von T_6 bis zurück zu T_2 gelangen kann. Es also reicht nicht aus, einzelne Rückwärtskanten zu betrachten, sondern man muss *Pfade von Rückwärtskanten* einbeziehen. Im vorliegenden Fall sollte man daher $T_i \rightarrow T_6$ mit $i > 2$ fordern. Somit ist $T_3 \rightarrow T_6$ zu recht in der Abbildung markiert.

Der Formalismus des nächsten Abschnitts gibt mit der Definition von „Z-passend“ ein allgemeines Kriterium für den Zeitstempel einer Transaktion T_i an, die Kanten $T_i \rightarrow T_j \in MCSG$ mit $ts(T_i) < ts(T_j)$ zur Transaktion T_j besitzt. Analog zu der Forderung aus dem Beispiel von oben, muss dazu $ts(T_i) < ts_{fit}(T_j)$ gelten. ts_{fit} ist dabei eine Funktion, die den längsten Pfad aus Rückwartskanten von T_j aus berechnet und den Zeitstempel der Transaktion am Ende Pfades liefert. ts_{fit} untersucht dazu die zu Grunde liegende Methoden-Cache-Historie.

„Z-passend“ ist der wichtigste Bestandteil für ein passungsbasiertes Zeitstempelprotokoll, das Serialisierbarkeit garantiert. Daneben benötigt man für den zugehörigen

Serialisierbarkeitsbeweis noch die beiden Randbedingungen „irreflexiv“ und „rm-Z-geordnet“. Ihre Definition ist leicht zu verstehen, und bezüglich einer entsprechenden m-Planer-Implementierung aus Abschnitt 7.3.3 verhalten sich diese Eigenschaften unkritisch.

7.3.2 Formalismus

Zunächst folgt die Definition der Funktion ts_{fit}. Vereinfachend berechnet sie zu einer abschließenden Transaktion T_i aus einer Methoden-Cache-Historie $(H^{MC}, <)$ den Zeitstempel der abgeschlossenen Transaktion am Ende des längsten Pfades aus Rückwärtskanten. Der Pfad beginnt dabei mit T_i und entsteht durch Methodenoperationen ähnlich wie in Beispiel 22. (Man erinnere sich, dass für eine Rückwärtskante $T_j \to T_i \in MCSG$ gerade $ts(T_i) \leq ts(T_j)$ gilt.)

Definition 26. *Z-passender Zeitstempel ts_{fit}: Es sei $(H^{MC}, <)$ eine (wohl geordnete) Methoden-Cache-Historie mit den Transaktionen $\{(T_1, <_1), \ldots, (T_n, <_n)\}$ und ts eine Zeitstempelfunktion. Der* Z-passende Zeitstempel $ts_{fit} : \{T_i \mid i \in \{1, \ldots, n\} \wedge c_i \in T_i\} \to \mathbb{R}$ *berechnet sich dann wie folgt:*

$$ts_{fit}(T_i) = \min(\{ts(T_i)\} \cup$$

$$\{\, ts_{fit}(T_j) \mid \exists w_j[x], m_i^{k,l}, r_k^l[x] \in H^{MC} : r_k^l[x] < w_j[x] \wedge ts(T_j) < ts(T_i) \wedge c_j \in H^{MC}\}).$$

Lemma 6. *Die Funktion ts_{fit} ist wohldefiniert.*

Beweis. Man betrachte $ts_{fit}(T_i)$ entsprechend Definition 26. Zunächst ist das Argument von $\min$ nicht die leere Menge, da $ts(T_i)$ darin enthalten ist. Weiter ist jedes in der Menge $\{ts_{fit}(T_j) \mid \ldots\}$ referenzierte T_j abgeschlossen (also $c_j \in T_j$) und liegt somit im Definitionsbereich von ts_{fit}.

Für jedes in der Menge $\{ts_{fit}(T_j) \mid \ldots\}$ referenzierte T_j gilt, dass $ts(T_j)$ echt kleiner ist als $ts(T_i)$. Da es höchstens n Zeitstempel im Bild von ts gibt, terminiert die Berechnung von $ts_{fit}(T_i)$ spätestens beim kleinsten Zeitstempel. □

Mit Hilfe der Funktion ts_{fit} kann man nun Z-passende Methoden-Cache-Historien definieren: In solchen Historien müssen Konfliktkanten $T_i \to T_j$, welche die Zeitstempelordnung einhalten, von Transaktionen T_i ausgehen, deren Zeitstempel kleiner ist als $ts_{fit}(T_j)$.

Definition 27. *Z-passende Methoden-Cache-Historie: Es sei $(H^{MC}, <)$ eine (wohl geordnete) Methoden-Cache-Historie mit den Transaktionen $\{(T_1, <_1), \ldots, (T_n, <_n)\}$ und dem Serialisierbarkeitsgraphen $MCSG$. $(H^{MC}, <)$ ist* Z-passend bezüglich der Zeitstempelfunktion ts *genau dann, wenn gilt:*

$$\forall i, j \in \{1, \ldots, n\} : (T_i \to T_j \in MCSG \wedge ts(T_i) < ts(T_j)) \Rightarrow ts(T_i) < ts_{fit}(T_j).$$

Als nächstes soll die gewünschte Aussage zur Serialisierbarkeit Z-passender Methoden-Cache-Historien entwickelt werden. Dazu sind noch zwei zusätzliche Definitionen nötig: „Irreflexiv“ verhindert Zyklen der Form $T_i \rightarrow T_i$ im Serialisierbarkeitsgraphen, denn „Z-passend“ schließt diese noch nicht aus. „rm-Z-geordnet“ garantiert, dass für eine Operationsfolge $w_i[x] < r_k^l[x] < m_j^{k,l}$, die Transaktion T_i einen kleineren Zeitstempel besitzt als T_j (sofern die beiden Transaktionen abschließen).

Definition 28. *Irreflexive Methoden-Cache-Historie: Eine (wohl geordnete) Methoden-Cache-Historie $(H^{MC}, <)$ ist* irreflexiv *genau dann, wenn gilt:*

$$\neg\exists w_i[x], m_i^{k,l}, r_k^l[x] \in H^{MC} : r_k^l[x] < w_i[x] < m_i^{k,l} \wedge c_i \in H^{MC}.$$

Definition 29. *rm-Z-geordnete Methoden-Cache-Historie: Eine (wohl geordnete) Methoden-Cache-Historie $(H^{MC}, <)$ ist rm-Z-*geordnet *genau dann, wenn gilt:*

$$\forall i, j \in \{1, \ldots, n\} : (\exists w_i[x], m_j^{k,l}, r_k^l[x] \in H^{MC} : w_i[x] < r_k^l[x] < m_j^{k,l}) \Rightarrow$$

$$(a_i \in H^{MC} \vee a_j \in H^{MC} \vee ts(T_i) < ts(T_j)).$$

Wie man noch genauer sehen wird, sichert der Invalidierungsmechanismus aus Kapitel 5 bereits „irreflexiv“ ab. Das nächste Lemma belegt darüber hinaus: Wenn man einen rw-Planer nutzt, der das strikte Zwei-Phasen-Sperrprotokoll anwendet, so ist „rm-Z-geordnet“ für eine Commit-Zeitstempelfunktion (gemäß Satz 9) automatisch erfüllt. Aus diesen Gründen sind die letzten beiden Definitionen unkritisch für den Implementierungsvorschlag des nächsten Abschnitts.

Lemma 7. *Es sei $(H^{MC}, <)$ eine (wohl geordnete) Methoden-Cache-Historie mit den Transaktionen $\{(T_1, <_1), \ldots, (T_n, <_n)\}$. Weiter sei $RW((H^{MC}, <))$ die durch das strikte Zwei-Phasen-Sperrprotokoll erzeugte rw-Projektion und ts eine zugehörige Commit-Zeitstempelfunktion. Dann ist $(H^{MC}, <)$ rm-Z-geordnet bezüglich ts.*

Beweis. Es seien Operationen gegeben mit $w_i[x] < r_k^l[x] < m_j^{k,l} \wedge c_i \in H^{MC} \wedge c_j \in H^{MC}$. Da $RW((H^{MC}, <))$ kaskadenfrei ist, folgt $w_i[x] < c_i < r_k^l[x] < m_j^{k,l} < c_j$ und somit $ts(T_i) < ts(T_j)$. □

Nun ist das „Material“ für eine geeignet Aussage zur Serialisierbarkeit von Z-passenden Methoden-Cache-Historien beisammen. Da es um ein Zeitstempelprotokoll für einen m-Planer geht, wird vom rw-Planer wieder erwartet, dass er „Z-geordnet“ für die rw-Projektion einer Methoden-Cache-Historie zusichert.

Satz 11. *Eine bezüglich einer Z-Funktion ts Z-passende, rm-Z-geordnete, irreflexive (wohl geordnete) Methoden-Cache-Historie $(H^{MC}, <)$ ist MC-serialisierbar, wenn $RW((H^{MC}, <))$ Z-geordnet ist bezüglich ts.*

Beweis. Es sei $(H^{MC}, <)$ eine bezüglich der Zeitstempelfunktion ts Z-passende, rm-Z-geordnete, irreflexive (wohl geordnete) Methoden-Cache-Historie. Weiter sei $RW((H^{MC}, <))$ auch Z-geordnet bezüglich ts.

Angenommen der Serialisierbarkeitsgraph $MCSG$ zu $(H^{MC}, <)$ wäre zyklisch. Ein Zyklus in $MCSG$ hat mindestens die Länge zwei, denn für alle Disjunktionsklauseln aus Definition 13 außer dem Fall $r_k^l[x] < w_i[x] < m_i^{k,l}$ gilt $i \neq j$ für eine entsprechende Kante $T_i \rightarrow T_j \in MCSG$. Der Fall $r_k^l[x] < w_i[x] < m_i^{k,l}$ ist aber ausgeschlossen, weil $(H^{MC}, <)$ irreflexiv ist. Ein Zyklus in $MCSG$ enthält (mindestens) eine Rückwärtskante $T_i \rightarrow T_j$ mit $ts(T_j) < ts(T_i)$. Andernfalls hätte man einen Zyklus $T_k \rightarrow \ldots \rightarrow T_k$ mit $ts(T_k) < ts(T_k)$.

Wie die folgenden Überlegungen zeigen, gilt für eine Rückwärtskante $T_i \rightarrow T_j$ aus $MCSG$ weiter: Man hat Operationen $r_k^l[x] < w_j[x]$ und $m_i^{k,l}$ mit $ts(T_j) < ts(T_i)$ aus der zweiten Disjunktionsklausel von Definition 13. Kanten aus dem Teilgraph $SG \subseteq MCSG$ können keine Rückwärtskanten sein, weil $RW((H^{MC}, <))$ nach Voraussetzung Z-geordnet ist. Eine Kante aus der dritten Disjunktionsklausel von Definition 13 ist keine Rückwärtskante, denn entsprechende Operationen $w_i[x] < r_k^l[x] < m_j^{k,l}$ mit $ts(T_j) < ts(T_i)$ widersprechen der Voraussetzung, dass $(H^{MC}, <)$ rm-Z-geordnet ist.

Sei Z ein fester Zyklus und T_k der Knoten aus Z mit dem kleinsten Zeitstempel. Dann gibt es eine Rückwärtskante $T_h \rightarrow T_k \in Z$ für ein T_h. Wäre $T_h \rightarrow T_k \in Z$ keine Rückwärtskante, so wäre der Zeitstempel von T_k nicht minimal bezüglich Z.

Es sei weiter $T_j \rightarrow \ldots \rightarrow T_k$ der in Z längst mögliche (azyklische) Pfad von Rückwärtskanten, der zu T_k führt. Dann gibt es eine Kante $T_i \rightarrow T_j$, die keine Rückwärtskante ist. Andernfalls würde Z nur aus Rückwärtskanten bestehen, und man erhielte $T_k \rightarrow \ldots \rightarrow T_k$ mit $ts(T_k) < ts(T_k)$.

Da $T_i \rightarrow T_j$ keine Rückwärtskante ist, hat man $ts(T_i) < ts(T_j)$ und somit sogar $ts(T_i) < ts_{fit}(T_j)$, weil $(H^{MC}, <)$ Z-passend ist. Da $T_j \rightarrow \ldots \rightarrow T_k$ ein Pfad aus Rückwärtskanten ist, folgt durch induktive Anwendung von Definition 26 die Aussage $ts_{fit}(T_j) \leq ts(T_k)$. Somit erhält man $ts(T_i) < ts(T_k)$ im Widerspruch zur Minimalitätsannahme für den Zeitstempel von T_k im Zyklus Z. □

Wie schneidet das passungsbasierte Zeitstempelprotokoll im Vergleich zum aktualitätsbasierten Zeitstempelprotokoll aus Abschnitt 7.2 ab? Diese Frage beantwortet teilweise der nächste Satz: Jede Historie, die durch ein aktualitätsbasierten Zeitstempelprotokoll produziert wird, erfüllt, bei gleicher Zeitstempelfunktion, auch die Kriterien des passungsbasierten Zeitstempelprotokolls.

Satz 12. *Eine bezüglich einer Z-Funktion ts Z-geordnete, Z-aktuelle (wohl geordnete) Methoden-Cache-Historie $(H^{MC}, <)$ mit den Transaktionen $\{(T_1, <_1), \ldots, (T_n, <_n)\}$ ist rm-Z-geordnet, irreflexiv, Z-passend bezüglich ts.*

Beweis. Zunächst ist jede Z-geordnete Methoden-Cache-Historie rm-Z-geordnet – dies folgt sofort aus einem Vergleich der Definitionen 22 und 29. Man beachte hierzu, dass die Operationen $w_i[x]$ und $m_j^{k,l}$ aus Definition 29 zueinander in Konflikt stehen.

Jede Z-aktuelle ist irreflexiv – dies folgt ebenfalls aus den entsprechenden Definitionen. In Definition 23 setzte man dazu $T_i = T_j$ an.

Bei Z-aktuelle Historien gilt für ein beliebiges T_i mit $i \in \{1, \ldots, n\}$ weiter: $ts_{fit}(T_i) = ts(T_i)$. Angenommen dies wäre nicht so, dann hätte man nach Definition 26 zwei Ope-

rationen $m_i^{k,l} \not\parallel w_j[x]$ mit $T_j < T_i$ und weiter $r_k^l[x] < w_j[x]$ sowie $c_i, c_j \in H^{MC}$. Dies widerspricht direkt der Definition von „Z-aktuell“. Mit $ts_{fit}(T_i) = ts(T_i)$ ist „Z-passend“ aus Definition 27 trivialerweise erfüllt. □

Das passungsbasierte Zeitstempelprotokoll ist also mindestens so gut wie das aktualitätsbasierte Zeitstempelprotokoll. Das erstere Protokoll ist sogar besser, wie man am nächsten Beispiel erkennt.

Beispiel 23. *Die folgende Methoden-Cache-Historie ist für die Zeitstempelfunktion $ts(T_i) = i$ Z-passend, rm-Z-geordnet und irreflexiv, aber nicht Z-geordnet und auch nicht Z-aktuell:*

$$(H_2, <) = r_1^1[x] c_1 m_3^{1,1} w_2[x] c_2 c_3.$$

Es gilt $ts_{fit}(T_3) = 2$. Das Nachprüfen der angegebenen Eigenschaften sei dem Leser überlassen.

7.3.3 Implementierung

Die im Folgenden vorgestellte Implementierung für einen m-Planer ist, ähnlich wie bereits in Abschnitt 7.2.4, auf einen rw-Planer mit striktem Zwei-Phasen-Sperrprotokoll und eine daraus resultierende Commit-Zeitstempelfunktion ausgelegt.

Wegen der Komplexität des Verfahrens greift dieser Abschnitt auf eine Darstellung mittels Pseudocode zurück. Dabei wird zunächst gezeigt, wie der m-Planer die Eigenschaft „Z-passend“ ohne die Berücksichtigung von Speicherverwaltungsaspekten verwirklichen kann. Eine informelle Korrekheitsbetrachtung geht mit der Erläuterung des Verfahrens einher. Als zweiter Schritt folgt eine Speicher-effiziente Erweiterung des Ansatzes.

Die Integration in das Entwurfsrahmenwerk aus Kapitel 5 gestaltet sich ähnlich wie bereits bei den vorangegangenen beiden Protokollen und wird daher nur knapp skizziert.

Verfahren Abbildungen 16 und 16 zeigen den Java-Pseudocode für ein Verfahren das im m-Planer „Z-passend“ zusichert.

Der Pseudocode behandelt im Wesentlichen die vier relevanten Operationstypen $r_i^k[x]$, $w_i[x]$, $m_i^{k,l}$ und c_i, die ein m-Planer für eine Transaktion T_i berücksichtigen muss. Dies drückt sich in der Klasse `MScheduler` durch die Methoden `read()`, `write()`, `cachedMethodCall()` und `verify()` aus. `cachedMethodCall()` soll also m-Operationen verarbeiten. `verify()` dient zur Verifikation einer Transaktion T_i vor deren Abschluss. Ist die Verifikation erfolgreich, so reicht das System ein c_i an den rw-Planer weiter. Das Java-Schlüsselwort `synchronized` verdeutlicht, dass alle vier Methoden unteilbar operieren. Dies ist notwendig für die Korrektheit des Verfahrens.

Die Klasse `MScheduler` verwaltet mehrere Tabellen, die allesamt den Typ `MultiMap` besitzen. `MultiMap` implementiert Relationen im mathematischen Sinn. Im Gegensatz zu einer gewöhnlichen Hash-Tabelle können einem Schlüssel mittels der Methode `MultiMap.put()` *mehrere* Werte zugeordnet werden. `MultiMap.getMulti(x)` liefert die Liste der bisher zugeordneten Werte für einen Schlüssel `x`.[28]

[28] Die Schnittstelle zu `MultiMap` wurde bereits in Abschnitt 5.4 angesprochen.

```
1  // Eine Transaktion T_i wird repräsentiert durch eine Instanz dieser Klasse
2  class Transaction {
3    int ts  = ∞; // Zeitstempel ts(T_i) der Transaktion
4    int ts_fit= ∞; // Zeitstempel, der dem Funktionswert ts_fit(T_i) entspricht
5    List rl = ∅;  // Liste der von T_i durch Operationen r_i[x] gelesenen Datenelemente x
6    List wl = ∅;  // Liste der von T_i durch Operationen w_i[x] geschriebenen Datenelemente x
7    List ml = ∅;  // Liste der MethodCallId-Objekte die Methodenoperationen m_i^{k,l} zu T_i repräsentieren
8  }
9
10 class MethodCallId {
11   int tid, mid; // Transaktions- und Methodenidentifikator einer Methodenoperation m_i^{k,l}, entspricht (k,l)
12 }
13
14 class MScheduler {
15   int nextTs = 0;   // Nummer des nächsten zu vergebenden Zeitstempels ts
16   MultiMap V  = ∅;  // Ordnet Datenelementen x die Menge der Operationen m_i^{k,l} mit x ∈ d(m_i^{k,l}) zu
17   MultiMap V^{-1}= ∅; // Ordnet Operationen m_i^{k,l} die Menge der Datenelemente d(m_i^{k,l}) zu
18   MultiMap rt = ∅; // Ordnet Datenelementen x die Menge der Transaktionen T_i mit r_i^k[x] zu
19   MultiMap wt = ∅; // Ordnet Datenelementen x die Menge der Transaktionen T_i mit w_i[x] zu
20   MultiMap mt = ∅; // Ordnet einem gecachten Methodenaufruf (identifiziert durch ein MethodCallId-Objekt) die
21                    // Transaktionen zu, die das entsprechende Methodenresultat aus dem Cache gelesen haben;
22                    // für m_i^{k,l} ist dann ein Eintrag entsprechend ((k,l),{...,T_i,...}) in mt
23
24   // Lese Datenelement x in Transaktion t (eventuell als Teil des durch m repräsentierten Dienstmethodenaufrufs)
25   synchronized void read(Transaction t, Object x, MethodCallId m) {
26     // Behandle entstehende rw-Konflikte in Bezug auf „Z-passend“
27     for each s ∈ wt.getMulti(x)
28       if (s.ts < ∞ && s.ts > t.ts_fit) { abort(t); return; }
29     // Aktualisiere Tabellen
30     t.rl.add(x);
31     rt.put(x, t);
32     if (m != null) { V.put(x, m); V^{-1}.put(m, x); }
33   }
34
35   // Schreibe Datenelement x in Transaktion t
36   synchronized void write(Transaction t, Object x) {
37     // Behandle entstehende ww- und wr-Konflikte in Bezug auf „Z-passend“
38     for each s ∈ wt.getMulti(x) ∪ rt.getMulti(x)
39       if (s.ts < ∞ && s.ts > t.ts_fit) { abort(t); return; }
40     // Behandle entstehende wm-Konflikte in Bezug auf „Z-passend“
41     for each m ∈ V.getMulti(x)
42       for each s ∈ mt.getMulti(m)
43         if (s.ts < ∞ && s.ts > t.ts_fit) { abort(t); return; }
44     // Aktualisiere Tabellen
45     t.wl.add(x);
46     wt.put(x, t);
47   }
```

Abbildung 16: Pseudocode zur Implementierung von „Z-passend“ im m-Planer ohne Berücksichtigung von Speicherverwaltungsaspekten (erster Teil)

`MScheduler` ist auch für die Vergabe von Zeitstempeln während der Verifikation einer Transaktion verantwortlich. Hierfür existiert der Zähler `MScheduler.nextTs`, der den jeweils nächsten zu vergebenden Zeitstempel bestimmt.

```
48    // Lese ein gecachtes Methodenresultat in Transaktion t,
49    // das durch den mittels m identifizierten Methodenaufruf berechnet wurde
50    synchronized void cachedMethodCall(Transaction t, MethodCallId m) {
51      // Stelle sicher, dass m noch in V (und V^-1) verzeichnet ist
52      if (!V^-1.containsKey(m)) { abort(t); return; }
53      // Behandle entstehende mw-Konflikte in Bezug auf „Z-passend“
54      for each x ∈ V^-1.getMulti(m)
55        for each s ∈ wt.getMulti(x)
56          if (s.ts < ∞ && s.ts > t.ts_fit) { abort(t); return; }
57      // Aktualisiere Tabellen
58      t.ml.add(m);
59      mt.put(m, t);
60    }
61
62    // Verifiziere Transaktion t vor dem Abschluss
63    synchronized verify(Transaction t) {
64      t.ts = nextTs++; // Setze Zeitstempel ts für t
65      if (t.ts_fit == ∞) t.ts_fit = t.ts; // Passe ts_fit an, falls nötig
66      // Aktualisiere Zeitstempel ts_fit für aktive Transaktionen
67      for each x ∈ t.wl
68        for each m ∈ V.getMulti(x)
69          for each s ∈ mt.getMulti(m)
70            { if (t.ts_fit < s.ts_fit) updateTs_fit(s,t); }
71      // Breche Transaktionen ab, die „Z-passend“ nicht mehr erfüllen wegen des Zeitstempels von t
72      for each x ∈ t.rl // rw-Konflikte
73        for s ∈ wt.getMulti(x)
74          { if (s.ts == ∞ && t.ts > s.ts_fit) abort(s); }
75      for each x ∈ t.wl // ww-Konflikte und wr-Konflikte
76        for s ∈ wt.getMulti(x) ∪ rt.getMulti(x)
77          { if (s.ts == ∞ && t.ts > s.ts_fit) abort(s); }
78      for each m ∈ t.ml // mw-Konflikte
79        for each x ∈ V^-1.getMulti(m)
80          for each s ∈ wt.getMulti(x)
81            { if (s.ts == ∞ && t.ts > s.ts_fit) abort(s); }
82    }
83
84    // Aktualisiere den Zeitstempel ts_fit für Transaktion s und übernimm ihn aus der Transaktion t
85    void updateTs_fit(Transaction s, Transaction t) {
86      s.ts_fit = t.ts_fit;
87    }
88
89    // Brich Transaktion t ab
90    void abort(Transaction t) {
91      ...
92    }
93  }
```

Abbildung 17: Fortsetzung von Abbildung 16: Pseudocode zur Implementierung von „Z-passend“ im m-Planer ohne Berücksichtigung von Speicherverwaltungsaspekten

Transaktionen entsprechen Instanzen der Klasse `Transaction` mit einer Reihe von transaktionsspezifischen Attributen. Der Zeitstempel einer Transaktion wird durch `Transaction.ts` repräsentiert. Da dieser Wert aber erst bei der Verifikation der Transaktion (also in `verify()`) endgültig festgelegt wird, setzt ihn das System initial auf un-

endlich (∞). Der Wert unendlich sagt aus, dass der noch zu bestimmende Zeitstempel größer sein wird als der von allen Transaktionen, welche bereits abgeschlossen wurden.

Das Attribut $\texttt{ts}_{fit}$ entspricht für eine Transaktion T_i dem Funktionswert $ts_{fit}(T_i)$ aus Definition 26. Der Attributwert ist initial ebenfalls unendlich, kann sich aber durch andere Transaktionen in deren Verifikationsphase verändern. Die Listen `Transaction.rl` und `Transaction.wl` nehmen jeweils die Datenelemente auf, welche von der betreffenden Transaktion durch `read()` bzw. `write()` gelesen wurden. `Transaction.ml` speichert hingegen alle m-Operationen einer Transaktion.

Die Klasse `MethodCallId` ist eine Datenstruktur, welche Methodenaufrufe anhand eines Transaktionsidenfikators und eines Methodenidentifikators identifiziert. Für eine Operation $m_i^{k,l}$ entspricht eine `MethodCallId`-Instanz also dem Tupel (k, l).[29]

Zu den Tabellen im m-Planer: `MScheduler.V` hat die gleiche Funktion wie bereits in den Verfahren aus den Abschnitten 7.1.3 und 7.2.4. Darüber hinaus muss der m-Planer auch umgekehrt von einer verarbeiteten Operationen $m_i^{k,l}$ auf deren verwendete Datenelemente $d(m_i^{k,l}) = \{x \mid \exists r_k^l[x] \in H^{MC}\}$ schließen können (siehe hierzu Definition 3). Dies leistet $\texttt{MScheduler.V}^{-1}$ sozusagen als „inverse Relation“ zu `MScheduler.V`. Wie schon in Abbildung 16 erklärt, speichern `MScheduler.rt` und `MScheduler.wt` zu einem Datenelement x jene Transaktionen in Form von `Transaction`-Objekten, welche x gelesen bzw. geschrieben haben. Schließlich ordnet die Tabelle `MScheduler.mt` einem `MethodCacheId`-Objekt die Transaktionen zu, in denen das entsprechende gecachte Methodenresultat gelesen wurde (siehe auch Kommentare in Abbildung 16).

Das eigentliche Verfahren zur Absicherung von „Z-passend“ ist in den Methoden der Klasse `MScheduler` kodiert. Prinzipiell reagieren die Methoden hierbei lediglich gemäß Definition 27 auf Veränderungen im Serialisierbarkeitsgraphen, die sich durch neue Operationen in der Historie bzw. durch die Zuordnung eines Zeitstempels während einer Verifikation ergeben. Wie auch sonst bei der Transaktionsverwaltung üblich, muss der m-Planer hierbei einen „verallgemeinerten“ Serialisierbarkeitsgraphen betrachten, bei dem auch Konflikte zwischen Transaktionen, die noch nicht abgeschlossen wurden, als Kanten einfließen. Die Hauptschwierigkeit des Verfahrens liegt darin, dass ein Zeitstempel erst sehr spät bekannt wird, und somit Kanten im Serialisierbarkeitsgraphen eventuell erst verzögert berücksichtigt werden können.

Zu den Methoden im Einzelnen:

- `read(t,x)` bestimmt in Zeile 27 alle Transaktionen, die mit einer Transaktion T_j (repräsentiert durch `t`) in Konflikt stehen, weil diese `x` zuvor geschrieben haben. Sei T_i (repräsentiert durch `s`) eine solche Transaktion.

  Weil die Zeitstempelordnung bezüglich T_i berücksichtigt werden soll, betrachtet das Verfahren nur solche Transaktionen T_i, die bereits verifiziert wurden und deren Zeitstempel somit feststeht. Konflikte zu aktiven Transaktionen werden später in `verify()` sozusagen „rückwirkend“ untersucht, sobald T_j oder eine entsprechende aktive Transaktion abschließt. Der Teilausdruck $\texttt{s.ts} < \infty$ in der Abfrage

[29] In Bezug auf die Suche von Schlüsseln in `MultiMap`-Tabellen sind zwei `MethodCallId`-Instanzen $\texttt{m}_1$ und $\texttt{m}_2$ gleich, wenn $\texttt{m}_1\texttt{.tid == m}_2\texttt{.tid}$ und $\texttt{m}_1\texttt{.mid == m}_2\texttt{.mid}$ gilt.

aus Zeile 28 garantiert, dass T_i (repräsentiert durch `s`) tatsächlich verifiziert bzw. abgeschlossen wurde.

Ein wichtige Invariante des Verfahren ist, dass `t.ts`$_{fit}$ stets dem (korrekten) Wert $ts_{fit}(T_j)$ entspricht. Im Fall $ts_{fit}(T_j) = ts(T_j)$ ist `t.ts`$_{fit}$ `==` ∞, weil T_j natürlich noch nicht verifiziert wurde. Die Bedingung zur Kante $T_i \rightarrow T_j$ aus Definition 27 ist also erfüllt.

Wenn sowohl `s.ts` als auch `t.ts`$_{fit}$ kleiner als unendlich sind, kann die Bedingung aus Definition 27 verletzt werden, und genau dies prüft die `if`-Klausel in Zeile 28 durch den Ausdruck `s.ts > t.ts`$_{fit}$. T_i muss dann abgebrochen werden. (Da T_j in diesem Fall bereits abgeschlossen wurde, bildet es keine Alternative für einen Transaktionsabbruch.) Man beachte auch, dass der Zeitstempel von T_j sicher größer sein wird als der von T_i, da T_j (wenn überhaupt) erst in der Zukunft abschließen wird.

Sind `s.ts` und `t.ts` beide unendlich, kann noch keine Aussage darüber getroffen werden, ob die Kante $T_i \rightarrow T_j$ die Bedingung zu „Z-passend“ aus Definition 27 verletzt, denn je nach dem, ob T_i oder T_j zuerst abschließt, könnte sich später $ts(T_i) < ts(T_j)$ bzw. $ts(T_j) < ts(T_i)$ herausstellen. Deshalb muss die Kante $T_i \rightarrow T_j$ nach Bekanntwerden eines der beiden Zeistempel, also bei der Verifikation von T_i bzw. T_j, nochmals betrachtet werden.

Abschließend gilt es noch, die Tabellen und Listen aus `t` bzw. aus dem m-Planer entsprechend ihrer bereits behandelten Funktion zu aktualisieren.

- Für die Zeilen 38 und 39 in der Methode `write()` gelten ähnlich Überlegung wie bei `read()`. Neben wr- und ww-Konflikten muss die Methode aber auch Konflikte zu m-Operationen untersuchen, denn auch durch solche Konflikte können Kanten entstehen, die die Bedingung aus Definition 27 verletzen. Zeile 41 bestimmt deshalb über `MScheduler.V` alle gecachten Methodenaufrufe, bei denen `x` gelesen wurde. In der nächsten Zeile findet das System Transaktionen (repräsentiert durch `s`), welche ein entsprechendes gecachtes Methodenresultat gelesen haben. Für die Frage, ob `t` wegen `s` abzubrechen ist, gelten die gleichen Überlegungen wie bei der Methode `read()`.

- Die Methode `cachedMethodCall()` behandelt eine m-Operation $m_i^{k,l}$ (repräsentiert durch `m`) für eine Transaktion (repräsentiert durch `t`) bezüglich „Z-passend“. Um Konflikte zwischen der m-Operation und Schreiboperationen anderer Transaktionen zu finden, bestimmt die Methode in Zeile 54 zunächst alle Datenelemente $d(m_i^{k,l})$ mit Hilfe der Tabelle `V`$^{-1}$. Die nachfolgende Zeile iteriert über Transaktionen, welche derartige Datenelemente geschrieben haben. Daraufhin kann die Methode prüfen, ob `t` wegen eines entsprechenden Konflikts abzubrechen ist. Die detaillierten Überlegungen für die Prüfung gestalten sich analog zur Methode `read()`.

- `verify()` weist einer Transaktion, die abgeschlossen werden soll, einen Zeitstempel zu und stellt „Z-passend“ für sie sicher. Die vorliegende Variante der Methode

kann eine Transaktion T_i *immer* verifizieren, muss dazu aber eventuell andere aktive Transaktionen abbrechen. Prinzipiell gibt es bei der Konstruktion von `verify()` den Freiheitsgrad, anstatt anderer aktiver Transaktion auch T_i selbst abzubrechen (und zwar in den Zeilen 74, 77 und 81). Dies rechtfertigt den angewendeten Begriff „Verifikation".

Im Einzelnen geht die Methode `verify()` wie folgt vor: Zunächst erhält die zu verifizierende Transaktion T_i (repräsentiert durch `t`) einen Zeitstempel (Zeile 64). Dieser ist wegen des `synchronized`-Modifikators aus Zeile 63 eindeutig. Falls $\texttt{t.ts}_{fit}$ bisher noch nicht verändert wurde, wird es auf den Wert `t.ts` gesetzt. Der Wert entspricht bezüglich Definition 26 dem Funktionsergebnis $ts_{fit}(T_i) = ts(T_i)$.

Mit dem Bekanntwerden des Zeitstempels $ts(T_i)$ muss der Serialisierbarkeitsgraph auf etwaige, neue Rückwärtskanten von T_i zu anderen Transaktionen, etwa T_j untersucht werden. Wegen einer solchen Rückwärtskante muss der Z-passende Zeitstempel einer Transaktion T_j dann gegebenenfalls aktualisiert werden. Dies ist der Fall, wenn der aktuelle, Z-passende Zeitstempel von T_j größer ist als $\texttt{t.ts}_{fit}$ (Zeile 70). Damit folgt die Implementierung dem Berechungsaspekt $ts_{fit}(T_j) = \min(\ldots \cup \{ts_{fit}(T_i) \mid \ldots\})$ aus Definition 26. Der eigentliche Aktualisierungsvorgang wurde in die Methode $\texttt{updateTs}_{fit}\texttt{()}$ ausgelagert, denn im nächsten Abschnitt soll er in einer Unterklasse zu Zwecken der Speicherverwaltung erweitert werden.

Nachdem die Z-passenden Zeitstempel aller aktiven Transaktionen geeignet angepasst wurden, bestimmt die Methode in den Zeilen 72 bis 81 solche aktiven Transaktionen, die „Z-passend" nicht mehr erfüllen, und bricht sie ab. Die entsprechenden Konflikte werden dabei auf ähnliche Weise wie in den Methoden `read()`, `write()` und `cachedMethodCall()` untersucht, allerdings sind nun die Rollen von `s` und `t` vertauscht. Es müssen alle Konfliktkanten behandelt werden, die von aktiven Transaktionen T_j auf die verifizierte Transaktion T_i verweisen – dies betrifft rw-, ww-, wr, und mw-Konflikte zwischen T_i und T_j. wm-Konflikte darf man ignorieren, da sie Rückwärtskanten im Serialisierbarkeitsgraphen erzeugen und bereits implizit über die Aktualisierung der Z-passenden Zeitstempel abgehandelt wurden.

Die folgende Überlegung zeigt, dass ein wm-Konflikt zwischen T_i und einem T_j tatsächlich zu einer Rückwärtskante führt. Seien etwa $r_k^l[x] < w_i[x]$ und $m_j^{k,l}$ entsprechende Operationen. Da T_i gerade abschließt, aber T_j noch aktiv ist (und, wenn überhaupt, nach T_i abschließen wird), erhält man die Kante $T_j \to T_i$ mit $ts(T_i) < ts(T_j)$ (also eine Rückwärtskante). Die Alternative $w_i[x] < r_k^l[x] < m_j^{k,l} < c_i$ kann nicht eintreten, denn es müsste $w_i[x] < c_i < r_k^l[x]$ gelten, da der rw-Planer „rücksetzbar" (ja sogar „strikt") garantiert.

Umgekehrt erzeugt ein mw-Konflikt zwischen T_i und einem T_j sicher *keine* Rückwärtskante. Die Operationen $r_k^l[x] < w_j[x]$ und $m_i^{k,l}$ liefern die Kante $T_i \to T_j$, und wie im vorigen Paragraphen folgt wieder $ts(T_i) < ts(T_j)$. $w_j[x] < r_k^l[x] < m_i^{k,l}$ ist ausgeschlossen, weil sonst $w_j[x] < c_j < r_k^l[x]$ folgen würde, und T_j dann nicht

mehr aktive wäre. (Man bedenke wieder, dass der rw-Planer „rücksetzbar“ zusichert.)

Das vorgestellte passungsbasierte Verfahren kann ähnlich wie für die Protokolle aus Abschnitt 7.1 und 7.2 in den Entwurf aus Kapitel 5 eingebettet werden. Dazu bietet es sich an, für den m-Planer geeignete Unterklassen von `LockMScheduler` und `LockMTransaction` aus Abbildung 13 zu bilden. Das Attribut `LockMTransaction.cacheHitCall` kann dazu dienen, die Liste `Transaction.ml` aus Abbildung 16 umzusetzen. Die Tabellen `V` und $\texttt{V}^{-1}$ aus Abbildung 13 entsprechen den Attributen `MScheduler.dataElementsToCallIds` bzw. `MScheduler.callIdsToDataElements` aus Abbildung 10.

Im Gegensatz zum Implementierungsvorschlag aus Abbildung 10 sollten durch `write()` invalidierte Einträge `m` für Methodenresultate aber nicht unmittelbar aus `V` bzw. $\texttt{V}^{-1}$ entfernt werden. Dies hätte nämlich zur Folge, dass Aufrufe an `cachedMethodCall()`, die sich auf einen entsprechend invalidierten Eintrag beziehen, in Zeile 52 scheitern würden. Die zugehörige Transaktion `t` würde abgebrochen und das Verfahren wäre damit ähnlich rigide wie das Sperrprotokoll aus Abschnitt 7.1.

Damit andererseits `V` und $\texttt{V}^{-1}$ aber nicht beliebig wachsen, sollten sie mit einer Verdrängungsstrategie arbeiten, die bereits invalidierten Einträgen die höchste Verdrängungspriorität einräumt. Als nachgeordnetes Kriterium bietet es sich an, LRU zur Bestimmung zu verdrängender Einträge zu verwenden.

Neben der Eigenschaft „Z-passend“ muss der m-Planer gemäß Satz 11 für die Serialisierbarkeit von Methoden-Cache-Historien auch „irreflexiv“ und „rm-Z-geordnet“ unterstützen. Gemäß Lemma 7 garantiert bereits die (verwendete) Commit-Zeitstempelfunktion und der kaskadenfreie (und nach Voraussetzung ja sogar strikte) rw-Planer „rm-Z-geordnet“.

„Irreflexiv“ soll Operationsfolgen der Art $r_k^l[x] < w_i[x] < m_i^{k,l} < c_i$ vermeiden. Im Kontext des Invalidierungsverfahrens aus Kapitel 5 invalidiert aber die Operation $w_i[x]$ nach der Ausführung beim Klienten unmittelbar jenen Eintrag im Methoden-Cache, der $m_i^{k,l}$ repräsentiert. $m_i^{k,l}$ kann somit in der obigen Operationsfolge gar nicht ausgeführt werden. Falls ein entsprechender Eintrag aus `V` bereits verdrängt wurde, könnte es sein, dass der Eintrag für $m_i^{k,l}$ im Methoden-Cache doch nicht durch $w_i[x]$ gelöscht wird. Dann scheitert aber die Ausführung von $m_i^{k,l}$ in Zeile 52 von Abbildung 17. Insgesamt garantiert dies „irreflexiv“.

Man beachte, dass die vorgestellte Protokollimplementierung die Arbeit des rw-Planers *nicht* bereits vorwegnimmt. Der m-Planer weist r- und w-Operationen nur dann durch Transaktionssabbrüche zurück, wenn diese die Eigenschaft „Z-passend“ gefährden. Ansonsten „beobachtet“ er r- und w-Operationen nur und erwartet vom rw-Planer die nötigen Garantien zum Einsatz der Commit-Zeitstempelfunktion aus Satz 9.

Speicherverwaltung Der Implementierungsvorschlag des vorigen Abschnitts garantiert zwar die Eigenschaft „Z-passend“, hat aber ein erhebliches Problem: Einträge aus den Tabellen `MScheduler.rt`, `MScheduler.wt`, `MScheduler.mt` des m-Planers werden *niemals* entfernt. Da durch neue Transaktionsoperationen immer wieder neue Einträge in

den Tabellen entstehen, würde ein m-Planer auf Basis des gegebenen Verfahrens früher oder später den gesamten verfügbaren Speicher aufbrauchen, und das System würde ausfallen. Man beachte, dass auch eine automatische Speicherbereinigung (Garbage Collection) daran nichts ändert, denn diese kann nur Objekte freigeben, die im System nicht mehr referenziert werden. Dies ist bei den Tabelleneinträgen im m-Planer aber nicht der Fall.

Die Speicherverwaltung für die Tabellen `MScheduler.V` und `MScheduler.V`$^{-1}$ ist unproblematisch – sie wurde ja schon im vorigen Abschnitt diskutiert.

Als nächstes soll nun die Frage untersucht werden, wie und wann der m-Planer Einträge aus den Tabellen `rt`, `wt`, `mt` löschen darf, denn dadurch kann man das Verfahren des vorigen Abschnitts um geeignete Speicherverwaltungsmaßnahmen erweitern. Man betrachte in diesem Zusammenhang nochmals die Abbildungen 16 und 17: Wozu werden die entsprechenden Tabelleneinträge dort eigentlich benötigt? Das Verfahren nutzt sie ausschließlich in den Zeilen 28, 29, 43, 56, 70, 74, 77 und 81, um Konflikte zwischen Transaktion festzustellen und falls nötig eine Transaktion abzubrechen. Der m-Planer bricht dabei eine Transaktion $\mathtt{t}_1$ höchstens dann ab, wenn der Zeitstempel einer bereits verifizierten oder gerade zu verifizierenden Transaktion $\mathtt{t}_2$ (mit $\mathtt{t}_2\mathtt{.ts} < \infty$) größer ist als $\mathtt{t}_1\mathtt{.ts}_{fit}$.

Der m-Planer darf also Tabelleneinträge für solche Transaktionen `t` löschen, deren Zeitstempel `t.ts` kleiner sind als der kleinste Zeitstempel $\mathtt{s.ts}_{fit}$ bezüglich aller Transaktionen `s`, die noch nicht (komplett) verifiziert wurden. Etwas präziser gilt:

$$\mathtt{t.ts} < \min\{\, \mathtt{s.ts}_{fit} \mid \mathtt{s.ts} == \infty \vee \text{„s wird gerade verifiziert“} \,\} \wedge \mathtt{t.ts} < \infty \Rightarrow$$
$$\text{„Die Tabelleneinträge von t dürfen beim } m\text{-Planer entfernt werden.“}$$

Man beachte auch, dass der Zeitstempel `tx.ts` einer Transaktion `tx`, die diese Bedingung einmal erfüllt, niemals wieder als Z-passender Zeitstempel $\mathtt{s.ts}_{fit}$ einer aktiven Transaktion `s` auftreten kann. Angenommen dies wäre nicht so: Zeile 70 ist die einzige Stelle, die $\mathtt{ts}_{fit}$ für Transaktionen verändert. Dabei gibt die Methode `update()`, den Zeitstempel einer Transaktion `t` weiter, die gerade verifiziert wird. Mithin trifft die obige Bedingung für `tx` mit $\mathtt{tx.ts} == \mathtt{t.ts}_{fit}$ erst gar nicht zu (Widerspruch).

Daraus kann man die folgende Vorgehensweise ableiten: Man merkt sich alle verifizierten Transaktionen nach ihren Zeitstempeln geordnet in einer Liste `verifiedTxList`. Neu verifizierte Transaktionen werden einfach am Ende von `verifiedTxList` eingetragen. Zusätzlich zählt man in jeder Transaktionen `t` über einen Zähler `t.refCount` mit, wieviele aktive Transaktionen `s` den Zeitstempel von `t` als Z-passenden Zeitstempel $\mathtt{s.ts}_{fit}$ verwenden. Hat der Zähler für die Transaktion `tx` am Kopf von `verifiedTxList` den Wert 0, so entfernt man `tx` aus `verifiedTxList` und gibt die zugehörigen Tabelleneinträge im m-Planer frei. Das neue Kopfelement von `verifiedTxList` ist mittels einer Schleife analog zu behandeln.

Man darf mit der Transaktion `tx` am Listenkopf so verfahren, weil nach Konstruktion von `verifiedTxList` der Zeitstempel von `tx` sicher kleiner ist als $h = \min\{\mathtt{s.ts}_{fit} \mid \mathtt{s.ts} == \infty \vee \text{„s wird gerade verifiziert“}\}$. Denn falls eine Transaktion `q` in `verifiedTxList` mit $\mathtt{q.ts} == h$ existiert, muss $\mathtt{q.refCount} > 0$ gelten. Da `tx`

```
1  // Speicher-verwaltende Erweiterung der Transaction-Klasse aus Abbildung 16
2  class MemoryManagingTransaction extends Transaction {
3    Transaction ts_fitTx = this; // Referenz auf das Transaktionsobjekt, von dem der Zeitstempel ts_fit
4                                 // übernommen ist
5                                 // Initial ist dies wegen this.ts_fit == ∞ das „this"-Transaktionsobjekt selbst
6    int refCount = 1;            // Zähler für die Anzahl der (relevanten) Referenzen von Transaktionsobjekten,
7                                 // die über ihr Attribut ts_fitTx auf die „this"-Transaktion verweisen.
8                                 // Initial ist der Wert 1 wegen this.ts_fitTx == this.
9  }
10
11 // Speicher-verwaltende Erweiterung der MScheduler-Klasse aus Abbildung 16 und 17
12 class MemoryManagingMScheduler extends MScheduler {
13   List verifiedTxList = ∅; // Eine Liste verifizierter Transaktionen, deren Einträge zum Erhalt der
14                            // Protokollkonsistenz (noch) nicht aus den Tabellen des m-Planers entfernt
15                            // werden dürfen
16                            // Die Transaktionen in der Liste sind nach dem Zeitstempel Transaction.ts
17                            // aufsteigend geordnet
18
19   // Verifiziere Transaktion t vor dem Abschluss (überschreibende Version mit Speicherverwaltungsaspekten)
20   synchronized verify(Transaction t) {
21     super.verify(t);             // Eigentliche Verifikation wie in der Oberklasse
22     t.ts_fitTx.refCount--;       // Die Referenz auf die Transaktion t.ts_fitTx wird zurückgenommen
23     verifiedTxList.addLast(t); // Reihe t ans Ende der Liste für verifizierte Transaktionen ein
24     for (int i = 0; i < verifiedTxList.size(); i++) {       // Iteriere von vorne über die
25       Transaction tx = (Transaction) verifiedTxList.get(i); // Liste und gib die Anfangsfolge
26       if (tx.refCount == 0) {      // von Transaktionen frei, deren Ref.zähler gleich 0 ist
27         verifiedTxList.remove(tx); // Lösche dazu die jeweilige Transaktion aus der Liste und
28         release(tx);               // entferne deren Tabelleneinträge im m-Planer
29       }
30       else break; // Breche ab bei der ersten Transaktion in der Liste, deren Referenzzähler ungleich 0 ist.
31     }
32   }
33   // Aktualisiere Zeitstempel s.ts_fit und Referenz s.ts_fitTx, übernimm dazu die Werte aus der Transaktion t
34   void updateTs_fit(Transaction s, Transaction t) {
35     super.updateTs_fit(s, t); // Übernimm Zeitstempel wie in der Oberklasse
36     s.ts_fitTx.refCount--;     // Entferne ursprüngliche Referenz und setze
37     s.ts_fitTx = t.ts_fitTx;   // die neue aus t ein, passe die dabei die Referenzzähler
38     s.ts_fitTx.refCount++;     // des alten und neu referenzierten Transaktionsobjekts an
39   }
40   // Brich Transaktion t ab
41   void abort(Transaction t) {
42     super.abort();         // Abbruch zunächst wie in der Oberklasse
43     t.ts_fitTx.refCount--; // Die Referenz auf die Transaktion t.ts_fitTx wird zurückgenommen und die
44     release(t);            // entsprechenden Tabelleneinträge werden aus dem m-Planer entfernt
45   }
46   // Gib Tabelleneinträge im m-Planer bezüglich der Transaktion t frei
47   void release(Transaction t) {
48     for each x ∈ t.rl rt.remove(x, t);
49     for each x ∈ t.wl wt.remove(x, t);
50     for each m ∈ t.ml mt.remove(m, t);
51   }
52 }
```

Abbildung 18: Pseudocode zur Speicherverwaltung bei der Implementierung von „Z-passend“ aus Abbildung 16 und 17

aber die älteste im System referenzierte Transaktion ist, muss die minimale Transaktion q jünger sein und in `verifiedTxList` *hinter* `tx` liegen.

Abbildung 18 liefert den Pseudocode zu der erläuterten Idee. Der Speicherveraltungsmechanismus erweitert dazu die Klassen `Transaction` und `MScheduler` aus den Abbildungen 16 und 17. Die Details zur Vorgehensweise ergeben sich aus den Kommentaren hinter den Codezeilen.

Abschließend sollte man sich noch überlegen, wie große die Liste `verifiedTxList` im ungünstigsten Fall werden kann: Für eine abgeschlossene Transaktionen `t` ist irgendwann die obige Minimumsbedingungen erfüllt, denn eine Transaktion `s`, die `t` durch $\min \{\texttt{s.ts}_{fit} \mid \texttt{s.ts} == \infty \vee$ „`s` wird gerade verifiziert“ $\}$ in der Liste `verifiedTxList` „hält“ muss selbst irgendwann enden. Somit kann `t` an den Kopf der Liste wandern und schließlich „aufgeräumt“ werden. Die Liste `verifiedTxList` wächst also niemals monoton – es gibt aber auch keine feste Obergrenze für deren Länge.

Die theoretisch zu erwartende Länge für `verifiedTxList` ist sehr klein (etwa < 2). Dazu gibt Abschnitt 8.2 einen Hinweis. Ohnehin kann man für `verifiedTxList` auch eine feste Maximalgröße *vorgeben*. Droht `verifiedTxList` diese Größe zu überschreiten, so kann man das Kopfelement `tx` löschen, auch wenn der Referenzzähler von `tx` ungleich null ist. Es genügt dann all jene aktiven Transaktionen `s` abzubrechen, die auf `tx` mittels $\texttt{s.ts}_{fit}\texttt{Tx}$ verweisen (für $\texttt{ts}_{fit}\texttt{Tx}$ siehe Abbildung 18).

8 Resultierende Protokolleigenschaften

Dieses Kapitel diskutiert allgemeine Eigenschaften der vorgestellten Protokolle, die sich aus ihrer Struktur ergeben. Einerseits findet ein Vergleich mit Protokolltypen aus der Literatur statt (siehe hierzu auch Abschnitt 2). Andererseits wird die Qualität der Methoden-Cache-Protokolle anhand *vereinfachender analytischer Modelle* abgeschätzt und verglichen.

8.1 Bezug zum Stand der Forschung

In Bezug auf die Protokolltaxonomie aus [10] handelt es sich bei allen drei Protokollimplementierungen aus Kapitel 7 um detektierende Protokolle (Detection Based). Der Grund ist offensichtlich, dass die Protokolle auf teilweise ungültige Methodenresultate zugreifen können – dies bemerkt das System spätestens während der Validierungsphase, also am Ende einer Transaktion. Damit ist auch bereits die Einordnung der vorgestellten Protokolle in der zweiten Taxonomieebene von [10] gegeben (Deferred Until Commit).

Weiter in der Taxonimie absteigend geben die Protokolle gemäß Abschnitt 5.4 Invalidierungshinweise an einen Methoden-Cache noch während des Verlaufs einer invalidierenden Transaktion weiter (During Transaction). Allerdings könnten die Protokolle mit geringem Aufwand auch so verändert werden, dass die Invalidierungsnachrichten erst nach dem Abschluss der invalidierenden Transaktion erfolgen.

Es existiert keine Vorab-Neuberechnung von ungültig Methodenresulaten im Falle

einer Invalidierung – dies würde der Taxonomiemerkmal „Propagation“ aus [10] entsprechen. Da beim Server für diesen Fall nicht einmal die Argumente zur Neuberechnung des entsprechenden Methodenresultats vorliegen, würde dieser Ansatz auch keinen Sinn ergeben.

Alle drei Protokolle ermöglichen den frühen Abbruch einer Transaktion (Early Abort nach [14]), wenn sich herausstellen sollte, dass eine aktive Transaktion Protokollbedingungen verletzt. Entsprechende Prüfungen finden immer dann statt, wenn wegen eines Cache-Fehlschlags (Cache Miss) oder einer Schreibmethodenausführung sowieso der Applikationsserver kontaktiert wird.

Gemäß Abschnitt 5.4 sind alle hier vorgestellten Protokolle „faul reaktiv“ (Lazy Reactive) nach [14], weil Invalidierungsnachrichten nicht unmittelbar verschickt, sondern an andere, sowieso zu versendende Nachrichten angehängt werden (Piggy Backing). Wie bereits weiter oben erwähnt, sind die Protokolle aus Kapitel 7 optimistisch und rückwärtsvalidierend (Backward Validating) nach [16].

Die folgenden Paragraphen untersuchen die Verwandtschaft der drei vorgestellten Protokolle zu bekannten Serialisierbarkeitsprotokollen noch etwas genauer.

Die Implementierung des Sperrprotokolls sowie des aktualitätsbasiertes Zeitstempelprotokolls ähneln am meisten dem transaktionalen Cache-Protokoll OCC ([14, 1], siehe auch Abschnitt 2). Sowohl die beiden vorgestellten Protokolle als auch OCC verwenden im Wesentlichen die Rückwärtsvalidierung. Die Nutzung von „Piggy Backing“ für Invaldierungsnachrichten sowie „Early Aborts“ haben alle drei vorgestellten Protokolle mit OCC gemein.

Ansonsten unterscheidet sich die Implementierung des passungsbasierten Zeitstempelprotokolls aber deutlich von anderen transaktionalen Cache-Protokollen. Zunächst ist es ein „echtes“ Zeitstempelprotokoll, da die Implementierung (im Gegensatz zum aktualitätsbasiertes Zeitstempelprotokoll) tatsächlich Zeitstempel berechnet.

Der wichtigste Unterschied zu *allen* bisherigen, transaktionalen Cache-Protokollen ist jedoch die Tatsache, *dass es Transaktionen, die auf nicht aktuelle Cache-Inhalte zugreifen, nicht notwendig abbricht.* (Siehe hierzu auch Abschnitt 2.) Dies ermöglicht die Eigenschaft „Z-passend“ gemäß Definition 27.

Satz 12 und Beispiel 23 wiesen bereits darauf hin, dass das passungsbasierte Zeitstempelprotokoll eine größere Menge von Historien zulässt als das aktualitätsbasierte Zeitstempelprotokoll und somit voraussichtlich zu geringeren Transaktionsabruchraten führt. In Abschnitt 7.2.4 wurde klar, dass die vorgestellte Implementierung des aktualitätsbasierten Zeitstempelprotokolls ihrerseits eine Verbesserung der Sperrprotokollimplementierung repräsentiert. Der nächste Abschnitt bestätigt die resultierende Ordnung der drei Protokolle hinsichtlich Transaktionabbruchraten durch eine Wahrscheinlichkeitsanalyse.

Außerhalb des transaktionalen Cachings für Klient-Server-Datenbanken gibt es noch einige weitere Serialierbarkeitsprotokolle, die mit dem passungsbasierten Zeitstempelprotokoll verwandt sind. Zunächst bietet sich ein Vergleich mit einer optimistischen Variante des Mehr-Versionen-Zeitstempelprotokolls (MVTO, [27]) an.

Um dies zu verdeutlichen sei kurz die Funktionsweise des Mehr-Versionen-Zeitstempelprotokolls bezüglich der Bearbeitung von Leseoperationen zusammenge-

fasst (siehe [4] Abschnitt 5.3): „Eine Operation $r_i[x]$ bearbeitet ein entsprechender Planer sofort und übersetzt sie in eine Mehr-Versionen-Operation $r_i[x_k]$, wobei x_k von der Transaktion T_k mit dem größten Zeitstempel geschrieben wurde, so dass noch $ts(T_k) \leq ts(T_i)$ gilt."

Die Bearbeitung von $r_i[x]$ kann man mit der Bearbeitung einer Operation $m_i^{k,l}$ vergleichen, die ein m-Planer im Rahmen des passungsbasierten Zeitstempelprotokolls ausführt. Dazu nehme man an, dass vor $m_i^{k,l}$ bereits $r_k^l[x]$ erfolgt ist. Anders als beim MVTO hat hier bereits eine Übersetzung auf eine Version von x stattgefunden, auf die $m_i^{k,l}$ mittels des entsprechenden Cache-Treffers implizit zugegriffen hat. Es handelt sich um die Version bzw. den Wert von x, der von $r_k^l[x]$ gelesen wurde. Anstatt nun wie beim MVTO den Index k zu bestimmen, hat der m-Planer nur die Möglichkeit den Zeitstempel von T_i sozusagen anzupassen. Genau dies geschieht mit Hilfe der Berechnung des Z-passenden Zeitstempels $ts_{fit}(T_i)$.

Die Berechnung des Z-passenden Zeitstempels ähnelt dem Konzept eines *dynamischen Zeitstempels* aus [2]. Dort schlagen die Autoren vor, einen Zeitstempel nicht a priori, etwa beim Beginn einer Transakion, festzulegen, sondern erst bei Bedarf zuzuweisen. Man betrachte hierzu die rw-Historie $r_1[x]w_2[y]c_2r_1[y]$. Ein naives Zeitstempelverfahren, das den Zeitstempel zum Transaktionsbeginn vergibt, hier etwa in der Form von $ts(T_1) = 1, ts(T_2) = 2$, muss T_1 wegen $r_1[y]$ abbrechen. Dass dynamische Verfahren ordnet erst bei der Bearbeitung von $r_1[y]$ der Transaktion T_1 einen Zeitstempel zu mit $ts(T_1) > ts(T_2)$ und kann T_1 deshalb fortführen.

Ein Z-passender Zeitstempel $ts_{fit}(T_i)$ ist jedoch kein wirklich dynamischer Zeitstempel, weil das aktualitätsbasierte Zeitstempelprotokoll trotzdem auch den Zeitstempel $ts(T_i)$ weiter verwendet. Daher bestehen Unterschiede zu dem Vorschlag aus [2].[30]

8.2 Analytische Abschätzung von Protokolleigenschaften

Dieser Abschnitt liefert eine grobe analytische Abschätzung wichtiger Leistungseigenschaften der beschriebenen Protokolle. Dazu zählen die zu erwartende Cache-Trefferrate und die Abbruchwahrscheinlichkeit von Transaktionen durch den m-Planer. Die verwendeten analytischen Modelle sind stark vereinfachend und sollen ein tatsächliches Leistungsverhalten nur überschlagen.

Ein wichtiges Ziel ist es, die relativen Leistungsunterschiede der drei behandelten Protokolle herauszuarbeiten. Die Beiträge zu Deadlock-Wahrscheinlichkeiten bei Datenbanktransaktionen von Gray und Kollegen in [13] Seite 428-429 sowie [12] haben diesen

[30] Die Autoren von [2] verallgemeinern den Ansatz dynamischer Zeitstempel durch *Zeitstempelintervalle*. Die Intervalle legen Ober- und Untergrenzen fest, in der sich ein gedachter, eindeutiger Zeitstempel für eine Transaktion befinden muss. Konflikte zwischen aktiven Transaktionen verkleinern die Intervalle vom Initialwert $[-\infty, +\infty]$ ausgehend dynamisch. Dadurch werden partielle Ordnungen, die sich aus einem zu Grunde liegenden (virtuellen) Serialisierbarkeitsgraphen ergeben, konservativ abgeschätzt.

[22] berichtet von einer optimistischen und eine pessimistischen Implementierungsvariante des Zeitstempelintervall-Verahrens. Das Papier kombiniert den Ansatz mit einem Mehr-Versionen-Zeitstempelprotokoll. Sowohl die optimistische als auch die pessismistische Variante haben deutlich bessere Performanzeigenschaften und Transaktionsabbruchraten als ein gewöhnliches Mehr-Versionen-Zeitstempelprotokoll.

Abschnitt „inspiriert“. In [9] findet man Überlegungen zur Abbruchwahrscheinlichkeit eines klassischen, optimistischen Transaktionsprotokolls (gemäß [19]). Die dortigen Ergebnisse stehen in Analogie zu entsprechenden Formeln von weiter unten.

8.2.1 Cache-Trefferrate

p_r bezeichne die Wahrscheinlichkeit, dass ein Dienstmethodenaufruf auf den transaktionalen Ressourcen des Applikationsservers rein lesend ist und damit sein Resultat im Methode-Cache vorgehalten werden kann. g sei die maximale Größe des Methoden-Caches als die Anzahl vorhaltbarer Methodenresultate. $p_{Hit}(g)$ repräsentiert die von der Cache-Größe g abhängige Wahrscheinlichkeit eines Cache-Treffers für lesende Dienstmethodenaufrufe, *sofern keine schreibenden Dienstmethodenaufrufe im System auftreten.*

Da Klienten Klienten aber schreibende Dienstmethodenaufrufe produzieren und eingelagerte Methodenaufrufe invalidieren reduziert sich die Trefferrate $p_{Hit}(g)$ entsprechend. Vereinfachend wird unterstellt, dass die Wahrscheinlichkeit für einen schreibenden Methodenaufruf ein vorgehaltenes Methodenresultat zu invalidieren ebenfalls $p_{Hit}(g)$ beträgt.

Daraus ergibt sich eine Formel zur Abschätzung der *effektiven* Cache-Trefferwahrscheinlichkeit eines Methodenaufrufs für ein System in dem lesende Methodenaufrufe mit der Wahrscheinlichkeit p_r und schreibende Aufrufe mit der Wahrscheinlichkeit $1 - p_r$ auftreten:

$$p_{HitReal}(g) = p_r \cdot p_{Hit}(g) \cdot (1 - (1 - p_r) \cdot p_{Hit}(g)/g).$$

Der Minuend in der Formel repräsentiert die Tatsache, dass ein bestimmtes Resultat durch einen Schreibmethodenaufruf bereits vor einem potentiellen Cache-Treffer invalidiert wurde. Für realistische Cache-Größen (etwa $g \geq 1000$) ist dieser Ausdruck vernachlässigbar klein und man erhält $p_{HitReal}(g) \approx p_r \cdot p_{Hit}(g)$.

8.2.2 Abbruchwahrscheinlichkeit durch den m-Planer

Es sei tps die Anzahl der (klientenbasierten) Anwendungstransaktionen, die pro Sekunde beim Applikationsserver gestartet werden. Weiter sei a die mittlere Anzahl von Dienstmethodenaufrufen pro Anwendungstransaktion und t_a die mittlere Ausführungszeit für einen entsprechenden Aufruf. Dann gibt es beim Applikationsserver im Mittel txs konkurrierend ablaufende (aktive) Anwendungstransaktionen mit $txs = tps \cdot a \cdot t_a$.

Zunächst zur Implementierung des *Sperrprotokolls* aus Abschnitt 7.1.3: Eine aktive Transaktion wird vom m-Planer abgebrochen, wenn sie auf ein gecachtes Methodenresulat zugegriffen hat, dass noch vor dem beabsichtigten Commit-Zeitpunkt invalidiert wird. Beim geplanten Abschluss hat eine Transaktion T ca. $p_r \cdot a$ Lesemethodenaufrufe ausgeführt, die jeweils Cache-Treffer mit der Wahrscheinlichkeit $p_{Hit}(g)$ verursachen. T erzeugt also ungefähr $p_r \cdot a \cdot p_{Hit}(g)$ Cache-Treffer. Mit $g \gg a$ darf man annehmen, dass diese Treffer alle paarweise verschieden sind.

Die übrigen aktiven Transaktionen haben beim Beenden von T ca. $(txs-1)\cdot(1-p_r)\cdot a$ Schreibmethodenaufrufe durchgeführt, denn während der gesamten Operationsphase von T sind etwa $2\cdot(tx-1)$ *verschiedene* Transaktionen aktiv, die sich im Mittel zur Hälfte mit der Operationsphase von T überschneiden. Es wird vereinfachend unterstellt, dass die $(txs-1)\cdot(1-p_r)\cdot a$ Schreibmethodenaufrufe paarweise verschieden sind.

Jeder der $(txs-1)\cdot(1-p_r)\cdot a$ Schreibmethodenaufrufe invalidiert ungefähr mit der Wahrscheinlichkeit $p_{Hit}(g)/g$ einen bestimmten Cache-Treffer aus T. Insgesamt ergibt sich die Abbruchwahrscheinlichkeit für T durch den m-Planer zu:

$$p_{LockAbort} = 1-(1-p_{Hit}(g)/g)^{p_r\cdot a\cdot p_{Hit}(g)\cdot(txs-1)\cdot(1-p_r)\cdot a} \approx$$

$$1-(1-p_r\cdot a\cdot p_{Hit}(g)^2/g)^{(txs-1)\cdot(1-p_r)\cdot a}.$$

Weil die Implementierung des *aktualitätsbasierten Zeitstempelprotokolls* aus Abschnitt 7.2.4 zu der Implementierung des Sperrprotokolls sehr ähnlich ist, sind auch die Überlegungen zur Abbruchwahrscheinlichkeit fast analog. Als wichtigster Unterschied bricht das aktualitätsbasierte Zeitstempelprotokoll (im Wesentlichen) die Transaktion T nur ab, wenn ein entsprechender invalidierender Schreibmethodenaufruf von einer bereits abgeschlossenen Transaktion ausgeht, die aber beim Start von T noch aktiv war. Beim Beenden von T kann man erwarten, dass die Hälfte der oben erwähnten $2\cdot(txs-1)$ Transaktionen bereits abgeschlossen sind. Dadurch reduziert sich im Vergleich zum Sperrprotokoll die mittlere Anzahl der Schreiboperationen, die zum Abbruch von T führen können, um die Hälfte, also $(txs-1)\cdot(1-p_r)\cdot a/2$. Die geänderte Abbruchwahrscheinlichkeit für T durch den m-Planer beträgt somit:

$$p_{CurrTStampAbort} \approx 1-(1-p_r\cdot a\cdot p_{Hit}(g)^2/g)^{(txs-1)\cdot(1-p_r)\cdot a/2}.$$

Für die Implementierung des passungbasierten Zeitstempelprotokolls aus Abschnitt 7.3.3 betrachtet man zunächst die Wahrscheinlichkeit einer Rückwärtskante für eine Transaktion T_i zu einer anderen, konkurrierend laufenden Transaktion. Es gelten wieder ähnliche Überlegungen wie bei der Berechnung von $p_{LockAbort}$, denn T_i hat am Ende ca. $p_{Hit}(g)\cdot p_r\cdot a$ Cache-Treffer, und die anderen Transaktionen erzeugten bis dahin ca. $(txs-1)\cdot(1-p_r)\cdot a$ Schreibmethodenaufrufe. Für eine Rückwärtskante darf man eine Operationsfolge der Art $r_k^l[x] < m_i^{k,l} < w_j[x]$ für eine zu T_i konkurrierende Transakion T_j erwarten. $r_k^l[x] < w_j[x] < m_i^{k,l}$ ist zwar möglich, aber extrem unwahrscheinlich, da die Ausführung von $w_j[x]$ unmittelbar darauf zu einer Invalidierung des Methoden-Cache-Eintrags führt, der $m_i^{k,l}$ entspricht. Daraus folgt für die Wahrscheinlichkeit $p_{Backward}$ einer Rückwärtskante der Transaktion T_i gerade $p_{Backward} = p_{CurrTStampAbort}$, denn die Hälfte der $(txs-1)\cdot(1-p_r)\cdot a$ Schreibmethodenaufrufe ist (wiederum) auszuschließen.

Als nächstes folgen Überlegungen zur mittleren Länge eines Rückwärtspfades, der bei T_i beginnt. Mit $p_{Backward}$ hat ein Rückwärtspfad der Länge eins die Wahrscheinlichkeit $p_{Backward}\cdot(1-p_{Backward})$, da die von T_i mittels der Rückwärtskante erreichte Transaktion selbst keine Rückwärtskante besitzen soll. Ein Rückwärtspfad der Länge

zwei hat entsprechend die Wahrscheinlichkeit $p_{Backward}^2 \cdot (1 - p_{Backward})$ usw. Daraus ergibt sich eine zu erwartende Rückwärtspfadlänge von

$$l = \lim_{n \to \infty} \sum_{i=0}^{n} i \cdot p_{Backward}^i \cdot (1 - p_{Backward}).$$

Für kleine Werte von $p_{Backward}$, etwa für $p_{Backward} \leq 0,1$, ist der Wert von l nur wenig größer als $p_{Backward}$ selbst. Da $p_{Backward}$ erwartungsgemäß niedriger als $0,1$ ausfallen wird, schätzt man $l \approx p_{Backward}$ ab.

Wegen der ca. $txs - 1$ Transaktionen, die vor T_i abschließen, aber noch während der Bearbeitung von T_i aktiv waren, ist zu erwarten, dass der Zeitstempel einer Transaktion T_j, die durch eine Rückwärtskante von T_i erreicht wird, ca. $txs/2$ Plätze vor dem (potentiellen) Zeitstempel von T_i liegt. Gemäß der Eigenschaft „Z-passend“ gibt es also im Schnitt $l \cdot (txs/2 - 1) \approx p_{Backward} \cdot (txs/2 - 1)$ Transaktionen, die keine „Vorwärtskante“ (also eine Kante, die keine Rückwärtskante ist) zu T_i besitzen dürfen. Ansonsten wird eine der beiden Transaktionen (entweder T_i oder die betreffende konkurrierende Transaktion) abgebrochen.

Der nächste Schritt befasst sich damit, die Wahrscheinlichkeit einer Vorwärtskante von einer festen Transaktion T_j nach T_i zu bestimmen. Dazu nimmt man an, dass es s verschiedene Datenelemente gibt, bei denen Konflikte in Bezug auf die Zugriffsoperationen der Ressourcenverwalter entstehen können. s kann etwa die Anzahl der physischen Seiten oder der Tabellenzeilen einer relationalen Datenbank repräsentieren.

Die genaue Bestimmung der Wahrscheinlichkeit einer Vorwärtskante von T_j nach T_i ist sehr aufwändig. Als stark vereinfachende Abschätzung wird stattdessen die Wahrscheinlichkeit, *dass T_i und T_j auf ein gemeinsames Datenelement bzw. ein Methodenresultat zugreifen*, eingesetzt. Die Abschätzung ist angemessen, denn eine Vorwärtskante impliziert das letztere Ereignis.

Zunächst zur Wahrscheinlichkeit, dass *zwei Operationen* auf das gleiche Datenelement (bzw. Methodenresultat) zugreifen. Es ist sinnvoll dabei Cache-Treffer und Cache-Fehlschläge zu unterscheiden, da Cache-Treffer erwartungsgemäß viel häufiger auftreten. Man erhält ungefähr die Gesamtwahrscheinlichkeit $p_{DoubleAccess} \approx p_{Hit}(g)^2/g + (1 - p_{Hit}(g))^2/s$.

Für die Operationen der ca. $p_{Backward} \cdot (txs/2 - 1)$ Transaktion, die keine Vorwärtskante zu T_i besitzen dürfen, kann man nun rw- (bzw. mw-), wr- und ww-Konflikte unterscheiden und erhält insgesamt:

$$p_{FitTStampAbort} \approx (1 - (1 - p_r \cdot a \cdot p_{DoubleAccess})^{p_{Backward} \cdot (txs/2-1) \cdot (1-p_r) \cdot a}) +$$
$$(1 - (1 - (1 - p_r) \cdot a \cdot p_{DoubleAccess})^{p_{Backward} \cdot (txs/2-1) \cdot p_r \cdot a}) +$$
$$(1 - (1 - (1 - p_r) \cdot a \cdot p_{DoubleAccess})^{p_{Backward} \cdot (txs/2-1) \cdot (1-p_r) \cdot a}).$$

Die zu erwartende Länge der Liste `verifiedTxList` für die speichereffiziente Implementierung des passungsbasierten Zeitstempelprotokolls aus Abschnitt 7.3.3 liegt übrigens in einem ähnlichen Bereich wie $p_{Backward}$ und ist daher sehr klein. Dies folgt

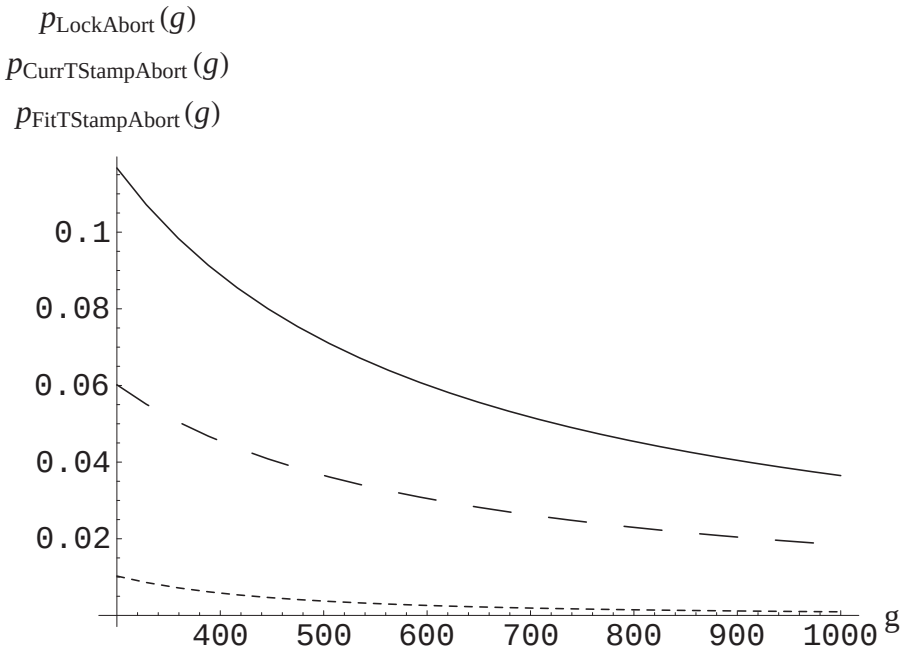

Abbildung 19: Beispielgraphen für die modellierten Abbruchwahrscheinlichkeiten in Abhängigkeit der Methoden-Cache-Größe g ($p_{LockAbort}(g)$ entspricht der durchgezogenen Linie, $p_{CurrTStampAbort}(g)$ entspricht der lang gestrichelten Linie, $p_{FitTStampAbort}(g)$ der kurz gestrichelten Linie)

aus den obigen Überlegungen, zur mittleren Länge l eines Rückwärtspfades, denn die Länge von `verifiedTxList` wird durch l dominiert. Da die Wahrscheinlichkeiten für Rückwärtspfade der Länge > 3 schnell abnehmen, ist auch die Varianz für die Länge von `verifiedTxList` vernachlässigbar.

Um ein Gefühl für die Aussage der obigen Formeln zu erhalten, zeigt Abbildung 19 drei Funktionsgraphen zur Abbruchwahrscheinlichkeit einer Transaktion für reale Systemparameter. Die Parameter sind aus Leistungsexperimenten mit einem entsprechenden Methoden-Cache-Prototyp abgeleitet. Die Anzahl der möglichen Lesemethodenresultate wurde dabei so klein gehalten, dass diese *vollständig* in den Methoden-Cache passen. Man hat $p_{Hit}(g) = 1$, $p_r = 0,5$, $a = 3,5$, $tps \approx 30/s$ und $t_a \approx 0,125s$.

Abbildung 19 verdeutlicht die unterschiedliche Qualität der Protokolle in Bezug auf das Abbruchverhalten. Das aktualitätsbasierte Zeitstempelprotokoll verhält sich diesbezüglich am besten.

Referenzen

[1] ADYA, A., R. GRUBER, B. LISKOV und U. MAHESHWARI: *Efficient optimistic concurrency control using loosely synchronized clocks*. In: *Proceedings of the 1995 ACM SIGMOD Conference on Management of Data*, S. 23–34. ACM Press, 1995.

[2] BAYER, R., K. ELHARDT, J. HEIGERT und A. REISER: *Dynamic Timestamp Allocation for Transactions in Database Systems*. In: *Distributed Data Bases*, S. 9–20. North-Holland Publishing Company, 1982.

[3] BERNSTEIN, P. und N. GOODMAN: *Multiversion concurrency control – theory and algorithms*. ACM Transactions on Database Systems, 8(4):465–483, 1983.

[4] BERNSTEIN, P., V. HADZILACOS und N. GOODMAN: *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[5] BERNSTEIN, P. und E. NEWCOMER: *Principles of transaction processing: for the systems professional*. Morgan Kaufmann, 1997.

[6] BREITBART, Y., D. GEORGAKOPOULOS, M. RUSINKIEWICZ und A. SILBERSCHATZ: *On rigorous Transaction Scheduling*. IEEE Transactions on Software Engineering, 17(9):954–960, 1991.

[7] CAREY, M. J., M. J. FRANKLIN, M. LIVNY und E. J. SHEKITA: *Data caching tradeoffs in client-server DBMS architectures*. In: *Proceedings of the 1991 ACM SIGMOD Conference on Management of Data*, S. 357–366. ACM Press, 1991.

[8] *Dynamic Proxy Classes*. `http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html`.

[9] FRANASZEK, P. A., J. T. ROBINSON und A. THOMASIAN: *Concurrency control for high contention environments*. ACM Transactions on Database Systems, 17(2):304–345, 1992.

[10] FRANKLIN, M. J., M. J. CAREY und M. LIVNY: *Transactional client-server cache consistency: alternatives and performance*. ACM Transactions on Database Systems, 22(3):315–363, 1997.

[11] GAMMA, E., R. HELM, R. JOHNSON und J. VLISSIDES: *Design Patterns*. Addison-Wesley, 1994.

[12] GRAY, J., P. HELLAND, P. O'NEIL und D. SHASHA: *The dangers of replication and a solution*. In: *Proceedings of the 1996 ACM SIGMOD Conference on Management of Data*, S. 173–182. ACM Press, 1996.

[13] GRAY, J. und A. REUTER: *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo (CA), USA, 1993.

[14] GRUBER, R.: *Optimism vs. Locking: A Study of Concurrency Control for Client-Server Object-Oriented Databases*. Technical Report MIT-LCS-TR-708, MIT, February 1997.

[15] HADZILACOS, T.: *Serialization graph algorithms for multiversion concurrency control*. In: *Proceedings of the 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, S. 135–141. ACM Press, 1988.

[16] HÄRDER, T.: *Observations on optimistic concurrency control schemes*. Information Systems, 9(2):111–120, 1984.

[17] *JBoss*, 2002. `http://www.jboss.org`.

[18] KIM, W., J. F. GARZA, N. BALLOU und D. WOELK: *Architecture of the ORION Next-Generation Database System*. IEEE Transactions on Knowledge and Data Engineering, 2(1):109–124, 1990.

[19] KUNG, H. T. und J. T. ROBINSON: *On Optimistic Methods for Concurrency Control*. ACM Transactions on Database Systems, 6(2):213–226, Juni 1981.

[20] MICROSOFT: *Microsoft .NET Framework (.NET)*. `http://msdn.microsoft.com/netframework`.

[21] MICROSOFT: *Microsoft .NET Remoting*. `http://msdn.microsoft.com/library`.

[22] NOE, J. D. und D. B. WAGNER: *Measured Performance of Time Interval Concurrency Control Techniques*. In: *Proceedings of the 13th International Conference on Very Large Data Bases*, S. 359–367. Morgan Kaufmann, 1987.

[23] ÖZSU, M. T., K. VORUGANTI und R. C. UNRAU: *An Asynchronous Avoidance-Based Cache Consistency Algorithm for Client Caching DBMSs*. In: GUPTA, A., O. SHMUELI und J. WIDOM (Hrsg.): *Proceedings of the 24th Conference on Very Large Databases (VLDB)*, S. 440–451. Morgan Kaufmann, 1998.

[24] PFEIFER, D.: *Eine Algebra für Cache-Modelle zum methodenbasierten Caching im Applikationsserver-Bereich*. Technical Report (in German) 2004-3, Universität Karlsruhe, 2004.

[25] PFEIFER, D. und H. JAKSCHITSCH: *Method-Based Caching in Multi-tiered Server Applications*. In: *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, S. 1312–1332. Springer-Verlag, November 2003.

[26] PFEIFER, D. und H. JAKSCHITSCH: *Method-Based Caching in Multi-Tiered Server Applications*. Technical Report 2003-11, Universität Karlsruhe, 2003.

[27] REED, D. P.: *Implementing atomic actions on decentralized data*. ACM Transactions on Database Systems, 1(1):3–23, 1983.

[28] STONEBRAKER, M.: *Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES*. IEEE Transactions on Software Engineering, 5(3):188–194, May 1979.

[29] SUN: *Enterprise Java Beans Specification (EJB), Version 2.1*. `http://java.sun.com/products/ejb/docs.html`.

[30] SUN: *Java Database Connectivity – JDBC*. `http://java.sun.com/products/jdbc`.

[31] SUN: *Java Remote Method Invodation (RMI)*. `http://incubator.apache.org/projects/altrmi/`.

[32] SUN: *Java Transaction API (JTA), Version 1.0*. `http://java.sun.com/products/jta/`.

[33] THE OBJECT MANAGEMENT GROUP (OMG): *CORBA Transaction Service Specification*. 2003. `http://www.omg.org/docs/formal/03-09-02.pdf`.

[34] THE OBJECT MANAGEMENT GROUP (OMG): *Common Object Request Broker Architecture: Core Specification (CORBA)*. 2004. `http://www.omg.org/docs/formal/04-03-01.pdf`.

[35] THE OPEN GROUP: *Distributed Transaction Processing*. `http://www.opengroup.org/public/pubs/catalog/tp.htm`.

[36] VOSSEN, G. W. G.: *Transactional Information Systems. Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

[37] WANG, Y. und L. A. ROWE: *Cache consistency and concurrency control in a client/-server DBMS architecture*. In: *Proceedings of the 1991 ACM SIGMOD Conference on Management of Data*, S. 367–376. ACM Press, 1991.

[38] WILKINSON, W. K. und M.-A. NEIMAT: *Maintaining Consistency of Client-Cached Data*. In: *Proceedings of the 16th Conference on Very Large Databases (VLDB)*, S. 122–133. Morgan Kaufmann, 1990.

[39] WU, K., P. FEI CHUANG und D. J. LILJA: *An active data-aware cache consistency protocol for highly-scalable data-shipping DBMS architectures*. In: *Proceedings of the 1st Conference on Computing Frontiers*, S. 222–234. ACM Press, 2004.