UNIVERSITÄT PASSAU

Seminar:

Verteilte Datenbanken

Thema:

Client/Server-Datenbanken

Sommersemester 1997

Markus Keidl

Prof. A. Kemper, Ph. D.
Lehrstuhl für dialogorientierte Systeme

Inhaltsverzeichnis

1 Einführung

Das Aufkommen von immer schnellerer und billigerer Computerhardware, wie etwa CPUs, RAM-Chips und Festplatten, führte dazu, daß heutige Arbeitsplatzrechner Ressourcen besitzen, die vor einigen Jahren nur in teureren Mainframe-Rechnern zur Verfügung standen. Wurden Desktop-Rechner davor dazu benutzt, Daten für einzelne Benutzer zu verwalten und zu speichern, so bot es sich jetzt an, Aufgaben von Mainframe-Computern auf Arbeitsplatzrechner zu verlagern. Zusätzlich forderten neue Anwendungen wie CAD oder die Automation von Büroaufgaben eine Verbindung von verschiedensten Arbeitsgruppen und deren Rechnern. Dies und die Fortschritte in der Netzwerktechnologie, hier sowohl im Hard- als auch im Softwarebereich, machten es möglich und nötig, die Rechner in großen Organisationen zu verbinden. Client/Server-Architekturen waren die Folge dieser Entwicklungen, und im Datenbankbereich führte dies zum Auftauchen von Client/Server-Datenbanken.

Eine kurze Einführung in den Aufbau von Client/Server-Netzen folgt nun in Abschnitt 2, bevor die grundsätzlichen Eigenschaften von Client/Server-Datenbanken in Abschnitt 3 dargestellt werden. Bei bestimmten Client/Server-Datenbanken legen einzelne Clients eigene Datencaches an; die dabei entstehenden Probleme und ihre Lösungen werden in Abschnitt 4 diskutiert. Abschließend wird noch etwas auf das World Wide Web als Client/Server-Datenbank eingegangen (Abschnitt 5).

2 Die Client/Server-Architektur

> *Client-Server is one of the buzziest of the buzzwords. It refers to a computing system that splits much of the workload between PC's and one or more large computers on a network. Think of it as a restaurant where the waiter takes your order for a hamburger, goes to the kitchen an comes back with some raw meat and a bun. You get to cook the burger at your table and add your favorite condiments. [...] It sounds like more work, but it has many advantages: the service is much faster, the food is cooked exactly to your liking, and the giant, expensive stove in the kitchen can be replaced by lots of cheap little grills.*
>
> Peter H. Lewis
> New York Times, Sonntag, 2. Mai 1993

Im Gegensatz zur allgemein verbreiteten Meinung beziehen sich die Begriffe *Client* und *Server* eigentlich nicht auf bestimmte Computer. So ist ein großer Mainframe-Rechner nicht unbedingt ein Server und eine kleines (Text-)Terminal ein Client. Beide Begriffe bezeichnen vielmehr bestimmte Programme. Ein Server-Programm stellt einen Service oder Dienst zur Verfügung, der über ein angeschlossenes Netzwerk angefordert werden kann. Im Gegensatz dazu bezeichnet Client ein Programm, das über ein Netzwerk eben jenen Dienst anfordert. In diesem Sinne kann auch eine Sun UltraSPARC der Client und ein PC der Server sein. Meistens werden auf einem bestimmten Rechner aber mehrere Server-Programme ausgeführt, so daß sich die Bezeichnung *Server* für diesen Rechner durchgesetzt hat, ebenso wie ein Computer, der von diesem Server einen Dienst anfordert häufig als *Client* bezeichnet wird.

Die Kommunikation zwischen Client und Server läuft über ein dazwischengeschaltetes Netzwerk. Dabei meldet sich ein Client zuerst beim Server an, indem er eine Nachricht an einen von dessen Ports schickt. Der Port wird durch die Art des Dienstes festgelegt, d. h. jeder Dienst besitzt einen fest zugewiesenen Port, der dem Client bekannt sein muß. Der Server generiert nun für diesen neuen Client einen eigenen Thread, der dann für die weitere Kommunikation mit dem Client zuständig ist. Der Thread beantwortet die Anfrage des Clients und schickt dabei eine neue Portnummer mit, über die die folgende Kommunikation abgewickelt werden soll. Diese neue Portnummer wird dabei dynamisch generiert. Abbildung 1 stellt diesen Zusammenhang nochmals graphisch dar.

3 Client/Server-Datenbanken

Client/Server-Datenbanken arbeiten nach dem oben vorgestellten Prinzip, d. h. auf einem Rechner läuft ein Server-DBMS-Prozeß und auf anderen Computern die Client-DBMS-Prozesse. Diese sind dabei meistens zwischen die auf dem Client-Rechner laufenden Anwendungen und die Netzsoftware geschaltet, so daß es für eine Anwendung im Grunde transparent ist, ob sie auf eine Client/Server-Datenbank oder eine lokal laufende Datenbank zugreift. Nach dem Grad der Interaktion zwischen Client- und Server-Prozessen kann man zwei unterschiedliche Architekturen unterscheiden:

- Die Anfrage der Anwendung wird vom Client über das Netz an den Server geschickt, dieser wertet die Anfrage lokal aus und sendet dann die berechneten Daten zurück, der sie an die Anwendung weiterreicht. Dieser Ansatz, oft unter dem Begriff **query-shipping** zusammengefaßt, wird vor allem in aktuellen relationalen DBMS eingesetzt.

- Der Client wertet die Anfrage der Anwendung aus; dabei stellt er fest, welche Daten er zur Auswertung braucht und fordert diese vom Server an. Man bezeichnet diese Methode als **data-shipping**. Sie wird in den Objekt-orientierten DBMS bevorzugt.

Im folgenden werden nun die Vor- und Nachteile dieser beiden Ansätze betrachtet.

3.1 *query-shipping*-Architekturen

Dieser Ansatz bietet folgende Vorteile:

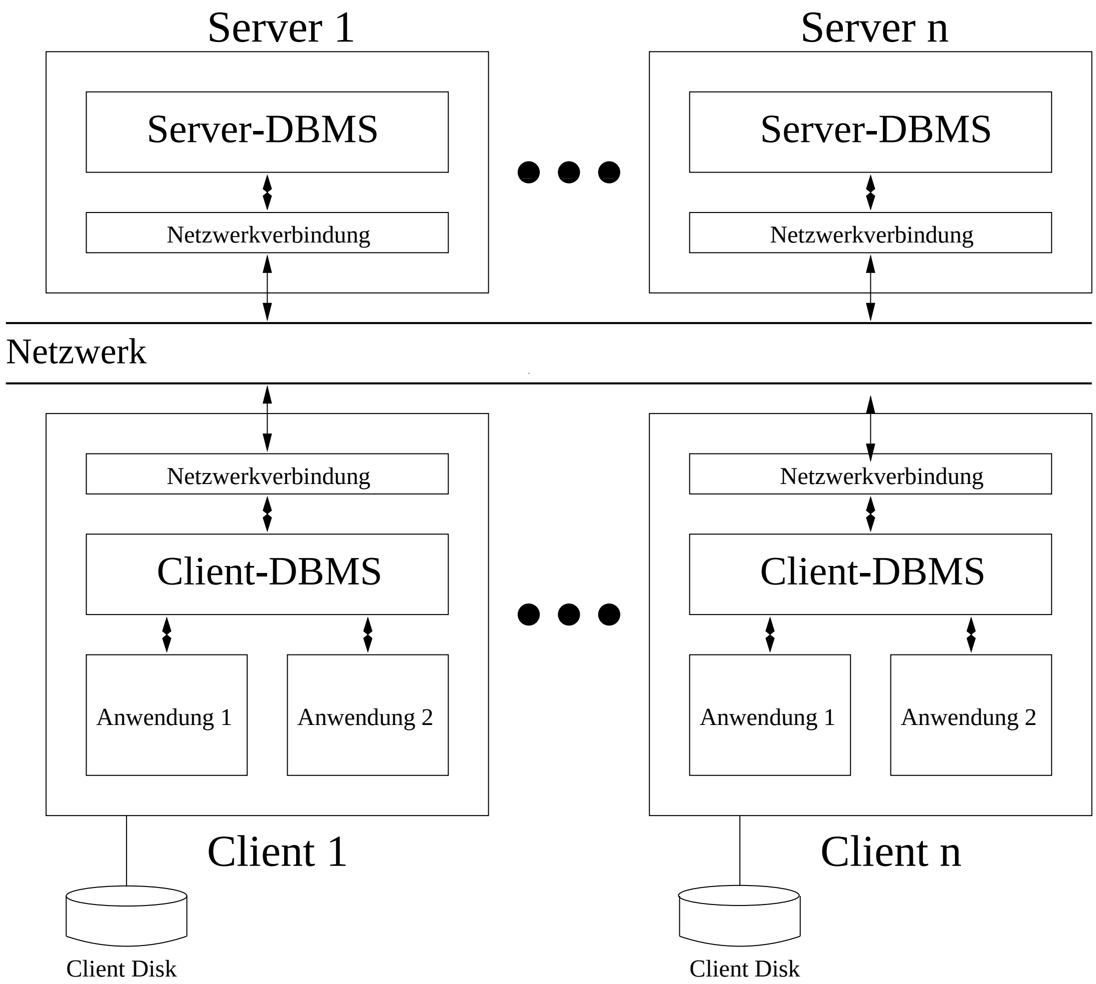

Abbildung 1: Die Client/Server-Architektur

- Da nur die für den Benutzer wirklich interessanten Daten über das Netzwerk gesendet werden, ist die Belastung desselben geringer als bei *data-shipping*. Wichtig ist dies vor allem in WANs. Zusätzlich ist auf den verschieden Client-Rechnern kein Pufferspeicher nötig.

- Der Umbau einer herkömmlicher Ein-Rechner-Datenbank zu einem Server-DBMS ist relativ einfach, da die grundsätzliche Struktur der beiden DBMS gleich ist. Dadurch kann man auf bekannte Algorithmen für Synchronisationskontrolle und Recovery zurückgreifen.

3.2 *data-shipping*-Architekturen

Im Gegensatz zur *query-shipping*-Architektur sind folgende Vorteile zu erkennen:

- Ein Großteil der DBMS-Funktionalität wird von der Server- auf die Client-Seite verlagert. Das kann die Performance verbessern, da viele Ressourcen bei heutigen Workstation-Server Umgebungen meistens auf der Client-Seite liegen.

- Eine bessere Skalierbarkeit ist gegeben, da das Hinzufügen von neuen Clients zu einem Anwachsen der Ressourcen führt.

Eine weitere Unterteilung dieser Architektur folgt daraus, daß ein Server entweder Seiten, d. h. physikalische Datenblöcke, versendet, wie sie etwa auf Hintergrundspeichern verwendet werden - man spricht dann von einem *Page-Server* - oder Objekte, d. h. logische Datenblöcke - in diesem Fall bezeichnet man den Server als *Object-Server*.

Bei *data-shipping* ist es hilfreich, daß die vom Server empfangenen Daten in einem Cache gespeichert werden, um sie etwa später wiederverwenden zu können. Zusätzlich kann sich dadurch der Kommunikationsbedarf zwischen Client und Server vermindern, was eine Entlastung des Netzwerkes bedeutet. Allerdings ist Caching kein Allheilmittel; um es sinnvoll benutzen zu können, muß die Datenbank ein Protokoll verwenden, daß die Cache-Konsistenz sicherstellt.

4 Caching

Beim dynamischen Vervielfältigen von Daten - wie es beim Caching auf Clients geschieht - muß gewährleistet sein, daß die Korrektheit der laufenden Programme erhalten bleibt. Durch das Vorhandensein von mehreren Kopien eines Datums kann das Transaktionskonzept gefährdet werden. Um dies genauer untersuchen zu können, muß man zwei Varianten des Caching unterscheiden:

- Caching innerhalb einer Transaktion: Dabei werden Daten, die von keiner aktiven Transaktion mehr geschützt werden, aus dem Cache entfernt. Die

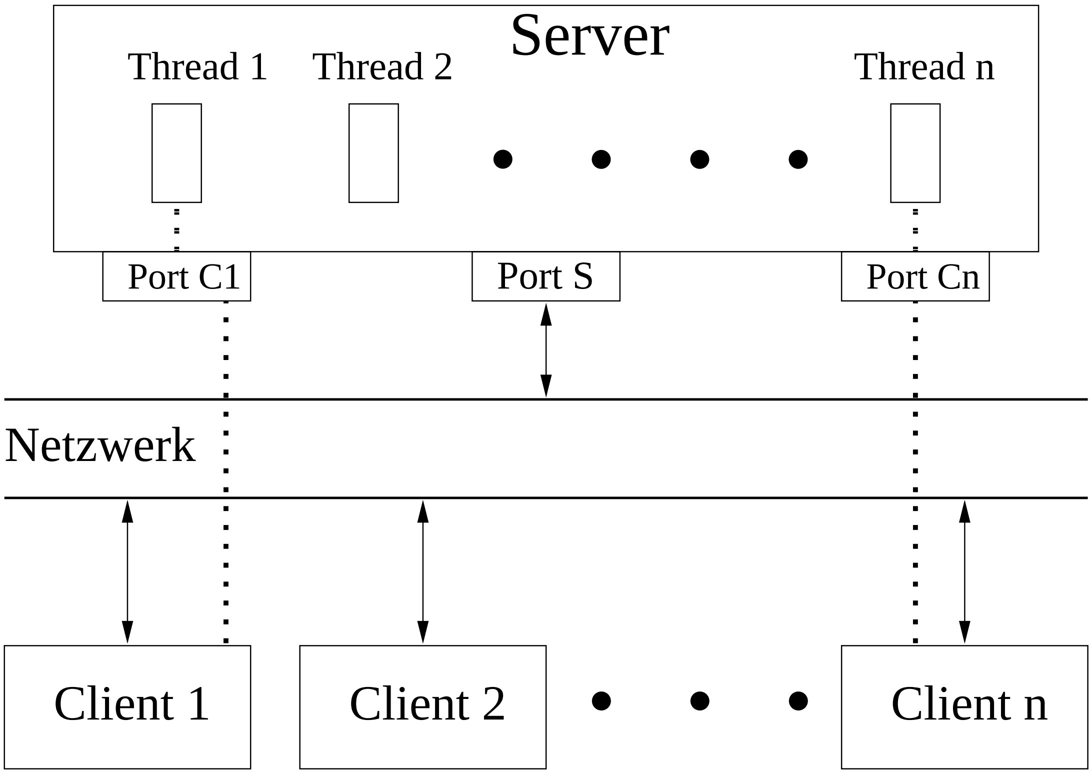

Abbildung 2: DBMS-Architektur bei *data-shipping*

Gültigkeit der restlichen Daten wird durch die normalen Synchronisationsmechanismen kontrolliert.

- Caching zwischen Transaktionen: Hier verbleiben Daten auch außerhalb der Transaktionsgrenzen im Client-Cache. Da diese Daten nun nicht durch die normalen Synchronisationsmechanismen kontrolliert werden, ist ein zusätzliches Protokoll nötig, daß die Cache-Konsistenz gewährleistet.

Um später genauer über die verschiedenen Algorithmen, die ein solches Protokoll realisieren, reden zu können, und um die Randbedingungen zu erkennen, die diese erfüllen müssen, folgt eine kurze Beschreibung der Client/Server-Architektur im *data-shipping*-Fall.

4.1 Client/Server-Architektur im *data-shipping*-Fall

Wie aus Abbildung 2 ersichtlich, besteht eine Client/Server-Datenbank bei der *data-shipping*-Architektur aus zwei verschiedenen Prozessen, die auf verschiedenen Rechnern laufen. Ein Client-DBMS-Prozeß läuft, ebenso wie alle Anwendungen, auf einem Client. Er ist dafür zuständig, die Anfragen der lokal laufenden

Anwendungen entgegenzunehmen und auszuwerten. Dabei sendet er Nachrichten an die Server-DBMS-Prozesse, entweder um Daten zu erhalten oder zur Transaktionsunterstützung. Aus Sicherheitsgründen laufen Client-DBMS-Prozesse und Anwendungen meistens in verschiedenen Adreßräumen. Ein Server-DBMS-Prozeß ist der eigentliche Eigentümer der Daten und muß dafür sorgen, daß die Datenintegrität und die Transaktionssemantik erhalten bleiben. Er verwaltet den Speicher, auf dem die permanente Datenbankversion, sowie die Logs gespeichert werden. Weiterhin bietet er Mechanismen zur Synchronisationskontrolle und zum Kopieren von Daten an.

Wie schon in 3.2 erwähnt, existieren zwei Möglichkeiten, Daten vom einem Server zu einem Client zu transferieren. Entweder werden die Daten seitenweise übermittelt, d. h. es werden Datenblöcke fester Größe gesendet (meistens 4 oder 8 KByte), oder die Daten werden objektweise übermittelt, d. h. logische Datenblöcke variabler Größe, die aber immer komplette Objekte enthalten. Die später vorgestellen Caching-Algorithmen können in beiden Fällen angewandt werden. Im folgenden wird jedoch zur Vereinfachung von einer Page-Server-Architektur ausgegangen.

4.2 Einteilung der Algorithmen

Bei der Einteilung der Algorithmen, die ab Abschnitt 4.2.3 vorgestellt werden, stütze ich mich auf [FCL95]. Dabei wird zuerst danach unterschieden, wie versucht wird, den Zugriff auf ungültige Daten zu verhindern. Grundsätzlich müssen solche Caching-Algorithmen verhindern, daß Transaktionen auf veraltete Daten zugreifen. Veraltet bedeutet, daß der Wert dieses Datums älter ist als der letzte Wert, der durch einen Transaktionscommit bestätigt wurde. Zwei Möglichkeiten können unterschieden werden:

- *Detection-based*-Algorithmen: Diese Algorithmen erlauben das Vorhandensein von veralteten Daten im Cache. Transaktionen dürfen aber nur einen Commit ausführen, wenn bestätigt werden kann, daß sie **nicht** auf solche veraltete Daten zugegriffen haben.

- *Avoidance-based*-Algorithmen: Sie verhindern, daß sich überhaupt veraltete Daten im Cache eines Clients befinden.

4.2.1 *Detection-based*-Algorithmen

Wie gerade vorher erwähnt, erlauben es diese Algorithmen, daß sich veraltete Daten im Cache befinden. Transaktionen, die einen Commit ausführen wollen, müssen also testen, ob sie auf veraltete Daten zugegriffen haben. Der Server muß dabei Informationen zu Verfügung stellen, die einem Client diesen Test erlauben. Bei diesem Verfahren wird also explizit danach geforscht, ob veraltete Daten verwendet wurden, deshalb auch die Bezeichnung *detection-based*.

Der Vorteil dieser Algorithmen liegt in ihrer Einfachheit. Durchzuführende Konsistenzaktionen betreffen nur einen einzelnen Client und den Server. Allerdings sind sie damit auch stärker von diesem Server abhängig, wodurch ein vergrößerter Verwaltungsaufwand entstehen kann.

Um die verschiedenen Algorithmen später besser beurteilen zu können, ist eine weitere Unterscheidung hilfreich.

Beginn der Gültigkeitsüberprüfung

Bevor eine Transaktion einen Commit durchführen kann, muß sie sicherstellen, daß die Daten, auf die sie zugegriffen hat, gültig waren. Je nachdem wann diese Prüfung gestartet wird, kann man drei Fälle unterscheiden. Allen ist gemeinsam, daß die Gültigkeit eines Wertes, nachdem sie bestätigt wurde, bis zum Ende der Transaktion bestehen bleibt. Deshalb darf der Server keiner anderen Transaktion erlauben, einen Wert zu ändern und dann einen Commit auszuführen, bevor nicht die Transaktion, der er die Gültigkeit diese Wertes bestätigt hat, beendet ist. Folglich muß eine Transaktion die Erlaubnis des Servers besitzen, um die Änderung eines Wertes zu commiten. Die drei unterscheidbaren Fälle sind:

- Synchron: Beim ersten Zugriff auf ein Datum im Cache prüft der Client die Gültigkeit des Wertes beim Server nach. Das geschieht synchron, d. h. die Transaktion kann nicht auf dieses Datum zugreifen, solange seine Gültigkeit nicht bestätigt ist.

- Asynchron: Auch hier wird beim ersten Zugriff die Gültigkeit des Wertes geprüft. Allerdings wartet die Transaktion nicht auf die Antwort des Servers, sondern geht stillschweigend davon aus, daß der Wert des Datums gültig ist. Erhält die Transaktion später die Nachricht, daß der Wert doch ungültig war, muß sie einen Abort ausführen.

- Zurückgestellt: Diese Lösung ist noch optimistischer als die asynchrone. Es wird nämlich keine Gültigkeitsprüfung der Werte im Cache vorgenommen, solange sich die Transaktion in der Ausführungsphase befindet. Erst wenn sie in die Commitphase eintritt, sendet sie Informationen über alle benutzten Daten an den Server. Dieser entscheidet dann, ob die Transaktion einen Commit ausführen darf oder einen Abort durchführen muß.

Jeder Ansatz stellt einen Tradeoff zwischen zu großem Verwaltungsaufwand und zu vielen Aborts da. Das Zurückstellen von Gültigkeitsprüfungen bietet etwa den Vorteil, daß die Gültigkeitsanfragen zusammengefaßt an der Server geschickt werden. Auch Gültigkeitsprüfungen für Transaktionen, die sowieso einen Abort hätten ausführen müssen, wären verschwendet. Die Nachteil dabei ist allerdings eine erhöhte Zahl von abgebrochenen Transaktionen, weil etwa ein Datenkonflikt zu spät entdeckt wurde.

Benachrichtungshinweise bei Änderungen

Die Kommunikation mit einem Server ist meistens eine teure Operation, so daß man versucht, möglichst wenig Nachrichten mit diesem auszutauschen. Dazu werden oft optimistische Algorithmen verwendet, die von wenigen Datenkonflikten zwischen verschiedenen Transaktionen ausgehen. Dies führt meist zu einer erhöhten Anzahl von Aborts, wenn es eben doch Konflikte gab und die aufgrund veralteter Daten im Cache eines Clients erst zu spät erkannt wurden. Eine Möglichkeit, die Menge veralteter Daten im Cache eines Clients und damit die Aborthäufigkeit zu vermindern, bieten Techniken, die von *avoidance-based*-Algorithmen übernommen wurden. Bei der Änderung eines Wertes werden Benachrichtigungen an die anderen Clients versendet. Eine solche Meldung führt dann dazu, daß eine veraltete Kopie im Client-Cache entfernt oder geupdatet wird.

Diese Benachrichtigungen können asynchron, während der Ausführung, oder nach dem Commit der entsprechenden Transaktion gesendet werden. Das Abschicken solcher Meldungen vor dem Commit ist allerdings gefährlich, wenn die veraltete Kopie geupdatet wird. Dann kann nämlich eine andere Transaktion darauf zugreifen. Sollte die ändernde Transaktion nun einen Abort ausführen müssen, so muß auch die andere aborten. Auch das Entfernen einer veralteten Kopie aus dem Cache kann zu solch unnötigen Aborts führen.

Update-Aktionen auf entfernten Clients

Zuletzt kann man die verschiedenen Algorithmen noch danach unterscheiden, welche Aktion auf einem anderen Client ausgeführt wird, wenn eine Benachrichtigung über die Änderung eines Datums eintrifft. Es sind drei verschiedene Verhaltensweisen denkbar:

- Die veraltete Kopie wird durch die neue ersetzt.
- Der veraltete Kopie wird aus dem Cache des Client entfernt. Will eine Transaktion wieder auf diese zugreifen, so muß sie von neuem beim Server angefordert werden.
- Man entscheidet dynamisch zwischen den beiden ersten Alternativen, wodurch man möglicherweise die Performance des Systems erhöhen kann.

4.2.2 *Avoidance-based*-Algorithmen

Algorithmen, die dieser Gruppe angehören, erlauben es nicht, daß sich veraltete Daten im Cache eines Clients befinden. Das heißt, eine Transaktion hat keine Möglichkeit auf ungültige Daten zuzugreifen. Erreicht wird dies durch ein ROWA-Protokoll (read-one/write-all). Dieses stellt sicher, daß alle existierenden Kopien eines geänderten Datums denselben Wert besitzen, wenn die Transaktion, die das

Datum verändert hat, einen Commit ausgeführt hat. Unter ROWA ist es einer Transaktion erlaubt, beliebige Kopien eines Datums zu lesen (meistens wohl eine im Cache des lokalen Clients). Änderungen an einem Datum müssen allerdings an alle existierenden Kopien, die nach dem Commit der ändernden Transaktion noch weiterbestehen, weitergeleitet werden. Dazu müssen sowohl Client- als auch Server-Software Funktionen bieten, die dies unterstützen. Ein Server muß zu jedem Zeitpunkt wissen, wo sich Kopien eines bestimmen Datums befinden, damit er die geänderte Kopie an die entsprechenden Clients weiterreichen kann. Auch hier ist es wieder sinnvoll, die Algorithmen weiter zu unterteilen.

Erklärung der Schreibabsicht

Aufgrund des ROWA-Protokolls können Transaktionen Daten, die sich im Client-Cache befinden, immer lesen. Interaktion mit dem Server ist also nur nötig, wenn ein Datum im Cache geändert werden soll oder wenn sich ein Datum, das eine Transaktion lesen will, nicht im Cache befindet. Im zweiten Fall wird das entsprechende Datum beim Server synchron angefordert. Erhält der Client eine Kopie des Datums, so bedeutet das gleichzeitig, daß er informiert wird, falls ein anderer Client den Wert ändern sollte. Beim Lesen eines Datums verhalten sich alle Algorithmen gleich. Beim Schreiben zeigen sich allerdings deutliche Unterschiede. Aufgrund des ROWA-Protokolls muß der Server vor dem Commit der Transaktion von ihrer Schreibabsicht benachrichtigt werden. Bestätigt der Server die Schreibabsicht, so garantiert er, daß der Client für die Dauer dieser Erlaubnis kein zweites Mal mehr darum anfragen muß.

Will eine Transaktion nun ein Datum ändern, für das sie keine Schreiberlaubnis besitzt, muß sie diese vom Server erhalten. Dabei kann man unterscheiden, wie um diese Erlaubnis angefragt wird:

- Synchron: Bei Fehlen einer Schreiberlaubnis wird sofort beim Server um diese gebeten und auf die entsprechende Antwort gewartet.
- Asynchron: Analog zum ersten Fall, nur wird nicht auf die Antwort des Servers gewartet, sondern von einer positiven ausgegangen. Sollte der Server hingegen keine Schreiberlaubnis gewähren, so muß die Transaktion abgebrochen werden.
- Zurückgestellt: Die entsprechenden Anfragen beim Server werden bis zum Ende der Ausführungsphase der Transaktion zurückgestellt.

Wie im *detection-based*-Fall kann man auch hier zwischen optimistischem und pessimistischem Ansatz unterscheiden. Das Zurückstellen von Anfragen spricht für eine optimistische Sichtweise. Allerdings kann das bei *avoidance-based*-Algorithmen weitreichende Konsequenzen haben. Bevor ein Server eine Schreiberlaubnis erteilt, sind möglicherweise bei einigen Clients Update-Aktionen nötig, um die Konsistenz der Daten zu gewährleisten, wodurch solche Konsistenzaktionen ziemlich arbeitsaufwendig werden. Deshalb scheint das Zurückstellen der Anfragen die bessere Lösung zu sein, auch wenn dies zu einer erhöhten Anzahl von Aborts führen kann.

Dauer der Schreiberlaubnis

Zusätzlich zum Zeitpunkt, wann eine Schreibabsicht abgegeben wird, kann man Algorithmen auch noch danach unterscheiden, wie lange eine von Server gewährte Schreiberlaubnis gültig ist.

- Die Dauer kann auf das Bestehen einer bestimmten Transaktion festgelegt sein. In diesem Fall startet eine Transaktion mit keiner Schreiberlaubnis und muß deshalb zu Beginn ihrer Ausführungszeit für alle Seiten, die sie ändern will, eine solche beantragen. Nach dem Ende der Transaktion wird jede einzelne vergebene Schreiberlaubnis vom Server wieder zurückgefordert.
- Im zweiten Fall erhält ein Client diese Erlaubnis solange, bis er sie von sich aus wieder an den Server zurückgibt, oder bis der Server sie zurückfordert.

Priorität eines von einem anderen Client ausgelösten Konflikts

Wird bei einem Client von einem anderen Client eine Konsistenzaktion ausgelöst, so kann man ihr zwei verschiedene Prioritäten zuordnen:

- Die auszuführende Konsistenzaktion muß warten, wenn sie sich mit den Aktionen einer lokal ablaufenden Transaktion überschneidet. Und zwar solange, bis die lokale Transaktion beendet ist.
- Die lokale Transaktion, mit deren Aktionen sich die Konsistenzaktion überschneidet, wird abgebrochen.

Im ersten Fall werden den Clients etwas stärkere Rechte zugestanden, was das Lesen von lokalen Kopien im Cache betrifft. Eine schreibende Transaktion, die nicht auf dem lokalen Client läuft, muß sich nämlich hinter einer lokalen Transaktion, die gerade eine Seite lesen möchte, einordnen. Im zweiten Fall würde die lokale Transaktion einfach abgebrochen.

Update-Aktionen auf entfernten Clients

Hierbei gibt es die drei gleichen Ansätze wie in 4.2.1. Im *avoidance-based*-Fall müssen die Update-Aktionen allerdings begonnen und beendet werden, bevor es der Transaktion erlaubt ist, in die Commitphase einzutreten. Das fordert das ROWA-Protokoll. Wird zusätzlich als Update-Aktion noch das Ändern von veralteten Seiten gewählt, so muß jeder Client, der eine Kopie der geänderten Seite im Cache hatte, mit dem Server und dem Client, auf dem die Transaktion läuft, die für die Änderung verantwortlich war, an einem Zwei-Phasen-Commit teilnehmen. Werden die veralteten Seiten einfach aus dem Cache entfernt, so ist dies nicht nötig.

4.2.3 Verschiedene Caching-Algorithmen

Im folgenden werden drei verschiedene Algorithmenarten vorgestellt (entnommen aus [FCL95]), die für die Konsistenz des Caches auf einem Client sorgen. Zusätzlich werden verschiedene Implementationen angeführt.

Server-Based Two-Phase Locking (S2PL)

S2PL ist ein *detection-based*-Algorithmus, der beim ersten Zugriff auf eine Seite synchron die Gültigkeit prüft. Bevor eine Transaktion einen Commit ausführen kann, muß sie zuerst Zugriff auf eine bestimmte Kopie jedes Datums haben, das sie gelesen oder geschrieben hat. Diese bestimmte Kopie liegt meistens auf dem Server. Beim Lesen muß bestätigt werden, daß die lokale Kopie im Cache denselben Wert besitzt wie die Kopie auf dem Server, beim Schreiben muß der neue Wert in das Server-Orginal übernommen werden.

Eine Implementation hiervon ist *Caching 2PL* (C2PL). Dabei besitzt jede Kopie eine spezielle Versionsnummer, die den Status der Seite eindeutig bestimmt. Wenn nun eine Transaktion auf eine Seite zugreifen möchte, für die sie noch keinen Lock besitzt, fordert sie beim Server den entsprechenden an und wartet auf dessen Antwort. Sollte die Seite lokal im Cache vorhanden sein, wird dabei die aktuelle Versionsnummer mitgeschickt. Der Server blockiert die Lockanfrage, wenn eine andere Transaktion bereits einen Lock hält, der dazu im Konflikt steht, bis die andere Transaktion ihren Lock wieder freigibt. Wenn der Server einen Read-Lock gewährt, testet er gleichzeitig ob die mitgeschickte Versionsnummer aktuell ist. Falls nicht, sendet er eine aktuelle Kopie mit dem gewährten Lock zurück. Dabei realisiert C2PL striktes Zwei-Phasen-Locking, d. h. Locks werden bis zum Commit oder Abort nicht mehr freigegeben. Deadlocks werden durch eine zentrale Instanz auf der Server erkannt und dadurch aufgelöst, das die jüngste Transaktion abgebrochen wird.

Callback Locking (CBL)

CBL ist ein *avoidance-based*-Algorithmus. Damit sind Seiten im lokalen Cache immer gültig, so daß Transaktionen sie lesen können ohne den Server zu kontaktieren (sie brauchen also nur einen lokalen Read-Lock). Ist eine Seite, auf die zugegriffen werden soll, nicht im Cache, sendet der Client eine Anfrage an den Server, und erhält, falls kein anderer Client die Schreiberlaubnis dazu besitzt, eine gültige Kopie. Schreibabsichten werden in CBL synchron erklärt, deshalb muß ein Client eine Schreiberlaubnis für eine Seite haben, bevor er lokal darauf einen Write-Lock vergeben kann. Eine Transaktion kann nach dem Ende ihrer

Ausführungsphase sofort einen Commit ausführen, da die Schreibrechte während der Ausführungsphase erhalten wurden.

Implementationen von CBL sind Callback-Read (CB-R), wobei die Schreiberlaubnis abläuft, wenn die Transaktion beendet ist, die sie angefordert hat, oder Callback-All (CB-A), der die Schreiberlaubnis solange behält, bis sie vom Server zurückgefordert oder die Seite aus dem Cache entfernt wird. Beide Varianten entfernen Seiten aus dem lokalen Cache, falls es zu einer durch einen anderen Client ausgelösten Update-Aktion kommen sollte.

Da der Algorithmus *avoidance-based* ist, behält der Server immer den Überblick über alle Kopien einer bestimmten Seite. Entfernt ein Client eine Seite aus seinem Cache, so teilt er dies dem Server mit seiner nächsten Nachricht mit wodurch die Übersicht des Servers leicht zeitverzögert aktualisiert wird.

Transaktionen erhalten ihre Locks von einem lokalen Lock-Manager. Dabei werden Read-Locks ebenso wie Write-Locks auf Seiten, die sich im lokalen Cache befinden, vom Lock-Manager des Clients vergeben, auf dem die Lock-Anfrage gestellt wurde, ohne den Server davon zu benachrichtigen. Write-Lock-Anforderungen für Seiten, für die noch keine Schreiberlaubnis besteht, führen dazu, daß der Client die Schreibabsicht beim Server registriert und solange wartet bis dieser die Schreiberlaubnis bestätigt.

Erhält ein Server eine solche Schreibanforderung, sendet er an alle Clients, die eine Kopie derjenigen Seite im Cache haben, eine Callback-Anfrage. Diese behandeln die Anfrage wie eine Write-Lock-Anforderung. Kann der Lock gewährt werden, so wird die Seite aus dem Cache entfernt und die Server-Anfrage bestätigt. Kann der Lock nicht gewährt werden, weil etwa ein Konflikt mit einem lokalen Lock besteht, teilt der Client dem Server mit, daß die Seite zur Zeit gerade benutzt wird. Wenn alle Callbacks des Server beantwortet sind, registriert dieser die Schreiberlaubnis für die entsprechende Seite und teilt dies dem Client, der die Anfrage gestellt hat, mit. Nachfolgende Lese- oder Schreibanforderungen werden vom Server solange blockiert, bis der Client die Schreiberlaubnis zurückgibt oder der Server sie sonstwie zurückfordert.

Trifft eine Leseanforderung für eine Seite bei einem Server ein, so ist die Reaktion des Servers Algorithmus-abhängig. Bei CB-R, wird die Anfrage einfach solange blockiert, bis die Transaktion beendet ist, die die Schreiberlaubnis besitzt. CB-A hingegen sendet eine Downgrade-Anfrage an den Client mit der Schreiberlaubnis, wonach dieser bestätigt, daß er die Erlaubnis für diese Seite nun nicht mehr besitzt. Ein Client, der eine Downgrade-Anfrage erhält, versucht zuerst einen Read-Lock für diese Seite zu erhalten, um sicherzustellen, daß kein Konflikt mit einem Write-Lock einer lokal laufenden Transaktion auftritt. Wird ein Konflikt entdeckt, blockiert die Downgrade-Anfrage und der Server wird darüber informiert. Deadlocks werden demnach auf dem Server entdeckt und dadurch aufgelöst, daß die jüngste Transaktion abgebrochen wird. Um die Recovery zu vereinfachen, sendet der Client am Ende einer Transaktion die geänderten Seiten im Cache an den Server.

Optimistic Two-Phase Locking (O2PL)

O2PL ist ebenso wie CBL *avoidance-based*, aber optimistischer, weil er die Schreibabsicht bis zum Ende der Ausführungsphase der Transaktion zurückstellt. Die zwei nachfolgend beschriebenen Beispiele unterscheiden sich nur darin, wie Update-Aktionen bei anderen Clients gehandhabt werden. O2PL-I macht die entsprechende Seite ungültig und entfernt sie aus dem Cache, O2PL-P ersetzt die entsprechende Seite bei allen Clients, die Kopien der geänderten Seite in ihrem lokalen Cache halten.

Bei O2PL besitzt jeder Client einen lokalen Lock-Manager, bei dem eine Transaktion den entsprechenden Lock anfordern muß, bevor sie auf eine Seite zugreifen kann. Während der Ausführungsphase erhält keine Transaktion Locks vom Server. Eine Transaktion ändert Seiten nur im lokalem Cache und behält diese dort solange, bis die Commitphase betreten wird (außer sie sind veraltet). Wenn eine solche Transaktion in die Commitphase eintreten will, sendet sie die geänderten Seiten an den Server. Dieser fordert dann exklusive Locks auf diese Seiten an, damit er sie ersetzen kann, und behält sie solange, bis die Transaktion beendet ist.

Wenn der Server die benötigten Locks erhalten hat, sendet er an alle Clients, die Kopien der entsprechenden Seiten besitzen, eine entsprechende Nachricht. Jeder dieser Clients fordert im Auftrag der sich im Commit befindlichen Transaktion exklusive Locks auf die Seiten an, sofern sich sich in seinem Cache befinden. Sollte eine der lokal laufenden Transaktionen ein Read-Lock auf eine lokale Kopie einer dieser Seiten besitzen, muß die sich beendende Transaktion warten, bis jene fertig ist. Hat ein Client alle benötigten Locks, so führt er für die entsprechenden Seiten Konsistenzaktionen aus.

Bei O2PL-I werden die geänderten Seiten aus dem Cache entfernt, alle Locks freigegeben und dem Server eine Bestätigung geschickt (dabei ist kein Zwei-Phasen-Commit erforderlich). O2PL-P benötigt im Gegensatz dazu das Zwei-Phasen-Commit, damit alle geänderten Werte atomar ersetzt werden können. Zuerst sendet dazu jeder Client eine Bestätigung, daß er alle benötigten Locks erhalten hat - diese Nachricht dient auch als “prepare“-Nachricht. Hat der Server alle Bestätigungen erhalten, sendet er Kopien der geänderten Seiten an die Clients, womit die zweite Phase des Zwei-Phasen-Commits eingeleitet wird. Beim Empfang dieser Seite fügt jeder Client sie in seinen Cache ein und gibt die entsprechenden Locks frei. Lokale Deadlocks werden durch einen lokalen Wartegraphen erkannt, globale durch einen Algorithmus auf dem Server, der periodisch alle lokalen Wartegraphen abfrägt und daraus einen globalen Wartegraphen bildet. Bei einem entdecktem Deadlock wird die jüngste Transaktion abgebrochen.

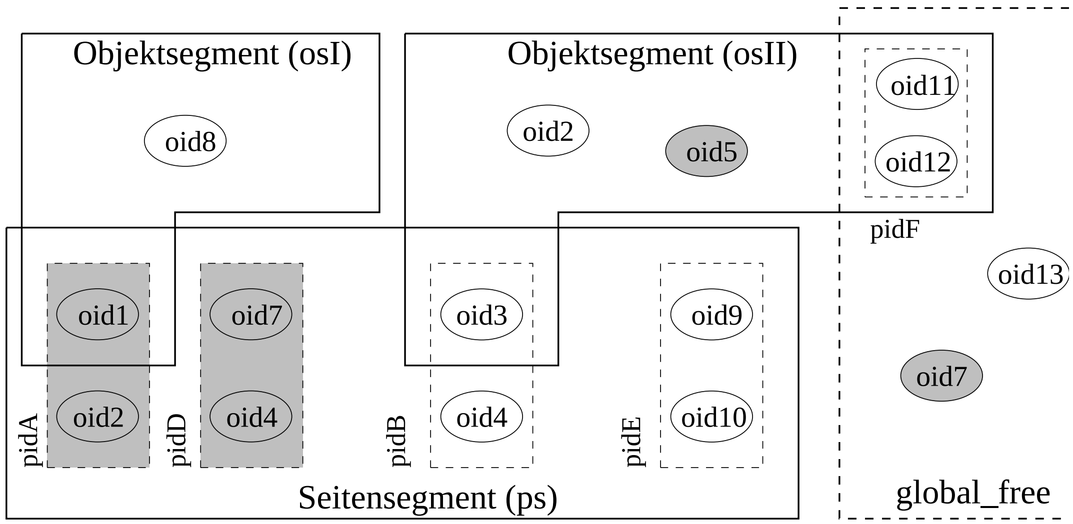

Abbildung 3: Beispiel für einen Dual-Buffer Pool

4.3 Caching von Seiten und Objekten im selben Cache

Bisher wurde immer davon ausgegangen, daß sich im Cache eines Clients Seiten (oder aber Objekte) befinden. Obwohl das bei relationalen DBMS ausreicht, wäre es bei OODBMS von Vorteil, Objekte und Seiten gleichzeitig im Cache zu halten. Wie man in [KK93] nachlesen kann, ist dies durchaus möglich. Im folgenden soll dieses Verfahren kurz erklärt werden.

4.3.1 Dual-Buffer Management

Es hat sich gezeigt, daß die Segmentierung eines Puffers - und ein Cache ist ja nur ein Spezialfall davon - effizienter ist, als die Verwendung eines globalen Puffers, weil man dabei für jedes Segment eine optimale Ersetzungsstrategie festlegen kann. Zusätzlich ist es nun möglich, in einem Segment entweder Objekte oder Seiten zu puffern - im folgenden werden diese der Einfachheit halber als *granules* bezeichnet. Bei einem Puffer, der nur Seiten enthält, entstehen Probleme, wenn die Objektbasis schlecht geclustert ist. Andererseits kann bei einem puren Objektpuffer ein Verwaltungs-Overhead entstehen, nämlich wenn die Seiten, auf denen die Objekte - im Allgemeinen auf dem Hintergrundspeicher - gespeichert sind, eine gute Clusterung aufweisen (diese Seiten werden auch als *Homepage*-Seiten der Objekte bezeichnet).

Abbildung 3 (entnommen aus [KK93]) zeigt ein Beispiel für einen Dual-Buffer Pool. Er ist in vier Segmente unterteilt, wovon zwei Objektpuffer darstellen (*osI* und *osII*), eines einen Seitenpuffer (*ps*) und das letzte den standardmäßig generierten Puffer *global_free*, der beides, Objekte und Seiten, enthalten kann. Dabei wird von höchstens einem Segment eine Nutzungsstatistik über ein *granule* geführt. Die Nutzungsstatistik von *oid1* liegt etwa im Segment *osI*, die von *pidA* unabhängig davon in *ps*. Die Nutzungsstatistik für Objekte wird auf den neuesten Stand gebracht, wenn auf ein Objekt zugegriffen wird; bei einer Seite wird sie aktualisiert, wenn ein Objekt dieser Seite zum ersten Mal referenziert und dazu die Seite benutzt wird. Führt die gewählte Ersetzungsstrategie dazu, daß ein Objekt oder eine Seite aus einem Segment entfernt werden muß, so wird es zuerst in das Segment *global_free* verschoben; dort sind alle *granules* enthalten, die entfernt werden können, wenn Pufferspeicher gebraucht wird.

Wie in Abbildung 3 ersichtlich, können sich einzelne Segmente durchaus überlappen. So ist etwa *oid3* sowohl in *osII* als auch, über seine Homepage *pidB*, in *ps* enthalten. Dadurch vermindern sich die Kopierkosten ebenso wie sich die Pufferausnutzung erhöht. Auch das Vorhandensein von Duplikaten ist möglich, wie das Objekt *oid2* zeigt; es ist sowohl getrennt in *ps*, als auch in *os2* enthalten.

Die Schattierung eines *granule* zeigt an, daß es verändert wurde und deshalb gespeichert werden muß, bevor es aus dem Hauptspeicher entfernt werden kann. Speichern heißt dabei, das die Homepage des Objektes geladen (falls sie sich nicht sowieso im Speicher befindet), das Objekt in die Seite kopiert, und diese dann auf den Hintergrundspeicher geschrieben wird.

4.3.2 Strategien

Algorithmen, die diese verschiedene Segmente effizient verwalten wollen, müssen zwei verschiedene Zeitpunkten berücksichtigen:

- Kopierzeitpunkt: Dieser gibt an, zu welchem Zeitpunkt - genauer, aufgrund welchen Ereignisses - ein Objekt von seiner Homepage in ein Objektsegment kopiert wird.

- Relokationszeitpunkt: Bestimmt den Zeitpunkt, wann eine Objektkopie entfernt und wegen möglicher Änderungen auf seine Homepage zurückverschoben wird.

Bei beiden Zeitpunkten gibt es zwei Extreme, diese werden als *eager* und *lazy* bezeichnet. Sie werden im folgenden kurz vorgestellt:

Eager Object Copying (EOC)

Hierbei wird ein Objekt bei seiner ersten Referenzierung aus seiner Homepage in ein Objektsegment kopiert. Somit erfolgen Zugriffe auf Objekte immer auf Kopien dieser Objekte. Aus Effizientgründen sollten die Objektsegmente dabei im Vergleich zu den Seitensegmenten groß gewählt werden, damit kein Speicher für unbenutzte Seiten verschwendet wird.

Im Allgemeinen wird EOC, im Vergleich zu NOC[1], die Zahl der Seitenfehler verringern. Es ist allerdings schlecht geeignet, wenn viele sequentielle Scans durch einen Datenbestand ausgeführt werden, da dabei die Objekte unnötigerweise kopiert werden. Werden die Objekte zusätzlich noch geändert, so erhöht sie die Zahl der Seitenfehler weiter, wenn die Homepage eines geänderten Objekts davor schon aus dem Pufferspeicher verdrängt wurde. Auch wenn die Daten gut geclustert sind, besitzt EOC Nachteile, da das Kopieren der Objekte die Ausnutzung des Speichers nicht wesentlich verbessert.

Lazy Object Copying (LOC)

Objekte werden in diesem Fall nur kopiert, wenn ihre Homepage aus dem Speicher entfernt wird, sie selbst aber laut der gewählten Ersetzungsstrategie noch im Speicher bleiben sollen.

Bei einem sequentiellen Scan durch einen Datenbestand, bei dem NOC die besten Ergebnisse liefert, werden Seiten häufiger referenziert als Objekte, und deshalb diese eher aus dem Speicher entfernt als ihre Homepages. Daher liefert dieser Ansatz hierbei annehmbare Ergebnisse. Wird immer wieder auf die gleiche Menge von Objekte zugegriffen, die über größere Zahl von Seiten verteilt sind, so kopiert LOC Objekte so, daß der Puffer gut ausgenutzt wird.

Eager Relocation (ERL)

ERL kopiert ein Objekt sofort auf seine Homepage zurück, wenn diese in den Speicher geladen wird, egal ob es geändert wurde oder nicht. Duplikate werden vermieden, wenn man LOC und ERL zusammen verwendet, da eine Objektkopie nur existiert, wenn die Homepage sich nicht im Speicher befindet. Allerdings werden bei ERL einige Objekte unnötigerweise zurückverschoben, etwa wenn die Homepage von *oid5* geladen und sofort wieder aus dem Speicher entfernt wird, wird *oid5* zuerst zurückgeschrieben und darauffolgend in ein Objektsegment kopiert.

Lazy Relocation (LRL)

Lazy Relocation schreibt Objekte so spät wie möglich zurück. Wenn einmal eine Objektkopie angelegt wurde, so wird diese nicht eher aufgegeben, bis sie aus dem Speicher entfernt werden soll. Damit wird unnötiges Zurückschreiben von Objekten vermieden, aber möglicherweise auch die letzte Chance übersehen, ein Objekt zu speichern ohne einen Seitenfehler auszulösen. Da beim Neuladen von Homepages Duplikate enstehen können, falls sich Objekte der Seite noch im Speicher befinden, ist die Speicherausnutzung etwas schlechter als bei ERL.

[1] No Object Copying, d. h. es werden keine Objekte von ihrer Homepage in ein Objektsegment kopiert. Dabei ist dann auch keine Relokation nötig.

Verschieden Performance-Tests in [KK93] haben gezeigt, daß LOC zusammen mit ERL die besten Ergebnisse liefert.

5 Das WWW als Client/Server-Architektur

Mit Einführung des WWW stieg die Bedeutung des Internet von Jahr zu Jahr fast exponentiell an. Das führte dazu, daß es sich heute fast keine Firma mehr leisten kann, nicht im Internet vertreten zu sein. Waren es früher vor allem wissenschaftliche Daten, die dort angeboten wurden, so lassen sich heute beliebige Informationen finde; besonders, da auch viele verschiedene Datenbanken über das WWW erreichbar sind.

Der Aufbau des WWW legt nahe, daß es sich um eine Client/Server-Architektur handelt. Auf einem Rechner läuft ein HTTP-Server. Tippt man in einen Internet-Browser eine URL [2] ein, so wird aus dieser die Adresse des Servers berechnet und die Anfrage an ihn weitergeleitet. Dabei agiert der Browser als Client. Der weitere Ablauf entspricht dem Schema, daß in Abschnitt 2 vorgestellt wurde. Über solche HTTP-Server sind nun auch Datenbanken ansprechbar, etwa solche der Deutschen Bundesbahn, in denen verschiedene Fahrpläne gespeichert sind. Der Server ist aber kein DBMS-Prozeß, wie es bei einer Client/Server-Datenbank der Fall wäre, sondern ein Programm, daß bei einer Anfrage auf einen entsprechenden DBMS-Prozeß zugreift, und die von ihm gelieferten Daten an den HTTP-Client weiterleitet. Zwischen den Datenbanken im Internet gibt es aber keine Verbindungen, so daß man nicht von einem verteilten Datenbanksystem sprechen kann.

Wie bei Client/Server-Datenbanken üblich, wird auch im WWW Caching verwendet. Dabei werden aber nicht solch ausgefeilte Algorithmen eingesetzt, wie in Abschnitt 4.2.3 vorgestellt wurden.

5.1 Proxies

Proxies sind spezielle Server-Prozesse, die ein Internet-Browser konsultiert, bevor er eine Anfrage an den mit der URL spezifizierten Server sendet. Dazu ist es allerdings nötig, daß der Browser entsprechend konfiguriert wird, d. h. Adressen und Ports der Proxies müssen ihm mitgeteilt werden. Für die verschiedenen verwendeten Protokolle (HTTP, FTP, Gopher, ...) kann jeweils ein eigener Proxie-Server angegeben werden. Will ein Internet Browser eine Seite, spezifiziert durch eine URL, laden, so prüft er zuerst, ob der Proxie eine Kopie in seinem Cache liegen hat. Wenn ja, wird diese vom Proxie an den Browser gesendet und angezeigt. Andernfalls wird die Anfrage vom Browser an den in der URL angegebenen

[2] Uniform Resource Locator

HTTP-Server geleitet. Die Einstellmöglichkeiten eines Proxies sind vielfältig. Er kann zu bestimmten Zeitpunkten festgelegte Seiten auf ihre Aktualität testen oder, falls er eine von einem Browser verlangte Seite nicht im Cache hatte, diese beim entsprechenden Server anfordern und dann in seinen Cache kopieren.

5.2 Lokale Browser-Caches

Die meisten Internet-Browser bieten die Möglichkeit, auf einer lokalen Festplatte einen Cache einstellbarer Größe einzurichten. Dieser enthält verschiedene Seiten, die in letzter Zeit vom Benutzer abgerufen wurden. Die Konfigurationsmöglichkeiten variieren dabei von Browser zu Browser. Gemeinsam ist den meisten jedoch, daß man einstellen kann, wie lange eine Seite maximal im Cache liegen soll, oder wann geprüft wird, ob eine Seite noch aktuell ist. Der Netscape Navigator bietet etwa die Möglichkeit, die Aktualität einer Seite beim ersten Zugriff in einer Session, bei jedem Zugriff oder nie zu prüfen. Zusätzlich bietet fast jeder Browser eine Möglichkeit, die gerade aktuell angezeigte Seite frisch vom dem Server zu laden, von dem sie stammt.

Literatur

[Com91] COMER, D. E.: *Internetworking with TCP/IP Volume I – Principles, Protocols, and Architecture*. Englewood Cliffs, NJ, USA: Prentice Hall, 1991

[FCL95] FRANKLIN, M. J.; CAREY, M. J.; LIVNY, M.: Transactional Client-Server Consistency: Alternatives and Performance / University of Maryland, College Park. 1995. – Technical Report CS-TR-3511

[Fra96] FRANKLIN, M.: *Client Data Caching: A Foundation*. Kluwer Academic Press, 1996

[KE96] KEMPER, A.; EICKLER, A.: *Datenbanksysteme – Eine Einführung*. R. Oldenbourg Verlag, 1996

[KK93] KEMPER, A.; KOSSMANN, D.: Dual-Buffering Strategies in Object Bases / Universität Passau. 1993. – Technical Report MIP-9309