JTS

JAVA TRANSACTION SERVICE

unter besonderer Berücksichtigung der impliziten Kontextpropagierung

Autor: Michael Hurler

Seminar

Kommunikation in verteilten Systemen

Informationstechnologie Transfer Office

Technische Universität Darmstadt

Betreuer: Hartmut Vogler

Zusammenfassung

Diese Arbeit stellt eine Einführung in den „Java Transaction Service“ (JTS) dar. JTS ist die Java-Version des „Object Transaction Service“ (OTS), des Transaktionsdienstes für die Common Object Request Broker Architecture (CORBA), deren Standardisierung durch die Object Management Group (OMG) erfolgt.

Es werden Grundkenntnisse über Java, CORBA und den OTS vorausgesetzt, und auf diesen teilweise aufbauend wird das Mapping der CORBA Interface Definition Language auf Java in Grundzügen vorgestellt. Weiterhin wird aufgezeigt, wie die Umsetzung des OTS-Standards auf die Java Plattform erfolgt. Es werden Erweiterungen des JTS gegenüber der OTS-Spezifikation Version 1.0 vorgestellt, die zum Teil in der OTS-Spezifikation Version 1.1 enthalten sind. Des weiteren wird detailliert auf die implizite Kontextpropagierung des Transaktionskontextes eingegangen, die es ermöglicht, den Transaktionsservice ohne Änderungen der Schnittstellensignatur in bestehende Produkte einzubauen.

Den Abschluß bildet ein Beispielprojekt, das die Verwendung von CORBA in Verbindung mit dem Java Transaction Service unter Java vorstellt.

1. Inhaltsverzeichnis

1. Inhaltsverzeichnis ____ *I*

2. Literaturverzeichnis ____ *III*

3. Abkürzungsverzeichnis ____ *IV*

4. Einführung ____ *1*

5. JTS und das IDL/Java Language Mapping ____ *4*

5.1. IDL/Java Language Mapping der OMG ____ **4**

5.2. PIDL/Java Language Mapping ____ **5**

5.3. Das IDL/Java-Mapping des OTS ____ **6**

5.3.1. Das Interface „Control“ ____ 6

5.3.2. Das Interface „Coordinator“ ____ 7

5.3.3. Das Interface „Terminator“ ____ 7

5.3.4. „TransIdentity“ ____ 8

5.3.5. „PropagationContext“ ____ 8

5.4. Exkurs: Wie steuern Applikationen Transaktionen mittels JTS/OTS? ____ **8**

5.4.1. Kontextmanagement ____ 8

5.4.2. Kontextpropagierung ____ 9

5.5. Das PIDL/Java-Mapping des OTS ____ **9**

5.5.1. „Current“-Pseudointerface ____ 10

5.5.2. „CosTSPortability“-Pseudomodul ____ 10

5.6. Beispiel ____ **11**

5.6.1. Hello.idl ____ 11

5.6.2. Hello.java ____ 11

5.6.3. _HelloStub.java ____ 11

5.6.4. _HelloImplBase.java ____ 12

5.6.5. HelloHolder.java ____ 13

5.6.6. HelloHelper.java ____ 14

6. Erweiterungen gegenüber OTS 1.0 ____ *15*

6.1. Das „Synchronization“-Interface ____ **15**

6.2. Unterstützung mehrerer Kommunikationsmanager ____ **15**

6.3. Implizite Kontextpropagierung über RMI ____ **16**

7. Die implizite Kontextpropagierung ____ *17*

7.1. Kommunikation zwischen TS und ORB ____ **17**

7.2. Kontextpropagierung von ORB zu ORB ____ **19**

7.3. Das Zusammenspiel mehrerer TS ____ **19**

8. Beispiel für JTS ____ *21*

8.1. Aufbau des Beispiels ____ **21**

8.2. IDL-Definition ____ **22**

8.3. Der Transactional Client ____ **23**

8.4. Das Transactional Object (Umbucher) ____ **24**

8.5. Das erste Recoverable Object inkl. Ressource (KontoObjekt1 & KontoObjekt1Res) ____ **27**

8.6. Das zweite Recoverable Object inkl. Ressource (KontoObjekt2 & KontoObjekt2Res) ____ **31**

9. Resümee ____ *34*

2. Literaturverzeichnis

Java Transaction Service

[JTS96] **JTS: A Java Transaction Service API Version 0.5**
Autoren: Vlada Matena, Rick Cattell
Sun Microsystem / Javasoft
10. Dezember 1996
Quelle: http://www.javasoft.com/products/jts

Object Transaction Service

[OTS95] **Transaction Service Specification Version 1.0**
aus: CORBA Services: Common Object Services Specification
März 1995
Quelle: http://www.omg.org

[OTS97] **Transaction Service Specification Version 1.1**
aus: CORBA Services: Common Object Services Specification
November 1997
Quelle: http://www.omg.org

[BullTS] Dokumentation zur Bull OTS-Implementierung OrbTP für den Iona-ORB Orbix

CORBA

[OMG98] **The Common Object Request Broker – Architecture and Specification**
Revision 2.2 – Februar 1998
OMG
Quelle: http://www.omg.org

IDL/Java Language Mapping

[IDL97] **IDL/Java Language Mapping**
Joint Revised Submission
OMG TC Document
19. März 1997
Quelle: http://www.omg.org

3. Abkürzungsverzeichnis

2PC ..... Two-Phase-Commit

CORBA..... Common Object Request Broker Architecture

GIOP ..... General Inter-ORB Protocol

IDL..... Interface Definition Language

IIOP..... Internet Inter-ORB Protocol

JTS ..... Java Transaction Service

OMG ..... Object Management Group

ORB ..... Object Request Broker

OTS..... Object Transaction Service

PIDL..... Pseudo Interface Definition Language

RMI..... Remote Method Invocation

TS..... Transaction Service

4. Einführung

Bei der Abwicklung geschäftlicher Vorgänge, wie zum Beispiel der Buchung zwischen Konten, ist es unumgänglich sicherzustellen, daß ein solcher geschäftlicher Vorgang entweder vollständig oder gar nicht abgewikkelt wird, d.h. es ist ein Transaktionskonzept notwendig, das die sogenannte Atomarität, die Unteilbarkeit einer Transaktion garantiert. Dieses Konzept ist beispielsweise bei Datenbanken Standard. Es gibt allerdings auch Anwendungssoftware, bei der es zwingend ist, die Atomarität bestimmter interner Vorgänge zu garantieren, um einen konsistenten Systemzustand zu erhalten.

Auf der anderen Seite gibt es Softwaresysteme, die in verteilen, heterogenen Umgebungen arbeiten und dies durch die Verwendung des CORBA-Standards erreichen. Dieser Standard definiert ein Umfeld für Applikationen, das es durch die Verwendung sogenannter Object Request Broker und vordefinierter Dienste ermöglicht, ein transparentes Aufrufen bzw. Anbieten von „Diensten“ und Senden bzw. Empfangen von Antworten in besagten verteilten und heterogenen Umgebungen durchzuführen.

Aufgrund der Bedeutung von Transaktionen wurde ein entsprechender Dienst für CORBA definiert, der Object Transaction Service. Dieser bietet eine standardisierte Möglichkeit, ein Two-Phase-Commit-Protokoll im Umfeld von CORBA zu implementieren. Darüber hinaus bietet OTS weiterhin noch Interoperabilität mit dem bereits etablierten Transaktionsstandard XOpen/DTP der XOpen-Gruppe,[1] der von einer Vielzahl von Standardsoftware unterstützt wird, nicht zuletzt von den meisten Datenbanken. Dadurch erweitern sich die Einsatzmöglichkeiten ungemein, und insbesondere die Eingliederung in eine bestehende Infrastruktur wird deutlich vereinfacht.

Nachdem die Programmiersprache Java bzw. deren Interface Definition Language-Mapping in den CORBA-Standard aufgenommen wurde, Java zusätzlich immer häufiger in heterogenen Umgebungen verwendet wird und schließlich CORBA, in Form eines ORBs, eines IDL-Compilers und der entsprechenden Schnittstellen, in der Sprachversion 1.2 in den Java-Standard aufgenommen wurde, war es nur noch eine Frage der Zeit, bis mit der Umsetzung des OTS in Java begonnen würde, denn ein Transaktionskonzept ist, wie einleitend erläutert, in nahezu allen Umgebungen notwendig.

Diese Umsetzung ist der Java Transaction Service. Er stellt das Mapping des OTS nach Java dar oder wie die Javasoftentwickler es nennen: JTS ist OTS für Java.[2] Die Anforderungen an JTS sind die gleichen wie an OTS, jedoch mit dem besonderen Anspruch der Interoperabilität von Applikationen. Dies ist deshalb erforderlich, weil z.B. Java-Applets von einem Anwender über das Internet auf seinen Rechner geladen und dort ausgeführt werden. In der Systemumgebung des Anwenders muß dieses Applet nun sowohl mit der dortigen ORB-Implementierung als auch z.B. mit der dortigen JTS-Implementierung zusammenarbeiten können. Dies ist eine

---

[1] siehe [OTS97], Seite 10-67 ff., Abschnitt 10.5.3

[2] siehe [JTS96], Seite 6, Abschnitt 2.3

besondere Anforderung an JTS, die bei nicht Java-basierten OTS-Anwendungen in dieser Form nicht erforderlich ist.

Die Anwendungsarchitektur, die JTS zugrunde liegt, ist identisch mit der von OTS: Ein Anwendungsprogramm, das Zugriff auf einen Transaction Service (TS) hat, startet eine Transaktion und greift dann im Rahmen dieser Transaktion auf ein CORBA-Objekt zu, das das JTS/CORBA-Interface *TransactionalObject* implementiert. Dieses Objekt kann wiederum auf andere transaktionale Objekte zugreifen, die dann i.d.R. auch Teil der Transaktion werden. Transaktionale Objekte können wiederum Objekte, die das Interface *Resource* implementieren, sogenannte Ressourcen, beim TS registrieren, die dann

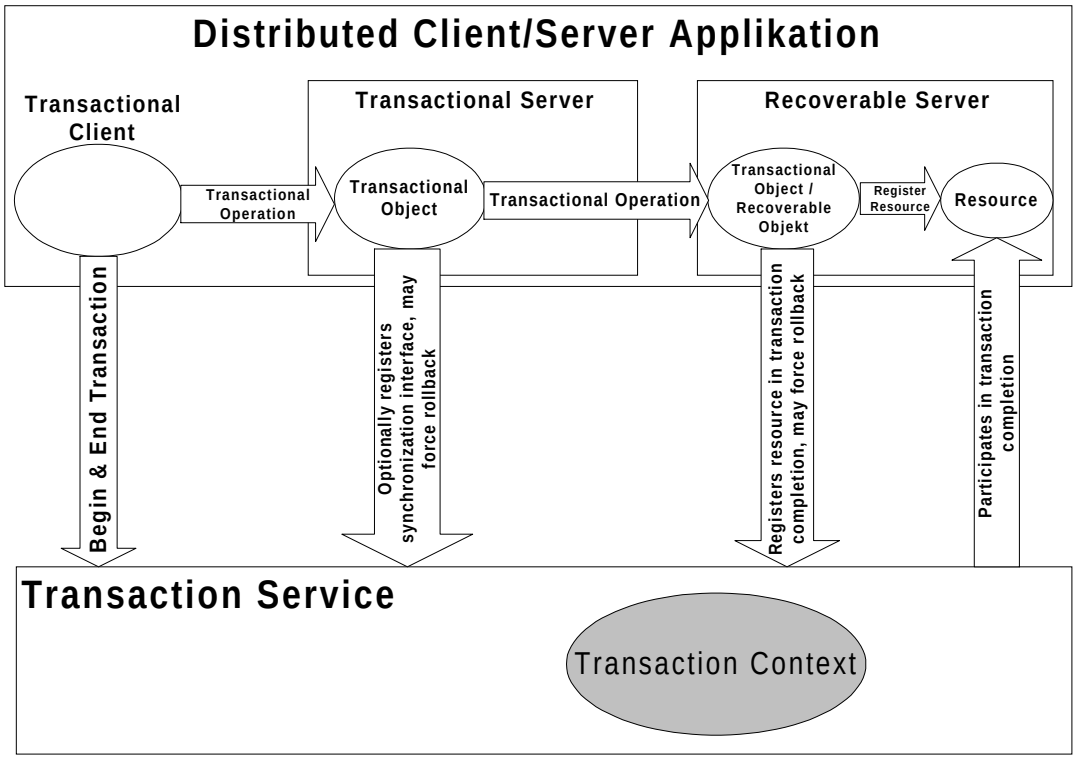

Abbildung 1 Transaktionales Anwendungssystem

vom TS im Falle des Starts des Two-Phase-Commit-Protokolls, “befragt” werden, ob sie für ein Commit oder für ein Rollback votieren. Wenn alle Ressourcen für Commit votieren, werden die Ressourcen aufgefordert, die Änderungen an den Daten, mit denen sie assoziiert sind, in den jeweiligen Datenquellen permanent zu speichern, im Falle eines Rollbacks müssen die Ressourcen dafür sorgen, daß keine Änderungen an den Daten im Verhältnis zu ihrem Zustand vor dem Start der Transaktion vorhanden sind. Wie die Ressourcen das Commit- oder Rollbackverhalten implementieren ist nicht Teil der TS-Spezifikation. Beim Zugriffs auf eine Datenbank könnte eine Ressource den Transaktionsmechanismus der Datenbank nutzen, um Änderungen auf der Datenbank permanent zu machen respektive die Änderungen zurückzusetzen. Eine Ressource, die nicht auf eine Datenbank zugreift, könnte eine Art “Schattenkopie” der alten Daten speichern und so durch Zurückschreiben dieser Kopien die Änderungen im Falle eines Rollbacks rückgängig machen.

Der *Transactional Client*, alle *Transactional Objects* und die *Ressourcen* können auf den Ausgang einer Transaktion Einfluß nehmen, indem sie das 2PC-Protokoll starten oder die Transaktion zurücksetzen. Gestartet wird das 2PC-Protokoll allerdings üblicher Weise nur vom *Transactional Client*, abgebrochen werden kann eine Transaction sowohl vom *Transactional Client* oder aber, im Rahmen des 2PC-Protokolls, von den *Ressourcen*.

Ein *Transactional Object* wird auch *Recoverable Object* genannt, wenn es eine Ressource beim TS registriert.

5. JTS und das IDL/Java Language Mapping[3]

Im folgenden eine kurze Einführung in das Mapping, d.h. die Umsetzung der CORBA Interface Definition Language (IDL) nach Java.

Die IDL enthält die Definitionen der Schnittstellen, Datentypen und –strukturen von CORBA-Objekten. Diese IDL wird mittels eines IDL-Compilers in Programmcode einer bestimmten Ziel-Programmiersprache übersetzt. Für eine Reihe von Programmiersprachen, wie z.B. C++, Smalltalk und neuerdings auch Java existieren genormte Mappings, so daß jeder IDL-Compiler eines jeden Herstellers das gleiche Mapping erzeugen sollte. Außerdem haben Hersteller verschiedener ORBs für weitere Sprachen eigene Mappings definiert und so auch diese Sprachen in die Lage versetzt, über einen ORB mit anderen Objekten in einem System in Verbindung zu stehen.

Um nicht den Rahmen einer Einführung zu sprengen, wird ausschließlich auf den Punkt des Mappings von CORBA Interfaces, also den Schnittstellendefinitionen, eingegangen. Es soll damit eine Grundidee vermittelt werden, wie das Mapping von IDL nach Java definiert wurde.

Dazu wird im folgenden ein Beispiel verwendet, das ein Interface „Hello“ innerhalb eines Moduls „HelloApp“ beinhaltet.

5.1. IDL/Java Language Mapping der OMG[4]

IDL-Module werden in Java auf Java-Packages gemapped.

Aus einem IDL-Interface namens „Hello“, das außer einer Interface-Definition nichts enthält, werden vom IDL-to-JAVA-Compiler fünf Dateien generiert. Dies sind:

- ein Java-Interface namens „Hello“
- eine Stub-Klasse, namens „_HelloStub“
- eine abstrakte Klasse „_HelloImplBase“, als Vorfahr der Implementierung der Klasse, die die in der IDL definierte Funktionalität enthält. Diese Klasse nimmt somit die Rolle des bei CORBA üblichen Skeleton wahr.
- eine Java-Klasse namens „HelloHolder“
- eine Java-Klasse namens „HelloHelper“

---

[3] siehe [IDL97]

[4] siehe [IDL97], Kapitel 5, Abschnitte 5.1, 5.3, 5.5, 5.12

Das Java-Interface „Hello“ entspricht dem in der IDL definierten Interface. Der Stub dient als Schnittstelle und Bridge auf der Seite eines Clients um die Funktionen auf einem CORBA-Objekt des Typs „Hello“ aufzurufen ohne zwischen lokalen und entfernten Aufrufen unterscheiden zu müssen. Das Skeleton dient als Grundlage für die eigentliche Implementierung der in der IDL definierten Funktionen. Die eigentliche Implementierung erbt i.d.R. von dieser Klasse. Für den Fall, daß die „Implementierung“ des Objekts schon von einer anderen Klasse erben muß, gibt es eine weitere Compilierungsoption, die ein Skeleton liefert, in das direkt der Programmcode eingefügt werden muß, so daß man von einer anderen Klasse erben kann. Dies ist notwendig, da Java keine Mehrfachvererbung kennt.

Die Hilfsklasse „HelloHolder“ ist notwendig, um im Falle von out- bzw. inout-Parametern in der IDL eine „Hello“-Objektreferenz zurückgeben zu können. Da Java Objektreferenzen nur als Wertparameter („call by value“) übergibt, wäre es normalerweise nicht möglich eine geänderte Objektreferenz zurückzugeben. Dies wird aber dadurch ermöglicht, daß die eigentliche Objektreferenz, die, z.B. im Falle eines out-Parameters, zurückgegeben werden soll, in der Holderklasse gespeichert wird. Die Konstruktoren der „Holderklasse“ spiegeln diese Funktion ebenfalls wieder. Es gibt sowohl einen Konstruktor, der eine „leere“ Holderklasse erzeugt und somit als Container für out-Parameter dient und es gibt einen Konstruktor, der eine „Holderklasse“ erzeugt, die sofort eine Referenz auf ein Objekt enthält. Dieser Konstruktor wird bei inout-Parametern verwendet. Gemäß des Java-Language-Mappings werden in-Parameter auf ganz normale Java-Parameter abgebildet.

Die Hilfsklasse „HelloHelper“ stellt eine Reihe von statischen Java-Methoden zur Verfügung, die beim Umgang mit einem CORBA-Objekt des „Hello“-Typs benötigt werden könnten. Dies sind Methoden, um ein Hello-Objekt aus einem Stream zu lesen bzw. das Objekt in einen Stream zu schreiben, um somit eine Übertragung des Objekts über ein Netzwerk zu ermöglichen (Methoden: read und write). Es gibt Methoden, um aus einem „Hello“-Objekt ein CORBA Any-Objekt zu erzeugen und umgekehrt (Methoden: insert und extract). Schließlich gibt es eine Methode „narrow“, die aus einer simplen CORBA-Objektreferenz auf ein „Hello“-Objekt eine „Hello“-Objektreferenz generiert.

5.2. PIDL/Java Language Mapping[5]

Eine Reihe von Objekten respektive deren Interfaces, die im CORBA-Standard definiert sind, werden zwar mittels IDL definiert, halten aber in dieser Form nie Einzug in die Implementierungen, sondern werden häufig sogar sprachabhängig gemapped.[6] Man spricht bei diesen Objekten von Pseudoobjekten, die in Pseudo IDL (PIDL) definiert werden. Diese Objekte sind i.d.R. keine „echten“ CORBA-Objekte, was insbesondere bedeutet, daß sie nicht im Interface Repository repräsentiert werden, daß es für sie keine Holder- und Helperklassen gibt und daß sie auf eine Java „public abstract class“ gemapped werden, die von keiner anderen Klasse und von keinem anderen Interface erbt oder dieses erweitert.[7] Der häufigste Grund für die Verwendung von Pseudoob-

---

[5] siehe [IDL97], Kapitel 6

[6] siehe [IDL97], Kapitel 6.1, S. 6-53, Absatz 1

[7] siehe [IDL97], Kapitel 6.1.1, S 6.53

jekten ist Effizienz. Zugriffe auf CORBA-Objekt-Methoden sind i.d.R. immer langsamer als ein einfacher, lokaler Prozeduraufruf. Daher werden bestimmte Objekte, die häufig verwendet werden, von Anfang an so konzipiert, daß sich ihr Gebrauch stark optimieren läßt. Einer Einschränkung unterliegen jedoch Pseudoobjekte. Sie können nicht über Rechnergrenzen hinweg verwendet werden, weil sie, wie bereits dargestellt, keine „echten“ CORBA-Objekte sind.

Bei der Verwendung von Pseudoobjekten erscheinen diese häufig wie ein eigenständiger Prozeß, eine thread-lokale Variable oder wie ein Programmiersprachenkonstrukt.[8]

Alle Pseudointerfaces werden so gemapped, als seien sie in

```
module org {
      module omg {
            module CORBA {
                  ...
```

deklariert, d.h. sie werden in das Package ‚org.omg.CORBA‘ gemapped.

5.3. Das IDL/Java-Mapping des OTS

Das IDL/Java-Mapping des OTS entspricht ohne Ausnahme dem Standardmapping von IDL nach Java. Daher wird auf dieses Mapping nicht weiter eingegangen.

Zur Erinnerung werden jedoch noch einmal in Kürze die wichtigsten Interfaces bzw.Records von OTS erläutert. Außerdem werden die Datenstrukturen zur impliziten Kontextpropagierung vorgestellt.

5.3.1. Das Interface „Control“[9]

Das „Control“-Objekt dient dazu, das „Coordinator“-Objekt und das „Terminator“-Objekt“ einer Transaktion durch entsprechende Methodenaufrufe zu ermitteln. Dies geschieht mittels der Methoden **get_coordinator** und **get_terminator**. Das „Control“-Interface kann bei explizitem Kontextmanagement – dazu im nächsten Abschnitt mehr - verwendet werden, um den Transaktionskontext zu übermitteln. Eine weitere Möglichkeit diesen zu übergeben besteht darin, das „Coordinator“-Objekt, das im nächsten Abschnitt beschrieben wird, zu übergeben. Dadurch ist es einem Objekt, das „nur“ das „Coordinator“-Objekt erhält, unmöglich eine Transaktion selbständig zu beenden bzw. abzubrechen.

---

[8] siehe [IDL97], Kapitel 6.1, S. 6-53, Absatz 1

[9] siehe [OTS97], Kapitel 10.3.3, S. 10-22 ff.
siehe [JTS96], Kapitel 4, S. 10
siehe [JTS96], Appendix D, S. 32 ff.
siehe [JTS96], Appendix E, S. 43 ff.

5.3.2. Das Interface „Coordinator“[10]

Das „Coordinator“-Objekt dient dazu, Koordinationsaufgaben innerhalb einer Transaktion durchzuführen. Es besteht über Methoden wie **get_status**, **get_parent_status** oder **get_top_level_status** die Möglichkeit, den Status von Transaktionen abzufragen. Es wird einem Aufrufer dieser Funktionen dann mitgeteilt, ob eine Transaktion noch aktiv („Active“) ist, ob sie für ein Rollback gekennzeichnet ist („Marked_Rollback“), ob für die Transaktion schon die Preparephase des 2PC abgeschlossen („Prepared“) oder gar die Transaktion schon abgeschlossen („Commited“) ist. Des weiteren besteht die Möglichkeit, daß der Aufrufer „erfährt“, daß die Transaktion, nach deren Status er fragt, dem aktuellen Transaktionsservice nicht bekannt ist („Unknown“) oder daß gar keine Transaktion existiert („NoTransaction“).

Des weiteren wird ein „Coordinator“-Objekt dazu verwendet, über **register_resource** eine Ressource zu registieren, die dann später am 2PC-Protokoll teilnimmt. Auch kann der Transaktionsname (**get_transaction_name**) nachgefragt oder aber eine Subtransaktion (**create_subtransaction**) erzeugt werden. In der neusten Spezifikation Version 1.1 des OTS ist hier auch noch eine Methode **get_txcontext** aufgeführt, die dazu dient, einen speziellen Transaktionskontext, der auch bei der sogenannten impliziten Kontextpropagierung verwendet wird, abzufragen. Ob diese Funktion in die Version 1.0 der JTS-Spezifikation Einzug hält, ist bisher nicht zu klären.

Es stehen auch noch weitere Funktionen, wie zum Beispiel zum Unterbrechen und Wiederaufnehmen von Transaktionen zur Verfügung, auf deren Erläuterung hier verzichtet wird. Es wird dazu auf entsprechende Arbeiten zum Thema OTS verwiesen.

5.3.3. Das Interface „Terminator“[11]

Ein „Terminator“-Objekt stellt die Funktionen zur Vefügung, die ein Beenden (**commit**) bzw. Abbrechen (**rollback**) einer Transaktion veranlassen. Bei **commit** wird allerdings nur das 2PC-Protokoll initiiert. Ob die Transaktion erfolgreich abgeschlossen wird, hängt von den Vota der für diese Transaktion registrierten Ressourcen ab.

---

[10] siehe [OTS97], Kapitel 10.3.5, S. 10-24 ff.
siehe [JTS96], Kapitel 4, S. 10
siehe [JTS96], Appendix E, S. 47 ff.

[11] siehe [OTS97], Kapitel 10.3.4, S. 10-23 f.
siehe [JTS96], Kapitel 4, S. 10
siehe [JTS96], Appendix E, S. 85 ff.

5.3.4. „TransIdentity“[12]

Dieser Record speichert die Referenzen auf das „Coordinator“- und das „Terminator“-Objekt einer Transaktion. Er dient dem TS zur Verwaltung und impliziten Übermittlung eines Transaktionskontextes.

5.3.5. „PropagationContext“[13]

Dieser Record speichert die **TransIdentity**, eine Transaktions-ID, den Timeout-Wert der Transaktion, Informationen über die Vatertransaktion einer Transaktion und implementierungsspezifische Daten. Es handelt sich hierbei um die Datenstruktur, die als Transaktionskontext bei impliziter Kontextpropagierung übermittelt wird.

5.4. Exkurs: Wie steuern Applikationen Transaktionen mittels JTS/OTS?

An dieser Stelle soll ein kurzer Exkurs eingeschoben werden, um die Möglichkeiten darzustellen, wie ein Transactional Client oder ein Transactional Object eine Transaktion über JTS / OTS steuern bzw. verwalten kann. Dabei werden zwei spezielle Aspekte unterschieden: zum einen der Aspekt Kontextmanagement, zum anderen der Aspekt Kontextpropagierung. Insbesondere das Verständnis der Kontextpropagierung ist erforderlich um den weiteren Darstellungen dieser Arbeit folgen zu können.
In der Praxis treten die im folgenden beschriebenen Möglichkeiten selten in reiner Form auf, vielmehr werden meist Mischformen verwendet, die sich aus der Einfachheit der Nutzung einerseits und der mangelnden Mächtigkeit der einfachen Schnittstellen auf der anderen Seite ergeben.

5.4.1. Kontextmanagement[14]

Unter dem Aspekt Kontextmanagement unterscheidet man die beiden Fälle des direkten und des indirekten Kontextmanagements.

Beim **direkten Kontextmanagement** bedient sich ein, auf eine Transaktion Einfluß nehmendes Objekt, der „Control“-, „Coordinator“- und „Terminator“-Objekte, um eine Transaktion zu verwalten, d.h. ihren Start und ihr Ende zu beeinflussen oder z.B. den Status einer Transaktion zu erfragen. Schon an dieser Stelle soll ange-

---

[12] siehe [OTS97], Kapitel 10.2.3, S. 10-14 ff.
siehe [JTS96], Kapitel 4.2, S. 11
siehe [JTS96], Kapitel 5.2, S. 12 ff.

[13] siehe [OTS97], Kapitel 10.2.3, S. 10-14 ff.
siehe [JTS96], Kapitel 4.2, S. 11
siehe [JTS96], Kapitel 5.2, S. 12 ff.

[14] siehe [OTS97], Kapitel 10.2.3, S. 10-14 ff.

merkt werden, daß nur über direktes Kontextmanagement die Verwaltung von Subtransaktionen möglich ist. Und auch die Registrierung einer Ressource durch ein Recoverable Object kann nur über das „Coordinator“-Interface erfolgen. Soll bei Verwendung von direktem Kontextmanagement einen neue Transaktion gestartet werden, so wird dies durch den Aufruf einer entsprechenden Transaction-Factory erreicht.

Beim **indirekten Kontextmanagement** erfolgt die Verwaltung und Steuerung der Transaktion nur über das, im nächten Abschnitt erläuterte, „Current“-Interface. Ein Transactional Client oder ein Transactional Object kann damit prinzipiell alle zwingend notwendigen TS-Aufrufe durchführen.

5.4.2. Kontextpropagierung

Unter dem Aspekt Kontextpropagierung versteht man die Weitergabe des Kontextes, in dem die Transaktion abläuft, zwischen Objekten, die an einer Transaktion beteiligt sind. Dieser Kontext enthält Informationen, die notwendig sind, um Objekte an einer Transaktion teilnehmen zu lassen.

Bei der **expliziten Kontextpropagierung** wird der Transaktionskontext als Teil der Signatur der Methoden explizit übergeben. Dies bedeutet, daß entweder das zu der Transaktion gehörende „Control“-Objekt oder auch nur das „Coordinator“-Objekt übergeben wird. Die Methode des Objekts, das aufgerufen wird, kann dann diese Objekte nutzen, um die Transaktion zu beeinflussen. Die Verwaltung der an der Transaktion teilnehmenden Objekte obliegt so ausschließlich den Objekten, die an der Transaktion teilnehmen.

Bei der **impliziten Kontextpropagierung** ist der Transaktionskontext nicht Teil der Signatur von Aufrufen. Statt dessen wird hierbei ein in einem nachfolgenden Kapitel erläutertes Protokoll zur automatischen Weitergabe des Transaktionskontextes verwendet. Dieser Transaktionskontext ist der oben bereits erläuterte **PropagationContext**. Durch die automatische Weitergabe des Transaktionskontextes ist es sogar im Gegensatz zur expliziten Kontextpropagierung möglich, eine Anzahl von Optimierungen in die Transaktionsaufrufe, z.B. beim Zugriff auf das „Current“-Objekt, einzubauen.

5.5. Das PIDL/Java-Mapping des OTS

Im folgenden soll auf die Pseudointerfaces und ein Pseudomodul von OTS und ihr Mapping in JTS eingegangen werden.

5.5.1. „Current“-Pseudointerface[15]

Das wichtigste Pseudointerface / -objekt ist das „Current“-Interface, das insbesondere bei impliziter Kontextpropagierung genutzt wird. Der Vorteil der Nutzung des „Current“-Interfaces gegenüber der Nutzung von „Current“, „Coordinator“ und „Terminator“ liegt darin, daß es sehr einfach und überschaubar ist ein Transaktionskonzept zu verwirklichen, da die Schnittstellen sehr klein gehalten sind. Bei Verwendung von impliziter Kontextpropagierung findet eine Zuordnung der Transaktion auf einer „per Thread“-Basis statt, d.h. ein Thread ist, wenn überhaupt, zu einem Zeitpunkt genau einer Transaktion zugeordnet.

Das „Current“-Interface wird in Java auf das Interface „java.CosTransactions.current“ gemapped. Die Implementierung wird, zumindest zum aktuellen Spezifiaktionsstand, dem jeweiligen Entwickler überlassen.

Das „Current“-Interface umfaßt im wesentlichen folgende Funktionen:

- **Begin**, um eine neue Transaktion zu starten.
- **Commit**, um das 2PC-Protokoll zum Beenden einer Transaktion zu starten.
- **Rollback**, um eine Transaktion zurückzusetzen.
- **get_status**, um den Status der dem „Current“-Objekt zugeordneten Transaktion abzufragen.
- **set_timeout**, um einen Timeout-Wert für die Transaktion zu bestimmen, nach dem die Transaktion automatisch zurückgesetzt wird.
- **get_control**, um das zu der dem „Current“-Objekt zugeordneten Transaktion gehörende „Control“-Objekt abzufragen.
- **suspend**, um die aktuelle Transaktion zu unterbrechen.
- **resume**, um eine unterbrochene Transaktion wieder aufzunehmen.

5.5.2. „CosTSPortability“-Pseudomodul[16]

Das Mapping von „CosTSPortability“ ist in JTS ebenso wie in OTS implementierungsabhängig, zumindest existiert hierfür bisher kein Mapping. Das Modul enthält Interfaces und andere Definitionen, die der Kommunikation zwischen einem ORB und einem TS zur Übermittlung des impliziten Transaktionskontextes dienen. Zu nennen sind hier insbesondere die beiden Interfaces „Sender“ und „Receiver“, die auf der Seite des TS eine Schnittstelle für den Aufruf durch einen ORB bieten und das „TSIdentification“-Interface, das auf der Seite des ORBs eine Schnittstelle für die Identifikation des „Sender“- und des „Receiver“-Interfaces des TS bietet. Die

---

[15] siehe [OTS97], Kapitel 10.3.1, S. 10-18 ff.
siehe [JTS96], Kapitel 5.1, S. 12
siehe [JTS96], Appendix D.1, S. 32 ff.

[16] siehe [OTS97], Kapitel 10.6.1, Seite 10-73
siehe [JTS96], Kapitel 5.3, S. 13 ff.
siehe [JTS96], Appendix D, S. 35 ff.

Kommunikation zwischen ORB und TS erfolgt danach über Funktionen, die zu den „Sender“- und „Receiver“-Interfaces gehören. Es wird also über die Bekanntgabe dieser Interfaces ein Callback zwischen ORB und TS eingerichtet.

Weitere Einzelheiten zum Aufbau der Kommunikation zur impliziten Kontextpropagierung werden später in dieser Arbeit erläutert.

5.6. Beispiel

Im folgenden ist ein Beispiel zum Mapping von IDL nach Java aufgeführt. Das Beispiel sollte mit den vorausgehenden Erläuterungen verständlich sein.

5.6.1. Hello.idl

```
module HelloApp
{
    interface Hello
    {
        string sayHello();
    };
};
```

5.6.2. Hello.java

```
/*
 * File: ./HELLOAPP/HELLO.JAVA
 * From: HELLO.IDL
 * Date: Wed Oct 29 14:29:04 1997
 *   By: D:\JAVA\JDK1~1.2BE\BIN\IDLTOJ~1.EXE Java IDL 1.2 Nov 10 1997
13:52:11
 */

package HelloApp;
public interface Hello
    extends org.omg.CORBA.Object {
    String sayHello()
;
}
```

5.6.3. _HelloStub.java

```
/*
 * File: ./HELLOAPP/_HELLOSTUB.JAVA
 * From: HELLO.IDL
 * Date: Wed Oct 29 14:29:04 1997
 *   By: D:\JAVA\JDK1~1.2BE\BIN\IDLTOJ~1.EXE Java IDL 1.2 Nov 10 1997
13:52:11
 */
```

```
package HelloApp;
public class _HelloStub
      extends org.omg.CORBA.portable.ObjectImpl
      implements HelloApp.Hello {

    public _HelloStub(org.omg.CORBA.portable.Delegate d) {
          super();
          _set_delegate(d);
    }

    private static final String _type_ids[] = {
        "IDL:HelloApp/Hello:1.0"
    };

    public String[] _ids() { return (String[]) _type_ids.clone(); }

    //      IDL operations
    //          Implementation of ::HelloApp::Hello::sayHello
    public String sayHello()
 {
            org.omg.CORBA.Request r = _request("sayHello");
            r.set_return_type(org.omg.CORBA.ORB.init().get_primitive_tc
                                        (org.omg.CORBA.TCKind.tk_string));
            r.invoke();
            String __result;
            __result = r.return_value().extract_string();
            return __result;
   }

};
```

5.6.4. _HelloImplBase.java

```
/*
 * File: ./HELLOAPP/_HELLOIMPLBASE.JAVA
 * From: HELLO.IDL
 * Date: Wed Oct 29 14:29:04 1997
 *   By: D:\JAVA\JDK1~1.2BE\BIN\IDLTOJ~1.EXE Java IDL 1.2 Nov 10 1997
13:52:11
 */

package HelloApp;
public abstract class _HelloImplBase extends
            org.omg.CORBA.DynamicImplementation implements HelloApp.Hello {
    // Constructor
    public _HelloImplBase() {
         super();
    }
    // Type strings for this class and its superclases
    private static final String _type_ids[] = {
        "IDL:HelloApp/Hello:1.0"
    };

    public String[] _ids() { return (String[]) _type_ids.clone(); }

    private static java.util.Dictionary _methods = new
                                                    java.util.Hashtable();
    static {
```

```
      _methods.put("sayHello", new java.lang.Integer(0));
     }
    // DSI Dispatch call
    public void invoke(org.omg.CORBA.ServerRequest r) {
       switch (((java.lang.Integer) _methods.get(r.op_name())).intValue())
{
           case 0: // HelloApp.Hello.sayHello
              {
              org.omg.CORBA.NVList _list = _orb().create_list(0);
              r.params(_list);
              String ___result;
                           ___result = this.sayHello();
              org.omg.CORBA.Any __result = _orb().create_any();
              __result.insert_string(___result);
              r.result(__result);
              }
              break;
            default:
              throw new org.omg.CORBA.BAD_OPERATION(0,
                          org.omg.CORBA.CompletionStatus.COMPLETED_MAYBE);
       }
 }
}
```

5.6.5. HelloHolder.java

```
/*
 * File: ./HELLOAPP/HELLOHOLDER.JAVA
 * From: HELLO.IDL
 * Date: Wed Oct 29 14:29:04 1997
 * By: D:\JAVA\JDK1~1.2BE\BIN\IDLTOJ~1.EXE Java IDL 1.2 Nov 10 1997
                                                                  13:52:11
 */

package HelloApp;
public final class HelloHolder
     implements org.omg.CORBA.portable.Streamable{
    //      instance variable
    public HelloApp.Hello value;
    //      constructors
    public HelloHolder() {
      this(null);
    }
    public HelloHolder(HelloApp.Hello __arg) {
      value = __arg;
    }

    public void _write(org.omg.CORBA.portable.OutputStream out) {
        HelloApp.HelloHelper.write(out, value);
    }

    public void _read(org.omg.CORBA.portable.InputStream in) {
        value = HelloApp.HelloHelper.read(in);
    }

    public org.omg.CORBA.TypeCode _type() {
        return HelloApp.HelloHelper.type();
    }
}
```

5.6.6. HelloHelper.java

```
/*
 * File: ./HELLOAPP/HELLOHELPER.JAVA
 * From: HELLO.IDL
 * Date: Wed Oct 29 14:29:04 1997
 *   By: D:\JAVA\JDK1~1.2BE\BIN\IDLTOJ~1.EXE Java IDL 1.2 Nov 10 1997
13:52:11
 */

package HelloApp;
public class HelloHelper {
     // It is useless to have instances of this class
     private HelloHelper() { }

    public static void write(org.omg.CORBA.portable.OutputStream out,
       HelloApp.Hello that) {
        out.write_Object(that);
    }
    public static HelloApp.Hello read(org.omg.CORBA.portable.InputStream
in) {
        return HelloApp.HelloHelper.narrow(in.read_Object());
    }
   public static HelloApp.Hello extract(org.omg.CORBA.Any a) {
     org.omg.CORBA.portable.InputStream in = a.create_input_stream();
     return read(in);
   }
   public static void insert(org.omg.CORBA.Any a, HelloApp.Hello that) {
     org.omg.CORBA.portable.OutputStream out = a.create_output_stream();
     write(out, that);
     a.read_value(out.create_input_stream(), type());
   }
   private static org.omg.CORBA.TypeCode _tc;
   synchronized public static org.omg.CORBA.TypeCode type() {
          if (_tc == null)
             _tc = org.omg.CORBA.ORB.init().create_interface_tc(id(),
"Hello");
      return _tc;
   }
   public static String id() {
       return "IDL:HelloApp/Hello:1.0";
   }
   public static HelloApp.Hello narrow(org.omg.CORBA.Object that)
          throws org.omg.CORBA.BAD_PARAM {
        if (that == null)
            return null;
        if (that instanceof HelloApp.Hello)
            return (HelloApp.Hello) that;
      if (!that._is_a(id())) {
          throw new org.omg.CORBA.BAD_PARAM();
      }
        org.omg.CORBA.portable.Delegate dup =
                ((org.omg.CORBA.portable.ObjectImpl)that)._get_delegate();
        HelloApp.Hello result = new HelloApp._HelloStub(dup);
        return result;
   }
}
```

6. Erweiterungen gegenüber OTS 1.0[17]

In diesem Abschnitt werden zwei Erweiterungen vorgestellt, die JTS gegenüber OTS 1.0 und OTS 1.1 vornimmt.

6.1. Das „Synchronization“-Interface[18]

Das „Synchronization“-Interface wurde in der OTS-Spezifikation Version 1.1 eingeführt, und schon vor deren Verabschiedung in JTS übernommen.

Das „Synchronization“-Interface dient dazu, es Objekten zu ermöglichen, sich vom TS benachrichtigen zu lassen bevor der TS die Prepare-Phase des 2PC beginnt (**before_completion**) und nachdem das Commit vollständig beendet wurde (**after_completion**).

Ein Grund für ein Objekt sich vom TS vor Beginn der Prepare-Phase informieren zu lassen, könnte z.B. sein, daß dieses Objekt zu diesem Zeitpunkt transiente Zustandsinformationen bei einem Recoverable Resource Manager speichern muß, um so diese Informationen für die Commit-Phase zu sichern.

6.2. Unterstützung mehrerer Kommunikationsmanager[19]

Ab der Version 1.2 liegen Java zwei Kommunikationsarchitekturen zugrunde. Das eine Protokoll, CORBA, dient zur Kommunikation mit der Nicht-Java-Außenwelt. Die zweite Architektur, die Remote Method Invocation (RMI), das Pendant zum Remote Procedure Call verschiedener Betriebssysteme, ist das von Sun bevorzugte Kommunikationsprotokoll innerhalb der „Java-Welt“. Prinzipiell kann jede Kommunikationsarchitektur unterstützt werden, die das Anfrage/Antwort-Paradigma verfolgt, also z.B. auch ein Netzwerk-JDBC-Treiber. OTS sieht jedoch prinzipiell nur eine unterstützte Architektur vor: CORBA.

Jede Kommunikation über ein anderes Protokoll als den ORB, wird als externe Propagierung bezeichnet.

Will man mehrere Kommunikationsarchitekturen unterstützen, so ist dafür sorgen, daß diese nicht in Konflikt geraten. Um dies zu vermeiden, darf jedoch kein OTS-Interface geändert werden, da sonst die Kompatibilität

---

[17] siehe [JTS96], Kapitel 6, S. 15 ff.

[18] siehe [JTS96], Kapitel 6.1, S.15
siehe [OTS97], Kapitel 10.1.4, S. 10-8
siehe [OTS97], Kapitel 10.3.8, S. 10-32
siehe [OTS97], Kapitel 10.5.1, S. 10-52 ff.

[19] siehe [JTS96], Kapitel 6.2, S. 16 ff.

mit OTS nicht mehr gewährleistet wäre. Aus diesem Grund stellt der TS für jeden Kommunikationsmanager ein eigenes, individuelles Paar von „Sender“/“Receiver“-Callback-Interfaces zur Verfügung. Dadurch geraten z.B. die Anfrage-IDs, die jede Anfrage an den TS identifizieren, mehrerer Kommunikationsmanager nicht durcheinander, denn es ist nicht garantiert, daß diese IDs getrennte „ID-Wertebereiche“ haben.

6.3. Implizite Kontextpropagierung über RMI[20]

Ein Problem, das bei Verwendung von RMI mit einem JTS noch zu lösen ist, ist die implizite Kontextpropagierung über RMI. Dazu wird eine neue Variante von RMI, das sogenannte Transactional RMI eingeführt. Dazu werden auf unterster Ebene des RMI-Protokolls die Kommunikationsobjekte „UnicastRef“ auf Clientseite und „UnicastServerRef“ auf Serverseite durch entsprechende transaktionale Objekte „TransactionalUnicastRef“ und „TransactionalUnicastServerRef“ ersetzt. Diese Objekte übergeben den impliziten Transaktionskontext über ein „Tunnelprotokoll“.

Das Standard-RMI-Protokoll wird dadurch nicht geändert. Das transaktionale Protokoll wird nur verwendet, wenn der RMI-Server dies so entscheidet. Dadurch entsteht auch bei nicht-transaktionalen RMI-Aufrufen kein Overhead.

[20] siehe [JTS96], Kapitel 7, S. 21 f.

7. Die implizite Kontextpropagierung[21]

Die implizite Kontextpropagierung bietet einige Vorteile gegenüber der expliziten Variante. Zum einen läßt sich eine Applikation leichter um ein Transaktionskonzept erweitern, wenn man nicht jede Schnittstelle ändern muß, damit man den Transaktionskontext übergeben kann. Zum anderen ergeben sich aus der Nutzung der impliziten Kontextpropagierung Optimierungsmöglichkeiten beim Zusammenspiel mehrerer Transactionservices, die bei der expliziten Kontextpropagierung nicht existieren. In diesem Kapitel soll nun ein Blick auf die Implementierung dieser impliziten Propagierung geworfen werden.

Im folgenden werden die drei Hauptaspekte, die sich aus diesem Thema ergeben betrachtet. Dabei geht es zunächst um die Kommunikation zwischen TS und ORB, danach um die Übermittlung des Transaktionskontextes von ORB zu ORB und zuletzt um die Zusammenarbeit mehrerer TS.

7.1. Kommunikation zwischen TS und ORB[22]

Zu Beginn des Zusammenspiels zwischen ORB und TS, ist es notwendig, daß die Kommunikationsschnittstelle

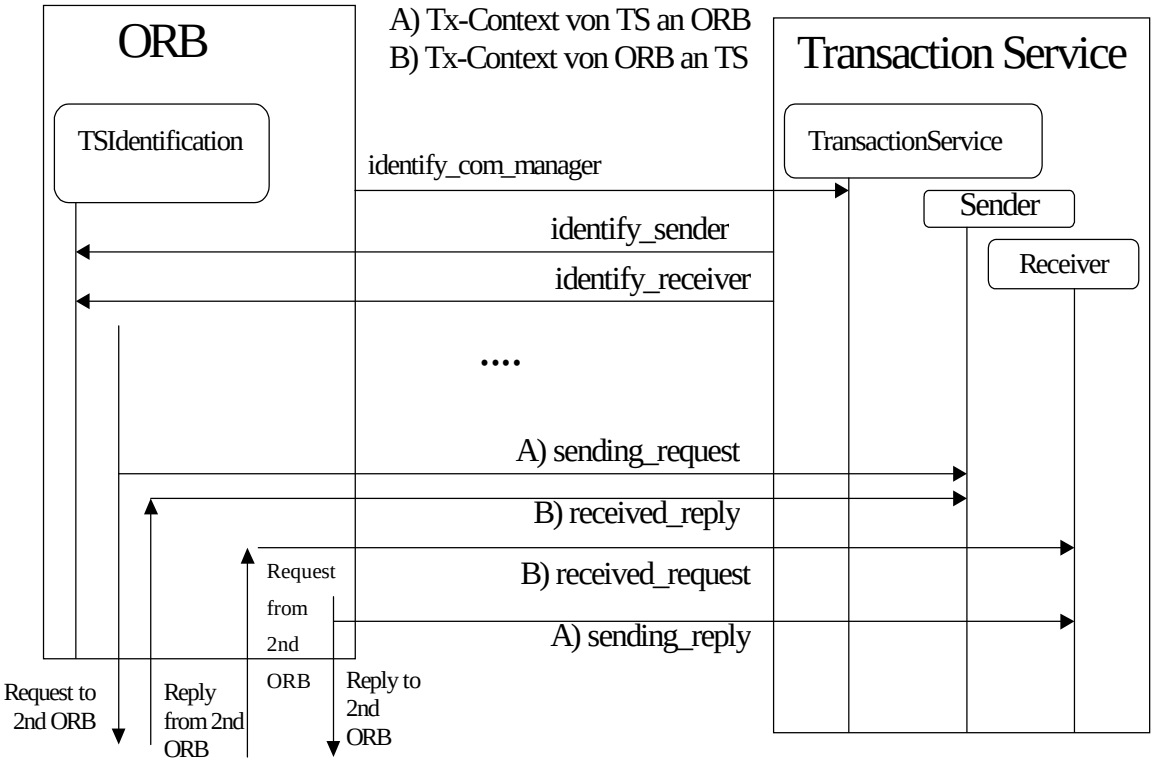

Abbildung 2 Kommunikation zwischen ORB und TS

[21] siehe [OTS97], Kapitel 10.5, S. 10-49 ff.

[22] siehe [OTS97], Kapitel 10.5.2, S. 10-64 ff.

zwischen ORB und TS bestimmt wird. Dies geschieht, indem beim Start der Zusammenarbeit zwischen ORB und TS der ORB an den TS die Aufforderung sendet, die Kommunikationsschnittstellen zu identifizieren (**identify_com_manager**). Daraufhin ruft der TS zwei Methoden aus dem „TSIdentification“-Interface des ORB auf. Der Methode **identify_sender** übergibt der TS die Objektreferenz seines „Sender“-Interfaces. Der Methode **idenity_reveiver** übergibt der TS die Adresse seines „Receiver“-Interfaces. Damit ist die Callback-Verbindung zwischen ORB und TS aufgebaut, und der ORB kann bei Bedarf mit dem TS kommunizieren.

Wenn nun der ORB einen Aufruf an ein Objekt in einem anderen Prozeß weiterleiten soll, so ruft er vor der Übertragung des Aufrufs an den anderen ORB die Methode **sending_request** des „Sender“-Interfaces des TS auf. Diesem Aufruf übergibt der ORB eine Request-ID, die der ORB bis zum Abschluß des Aufrufs an das andere Transactional Object beibehält. Der ORB erhält daraufhin von TS einen Transaktionskontext. Diesen Transaktionskontext übergibt der ORB im Rahmen des Aufrufs an den zweiten ORB. Wie dies geschieht, wird im nächsten Abschnitt erläutert.

Wenn nun der zweite ORB den Aufruf des bei ihm registrierten Transactional Object X erhält, so findet er auch den Transaktionskontext vor. Der ORB vergibt nun seinerseits eine Request-ID und ruft mit dieser und dem empfangenen Transaktionskontext die Methode **received_request** im „Receiver“-Interface des ihm zugeordneten TS auf. Ist das Zielobjekt ein Nachfahre von „TransactionalObject“, dann wird das Zielobjekt mit der Transaktion assoziiert. Ist dies geschehen, leitet der ORB den Aufruf an das Objekt X weiter. Wenn nun das Objekt X den Aufruf beendet, so wird vor der Übermittlung eines eventuellen Rückgabewertes an den, den Aufruf übermittelnden, ORB, wiederum das „Receiver“-Interface aufgerufen. Dieses Mal wird die Methode **sending_reply** aufgerufen. Diesem Aufruf wird zur Zuordnung des richtigen Requests die gleiche Request-ID wie beim ersten Aufruf übergeben. Der TS übergibt daraufhin dem ORB einen Transaktionskontext, den dieser zusammen mit dem Rückgabewert des Objektes X und dem entsprechenden CORBA-Environment an den ersten ORB übergibt. Dieser ordnet dem Rückgabewert die passende Request-ID zu und ruft nun seinerseits wieder das „Sender“-Interface des lokalen TS auf (**received_reply**). Er übergibt dem lokalen TS den Transaktionskontext des TS des aufgerufenen Objekts, sowie das mit dem Rückgabewert vom ersten ORB zurückgelieferte CORBA-Environment. Aus diesem sind eventuelle Exceptions auf der Seite des aufgerufenen Objekt ersichtlich. Wenn keine Ausnahmen aufgetreten sind, die das Zurücksetzen der Transaktion erfoderlich machen, übergibt danach der ORB den Rückgabewert an den Aufrufer.

Die OTS 1.1-Spezifikation besagt, daß Objekte im gleichen Prozeß sich implizit den gleichen Transaktionskontext teilen.[23] Dies steht im Widerspruch zur JTS 0.5-Spezifikation, die von Objekten im gleichen Thread spricht.[24] Eine endgültige Klärung konnte leider nicht erreicht werden, aber eine impliziete Assoziierung von Thread und Transaktionskontext erscheint plausibel, da ansonsten bei impliziter Propagierung mehrere Threads, z.B. eines Servers, immer nur mit einem Transaktionskontext assoziiert sein könnten. Auch konnte

[23] siehe [OTS97], Kapitel 10.5.2 „Requirements on the ORB“, S. 10-67

[24] siehe [JTS96], Kapitel 5.1.1, 1. Absatz, S. 12

nicht ermittelt werden, wann genau die Assoziation zwischen einem aufgerufenen Transactional Object und der dazugehörigen Transaktion vorgenommen wird.

7.2. Kontextpropagierung von ORB zu ORB[25]

Für das Funktionieren der impliziten Kontextpropagierung ist es notwendig, daß die ORBs auf die Unterstützung von Transaktionen ausgelegt sind („Transactional-aware ORB“). Die implizite Kontextpropagierung von ORB zu ORB basiert auf einen Mechanismus, der für die Kommunikation zwischen ORBs für die Übergabe servicespezifischer Kontextinformationen vorgesehen ist. Damit können zusammen mit einem Aufruf zusätzliche Informationen übermittelt werden. Dies soll im Falle von JTS natürlich der Transaktionsontext „PropagationContext“ sein. Dieser Mechanismus basiert auf folgenden Annahmen:

- Der zu übergebende Kontext ist vollständig in IDL definiert.
- Die ORB-APIs erlauben es Diensten zum richtigen Zeitpunkt während der Bearbeitung von Anfragen oder Antworten Kontextinformationen zu liefern bzw. abzufragen.
- Der ORB muß wissen, wann er spezielle Kontextinformationen senden muß und was er mit eintreffenden Kontextinformationen tun muß.

Ist dies gegeben, so steht der impliziten Kontextübermittlung nichts mehr im Wege.

Im CORBA-Modul „IOP“ ist ein Record namens ServiceContext definiert. Diesem Record sind immer eine ID und Daten zugeordnet. Die ID ist von der OMG definiert und bestimmt den Dienst, zu dem diese Daten gehören. Die Daten enthalten servicespezifische Informationen.

Die servicespezifischen Kontextinformationen werden nun vom sendenden ORB von den CORBA-Services abgefragt und gesammelt und danach im Rahmen der Sequence „ServiceContextList“ des Moduls „IOP“ verschickt.

Der ORB auf der Seite des Aufgerufenen muß nun die „ServiceContextList“ analysieren und die dort enthaltenen Daten den entsprechenden Diensten, die bei ihm registriert sind, zugänglich machen.

7.3. Das Zusammenspiel mehrerer TS[26]

Wie schon erwähnt, können verschiedenen TS bei impliziter Kontextpropagierung zusammenarbeiten und so das Laufzeitverhalten verbessern. Wie dies geschehen kann, wird im folgenden erläutert:

---

[25] siehe [OMG98], Kapitel 11.6.7 „Object Service Context“, S. 314 (11-20) ff.
siehe [OMG98], Kapitel 13.3.3 „Encapsulation“, S. 356 (13-12) ff.

[26] siehe [OTS97], Kapitel 10.5.1, S. 10-49 ff.
siehe [OTS97], Kapitel 10.5.2, S. 10-60 ff.

Prinzipiell besteht zwischen ORBs und Transaction-Services ein n:m-Verhältnis. Hier soll nun nur der einfache Fall betrachtet werden, nämlich, daß man zwei vernetzte Rechner hat und auf jedem Rechner ein ORB und ein TS läuft. Vorteil der Verwendung mehrerer TS für mehrere Rechner ist die Tatsache, daß man damit die Netzlast senkt und somit eine höhere Performance erreicht.

Der erste in eine Transaktion eingebundene TS wird Root-TS genannt. Er ist für die Steuerung der Transaktion maßgebend und kann als einziger die Transaktion beenden, d.h. er hat als einziger TS, der in diese Transaktion eingebunden ist, ein „Terminator“-Objekt für diese Transaktion.

Wenn nun auf dem zweiten Rechner mit einem Methodenaufruf vom ersten Rechner aus, ein impliziter Transaktionskontext „eintrifft“ und dieser Transaktionskontext an den dortigen TS übergeben wird, so erzeugt dieser TS (sofern er dies unterstützt) für diesen Transaktionskontext und die entsprechende Transaktion ein neues „Control“-Objekt. Diesem „Control“-Objekt weist er nun das originale „Terminator“-Objekt zu, erzeugt jedoch eine lokale Version des „Coordinator“-Objekts. Somit können alle „Coordinator“-Aufrufe die auf diesem zweiten Rechner mit dieser Transaktion assoziiert sind, lokal abgearbeitet werden und müssen nicht an den ORB, bei dem die Transaktion begann, übermittelt werden. Die lokalen „Coordinators“ wiederum registrieren sich bei dem nächsthöheren „Coordinator“ sowohl als Ressourcen als auch über das „Synchronization“-Interface. Dadurch erhält der sogenannte „Subordinate Coordinator“ alle für sein Umfeld erforderlichen Aufrufe bzgl. des 2PC, um diese an bei ihm registrierte Objekte weiterzuleiten.

8. Beispiel für JTS

8.1. Aufbau des Beispiels

Im folgenden wird ein Beispiel zur Programmierung einer Anwendung unter Verwendung von Java und JTS dargelegt.

Das Anwendungsprogramm, das einen Transactional Client darstellt, ruft die Methode **BucheK1nachK2** eines Transactional Objects namens „Umbucher“ auf. In diesem Beispiel sollte man sich „Umbucher“ wie einen „Buchungsserver“ vorstellen, der zwar im Rahmen einer Transaktion Buchungen zwischen Konten durchführt, jedoch selbst keine Ressource besitzt, die im Rahmen eines 2PC eine Stimme für oder gegen eine Commit abgeben muß. Umbucher läuft in dem Beispiel als ein Serverprozeß, auf den der Transactional Client über den NamingService des ORBs zugreift.

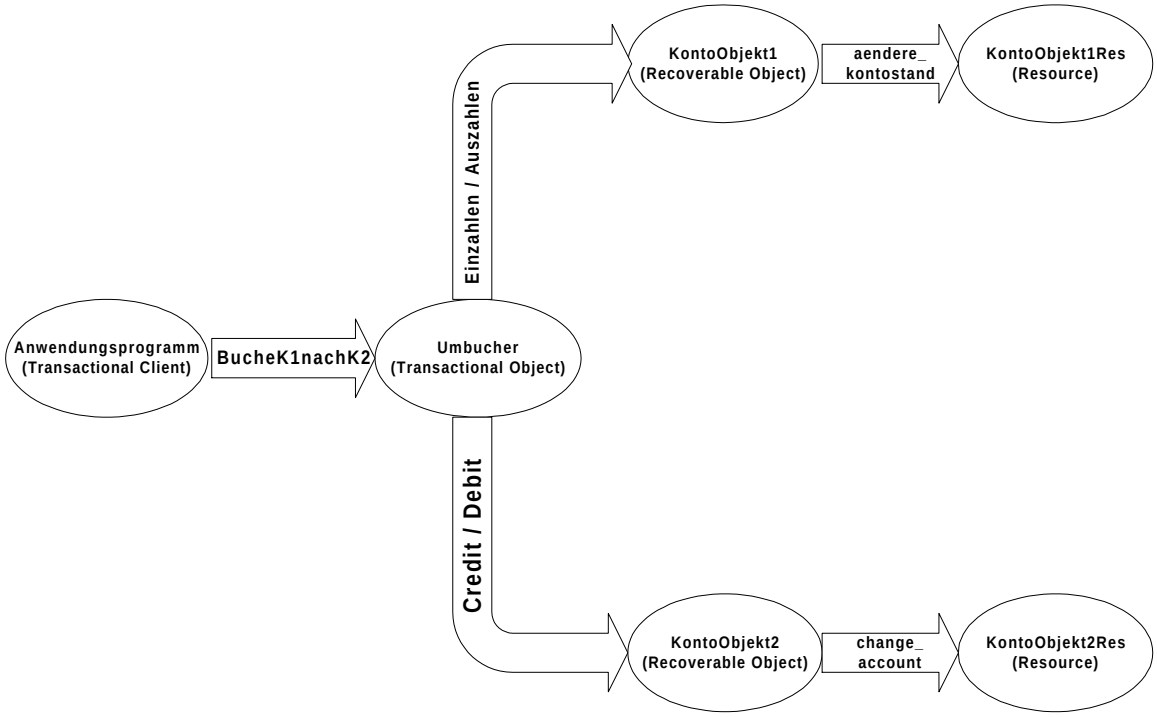

Abbildung 3 Aufbau des JTS-Beispiels

Der „Umbucher“ greift seinerseits auf zwei Serverprozesse zu, die jeweils ein Konto repräsentieren. Der Serverprozeß „KontoObjekt1“ kann Buchungen mittels der Methoden **Einzahlen** und **Auszahlen** durchführen. Wenn nötig, d.h. wenn sich der Kontostand durch eine Transaktion geändert hat, registriert KontoObjekt1 eine Ressource „KontoObjekt1Res“ beim „Coordinator“-Objekt, so daß diese Ressource in das 2PC-Protokoll eingebunden ist. „KontoObjekt1“ greift auf „KontoObjekt1Res“ mittels der Methode **aendere_kontostand** zu.

Der zweite Kontoserver „KontoObjekt2“ arbeitet genauso wie „KontoObjekt1“, nur daß seine Buchungsmethoden **credit** und **debit** heißen und „KontoObjekt2“ auf „KontoObjekt2Res“ mittels der Methode **change_account** zugreift.

Das Beispiel verwendet implizite Kontextpropagierung und, soweit möglich, indirektes Kontextmanagement. Indirektes Kontextmanagement ist aber z.B. nicht möglich, wenn man eine Ressource bei einem „Coordinator“ registrieren muß.

8.2. IDL-Definition

Im folgenden die leicht verständliche IDL-Definition des oben beschriebenen Beispiels.

```
// JTS-Sample v 0.5
//  Copyright (c) 1998 Michael Hurler
//  All Rights Reserved
//
// Compile via: idltojava -ID:\Java\JTS\CORBA -ftie -fverbose
                                                        JTSSample.idl
//
// IDL-Definitionsdatei

#include "d:\java\jts\CORBA\CosTransactions.idl"

#pragma prefix ""
#pragma javaPackage ""

module JTSSample
{
   interface KontoObjekt1: CosTransactions::TransactionalObject {

       long einzahlen(in long nb);
       long auszahlen(in long nb);
   };

   interface KontoObjekt1Res: CosTransactions::TransactionalObject,
                                          CosTransactions::Resource {

       long aendere_Kontostand(in long nb);
   };

   interface KontoObjekt2: CosTransactions::TransactionalObject {

       long credit(in long nb);
       long debit(in long nb);
   };

   interface KontoObjekt2Res: CosTransactions::TransactionalObject,
                                          CosTransactions::Resource {

       long change_account(in long nb);
   };
```

```
   interface Umbucher: CosTransactions::TransactionalObject {

       long BucheK1nachK2(in long nb);
   };
}; // Module JTSSample
```

8.3. Der Transactional Client

Der Transactional Client ruft den Umbucher, ein Transactional Object, auf.

```
// JTS-Sample v 0.5
//  Copyright (c) 1998 Michael Hurler
//  All Rights Reserved
//
// Transactional Client

import JTSSample.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;



public class TSClient
{
    public static void main(String[] args)
    {
       new TSClient(String[] args);
    }

    public void TSClient(String[] args)
    {
         private ORB myorb;
         private Current mycurrent;
         private org.omg.CORBA.Object objRef;
         private NamingContext ncRef;
         private NameComponent nc;
         private NameComponent path[];
         private Umbucher UmbucherRef;
```

Um eventuell auftretende Exceptions abzufangen, wird der ganze Anweisungsteil von einem try-catch-Statement eingerahmt.

```
         try
         {
            // Init ORB
            myorb = ORB.init(args, null);
```

Als erstes wird der ORB initialisiert.

```
            // "Find" NamingService
            objRef = orb.resolve_initial_references("NameService");
            ncRef = NamingContextHelper.narrow(objRef);
```

Danach wird der CORBA „NameService“ ausfindig gemacht. Er dient dazu, Server anhand des Names ausfindig zu machen.

```
// Find and Init TS
CosTransaction.init(myorb);
```

Nun muß der TS initialisiert werden. Hier muß dann auch die Identifikation der „Sender“- und „Receiver“-Interfaces zwischen ORB und TS stattfinden, von der der Entwickler allerdings nichts bemerkt.

```
// Find "Umbucher"
nc = new NameComponent("Umbucher", "");
path[] = {nc};
UmbucherRef = UmbucherHelper.narrow(ncRef.resolve(path));
```

Der Umbucherserver wird anhand seines Namens ausfindig gemacht und eine CORBA-Referenz auf ihn erzeugt.

```
CosTransactions.get_current().begin();
```

Mittels indirektem Kontextmanagement wird eine Transaktion gestartet.

```
UmbucherRef.BucheK1nachK2(2000);
```

Der Umbucherserver wird aufgerufen.

```
CosTransactions.get_current().commit();
```

Es wird ein Commit der Transaktion angefordert. Hier startet dann das 2PC-Protokoll.

```
        }
        catch(Exception e)
        {
            System.out.println("ERROR : " + e);
            e.printStackTrace(System.out);
        }
    }
}
```

8.4. Das Transactional Object (Umbucher)

Dieser, wie auch die nachfolgenden Serverprozesse, ist so aufgeteilt, daß er zum einen eine Java-Klasse enthält, die dazu dient, den eigentlichen CORBA-Server zu erzeugen und über einen ORB zur Verfügung zu stellen als auch zum anderen die eigentliche CORBA-Serverklasse.

```
// JTS-Sample v 0.5
//  Copyright (c) 1998 Michael Hurler
//  All Rights Reserved
```

```
//
// Buchungsserver

import JTSSample.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;
import org.omg.CosTransactions.*;

class UmbucherServant extends _UmbucherImplBase
```

_UmbucherImplBase ist das Skeleton für den Umbucher-Server.

```
{
    private ORB orb;
    private org.omg.CORBA.Object objRef;
    private NamingContext ncRef;
    private NameComponent nc;
    private NameComponent path[];
    private KontoObjekt1 ko1Ref;
    private KontoObjekt2 ko2Ref;

    public int UmbucherServant(ORB orb1)
```

Wenn der UmbucherServant erzeugt wird, ermittelt dieser die Referenzen auf die beiden Kontenobjekte, auf die er zugreifen muß. Dies geschieht genauso wie beim Transactional Client.

```
    { super();
      orb=orb1;

      // "Find" NamingService
      objRef = orb.resolve_initial_references("NameService");
      ncRef = NamingContextHelper.narrow(objRef);

      // Find "KontoObjekt1"
      nc = new NameComponent("KontoObjekt1", "");
      path[] = {nc};

      ko1Ref = KontoObjekt1Helper.narrow(ncRef.resolve(path));

      // Find "KontoObjekt2"
      nc = new NameComponent("KontoObjekt2", "");
      path[] = {nc};

      ko2Ref = KontoObjekt2Helper.narrow(ncRef.resolve(path));
    }

    public int BucheK1nachK2(int nb)
```

BucheK1nachK2 bucht einen Betrag von Konto 1 ab und schreibt ihn auf Konto 2 gut. Es fällt auf, daß hier keine Anweisung für die Steuerung der Transaktion steht, und trotzdem wird der Transaktionskontext an die beiden Konto-Objekte weitergeleitet.

```
    {
        ko1Ref.auszahlen(nb);
        ko2Ref.credit(nb);
```

```
    }
}

public class UmbucherServer
```

Der UmbucherServer ist nur ein Prozeß, der den ORB und den TS initialisiert, dann den eigentlichen Server lädt und danach wartet, bis er vom Anwender gestoppt wird.

```
{
    public static void main(String args[])
    {
       try
       {
           // create and initialize the ORB
           ORB orb = ORB.init(args, null);

           // create servant and register it with the ORB
           UmbucherServant UmbucherRef = new UmbucherServant();
           orb.connect(UmbucherRef);

           // get the root naming context
           org.omg.CORBA.Object objRef =
                   orb.resolve_initial_references("NameService");
           NamingContext ncRef = NamingContextHelper.narrow(objRef);

           // bind the Object Reference in NamingService
           NameComponent nc = new NameComponent("Umbucher", "");
           NameComponent path[] = {nc};
           ncRef.rebind(path, KontoObjekt1Ref);
```

Der obige Teil ist schon bekannt. Nur daß dieses Mal der Umbucher beim NameService bekanntgemacht wird, damit ein Client darauf zugreifen kann.

```
           // wait for invocations from clients
           java.lang.Object sync = new java.lang.Object();
           synchronized (sync)
           {
                sync.wait();
           }
```

Der Prozeß wartet nun bis er abgebrochen wird.

```
       }
       catch (Exception e)
       {
          System.err.println("ERROR: " + e);
          e.printStackTrace(System.out);
       }
    }
}
```

8.5. Das erste Recoverable Object inkl. Ressource (KontoObjekt1 & KontoObjekt1Res)

```
// JTS-Sample v 0.5
//  Copyright (c) 1998 Michael Hurler
//  All Rights Reserved
//
// Kontoserver 1

import JTSSample.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;
import org.omg.CosTransactions.*;

class KontoObjekt1Res extends _KontoObjekt1ResImplBase
```

Dies ist die Ressource, die beim Commit der Transaktion am 2PC-Protokoll teilnimmt. In der IDL sah man, daß die Ressource vom JTS-Interface Resource erbt.

```
{
    private int Bal;

    public void KontoObjekt1Res(String cmd)
    {
       super();
       < Load balance in "Bal" variable >
```

Im Konstruktor wird – von irgendwo her – der aktuelle Kontostand geladen.

```
    }
```

Es folgen die Methoden, die von einer Ressource implementiert werden müssen. Diese Methoden dienen der Teilnahme am 2PC-Protokoll.

```
    public Vote prepare()
```

prepare wird aufgerufen, wenn der TS bei den registrierten Ressourcen anfragt, wie sie bezüglich eines Commit abstimmen.

```
        throws org.omg.CosTransactions.HeuristicMixed,
                                  org.omg.CosTransactions.HeuristicHazard
    {
        if  ( <no problems, so far> )
        {    < If necessary, make changes temporarily persistent –
                                          e.g using persistence service >

             return Vote.VOTE_COMMIT;
```

Wenn keine Probleme aufgetreten sind, stimme für Commit, ansonsten für Rollback. Wenn für Commit gestimmt wird, so muß die Ressource dafür sorgen, daß sie ihre transienten Daten für die Zeit bis zum Commit persistent macht, so daß ein Commit auch im Falle eines zwischenzeitlichen Systemversagens zu einem späteren Zeitpunkt durchgeführt werden kann. Für die persistente Speicherung kann zum Beispiel ein Resourcemanager eingesetzt werden.

```
        }
        else
        {    return Vote.VOTE_ROLLBACK;
        }
    }

    public void rollback()
```

rollback wird vom TS aufgerufen, um die Ressource aufzufordern, alle eventuell durchgeführten Änderungen ungeschehen zu machen. Wenn die Ressource auf eine Datenbank zugreift, könnte sie nun also einfach ein Rollback auf die Datenbanktransaktion absetzen.

```
        throws org.omg.CosTransactions.HeuristicCommit,
                                   org.omg.CosTransactions.HeuristicMixed,
                                   org.omg.CosTransactions.HeuristicHazard
    {
        < throw away any changes - or use some Service to do so >
    }

    public void commit()
```

commit wird vom TS aufgerufen, um die Ressource aufzufodern, die gemachten Änderungen permanent zu machen. Wenn die Ressource auf eine Datenbank zugreift, könnte sie nun also einfach ein Commit auf die Datenbanktransaktion absetzen.

```
        throws org.omg.CosTransactions.NotPrepared,
                                org.omg.CosTransactions.HeuristicRollback,
                                   org.omg.CosTransactions.HeuristicMixed,
                                   org.omg.CosTransactions.HeuristicHazard
    {
        < Make changes permanent >
    }
    public void commit_one_phase()
```

commit_one_phase wird in einem speziellen Commit-Fall aufgerufen und soll hier nicht weiter erläutert werden.

```
        throws org.omg.CosTransactions.HeuristicHazard
    {
        < Make Changes Permanent >
    }
    public void forget()
```

forget hängt mit heuristischen Entscheidungen bzgl. Commit und Rollback zusammen. Da auf diesen Typus von Entscheidungen hier nicht eingegangen wurde, soll auch diese Funktion hier nicht weiter erläutert werden.

```
    {
        < Forget about heuristic descicions >
    }

    public int aendere_Kontostand(int nb)
```

Diese Methode ist für die Änderung des Zustandes der Ressource zuständig.

```
    {
        Bal = Bal + nb;
    }
}

class KontoObjekt1Servant extends _KontoObjekt1ImplBase
```

Der Servant ist der Prozeß, auf den der Umbucher zugreift, um den Kontostand zu verändern.

```
{
    private KontoObjekt1Res Balance;
    private ORB orb1;

    public void KontoObjekt1Servant(ORB orb)
```

Im Konstruktor wird das Ressource-Objekt erzeugt.

```
    {   super();
        // Load Account
        Balance = new KontoObjekt1Res("load");  // Create resource with
                                                   current balance
        orb1 = orb;
    }

    public int einzahlen(int nb)
    {
        // Link into 2PC-Protocol
        if ( <not registered as resource yet> )
```

Wenn die zu diesem Server gehörende Ressource noch nicht als solche beim Coordinator registriert worden ist, dann wird dies jetzt nachgeholt. Es ist möglich die Registrierung der Ressource bis zu diesem Zeitpunkt zu verschieben, da vorher noch keine Änderungen an Daten vorgenommen werden konnten, die dies notwendig gemacht hätten.

```
        {
             orb1.connect(Balace);
             CosTransactions.get_current().get_control().
                             get_coordinator().register_resource(Balance);
        }
        Balance.aendere_Kontostand(nb);
    }
```

```
    public int auszahlen(int nb)
    {
        // Link into 2PC-Protocol
        if ( <not registered as resource yet> )
        {
            orb1.connect(Balance);
            CosTransactions.get_current().get_control().
                                get_coordinator().register_resource(Balance);
        }
        Balance.aendere_Kontostand(-1*nb);
    }
}

public class KontoObjekt1Server
```

Diese Klasse ist, wie auch schon die entsprechende Klasse beim Umbucher-Server, für das Starten des eigentlichen Servers zuständig.

```
{
    public static void main(String args[])
    {
       try
       {
           // create and initialize the ORB
           ORB orb = ORB.init(args, null);

           // create servant and register it with the ORB
           KontoObjekt1Servant KontoObjekt1Ref =
                                          new KontoObjekt1Servant(orb);
           orb.connect(KontoObjekt1Ref);

           // get the root naming context
           org.omg.CORBA.Object objRef =
                   orb.resolve_initial_references("NameService");
           NamingContext ncRef = NamingContextHelper.narrow(objRef);

           // bind the Object Reference in NamingService
           NameComponent nc = new NameComponent("KontoObjekt1", "");
           NameComponent path[] = {nc};
           ncRef.rebind(path, KontoObjekt1Ref);

           // wait for invocations from clients
           java.lang.Object sync = new java.lang.Object();
           synchronized (sync)
           {
                sync.wait();
           }

       }
       catch (Exception e)
       {
          System.err.println("ERROR: " + e);
          e.printStackTrace(System.out);
       }
    }
}
```

8.6. Das zweite Recoverable Object inkl. Ressource (KontoObjekt2 & KontoObjekt2Res)

Der zweite Kontoserver ist bis auf einige Methodennamen vollkommen identisch mit dem ersten. Daher wird auf eine weitere Erläuterung verzichtet.

```
// JTS-Sample v 0.5
//  Copyright (c) 1998 Michael Hurler
//  All Rights Reserved
//
// Kontoserver 2

import JTSSample.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;
import org.omg.CosTransactions.*;

class KontoObjekt2Res extends _KontoObjekt2ResImplBase
{
    private int Bal;

    public void KontoObjekt2Res(String cmd)
    {
       super();
       < Load balance in "Bal" variable >
    }

    public Vote prepare()
        throws org.omg.CosTransactions.HeuristicMixed,
                                org.omg.CosTransactions.HeuristicHazard
    {
        if  ( <no problems, so far> )
        {    < If necessary, make changes temporarily persistent –
                                        e.g using persistence service >

             return Vote.VOTE_COMMIT;
        }
        else
        {    return Vote.VOTE_ROLLBACK;
        }
    }

    public void rollback()
        throws org.omg.CosTransactions.HeuristicCommit,
                                org.omg.CosTransactions.HeuristicMixed,
                                org.omg.CosTransactions.HeuristicHazard
    {
        < throw away any changes - or use some Service to do so >
    }

    public void commit()
        throws org.omg.CosTransactions.NotPrepared,
                            org.omg.CosTransactions.HeuristicRollback,
                                org.omg.CosTransactions.HeuristicMixed,
                                org.omg.CosTransactions.HeuristicHazard
    {
        < Make changes permanent >
```

```
    }

    public void commit_one_phase()
        throws org.omg.CosTransactions.HeuristicHazard
    {
        < Make Changes Permanent >
    }
    public void forget()
    {
        < Forget about heuristic descicions >
    }

    public int change_account(int nb)
    {
        Bal = Bal + nb;
    }
}

class KontoObjekt2Servant extends _KontoObjekt2ImplBase
{
    private KontoObjekt2Res Balance;
    private ORB orb1;

    public void KontoObjekt1Servant(ORB orb)
    {   super();
        // Load Account
        Balance = new KontoObjekt2Res("load");  // Create resource with
                                                                current balance
        orb1 = orb;
    }

    public int einzahlen(int nb)
    {
        // Link into 2PC-Protocol
        if ( <not registered as resource yet> )
        {
             orb1.connect(Balance);
             CosTransactions.get_current().get_control().
                             get_coordinator().register_resource(Balance);
        }
        Balance.change_account(nb);
    }
    public int auszahlen(int nb)
    {
        // Link into 2PC-Protocol
        if ( <not registered as resource yet> )
        {
            orb1.connect(Balance);
            CosTransactions.get_current().get_control().
                             get_coordinator().register_resource(Balance);
        }
        Balance.change_account(-1*nb);
    }
}

public class KontoObjekt2Server
{
    public static void main(String args[])
    {
       try
       {
```

```
            // create and initialize the ORB
            ORB orb = ORB.init(args, null);

            // create servant and register it with the ORB
            KontoObjekt2Servant KontoObjekt2Ref =
                                        new KontoObjekt2Servant(orb);
            orb.connect(KontoObjekt2Ref);

            // get the root naming context
            org.omg.CORBA.Object objRef =
                    orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);

            // bind the Object Reference in Naming
            NameComponent nc = new NameComponent("KontoObjekt2", "");
            NameComponent path[] = {nc};
            ncRef.rebind(path, KontoObjekt2Ref);

            // wait for invocations from clients
            java.lang.Object sync = new java.lang.Object();
            synchronized (sync)
            {
                 sync.wait();
            }

        }
        catch (Exception e)
        {
           System.err.println("ERROR: " + e);
           e.printStackTrace(System.out);
        }
    }
}
```

9. Resümee

In diesem Dokument wurde der Java Transaction Service als Implementierung des CORBA Object Transaction Service in Java vorgestellt. Der JTS implementiert ausschließlich ein Two-Phase-Commit-Protokoll, das verwendet wird um ein Transaktionskonzept zu verwirklichen. Persistenz oder Recovery kann JTS nicht bieten, dafür müssen entsprechende (CORBA)-Dienste wie ein Recoverable-Server oder ein Ressourcemanager verwendet werden.

Da bisher noch keine Implementierung des JTS existiert und es auch noch nicht sicher ist, ob es von Sun eine Referenzimplementierung geben wird, können Aspekte wie z.B. die Performance oder die Stabilität von JTS–Implementierungen nicht beurteilt werden. Bezüglich der Performance wäre insbesondere zu überprüfen, ob die Verwendung einer JTS-Implementierung in Java tatsächlich performanter ist, als die Verwendung einer OTS-Implementierung, selbst wenn dann zusätzlicher Kommunikationsaufwand entstände. Außerdem sind auch noch einige Bereiche in der aktuellen Spezifikation unterspezifiziert. So ist noch nicht klar, wie beispielsweise der JTS-Dienst initalisiert werden muß oder wie ein Current-Objekt erzeugt werden kann. Dies ist jedoch aufgrund der Forderungen von Sun an JTS dringend notwendig, denn nur so sind absolut portable JTS-Anwendungen realisierbar.

Leider war es im Rahmen dieser Arbeit nicht möglich aktuelle Entwicklungen bei OTS oder JTS zu berücksichtigen, da weder die OMG noch Sun Einblick in die laufenden Entwicklungsarbeiten gewähren.