

Neuimplementierung einer transaktionalen Message Queue

Studienarbeit Nr. 1750
Fakultät Informatik
Institut für Parallele und
Verteilte Höchstleistungsrechner
Universität Stuttgart
Breitwiesenstraße 20-22
70565 Stuttgart

Albrecht Messner

30. September 1999

Zusammenfassung

In dieser Arbeit wird die Entwicklung einer transaktionalen Warteschlange für Nachrichten (Message Queue) dargestellt. Der primäre Einsatzbereich der Queue ist ein Mobile Agenten-Umfeld, in dem mit Hilfe der Queue die Exactly-Once-Ausführung von Agenten sichergestellt werden soll. Der im Rahmen einer Arbeit über die Exactly-Once-Ausführung (s. [2]) entstandene Prototyp einer Message Queue soll durch die hier vorgestellte Implementierung ersetzt werden.

Als Kommunikationsschicht kommt CORBA zum Einsatz, die Transaktionen verwaltet eine Implementierung des für CORBA spezifizierten Object Transaction Service (OTS). Diese Technologien werden kurz beschrieben.

Für die Speicherung der Nachrichten werden zwei Ansätze vorgestellt. Im ersten wird gezeigt, wie zu diesem Zweck ein relationales Datenbanksystem (RDBMS) in die Anwendung eingebunden wird; im zweiten wird ein eigenes System zur Speicherung entwickelt. Auf diesem zweiten Ansatz liegt der Schwerpunkt der Arbeit. Die verwendeten Programmier- und Optimierungstechniken werden diskutiert. Es wird gezeigt, wie das System durch Logging- und Recovery-Mechanismen den Transaktionseigenschaften genügt.

Zum Schluß wird die Leistungsfähigkeit der entstandenen Systeme anhand von Messungen beurteilt. Zum Vergleich wird auch der Prototyp aus [2] herangezogen.

Inhaltsverzeichnis

1	Einleitung	4
1.1	Was ist eine Message Queue?	4
1.2	Ein typisches Szenario	4
1.3	Vorteile des Message Queuing	5
1.4	Überblick über kommerzielle Produkte	6
1.4.1	BEA Tuxedo/Q	6
1.4.2	IBM MQSeries	7
2	Anwendungsgebiet Mobile Agenten	8
2.1	Exactly-Once-Ausführung für Mobile Agenten	8
2.2	Anforderungen an die Implementierung	9
3	Die Entwicklungsumgebung	11
3.1	CORBA	11
3.2	CORBA Services	13
3.2.1	Object Transaction Service	13
3.2.2	Event Service	14
4	Interfaces und Datentypen	16
4.1	Aufbau einer Nachricht	17
4.1.1	Header	17
4.1.2	Attribute	18
4.1.3	Nutzlast	18
4.2	Schnittstellen zur MQ	18
4.2.1	Standard-Interface	18
4.2.2	FIFO-Interface	19
4.2.3	Service-Interface	20
5	Internes Design der MQ	21
5.1	Erster Ansatz: Verwendung eines RDBMS	21
5.2	Zweiter Ansatz: transaktionale Resource	25

5.2.1	Programmiertechniken	25
5.2.2	OTS und transaktionale Ressourcen	27
5.2.3	Überblick über die Module der MQ	28
5.2.4	Logging und Recovery	28
5.2.5	Filemanager	31
5.2.6	Logmanager	36
5.2.7	Resource-Objekt	40
5.2.8	Restart-Objekt	42
5.2.9	Locking	43
5.2.10	“Gesamtansicht”	44
6	Leistungsbewertung	46
6.1	Messungen mit dem Testclient	46
6.1.1	Ursprüngliche Implementierung	47
6.1.2	Implementierung mit DB2	48
6.1.3	Implementierung mit eigenem Resource Manager	49
6.2	Test mit der EOP-Implementierung	49
7	Zusammenfassung und Ausblick	50
7.1	Zusammenfassung	50
7.2	Ausblick	51
A	Listings	52
B	Verwendungshinweise für die Message Queue	56
C	Meßergebnisse	58

Kapitel 1

Einleitung

1.1 Was ist eine Message Queue?

Eine Message Queue (MQ) ist ein Zwischenspeicher. Es stehen für den Benutzer zwei Hauptoperationen zur Verfügung: das Einfügen und das Entnehmen von Nachrichten. Dabei besteht aus Sicht der Message Queue eine Nachricht aus binären Daten beliebiger Länge und beliebigen Inhaltes.

Normalerweise ist eine Message Queue transaktionsorientiert. Damit gelten – wie für alle anderen transaktionalen Systeme – die “ACID-Eigenschaften”: Atomicity, Consistency, Isolation und Durability. Da eine Message Queue aus anderer Sicht ein Datenbanksystem ist, das lediglich die atomaren Operationen *insert* und (*read; delete*) zur Verfügung stellt, haben die genannten Eigenschaften hier keinerlei eingeschränkte oder erweiterte Bedeutung gegenüber einem DBMS: Details hierzu können aus jedem Lehrbuch über Datenbanksysteme entnommen werden (z. B. [1] und [3]).

Eine Message Queue ist ein Hilfsmittel für die Kommunikation zwischen Prozessen und kann somit in die Kategorie “Middleware” eingereiht werden.

Anhand der Abbildung 1.1 soll verdeutlicht werden, wie eine Warteschlange in ihr Umfeld eingebunden wird. Ein *Producer* ist ein Client, der Nachrichten erzeugt und sie per *put* in der MQ zwischenspeichert. Der *Consumer* nimmt die Nachricht mittels eines *get*-Aufrufs wieder aus der Queue heraus. Natürlich kann ein Client gleichzeitig *Producer* und *Consumer* sein.

1.2 Ein typisches Szenario

In Abbildung 1.2 ist ein typischer Anwendungsfall für den Einsatz einer transaktionalen Message Queue dargestellt. Ein Prozess erzeugt Daten, die von einem anderen Prozess weiterverarbeitet werden sollen. Die Daten werden nicht direkt an den nächsten Prozess weitergegeben, sondern als Nachricht in einer Queue zwischengespeichert. Befindet

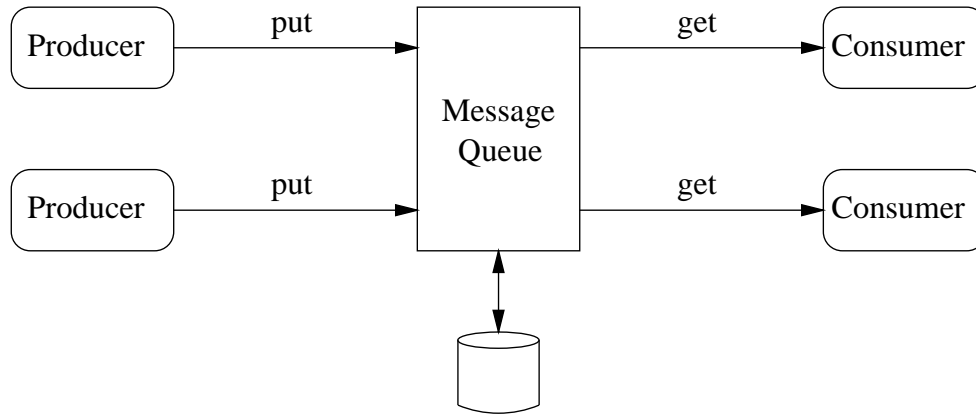


Abbildung 1.1: Skizze einer Anwendung mit Message Queue

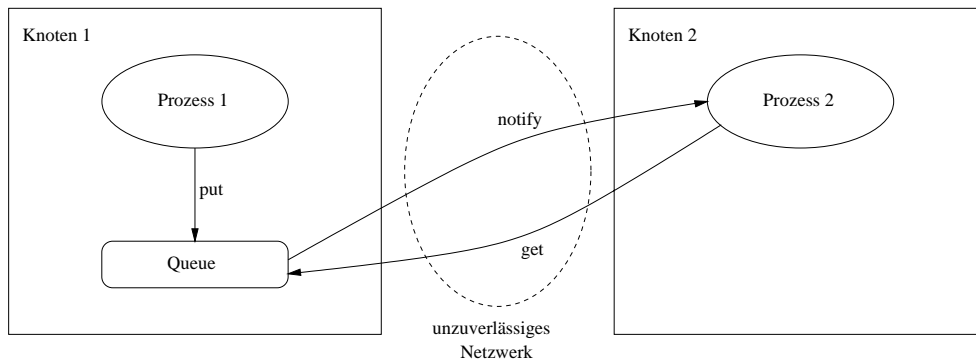


Abbildung 1.2: Szenario für die Verwendung einer MQ

sich die Nachricht einmal in der Warteschlange, so ist sichergestellt, daß sie auch weiterverarbeitet wird, weil der *get*-Aufruf immer im Rahmen einer Transaktion erfolgt. Bricht wegen eines Netzwerkfehlers die Verarbeitung ab, so verbleibt die Nachricht in der Warteschlange. Zu einem späteren Zeitpunkt kann dann erneut versucht werden, die Nachricht zu verarbeiten.

Mit derselben Architektur lassen sich ganze Workflow-Ketten aufbauen, bei denen mehrere Prozesse mit MQs hintereinandergeschaltet werden.

1.3 Vorteile des Message Queuing

Asynchronität: Daten können als Nachrichten in der Message Queue zwischengespeichert werden, um von einem nachfolgenden Prozess zu einem beliebigen Zeitpunkt entnommen und weiterverarbeitet zu werden. Damit sind Erzeuger und Empfänger der Nachricht zeitlich entkoppelt. Ein Beispiel für diese Art der Anwendung ist

der Email-Eingangsordner.

Zuverlässigkeit: Falls in einem Netzwerk Knoten oder Kommunikationskanäle unzuverlässig sind, so kann eine Nachricht in einer Message Queue auf dem System, auf dem sie erzeugt wurde, abgelegt werden. Von dort wird sie dann im Rahmen einer Transaktion an ein anderes System weitergereicht und verarbeitet. Auf diese Art wird sichergestellt, daß die Nachricht garantiert verarbeitet wird und nicht verloren geht. Dies ist die Art der Anwendung, wie sie im vorliegenden Fall für Mobile Agenten vorgesehen ist.

Lange Transaktionen: Wird ein Funktionsaufruf als Nachricht vom Client zum Server weitergeleitet, so braucht der Client nicht zu warten, bis die Transaktion, in deren Rahmen der Aufruf geschieht, abgelaufen ist. Dies ist besonders praktisch für sehr lange laufende Transaktionen. Trotzdem kann der Client sicher sein, daß sein Aufruf genau einmal bearbeitet wird, da die Nachricht ebenfalls im Rahmen einer Transaktion aus der Queue entnommen und verarbeitet wird.

Kopplung verschiedener Systeme: Die direkte Kopplung verschiedener Software-Systeme ist unter Umständen ein schwer zu bewerkstellendes oder unmögliches Unterfangen. Andererseits kann es oftmals einfacher sein, für ein System einen Ausgabe- bzw. Eingabefilter zu implementieren, der Daten in Form einer Nachricht an eine Queue weitergibt beziehungsweise von einer Queue entgegennimmt. So kann beispielsweise die Grenze zwischen unterschiedlichen Plattformen recht einfach überwunden werden, wenn auf beiden Seiten eine Message Queue zur Verfügung steht.

1.4 Überblick über kommerzielle Produkte

Auf dem Markt befinden sich etliche Systeme, von denen hier in aller Kürze zwei vorgestellt werden.

1.4.1 BEA Tuxedo/Q

Tuxedo/Q ist Teil der unter dem Namen Tuxedo vertriebenen Middleware-Plattform von BEA Systems, Inc. Tuxedo/Q stellt neben den beiden Primitiven *enqueue()* und *dequeue()* noch einen Store-and-Forward-Service zur Verfügung. Dieser Dienst reicht Nachrichten, die von der Queue entgegengenommen wurden, an einen Tuxedo-Server weiter, nimmt die Antwort des Servers entgegen und reicht diese zurück an die Queue. Dadurch braucht bei der Implementierung des Servers keine Rücksicht darauf genommen werden, ob die Aufrufe direkt vom Client oder mittels einer Queue geschehen.

Tuxedo/Q bietet wie die meisten kommerziellen Produkte einen “Thin Client”-Ansatz. Damit braucht auf dem Client-Rechner kein eigener Queue Manager installiert werden, sondern nur ein kleiner “Queue Proxy”, der Nachrichten vom Client-Programm entgegennimmt und zur physikalischen Speicherung an die tatsächliche Queue weiterleitet. Die Transaktionsverwaltung für Tuxedo/Q übernimmt der in der Tuxedo-Suite enthaltene Transaktionsmanager.

1.4.2 IBM MQSeries

IBM reklamiert für sich die Marktführung im Bereich des Message Queuing. Entsprechend umfangreich und mächtig ist auch das Produkt, das angeboten wird: neben einfachem Message Queuing und Store-and-Forward beherrscht MQSeries auch das Weiterleiten (Routing) von Messages anhand von Regeln, es stellt Trigger zur Verfügung, die beim Eintreten von bestimmten Bedingungen Programme starten, um Nachrichten zu verarbeiten, und vieles mehr. Daneben ist MQSeries auf rund 35 Plattformen verfügbar und bietet so eine nahezu universelle Möglichkeit, Applikationen in heterogenen Systemen zu integrieren.

Kapitel 2

Anwendungsgebiet Mobile Agenten

Ein Mobiler Agent ist ein Software-Objekt, das aus eigener Initiative aufgrund von vor Ort getroffenen Entscheidungen den Ort seiner Ausführung von einem Netzknoten (Computer) auf einen anderen verlegen kann (Migration). Dazu muß der Code des Agenten samt dem momentanen Ausführungszustand in eine transportable Form umgewandelt und über das Netzwerk zu dem Zielknoten übertragen werden. Dort wird der Agent empfangen, entpackt und die Ausführung fortgesetzt.

Ein Agentensystem besteht hauptsächlich aus zwei Teilen. Zum einen sind dies die Agenten selbst, zum anderen die Ausführungsumgebung, der sogenannte *Platz*. Aufgabe dieses Platzes ist es, die Ausführung und die Migration eines Agenten zu ermöglichen. Der Platz muß auf allen beteiligten Netzknoten installiert sein.

2.1 Exactly-Once-Ausführung für Mobile Agenten

Für die korrekte Ausführung der dem Agenten anvertrauten Aufgaben wird oftmals vorausgesetzt, daß der Agent jede notwendige Operation genau einmal ausführt. Besonders wichtig ist diese Exactly-Once-Eigenschaft immer dann, wenn es um Geld geht: beispielsweise, wenn Flüge gebucht oder Konzertkarten gekauft werden. Es wäre nicht akzeptabel, wenn aufgrund eines Knoten- oder Netzwerkfehlers der Agent eine Reservierung doppelt durchführen würde und dann auch die doppelten Kosten auf der Rechnung erschienen. Genausowenig darf der Agent überhaupt nicht ausgeführt werden, weil er verloren gegangen ist.

Ein primitiver Ansatz für die Exactly-Once-Ausführung wird im folgenden beschrieben (siehe Abb. 2.1). Angenommen, ein Agent soll n Knoten K_1 bis K_n besuchen und dort jeweils eine Aufgabe ausführen. (Ob die Reiseroute im voraus festgelegt ist oder sich erst im Rahmen der Ausführung ergibt, ist unerheblich.) Es wird vorausgesetzt, daß sich auf jedem der n Knoten eine Instanz einer transaktionalen Message Queue befindet. Nun wird innerhalb einer Transaktion der Agent aus der Message Queue des Knotens K_m

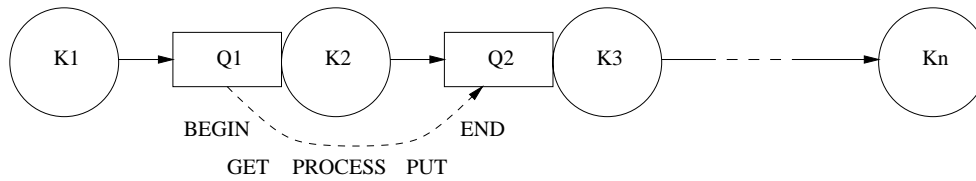


Abbildung 2.1: Primitives Exactly-Once-Protokoll

entnommen, auf dem Knoten ausgeführt und dann an die Message Queue des Knotens K_{m+1} weitergereicht. Tritt während der Verarbeitung auf K_m ein Fehler auf, der den erfolgreichen Abschluß der Transaktion verhindert, so verbleibt der Agent in der Queue von K_m . Falls die Transaktion erfolgreich beendet werden konnte, so ist der Agent einen Schritt weiter. Wird davon ausgegangen, daß jeder Knotenfehler nur endlich lange dauert, so ist mit diesem Ausführungsmodell garantiert, daß der Agent nach endlicher Zeit die Knoten K_1 bis K_n besucht hat und auf jedem Knoten genau ein mal ausgeführt wurde (vorausgesetzt, daß alle Arbeiten, die der Agent auf einem Knoten ausführt, ebenfalls im Rahmen der umgebenden Transaktion stattfinden).

Dieses Modell hat eine große Schwäche. Befindet sich der Agent auf dem Knoten K_m und der Knoten K_{m+1} ist ausgefallen, so sitzt der Agent vorerst fest, bis der Fehler behoben ist. Es könnte nun aber sein, daß ein Knoten K'_{m+1} existiert, der dieselbe Aufgabe wie K_{m+1} übernehmen könnte. Trotzdem wird dieser Ersatzknoten im einfachen Exactly-Once-Modell nicht verwendet.

Es existiert ein wesentlich verbesserter Ansatz, bei dem eine Reiseroute nicht durch einzelne Knoten, die nacheinander zu besuchen sind, festgelegt wird, sondern durch Stufen, die jeweils aus einer Menge von Knoten bestehen. Der Agent muß nun aus jeder Stufe einen Knoten besuchen. Natürlich ist hier zusätzlicher Aufwand erforderlich, um sicherzustellen, daß der Agent tatsächlich pro Stufe nur genau einmal ausgeführt wird. Dieser Aufwand spiegelt sich in einer ganzen Reihe von Protokollen wieder. Eine Beschreibung würde den Rahmen dieser Arbeit sprengen, an dieser Stelle sei auf [8] und [10] verwiesen.

Auch dieser Ansatz erfordert als wesentlichen Teil des Systems eine transaktionale Message Queue, die auf jedem Knoten verfügbar sein muß.

2.2 Anforderungen an die Implementierung

Für die Verwendung im Exactly-Once-Protokoll ergeben sich verschiedene Anforderungen an die Message Queue.

Performance: Da es sich bei der MQ um eine zentrale Komponente des Systems handelt, kann bei einer langsamen Implementierung leicht ein Flaschenhals entstehen. Es ist also darauf zu achten, daß die MQ in der Lage ist, mit dem Datendurchsatz des

Gesamtsystems mitzuhalten.

Schlankheit: Der “Thin Client”-Ansatz (s. Kap. 1.4.1) ist im Umfeld eines Mobile Agenten-Systems nur schlecht zu verwenden, da davon ausgegangen werden muß, daß die Agenten über ein sehr weitläufiges Netz (Internet) wandern müssen. Hier ist eine zentrale Komponente zur Speicherung der Daten schlecht verwendbar.

Es soll deswegen auf jedem Knoten eine voll funktionale Instanz der Message Queue verfügbar sein. Das bedeutet, daß die Implementierung so schonend wie möglich mit den Ressourcen umgehen sollte.

Kapitel 3

Die Entwicklungsumgebung

Zur Entwicklung der MQ stand die CORBA-Implementierung “Orbix” von IONA Technologies zur Verfügung. Zwar sind auf dem Markt der reinen Object Request Broker noch unzählige Alternativen verfügbar, von denen einige sogar frei sind, so zum Beispiel der für Forschung und privaten Einsatz kostenlose Orbacus, bei dem auch der Quelltext mitgeliefert wird. Da jedoch auch ein Transaktionsmonitor benötigt wurde, mußte die Wahl auf einen ORB fallen, für den ein Object Transaction Service zu haben war. Hier war am IPVR eine Installation von OrbixOTS vorhanden, so daß trotz der Schwächen des Produktes ein Umstieg auf eine andere Implementierung wegen der zu hohen Kosten nicht möglich war.

Damit war dann auch die Implementierungssprache vorgegeben. Orbix bietet ein IDL-Mapping und Laufzeitbibliotheken für C/C++ sowie Java an. Allerdings können transaktionale Server nur in C++ geschrieben werden.

3.1 CORBA

Die von der Object Management Group (OMG) vorgestellte Common Object Request Broker Architecture (CORBA) definiert ein Gerüst für die Entwicklung verteilter objekt-orientierter Anwendungen. Zentrale Komponente ist der Object Request Broker (ORB), der Methodenaufrufe eines Clients über das Netzwerk zu einem Server leitet und die Ergebnisse zurück zum Client transportiert. Damit wird die Komplexität der Netzwerkprogrammierung vor dem Programmierer verborgen.

CORBA wurde auch im Hinblick auf heterogene Umgebungen spezifiziert. Schnittstellen werden in einer speziellen Programmiersprache, der Interface Definition Language (IDL) beschrieben. CORBA IDL baut auf einer C++-ähnlichen Syntax auf. Sie bietet neben primitiven Datentypen wie den verschiedenen Integer-, Fließkomma- und Character-Typen die Möglichkeit, zusammengesetzte Typen (*struct*, *union*), Aufzählungstypen (*enum*), Arrays und Sequenzen zu beschreiben. Dazu kommt die Beschreibung der

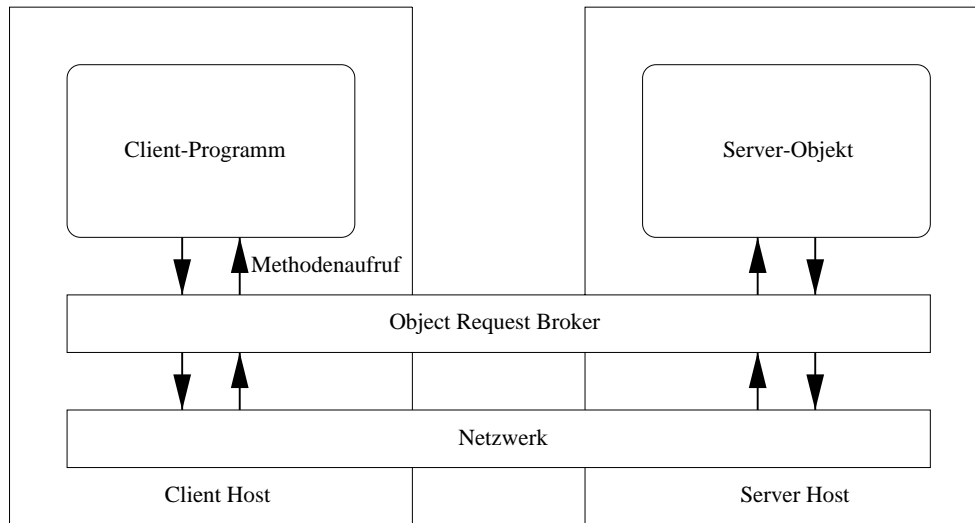


Abbildung 3.1: Methodenaufruf über CORBA

Schnittstellen (*interface*) selbst. Eine Schnittstelle definiert ein Objekt, dessen Methoden mittels CORBA aufgerufen werden können. In CORBA IDL können lediglich die Namen und Signaturen der Methoden angegeben werden, es gibt keinerlei Möglichkeit, die Funktionalität einer Schnittstelle zu beschreiben. Das ist die Aufgabe der Sprache, in der ein CORBA-Objekt dann implementiert wird.

Es existieren Abbildungen von IDL auf die verschiedensten Sprachen (“language mapping”). In diesen Mappings wird festgelegt, wie IDL-Datentypen und Schnittstellen auf die Sprache abgebildet werden.

Um eine CORBA-Anwendung zu erzeugen, sind folgende Schritte nötig: Zuerst wird die Schnittstelle der Server-Objekte in IDL spezifiziert. Dann erzeugt ein spezieller IDL-Compiler aus dem IDL-File einen *Client Stub* und ein *Server Skeleton*. Der Client Stub wird in den Client eingebunden und stellt alle im Interface spezifizierten Methoden zur Verfügung. Wird eine dieser Methoden aufgerufen, so serialisiert der Stub die Argumente und schickt sie durch den ORB an den Server. Im Server nimmt das Skeleton den Methodenaufruf entgegen, erzeugt aus dem seriellen Datenstrom wieder die ursprünglichen Argumente und ruft die vom Programmierer bereitzustellende Implementierung der Methode auf.

Damit sind die Methodenaufrufe im Client tatsächlich lokale Aufrufe, und auch die Methoden im Server können so implementiert werden, als ob sie nur lokal aufgerufen würden. Die Verteilung der Anwendung wird also fast komplett vor dem Programmierer verborgen. “Fast” deshalb, weil zur Erzeugung eines Objektes nicht einfach ein *new*-Operator verwendet werden kann, vielmehr muß hier eine Instanz des Objekts im Netzwerk lokalisiert werden (zu diesem Zweck z. B. Namensdienste zur Verfügung), sodann muß das lokal erzeugte Proxy-Objekt an das entfernte Objekt gebunden werden.

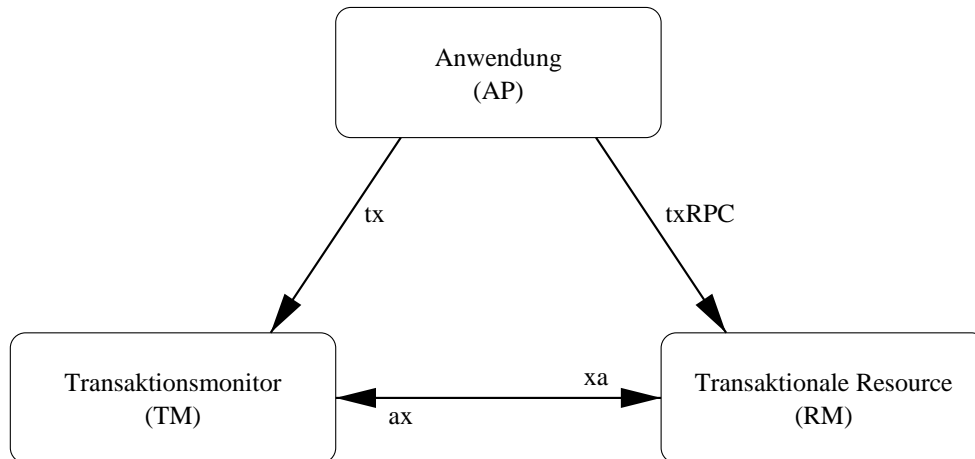


Abbildung 3.2: DTP-Referenzmodell

3.2 CORBA Services

Neben der Basisspezifikation von CORBA hat die OMG noch eine ganze Reihe von zusätzlichen Diensten für CORBA beschrieben. Diese CORBA Services erweitern die Kernarchitektur um eine Reihe von Diensten, die allgemein in CORBA-Anwendungen benötigt werden. Dazu gehören unter anderem Namensdienste, Dienste zur Verwaltung von Transaktionen, Ereignissen und viele andere. Die Dienste werden als CORBA-Server implementiert, von der OMG sind das IDL-Interface und – soweit nötig – das Verhalten vorgegeben. Im Rahmen dieser Arbeit wurden der Object Transaction Service (OTS) und der Event Service (ES) verwendet. Bei der verwendeten Implementierung handelte es sich ebenfalls um die Orbix-Produkte der Firma IONA.

3.2.1 Object Transaction Service

Zur Verwaltung verteilter Transaktionen dienen seit langem Transaktionsmonitore, die nach dem Distributed Transaction Processing (DTP)-Referenzmodell der X/Open Company arbeiten (Abbildung 3.2). Hierbei initiiert die Anwendung eine Transaktion, indem sie über die tx-Schnittstelle den Transaktionsmonitor (TM) aufruft. Dieser erzeugt einen Transaktionskontext. Nun kann die Anwendung über transaktionale Remote Procedure Calls (txRPC) Daten in transaktionalen Ressourcen modifizieren. Bei jedem txRPC-Aufruf wird der anfangs erzeugte Kontext implizit mitübergeben. Eine transaktionale Resource, die am Zwei-Phasen-Commit-(2PC)-Protokoll teilnehmen will, muß im Rahmen der Transaktion beim Transaktionsmonitor registriert werden. Dies kann entweder die Anwendung (über die tx-Schnittstelle) oder die Resource selbst (über die ax-Schnittstelle) übernehmen.

Das 2PC-Protokoll wird eingeleitet, wenn die Anwendung über das tx-Interface einen

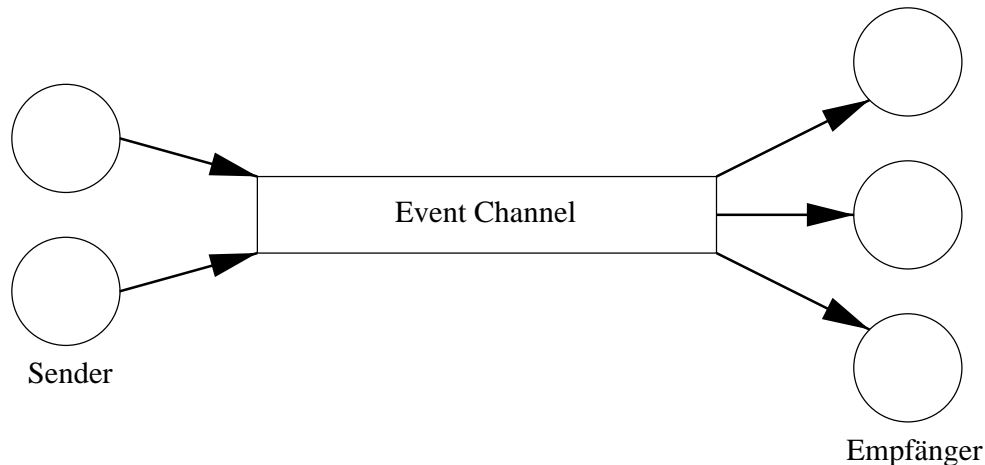


Abbildung 3.3: CORBA Event Service

commit-Aufruf an den Transaktionsmonitor sendet. Der TM ruft dann für jede registrierte Resource *xa_prepare()* und *xa_commit()* auf.

Dieses weitverbreitete Modell der verteilten Transaktionsverarbeitung wurde für die Verwendung in der objektorientierten CORBA-Umgebung überarbeitet. Die prozeduralen tx- und xa-Schnittstellen wurden durch CORBA-Interfaces gekapselt. Eine transaktionale Resource wird als CORBA-Objekt implementiert. Dabei kann das IDL-Interface wahlweise von einem transaktionalen Interface (`CorbaTransactions::TransactionalObject`) abgeleitet werden, in diesem Fall sind alle Methoden transaktional und der Transaktionskontext wird implizit propagiert, oder den transaktionalen Methoden muß ein zusätzliches Argument vom Typ `CorbaTransactions::Control` übergeben werden, das den Kontext enthält.

Der Object Transaction Service enthält weiterhin auch das xa-Interface, so daß X/Open-DTP konforme Ressourcen ohne Probleme in eine transaktionale CORBA-Umgebung eingebunden werden können.

3.2.2 Event Service

Der CORBA Event Service definiert ein Modell für die indirekte Kommunikation zwischen ORB-Anwendungen. Anstelle von direkten Methodenaufrufen wird vom Sender ein Ereignis (Event) erzeugt, das er mit Hilfe eines Ereignis-Kanals an eine beliebige Anzahl von Empfängern weitergeben kann. Die Übertragung eines Events kann entweder vom Sender (*Push-Model*) oder vom Empfänger (*Pull-Model*) angestoßen werden. Allerdings implementiert OrbixEvents nur das Push-Model.

Das Herzstück des Event Service ist der Event Channel. Er stellt ein Proxy-Objekt dar: aus Sicht des Empfängers erscheint er als Sender, aus Sicht des Senders als Empfänger.

Die Kommunikation mit dem Event Channel funktioniert per CORBA-Methodenaufruf. Will also ein Sender ein Objekt verschicken, so ruft er die *push()*-Methode des Event Channels auf.

Ein Empfänger muß ebenfalls die *push()*-Methode implementieren und sich dann beim Event Channel registrieren. Erhält der Channel nun ein Event, so ruft er die *push()*-Methode der registrierten Empfänger auf und reicht das Event so weiter. Dies bedeutet, daß ein Empfänger im Prinzip ein Server-Objekt ist, da seine Methoden per CORBA aufgerufen werden.

Der Event Service definiert typisierte und nicht typisierte Kommunikation. Die nicht typisierte (*untyped*) Event-Kommunikation ist recht einfach: eine Nachricht wird einfach in den *ANY*-Datentyp verpackt und an den Event Channel gesandt. Dieser reicht die Nachricht weiter an den Empfänger. Der Empfänger muß dann in der Lage sein, die Nachricht wieder auszupacken und zu interpretieren. Dieses Modell ist sehr einfach, allerdings funktioniert es wegen der in Kapitel 4.1.3 erwähnten Schwäche des *ANY*-Datentyps in Orbix nur sehr eingeschränkt, d. h. mit den primitiven CORBA-Datentypen. Dies ist für den vorliegenden Anwendungsfall jedoch ausreichend.

Bei der typisierten Event-Kommunikation wird für den Empfänger ein Interface definiert, das von `CostypedEventComm::TypedPushConsumer` abgeleitet ist. Namen und Signaturen der Methoden sind frei wählbar. Der Empfänger registriert sich dann beim Event Channel und übergibt eine Objektreferenz auf eine Implementierung dieses Interfaces. Der Event Channel dient damit als Proxy für das vom Empfänger implementierte Server-Objekt. Eine Nachricht besteht aus einem Methodenaufruf, den der Event Channel vom Sender entgegennimmt und an den Empfänger weiterleitet.

Kapitel 4

Interfaces und Datentypen

Obwohl die Message Queue für einen ganz speziellen Einsatzbereich vorgesehen war, sollte dennoch eine möglichst universelle Implementierung geliefert werden. Damit war es nicht möglich, irgendwelche Vereinfachungen und Optimierungen einzubauen, die auf besonderen Eigenschaften des Exactly-Once-Protokolls basieren.

Neben den grundlegenden Queuing-Primitiven *put* und *get* stellt die vorliegende Implementierung eine Reihe weiterer Eigenschaften zur Verfügung:

- Die MQ besitzt drei Schnittstellen in Form von CORBA-Interfaces. Mittels der ersten Schnittstelle (*interface Mq_Standard*) kann wahlfrei auf in der Queue befindliche Nachrichten zugegriffen werden. Die Identifizierung erfolgt mit Hilfe eines Schlüssels, der von der Queue beim Einstellen einer Nachricht erzeugt wird. Die zweite Schnittstelle (*interface Mq_Fifo*) realisiert das First-In-First-Out-(FIFO)-Paradigma. Hier wird bei der *get*-Operation immer die nächste verfügbare Nachricht ausgegeben. Die interne Sortierung der Nachrichten erfolgt wahlweise nach dem Zeitstempel oder nach der Priorität der Nachricht.

Das dritte Interface (*interface Mq_Service*) dient zur Administration der Warteschlange.

- Damit ein Abnehmer von Nachrichten (Consumer-Client) nicht ständig bei der MQ nachfragen muß, ob neue Nachrichten eingegangen sind (“polling” oder “busy waiting”), wird beim Eintreffen einer Nachricht ein Ereignis erzeugt, das an alle registrierten Consumer weitergegeben wird. Zur Propagation des Ereignisses wird der CORBA Event Service genutzt. Hier ist eine Filterung von Nachrichten möglich: ein Consumer kann bei seiner Registrierung eine Gruppenzugehörigkeit angeben und erhält dann nur ein Event, wenn die eingegangene Nachricht die entsprechende Gruppennummer trägt.
- Um beliebige weitere (Meta-)Informationen mit einer Nachricht mitzuliefern, wurde der Nachricht ein Satz von Attributen beigelegt. Ein Attribut besteht aus einem

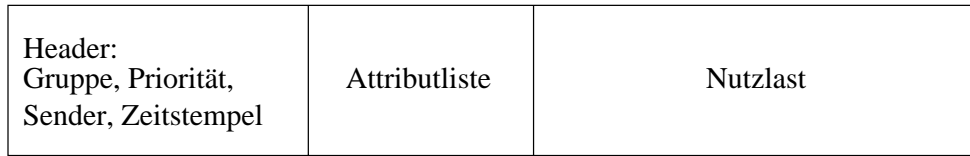


Abbildung 4.1: Aufbau einer Nachricht

Schlüssel und einem Wert, die beide Strings beliebiger Länge sind. Auch hier besteht die Möglichkeit zur Filterung von Nachrichten, vorerst einmal auf Seite des Consumers, der Nachrichten anhand von Attributen bearbeiten oder verwerfen kann. Aber auch auf Seite der MQ könnte man hier einen Filter implementieren, der eine große Flexibilität mitbringen würde.

4.1 Aufbau einer Nachricht

Eine Nachricht (s. Abb. 4.1) besteht in der vorliegenden Implementierung aus drei Teilen, einem *Header*, einer *Liste von Attributen* und der *Nutzlast*. Die für diese drei Teile nötigen Datentypen wurden in CORBA IDL definiert. Die entsprechenden Listings sind im Anhang abgedruckt.

4.1.1 Header

Im Header sind alle notwendigen Verwaltungsinformationen wie Priorität, Gruppennummer und Zeitstempel zusammengefaßt. In Listing A.1 ist die IDL-Definition abgedruckt.

priority bezeichnet die Priorität der Nachricht. Die Priorität ist ein ganzzahliger 16bit-Wert ohne Vorzeichen. Höhere Werte kennzeichnen eine größere Priorität. Dieses Feld wird nur genutzt, wenn die MQ als FIFO betrieben wird, da im Standard-Modus ohnehin wahlfrei auf die Nachrichten zugegriffen werden kann.

groupID gibt an, an welche Gruppen beim Eintreffen der Nachricht ein Ereignis gesandt werden soll. Auch hier wird ein ganzzahliger Wert zwischen 0 und 65536 (16bit) verwendet. Der Wert 0 ist ein besonderer Wert: registriert sich ein Consumer mit der Gruppen-ID 0, so erhält er bei jeder eingegangenen Nachricht ein Event. Wenn eine Nachricht die Gruppennummer 0 trägt, so werden alle registrierten Clients über das Eintreffen benachrichtigt.

time ist der Zeitstempel der Nachricht. Dieser wird von der MQ selbst ausgefüllt, er kann also vom Producer-Client leergelassen werden. Als Zeitformat wird dasjenige von UNIX verwendet: es werden Sekunden und Microsekunden seit dem 1.1.1970 gezählt.

4.1.2 Attribute

Die Attributliste ist eine Sequenz von Schlüssel-Wert-Paaren, auf die mit Hilfe spezieller Funktionen der Message Queue zugegriffen werden kann. Listing A.2 gibt die IDL-Definition wieder.

4.1.3 Nutzlast

Der Datentyp für die Nutzlast, ein Byte-Array unbegrenzter Länge, wird in IDL folgendermaßen definiert:

```
typedef sequence<octet> mqPayload;
```

Eigentlich sollte hier der spezielle CORBA-Datentyp *Any* verwendet werden. Leider erwies sich nach einem mehrtägigen Test, daß die verfügbare CORBA-Implementierung (Orbix von IONA) neben einigen andern Schwächen auch einen fehlerhaften Any-Datentyp aufwies. Das Problem war, daß in einem Any nicht ein beliebiger anderer CORBA-Datentyp gekapselt werden konnte, sondern nur die primitiven Typen wie short, long, float, char und so weiter; sowie zusammengesetzte Datentypen, die in demselben IDL-Modul definiert waren, in dem auch die Funktion definiert wurde, die die Any-Argumente verwendete. Das würde bedeuten, daß jedesmal, wenn ein neuer zusammengesetzter Datentyp mit der MQ verwendet werden sollte, das IDL-Interface der MQ zu modifizieren wäre, sowie die MQ neu kompiliert und gelinkt werden müßte. Aus diesem Grund wurde auf den Einsatz von Any verzichtet. Dies bedeutet jedoch keine nennenswerte Einschränkung, da schließlich (unter Umständen zwar mit etwas Aufwand) jeder Datentyp in ein Byte-Array konvertiert werden kann.

4.2 Schnittstellen zur MQ

Nachdem nun alle nötigen Datentypen vorgestellt wurden, werden in diesem Abschnitt die Schnittstellen zur Message Queue beschrieben.

4.2.1 Standard-Interface

Listing A.4 gibt die Definition des Standard-Interfaces in CORBA IDL wieder. Die beiden Methoden `mqPut()` und `mqGet()` implementieren die grundlegenden Funktionen einer Message Queue. Für die Transaktionen wird explizite Propagation verwendet, weil das Interface auch noch vier Methoden enthält, die nicht transaktional zu sein brauchen. Wäre das Interface von `CosTransactions::TransactionalObject` abgeleitet, so würde auch bei diesen Methoden immer der Transaktionskontext mit übergeben, was einen unnötigen Overhead mitbrächte.

Der einzige *out*-Parameter der `mqPut`-Methode ist der eindeutige Schlüssel, den die MQ für diese Nachricht erzeugt. Dieser Schlüssel wird aus der Systemzeit und einem Zähler generiert. Falls die Systemzeit geändert wird, können also Schlüssel doppelt auftreten. Die `mqGet`()-Methode liefert die durch den Schlüssel bezeichnete Nachricht zurück; falls die Transaktion erfolgreich abgeschlossen wird, wird die Nachricht aus der Queue gelöscht.

Im Gegensatz zu den ersten beiden Methoden sind `mqRead`() und `mqRemove`() nicht transaktional. Diese Methoden sind Überbleibsel von der ursprünglichen MQ, die im Rahmen der Implementierung des Exactly-Once-Protokolls (EOP, [2]) entstand. Da das EOP momentan noch diese Funktionen benötigt, wurden sie in die neue MQ mit übernommen. Anders als `mqGet`() löscht `mqRead`() die Nachricht nach dem Lesen nicht aus der Message Queue. `mqRemove`() entfernt eine Nachricht aus der Queue. Zwar läuft die Operation nicht im Rahmen einer Transaktion ab, jedoch wird auch hier durch Logging- und Recovery-Mechanismen ein konsistenter Zustand der Queue nach einem Fehler garantiert.

Ebenfalls ohne Transaktionseigenschaften kommt die Methode `mqGetAttribList`() aus, da es durchaus akzeptabel ist, wenn hier Daten gelesen werden, die noch nicht “committed” sind (dirty read).

Über die Methode `mqStartNotify`() schließlich kann sich ein Client bei der MQ als Abnehmer von Nachrichten registrieren. Als Parameter müssen der Name des Event Channels, über den die Notification empfangen werden soll, sowie die Gruppenzugehörigkeit angegeben werden. Die MQ sendet direkt nach der Registrierung eine Benachrichtigung über alle bereits in der Queue befindlichen Nachrichten an den neuen Consumer. Danach wird immer beim Eintreffen einer neuen Nachricht ein Event gesandt, das den Schlüssel beinhaltet, der zum Abholen der Nachricht benötigt wird.

Da diese Notification erst in der Commit-Phase der Transaktion geschieht, ist somit auch sichergestellt, daß mit der nichttransaktionalen Methode `mqRead`() keine Nachrichten gelesen werden, die noch nicht committed sind.

Die `QueueDisabled`-Ausnahme wird erhoben, wenn eine Methode aufgerufen wird, solange sich die Queue im ausgeschalteten Zustand befindet.

4.2.2 FIFO-Interface

In Listing A.3 ist die Definition des FIFO-Interfaces in CORBA-IDL gegeben.

Die `mqPut`()-Operation ist fast identisch mit derjenigen des Standard-Interfaces, nur wird hier kein Schlüssel zurückgegeben. Damit ist ein direkter Zugriff auf die Nachricht nicht mehr möglich. Außerdem sortiert `mqPut`() die Nachrichten gleich entsprechend ihrer Priorität, so daß die *get*-Operation immer nur auf die erste Nachricht der Schlange zugreifen muß.

`mqGetNext`() gibt die nächste verfügbare Nachricht in der Warteschlange zurück. Dabei bedeutet verfügbar, daß die Nachricht sofort lesbar ist. Falls die erste Nachricht in der

Queue gerade gesperrt ist, weil darauf zugegriffen wird, so wird die zweite Nachricht zurückgeliefert. Wird die Transaktion, die auf die erste Nachricht zugreift, nicht erfolgreich abgeschlossen, so verbleibt die Nachricht in der Queue, so daß es unter Umständen passieren kann, daß das FIFO-Paradigma nicht hundertprozentig erfüllt wird.

Die Methode `mqStartNotify()` erfüllt hier genau den gleichen Zweck wie im Standard-Interface, nämlich einen Consumer über das Eintreffen einer neuen Nachricht zu informieren. Aus Gründen der Einfachheit wird auch hier der Schlüssel der Nachricht versandt, er findet aber beim Consumer keine Verwendung.

Die Funktion `mqSetPriorization()` schließlich dient dazu, die Auswertung des Prioritätsfeldes einer Nachricht ein- oder auszuschalten. Bei ausgeschalteter Priorisierung werden Nachrichten nur nach dem Zeitstempel sortiert, d. h. jede neue Nachricht wird der Warteschlange hinten angehängt. Ist die Priorisierung eingeschaltet, so werden Nachrichten mit höherer Priorität vor solchen mit niedrigerer eingefügt. Bei gleicher Priorität ist wiederum der Zeitpunkt des Eintreffens das Sortierkriterium. `mqGetPriorization()` gibt Auskunft darüber, ob die Priorisierung gerade ein- oder ausgeschaltet ist.

4.2.3 Service-Interface

Das Service-Interface schließlich stellt vier Methoden zur Verfügung, mit deren Hilfe die Funktion der MQ gesteuert und überwacht werden kann.

`mqShutdown()` beendet das System ordentlich. Alle noch anstehenden Schreibvorgänge werden durchgeführt, dann wird die Queue heruntergefahren. Dabei werden noch laufende Transaktionen abgebrochen.

`setEnabled()` schaltet die Queue ein oder aus, je nach dem ob der übergebene Parameter "wahr" oder "falsch" ist. Im ausgeschalteten Zustand funktioniert nur das Service-Interface, die beiden anderen Schnittstellen erheben eine Ausnahme (*QueueDisabled*).

`isEnabled()` gibt Auskunft darüber, ob die Queue gerade ein- oder ausgeschaltet ist.

`getStatus()` schließlich liefert den Zustand der Queue in Form eines Strings zurück. Hier wird eine Liste mit momentan aktiven Transaktionen sowie die Anzahl der gerade in der Queue befindlichen Nachrichten ausgegeben.

Kapitel 5

Internes Design der MQ

Nachdem mit der Definition der Datentypen und Schnittstellen festgelegt ist, wie sich die MQ nach außen darstellen soll, folgt nun der Entwurf des inneren Aufbaus. Dabei wurde Wert darauf gelegt, die Software in möglichst generische und damit gut wiederverwendbare Module zu zergliedern.

Um die Entwicklung in zwei Stufen vollziehen zu können, wurde folgende Vorgehensweise gewählt: im ersten Schritt sollte die Speicherung der Nachrichten von einem relationalen Datenbanksystem (RDBMS) übernommen werden. Zur Verfügung stand eine Installation von IBM DB2, Version 5.0. Wichtig war, daß das RDBMS über das XA-Interface verfügte, damit es ohne Probleme in die OrbixOTS-Umgebung eingebunden werden konnte. Dies trifft für DB2 zu. Damit konnte sich die Entwicklung im ersten Schritt auf die Implementierung der Schnittstellen zur Außenwelt beschränken.

In der zweiten Stufe sollten dann Module zur Speicherung der Nachrichten entwickelt werden, die das RDBMS ersetzen sollten.

5.1 Erster Ansatz: Verwendung eines RDBMS

Abbildung 5.1 zeigt im Überblick, wie das Datenbanksystem in die Anwendung eingebunden wird. Beim Start des Servers wird das DBMS als XA-fähiger Resource Manager beim Object Transaction Service registriert. Dies muß geschehen, bevor die `init()`-Methode des OTS aufgerufen wird, weil hier der OTS seinen Restart durchführt und dabei eventuell unvollständige Transaktionen komplettiert. Dazu werden natürlich alle Resource Manager benötigt, da das 2PC-Protokoll für diese Transaktionen erneut durchgeführt werden muß.

Beginnt der Client eine Transaktion, so erzeugt der OTS einen Transaktionskontext (Objekt vom Typ `CoTransactions::Control`). Danach kann der Client die transaktionalen Methoden `mqPut()` und `mqGet()` des Servers aufrufen. Der Server konvertiert die erhaltenen Daten in ein internes Format und fügt sie per SQL INSERT in die Da-

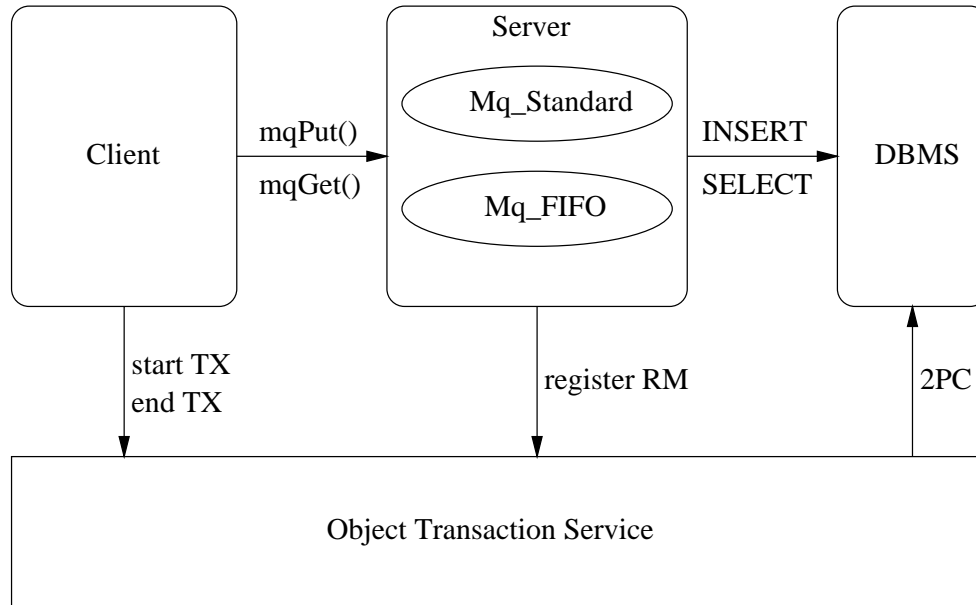


Abbildung 5.1: Verwendung eines DBMS zur Speicherung der Messages

tenbank ein, beziehungsweise liest er die zum erhaltenen Schlüssel gehörigen Daten per SELECT aus, erzeugt daraus wieder die externen Datentypen und gibt diese zurück.

Beendet der Client die Transaktion, so führt der Transaktionsmonitor mit allen registrierten Resource Managern das 2 Phasen Commit Protokoll aus.

Dieser Ansatz ist relativ einfach zu implementieren, weil sich der Server garnicht um die Abwicklung der Transaktionen kümmern muß. Er reicht im Prinzip nur Daten vom Client an das DBMS und vom DBMS an den Client weiter, wobei noch die Konvertierung vom externen in das interne Format zu erledigen ist.

Logisches Datenbankdesign

Das Datenbankdesign für diese Anwendung ist trivial: es gibt nur zwei Entitäten, *Message* und *Attribut*, als primärer Schlüssel wird die Message-ID verwendet, weil diese ohnehin eindeutig sein muß. *Attribut* besitzt keinen eigenen Primärschlüssel, was bedeutet, daß dasselbe Key/Value-Paar mehrfach bei einer Nachricht auftreten darf. Ob es sinnvoll ist, dies zu nutzen, muß von Anwendungsfall zu Anwendungsfall entschieden werden.

Das ER-Diagramm in Abb. 5.2 läßt sich dann auf die Tabellenstruktur in Abb. 5.4 zurückführen. Zur Erzeugung der Tabellen wurden die SQL-Anweisungen in Listing 5.3 verwendet.

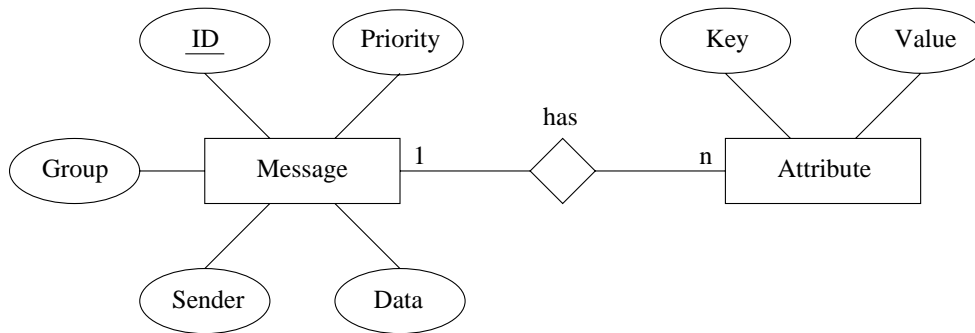


Abbildung 5.2: ER-Diagramm für das Datenmodell der MQ

```

create table messages
(
  msg_id      char(15) primary key not null,
  priority    integer not null,
  group_id    integer not null,
  sender      varchar(254),
  data        blob(2M)
);

create table attributes
(
  msg_id      char(13) not null,
  key         varchar(254),
  value       varchar(254),
  foreign key (msg_id) references messages (msg_id)
  on delete cascade
);
  
```

Listing 5.3: Erzeugung der Tabellen in DB2

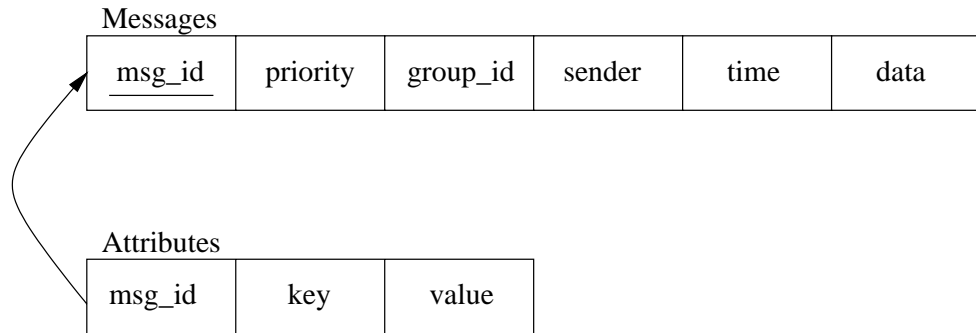


Abbildung 5.4: Datenbankschema für die MQ

Bewertung

Dieser Ansatz ist mit demselben Nachteil behaftet wie eine Message Queue mit “Thin Clients” und einem zentralen Resource Manager zum Speichern der Daten. Auch hier gibt es eine zentrale Instanz des DBMS, die für die physikalische Speicherung der Daten verantwortlich ist. Dies ist für den beabsichtigten Anwendungsfall (EOP für Mobile Agenten) ein besonders ungünstiges Szenario, da Agenten oftmals nicht nur in einem lokalen Netzwerk (LAN), sondern über ein weit verteiltes Netz (Internet) übertragen werden müssen. Hier ist es erstens ungünstig, den Agenten mehrmals über das Netz schicken zu müssen (vom Ausführungsplatz zur MQ des nächsten Knotens, von dort zum DBMS, vom DBMS wieder zurück zur Queue), sondern es wird auch in vielen Fällen nicht möglich sein, von mehreren Knoten aus dasselbe DBMS zur Speicherung zu nutzen. Damit wäre dann im ungünstigsten Fall pro Knoten eine Instanz der Datenbank notwendig.

Trotzdem war es lohnend, bei der Entwicklung der MQ zunächst diesen Ansatz zu verfolgen, da somit das Hauptaugenmerk zunächst auf die Implementierung der Schnittstellen gelegt werden konnte.

Leider hat sich auch hier wieder eine weitere Schwäche von OrbixOTS offenbart. Damit nach dem Eintreffen einer Nachricht den registrierten Clients ein entsprechendes Event zugesandt werden kann, ist es nötig, die MQ über den erfolgreichen Abschluß der Transaktion zu informieren. In dem in Abbildung 5.1 dargestellten Schema wird jedoch deutlich, daß der OTS die Abwicklung des 2PC-Protokolls nur mit den registrierten Resource Managern durchführt. Hier wird der MQ-Server selbst garnicht mit einbezogen. Es gibt nun zwei Möglichkeiten für den Server, den Ausgang der Transaktion zu erfahren: entweder, er erzeugt ein internes Resource-Objekt, das die für die Abwicklung des 2PC-Protokolls notwendigen Methoden `prepare()`, `commit()` und `rollback()` definiert, beim OTS registriert wird und damit auch am Abschluß der Transaktion partizipiert, oder er erzeugt ein *Synchronization*-Objekt, das die Methoden `before_completion()` und `after_completion()` enthält. Dieses Objekt muß dann ebenfalls beim OTS regi-

striert werden. Die erste der beiden Methoden wird aufgerufen, bevor der Abschluß der Transaktion begonnen wird, die zweite Methode nach Beendigung der Transaktion. Dieser Methode wird als Argument das Abschlußergebnis (*COMMIT* oder *ROLLBACK*) mitgeteilt. Das Problem bei OrbixOTS war nun, daß lediglich die `before_completion()`-Methode aufgerufen wurde, nicht aber `after_completion()`.

Aufgrund dieses Fehlers von OrbixOTS und der Tatsache, daß sich der DBMS-Ansatz für den vorgesehenen Anwendungsfall ohnehin nur beschränkt eignet, wurde die Entwicklung der datenbankbasierten MQ abgebrochen, sobald die grundlegende Funktionalität erreicht war, d. h. sobald die Queuing-Primitive *put* und *get* funktionierten.

5.2 Zweiter Ansatz: transaktionale Resource

Um die Speicherung einer Nachricht direkt auf dem Knoten zu ermöglichen, auf dem die Message Queue läuft, war es nötig, einen eigenen Filemanager zu entwickeln. Zudem mußte ein Logmanager entworfen und implementiert werden, der in der Lage war, die Aktionen des Filemanagers und den Zustand der transaktionalen Resource mitzuprotokollieren, damit das Zurücknehmen von Operationen bei einem Rollback oder beim Restart nach einem Crash möglich war.

5.2.1 Programmiertechniken

In diesem Abschnitt sollen zunächst einige der verwendeten Techniken vorgestellt werden, die bei der Implementierung des File- und des Logmanagers zum Einsatz kamen. Vorweg muß erwähnt werden, daß sowohl die Logdatei als auch die Datendatei seitenweise organisiert ist. Dies ist sinnvoll, da das Betriebssystem Daten ohnehin blockweise von der Festplatte liest und auf die Festplatte schreibt. Eine Seite ist als Block fester Länge zu verstehen. Der Hauptteil der Seite steht für Nutzdaten zur Verfügung, einige Bytes sind für Verwaltungsinformationen reserviert. Auf den genauen Aufbau der Seiten wird weiter unten noch näher eingegangen.

Write Check Patterns

Um zu garantieren, daß eine Seite immer korrekt, d. h. vollständig auf stabilen Speicher geschrieben wurde, können mehrere Techniken verwendet werden. Eine Alternative ist, von der Seite eine CRC-Checksumme zu berechnen und diese mit auf die Festplatte zu schreiben. Das hat jedoch den Nachteil, daß die CRC-Algorithmen recht rechenintensiv sind, so daß dadurch eine hohe CPU-Last entstehen kann. Zudem muß die Checksumme sowohl vor dem Schreiben als auch bei jedem Lesen berechnet werden.

Eine einfachere, für den vorliegenden Anwendungsfall aber genauso wirkungsvolle Methode ist die Verwendung von *Write Check Patterns*. Hier wird am Anfang und am Ende

der Seite ein Byte reserviert. Beim Erzeugen der Seite, also beim erstmaligen Schreiben, werden diese Bytes auf einen definierten Wert gesetzt, z. B. 0x00. Vor jedem Schreibvorgang wird nun der Inhalt bitweise negiert, d. h. aus 0x00 wird 0xFF und umgekehrt. Damit werden die Write Check Patterns der Seite auf der Festplatte jedesmal mit ihrem negierten Wert überschrieben. Stimmen nun beim Lesen einer Seite die Bytes am Anfang und am Ende der Seite nicht überein, so bedeutet dies, daß der Schreibvorgang dieser Seite unvollständig abgebrochen wurde. Die Seite kann somit als ungültig markiert werden.

Careful Writes

Das Datenfile ist so organisiert, daß es nie nötig ist, gültige Daten zu überschreiben. Für das Logfile trifft dies indes nicht zu. Hier ist es nötig, die letzte Seite mit einer neueren Version dieser Seite zu überschreiben, wenn weitere Daten angehängt wurden. Da ein Schreibvorgang auf die Festplatte jedoch nicht als atomarer Vorgang betrachtet werden kann, ist es möglich, daß durch ein einfaches Überschreiben der Seite deren Inhalt zerstört wird. Deshalb ist es nötig, hier spezielle Vorsichtsmaßnahmen zu ergreifen.

Der verwendete Algorithmus läuft so ab: soll die Log-Seite i geschrieben werden, so wird zunächst eine Kopie dieser Seite auf die Seite $i + 1$ geschrieben. Wird synchrones Schreiben verwendet, so ist sichergestellt, daß sich die Seite auf der Festplatte befindet, wenn der entsprechende Systemaufruf zurückkehrt. Danach kann die Seite i überschrieben werden, denn im Fall eines Fehlers kann ja nun jederzeit auf die Kopie zurückgegriffen werden.

Strukturen als Seitenpuffer

Eine Datenseite einer Datei kann vom Betriebssystem nur in Form von binären Daten gelesen werden. Andererseits wird die Anwendung eine Seite im Normalfall als zusammengesetzten Datentyp (Struktur) vorliegen haben. Ein naiver Ansatz zum Schreiben einer Seite wäre also, die Struktur komponentenweise in ein Byte-Array zu kopieren und der Schreibroutine einen Zeiger und die Größe dieses Byte-Arrays zu übergeben. Genauso müßte man beim Lesen die Seite in einen Byte-Puffer holen und von dort die entsprechende Struktur Byte für Byte wieder initialisieren.

Eine weit einfachere Methode läßt sich anwenden, wenn die Struktur die folgenden beiden Voraussetzungen erfüllt: erstens muß der Inhalt der Struktur von fester Länge sein und darf keine Zeiger enthalten. Diese Voraussetzung ist für Strukturen, die eine Datenseite repräsentieren, immer erfüllt, da die Seitenlänge ja fest vorgegeben ist. Zweitens muß dafür gesorgt werden, daß zwischen den einzelnen Komponenten der Struktur keine unbenutzten Bytes übrigbleiben. Zwar machen die Sprachdefinitionen der hier verwendeten Programmiersprachen C und C++ keine Aussagen darüber, wie Strukturen im Speicher angelegt werden. Jedoch kann bei den meisten C- und C++-Implementierungen

davon ausgegangen werden, daß keine “leeren” Bytes zwischen den Komponenten liegen, wenn diese alle ein Vielfaches der Wortlänge der verwendeten Maschine an Speicherplatz belegen. Für den vorliegenden Fall, in dem auf 32bit-Rechnern entwickelt wurde, würde dies bedeuten, daß alle Komponenten der Struktur ein Vielfaches von vier Bytes groß sein müssen. Diese Einschränkung bedeutet auch, daß für die oben beschriebenen Write Check Patterns vier Bytes verwendet werden müssen, obwohl eigentlich auch ein einziges Byte ausreichen würde.

Sind diese Voraussetzungen erfüllt, so kann einfach ein Zeiger auf den Speicherbereich, in dem die Struktur liegt, an die Schreib- bzw. Leseroutine übergeben werden. Damit wird die Struktur direkt geschrieben bzw. gelesen, ohne daß die lästige Konvertierung zu oder von einem Byte-Array explizit erfolgen müßte.

Natürlich ist diese Methode nicht geeignet, um portable Dateien zu erzeugen, da alle Werte in der Byte-Ordnung der verwendeten Maschine auf den Datenträger geschrieben werden. Aber für Dateien, die nur auf einem Rechner bzw. auf einer Architektur verwendet werden, bietet sie eine elegante Möglichkeit, Strukturen permanent zu speichern.

5.2.2 OTS und transaktionale Ressourcen

Verwaltet ein transaktionaler Server seine Daten selbst, ohne sie mit Hilfe eines externen, XA-fähigen Resource Managers zu speichern, so muß er ein Resource-Objekt erzeugen, das am 2PC-Protokoll teilnimmt. Die Schnittstelle für solche Resource-Objekte ist durch die OTS-Spezifikation definiert, sie beinhaltet als wichtigste Methoden die Aufrufe `prepare()`, `commit()` und `rollback()`, die für die Abwicklung des 2PC-Protokolls benötigt werden. Abbildung 5.5 gibt eine Anschauung von der Struktur eines solchen Servers am Beispiel der MQ.

Um nach einem Absturz des Systems einen fehlerfreien Wiederanlauf zu ermöglichen, muß ein transaktionaler Server zusätzlich ein *Restart*-Objekt erzeugen, das ebenfalls beim OTS registriert werden muß. Beim Start des Systems wird dann über dieses Objekt die Beendigung von unvollständigen Transaktionen angestoßen.

Der Restart-Vorgang

Die Schnittstellendefinition des Restart-Objekts enthält nur eine Methode, nämlich `recovery()`. Diese Methode wird vom OTS beim Start des Systems aufgerufen. Kehrt der Aufruf zurück, so erwartet das OTS, daß für alle unvollständigen Transaktionen die Resource-Objekte wieder erzeugt wurden und bereit sind, `commit()`- oder `rollback()`-Aufrufe entgegenzunehmen. In diesem Szenario ist es möglich, daß eine unvollständige Transaktion beim Wiederanlauf trotzdem nicht terminiert wird: wenn nämlich der Zustand der Transaktion im OTS-Logfile als “beendet” aufgezeichnet wurde, hingegen im Logfile des transaktionalen Servers als “unvollendet”. Eine solche Situation kann auftreten, wenn eine *COMMIT*- oder *ROLLBACK*-Nachricht im Netzwerk verloren geht,

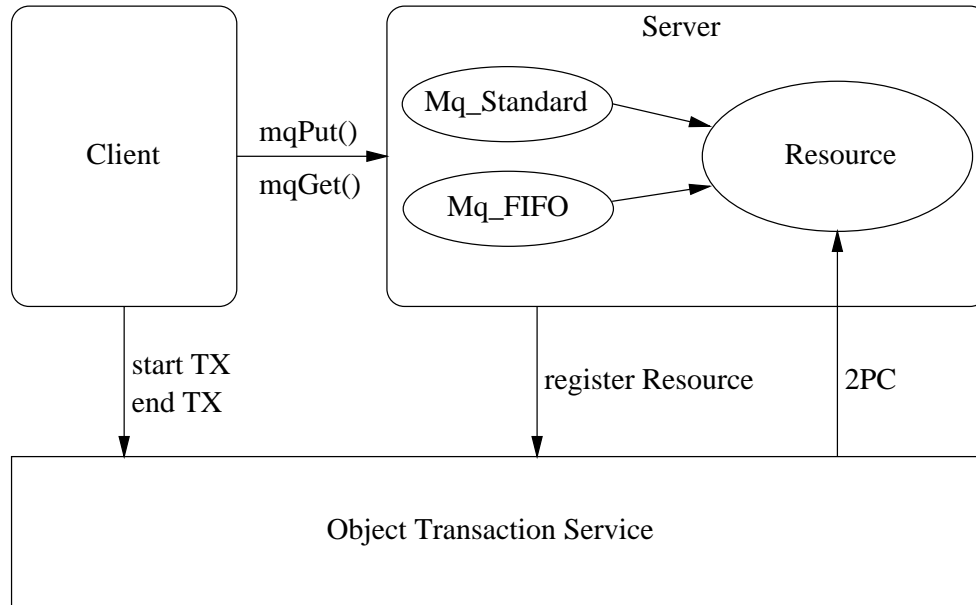


Abbildung 5.5: OTS-Anwendung mit Resource-Objekt

oder wenn der *COMMIT*-Logsatz des transaktionalen Servers nicht mehr ins Logfile geschrieben werden kann.

Um hier trotzdem einen Abschluß der Transaktion zu erwirken, kann ein transaktionaler Server dem OTS durch Aufruf der Methode `replay_completion()` mitteilen, daß für die genannte Transaktion das 2PC-Protokoll nicht vollständig abgewickelt wurde. Der OTS ruft dann nochmals die `commit()`- bzw. die `rollback()`-Methode auf.

5.2.3 Überblick über die Module der MQ

Wie in Abbildung 5.6 dargestellt, besteht die Message Queue aus einer Anzahl verschiedener Module. Drei Schichten lassen sich erkennen: auf unterster Ebene befinden sich der File Manager und der Log Manager. Diese sind für die Speicherung von Nutzlast und Logdaten zuständig. Eine Ebene höher befinden sich die Objekte, die durch Interaktion mit dem OTS die Transaktionseigenschaften und den korrekten Systemstart sicherstellen. Die oberste Schicht schließlich besteht aus den Objekten, die die CORBA-Schnittstellen implementieren.

5.2.4 Logging und Recovery

Die Programmlogik der Objekte in der mittleren Schicht stellt sicher, daß die transaktionalen Eigenschaften des Servers unter allen Umständen gewährleistet bleiben. Dazu

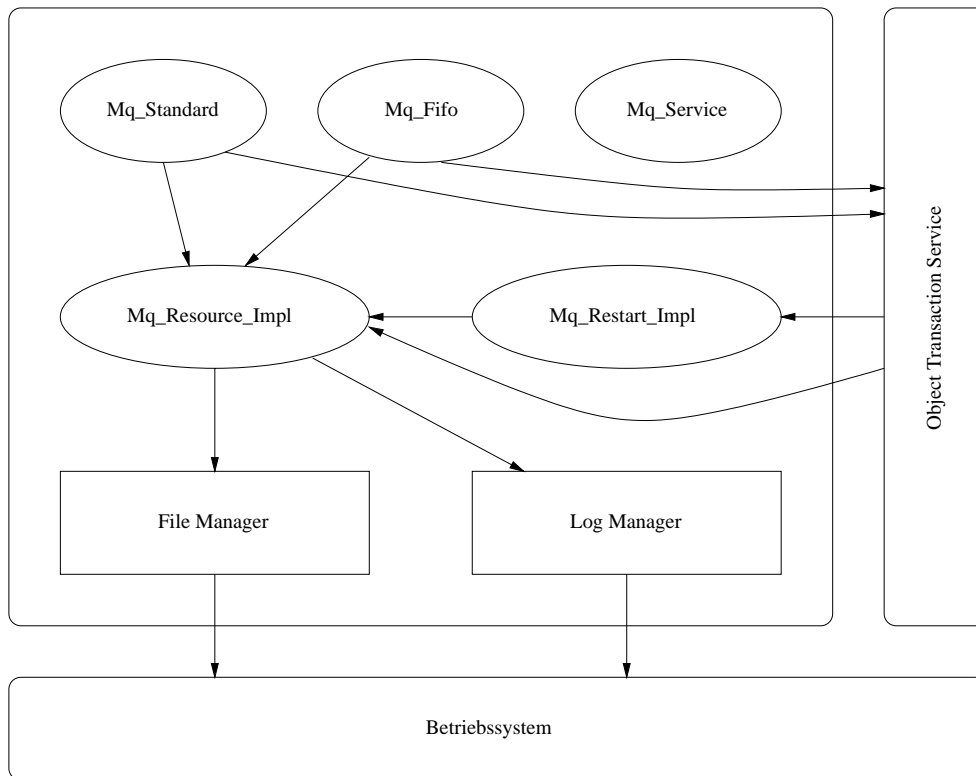


Abbildung 5.6: Module der Message Queue

gehört, daß die Aktionen einer Transaktion rückgängig gemacht werden können, des weiteren, daß beim Systemstart unvollständige Transaktionen korrekt abgeschlossen werden.

Um diese Eigenschaften zu garantieren, müssen der Zustand einer Transaktion sowie die durchgeführten Aktionen protokolliert werden. Dazu dient der Logmanager.

Die Kommunikationsprimitive des Message Queuing lassen sich auf drei Grundoperationen zurückführen: *read()*, *write()* und *delete()*. Im allgemeinen Fall ist die *delete()*-Operation lediglich eine Sonderform einer *write()*-Operation, im vorliegenden Fall jedoch verdient diese Sonderform eine separate Betrachtung, da sich damit sowohl das Datenvolumen als auch die Komplexität der Restart-Logik drastisch reduzieren lassen. Für die *read()*-Operation sind zudem weder Logging noch Recovery nötig, was eine zusätzliche Vereinfachung bedeutet.

Für Rollback und Recovery in einer OTS-Anwendung sind folgende Daten für jede Transaktion aufzuzeichnen:

- der Name der Transaktion
- der Marker des mit der Transaktion assoziierten Resource-Objekts. Marker werden in CORBA verwendet, um Instanzen von Objekten zu adressieren.
- die Objektreferenz des Recovery Coordinators in String-Form. Diese Objektreferenz wird vom OTS zurückgegeben, wenn ein Resource-Objekt registriert wird.
- der Zustand der Transaktion
- Information, um die Änderungen der Transaktion beim Restart erneut nachzuvollziehen bzw. rückgängig zu machen.

Logstrategien

Es gibt zwei Strategien, die den Restart eines transaktionalen Systems sehr vereinfachen können. Dabei geht es um den Zeitpunkt, zu dem Daten aus dem Hauptspeicher auf stabilen Speicher geschrieben werden. Bei der *force*-Strategie müssen die Daten sich vor dem Commit-Zeitpunkt schon auf stabilem Speicher befinden. In diesem Fall ist es beim Systemrestart nicht nötig, irgendwelche Aktionen der Transaktion nochmals auszuführen. Es ist also kein *REDO* notwendig. Wenn die Transaktion allerdings zurückgenommen wird, so müssen alle Änderungen rückgängig gemacht werden.

Im Gegensatz dazu steht die *no-steal*-Strategie. Hier dürfen die Daten erst nach dem Commit auf stabilen Speicher geschrieben werden. Dies bedeutet, daß für Transaktionen, die zurückgenommen werden, keinerlei Änderungen rückgängig gemacht werden müssen (kein *UNDO* nötig). Bricht allerdings eine Transaktion im Zustand "prepared" ab, so müssen die Aktionen der Transaktion beim Restart erneut nachvollzogen werden.

Um das Logging so einfach wie möglich zu halten, wurde bei der Implementierung der MQ eine Mischung aus beiden Strategien verwendet.

Logging für *write()*-Operationen

Um eine *write()*-Operation rückgängig machen zu können, müssen nur die Seitennummern der verwendeten Seiten ins Logfile geschrieben werden. Da das Datenfile so organisiert ist, daß niemals gültige Daten überschrieben werden, genügt es beim Undo, die Seiten wieder als frei zu markieren. Um jedoch die Änderungen erneut nachvollziehen zu können (Redo), müßte der Inhalt der Nachricht geloggt werden. Deshalb bietet es sich für *write()*-Operationen an, die *force*-Strategie anzuwenden.

Logging für *delete()*-Operationen

Bei *delete()*-Operationen ist die Situation genau umgekehrt. Werden die Seiten der Nachricht schon vor dem Commit-Zeitpunkt freigegeben, so kann es passieren, daß sie von konkurrierenden Transaktionen überschrieben werden. Sollte dann die *delete()*-Operation zurückgenommen werden, so müßte sich der Seiteninhalt aller freigegebenen Seiten im Logfile befinden. Werden hingegen die Seiten erst beim Commit freigegeben, so ist es nie nötig, die Operation zurückzunehmen. Allerdings kann es dann passieren, daß die Operation beim Systemstart erneut durchzuführen ist. Dafür ist es jedoch ausreichend, wenn wiederum die Seitennummern der betroffenen Seiten ins Logfile geschrieben werden. Deshalb wird für *delete()*-Operationen die *no-steal*-Strategie angewandt.

Idempotenz von *UNDO*- und *REDO*-Operationen

Im ungünstigsten Fall kann es passieren, daß ein System während der Restart-Phase abstürzt. Um auch diesen Fall abzudecken, ist es a priori notwendig, auch *UNDO*- und *REDO*-Operationen zu loggen, damit bei einem erneuten Systemstart der abgebrochene Recovery-Prozess vervollständigt bzw. rückgängig gemacht werden kann. Diese Notwendigkeit kann vermieden werden, wenn die beim Recovery verwendeten Operationen *idempotent* sind, d. h. wenn das Endergebnis bei mehrfacher Ausführung der Operation dasselbe ist wie bei einfacher Ausführung.

Dies trifft für die oben beschriebenen Operationen zu: sowohl das Undo der *write()*-Operation als auch das Redo der *delete()*-Operation besteht darin, eine Seite des Datenfiles als frei zu markieren. Ob diese Seite einmal oder hundertmal freigegeben wird, spielt keine Rolle.

5.2.5 Filemanager

Die Speicherung von Daten erfolgt in einer einzigen Datei, die seitenweise organisiert ist. Dabei besteht eine Seite aus der in Listing 5.7 abgedruckten Datenstruktur. Die

Größe der `data`-Komponente berechnet sich aus der Seitengröße abzüglich der anderen Komponenten. `flip` und `flop` sind für die Write Check Patterns reserviert. `state` markiert die Seite als gültig oder frei, zu diesem Zweck wurden die beiden Konstanten `ST_VALID` und `ST_FREE` definiert. `pageno` gibt die Seitennummer an, und `lsn` die Log-Sequenz-Nummer des Logsatzes, in dem das Schreiben dieser Seite protokolliert wurde. Dies ist notwendig, da die Logsätze erst nach dem Schreiben der Seiten ins Logfile gespeichert werden. Angenommen, eine Datenseite wäre erfolgreich auf die Platte geschrieben worden, das System jedoch abgestürzt, bevor der dazugehörige Logsatz ins Logfile geschrieben wurde. Dann ist die Seite belegt, und es existiert kein Logsatz, mit dessen Hilfe sie wieder freigegeben werden kann. Damit wäre die Seite für das System verloren. In diesem Fall muß jedoch die LSN der Seite größer sein als die größte LSN, die im Logfile auftritt, da Log-Sequenz-Nummern streng monoton steigen. (Der Überlauf der LSN ist in dieser Implementierung nicht berücksichtigt). Damit steht fest, daß die Seite von einer abgebrochenen Transaktion geschrieben wurde und wieder freigegeben werden kann.

Nachrichten werden nicht fragmentiert, d. h. eine mehrere Seiten lange Nachricht wird immer in adjazenten Seiten gespeichert. Wird im Datenfile kein Block mit genügend freien Seiten gefunden, so wird das File erweitert. Dadurch kann mit der Zeit eine gewisse Fragmentierung auftreten, so daß von Zeit zu Zeit eine Neuorganisation nötig wäre. Diese ist jedoch in der jetzigen Version des Filemanagers nicht implementiert.

Dadurch, daß niemals Teile verschiedener Nachrichten auf einer Datenseite abgelegt werden, kann es nicht passieren, daß gültige Inhalte einer Seite überschrieben werden. Es werden immer nur freie Seiten beschrieben. Das hat zwar den Vorteil, daß für den Filemanager auf Careful Writes verzichtet werden kann, andererseits muß jedoch davon ausgegangen werden, daß jeweils die letzte Seite einer Nachricht im Mittel nur zu 50% gefüllt ist. Dies bedeutet eine gewisse Platzverschwendung. Allerdings ist eine Message Queue nur ein Zwischenspeicher, in dem im Normalfall die Daten mit einer ähnlichen Geschwindigkeit entnommen werden, wie sie eintreffen. Geht man davon aus, daß sich selten mehr als 500 Nachrichten in der Queue befinden und daß die Seitengröße 8 Kilobytes beträgt, so gehen höchstens $500 \cdot 4\text{kB} = 2\text{MB}$ Speicher verloren. Dies ist bei heute üblichen Größen und Preisen von Festplattenspeicher durchaus tolerierbar.

Zur Speicherung liegt eine Nachricht bereits im internen Datenformat, d. h. als Byte-Array, vor. Der Filemanager stellt ein API zur Verfügung, das es ermöglicht, diese Daten in eine Datei zu speichern und aus dieser Datei wieder zu lesen.

Die Funktionalität ist in drei Sub-Module aufgeteilt: `fm_io` ist für die Dateiein- und Ausgabe zuständig, `fm_buffer` stellt einen Pufferbereich zur Verfügung, und `fm_freespace` verwaltet die Liste der freien Seiten im Datenfile.

Die Beschreibung der Sub-Module erfolgt anhand der darin enthaltenen Funktionen.

```
enum {
    ST_VALID,
    ST_FREE
};

struct data_page_struct {
    unsigned long flip;
    unsigned long pageno;
    LSN page_lsn;
    unsigned long state;
    unsigned char data[PAYLOAD_SIZE];
    unsigned long flop;
};
```

Listing 5.7: Seitenstruktur des Datenfiles

Ein/Ausgabe (*IO*)

Die Größe einer Seite ist momentan noch ein Parameter, der zur Übersetzungszeit festgelegt werden muß. In einer späteren Version ist aber beabsichtigt, auch diesen Parameter per Konfigurationsdatei einstellbar zu machen.

`int fm_io_open(char * file);` öffnet die Datei, deren Name als Argument übergeben wurde. Ein File Descriptor wird zurückgegeben. Den Dateinamen liest die MQ aus einer Konfigurationsdatei. Als Schreibmodus wird synchrones Schreiben verwendet. Dies ist nötig, da bei `write()`-Operationen die oben beschriebene *force*-Strategie verwendet wird. Hier muß garantiert sein, daß die entsprechenden Seiten tatsächlich auf die Festplatte und nicht nur in einen Puffer des Betriebssystems geschrieben wurde, wenn die Schreibroutine zurückkehrt.

`void fm_io_close(int fd);` schließt die durch den File Descriptor `fd` bezeichnete Datei.

`char fm_io_read_page(int fd, int pageno, char * buffer);` liest den Inhalt der Seite `pageno` aus der Datei in den Speicherbereich, auf den `buffer` zeigt. Der Speicherbereich muß natürlich groß genug sein, um eine komplette Seite aufzunehmen.

`char fm_io_write_page(int fd, int pageno, char * buffer);` schreibt den Inhalt des durch `buffer` bezeichneten Speicherbereichs auf die entsprechende Seite des Datenfiles.

Puffer (*Buffer*)

`void fm_buffer_init();` initialisiert das Puffer-Modul. Es werden vorerst acht Pufferseiten erzeugt. Falls Bedarf besteht, so können weitere Seiten erzeugt werden. Da auf den Puffermanager von mehreren Threads aus gleichzeitig zugegriffen werden kann, wird jeder Pufferseite ein Mutex hinzugefügt, mit dem ein Thread die Seite exklusiv beanspruchen kann.

`int fm_buffer_fix_page();` sperrt eine Seite exklusiv für den aufrufenden Thread und gibt die Nummer der Seite zurück. Dabei handelt es sich um eine pufferinterne Seitennummer (die n -te Seite im Puffer und nicht die n -te Seite im Datenfile).

`char * fm_buffer_get_pointer(int pageno);` gibt einen Zeiger auf die n -te Seite im Puffer zurück.

`int fm_buffer_unfix_page(int pageno);` schließlich gibt eine reservierte Seite wieder zurück in den Pool und stellt sie damit anderen Threads zur Verfügung.

Freispeicherliste (*Freespace*)

Die Freispeicherliste verwendet zur Verwaltung der freien Seiten Bitlisten. Dies ist erstens eine sehr platzsparende Art, die benötigte Information im Speicher unterzubringen, zweitens werden quasi automatisch adjazente Blöcke von freien Seiten zu einem einzigen Block verschmolzen.

`void fm_freespace_init(long num_pages);` erzeugt und initialisiert die Bitliste sowie ein Mutex zur Synchronisation des Zugriffs.

`void fm_freespace_lock();` sperrt den Freespace-Manager für den exklusiven Zugriff des aufrufenden Threads.

`void fm_freespace_unlock();` gibt den Zugriff auf den Freespace-Manager wieder frei.

`long fm_freespace_alloc_pages(int number);` reserviert die angegebene Anzahl von Seiten im Datenfile und gibt die Seitennummer der ersten reservierten Seite zurück. Vor Verwendung dieser Funktion muß der Freespace-Manager durch Aufruf von `fm_freespace_lock()` gesperrt und danach mit `fm_freespace_unlock()` wieder freigegeben werden.

`long fm_freespace_free_pages(int startpage, int number);` gibt `number` Seiten frei, beginnend bei Seite `startpage`. Auch dieser Aufruf muß durch die beschriebenen Sperrmechanismen gesichert werden.

Zusammenspiel der Submodule

Um Daten mit Hilfe des Filemanagers auf die Festplatte zu schreiben, wird wie folgt vorgegangen:

- Die Anzahl der benötigten Seiten wird aus der Länge der Daten und der Seitengröße berechnet. Dann wird bei der Freispeicherliste ein Block entsprechender Größe angefordert.
- Als nächstes wird eine Pufferseite reserviert. Ist keine Pufferseite mehr frei, so wird vom Puffermanager eine erzeugt.
- Nun werden die Daten in die Pufferseite kopiert und dann mit Hilfe des Ein-/Ausgabe-Moduls an die entsprechenden Stellen im Datenfile geschrieben. Dieser Vorgang wird so lange wiederholt, bis sich alle Daten in der Datei befinden.
- Am Ende muß die Pufferseite wieder an den Puffermanager zurückgegeben werden.

Das Lesen und nachfolgende Löschen eines Bereiches funktioniert genau umgekehrt:

- Es wird eine Pufferseite reserviert
- Die Daten werden in diese Seite gelesen und von dort aus weiterverarbeitet, d. h. in die externe Repräsentation der Nachricht umgewandelt.
- Zum Löschen von Seiten wird der Freispeicherverwaltung mitgeteilt, welche Seiten freizugeben sind.
- Am Ende der Operation wird die Pufferseite wieder freigegeben.

Das Filemanager-Modul erzeugt und schreibt keine Logsätze, dies liegt in der Verantwortung der darüberliegenden Schicht.

Da es sich bei dem Filemanager um ein mehr oder weniger generisches Modul handelt, das unter Umständen auch in anderen Anwendungen zum Einsatz kommen könnte, wurde als Entwicklungssprache C gewählt. Das liegt einfach daran, daß sich C im Unix-Bereich nach wie vor größter Beliebtheit erfreut, und daß sehr viel Software, insbesondere systemnahe Software, weiterhin in C entwickelt wird. Um die Verwendung in einer C++-Umgebung zu vereinfachen, wurde eine C++-Wrapper-Klasse geschrieben, die das prozedurale Interface des Filemanagers in einem Objekt kapselt und außerdem Informationen über das Datenfile, insbesondere den File-Deskriptor, beinhaltet. Auf diese Art ist es auch möglich, mehrere Instanzen des File-Managers zu verwenden.

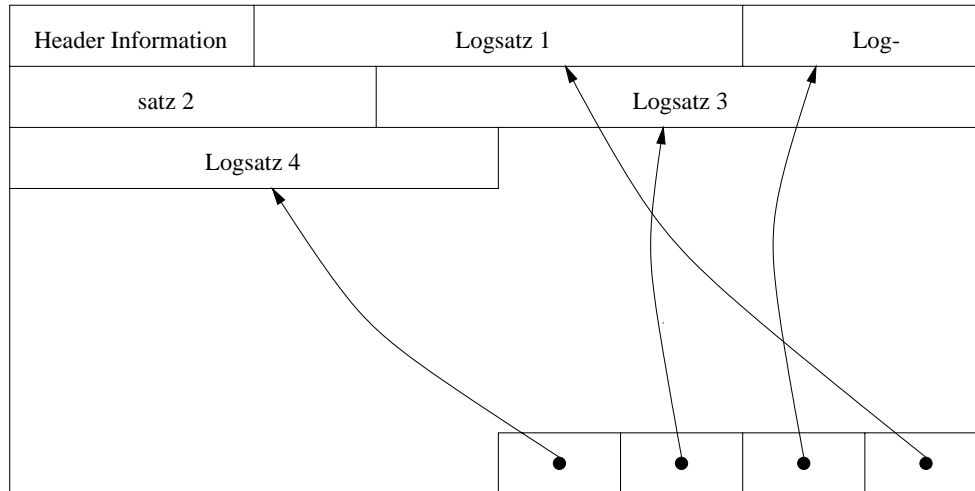


Abbildung 5.8: Aufbau einer Logseite

5.2.6 Logmanager

Der Logmanager ist in der Lage, Logsätze zum Logfile hinzuzufügen und aus dem Logfile zu lesen. Da niemals Änderungen an bestehenden Logsätzen vorgenommen werden, wird das Logfile immer nur sequentiell geschrieben, d. h. Daten werden immer hinten angehängt. Zu diesem Zweck hält der Logmanager einen Seitenpuffer bereit, in dem sich die letzte Seite des Logfiles befindet. Hier werden Logsätze eingefügt. Wenn es nötig ist, die Seite auf stabilen Speicher zu schreiben, so wird die Careful Write-Technik angewandt, da die bereits auf der Platte bestehende Version gültige Daten enthält.

Aufbau von Logseiten

Eine Logseite enthält eine beliebige Anzahl von Logsätzen. Abbildung 5.8 gibt eine schematische Darstellung vom Aufbau der Seite. Logsätze werden hinter der Header-Information eingefügt, vom Ende der Seite wächst das Seitenverzeichnis. Im Seitenverzeichnis steht der Offset der Einträge relativ zum Seitenbeginn.

Listing 5.9 zeigt die Struktur für eine Logseite an. Auch hier werden Write Check Patterns verwendet. `page_no` gibt die Seitennummer an, `page_version` ist ein Versionszähler, der jedesmal beim Schreiben der Seite erhöht wird. Damit lässt sich feststellen, ob die Backup-Version der Seite, die durch die Verwendung von Careful Writes entsteht, neuer ist als die Originalseite. `freespace` gibt an, wieviel freier Platz noch auf der Seite existiert, und `num_entries` schließlich, wieviele Einträge die Seite schon enthält.

Schließlich existiert noch eine Struktur für die Logsätze selbst. Diese ist in Listing 5.10 abgedruckt. `lsn` ist die Log-Sequenz-Nummer des Logsatzes. `tran_prev_lsn` gibt die LSN des letzten Satzes an, der im Rahmen dieser Transaktion geschrieben wurde. Mit

```
struct log_page_struct {
    unsigned long flip;
    unsigned long page_no;
    unsigned long page_version;
    unsigned long freespace;
    unsigned long num_entries;
    unsigned char data[LOG_DATA_SIZE];
    unsigned long flop;
};
```

Listing 5.9: Seitenstruktur des Logfiles

```
struct log_entry_struct {
    LSN lsn;
    LSN tran_prev_lsn;
    TIMESTAMP createtime;
    RMID rmid;
    long tran_name_length;
    char * tran_name;
    long data_length;
    char * data;
};
```

Listing 5.10: Struktur eines Logsatzes

Hilfe dieser Information können ausgehend vom letzten Satz alle Logsätze einer Transaktion einfach gefunden werden. Der erste Logsatz einer Transaktion enthält hier den speziellen Wert *NullLSN*, damit das Ende der Kette erkannt werden kann. *createtime* ist der Zeitstempel des Logsatzes, dieser wird vom Logmanager selbst ausgefüllt. Für die vorliegende Anwendung ist der Zeitstempel eigentlich nicht von Bedeutung, da erstens die Wiederherstellung des Zustandes zu einem bestimmten Zeitpunkt nicht vorgesehen ist, und außerdem auch garnicht möglich wäre, da für diesen Zweck nicht ausreichend viele Daten geloggt werden. Aber wie schon im Fall des Filemanagers gilt auch hier: der Logmanager ist eine generische Komponente, die in anderen Anwendungen wiederverwendbar sein soll. Deshalb wurden hier einige zusätzliche Möglichkeiten eingebaut, um den Logmanager universeller zu gestalten. Ebenfalls für die vorliegende Anwendung nicht nötig ist die nächste Komponente der Logsatz-Struktur, *rmid*. Hier kann eine Identifikationsnummer des Resource Managers eingetragen werden, der den Logsatz schreibt. Dies ist für komplexere Anwendungen, die mehrere Resource Manager enthalten, unabdingbar. *tran_name* gibt den Namen der Transaktion an, zu der dieser Logsatz gehört. Im Fall der MQ wird hier der vom OTS erzeugte Transaktionsname verwendet. Da nicht davon ausgegangen werden kann, daß es sich hier in jedem Fall um einen *NULL*-terminierten String handelt, ist auch ein Feld vorhanden, in dem die Länge des Transaktionsnamens eingetragen werden kann. *data* schließlich enthält die eigentlichen Logdaten, diese können vom jeweiligen Resource Manager beliebig gewählt werden und werden vom Logmanager nicht interpretiert. Auch hier ist ein zusätzliches Feld vorhanden, das die Länge der Logdaten angibt.

Die Länge von Datensätzen ist auf maximal eine Logseite begrenzt. Für die MQ bedeutet dies keine Einschränkung, da aufgrund der beschriebenen Optimierung nur sehr kurze Logsätze geschrieben werden müssen.

Auch der Log-Manager enthält mehrere Sub-Module, über die hier ein Überblick gegeben werden soll. Dazu werden jeweils die wichtigsten Funktionen der Module kurz beschrieben.

Datei-Modul (*log_file*)

`int log_file_open(logfile_info *log_info);` öffnet die Logdatei. Das übergebene Argument ist eine Struktur, die alle nötigen Informationen über das Logfile enthält, so zum Beispiel den Filedeskriptor, die höchste enthaltene Logsequenznummer und die Nummer der letzten Logseite, d. h. der Seite, die gerade beschrieben wird. Auch hier ist es notwendig, die Datei im synchronen Modus zu öffnen, damit sichergestellt ist, daß sich die Seite auf der Festplatte befindet, wenn der *write()*-Systemaufruf zurückkehrt. An dieser Stelle noch eine weitere Bemerkung, die natürlich auch für das Datenfile gilt: der synchrone Modus bringt nichts, wenn sich die Datei in einem verteilten Dateisystem (z. B. NFS) befindet. Kehrt die Schreibroutine zurück, so ist lediglich sichergestellt, daß die Seite zu dem entfern-

ten Knoten übertragen wurde, auf dem die Datei physikalisch liegt. Die Seite kann sich jedoch immer noch im Dateisystem-Cache des Knotens befinden und durch einen Absturz verlorengehen. Aus diesem Grund ist es nicht sinnvoll, das Daten- oder Logfile auf einem verteilten Dateisystem anzulegen, es sollte sich immer auf der lokalen Festplatte befinden.

`int log_file_close(logfile_info *log_info);` schließt die Logdatei wieder. Noch im Puffer befindliche Seiten werden nicht explizit auf die Festplatte geschrieben.

`int log_file_read_page(logfile_info *log_info, int pageno, char *buffer);` liest den Inhalt der durch `pageno` bezeichneten Logseite in den Puffer, auf den `buffer` zeigt.

`int log_file_write_page(logfile_info *log_info, int pageno, char *buffer);` schreibt eine Seite aus dem Puffer in die Logdatei. Diese Funktion implementiert Careful Writes.

Schreiben von Logsätzen (*log_write*)

`LSN log_write_insert(logfile_info *log_info, log_entry entry);` fügt den Log-satz `entry` in die letzte Seite des Logfiles ein. Dazu wird die Gesamtlänge des Satzes berechnet. Falls in der aktuellen Seite noch genügend freier Speicher vorhanden ist, so wird der Satz direkt in die im Logpuffer befindliche Seite eingefügt. Reicht der Platz nicht mehr aus, so wird die Seite auf die Festplatte geschrieben, und es wird eine neue Logseite im Puffer angelegt.

`int log_write_flush(logfile_info *log_info);` erzwingt einen Schreibvorgang, in dem die Seite auf stabilen Speicher übertragen wird.

Lesen von Logsätzen (*log_read*)

`int log_read_get_entry(logfile_info *log_info, LSN lsn, log_entry *entry);` liest den Log-satz `lsn` und füllt damit die Struktur aus, auf die `entry` zeigt. Um Log-sätze aufzufinden, dient das Modul *log_dir*, das die Zugriffspfade für das Logfile zur Verfügung stellt.

`LSN log_read_page_max_lsn(logfile_info *log_info, unsigned long pageno);` liest die maximale LSN einer Seite. Diese Funktion wird benötigt, um einerseits beim Systemstart herauszufinden, welches die nächste zu vergebende LSN ist (in-dem die maximale LSN der letzten Logseite gelesen wird), andererseits, um das Verzeichnis der Log-sätze zu restaurieren, falls es durch einen Fehler zerstört oder korrumpiert werden sollte.

Zugriffspfad für Logsätze (*log_dir*)

`unsigned long log_dir_get_page(LSN lsn);` gibt die Nummer derjenigen Seite in der Logdatei zurück, in der sich der Logsatz `lsn` befindet. Ein besseres Directory würde hier auch die Nummer des Logsatzes innerhalb der Seite mit angeben. Dies ist jedoch für den vorliegenden Anwendungsfall nicht zwingend nötig, da der Log ausschließlich beim Systemstart gelesen werden muß. Für das Zurücknehmen von Transaktionen im laufenden Betrieb werden alle benötigten Informationen im Resource Manager vorgehalten, so daß die Zugriffs- und Lesefunktionen für das Log nicht so hochoptimiert sein müssen.

`void log_dir_set_page_max_lsn(unsigned long pageno, LSN max_lsn);` setzt bzw. ändert die maximale LSN einer Seite. Diese Information genügt dem Verzeichnis, um Logsätze aufzufinden. Wird ein Satz gesucht, so wird die erste Seite ermittelt, deren maximale LSN größer ist als die LSN des gesuchten Satzes. Auf dieser Seite muß sich der Logsatz dann befinden.

`void log_dir_flush();` schreibt das Logverzeichnis auf stabilen Speicher. Die Funktion wird in regelmäßigen Abständen ausgeführt und verwendet Careful Writes.

5.2.7 Resource-Objekt

Das Resource-Objekt nimmt, wie in Abschnitt 5.2.2 beschrieben, am Abschluß der Transaktion teil. Dabei werden vom OTS die Methoden *prepare()* und *commit()* aufgerufen. Das Resource-Objekt muß während der *prepare*-Phase dafür sorgen, daß alle beim Neustart des Systems benötigten Daten auf stabilen Speicher geschrieben werden. Pro Transaktion können durchaus mehrere Resource-Objekte involviert sein. Dabei ist lediglich darauf zu achten, daß ein und dasselbe Objekt nicht mehrmals beim OTS registriert wird.

Bei der Implementierung der MQ wurde ein anderer Weg gewählt: für jede Transaktion existiert nur ein einziges Resource-Objekt. Dafür wird für jeden Methodenaufruf innerhalb einer Transaktion ein Aktionsobjekt erzeugt. Dieses Aktionsobjekt stellt – genau gleich wie ein Resource-Objekt – die für das 2PC-Protokoll notwendigen Methoden zur Verfügung. Im Gegensatz zu einer Resource wird eine Aktion jedoch nicht beim OTS, sondern beim Resource-Objekt registriert. Das Resource-Objekt führt dann für jede registrierte Aktion sozusagen ein “lokales 2PC-Protokoll” durch: in der *prepare*-Phase wird für jede Aktion die *prepare()*-Methode aufgerufen, entsprechendes geschieht während der *commit*- und der *rollback*-Phase.

Dieser Ansatz hat mehrere Vorteile: erstens verringert sich der Umfang der Logdaten merklich bei Transaktionen, die mehrere transaktionale Aufrufe umfassen. Würde pro Aufruf ein Resource-Objekt erzeugt, so müßte auch ein Objekt-Marker und eine Referenz

auf den RecoveryCoordinator gespeichert werden. Zweitens läuft durch die Aktionsobjekte das 2PC-Protokoll schneller ab, da in jeder Phase nur ein Aufruf über CORBA erfolgen muß (vom OTS zum Resource-Objekt), alle anderen Methoden werden lokal aufgerufen. Dies fällt sicher ins Gewicht, weil die Kommunikation über CORBA recht langsam und aufwendig ist. Ein weiterer Vorteil ist, daß der Systemstart beschleunigt werden kann, da pro Transaktion lediglich ein Resource-Objekt und zwei Aktionsobjekte zu erzeugen sind.

Die folgende Übersicht beschreibt, was in den einzelnen Methoden es Resource-Objektes zu tun ist.

`prepare()` erzeugt einen Logsatz mit der Information, daß die *prepare*-Phase beginnt.

Dann wird für alle registrierten Aktionsobjekte die `prepare()`-Methode aufgerufen. Waren alle diese Aufrufe erfolgreich, so wird der Zustand der Transaktion auf *prepared* gesetzt und ein entsprechender Logsatz erzeugt. Dann veranlaßt das Resource-Objekt den Logmanager, alle Logdaten auf stabilen Speicher zu schreiben. Danach kann dem OTS mitgeteilt werden, daß die Transaktion jetzt bereit ist, abgeschlossen (*committed*) zu werden. Falls ein `prepare()`-Aufruf bei einem der Aktionsobjekte fehlschlug, so wird der Zustand entsprechend auf *marked rollback* gesetzt und das OTS zur Rücknahme der Transaktion veranlasst.

`commit()` ruft analog die `commit()`-Methode für jedes Aktionsobjekt auf. Dann wird der *Commit*-Logsatz erzeugt und die Logdaten auf die Festplatte geschrieben.

`rollback()` ruft die entsprechende Methode der Aktionsobjekte auf, auch hier wird zum Schluß ein Logsatz erzeugt und auf die Platte geschrieben.

`commit_one_phase()` wird niemals aufgerufen, da die verwendete OTS-Implementierung diese Art der Optimierung noch nicht kennt.

`forget()` schließlich bleibt ebenfalls unbenützt, da im vorliegenden Anwendungsfall niemals eine heuristische Entscheidung über den Abschluß einer Transaktion getroffen wird. Solche Entscheidungen sind nötig, wenn eine Resource den Kontakt mit dem Transaktionsmanager verliert, während noch Transaktionen offen sind. Um dann das System nicht beliebig lange zu blockieren, kann eine Transaktion heuristisch abgeschlossen oder zurückgenommen werden. In diesem Fall muß sich das System aber alle nötigen Daten der Transaktion merken, da zu einem späteren Zeitpunkt der Transaktionsmanager ein gegenteiliges Endergebnis der Transaktion mitteilen kann. Wenn dann schließlich die `forget()`-Methode aufgerufen wird, so weiß die Resource, daß die entsprechende Transaktion jetzt bei allen Teilnehmern zu einem konsistenten Abschluß geführt wurde, und daß alle Informationen über die Transaktion gelöscht werden können.

Aktionsobjekte

In der MQ existieren zwei Aktionsobjekte, nämlich *StoreAction* zum Speichern von Nachrichten und *DeleteAction* zum Löschen. Die Aktionsobjekte sind von der abstrakten Klasse *QAction* abgeleitet, deren Methoden sie überschreiben müssen. Dem Resource-Objekt kann dann einfach ein Objekt vom Typ *QAction* übergeben werden. Was das Objekt dann im einzelnen tut, darum braucht sich die Resource nicht zu kümmern.

Ein *StoreAction*-Objekt bekommt im Normalfall (im Gegensatz zum *Recovery*-Fall) bei der Erzeugung ein Message-Objekt übergeben, das die interne Repräsentation einer Nachricht, d. h. ein Byte-Array, enthält. Das Objekt fordert dann bei der Freispeicherverwaltung eine ausreichende Anzahl von Datenseiten an. Während der *prepare*-Phase wird der Inhalt der Nachricht auf diese Seiten geschrieben. Für jede beschriebene Seite wird ein Logsatz erzeugt, in dem die Seitennummer steht. Damit kann die Aktion wieder rückgängig gemacht werden. Wurden alle Seiten geschrieben, so kehrt der *prepare()*-Aufruf erfolgreich zurück. Während der *commit*-Phase benachrichtigt das *StoreAction*-Objekt alle registrierten Consumer über das Eintreffen einer neuen Nachricht. Im Falle eines *Rollback* werden alle beschriebenen Seiten wieder gelöscht, d. h. als frei markiert und bei der Freispeicherverwaltung freigegeben.

Das *DeleteAction*-Objekt bekommt bei der Erzeugung die Information, auf welchen Seiten im Datenfile sich die zu löschende Nachricht befindet. Hier wird während der *prepare*-Phase nur ein Logsatz geschrieben, der genau diese Information enthält. Während der *commit*-Phase wird dann die tatsächliche Löschung der Seiten vorgenommen. Da bis zum *rollback*-Zeitpunkt noch keine Aktion ausgeführt wurde, muß in diesem Fall auch nichts rückgängig gemacht werden.

5.2.8 Restart-Objekt

Dieses Objekt tut seinen Dienst ganz am Anfang, beim Systemstart. Es liest die Logdatei, um festzustellen, welche Transaktionen beim letzten Systemstop noch unvollständig waren.

Dabei ist es nicht nötig, das gesamte Logfile von Anfang an zu lesen. Vielmehr wird in regelmäßigen Abständen die LSN auf stabilen Speicher geschrieben, bei der der Restart-Prozess beginnt. Diese Restart-LSN wird wie folgt berechnet: Für jede momentan laufende Transaktion wird die LSN des ersten im Rahmen dieser Transaktion geschriebenen Logsatzes ermittelt. Diese LSN ist im Kontrollblock der Transaktion gespeichert. Das Minimum über alle "ersten LSNs" ergibt nun die Restart-LSN, ab der das Logfile beim Wiederanlauf des Systems zu bearbeiten ist.

Wurden beim Beenden der MQ keine laufenden Transaktionen unterbrochen, so ist beim Restart nichts zu tun. Falls die Queue jedoch abgestürzt ist oder heruntergefahren wurde, während noch Transaktionen offen waren, so werden diese unbeendeten Transaktionen beim Lesen der Logdatei gefunden.

Dann läuft der Restart folgendermaßen:

- Für alle unvollständigen Transaktionen wird ein Info-Block erzeugt. Er enthält den Zustand, in dem die Transaktion abgebrochen wurde, den Marker für das Resource-Objekt der Transaktion, eine Objektreferenz für das RecoveryCoordinator-Objekt in Form eines Strings, sowie zwei Aktionsobjekte: eine *StoreAction* und eine *DeleteAction*. Erstere enthält eine Liste aller Datenseiten, die in dieser Transaktion beschrieben wurden, letztere die Seiten, die gelöscht werden sollten bzw. schon gelöscht wurden.
- Aus jedem dieser Info-Blöcken werden nun wieder das Resource-Objekt sowie der Recovery-Coordinator erzeugt. Dann wird beim Recovery-Coordinator die Methode `replay_completion()` aufgerufen.
- Falls die Transaktion schon den Zustand *prepared* erreicht hatte, ruft der Recovery-Coordinator jetzt entweder die `commit()`- oder die `rollback()`-Methode auf.
- Falls die Transaktion noch nicht *prepared* war, so erhebt der Recovery-Coordinator eine *NotPrepared*-Ausnahme. In diesem Fall kann das Restart-Objekt die Rücknahme der Transaktion veranlassen.

5.2.9 Locking

Vorab ist anzumerken, daß eine Transaktion durchaus mehrere Threads umfassen kann. Die hier beschriebenen Sperrverfahren haben also nichts mit denjenigen zu tun, die zur Synchronisation von Threads verwendet werden, wie z. B. Semaphore und Monitore. Natürlich ist der Zugriff auf alle Datenstrukturen, die von Threads gemeinsam genutzt werden, durch solche Methoden geschützt.

Wenn konkurrierende Zugriffe auf die Warteschlange zugelassen werden, so muß natürlich auch verhindert werden, daß sich diese Zugriffe ins Gehege kommen. Im vorliegenden Fall kann ein sehr einfaches Sperrverfahren gewählt werden. Sowohl *put*- als auch *get*-Methoden greifen schreibend auf die MQ zu – beim *get* wird die Nachricht ja gleich gelöscht –, so daß auf die Verwendung von Lesesperren verzichtet werden kann. Nachrichten werden also immer exklusiv von einer Transaktion gesperrt. Diese Sperre erfolgt einfach, indem ein Zeiger auf den Transaktionskontext derjenigen Transaktion, die die Sperre hält, in den Kontrollblock der Nachricht eingehängt wird. Jeder Thread, der auf die Nachricht zugreifen will, kann so leicht überprüfen, ob die Nachricht von der Transaktion, zu der er gehört, gesperrt wird. Ist dies der Fall, so kann er auf die Nachricht zugreifen. Falls eine andere Transaktion die Sperre hält, so hängt sich der Thread in die Warteschlange ein und wartet, bis ihm signalisiert wird, daß die Sperre freigegeben wurde. Die Signalisierung erfolgt über einen Monitor. Zur Vermeidung von Deadlocks wird ein Timeout gesetzt, nach dem der Versuch, die Nachricht zu sperren, aufgegeben wird. Die Transaktion wird dann abgebrochen.

5.2.10 “Gesamtansicht”

Nachdem in den vorhergehenden Abschnitten alle nötigen Bausteine der MQ vorgestellt wurden, soll abschließend nochmal ein Überblick gegeben werden, wie die Komponenten interagieren. Zu diesem Zweck wird der typische Ablauf einer *put*-Operation unter die Lupe genommen.

mqPut()-Methode: Der Client startet eine Transaktion. Dann ruft er die Methode `mqPut()` beim Server-Objekt auf. Es werden die Nachricht und das Control-Objekt der Transaktion übergeben.

Der Server prüft, ob für die Transaktion schon ein Resource-Objekt vorhanden ist. Falls nicht, so wird eines erzeugt und beim OTS registriert. Die entsprechenden Logsätze (Resource Marker, Recovery Coordinator) werden geschrieben.

Der Server erzeugt einen eindeutigen Schlüssel für die Nachricht aus der Systemzeit und einem Zähler.

Der Server erzeugt ein Message-Objekt, das mit den vom Client empfangenen Daten initialisiert wird. Das Message-Objekt wird in die Liste der vorhandenen Nachrichten eingefügt. Im Kontrollblock des Objekts wird ein Zeiger auf das Control-Objekt der Transaktion eingehängt, der als Sperre des Objektes dient.

Nun wird ein *StoreAction*-Objekt erzeugt, das einen Zeiger auf das eben erstellte Message-Objekt erhält. Das Action-Objekt wird beim Resource-Objekt der Transaktion registriert.

Der Server gibt den erzeugten Schlüssel an den Client zurück. Damit ist die Methode `mqPut()` abgearbeitet.

PREPARE-Phase: Der Client ruft `commit()` auf und stößt damit die Ausführung des 2PC-Protokolls an.

Der OTS ruft die *prepare*-Methode des Resource-Objekts auf. Dieses schreibt einen PREPARING-Logsatz in die Logdatei und ruft dann die *prepare*-Methode des Action-Objekts auf.

Die StoreAction fordert beim Freispeichermanager die benötigte Anzahl von Datenseiten an, reserviert eine Pufferseite und speichert die Nachricht in den vom Freispeichermanager zugeteilten Seiten. Die Seitennummern der beschriebenen Seiten werden im Logfile gespeichert.

Ist die *prepare*-Methode des Action-Objektes erfolgreich zurückgekehrt, so schreibt das Resource-Objekt den PREPARED-Logsatz. Falls das Action-Objekt einen Fehler zurückgibt, so wird stattdessen ein PREPARE_FAILED-Logsatz erzeugt. Dann wird der Logmanager veranlaßt, die aktuelle Logseite auf stabilen Speicher zu schreiben.

Nun ist die *prepare*-Methode des Resource-Objekts abgeschlossen. Wenn kein Fehler auftrat, dann wird dem OTS angezeigt, daß die Resource für die Commit-Phase des 2PC-Protokolls bereit ist (*VoteCommit*), anderenfalls wird eine Rücknahme der Transaktion veranlaßt (*VoteRollback*).

COMMIT-Phase: Haben alle an der Transaktion beteiligten Ressourcen die Prepare-Phase erfolgreich abgeschlossen, so ruft der OTS schließlich die *commit*-Methode des Resource-Objekts auf.

Das Resource-Object schreibt den COMMITTING-Logsatz und ruft die *commit*-Methode des StoreAction-Objektes auf.

Das StoreAction-Objekt entfernt aus dem Kontrollblock der Nachricht den Zeiger auf den Transaktionskontext und gibt damit die Sperre auf die Nachricht zurück. Dann sendet es an alle registrierten Consumer eine Benachrichtigung, daß eine neue Message eingetroffen ist.

Damit ist die *commit*-Methode des Resource-Objektes ebenfalls abgeschlossen, es wird noch der COMMITTED-Logsatz geschrieben, dann kehrt die Methode zurück.

ROLLBACK-Phase: Wird von einer beteiligten Resource oder vom Client eine Rücknahme der Transaktion veranlaßt, dann wird vom OTS die *rollback*-Methode des Resource-Objekts aufgerufen.

Das Resource-Objekt schreibt den ROLLING_BACK-Logsatz und ruft seinerseits die *rollback*-Methode des Action-Objektes auf.

Das StoreAction-Objekt gibt alle im Lauf der Prepare-Phase beschriebenen Seiten wieder frei, indem es das *state*-Feld der Seiten (s. Listing 5.7) auf "frei" setzt und dies dem Freispeichermanager mitteilt. Außerdem wird das Message-Objekt gelöscht.

Das Resource-Objekt schreibt dann den ROLLED_BACK-Logsatz. Damit ist der Rollback der Transaktion beendet.

Der Ablauf einer *get*-Operation ist sehr ähnlich. Während der *mqGet()*-Methode wird die Nachricht aus dem Datenfile gelesen und an den Client zurückgegeben. Anstatt einem *StoreAction*-Objekt wird dann eine *DeleteAction* erzeugt. Diese schreibt während der Prepare-Phase die Liste der zu löschenden Seiten ins Logfile und löscht sie dann während der Commit-Phase.

Kapitel 6

Leistungsbewertung

6.1 Messungen mit dem Testclient

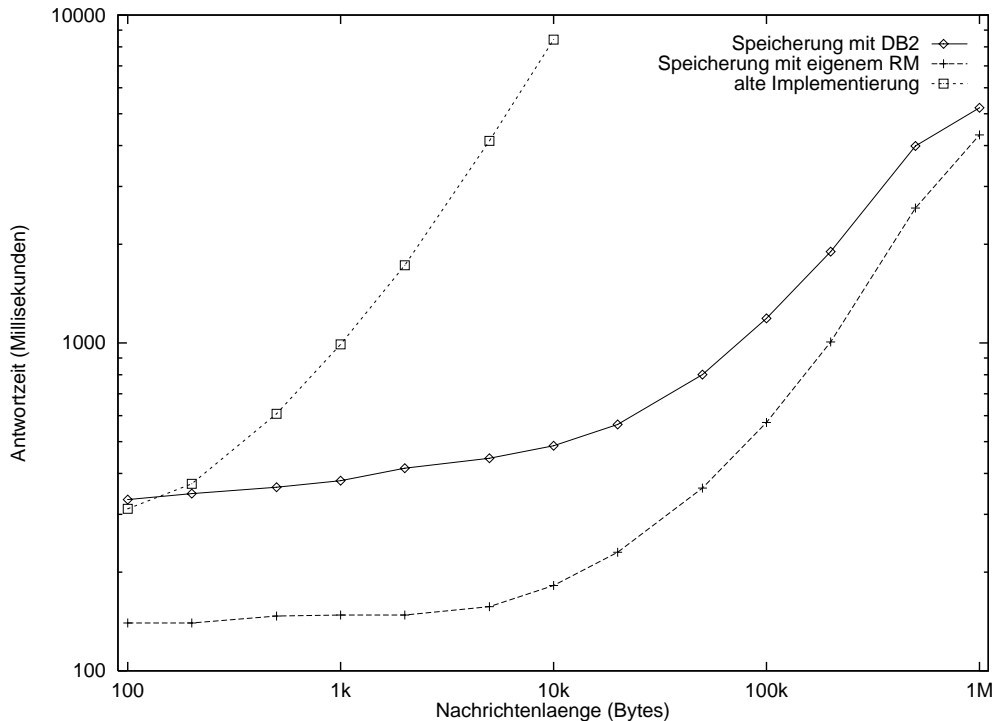
Um die Leistung der verschiedenen Implementierungen zu bewerten, wurde eine Reihe von Messungen vorgenommen. Als Maschine diente eine SUN SPARCstation 10 mit zwei 50MHz-CPU's und 256MByte RAM. Auf dieser Maschine liefen sowohl der Server als auch der Client, so daß die Kommunikation nicht über das Netz erfolgte.

Es wurden immer Serien von 100 Nachrichten zur Warteschlange übertragen, sofern sich das Volumen im Datenfile unterbringen ließ. Leider war in der Partition, die das Datenfile enthielt, nur sehr begrenzter Speicherplatz verfügbar, so daß die Anzahl für Nachrichten, die größer als 100kBytes waren, gesenkt werden mußte (50 Nachrichten bei 200kB und 500kB, 25 Nachrichten bei 1MByte).

Nachdem alle Nachrichten übertragen waren, wurden sie wieder aus der MQ ausgelesen. Die Zeit wurde jeweils von vor dem Beginn der Transaktion bis nach deren Beendigung gestoppt. Die Meßergebnisse aller Messungen sind in Anhang C abgedruckt. Die Abbildungen 6.1 und 6.2 geben eine graphische Aufbereitung der Ergebnisse wieder.

Bei den Messungen mit der neuimplementierten Message Queue fiel auf, daß sich die Streuung der Meßwerte in relativ engen Grenzen bewegte und daß sich die Messwerte ziemlich gut reproduzieren ließen.

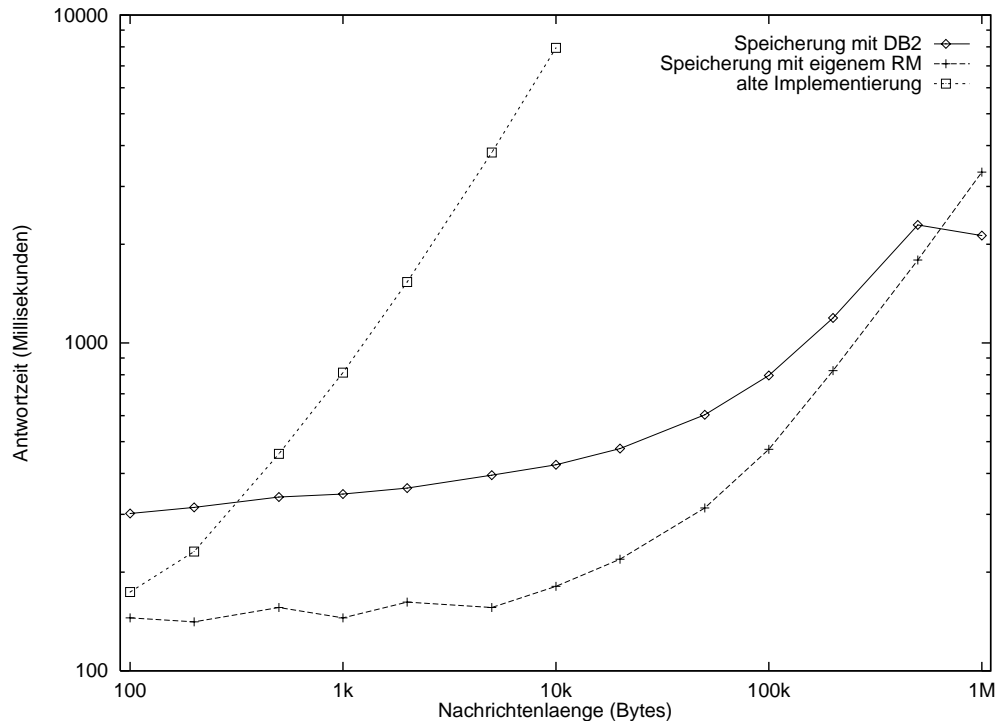
Während sich bei der alten Implementierung der MQ von Anfang an eine ziemlich lineare Abhängigkeit der Antwortzeit von der Länge der Nachricht zeigt, ist dieser Zusammenhang für die neue Warteschlange erst ab einer Nachrichtengröße von etwa 8-10kBytes deutlich sichtbar. Dies folgt aus der seitenweisen Organisation des Datenfiles. Erst wenn die Zahl der Ein-/Ausgabe-Operationen auf der Festplatte ein gewisses Minimum übersteigt kommt dieser Faktor zum tragen.

Abbildung 6.1: Zeitmessung mit *put*-Requests

6.1.1 Ursprüngliche Implementierung

Die Leistung der ursprünglichen Implementierung wurde zum Vergleich ebenfalls gemessen, sie bleibt weit hinter der neuen MQ zurück. Der Grund dafür ist wohl vor allem die ineffiziente Implementierung des Dateimanagers. Hier bricht die Leistung völlig ein, wenn sich schon einige Nachrichten in der Queue befinden. Während die ersten Nachrichten in annähernd der gleichen Zeit gespeichert werden wie in der neuen MQ, dauert der Vorgang für die letzten Nachrichten einer Meßreihe etwa 20mal so lange. Aus diesem Grund wurde die Messreihe bei einer Nachrichtengröße von 10kBytes abgebrochen.

Die alte Implementierung der Queue unterstützt kein transaktionales Lesen und Löschen von Nachrichten. Deshalb wurde die *get*-Operation hier durch einen *read*- und einen *remove*-Aufruf simuliert. Da diese Vorgänge nicht im Rahmen einer Transaktion stattfinden, laufen sie relativ schnell ab – wiederum vorausgesetzt, die Queue ist fast leer, so daß die schlechte Qualität des Dateimanagers nicht so ins Gewicht fällt. Vergleicht man die schnellsten Ergebnisse der alten Queue mit denen der neuen, so zeigt sich, daß ohne Transaktionsumgebung etwa 50-80ms einsparen lassen. Dies entspricht dann etwa dem zusätzlichen Kommunikations- und Verwaltungsaufwand pro Transaktion.

Abbildung 6.2: Zeitmessung mit *get*-Requests

6.1.2 Implementierung mit DB2

Die hier verwendete Datenbankinstallation dient vor allem zu Test- und Übungszwecken, so daß nicht davon auszugehen ist, daß sie in irgendeiner Weise optimiert wurde. Es kann also angenommen werden, daß hier noch einiges an Potential vorhanden ist, das für die vorliegende Meßreihe nicht genutzt wurde. Dennoch sind einige Tendenzen klar zu erkennen. Sogar im günstigsten Fall dauert ein Vorgang bei der DB2-Version ca. 150ms länger als bei der selbstentwickelten Version. Das liegt an mehreren Gründen:

- DB2 ist eine universelle Datenbank und kann deshalb die Daten nicht so optimiert speichern und loggen, wie es mit dem im Rahmen dieser Arbeit entwickelten Resource Manager möglich ist.
- Die Speicherung der Daten erfolgt in einem separaten Prozess, dadurch ist ein zusätzlicher Aufwand für die Kommunikation zwischen den Prozessen nötig. Der Inhalt der Nachricht muß ebenfalls an den DBMS-Prozess weitergereicht werden.

Neben den eingangs genannten Nachteilen des DBMS-Ansatzes erweist sich diese Methode zudem als relativ behäbig. Der große Vorteil ist andererseits, daß die Implementierung der MQ nachgeradezu trivial wird, und daß ein sehr solides Produkt zugrunde liegt.

6.1.3 Implementierung mit eigenem Resource Manager

Die Version der Queue, die ihren eigenen Resource Manager zur Speicherung der Daten mitbrachte, erzielte im Test die besten Ergebnisse. Dies ist einerseits der Tatsache zuzuschreiben, daß das vorhandene Optimierungspotential gut genutzt wurde. Andererseits ist es sicherlich von Vorteil, daß sich alle Komponenten der Queue “unter einem Dach”, d. h. innerhalb desselben Prozesses, befinden. Dadurch wird der Overhead durch Interprozesskommunikation, wie er bei der Verwendung eines externen Resource Managers ergibt, vermieden.

Allerdings steigt auch hier die Meßkurve schon ab einer Nachrichtengröße von 5 bis 10 kBytes relativ steil an. Hier könnte durch eine verbesserte Pufferverwaltung und durch das Schreiben bzw. Lesen von mehreren Seiten auf einmal sicher noch eine Leistungssteigerung herbeigeführt werden.

Allgemein kann jedoch gesagt werden, daß die Implementierung mit eigenem Resource Manager die an sie gestellten Erwartungen voll erfüllt hat.

6.2 Test mit der EOP-Implementierung

Als zweites Testszenario sollte die im Rahmen von [2] entwickelte Implementierung des Exactly-Once-Protokolls dienen. Allerdings wurden die Änderungen, die an dieser Implementierung vorzunehmen waren, stark unterschätzt. Erstens wird bei der neuen MQ ein völlig anderes Nachrichtenformat verwendet, zweitens geschieht die Benachrichtigung der registrierten Consumer auf einem anderen Weg als bisher (CORBA Event Service vs. UDP-Nachricht). Aus zeitlichen Gründen war es nicht mehr möglich, die EOP-Implementierung anzupassen. Es kann jedoch davon ausgegangen werden, daß auch in diesem Szenario die neue Message Queue deutlich schneller ist als die alte Version.

Kapitel 7

Zusammenfassung und Ausblick

7.1 Zusammenfassung

Bei der Entwicklung des vorliegenden Systems hat es sich als hilfreich erwiesen, die Grundlagenliteratur über Transaktionssysteme eingehend zu studieren. Allerdings hätte eine Implementierung etwa entlang der in [3] beschriebenen Richtlinien ein viel zu umfangreiches System ergeben. Hier hat es sich bezahlt gemacht, die Anforderungen und damit auch die Optimierungsmöglichkeiten bezüglich der Speicherung sehr genau unter die Lupe zu nehmen. Dadurch konnte das Volumen der pro Transaktion zu schreibenden Daten drastisch reduziert und damit die Geschwindigkeit gesteigert werden.

Die Arbeit mit modernen Technologien wie CORBA und OTS war spannend und lehrreich. Allerdings wäre eine etwas robustere Implementierung hier wünschenswert gewesen. Durch die auftretenden Fehler wurde viel Zeit verbraucht. So wurde etwa drei Tage lang versucht, mit dem *Any*-Datentyp zu arbeiten, bis erwiesen war, daß dieser nur sehr eingeschränkt funktionierte. Einer der harmloseren Fehler war, daß der IDL-Compiler teilweise bei einfachen Syntaxfehlern crashte. Hier wurde dann kurzerhand der IDL-Compiler eines anderen ORB verwendet, um die Fehler in der Eingabedatei zu suchen.

Kurz vor Fertigstellung der Arbeit trat noch ein Fehler auf, der bis jetzt nicht vollständig geklärt ist: sobald die *init*-Routine des OTS aufgerufen wurde, stürzte die Speicherallokationsroutine `malloc()` früher oder später ab. Nach einem Tag Debugging wurde dann eine Lösung gefunden, mit der sich das Problem umgehen ließ: Solaris bringt zwei `malloc()`-Routinen mit. Neben derjenigen, die standardmäßig verwendet wird, gibt es noch eine, die in der Bibliothek *libbsdmalloc* zu finden ist. Wird diese Bibliothek zu dem MQ-Server dazugebunden, so tritt der Fehler nicht mehr auf. Es steht zu vermuten, daß hier entweder OrbixOTS selbst einen Speicherzuordnungsfehler enthält, oder daß in der erstellten Software ein so unglücklicher Zeigerfehler steckt, so daß der eben beschriebene Effekt auftritt.

7.2 Ausblick

Die Möglichkeiten zur Optimierung sind für die MQ noch bei weitem nicht erschöpft. Es würde sich beispielsweise anbieten, für die interne Darstellung der Nachrichten ebenfalls einen Pool von Seitenpuffern zu reservieren, damit der Speicher hier nicht jedesmal frisch allokiert werden muß.

Sodann sollte das Logverzeichnis dahingehend erweitert werden, daß für einen Logsatz nicht nur die Seitennummer, sondern auch die Position innerhalb der Seite leicht ermittelt werden kann. Auf diese Art müßte nicht die ganze Logseite nach dem Eintrag durchsucht werden.

Dann gäbe es sowohl beim Schreiben als auch beim Lesen die Möglichkeit, mehrere Seiten zu einem Block zusammenzufassen und somit größere Blöcke ein- bzw. auszugeben. Das würde die Effizienz des Log- und des Dateimanagers nochmals deutlich erhöhen. Allerdings wäre dafür ein weitaus besserer Datenpuffer erforderlich, als er in der vorliegenden Version der MQ existiert. (Diese Art der Optimierung nimmt normalerweise das Betriebssystem vor, indem es Daten in einem Buffer Cache zwischenspeichert und dann zu optimierten I/O-Operationen zusammenfasst. Im vorliegenden Fall ist das jedoch nicht möglich, da sowohl für die Log- als auch für die Datendatei synchrone Ein-/Ausgabe erforderlich ist.)

Auch beim Logmanager gibt es noch einige verbesserungsfähige Dinge. Zwar sind hier auch komplizierte Optimierungen wie Group Commit denkbar, wo die *COMMIT*-Datensätze mehrerer Transaktionen zusammengefasst werden, um somit größere Logsätze und weniger I/O-Operationen zu erhalten. Diese Optimierung ist jedoch nicht so naheliegend wie einige andere. Zum Beispiel könnte versucht werden, pro Transaktion mehrere Logsätze zu einem einzigen zusammenzufassen, um damit den Overhead, der durch die Log-Header entsteht, zu reduzieren. Außerdem wäre auch hier beim Lesen eine bessere Pufferseitenverwaltung angebracht.

Eine weitere Schwäche der vorliegenden Implementierung ist, daß das Datenfile nicht von Zeit zu Zeit reorganisiert wird, was jedoch sinnvoll wäre. Einerseits tritt die bereits beschriebene Fragmentierung nach einigem Betrieb auf. Ein weiteres Problem, das erst beim Testen der MQ mit großen Nachrichten auffiel, ist, daß zwar im Bedarfsfall die Datendatei dynamisch vergrößert wird, sie wird aber nicht wieder verkleinert. Angenommen, es werden nacheinander 100 Nachrichten von 1MB Länge an die Warteschlange gesandt. Dann wird das Datenfile gute 100 MByte groß sein. Werden die Nachrichten dann alle wieder aus der Warteschlange herausgenommen, so liegt eine 100 MByte große Datei mit lauter leeren Seiten herum. Dies ist eine unnötige Platzverschwendung.

Zusammenfassend kann jedoch gesagt werden, daß mit der vorliegenden Software eine durchaus brauchbare Implementierung einer transaktionalen Warteschlange für Nachrichten entstanden ist. Besonders erfreulich ist, daß das gesetzte Ziel bezüglich der Leistungsfähigkeit der Queue gut erreicht wurde.

Anhang A

Listings

```
struct mqTimestamp_struct
{
    unsigned long mq_sec;
    unsigned long mq_usec;
};

typedef mqTimestamp_struct mqTimestamp;

struct mqHeader_struct
{
    unsigned short priority;
    unsigned short groupID;
    mqTimestamp time;
    string senderID;
};

typedef mqHeader_struct mqHeader;
```

Listing A.1: Header-Datentyp

```
struct mqAttribute_struct
{
    string key;
    string value;
};

typedef mqAttribute_struct mqAttribute;

typedef mqAttribList sequence<mqAttribute>;
```

Listing A.2: Attributliste

```
interface Mq_Fifo
{
    exception QueueDisabled {};

    void mqPut(in CosTransactions::Control ctrl,
              in mqHeader header,
              in mqAttribList attrib,
              in mqPayload data);

    void mqGetNext(in CosTransactions::Control ctrl,
                  out mqHeader header,
                  out mqAttribList attrib,
                  out mqPayload data);

    void mqStartNotify(in string channelName,
                      in unsigned short groupID);

    void mqSetPriorization(in boolean b);

    boolean mqGetPriorization();
};
```

Listing A.3: FIFO-Interface

```
interface Mq_Standard {  
  
    exception QueueDisabled {};  
  
    void mqPut(in CosTransactions::Control ctrl,  
              in mqHeader header,  
              in mqAttribList attrib,  
              in mqPayload data,  
              out string<15> key);  
  
    void mqGet(in CosTransactions::Control ctrl,  
              in string<15> key,  
              out mqHeader header,  
              out mqAttribList attrib,  
              out mqPayload data);  
  
    boolean mqRead(in string<15> key,  
                  out mqHeader header,  
                  out mqAttribList attrib,  
                  out mqPayload data);  
  
    boolean mqRemove(in string<15> key);  
  
    void mqGetAttribList(in string<15> key,  
                        out mqAttribList attrib);  
  
    void mqStartNotify(in string channelName,  
                      in unsigned short groupID);  
};
```

Listing A.4: Standard-Interface

```
interface Mq_Service {
    void mqShutdown();

    void setEnabled(in boolean b);

    boolean isEnabled();

    string getStatus();
};
```

Listing A.5: Service-Interface

Anhang B

Verwendungshinweise für die Message Queue

Hier noch ein paar Tips für Administratoren und Programmierer, die die Message Queue verwenden wollen.

Da nicht zu erwarten ist, daß die Queue eine allzu große Verbreitung findet – immerhin ist zum Betrieb kommerzielle Software im Wert von etlichen tausend Mark nötig –, wurde auf die Entwicklung eines Konfigurationsskriptes verzichtet, die Datei `Makefile` muß also von Hand angepasst werden. Insbesondere die Variablen im ersten Teil der Datei müssen überprüft werden. Das Verzeichnis `LOGDIR` darf nicht im NFS liegen, da dort die synchronen Schreibzugriffe nicht funktionieren.

Ist das `Makefile` angepasst, so sollte ein Aufruf von `make` den Queue Server übersetzen und binden.

Es stehen drei kleine Testprogramme zur Verfügung: der `TestClient` (wird erzeugt mit `make client`), der `Consumer`, der die `EventService-Notification` nutzt (`make consumer`), sowie ein kleines Administrationsprogramm (`make admin`), mit dessen Hilfe die Queue ein- und ausgeschaltet und heruntergefahren werden kann.

Ist der Übersetzungsvorgang problemlos vonstatten gegangen, so kann mit `make run` die Message Queue gestartet werden.

Damit die `Notification` beim Eintreffen neuer Nachrichten funktioniert, muß der `Orbix-EventService` bereits laufen, vor die MQ gestartet wird (s. [7]).

Bei der Verwendung der MQ wird ein Blick in die Quellen der Testprogramme sehr hilfreich sein, hier nur noch einige Hinweise:

- Es wird explizite Transaktionspropagation verwendet. Deshalb muß allen transaktionalen Methoden ein `Control-Objekt` übergeben werden, das den Transaktionskontext enthält. Wird das Pseudo-Object `Current` verwendet, so kann eine Referenz auf das `Control-Objekt` mit `Current.get_control()` erhalten werden.

-
- Consumer, die Notification per EventService verwenden, müssen den zu verwendenden Kanal nicht explizit erzeugen, solange die OrbixEvents-Implementierung zum Einsatz kommt. In diesem Fall sucht der Queue-Server den angegebenen Kanal und erstellt ihn, falls er noch nicht existiert. Für die Verwendung einer anderen EventService-Implementierung müßte die MQ angepasst werden.
 - Falls ein Consumer sich neu bei der MQ registriert und dabei einen Event-Kanal angibt, der schon von anderen Consumern verwendet wird, so erhalten diese anderen Consumer ebenfalls die Liste aller in der Queue enthaltenen Nachrichten. Um diesen Effekt zu vermeiden, müssen entweder alle Consumer bei der Queue registriert sein, bevor die erste Nachricht eingeht, oder sie müssen mehrfach erhaltene Notifications eliminieren.

Anhang C

Meßergebnisse

Größe (Bytes)	Zeit (ms)		
	agv	min	max
100	312	180	473
200	372	179	839
500	608	193	1148
1000	990	218	1903
2000	1726	261	3371
5000	4133	384	8602
10000	8419	619	18072

Tabelle C.1: Alte Implementierung: Put-Requests

Größe (Bytes)	Zeit (ms)		
	agv	min	max
100	174	54	346
200	231	51	444
500	459	54	920
1000	812	56	1746
2000	1534	59	3124
5000	3809	72	8272
10000	7941	85	17517

Tabelle C.2: Alte Implementierung: Get-Requests

Größe (Bytes)	Zeit (ms)		
	avg	min	max
100	333	311	554
200	347	321	613
500	363	333	439
1000	380	355	560
2000	415	387	568
5000	445	410	766
10000	486	465	603
20000	564	523	790
50000	801	700	992
100000	1188	1000	1634
200000	1898	1590	2494
500000	3986	3268	4704
1000000	5216	5070	6626

Tabelle C.3: Neue MQ mit DB2: Put-Requests

Größe (Bytes)	Zeit (ms)		
	avg	min	max
100	302	277	389
200	315	298	889
500	339	309	581
1000	346	322	698
2000	361	344	500
5000	395	377	655
10000	425	400	700
20000	476	454	778
50000	604	587	834
100000	796	766	933
200000	1192	1122	1778
500000	2291	2135	4347
1000000	2125	2055	2371

Tabelle C.4: Neue MQ mit DB2: Get-Requests

Größe (Bytes)	Zeit (ms)		
	avg	min	max
100	140	129	336
200	140	126	236
500	147	130	313
1000	148	130	475
2000	148	134	312
5000	157	147	209
10000	182	169	259
20000	230	214	326
50000	361	340	527
100000	572	545	771
200000	1007	966	1410
500000	2579	2209	10115
1000000	4311	4263	4491

Tabelle C.5: Neue MQ mit eigenem RM: Put-Requests

Größe (Bytes)	Zeit (ms)		
	avg	min	max
100	145	134	197
200	141	131	181
500	156	138	314
1000	145	133	345
2000	148	134	312
5000	156	148	224
10000	181	170	318
20000	219	201	405
50000	314	301	470
100000	474	455	661
200000	823	786	1089
500000	1790	1712	2599
1000000	3320	3294	3374

Tabelle C.6: Neue MQ mit eigenem RM: Get-Requests

Literaturverzeichnis

- [1] Elmasri, Ramez; Navathe, Shamkant: *Fundamentals of Database Systems*
The Benjamin/Cummings Publishing Company Inc. 1994
- [2] Friedel, Klaus: *Fehlertolerantes Protokoll zur Exactly-Once-Ausführung von Agenten*
Diplomarbeit Nr. 1651, IPVR Universität Stuttgart, 1998
- [3] Gray, Jim; Reuter, Andreas: *Transaction Processing, Concepts and Techniques*
Morgan Kaufman Publisher Inc. 1994
- [4] IONA Technologies PLC: *Orbix Programmer's Guide*
IONA Technologies PLC, 1997
- [5] IONA Technologies PLC: *Orbix Programmer's Reference*
IONA Technologies PLC, 1997
- [6] IONA Technologies PLC: *OrbixOTS Programmer's and Administrator's Guide*
IONA Technologies PLC, 1998
- [7] IONA Technologies PLC: *OrbixEvents Programmer's Guide*
IONA Technologies PLC, 1997
- [8] Maihöfer, Christian: *Ein Protokoll zur Wahrung der Exactly-Once-Eigenschaft Mobiler Agenten*
Diplomarbeit Nr. 1565, IPVR Universität Stuttgart, 1997
- [9] Object Management Group: *Corba Services Book*
<http://www.omg.org/library/csindex.html>
- [10] Rothermel, Kurt; Straßer, Markus: *A Protocol for Preserving the Exactly-Once Property of Mobile Agents*
IPVR Universität Stuttgart, Bericht 1997/18