Entwicklung einer verteilten Datenhaltungsschicht für virtuelle Welten

Am Beispiel des MRT-VR

Diplomarbeit

vorgelegt von Ralf Leonhard

Rheinische Friedrich-Wilhelms-Universität Bonn
Institut für Informatik III

26. Januar 2000

Versicherung

Hiermit versichere ich, die vorliegende Arbeit "Entwicklung einer verteilten Datenhaltungsschicht für virtuelle Welten" selbständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Niederkassel, den 26. Januar 2000

Ralf Leonhard

Danke !

An dieser Stelle möchte ich meinen Dank all jenen aussprechen, die mir während meiner Diplomarbeit zur Seite gestanden haben, sei es mit Ratschlägen, Einwänden, Diskussionen oder nur durch Zuhören.

Mein besonderer Dank gilt Armin Hopp für seine intensive fachliche Betreuung. Ohne seine Ideen und Vorschläge wäre diese Arbeit nicht zustandegekommen.

Nicht zuletzt danke ich auch Alexander Gross und meinem Vater für das aufmerksame Korrekturlesen und die vielen interessanten und fruchtbaren Anmerkungen.

Besonderer Dank gilt auch meinen Eltern für Ihre Geduld und Ihren Glauben an mich und meine Arbeit.

Inhaltsverzeichnis

1 EINLEITUNG ... 1

1.1 MOTIVATION ... 1
1.2 ZIELSETZUNG ... 1
1.3 AUFBAU DER ARBEIT ... 2

2 ANALYSE EINER MULTI-USER 3D UMGEBUNG ... 3

2.1 TEILNEHMERVERWALTUNG ... 3
2.2 SZENENVERWALTUNG ... 5
2.3 SICHERER MEHRBENUTZERBETRIEB ... 6
2.4 SICHERER NETZBETRIEB ... 7
2.5 UNTERSTÜTZUNG VERSCHIEDENER KONFERENZSZENARIEN ... 9
2.6 TEXTBASIERTER INFORMATIONSAUSTAUSCH (CHAT) ... 9
2.7 SPEZIELLE ANFORDERUNGEN AN DAS MULTI-USER MRT ... 9
2.8 ZUSAMMENFASSUNG DER ANALYSE ... 10

3 KONZEPT DATENBANK ... 11

3.1 ÜBERBLICK ... 11
3.2 TRANSAKTIONEN ... 13
3.3 SYNCHRONISATION ... 14
3.3.1 ZENTRALES SPERRPROTOKOLL ... 15
3.3.2 MEHRVERSIONEN-SYNCHRONISATIONSVERFAHREN ... 15
3.3.3 ZUSAMMENFASSUNG ... 15
3.4 DEADLOCK VERHÜTUNG ... 16
3.5 REPLIKATION ... 17
3.5.1 WRITE-ALL-ANSATZ ... 18
3.5.2 WRITE-ALL-AVAILABLE-VARIANTE ... 18
3.5.3 PRIMARY-COPY-VERFAHREN ... 18
3.6 TRANSPARENZEIGENSCHAFTEN ... 19
3.7 ZUSAMMENFASSUNG ... 20

4 COMMON OBJECT REQUEST BROKER ARCHITECTURE ... 21

4.1 OBJECT MANAGEMENT ARCHITECTURE ... 21
4.2 DIE STRUKTUR EINES OBJECT REQUEST BROKER ... 22
4.3 INTERFACE DEFINITION LANGUAGE ... 24
4.3.1 EINGEBAUTE TYPEN ... 25
4.3.2 KONSTRUIERTE TYPEN ... 25
4.3.3 TEMPLATE TYPEN ... 25
4.4 BEISPIELPROGRAMM ... 26
4.5 DYNAMIC INVOCATION INTERFACE ... 27
4.6 DYNAMIC SKELETON INTERFACE ... 28
4.7 BEWERTUNG ... 28

5 EIGENSCHAFTEN VON ORBACUS ... 29

5.1 UNTERSTÜTZUNGSUMFANG DER CORBA-SPEZIFIKATION ... 29
5.2 UNTERSTÜTZTE PLATTFORMEN ... 30
5.3 ERWEITERUNGEN GEGENÜBER DEM CORBA - STANDARD ... 30

6 ENTWURF DER DATENHALTUNGSSCHICHT ... 31

6.1 SZENENHIERARCHIE ÜBERTRAGEN ... 31
6.2 VERSIONSMANAGEMENT ... 33
6.3 TRANSAKTIONEN ... 34
6.4 NETZWERK ... 34
6.5 SCHICHTENMODELL ... 35

7 IMPLEMENTIERUNG ... 37

7.1 MRT-VR ... 37
7.2 INTERFACEBESCHREIBUNG ... 37
7.2.1 ADMINISTRATIONSINTERFACE (T_MUI) ... 37
7.2.2 KONFERENZINTERFACE (T_CONFINTERFACE) ... 38
7.3 DATENKATALOG ... 46
7.3.1 KONFERENZTEILNEHMERDATEN (T_USERDESCRIPTION UND T_FULLUSERDESCRIPTION) ... 46
7.3.2 SZENENOBJEKTDATENSATZ (T_REFPTR_ID) ... 47
7.3.3 PARAMETERLISTE (T_PARAMLIST) ... 48
7.3.4 PROTOKOLL (T_PROTOKOLL) ... 49
7.4 BESCHREIBUNG DER OBJEKTIMPLEMENTIERUNGEN ... 50
7.4.1 INTERFACE OBJEKTE T_MUI UND T_CONFINTERFACE ... 50
7.4.2 KONFERENZEN - ADMINISTRATION (T_CONFADMIN) ... 50
7.4.3 KONFERENZ (T_CONFERENCE) ... 50
7.4.4 TEILNEHMERLISTE (T_USERLIST) ... 51
7.4.5 OBJEKTLISTE (T_OBJECTLIST) ... 52
7.4.6 PROTOKOLLIERENDER NACHRICHTENVERTEILER (T_BROADCASTER) ... 54
7.4.7 KAMERAVERTEILER (T_CAMERATHREAD) ... 55
7.4.8 NACHRICHTENDISPATCHER (T_MSGHANDLER) ... 55

8 LAUFZEITVERHALTEN DES MUI-MRT ... 56

8.1 ABHÄNGIGKEIT VON DER SZENENGRÖßE ... 56
8.2 ABHÄNGIGKEIT VON DER TEILNEHMERANZAHL ... 57
8.3 PRAXISTEST ... 59
8.4 ZUSAMMENFASSUNG DER MEßERGEBNISSE ... 60

9 ERGEBNISSE UND AUSBLICK ... 61

ANHANG ... 64

ABBILDUNGSVERZEICHNIS ... 70

TABELLENVERZEICHNIS ... 70

LITERATURVERZEICHNIS ... 71

1 Einleitung

1.1 Motivation

Verteilte Anwendungen für Arbeiten, Lernen, Konferieren oder Diskutieren gewinnen bei zunehmender Vernetzung über das Internet immer mehr an Bedeutung. Es gibt viele Projekte und kommerzielle Anwendungen, die das Konferieren und gemeinsame Arbeiten im Internet unterstützen, z.B. Video- und Audiokonferenzsysteme wie vic und vat [MJ95], komfortable Chatumgebungen wie Netmeeting [MS1] oder ICQ [ICQ], u.s.w. .

Die auf diese Weise ausgetauschten Informationen werden zunehmend komplexer und deren Beschreibung mit Worten entsprechend schwerer. Hinzu kommt, daß die Beteiligten aus verschiedenen Fachgebieten kommen können und die Ausdrucksweisen unterschiedlich sind. Es wäre zweckmäßig den Sachverhalt an einem Modell interaktiv zu veranschaulichen. Jeder Betrachter hätte die Möglichkeit den Blickwinkel und die Art der Visualisierung selbst zu bestimmen. Nicht umsonst heißt es: "Ein Bild sagt mehr als 1000 Worte". Mit Mitteln der Computergrafik ist es heute möglich, dreidimensionale (virtuelle) Welten zu erschaffen und sich "in" ihnen zu bewegen. Darauf aufbauend gibt es eine unbegrenzte Zahl von Einsatzgebieten. Beispielsweise könnte der Kunde eines Architekten eine Wand einfach an der gewünschten Stelle einsetzen oder herausnehmen und die geänderten Lichtverhältnisse z.B. mittels Ray Tracing fotorealistisch betrachten. Der Architekt setzt die gleiche Szene als Wireframe für die geänderte Statikberechnung ein. Die Position eines Roboters in einer entfernten, unbekannten Testumgebung könnte in einem entsprechenden Modell aufgezeigt werden, das der Roboter mit seinen Sensoren selbst erstellt. In der Präsentation eines neuen Medikamentes könnte die Funktionsweise eines Proteins auf atomarer Ebene verdeutlicht und von jedem aus allen Perspektiven betrachtet werden.

Das gemeinsame Betrachten und Ändern virtueller Welten ist eine konsequente Weiterentwicklung und Ergänzung der zuvor erwähnten Anwendungen.

1.2 Zielsetzung

Basierend auf dem MRT, einer Visualisierungsanwendung für virtuelle Welten, ist die Entwicklung und Implementierung einer verteilten Datenhaltung, die die Daten zwischen im Internet verteilten Anwendungen austauschen und synchronisieren kann, das Ziel dieser Diplomarbeit.

Jeder User in einer Virtuellen-Multi-User-3D-Umgebung, wie dem MRT-VR [HA97a, HA97b], erhält eine Kopie der Szene, in der er sich bewegen, Objekte hinzufügen, löschen oder modifizieren kann. Alle diese Änderungen müssen zu den anderen Teilnehmern übertragen und mit deren Änderungen abgeglichen werden. Es können konkurrierende Zugriffe, Datenverluste, unterschiedliche Laufzeiten der Nachrichten sowie das Ein- und Aussteigen einzelner Anwender im Netzwerk auftreten. Es muß also entschieden werden, welche Daten wann und zu wem übertragen werden und welche Zugriffsrechte die einzelnen Teilnehmer haben.

1.3 Aufbau der Arbeit

Im ersten Kapitel *Einleitung* wird die in dieser Arbeit behandelte Problemstellung dargelegt und deren Inhalt sowie der Aufbau der folgenden Kapitel zusammengefaßt.

Kapitel 2 *Analyse einer Multi-User 3D Umgebung* analysiert und spezifiziert die Anforderungen einer verteilten Datenhaltung und skizziert erste Lösungsansätze.

In Kapitel 3 *Konzept Datenbank* werden Parallelen zwischen MUI-MRT und Datenbanken veranschaulicht und Ansätze zusammengestellt, die übertragen werden.

Die Datenhaltung soll auf einer verteilten Objektplattform realisiert werden. Kapitel 4 *Common Object Request Broker Architecture* führt kurz in CORBA ein und stellt die wesentlichen Eigenschaften vor.

In Kapitel 5 *Eigenschaften von ORBacus* werden die Eigenschaften der verwendeten CORBA-Implementierung von Object Oriented Concepts (OOC) ORBacus 3.1.1 beschrieben.

Kapitel 6 *Entwurf der Datenhaltungsschicht* beschreibt die anhand der zuvor entwikkelten Ansätze und übernommenen Konzepte entworfenen Lösungen.

Kapitel 7 *Implementierung* beschreibt die Implementierung der Datenhaltungsschicht mit CORBA und den von VDBMS'en übernommenen Konzepten.

In Kapitel 8 *Laufzeitverhalten des MUI-MRT* werden anhand von Meßergebnissen die erreichten Laufzeiten und Übertragungsraten dargestellt.

Die mit der Implementation erzielten Ergebnisse werden in Kapitel 9 *Ergebnisse und Ausblick* zusammengefaßt und einen Ausblick auf Erweiterungs- und Performancesteigerungsmöglichkeiten dargestellt.

Im Anhang werden die vollständigen OMG IDL Interfacedefinitionen und die C++-Headerdateien der Objekte im Source-Code aufgeführt.

2 Analyse einer Multi-User 3D Umgebung

Am Institut für Informatik in Bonn in der Abteilung Computergrafik ist basierend auf dem Minimal Rendering Tool (MRT) eine verteilte Multi-User Umgebung für virtuelle Welten (MRT-VR) entwickelt worden, die vor allem auf graphische Anforderungen in der Ausbildung und der Präsentation ausgerichtet ist. Dieses System nutzt existierende Kommunikationsmechanismen wie IRC, TCX oder Multicast für den Datenabgleich. IRC und TCX nutzen jeweils einen eigenen Server, über den die Daten der verschiedenen Protokolle verteilt werden. Bisher wurden nur rudimentäre Fähigkeiten für eine Multi-User Unterstützung implementiert. Den MRT-VR zu erweitern und die Datenhaltung und Verteilung in einer eigenen Schicht unabhängig vom MRT zu implementieren ist Aufgabe dieser Arbeit.

Die Hauptaufgabe eines Multi-User MRT-VR ist es, mehreren Benutzern über ein Netzwerk, z.B. das Internet, die gleichzeitige Betrachtung und Bearbeitung einer virtuellen Szene zu ermöglichen. Somit wird Computer unterstütztes gemeinsames Arbeiten bzw. Computer Supported Collaborative Work (CSCW) oder Teleworking an einem 3D Modell über das Internet oder Intranet ermöglicht.

Im folgenden wird eine Präsentation oder Vorlesung, ein Workshop oder eine beliebige andere Art der virtuellen Zusammenkunft über das Multi-User-Interface des MRT (MUI-MRT) als **Konferenz** bezeichnet.

Der MRT-VR soll um folgende Punkte erweitert oder ergänzt werden zum MUI-MRT:

- Teilnehmerverwaltung
- Szenenverwaltung
- Sicherer Mehrbenutzerbetrieb
- Sicherer Netzbetrieb
- Unterstützung verschiedener Präsentationsszenarien
- Transparenz
- Textbasierter Informationsaustausch (Chat)

Diese Punkte werden im folgenden näher betrachtet und daraus Ansätze für die Erweiterung entwickelt.

2.1 Teilnehmerverwaltung

Die Teilnehmerverwaltung beim MRT-VR besteht aus einer Liste von Teilnehmer-Nummern oder -Namen. Beim Anmelden (oder Abmelden) eines Teilnehmers wird dieser hinzugefügt (oder entfernt). Die Teilnehmerliste wird bei einer vom Client initiierten Aktualisierung neu übertragen.

Für den MUI-MRT soll die Teilnehmerliste zu einem vollständigen sich selbst aktualisierenden System ausgebaut werden. Konferenzteilnehmer werden mit den zusätzlichen Informationen eMail-Adresse, Rechnername (Host), Name und/oder Nickname, der

obligatorischen UserID (UID), der aktuellen Kameraposition und einem freien Kommentarstring, der z.B. die Adresse der Homepage enthalten kann, gespeichert. Jeder Benutzer kann seine Daten im laufenden Betrieb ändern. Ausgenommen sind die UID, die Zugriffsrechte und der Rechnername. Zusätzlich können über ein rudimentäres Chatsystem Textnachrichten an alle Teilnehmer versendet werden. (Kapitel 2.6)

Aus dem MRT-VR wird auch die Idee der Verteilung der Kameradaten der einzelnen Teilnehmer übernommen. Jeder soll die Kameradaten aller anderen Teilnehmer erhalten. Diese sollen auf verschiedene Weisen nutzbar sein.

Zum einen kann der Avatar eines Teilnehmers anhand seiner Kameraposition gesteuert werden, um seinen Standort in der virtuellen Welt zu repräsentieren. Den Avatar kann jeder Teilnehmer selbst wählen, indem er eine beliebige Szene als seinen Avatar einspielt. Zum anderen können Teilnehmer eine Kamera übernehmen, um die gleiche Sicht auf die Szene zu erhalten, wie der andere Teilnehmer oder der Vortragende.

Wie bei der Benutzung von Datenbanken soll es möglich sein, den Teilnehmern Zugriffsrechte zuzuweisen. So sollte es bei einer Präsentation nur dem Vortragenden und seinen Assistenten gestattet sein, Objekte einzufügen, zu löschen oder zu ändern. Die verschiedenen Konferenzszenarien werden in Kapitel 2.5 erläutert.

Es sollen nicht nur "menschliche" Teilnehmer an einer Konferenz partizipieren können. Über entsprechende einfach anzupassende Schnittstellen sollen auch Simulationen oder Berechnungen, Daten von Sensoren, Maschinen oder Robotern u.s.w. visualisiert werden können. Diese Systeme werden fortan als Automaten bezeichnet. Ein Automat braucht nicht die Informationen und Dienste, die ein Teilnehmer an einer Konferenz nutzt (z.B. den Chatdienst oder die Position der Avatare). Teilweise ist es nicht einmal nötig, daß der Automat irgendwelche Informationen aus der Szene benötigt. Eine weitere Art Teilnehmer könnten beispielsweise Monitorprogramme sein, die alle Vorgänge protokollieren und bei Bedarf wieder abspielen können (Logbuch). Und schließlich ist auch ein separater Konferenzserver ein Konferenzteilnehmer.

Ein Dienst besteht aus Nachrichten, die einer bestimmten Funktionsgruppe zugeordnet werden können.

Die Palette der möglichen Automaten, Monitore etc. ist unbegrenzt, weshalb die Flexibilität der Schnittstelle ein möglichst breites Spektrum bieten sollte. Jedem Teilnehmer ob Automat oder Mensch soll es möglich sein, verschiedene Dienste der Konferenz in Anspruch zu nehmen. Braucht der Automat keinen Chat, bestellt er diesen Dienst ab und erhält keine weiteren Textnachrichten vom Server, was zudem die Netzbelastung senkt. Diese Möglichkeit soll für alle Dienste, also Chat, Kameradaten, Avatare, Objektaktualisierungen und Benutzerdaten etc., bestehen.

Da die Daten bei den Teilnehmern lokal gehalten werden, sind Inkonsistenzen bei Änderungen die Folge. Die Benutzerdaten sollen bei allen Teilnehmern automatisch aktualisiert werden, ohne das die Anwendung oder der Teilnehmer sich darum kümmern muß. Die Teilnehmerverwaltung läuft also völlig transparent ab.

2.2 Szenenverwaltung

Die Szenendaten des MRT-VR werden durch die Datenhaltung zwischen den Teilnehmern abgeglichen. Diese entsprechenden Nachrichten werden unterstützt:

- Create ein neues Objekt erzeugen
- Modify ein bestehendes Objekt ändern
- Delete ein bestehendes Objekt löschen
- Request die gesamte Datenbasis übertragen

Die Nachrichten werden als Datenblock übertragen, dessen erste 64 Bit ein Tag spezifizieren. Ein Dispatcher liest dieses Tag aus und ordnet die anhängenden Daten je nach Nachrichtentyp den Parametern der Funktionen zu und ruft diese auf [HA97a]. Dieses Verfahren bringt einige Nachteile mit sich:

- Ändern der Implementierung für verschiedene Plattformen (siehe Kapitel 2.4 Heterogenität)
- Keine Typüberprüfung
- Dispatchen der eingehenden Nachrichten in Eigenregie

Alle Teilnehmer sollen die gleichen Daten erhalten. Die Szenenhierarchie soll identisch bei jedem Teilnehmer konstruiert werden können. Ein Einstieg in eine bereits laufende Konferenz führt zur Übersendung der zu diesem Zeitpunkt bestehenden Szene. Es werden nicht wie bisher die Ausgangsszene und alle seither durchgeführten Änderungen übertragen. Das heißt es ist ein Mechanismus gefragt, der eine vorhandene hierarchische Szenenstruktur analysieren und für den Versand verpacken kann.

Auch für Automaten ist die Kenntnis aller Szenendaten von Vorteil. Beispielsweise könnte ein Partikelsystem diese Daten für eine Kollisionserkennung nutzen ("Schnee kann auf Dächer von Häusern fallen, anstatt durch sie hindurch.").

Weiterhin soll es möglich sein, einzelne Objekte oder ganze Szenen

- der Hierarchie hinzuzufügen (z.B. die Avatare der Teilnehmer) oder sie zu löschen,
- an eine andere Stelle im Szenengraphen zu verschieben,
- zu ändern (Hierbei ist den Referenzobjekten (`t_RefObject`) besondere Beachtung zu schenken, da diese als einzige eine Manipulation durch das Anwenden beliebiger Transformationsmatrizen direkt erlauben.) und
- zu markieren, um die Aufmerksamkeit der Teilnehmer auf ein bestimmtes Objekt zu lenken.

Änderungen dieser Art müssen an alle Teilnehmer übertragen werden, um die Konsistenz der Szenen zu bewahren. Dabei ist zu unterscheiden zwischen wichtigen und unwichtigen Änderungen. Das Hinzufügen eines Objektes ist eine wichtige Information, die vom Ändernden einmal abgesandt wird und alle Teilnehmer erreichen muß. Die Position der Kamera eines Teilnehmers ist hingegen unwichtig, da diese kontinuierlich geändert und immer wieder aktualisiert wird. Somit ist das überspringen bzw. auslassen einzelner Nachrichten keine Verletzung der Konsistenz, zumal die Kameraposition nur von ihrem Besitzer geändert werden kann. Kapitel 2.3 geht näher auf die Definition der Konsistenz ein.

2.3 Sicherer Mehrbenutzerbetrieb

Ein bisher vernachlässigter Bereich im MRT-VR sind die auftretenden Probleme im Mehrbenutzerbetrieb.

Konsistenz: Jedem Teilnehmer ist es möglich, jedes Objekt zu ändern. Wenn es zu konkurrierenden Zugriffen auf ein Objekt kommt, setzt sich derjenige durch, der seine Änderungen als letzter übergibt. Nach mehreren konkurrierenden Änderungsoperationen können die Szenen voneinander abweichen. Das muß vermieden werden, um die Konsistenz der Daten zu erhalten. Konkurrierende Zugriffe sind zu ordnen, so daß ein Objekt immer nur von einem Teilnehmer zur gleichen Zeit editiert werden kann. Dieses Verhalten ist Grundlage jeder Datenbank, die z.B. mit dem Setzen von Lese- und Schreibsperren den Zugriff auf die Datensätze regelt. In Kapitel 3 werden die in Datenbanksystemen (DBS) verwendeten Konzepte näher untersucht.

Transparenz: Für Teilnehmer und Anwendung sollte der Ort eines Teilnehmers oder Objektes im Netzwerk transparent sein. Allein die Datenhaltung muß wissen, ob sich ein Objekt lokal im gleichen Prozeß oder auf einem Rechner am anderen Ende der Welt befindet. Dies erleichtert die Programmentwicklung und -bedienung und verringert die Fehlerquellen. Dazu gehört beispielsweise auch, daß Teilnehmer sich bei einem beliebigen Client anmelden können, um an der virtuellen Welt partizipieren zu können. Der Client leitet die Daten dann an den Server weiter, der sich um den Neuzugang kümmert und ihn mit den notwendigen Daten versorgt.

Garbage Collection: Dennoch sind verteilte Anwendungen anfälliger für Fehler als Applikationen, die auf einem einzelnen Rechner ablaufen. Allein die durch ein Weitverkehrsnetz entstehenden Probleme (Kapitel 2.4) und die unterschiedlichen Rechnerarchitekturen und Betriebssysteme in einer heterogenen Umgebung (wie dem Internet) sorgen für genügend Fehlerquellen. Somit werden unweigerlich einzelne Nachrichten verloren gehen oder Teilnehmer ausscheiden. Dadurch können lose Enden in der Szenenhierarchie entstehen, Avatare für ausgeschiedene Teilnehmer zurückbleiben etc. . Die Datenhaltung muß also regelmäßig unreferenzierte Objekte aufsammeln und beseitigen. Die sogenannte Garbage Collection sorgt dafür, daß Objekte, auf die mindestens ein aktiver Teilnehmer Referenzen hält beibehalten und unreferenzierte Objekte gelöscht werden. Die Konsistenz der Daten muß jedoch unabhängig davon gewährleistet bleiben.

Sicherheit: Schließlich gibt es noch die Sicherheitsproblematik. Die bereits in Kapitel 2.1 erwähnten Zugriffsrechte der Teilnehmer sorgen dafür, daß angemeldete Teilnehmer nur die ihnen erlaubten Operationen durchführen dürfen.

Die Sicherheit der Datenübertragung vor unberechtigten Zugriffen und Manipulationen Dritter soll hier nur der vollständigkeithalber erwähnt werden. In dieser Arbeit werden diese Aspekte nicht behandelt. Weitere Informationen zu diesem Thema sind in [GW90] aufgeführt.

2.4 Sicherer Netzbetrieb

Was dem bisherigen System (MRT-VR) völlig fehlt ist der Umgang mit Problemen, die aus dem Netzbetrieb bei verteilten Anwendungen resultieren. Eine wichtige Eigenschaft großer Computernetzwerke, wie dem Internet und Firmen-Intranets, ist, daß sie heterogen sind (z.B. UNIX Workstations und Server, PC Systeme mit Microsofts Windows, IBMs OS/2 oder Apple Macintosh, sogar Geräte wie Telefonanlagen und Roboter). Die zugrundeliegenden Netzwerke und Protokolle, die diese Systeme verbinden können ebenfalls sehr unterschiedlich sein (Ethernet, FDDI, ATM, TCP/IP, IPX, RPC Systeme etc.). Der Programmierer einer verteilten Anwendung muß neben den eigentlichen Aufgaben seiner Anwendung auch folgende Anforderungen beachten:

Ort der Objekte im Netz: Das Programm muß netzwerkprotokollspezifisch Adressen der Teilnehmer, Dienste und Objekte verwalten.

Netzwerkverbindungen: Netzwerkverbindungen zu einem Service müssen aufgebaut und nach Abschluß der Datenübertragung wieder geschlossen werden.

Marshaling: Wenn die Anwendungen auf verschiedenen Rechnerarchitekturen oder Betriebssystemen laufen, oder mit einem anderen Compiler übersetzt wurden, kann das bedeuten, daß gleiche Datentypen intern unterschiedlich repräsentiert sind. Der Programmierer muß also ein Format entwickeln, das auf allen Plattformen in die entsprechende Repräsentation konvertiert werden muß.

Type Checking: Da über die meisten Netzwerkprotokolle untypisierte Speicherblöcke übertragen werden und keine Basistypen wie long, double oder string und erst recht keine vom Benutzer definierten Datenstrukturen, muß sich der Entwickler um eine adäquate Typüberprüfung kümmern

Error Handling: In einer Netzwerkumgebung können vielfältige Fehler bei der Nachrichtenübertragung auftreten:

- Vorübergehende oder andauernde Verbindungsausfälle, auf die mit dem Nachsenden der Objektdaten nach dem Wiederherstellen der Verbindung oder dem Ausschluß des Teilnehmers nach einer maximalen Wartezeit (Timeout) reagiert wird. Es muß also ein Protokoll der nicht zugestellten Nachrichten geführt werden.
- Das Netz kann die Teilnehmer in verschieden große Gruppen aufteilen, die sogenannte Partitionierung.
- Einzelne Rechner oder der Server können ausfallen und die Übermittlung von Nachrichten verzögert sich.

Diese Fehler müssen dem Anwender soweit wie möglich verborgen bleiben. Das Programm sollte eine möglichst große Autonomie besitzen, um auch ohne ständige Verbindung mit dem Server aktiv zu bleiben.

Heterogenität der Architekturen führt zu Unterschieden in Parameterübergabetechniken, der Byteanordnung (little-endian oder big-endian) und der Konvertierung von Binärformaten, die diese Architekturen unterstützen. Betriebssysteme unterscheiden sich in der Unterstützung von Threads, den Systemfunktionen u.s.w. (z.B. POSIX und Win32).

Kommunikation: Nachrichten zwischen Objekten können auf verschiedene Weisen übertragen werden: synchron, semisynchron und asynchron. Im synchronen Modus ruft ein Objekt (oder eine Anwendung) einen Dienst auf und blockiert solange, bis es eine Antwort zurückerhält. Im asynchronen Modus wird die Anwendung unterbrochen, wenn die Antwort eingetroffen ist. So kann die Anwendung fortgeführt werden, und muß nicht blockierend auf die Antwort warten. Im semisynchronen Modus pollt die Anwendung in regelmäßigen Abständen, während die Programmausführung fortgeführt werden kann.

Multithreading in die Datenhaltung aufzunehmen, bedeutet, daß die Verfügbarkeit und Performance erhöht wird. Da der Anwender jederzeit die Darstellungsform ändern und z.B. ein Bild durch Ray Tracing oder Radiosity berechnen kann, muß die Datenhaltung im Hintergrund unabhängig vom Rechenaufwand der Visualisierung ankommende Nachrichten empfangen, weiterleiten und aufnehmen können. Umgekehrt darf die Interaktion und Reaktionszeit, die gerade bei graphischen Anwendungen besonders wichtig ist, nicht leiden, wenn beispielsweise ein neuer Teilnehmer die gesamte Szenenbeschreibung zugesandt bekommt. Das Multithreading kann jedoch Portabilität kosten und erhöht Entwicklungs- und Debugzeit der Anwendung, durch den zusätzlichen Synchronisationsaufwand.

Server Arbeitsweise: Server in verteilten Umgebungen können auf zwei Arten arbeiten: iterativ oder parallel. Iterative Server sammeln alle eingehenden Nachrichten in einer Warteschlange und arbeiten sie nach dem FIFO-Prinzip (first in first out [DI]) ab. Parallele Server benutzen Multithreading um eingehende Anfragen verschiedener Clients simultan abarbeiten zu können. Das iterative Design ist am besten für Dienste die nur kurz beansprucht werden, während der parallele Ansatz eingesetzt werden sollte, wenn der Server mit der Bearbeitung einer Anfrage längere Zeit beschäftigt ist. Grundsätzlich sollte ein Wechsel der Implementierung die darüberliegenden Anwendungsschichten nicht berühren.

Transparenz: Die verschiedensten Funktionen in einem verteilten System können für den Anwender transparent gehalten werden. Beispielsweise kann der Ort von Objekten durch das System behandelt werden, oder dem Anwender überlassen bleiben. Ein vollkommen transparentes System zu entwickeln würde einen gewaltigen Overhead und Ineffizienz in das System einbringen, während die völlige Freigabe aller Kontrollen an den Anwender die Schwierigkeiten an den Anwendungsentwickler weiterleiten würde. Kapitel 3.6 zeigt die wesentlichen Transparenzarten verteilter Anwendungen auf.

Netzwerkprotokolle: Die direkte Verwendung von Netzwerkprotokollen, wie UDP, MBone oder TCP/IP überläßt dem Programmierer die Implementation der hier aufgeführten Anforderungen. Die Verwendung verteilter Objektplattformen wie DCOM (Microsoft) oder CORBA (OMG), nimmt dem Entwickler einen Großteil dieser Aufgaben ab. Einen Überblick dieser Plattformen und eine detaillierte Beschreibung von CORBA gibt Kapitel 4.

2.5 Unterstützung verschiedener Konferenzszenarien

Je nach Art der Anwendung können an einer Konferenz von einigen wenigen Teilnehmern bis zu einem großen Publikum, sehr unterschiedliche Anwenderzahlen auftreten, die sich teilweise erst während der Konferenz anmelden können. Alle Anwender erhalten die aktuellen Szenendaten und sämtliche Änderungen der Objektdaten werden an sie weitergeleitet. Im Rahmen folgender Szenarien muß das MUI-MRT skalierbar sein:

- Direkte Verbindung zwischen zwei Kommunikationspartnern.
- CSCW mehrerer Personen (ca. 20) an einem Projekt, wobei jeder volle Zugriffsrechte auf alle Objekte erhält.
- Vorlesungen mit einem Vortragenden und eventuellen Comoderatoren mit vollem Zugriffsrecht und einer großen Anzahl an Zuhörern mit alleinigen Leserechten. Einen Avatar dürfen die Zuhörer nicht einsetzen, da diese die Szene sehr groß, die Visualisierung verlangsamen und die Übersicht stark reduzieren würden. In den meisten Fällen werden die Teilnehmer die Kamera des Vortragenden übernehmen.

In allen Szenarien sollen Automaten angemeldet werden können, die Animationen oder andere fortlaufende Änderungen der Szenendaten in das System einspeisen, um z.B. die Präsentation von Meßwerten oder die Position eines Roboters aus einer realen Testumgebung in die virtuelle Welt zu übertragen, u.s.w. .

2.6 Textbasierter Informationsaustausch (Chat)

In Kapitel 1.1 wurden bereits vorhandene Module für den Informationsaustausch erwähnt, die auch parallel zum MUI-MRT eingesetzt werden sollten. Dennoch ist eine rudimentäre Chatunterstützung sinnvoll, um spontan Informationen austauschen zu können.

In "großen" Konferenzen, die vorbereitet und geplant werden müssen, werden Tools wie vic, vat, etc. ergänzend angewendet. Bei "kleinen" spontan gestarteten Konferenzen mit wenigen Teilnehmer wird das integrierte Chatmodul für den Informationsaustausch genutzt, um Einsatz, Adressen und Ports anderer Tools abzustimmen.

2.7 Spezielle Anforderungen an das Multi-User MRT

Das MRT soll nur soweit geändert werden, wie es unbedingt erforderlich ist. Es ist modular aufgebaut und erlaubt die leichte Einbindung neuer Funktionen. Es wird in unterschiedlichsten Projekten und Umgebungen eingesetzt. Eine "Belastung" des MRT mit für andere Anwendungen unnötigen Erweiterungen ist zu vermeiden. Die Datenhaltungsschicht soll einfach hinzugenommen oder weggelassen werden können. Das MRT sollte unabhängig von der Implementation der Datenhaltungsschicht auf allen bisher unterstützten Plattformen lauffähig sein.

2.8 Zusammenfassung der Analyse

Aus der Analyse ergeben sich folgende Aufgaben und Ereignisse:

Konferenz- und Teilnehmerverwaltung:

- Starten und Beenden einer Konferenz
- Bei einer Konferenz an- und abmelden
- Teilnehmerliste anfordern
- Ändern der eigenen Benutzerinformationen
- Verwaltung von Zugriffsrechten
- Unterstützung automatisierter "Teilnehmer"
- Repräsentation von Teilnehmern durch Avatare
- Übermittlung der Kamera-/Avatarpositionen der Teilnehmer
- Konferenzdienste nach Bedarf auswählen
- Chatting
- Einfache Skalierung des Konferenzumfanges

Szenenverwaltung:

- Übertragen der gesamten Szene zu neuen Teilnehmern
- Einfügen, Löschen und Ändern von Objekten
- Markieren von Objekten
- Zusätzliches Einladen von Szenen durch die Teilnehmer (z.B. Avatar)
- Minimale Änderungen an MRT-Basisklassen

Mehrbenutzer- und Netzbetrieb:

Diese Punkte werfen viele Probleme auf, die allen verteilten Anwendungen gemeinsam sind. Daher werden in Kapitel 3 Parallelen zu einem verwandten Anwendungstyp, den verteilten Datenbankmanagementsystemen, aufgezeigt, um Lösungen und Konzepte übernehmen zu können. Die Anforderungen an die Netzwerkprogrammierung auf die vorhandenen (herkömmlichen) Art zu lösen, wäre nur unter großem Aufwand realisierbar. Deshalb soll eine moderne Netzwerktechnologie zur Erstellung verteilter Anwendungen genutzt werden. Von besonderem Interesse ist hier das Konzept der Object Management Group (OMG) mit CORBA, das in Kapitel 4 beschrieben wird.

3 Konzept Datenbank

In Kapitel 2 haben sich viele Parallelen zwischen MUI-MRT und Datenbankensystemen gezeigt. Diesen Parallelen folgt dieses Kapitel und beschäftigt sich näher mit der Theorie verteilter Datenbankmanagementsysteme (VDBMS).

Mit einem kurzen Überblick der Datenbankengrundlagen werden die Konzepte Transaktion, Synchronisation und Replikation untersucht. Abschließend werden die Transparenzkriterien von VDBMS mit denen aus Kapitel 2 hervorgegangenen Anforderungen verglichen.

3.1 Überblick

Ein Datenbanksystem läßt sich vereinfacht beschreiben als "ein System zur Beschreibung, Speicherung und Wiedergewinnung von umfangreichen Datenmengen, die in mehreren Anwendungsprogrammen benutzt werden. Es besteht aus der Datenbasis und den Verwaltungsprogrammen." [DI] Auch das MUI-MRT besteht aus einer Datenbasis - der Szenenbeschreibung und den Teilnehmerdaten - und mehreren Verwaltungs- bzw. Anwendungsprogrammen - Automat, Monitor, Visualisierung, etc. - die auf sie zugreifen. Die Daten sind nicht persistent gespeichert, sondern nur für die aktuelle zeitlich begrenzte Konferenz notwendig.

Die Datenbasis einer relationalen oder objektorientierten Datenbank besteht aus Tabellen oder Relationen. Die Spaltenüberschriften der Relationen heißen Attribute und die Zeilen Tupel oder Datensätze. Die Einträge in einer Spalte (Attributwerte) müssen alle vom gleichen Typ sein. In einer relationalen Datenbank sind nur wenige einfache Attributtypen erlaubt, z.B. Real, Integer, String und Boolean. In objektorientierten Datenbanken sind in der Regel beliebige Klassen (Objekte) als Attributtypen erlaubt.

Kunde			
Name	**Vorname**	**Alter**	**Wohnort**
Meier	Anna	24	Berlin
Müller	Robert	36	Köln
Schulz	Norbert	47	München

Auftrag			
Name	**Artikel**	**Stück**	**Preis**
Meier	Knopf	8	0,80 DM
Meier	Lampe	2	2,50 DM
Schulz	Topf	1	45,00 DM

Abbildung 3-1: Eine Datenbank mit den Relationen Kunde und Auftrag

Zwei Tabellen werden miteinander verbunden, indem eine Beziehung zwischen zwei Attributen gleichen Typs hergestellt wird. In der Beispieldatenbank aus Abbildung 3-1 wird diese Verbindung über das Attribut Name hergestellt. Mindestens eines der beiden Attribute muß ein Schlüsselattribut sein, das eine eindeutige Identifizierung eines Datensatzes ermöglicht. Die Beispieldatenbank ist nur solange funktionsfähig, bis zwei Kunden den gleichen Namen haben. Daher werden für die meisten Datenbanken zusätzliche Indizierungsfelder eingeführt (z.B. eine Kunden- oder Auftragsnummer). Mit Hilfe von Schlüsseln können

- Tabellen sortiert,
- verwandte Werte gruppiert und
- Verbindungen zu anderen Tabellen herstellt werden.

Die Beziehungen zwischen den Tabellen lassen sich mit einem Entity-Relationship-Diagramm [CP76] (ER-Diagramm Abbildung 3-2) darstellen. Der Pfeil mit der Beschriftung "gibt" wird in der dargestellten Form als "Ein Kunde gibt mehrere Aufträge" gelesen. 1 und n geben die Kardinalität der Beziehung auf der jeweiligen Tabelle an. Die unterstrichenen Attribute sind die Schlüsselfelder der Relationen.

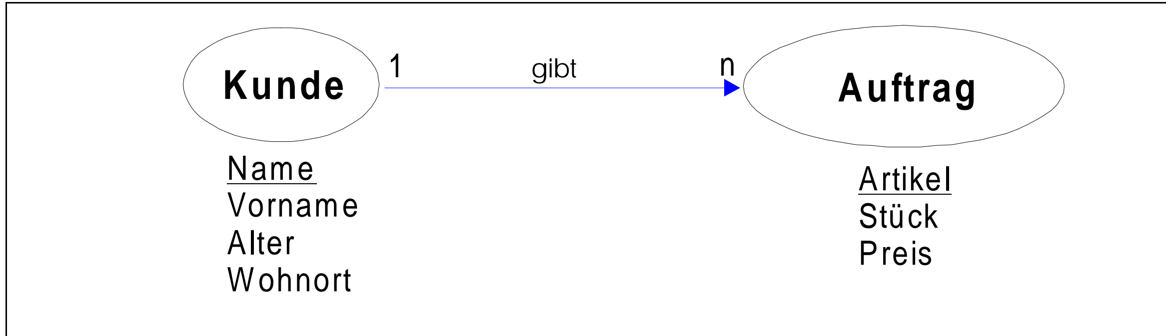


Abbildung 3-2: ER-Diagramm der Beispieldatenbank aus Abbildung 3-1

Die für den MUI-MRT relevanten Daten lassen sich, wie in Kapitel 2.1 und 2.2 spezifiziert, in Teilnehmer und Objektdaten einteilen. Für beide wird jeweils eine Relation gebildet, deren Attribute die erforderlichen Daten beinhalten. Abbildung 3-3 zeigt ein ER-Modell der beiden Relationen mit ihren Attributen und Beziehungen. Die Werte der einzelnen Felder und deren Bedeutung wird im Datenkatalog ausführlich beschrieben (Kapitel 7.3).

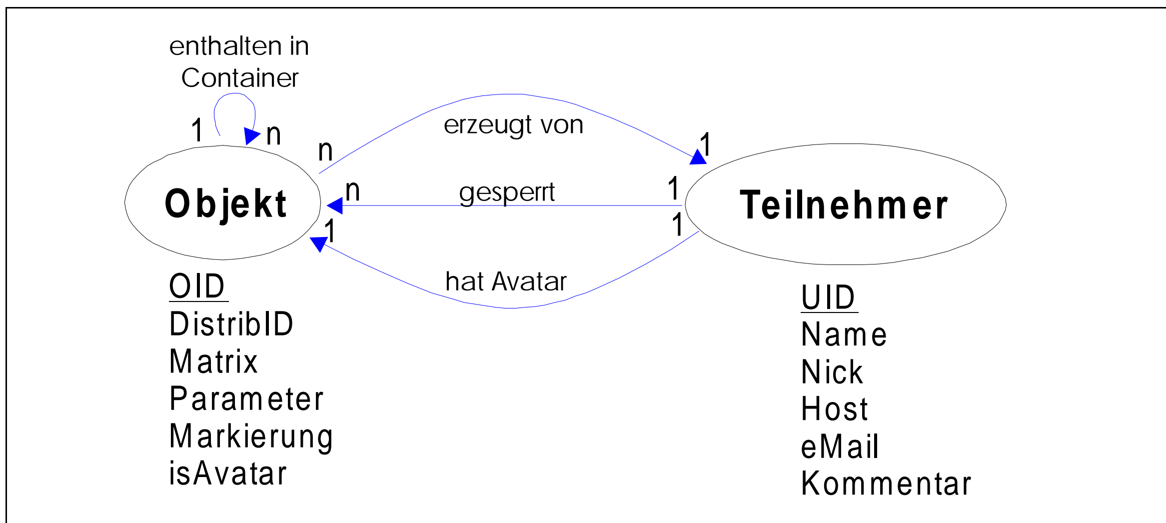


Abbildung 3-3: ER-Diagramm der MUI-MRT Daten

Zur eindeutigen Identifizierung der Objekte bzw. Teilnehmer werden eine Objekt ID (OID) und eine User ID (UID) eingeführt, die fortan als Schlüssel ihrer jeweiligen Relation dienen. Die Beziehungen zwischen den Relationen werden dementsprechend über die Speicherung des betroffenen Schlüssels in einem zusätzlichen Attribut gespeichert; z.B. erhält die Relation Objekt das Attribut `creatorID` um die Beziehung "erzeugt von" zu definieren.

Für einen effizienten Zugriff auf die Objekte der Szenenhierarchie wird also eine Tabelle der Objekte geführt. Basierend auf dieser Tabelle werden die Daten zwischen den Clients und dem Server ausgetauscht und nur diese Tabellen werden automatisch syn-

chronisiert. Die Szenenhierarchie, die der MRT verwendet, wird immer dann aktualisiert, wenn die Anwendung Zeit für eine Synchronisation beider Versionen hat. (Kapitel 3.3.3)

Auf eine Datenbank können die verschiedensten Operationen angewendet werden. Für unsere Zwecke reichen einfachste Grundfunktionen wie Lesen, Suchen, Ändern, Erzeugen und Löschen einzelner Datensätze aus.

3.2 Transaktionen

Jede Änderung an einer Datenbank wird als Transaktion bezeichnet. Eine Transaktion selbst besteht aus einer Folge von Änderungsoperationen. Bezüglich der Ausführung von Transaktionen garantiert ein Datenbanksystem (DBS) die Einhaltung des sogenannten Transaktionskonzeptes [Gr81, HR83, Hä88a, We88]. Dies betrifft die automatische Gewährleistung der folgenden vier ACID-Eigenschaften (Atomicity, Consistency, Isolation, Durability) von Transaktionen (nach [RE94]):

1. **Atomarität** ("Alles oder nichts")
   Änderungen einer Transaktion werden entweder vollkommen oder gar nicht in die Datenbank eingebracht. Diese Eigenschaft ermöglicht eine erhebliche Vereinfachung der Anwendungsprogrammierung, da Fehlersituationen während der Programmausführung (z.B. Rechnerausfall) nicht im Programm abgefangen werden müssen. Das Transaktionssystem sorgt dafür, daß die Transaktion in einem solchen Fall vollständig zurückgesetzt wird, so daß keine unerwünschten Spuren der Transaktion in der Datenbank verbleiben. Der Programmierer kann somit bei der Realisierung der Anwendungsfunktionen von einer fehlerfreien Umgebung ausgehen.
2. **Konsistenz** (Integrität erhalten)
   Die Transaktion ist die Einheit der Datenbankkonsistenz. Dies bedeutet, daß bei Beginn und nach Ende einer Transaktion sämtliche physischen und logischen Integritätsbedingungen erfüllt sind [SW85, Re87].
3. **Isolation**
   Datenbanksysteme unterstützen typischerweise eine große Anzahl von Benutzern, die gleichzeitig auf die Datenbank zugreifen können. Trotz dieses Mehrbenutzerbetriebes wird garantiert, daß dadurch keine unerwünschten Nebenwirkungen eintreten (z.B. das gegenseitige Überschreiben der selben Datenbankobjekte). Vielmehr bietet das DBS jedem Benutzer bzw. Programm einen "logischen Einbenutzerbetrieb", so daß parallele Datenbankzugriffe anderer Benutzer unsichtbar bleiben. Auch hierdurch ergibt sich eine erhebliche Vereinfachung der Programmierung.
4. **Dauerhaftigkeit**
   Die Dauerhaftigkeit von erfolgreich beendeten Transaktionen wird garantiert. Dies bedeutet, daß Änderungen dieser Transaktionen alle erwarteten Fehler (insbesondere Rechnerausfälle, Externspeicherfehler und Nachrichtenverlust) überdauern.

Transaktionen lassen sich gut auf den MUI-MRT übertragen. Grafische Objekte sind häufig aus mehreren Elementarobjekten zusammengesetzt. Ein Stuhl läßt sich beispielsweise aus 6 Boxen zusammensetzen. 4 Beine, Sitzfläche und Lehne. Um diesen Stuhl zu bewegen, müssen alle 6 Objekte bewegt werden. Tritt zwischen den Operationen ein Fehler auf, könnten die vier Beine bewegt worden sein, die Sitzfläche und Lehne jedoch nicht. Durch die Zusammenfassung dieser 6 Operationen zu einer Transaktion gilt die Atomarität. Entweder es wird der ganze Stuhl bewegt oder er bleibt, wo er war und die Konsistenz des Stuhles bleibt erhalten.

Zusätzlich wird der Kommunikationsaufwand für mehrere aufeinanderfolgende Änderungen minimiert, da Transaktionen in einem Block übertragen werden können.

3.3 Synchronisation

Eine der Haupteigenschaften verteilter Datenbanksysteme ist die gemeinsame Nutzung der Datenbasis durch viele Benutzer, die lesend und ändernd zugreifen können ohne die Konsistenz der Daten zu verletzen. Durch die Synchronisation paralleler Zugriffe wird die Konsistenz erhalten, wobei der Mehrbenutzerbetrieb vor den Anwendern verborgen bleibt. Wenn alle Transaktionen generell nacheinander ausgeführt würden, wäre die Konsistenz zwar gewährleistet, aber ein paralleler Betrieb ausgeschlossen. Es dürfte immer nur ein Anwender Daten ändern, während die anderen warten müßten.

Beim unkontrollierten Zugriff auf die Datenbasis im Mehrbenutzerbetrieb können eine Reihe von unerwünschten Phänomenen auftreten. Die wichtigsten dieser Anomalien sind verlorengegangene Änderungen ("lost update"), das Lesen "schmutziger" Änderungen ("dirty read"), die inkonsistente Analyse ("non repeatable read") sowie sogenannte Phantome. [Re87].

Beispiel: Anwender A holt sich den Datensatz einer Kugel, und ändert ihren Radius von 5 auf 10. Bevor er den Datensatz zurückschreibt, liest Anwender B die Kugel und ändert seinerseits den Radius von 5 auf 3. Jetzt schreibt Anwender A seine Änderung zurück und kurz darauf schreibt B seine darüber. Die Änderung von Anwender A ginge verloren.

Änderungen können verlorengehen, wenn zwei Transaktionen parallel dasselbe Objekt ändern, wobei die zuerst vorgenommene Änderung durch die zweite Transaktion fälschlicherweise überschrieben wird. Das Lesen schmutziger Änderungen, welche von noch nicht beendeten Transaktionen stammen, kann zu fehlerhaften Ergebnissen führen, wenn die ändernde Transaktion noch zurückgesetzt werden muß und somit ihre Änderungen ungültig werden. Die inkonsistente Analyse und das Phantom-Problem betreffen Lesetransaktionen, die aufgrund parallel durchgeführter Änderungen während ihrer Ausführungszeit unterschiedliche Datenbankzustände sehen.

Diese Anomalien werden durch Synchronisationsverfahren vermieden, welche das Korrektheitskriterium der Serialisierbarkeit erfüllen [EGLT76, BHG87]. Das heißt das Ergebnis einer parallelen Transaktionsausführung muß äquivalent zu dem Ergebnis irgendeiner der seriellen Ausführungsreihenfolgen der beteiligten Transaktionen sein. Zur Synchronisation werden verschiedene Verfahren eingesetzt, für die die Serialisierbarkeit nachgewiesen wurde. Im folgenden wird das Sperrverfahren betrachtet. Es ist dadurch gekennzeichnet, daß eine Anwendung vor dem Zugriff auf ein Objekt für die betreffende Transaktion eine Sperre erwirbt, deren Modus dem Zugriffswunsch entspricht. Im einfachsten Fall wird nur zwischen Lese- und Schreibsperren unterschieden. Dabei sind Lesesperren miteinander verträglich, während Schreibsperren weder mit sich noch mit Lesesperren kompatibel sind. So können bei gesetzter Lesesperre auf ein Objekt weitere Leseanforderungen gewährt werden, jedoch keine Schreibsperren. Bei gesetzter Schreibsperre sind alle weiteren Sperranforderungen abzulehnen. Ein Sperrkonflikt blockiert die Transaktion, deren Sperranforderung den Konflikt verursacht hat. Die wartende Transaktion wird aktiviert, sobald die unverträglichen Sperren wieder

freigegeben sind. Für den MUI-MRT werden Transaktionen generell zurückgesetzt, wenn sie auf Sperrkonflikte stoßen, anstatt auf die Freigaben von Sperren zu warten.

3.3.1 Zentrales Sperrprotokoll

Der naheliegende Ansatz zur Synchronisation in verteilten Datenbanksystemen liegt darin, sämtliche Sperranforderungen und -freigaben auf einem dedizierten Rechner zu bearbeiten. Als Vorteil ergibt sich, daß die Synchronisation quasi wie in einem zentralisierten Datenbanksystem abgewickelt werden kann, da im zentralen Knoten stets der aktuelle Synchronisationszustand bekannt ist. Insbesondere kann auch eine Deadlock-Erkennung (Kapitel 3.4) wie im Einrechnerfall vorgenommen werden.

Es sprechen jedoch einige Nachteile gegen ein solches Vorgehen:

- Jede Sperranforderung einer Transaktion, die nicht auf dem zentralen Knoten läuft, verursacht eine Nachricht, auf die die Transaktion synchron warten muß. Eine solche Nachrichtenhäufigkeit ist für Durchsatz und Antwortzeit gleichermaßen inakzeptabel.
- Der zentrale Knoten stellt einen Engpaß für Leistung und Verfügbarkeit dar ("single point of failure")
- Es wird keine Knotenautonomie unterstützt.

3.3.2 Mehrversionen-Synchronisationsverfahren

Das Mehrversionen-Synchronisationsverfahren strebt eine Reduzierung der Synchronisationskonflikte an, indem für geänderte Objekte zeitweilig mehrere Versionen geführt werden und Leser ggf. auf ältere Versionen zugreifen. Voraussetzung dabei ist, daß für jede Transaktion vorab Wissen darüber vorliegt, ob sie möglicherweise Änderungen vornimmt. Einer reinen Lesetransaktion T wird dabei während ihrer gesamten Laufzeit eine Sicht auf die Datenbank gewährt, wie sie bei ihrem Beginn gültig war; Änderungen, die während ihrer Bearbeitung vorgenommen werden, bleiben für T unsichtbar. Um dies zu realisieren erzeugt jede erfolgreiche Änderung eine neue Version des modifizierten Objekts; die Versionen werden im sogenannten Versionen-Pool verwaltet. Im Gegensatz zu Lesetransaktionen greifen Änderungstransaktionen stets auf die aktuelle Version des Objektes zu.

3.3.3 Zusammenfassung

Im Rahmen dieser Arbeit wird das Zentrale Sperrverfahren in Kombination mit dem auf 2 Versionen beschränkten Mehrversionen-Verfahren verwendet. Diese Kombination wird fortan als 2-Versionen-Ansatz bezeichnet.

Der Vorteil des zentralen Sperrprotokolles ist, daß der Server direkt entscheiden kann, ob eine Sperre gesetzt werden kann oder nicht. Damit der Server nicht jede Sperranfrage selbst überprüfen muß, wird die Information über gesetzte Sperren an alle Teilnehmer weitergeleitet, so daß jeder Client zunächst seine lokal vorliegende Sperrenliste überprüfen kann. Erst wenn diese keine Sperre anzeigt, wird der Server kontaktiert um eine Sperre anzufordern. So wird ein ständiges Anfragen beim Server nach Freigabe der Sperre verhindert.

Durch eine geschickte Verwendung des 2 Versionen-Ansatzes kann das Vergeben von Lesesperren und somit der Kommunikations- und Verwaltungsaufwand eingespart werden. Die Datenbasis wird stets in zwei Versionen vorgehalten. Die Szenenhierar-

chie, auf der das MRT basiert und auf der die Anwendung durchgeführt wird, dient als "ältere Version" oder Leseversion. Auf sie erhalten alle Leseoperationen kontinuierlich Zugriff. Die Szene ist vollständig vorhanden und die Anwendung kann von einer unveränderbaren Szene ausgehen, solange bis explizit eine Aktualisierung auf eine neuere konsistente Version angestoßen wird. Dies kann z.B. während einer Idle-Phase geschehen, oder zwischen je zwei Frames, also dann, wenn die Anwendung die Szenendaten nicht nutzt. So kann die Anwendung nahezu ungehindert lesend auf die Szene zugreifen.

Innerhalb der Datenhaltung werden die Daten in einer Tabelle gespeichert. Diese beinhaltet die aktuelle Version der Szene, die eine Replikation der Szene ist, die auf dem Server vorliegt. Die Erhaltung einer identischen Replikation wird in Kapitel 3.5 beschrieben.

Für Änderungen muß die Anwendung eine Schreibsperre vom Server anfordern. Die Änderungen werden direkt auf die aktuelle Version angewendet und entweder von der Anwendung simultan auf der Szenenhierarchie durchgeführt oder mit der nächsten Aktualisierung übernommen.

3.4 Deadlock Verhütung

Eine mit Sperrverfahren einhergehende Interferenz ist die Gefahr von Verklemmungen oder Deadlocks, deren charakteristische Eigenschaft eine zyklische Wartebeziehung zwischen zwei oder mehr Transaktionen ist. Im verteilten Fall können diese Verklemmungen zwischen Transaktionen verschiedener Rechner auftreten, so daß es zu sogenannten globalen Deadlocks kommt.

Beispiel: Zwei Teilnehmer wollen gleichzeitig zwei Objekte sperren. Der erste hat bereits Objekt 1 gesperrt, während der zweite bereits Objekt 2 gesperrt hat. Beide warten auf die Freigabe des jeweils anderen Objektes, und das theoretisch für immer.

Es gibt verschieden Strategien Deadlocks zu behandeln. Das einfachste Verfahren ist die sogenannte Deadlock Verhütung. Hierbei werden Deadlocks verhindert, ohne daß irgendwelche weiteren Maßnahmen erforderlich sind. Jede Transaktion muß zu ihrem Beginn bereits alle Sperren erfolgreich anfordern. Gelingt dies nicht, wird die Transaktion sofort abgebrochen und es ist kein Rollback (s.u.) außer der Freigabe der bereits gesperrten Objekte nötig. Verklemmungen werden weiterhin dadurch vermieden, daß Sperren in einer fest vorgegebenen Reihenfolge erfolgen.

Im obigen Beispiel bedeutet dies, daß der Teilnehmer, der sein erstes Objekt zuerst gesperrt hatte auch die Sperre auf das zweite Objekt direkt erhält und somit seine Transaktion durchführen kann. Der zweite Anwender kann auf die Beendigung dieser Transaktion warten, um selbst aktiv zu werden.

Die Deadlock-Verhütung hat für Datenbanksysteme keine praktische Relevanz, da zu Beginn einer Transaktion nicht alle zu sperrenden Elemente bekannt sein können und somit Obermengen gesperrt werden müssen. Zudem sind während der gesamten Transaktion alle Sperren aktiv. In unserem Fall ist dieses Verfahren hingegen ideal. Es wird ohnehin oft eine Entkopplung von Sperraktionen und Änderungsaktionen geben. Das Setzen von Markierungen ist gleichzeitig auch eine Sperre des Objektes ohne das Ziel es zu ändern. So wird man eine Sperre anfordern, um z.B. in einer virtuellen Szene aus

einem Tisch und einem Stuhl, den Stuhl unter den Tisch zu schieben. Man klickt also den Stuhl an und setzt eine Sperre. Dann wird er solange bewegt, bis er die gewünschte Position erreicht hat. Dabei könnte die Übertragung von Zwischenpositionen durchaus erwünscht sein, um die Transaktion verfolgen zu können.

Für die Teilnehmerverwaltung ist eine Sperrenstrategie nicht erforderlich, da die Änderungsrechte für alle Attribute und Datensätze fest zugeordnet sind. Benutzerdaten sind mit einer Pseudosperre versehen, die nur dem Benutzer selbst die Änderung erlaubt. Alle anderen administrativen Daten dürfen nur vom Server verändert werden. Somit ist die Sperrensetzung implizit vorgegeben. Zusätzlich ist die Teilnehmerliste teilrepliziert bei allen Teilnehmern vorhanden, mit den für die Clients wichtigen Daten.

3.5 Replikation

Der Vorteil voll-redundanter Datenbanken ist die Steigerung der Verfügbarkeit der Daten bei den Teilnehmern, denn an mehreren Knoten redundant gespeicherte Daten bleiben auch nach dem Ausfall eines Rechners erreichbar. Im Internet sind Ausfälle und Verzögerungen unvermeidbar. Die vollständige Replizierung ermöglicht den Teilnehmern die virtuelle Welt zu betrachten, ohne auf das Nachladen von Änderungen oder neuen Bereichen warten zu müssen. Erst wenn sie Änderungen vornehmen wollen müssen sie mit dem Server bewußt Kontakt aufnehmen.

Ein weiterer Vorteil voll-redundanter Datenbanken ist die Übernahme oder Verteilung der Serverfunktionalität auf andere bzw. mehrere Rechner. Fällt beispielsweise der Server aus, kann ein beliebiger an der Konferenz teilnehmender Rechner dessen Funktionen übernehmen. Für die Auswahl des entsprechenden Rechners bieten sich verschiedene Strategien an: Man könnte die Quality of Service aller Rechner bestimmen, die dann auf der Erreichbarkeit, der Rechnergeschwindigkeit oder anderen Parametern aufbaut. Ebenso muß beachtet werden, daß der gewählte Rechner die aktuelle Version der Szene hat, und im Falle einer Netzwerkpartitionierung ist darauf zu achten, daß nur in einer Partition ein neuer Rechner bestimmt wird (z.B. in der Partition in der mehr als die Hälfte aller Rechner liegen.), u.s.w. . Im Rahmen dieser Arbeit ist eine Server-Migration nicht realisierbar. In Kapitel 9 wird aber noch einmal darauf eingegangen.

Ein Nachteil replizierter Datenbanken ist der erhöhte Speicherbedarf durch die an allen Orten vorhandenen Daten. Da das MRT die vollständige Szenenbeschreibung immer benötigt, beschränkt sich dieser Mehraufwand auf die interne zweite Version.

Als weiterer Nachteil sind aber die hohen Änderungskosten zu erwähnen, die einen großen Kommunikationsaufwand erfordern, der in ortsverteilten Systemen besonders teuer ist. Ferner führen Replizierte Datenbanken zu einer Erhöhung der Implementierungskomplexität, da die Existenz der Replikanten dem Benutzer gegenüber transparent gehalten werden soll (Kapitel 3.6 Replikationstransparenz). Das DBS hat somit dafür zu sorgen, daß Änderungen automatisch auf alle Kopien übertragen werden, damit diese untereinander konsistent bleiben.

Im Folgenden werden zwei der wichtigsten Verfahren, der Write-All-Ansatz und das Primary-Copy-Verfahren, zur Aktualisierung und Synchronisierung von replizierten Datenbanken erläutert.

3.5.1 Write-All-Ansatz

Bei der Write-All-Read-Any- bzw. Read-Once, Write-All (ROWA)- Strategie wird die synchrone Änderung aller Replikate vor Abschluß einer Änderungstransaktion verlangt. Dadurch ist gesichert, daß jedes Replikat auf dem aktuellen Stand ist, so daß zum Lesen jede beliebige Kopie ausgewählt werden kann. Die Auswahl kann im Hinblick auf die minimalen Kommunikationskosten oder zur Unterstützung der Lastbalancierung erfolgen. Weiterhin ergibt sich für Lesezugriffe eine erhöhte Verfügbarkeit, da diese durchführbar sind, solange wenigstens eine Kopie erreichbar ist.

Diese Vorteile gehen jedoch auf Kosten der Änderungstransaktionen. Zum einen ist es beim Einsatz eines verteilten Sperrverfahrens erforderlich, vor jeder Änderung eine Schreibsperre auf allen Kopien zu erwerben, was einen enormen Zusatzaufwand an Kommunikation einführen kann. Zum anderen sind alle Knoten am Commit-Protokoll [RE94] zu beteiligen. Bei einem zwei-Phasen-Commit-Protokoll werden zunächst in Phase 1 die Änderungen an alle Knoten übertragen und dort protokolliert, bevor in der zweiten Commit-Phase die Replikate selbst aktualisiert und die Sperren freigegeben werden. Ein weiterer Schwachpunkt liegt darin, daß die Verfügbarkeit für Änderer schlechter ist als bei fehlender Replikation! Denn eine Änderung ist nicht mehr möglich sobald ein Rechner ausfällt, auf dem ein Replikat des zu ändernden Objekts gespeichert ist.

3.5.2 Write-All-Available-Variante

Eine Abschwächung des Verfügbarkeitsproblems ist die Write-All-Available-Variante [BHG87], bei der nur die Replikate der verfügbaren Rechner gesperrt und aktualisiert werden müssen. Für einen ausgefallenen Rechner werden die nicht vorgenommenen Modifikationen eigens protokolliert und nach Wiedereinbringen des Rechners nachgeholt. Dieses Verfahren eignet sich allerdings nicht bei Netz-Partitionierungen, da hierbei ein Auseinanderlaufen der in verschiedenen Teilnetzen vorgenommenen Änderungen droht. Ferner bleibt natürlich der hohe Mehraufwand für Änderer bezüglich Sperranforderungen und Commit-Protokoll.

3.5.3 Primary-Copy-Verfahren

Primary-Copy-Verfahren [AD76, St79] zielen auf eine effizientere Bearbeitung von Änderungen ab, indem eine Änderungstransaktion bei einer Modifikation nur eines der Replikate, die Primärkopie, zu ändern braucht. Die Aktualisierung der anderen Replikate wird dann asynchron vom Primärkopien-Rechner aus durchgeführt, im allgemeinen erst nach Ende der ändernden Transaktion ("as soon as possible"). Dabei können Nachrichten eingespart werden, indem der Primärkopien-Rechner mehrere Änderungen gebündelt an die kopienhaltenden Rechner weiterleitet. Dies geht dann jedoch zu Lasten einer verzögerten Anpassung der Replikate. Jetzt braucht nur noch beim Primärkopien-Rechner eine Sperre angefordert zu werden, und somit wird der Aufwand verteilter Schreibsperren umgangen. Die Behandlung der Lesezugriffe ist durch den 2-Versionen-Ansatzes jederzeit möglich. Die Aktualität der lokalen Kopien ist dabei aber nicht mehr gewährleistet, obwohl die Daten im allgemeinen höchstens wenige Sekunden veraltet sein dürften [Gr86].

In den MUI-MRT wird das Primary-Copy-Verfahren übernommen. Änderungen werden beim Server eingetragen und von ihm "as soon as possible" an die Teilnehmer verteilt.

3.6 Transparenzeigenschaften

An eine verteilte Anwendung, wie das MUI-MRT, werden folgende Anforderungen gestellt: (Regel 1-12 nach Date [Da90], Regel 13 und 14 sind allgemeine Grundregeln)

1. **Lokale Autonomie**
   Jeder Rechner sollte ein Maximum an Kontrolle über die bei ihm gespeicherten Daten haben. Der Zugriff auf diese Daten sollte nicht von anderen Rechnern abhängen.
2. **Keine Abhängigkeit von zentralen Systemfunktionen**
   Zur Unterstützung einer hohen Autonomie und Verfügbarkeit sollte der Client nicht von zentralen Systemfunktionen abhängen. Solche Komponenten stellen einen potentiellen Leistungengpaß dar.
3. **Hohe Verfügbarkeit**
   Die Datenhaltung sollte trotz Fehlern im System (z.B. Rechnerausfall) nicht unterbrochen werden.
4. **Ortstransparenz**
   Die physische Lokation von Datenobjekten sollte für den Benutzer verborgen bleiben. Der Datenzugriff sollte wie auf lokale Objekte möglich sein.
5. **Fragmentierungstransparenz**
   Die Zerlegung einer Relation auf mehrere Rechner sollte für den Benutzer transparent bleiben. Dies ist für das MUI-MRT nicht erforderlich, da die Datenlisten immer vollständig repliziert werden, und somit keine Fragmentierung auftritt.
6. **Replikationstranzparenz**
   Die replizierte Speicherung der Daten sollte für den Benutzer unsichtbar bleiben; die Wartung der Redundanz obliegt ausschließlich der Datenhaltung.
7. **Verteilte Anfrageverarbeitung**
   Innerhalb einer Operation sollte auf Daten mehrerer Rechner zugegriffen werden können. Beim MUI-MRT ist nur ein Zugriff auf die lokalen Daten erforderlich, somit ist diese Anforderung irrelevant.
8. **Verteilte Transaktionsverwaltung**
   Die Transaktionseigenschaften sind auch bei verteilter Bearbeitung einzuhalten, wozu Recovery- und Synchronisationstechniken bereitzustellen sind.
9. **Hardware-Unabhängigkeit**
   Die Verarbeitung sollte auf verschiedenen Hardwareplattformen möglich sein. Sämtliche Hardwareeigenschaften sind für den Benutzer zu verbergen.
10. **Betriebsystemunabhängigkeit**
    Die Benutzung sollte unabhängig vom eingesetzten Betriebssystem sein.
11. **Netzwerkunabhängigkeit**
    Die verwendeten Kommunikationsprotokolle und -netzwerke sollten ohne Einfluß auf die Verarbeitung bleiben.
12. **Datenbanksystemunabhängigkeit**
    Es ist nicht erforderlich diese Forderung für das MUI-MRT zu erfüllen.
13. **Verteilungstransparenz**
    Für den Anwender sollte die geographische oder logische Verteilung der Daten über die verschiedenen Rechner verborgen bleiben und wie ein lokaler Zugriff wirken.
14. **Leistungstransparenz**
    Die verteilte Bearbeitung von Anfragen und Transaktionen sollte trotz Kommunikationsverzögerungen möglichst keine merklichen Verschlechterungen in den Bearbeitungszeiten verursachen.

3.7 Zusammenfassung

Der Einsatz eines VDBMS für die Realisierung des MUI-MRT ist sicher nicht das richtige Mittel, da dies einen gewaltigen Overhead mit sich bringen würde, der im Vergleich zur Datenmenge, einer temporär existierenden virtuellen Welt (Szene), übertrieben wäre. Aber die Auswahl der in diesem Kapitel zusammengestellten Techniken, kann gut eingesetzt und in vereinfachter Form verwendet werden.

Die Darstellung der Daten innerhalb der Datenhaltung wird in Tabellen nach Art objektorientierter Datenbanken vorgenommen. Es werden zusätzliche Schlüsselfelder OID und UID eingeführt, sowie weitere Attribute, die die Beziehungen zwischen und innerhalb der Tabellen beschreiben. Änderungen werden in Transaktionen zusammengefaßt. Der in Kapitel 3.3.3 beschriebene 2-Versionen-Ansatz sorgt für die globale Serialisierbarkeit dieser Transaktionen. Die Deadlock-Verhütung verhindert den Stillstand des gesamten Systems durch Sperrkonflikte. Für die Replikation der Daten bei allen Teilnehmern wird das Primary-Copy-Verfahren eingesetzt.

Erweiterung der Aufgaben-/Ereignisliste für die Szenenverwaltung aus Kapitel 2.8:

- Schreibsperren auf Objekte setzen
- Starten, Beenden und Abbrechen von Transaktion
- Versionen der Szenendaten aktualisieren

Zusätzlich sollen die im vorhergehenden Kapitel aufgeführten Transparenzeigenschaften weitgehend eingehalten werden.

4 Common Object Request Broker Architecture

Im Jahre 1989 wurde von den Firmen 3Com Corporation, American Airlines, Canon, Inc., Data General, Hewlett-Packard Company, Philips Telecommunication, Sun Microsystems und Unisys Corporation die Object Management Group (OMG) ins Leben gerufen. Ziel dieses Konsortiums ist es, Standards zur Anwendungsentwicklung in verteilten heterogenen und objektorientierten Umgebungen zu entwickeln und zu spezifizieren.

In diesem Kapitel wird eine kurze Übersicht der Object Management Architectur (OMA) von OMG gegeben und dann näher auf die Kernkomponente, die Common Object Request Broker Architecture (CORBA) Spezifikation, und ihre Subkomponenten eingegangen. [OMG95a]

4.1 Object Management Architecture

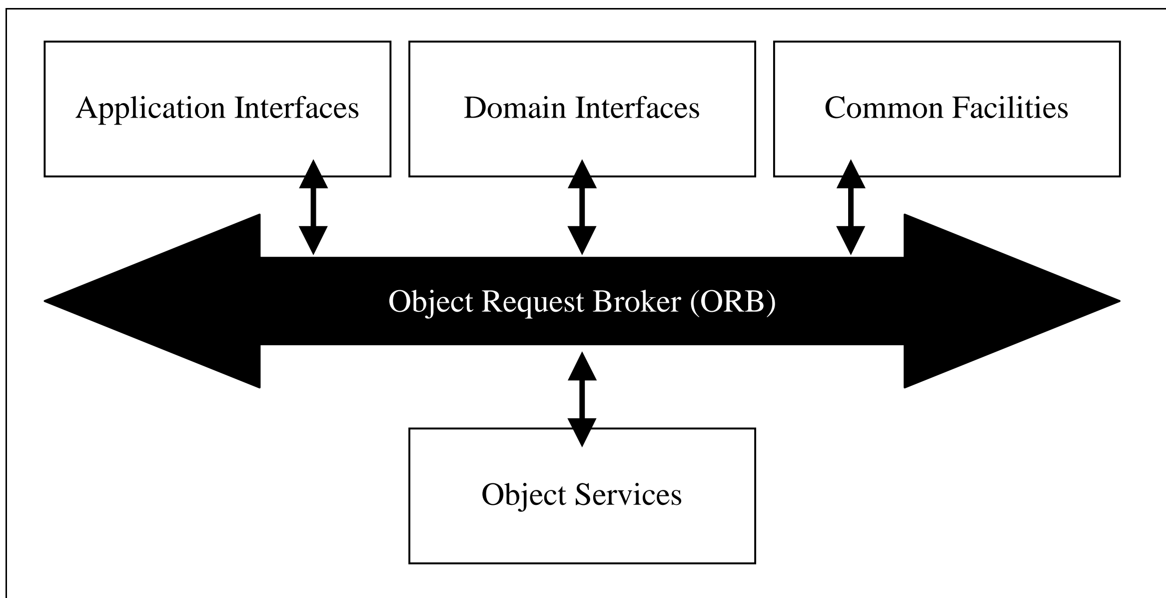


Abbildung 4-1: Das OMA Referenz-Modell (Quelle OMG)

Der zentrale Bestandteil der OMA ist der Object Request Broker (ORB). Der ORB stellt eine Infrastruktur zur Verfügung, die einen weitgehend transparenten Informationsaustausch zwischen Clients und Serverobjekten ermöglicht, unabhängig von spezifischen Plattformen und Techniken der Objektimplementierung. Daß auch Serverobjekte Clients weiterer Objekte sein können, versteht sich von selbst; komplexe, vernetzte Anwendungen können dadurch verwirklicht werden.

Die Object Services sind eine Sammlung fundamentaler Schnittstellen (und Objekte) zur Realisierung von CORBA-Anwendungen, die in vielen verteilten Anwendungen eingesetzt werden können. Diese können flexibel miteinander kombiniert werden. Sie erleichtern und vereinheitlichen die Anwendungsentwicklung. Beispielsweise ist ein Dienst, der es ermöglicht andere verfügbare Dienste zu ermitteln, für jede Art Anwendung sinnvoll. Zwei Object Services die diese Aufgabe erfüllen sind:

- Naming Service - Objekte können aufgrund eines Namens gefunden werden
- Trading Service - Objekte können aufgrund von Eigenschaften gefunden werden

Weitere Object Services sind LifeCycle Management, Security, Transaction und Event Notification [OMG95b].

Ziel der Common Facilities sind Schnittstellen auf höherer Anwendungsebene als die Object Services. Die OMG beschreibt sie als "Endnutzer orientiert". Ein Beispiel sind Schnittstellen für auf Objekten basierenden Dokumenten, um z.B. eine Tabellenkalkulation in einen Report einzubinden (ähnlich "Microsoft OLE") oder eMail zu versenden.

Domain Interfaces stehen auf ähnlich hoher Ebene wie die Common Facilities. Sie enthalten branchenspezifische Schnittstellen. Die OMG erwähnt unter anderem Finanz-, Gesundheits- und Telekommunikationswesen.

Application Objects bezeichnen die anwendungsspezifischen Schnittstellen. Da die OMG keine Anwendungen entwickelt (sondern nur Spezifikationen) sind diese Schnittstellen nicht standardisiert.

4.2 Die Struktur eines Object Request Broker

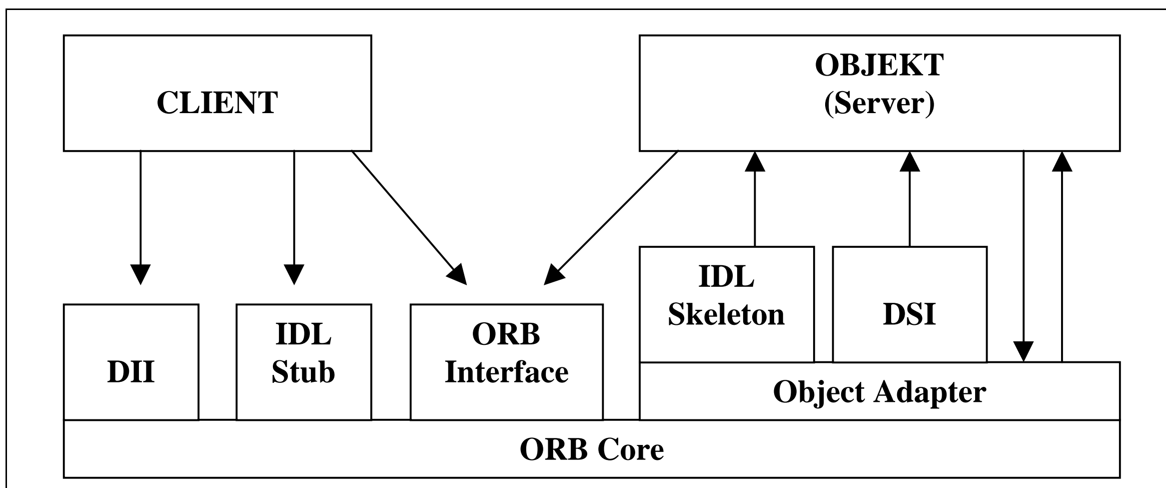


Abbildung 4-2: CORBA ORB Architektur

Die Common Object Request Broker Architecture (CORBA) ist ein von der OMG definierter Standard für den ORB im OMA Referenz-Modell. CORBA definiert alle Schnittstellen die für Objekte durch den ORB verfügbar sind. Die Entwicklung verteilter Anwendungen wird damit nicht schwieriger als die Entwicklung zentralisierter Anwendungen. D.h. der Anwender muß sich um die meisten Probleme, die im Kapitel 2.4 erwähnt wurden, nicht kümmern. Es wird eine Infrastruktur für die Integration von Anwendungskomponenten in eine verteilte Anwendung zur Verfügung gestellt. Diese Eigenschaften des ORB erlauben es Anwendungsentwicklern sich mehr um die Problemstellungen der eigenen Anwendung zu kümmern, und weniger um die verteilte System Programmierung. Der ORB übermittelt Anfragen zu Objekten und liefert die Antwort zu dem Client zurück, der die Anfrage stellte. Es werden folgende Punkte verborgen:

- **Ort des Objektes:** Der Client weiß nicht, wo sich das Zielobjekt befindet. Es kann in einem anderen Prozeß auf einer anderen Maschine irgendwo im Netzwerk, auf der selben Maschine in einem anderen Prozeß oder innerhalb des gleichen Prozesses existieren.

- **Implementation des Objektes:** Der Client weiß nicht, wie das Zielobjekt implementiert ist, ob es mit einer Programmier- oder Scriptsprache geschrieben wurde, oder auf welchem Betriebssystem oder welcher Hardwareplattform es ausgeführt wird.
- **Konvertierung der Parameter:** Die Parameter werden vor dem Versand zu und von einem Server verpackt (Marshaling). Der Client braucht sich keine Gedanken darüber zu machen, ob Architektur- oder Kompilierungsunterschiede zwischen den Datenformaten verschiedener Maschinen bestehen.
- **Ausführungsstatus des Objektes:** Bei einer Anfrage braucht der Client nicht zu wissen, ob das Zielobjekt aktiviert und damit bereit ist, Anfragen entgegen zu nehmen. Der ORB startet Objekte transparent, bevor eine Anfrage an sie weitergeleitet wird.
- **Transportmechanismus:** Der Client weiß nicht, welchen Transportmechanismus (z.B. TCP/IP, shared memory, local method call, etc.) der ORB verwendet um die Anfrage an das Zielobjekt weiterzuleiten und die Antwort zurückzuliefern.

Inhärente Probleme verteilter Systeme, wie Übertragungsverzögerungen, Netzwerkausfälle und Sicherheitsprobleme, existieren in CORBA weiterhin.

Für eine Anfrage spezifiziert der Client das Zielobjekt über eine Objektreferenz. Mit der Erzeugung eines CORBA-Objekt wird immer auch eine Objektreferenz generiert. Wenn ein Client diese benutzt, referenziert er damit immer das Objekt, für das die Objektreferenz erzeugt wurde, solange dieses Objekt existiert. Mit anderen Worten, eine Objektreferenz referenziert immer nur ein einzelnes Objekt. Objektreferenzen sind unveränderbar, so daß ein Client nicht in sie hineinsehen und sie nicht ändern kann. Nur der ORB weiß, was sich "innerhalb" einer Objektreferenz befindet. Clients können Objektreferenzen auf folgende Arten erhalten:

Objekt erzeugen: Ein Client kann ein neues Objekt erzeugen, um eine Objektreferenz zu erhalten. In CORBA gibt es keine spezielle Client-Operation um Objekte zu erzeugen. Objekte werden erstellt durch den Aufruf von Erzeugungsanfragen an andere Objekte, die sogenannten Fabrikobjekte. Eine derartige Anfrage liefert eine Objektreferenz des neuen Objektes zurück.

Verzeichnisdienste (Directory-Service): Ein Client kann einen Service nutzen, um eine Objektreferenz zu erhalten. Die folgenden zwei Dienste wurden bereits erwähnt: Der Naming-Service und der Trading-Service erlauben dem Client anhand eines Namens oder der Eigenschaften eines Objektes Objektreferenzen zu erfahren. Diese Dienste können nicht wie Fabrikobjekte neue Objekte erzeugen. Sie speichern die Objektreferenzen mit Zusatzinformationen in Datenbanken ab und liefern sie auf Anfrage zurück.

Konvertierung in Strings und zurück: Eine Anwendung kann den ORB nutzen, um eine Objektreferenz in einen String zu konvertieren. Dieser kann dann auf Festplatte oder in Datenbanken etc. gespeichert oder via eMail verschickt werden. Dieser String kann später wieder eingeladen und vom ORB in eine Objektreferenz zurückverwandelt werden.

Da CORBA keine speziellen Objekt-Erzeugungs-Operationen anbietet, werden Objektreferenzen immer über Anfragen an andere Objekte erhalten. Dies wirft die Frage auf,

wie eine Anwendung eine erste Objektreferenz erhalten kann. Der ORB enthält einen einfachen kleinen eigenen "Naming-Service", über den die Objektreferenzen der größeren Verzeichnisdienste gewonnen werden können.
Zum Beispiel erhält man durch Übergabe des Strings "Name-Service" an die Funktion `resolve_initial_rerferences()` eine Objektreferenz für einen Naming-Service, der dem ORB bekannt ist.

4.3 Interface Definition Language

Bevor ein Client Anfragen an ein Objekt machen kann, muß es die Operationen kennen, die dieses Objekt unterstützt. Ein Objektinterface spezifiziert die Operationen und Typen, die ein Objekt unterstützt und definiert so die Anfragen, die an ein Objekt gestellt werden können. Objektschnittstellen werden in OMG Interface Definition Language (OMG IDL) definiert. Diese Interfaces sind vergleichbar mit den Klassendefinitionen in C++. Ein kleines Beispiel für eine OMG IDL Interfacedefinition ist folgendes math-Objekt:

```
interface math {
      void seta(in double a);
      void setb(in double b);
      double add();
};
```

Abbildung 4-3: OMG IDL Interface des math-Objektes

Das math-Objekt unterstützt 3 Funktionen - `seta()` und `setb()` setzen die Werte des Objektes und `add()` liefert das Ergebnis der Addition der beiden Werte zurück.

Ein wichtiges Merkmal von OMG IDL ist seine Sprachunabhängigkeit. Da OMG IDL eine deklarative Sprache ist, wird die Trennung von Interface und Objektimplementierung erzwungen. So können Objekte mit verschiedenen Programmiersprachen implementiert werden, und dennoch miteinander kommunizieren. Sprachunabhängigkeit der Interfaces ist in heterogenen Systemen wichtig, da nicht alle Programmiersprachen auf allen Plattformen verfügbar sind.

OMG IDL stellt eine Reihe von Typen bereit, die in vielen Programmiersprachen verwendet werden. Es werden Basistypen wie `long`, `double` und `boolean`, sowie konstruierte Typen wie `struct` und `union` also auch die template Typen `sequence` und `string` verwendet. Typen werden genutzt, um Parameter- und Rückgabetypen der Operationen zu spezifizieren. Wie im obigen Beispiel zu sehen ist, werden Operationen innerhalb von `Interfaces` angegeben, um die Dienste zu spezifizieren, die die Objekte anbieten, welche das Interface unterstützen. Um Ausnahmebedingungen zu behandeln, die während einer Operation auftreten können, bietet OMG IDL die `exception` Definition an. Wie bei Struct-Typen können mehrere Datentypen angegeben werden. Das OMG IDL Konstrukt `module` erlaubt die Einteilung der Definitionen in Namensbereiche, um Namenskollisionen zu vermeiden.

Durch das Kompilieren einer IDL-Spezifikation erhält man sogenannte Stubs und Skeletons. Ein Stub ist eine Proxy-Klasse, die mit der Client-Anwendung kompiliert wird. Für obiges Beispiel besteht der Stub aus einer Klasse math, deren Funktionen transparent die Funktionen des auf dem Server liegenden math-Objektes aufrufen, das Ergebnis holen und dem Client zurückgeben. Der Skeleton ist eine Klasse, die mit dem Server

kompiliert wird. Der Skeleton nimmt die Aufrufe des Clients auf und generiert die Zugriffe auf das Zielobjekt.

4.3.1 Eingebaute Typen

OMG IDL unterstützt folgende Basistypen:

- `long` (`signed` und `unsigned`) - 32 bit Typ
- `long long` (`signed` und `unsigned`) - 64 bit Typ
- `short` (`signed` und `unsigned`) - 16 bit Typ
- `float`, `double`, `long double` - floating point Typen (IEEE 754 - 1985)
- `char`, `wchar` - character und wide character Typen
- `boolean` - Wahrheitswert
- `octet` - 8 bit Typ
- `enum` - Aufzählungstyp
- `any` - etikettierter Typ, der jeden Wert eines OMG IDL Typs enthalten kann, inklusive aller Standard- und selbstdefinierten Typen.

Die CORBA Spezifikation definiert präzise die Größe aller Basistypen, um die Interoperabilität auf heterogenen Hardwareplattformen sicherzustellen.

4.3.2 Konstruierte Typen

OMG IDL unterstützt folgende konstruierte Typen:

- `struct` - eine Sammlung mehrerer Datentypen (ähnlich einem C/C++ struct)
- `union` - ein zusammengesetzter Typ aus einem Diskriminator-Typ und den Datentypen. In Abhängigkeit vom Diskriminator stellt ein union-Typ unterschiedliche Werte dar. (Abbildung 4-4)

```
typedef enum { TEXTUAL, NUMBER } Type;
union User switch(Type) {
      case TEXTUAL: string loginName;
      case NUMBER: short userId;
};
```

Abbildung 4-4: Beispiel einer OMG IDL union-Definition

4.3.3 Template Typen

Zusätzlich unterstützt OMG IDL Template Typen, deren Eigenschaften zum Deklarationszeitpunkt festgelegt werden:

- `string` und `wstring` - String Typen. Beide können beschränkt und unbeschränkt deklariert werden. `string<10>` deklariert einen String der Länge 10 und `string` einen unbegrenzten String.
- `sequence` - ein linearer Container dynamischer Länge, dessen Elementtyp und Länge in Klammern angegeben wird. `sequence<factory>` definiert eine unbeschränkte Sequenz von `factory` Objektreferenzen. Mit `sequence<string,10>` wird eine beschränkte Sequenz von maximal 10 Strings definiert.
- `fixed` - ein Fixpunkt dezimal Wert mit maximal 31 Stellen. `fixed<5,2>` hat z.B. 5 Stellen inklusive zwei Nachkommastellen: 999,99.

4.4 Beispielprogramm

Eine typische Client-Anwendung, die eine Verbindung zum math-Objekt aufbaut und einige Anfragen durchführt, zeigt Abbildung 4-5. Der Client führt einige Initialisierungen durch und stellt die Verbindung zum math Objekt her. Danach nutzt der Client die Objektreferenz zum Aufrufen von Funktionen wie bei einem normalen C++-Objekt.

```
main(int argc, char *argv[])
{
      // initialize the ORB
      CORBA_ORB_ptr orb = CORBA_ORB_init(argc, argv);

      // get the object-reference somehow
      CORBA_Object_var obj = ...
      math_var m = math::_narrow(obj);

      // use the math object
      m->seta(3.1);
      m->setb(4.3);
      cout << m->add();
}
```

Abbildung 4-5: Client Anwendung für das math-Objekt

```
// implementation of the math object
class math_impl : public math_skel {
      CORBA_Double x, y;
public:
      void seta(CORBA_Double a) { x = a; };
      void setb(CORBA_Double b) { y = b; };
      CORBA_Double add()        { return (x + y}; };
}

// server's main
main(int argc, char *argv[])
{
      // initialize the ORB and BOA
      CORBA_ORB_ptr orb = CORBA_ORB_init(argc, argv);
      CORBA_BOA_ptr boa = orb->BOA_init(argc, argv);

      // instantiate math Object
      math_impl math_server;

      // tell the BOA that the object ist active and ready
      boa->obj_is_ready(&math_server);

      // Tell the BOA to enter the event-loop
      boa->impl_is_ready();
}
```

Abbildung 4-6: Server für das math-Objekt

Der Server hingegen muß die Funktionen, die das Objekt anbietet implementieren und das Objekt instanziieren. Abbildung 4-6 zeigt den Code einer Serverimplementierung. Der Server benutzt den Basic Object Adapter (BOA), um die Implementation des Objektes zu registrieren und auf Anfragen zu warten. Der BOA ist ein CORBA-Interface für Serverobjekte, mit dem Objektreferenzen erhalten und Server Implementationen registriert werden können. Der BOA ist weiterhin zuständig für die Zuordnung von

Objektreferenzen zu den zugehörigen Objekten, und unterstützt somit den ORB bei der Suche nach benötigten Objekten.

4.5 Dynamic Invocation Interface

Zusätzlich zu statischen Aufrufen über Stubs und Skeletons bietet CORBA zwei Schnittstellen für dynamische Funktionsaufrufe: Dynamic Invocation Interface (DII) und Dynamic Skeleton Interface (DSI).

Das DII ermöglicht es einer Client-Anwendung Anfragen an beliebige Objekte zu stellen, ohne zum Kompilierungszeitpunkt Informationen über dessen Schnittstelle zu haben.

Da jedes Objekt in CORBA von `CORBA_Object` abgeleitet ist, kann von jedem mit `create_request` ein Request Pseudo-Objekt erzeugt werden. Durch den Aufruf dieser Operation über eine Objektreferenz kann eine Anwendung eine dynamische Anfrage an ein Zielobjekt erzeugen. Bevor die Anfrage durchgeführt wird, müssen die Argumentwerte durch Aufrufe von Funktionen des Request Pseudo-Objektes gesetzt werden. Die Typen der Argumente können über das Interface Repository [OMG95a] festgestellt werden.

Ist das Request Pseudo-Objekt erzeugt und mit Parametern ausgestattet kann es auf drei Arten ausgeführt werden:

1. **Synchroner Aufruf** - der Client aktiviert den Aufruf und blockiert beim Warten auf die Antwort, auch wenn kein Rückgabewert erwartet wird.
2. **Semisynchroner Aufruf** - der Client aktiviert den Aufruf und fährt mit der Arbeit fort, während der Request weitergeleitet und ausgeführt wird. Später werden dann die Ergebnisse eingesammelt. Dies ist nützlich, wenn der Client mehrere unabhängig voneinander und lang laufende Dienste aufruft. Statt jeden einzeln aufzurufen und auf eine Antwort zu warten, können alle Anfragen parallel durchgeführt werden. Die Antworten werden behandelt, wenn sie eintreffen.
3. **Oneway Aufruf** - der Client aktiviert den Aufruf und fährt mit der Arbeit fort; eine Antwort gibt es nicht. Diese Art wird manchmal als "fire and forget" bezeichnet.

```
// get the object-reference somehow
CORBA_Object_var obj = ...
math_var m = math::_narrow(obj);

CORBA_Request_ptr request = m->_request("seta");
CORBA_NVList_ptr arguments = request->arguments();
CORBA_Double a = 3.1;
CORBA_Any any_a;
any_a <<= a;
arguments->add_value("a", any_a, CORBA_ARG_IN);
request->send_oneway();
```

Abbildung 4-7: Aufruf von `seta()` über das DII

Zur Zeit ist das DII die einzige Möglichkeit eine semisynchrone Übertragung durchzuführen, da die statischen Methoden lediglich den synchronen und den oneway Aufruf

unterstützen. Die OMG ist jedoch dabei einen Asynchronous Messaging Service zu spezifizieren, der diese Lücke schließt und das Handling sicherer und einfacher gestalten wird.

4.6 Dynamic Skeleton Interface

Das Analogon zum DII ist auf der Serverseite das Dynamic Skeleton Interface (DSI). Genauso wie das DII einem Client erlaubt Anfragen zu stellen, ohne Zugriff auf einen statischen Stub zu haben, erlaubt das DSI einem Server Anfragen an Objektimplementierungen weiterzuleiten, ohne statische Skeletons für diese Objekte zu besitzen.

4.7 Bewertung

Microsofts COM definiert einen auf dem Objektmodell basierenden Standard für die Kommunikation von Objekten die sich auf einem Rechner befinden und wurde nachträglich für den Zugriff auf verteilte Anwendungen zu DCOM erweitert [MS2].

DCOM ist nicht mit dem Grundgedanken an ein plattformübergreifendes System entwickelt worden und schleppt außerdem noch den Ballast früherer Versionen aus Kompatibilitätsgründen mit sich herum. Die SAG (Software AG) hat DCOM für einige UNIX-Plattformen (SUN, HP) entwickelt, in dem Teile der WIN32-API und der Windows NT Services portiert bzw. emuliert wurden [iX2]. DCOM wird es weiterhin nur aus einer Hand (Microsoft für Windows, SAG für UNIX) geben. Unter UNIX wird ein "kleines" Windows-System installiert um DCOM zu nutzen. Dies ist kein besonders überzeugender Ansatz. Von vielen wird CORBA deshalb ein effizienteres Design bescheinigt. Dank eines sparsameren Umgangs mit Systemaufrufen und einem geringeren Kommunikationsaufwand ist eine höhere Performance zu erwarten, die die Implementierungen von CORBA durch die verschiedenen Hersteller jedoch nicht immer bestätigen können [MC].

Die SAG plant auch die grafische Oberfläche zu portieren. Für Windows-Entwickler dürfte es erfreulich sein, die vorhandenen Tools und Methoden auch unter UNIX verwenden zu können.

Allgemein ist daher die persönliche Präferenz für eines der Systeme ausschlaggebend, sowie die hauptsächlich zu unterstützende Plattformwelt, Windows oder generell UNIX. Da das MRT von Beginn an plattformübergreifend konzipiert war und in beiden Welten verwendet wird, wurde CORBA für die Implementierung der Datenhaltung gewählt.

5 Eigenschaften von ORBacus

Die Suche eines passenden ORB geschah nach den folgenden Gesichtspunkten:

- Ist eine Threadunterstützung vorhanden und somit eine parallele Ausführung des Client neben dem MRT möglich?
- Werden alle Plattformen des MRT unterstützt?
- Ist das DII enthalten und eine semisynchrone Übertragung möglich?
- Sind Dokumentation und Support vorhanden?

ORBacus unterstützt alle vom MRT genutzten Plattformen, ist für akademische Zwecke kostenlos, bietet die für diese Arbeit erforderlichen Elemente/Dienste, ist mit allen Sourcen verfügbar. Es erlaubt eine einfache Einbindung von Threads über das Zusatzpaket JTC auf allen erforderlichen Plattformen. Neben der Implementierung in C++ gibt es das gleiche Paket auch für Java. Laut Herstellerangaben ist ORBacus außerdem eine der schnellsten Implementierungen des CORBA-Standards. Eine ausführliche Dokumentation und Support über eine Mailingliste direkt mit den Entwicklern ist vorhanden. ORBacus wird kontinuierlich weiterentwickelt.

5.1 Unterstützungsumfang der CORBA-Spezifikation

ORBacus 3.1.1 ist ein Object Request Broker (ORB) der die CORBA Spezifikation erfüllt und folgende Punkte unterstützt.

- Vollständiger CORBA IDL Umfang
- Vollständiges CORBA IDL-to-C++ Mapping
- Vollständiges CORBA IDL-to-Java Mapping
- Die CORBA Services *Naming, Event* and *Property*
- Unterstützung durch Single- und Multi-Threaded Umgebungen mit den verschiedenen Modellen: *Blocking*, *Reactive*, *Threaded*, *Thread-per-Client*, *Thread-per-Request* und *Thread Pool*
- Vollständige Unterstützung dynamischer Programmierung mit: Dynamic Invocation Interface, Dynamic Skeleton Interface, Interface Repository und Dynamic Any

Durch das Vorliegen der gesamten Sourcen ist eine Reduzierung des Umfangs von ORBacus auf die minimal benötigten Funktionen möglich. So können die Applikationen entsprechend schlanker gehalten werden.

5.2 Unterstützte Plattformen

ORBacus unterstützt die Plattformen auf denen das MRT compiliert werden kann und einige darüber hinaus:

ORBacus für C++:

* SGI Irix 6.2 und 6.3 mit den SGI C++ Compilern 7.0.1, 7.1 und 7.2
* SUN Solaris 2.5 mit SUN C++ Compiler 4.1 und 4.2
* SUN Solaris 2.5 mit GNU C++ Compiler 2.7.2
* HP-UX B.10.20 mit HP aC++ Compiler A.01.00 und A.01.03
* AIX Version 4.2.1 mit dem AIX C Set ++ Compiler (xlC 3.1.4.6)
* Linux 2.0 mit GNU C++ Compiler 2.7.2
* Windows NT 4.0 mit Visual C++ 4.2/5.0
* Windows 95 mit Visual C++ 4.2/5.0

ORBacus für Java:

* SUN's JDK 1.0.2 oder 1.1.x
* Microsoft's Visual J++ 1.1

5.3 Erweiterungen gegenüber dem CORBA - Standard

ORBacus bietet nur Erweiterungen an, die innerhalb der Anwendung genutzt werden und hauptsächlich der Vereinfachung der Programmierung dienen. Hier werden nur die in dieser Arbeit verwendeten Erweiterungen kurz aufgeführt. Eine ausführliche Beschreibung ist in [OB] enthalten.

- Erweiterung des String-Typs um den += Operator, der Strings und auch einfache Typen wie `long` oder `int` konvertiert und an einen String anhängt.
- Erweiterung des Sequence-Typs um `remove()`, `insert()` und `append()`.
- Addressierung eines Objektes über eine Internet Interorb Protokoll Adresse (IIOP) in der Form "iiop://hostname:port/objektname"
- Typanpassung von Variablen durch die Zusatzmethoden `in()`, `out()`, `inout()` und `_retn()` für die entsprechend durch OMG IDL definierten Parameter.

6 Entwurf der Datenhaltungsschicht

6.1 Szenenhierarchie übertragen

Durch den Parser wird die Beschreibung einer Szene aus einem MSD- oder VRML-Skript in eine Hierarchie von Grafik- und Kontrollobjekten übersetzt. Die Objekthierarchie ist für die einmalige Erzeugung durch den Parser konzipiert, weshalb keine Änderungsfunktionen vorhanden sind (die auch nicht im Rahmen dieser Arbeit eingeführt werden sollen (Kapitel 2.7)) und keine Funktion, um den Zustand eines Objektes auszulesen. Das heißt ein Objekt kann nur durch die Konstruktionsparameter erzeugt werden und ist danach unveränderbar (mit einigen Ausnahmen s.u.). Das bedeutet, ein Objekt wird durch diese Parameter eindeutig spezifiziert und der Parser kann anhand dieser Parameter ein identisches Objekt erzeugen. Alle Objekte werden mit einer zusätzlichen Funktion `writeObject()` ausgestattet (Abbildung 6-1), die die Konstruktionsparameter des Objektes, deren Anzahl und eine Transformationsmatrix zurückgibt. Zusätzlich wird ein Typidentifikationscode (DistribID) als Pseudoparameter für die Auswahl des zugehörigen Konstruktors mitgeliefert.

```
virtual void writeObject(
      t_ObjectDefinition::pStackContainer& param,
      unsigned& numParam,
      t_4x3Matrix& trfMatrix);

   t_Id globalObjID();
   void globalObjID(t_Id _globalObjID);

private:
   t_Id v_globalObjID;
```

Abbildung 6-1: Erweiterung der MRT-Basisklassen

Die Parameter bestehen aus einfachen Typen und kleinen Objekten. Mit kleinen Objekten sind `t_Color`, `t_3DVector`, `t_2DVector` oder `t_4x3Matrix` gemeint. Da mit CORBA nur Typen übertragen werden können, die in OMG IDL spezifiziert wurden, müssen sie in entsprechende Structs konvertiert bzw. kopiert werden. Da die Anzahl der Parameter von Objekt zu Objekt schwankt werden, alle Parameter in einer CORBA `sequence <CORBA_Any>` gespeichert. Das bedeutet alle Typen werden mit überladenen Funktionen `pack()` und `unpack()` direkt in `CORBA_Any` konvertiert.

Einige Parameter sind Pointer auf andere Objekte. So hat z.B. jedes Oberflächenobjekt (`t_SurfaceObject`) einen Shader als Parameter. Ein direktes Verpacken und Übertragen dieses Pointers ist jedoch nicht sinnvoll, da jeder Client Objekte in anderen Speicherbereichen unterbringen kann. Somit zeigt ein auf dem Zielrechner ausgepackter Pointer mit Sicherheit in einen Speicherbereich, der nicht das gewünschte Zielobjekt enthält. Zudem kann es vorkommen, daß Pointer übertragen werden, bevor das referenzierte Objekt versandt wurde. Statt eines Pointers muß eine Art Referenz auf das Objekt verschickt werden, anhand derer das Zielsystem das zugehörige Objekt finden und den entsprechenden Pointer einsetzen kann. Die Lösung dieses Problems ist die Zuweisung einer für eine gesamte Konferenz eindeutigen Objektidentifikationsnummer (OID). Die OID dient als Schlüssel für ein Objekt und seinen entsprechenden Datensatz innerhalb der Datenhaltung. Anhand der OID kann dann der Pointer für das Zielobjekt gefunden und als Parameter eingesetzt werden. Damit jede OID nur einmalig vergeben wird, muß

sie vom Server angefordert werden. Dies wird um den Kommunikationsaufwand zu verringern in größeren Blöcken von 10, 50 oder 100 OIDs durchgeführt. Im Konstruktor eines Objektes wird die OID auf 0 gesetzt. So erkennt die Datenhaltung bisher nicht registrierte Objekte, fordert eine neue OID an und trägt sie ein. Die OID ist der zweite und letzte Eingriff in die MRT-Basisklassen (Abbildung 6-1). Betroffen sind die Basisklassen `t_Light`, `t_Shader` und `t_Object` (bzw. `t_Scene` und `t_SurfaceObject`) und alle von ihnen abgeleiteten Klassen.

Diese Werte werden in der Tabelle (Objektliste) als Datensatz eingetragen. So kann jedes Objekt einer Szenenhierarchie, wie in Abbildung 6-2 dargestellt, in die Tabelle übernommen und zu anderen Teilnehmern übertragen werden. Die Liste wird als Binärbaum implementiert, um über den Schlüssel schnelle logarithmische Zugriffszeiten auf einzelne Datensätze zu ermöglichen.

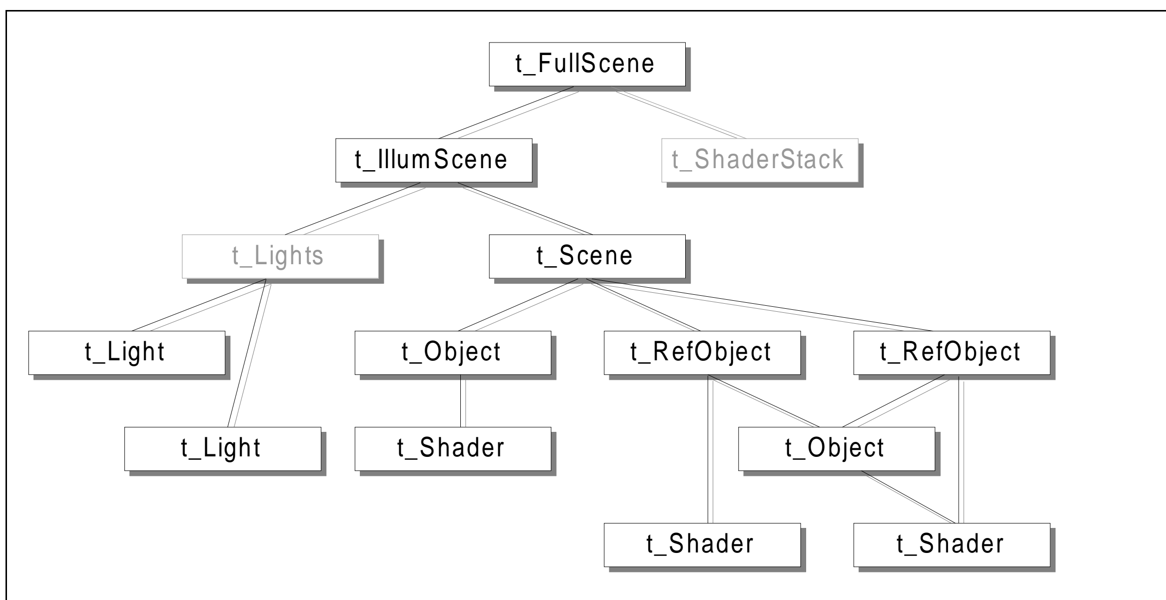


Abbildung 6-2: Die wichtigsten Klassen der Szenenhierarchie des MRT

Das dargestellte Vorgehen hat den Nebeneffekt, daß die vorher unveränderbaren Objekte von der Datenhaltung geändert werden können, indem die entsprechenden Konstruktionsparameter modifiziert werden. Der große Nachteil ist, daß das vorhandene und zu ändernde Objekt gelöscht und ein neues über den Parser generiert werden muß. Zusätzlich müssen alle Pointer auf das alte Objekt durch den neuen Pointer ersetzt werden. Für diese Ausnahmesituation werden in der Objektliste zusätzlich alle OIDs der jeweiligen Containerobjekte gespeichert, um sie schnell finden und aktualisieren zu können. Dadurch müssen die Containerobjekte ebenfalls gelöscht und neu erzeugt werden. Diese Kettenreaktion könnte sehr aufwendig werden, wären nicht die zuvor erwähnten Ausnahmen. Jedes Objekt erlaubt das Setzen seines Shaders mit `shader()`. Szenen können mit `init()` modifiziert werden. Referenzobjekte (`t_RefObject`) lassen die Modifizierung aller ihrer Parameter zu. Dadurch wird die Kette der rekursiven Aktualisierungen durchbrochen. Eine konsequente Verwendung und Modifizierung der Referenzobjekte reduziert sie auf ein Minimum.

6.2 Versionsmanagement

In Kapitel 3.3.3 wurde der 2-Versionen-Ansatz erläutert. Er ermöglicht die Trennung von Lese- und Schreibzugriffen erfordert jedoch eine Synchronisation der Versionen. Um nicht jedesmal alle Objekte aktualisieren zu müssen, werden Änderungsmarkierungen (change_flags) gesetzt. Diese Flags werden verwendet:

- `CHANGE_NONE` keine Änderungen vorhanden
- `CHANGE_DELETE` Für alle Containerobjekte wird das CHANGE_UPDATE -Flag gesetzt und der Datensatz aus der Objektliste entfernt.
- `CHANGE_NEW` Basierend auf der Parameterliste werden die CORBA-Konstrukte wieder in MRT-Typen konvertiert und die Konstruktor-Parameter mit der DistribID an den Parser übergeben. Vom Parser erhält die Objektliste den Pointer auf das neue Element und speichert ihn in der Leseversion ab. Für alle Containerobjekte wird das CHANGE_UPDATE -Flag gesetzt.
- `CHANGE_UPDATE` Element geändert und muß rekonstruiert werden. Soweit dies möglich ist werden Objekte nicht neu erzeugt, sondern direkt geändert. Wie bereits am Ende von Kapitel 6.1 erwähnt, können Szenen mit `init()` neu mit Elementen bevölkert werden und Referenzobjekte lassen sich vollständig rekonfigurieren.
- `CHANGE_TRANSF` Element modifiziert und kann direkt geändert werden. Anhand des in der Leseversion gespeicherten Pointers werden die entsprechenden Methoden (z.B. `shader()`) des Elementes aufgerufen, um die Änderungen durchzuführen.

Die Synchronisierung überprüft immer erst alle `t_Light` Elemente, dann die `t_Shader`, gefolgt von den `t_SurfaceObject`. Zum Schluß werden die `t_Scene` Elemente aktualisiert. Wurden dabei neue change_flags gesetzt, wiederholt sich der Vorgang.

Bei der Konstruktion eines Elementes A und der Zusammenstellung der Konstruktor-Parameter kann es vorkommen, daß ein für die Konstruktion erforderliches Element B (z.B. ein Shader) entweder noch nicht oder nicht mehr vorhanden ist. Es kann also kein Pointer auf dieses Phantomelement verwendet werden. Dennoch soll nicht auf Element A verzichtet werden. Deshalb werden in solchen Fällen Pointer auf Dummyelemente verwendet. Für jeden der Basistypen `t_Light`, `t_Shader` und `t_SurfaceObject` wird ein solches Element bereitgehalten, das im Falle einer fehlenden Referenz verwendet wird.

Zusätzlich wird eine Änderungsliste geführt, die die OIDs der betroffenen Objekte speichert. Für jedes Element der Änderungsliste muß mit maximal logarithmischem Aufwand in der Objektliste der entsprechende Datensatz gesucht werden. Solange die folgende Formel zutrifft, ist die Verwendung der Änderungsliste schneller, als ein kompletter sequentieller Durchlauf der Objektliste:

$$\text{Länge (ObjectList)} > \log_2(\text{ Länge (ObjectList) }) \bullet \text{ Länge (ChangeList)}$$

6.3 Transaktionen

Transaktionen erleichtern die Programmierung der Anwendung erheblich, wie in Kapitel 3.2 ff. erwähnt. Im folgenden wird die Umsetzung der Konzepte in das zu implementierende Transaktionsmanagement beschrieben.

Transaktionen werden innerhalb der Datenhaltungsschicht für alle Operationen angewendet, da einige Operationen rekursiv weitere anstoßen können (Kapitel 6.1 Kettenreaktion). Sie können aber auch aus der Anwendung heraus geöffnet, beendet und abgebrochen werden. Generell werden alle Operationen gespeichert und folgendermaßen sortiert:

1. Alle Schreibsperrenanforderungen werden direkt durchgeführt, um eine Bestätigung oder Ablehnung an die Anwendung weitergeben zu können.
2. Alle Änderungsoperationen in unveränderter Reihenfolge.
3. Freigabe aller in dieser Transaktion erhaltenen Schreibsperren.

Beim Beenden der Transaktion werden alle Änderungen in einer Nachricht an den Server übertragen und von diesem in die Szene übernommen, um anschließend alle Teilnehmer über diese Änderungen zu informieren (Primary-Copy-Verfahren Kapitel 3.5.3). Zum Schluß werden die Sperren wieder freigegeben.

Durch das Fehlschlagen einer Sperranforderung wird automatisch die Transaktion ungültig. Die Anwendung kann zwar weiter Operationen einfügen, jedoch werden beim Beenden der Transaktion alle Operationen gelöscht und alle Sperren werden wieder freigegeben. Das gleiche geschieht, wenn die Anwendung die Transaktion von sich aus abbricht.

Da einige Änderungen eine Modifizierung der Szenenhierarchie durch die Anwendung erfordern (z.B. das Einfügen neuer Objekte), wird bei einer fehlgeschlagenen Transaktion die Szenenhierarchie wieder mit der in der Objektliste gespeicherten Version synchronisiert bzw. rekonstruiert.

6.4 Netzwerk

Mit der Verwendung von CORBA bleiben folgende inhärente Probleme von verteilten Anwendungen (Kapitel 2.4) übrig:

- Sicherheitsprobleme
- Verzögerungen bei der Übertragung
- Netzwerkausfälle und Datenverluste

Sicherheitsprobleme werden im Rahmen dieser Diplomarbeit nicht näher betrachtet. Manipulationen der Datenpakete lassen sich aber durch die Verwendung von Speziallösungen beheben, z.B. bietet ORBacus eine SSL-verschlüsselte Übertragungsvariante als Plug-In an, die einfach bei der Initialisierung des ORBs eingebunden und aktiviert werden muß. Unauthorisierte Aufrufe der CORBA-Interfaces lassen sich durch zusätzliche Authorisationscodes überprüfen. Jeder Funktion würde dann der Authorisationscode als zusätzlicher Parameter hinzugefügt, der nur dem Client selber und dem Server bekannt wäre. Weitergehende Strategien sind in [GW90] beschrieben.

Verzögerungen bei der Übertragung über das Internet sind unvermeidbar. Sie führen bei den Clienten zu längeren Blockierungsphasen, dadurch daß die Aufrufe der Methoden im synchronen Betrieb auf die Empfangsbestätigung oder einen Rückgabewert warten müssen. Dieses Problem läßt sich durch die Verwendung semisynchroner Aufrufe mit dem DII (Kapitel 4.5) umgehen. Für viele Methoden läßt sich diese Lösung anwenden. Für einige Funktionen sind die Rückgabeparameter jedoch erforderlich, um mit der Arbeit fortfahren zu können. Hier kann man auf blockierende synchrone Aufrufe nicht verzichten. Ein Beispiel sind die Methoden zum Setzen von Schreibsperren, die mit einer positiven oder negativen Antwort zurückkehren müssen.

Das dritte Problem sind **Datenverluste** oder temporäre Verbindungsunterbrechungen, die zu Timeouts oder anderen Netzwerkfehlern führen. Diese sind insbesondere dann hinderlich, wenn eine Nachricht an z.B. 20 Teilnehmer versandt werden sollte, und einer von ihnen temporär nicht erreichbar ist. Es stellt sich die Frage, was geschehen soll, wenn weitere Nachrichten eintreffen, die an alle Teilnehmer versendet werden müssen. Für den einen Teilnehmer müßte ein Protokoll der fehlerhaft versandten Nachrichten erstellt werden. Dieses Protokoll wird dann zu leeren versucht, bevor neue Nachrichten an diesen Teilnehmer gehen. Solange die Verbindung unterbrochen bleibt wird das Protokoll erweitert und bei einer Wiederherstellung der Verbindung werden die Nachrichten in der ursprünglichen Reihenfolge zugestellt. Je nach Länge dieser Liste lassen sich eventuell Daten zusammenfassen oder veraltete Aktualisierungen überspringen. In Kapitel 7.4.6 wird auf die Implementierung der Protokollierung im Rahmen des Broadcasters detailliert eingegangen.

Um die Vorzüge von CORBA ausnutzen zu können, statt es zu einem reinen Transportmechanismus zu degradieren, können folgende Elemente der vorliegenden MRT-VR Version entfallen: Dispatcher (der BOA sorgt für den Aufruf der richtigen Methode), Marshaling (Stub und Skeleton "verpacken" die Aufrufparameter in einem plattformübergreifenden Format), Netzwerkprotokollverwaltung (der ORB übernimmt diese Aufgabe).

6.5 Schichtenmodell

Der MRT-VR besteht in der vorliegenden Version aus den 3 Schichten Visualisierung, Datenhaltung und Transport [HA97b]. Die Visualisierung und somit die interne Struktur des MRT bleiben erhalten.

Das erweiterte Schichtenmodell zeigt Abbildung 6-3. Die Visualisierungsschicht wurde hier durch eine allgemeinere Anwendungsschicht ersetzt, da z.B. ein Automat oder der Konferenzserver keine Visualisierung der Szenendaten benötigt. Die Trennung zwischen Transport- und Datenhaltungsschicht ist nicht mehr eindeutig zu ziehen, da der Transport durch CORBA vollständig gekapselt wird. Deshalb ersetzt der ORB die Position der Transportschicht. Das Einbinden anderer Transportprotokolle und Mechanismen läßt sich in zusätzlichen Threads innerhalb des Broadcasters (für den Versand) und des Messagehandlers (für den Empfang) unterbringen. Somit braucht der Rest der Struktur nicht modifiziert zu werden. Man behält jedoch auf jeden Fall die Gesamtverwaltung durch CORBA bei.

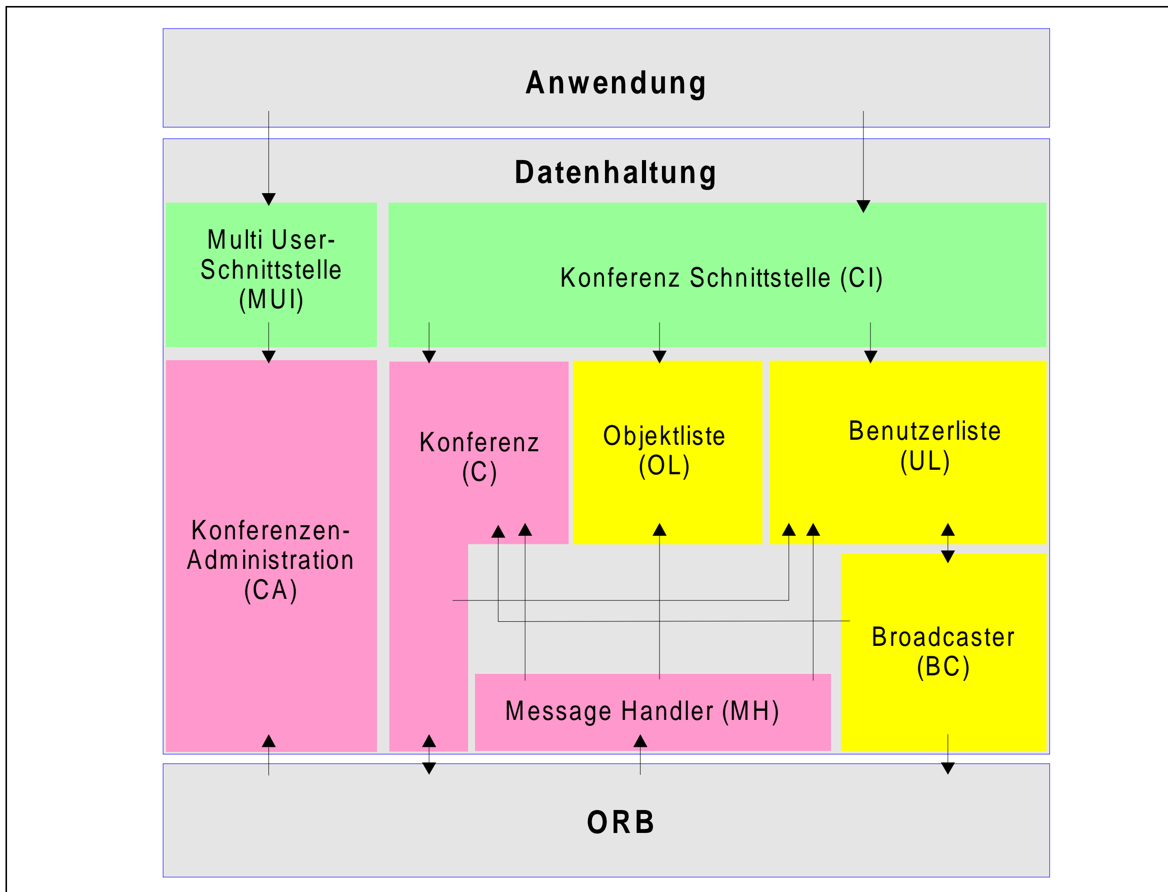


Abbildung 6-3: Schichtenmodell der MUI-MRT Datenhaltung

Die Datenhaltungsschicht läßt sich nochmals in 3 Bereiche einteilen. Der erste (grün hinterlegt) ist das nach außen hin sichtbare Interface der Datenhaltungsschicht (Kapitel 7.2). Es ist auf die zwei Objekte Multi User Interface (MUI) und Konferenz-Interface (CI) verteilt.

Der zweite Teil besteht aus den rot hinterlegten Objekte, welche als CORBA-Objekte implementiert wurden. Diese haben eine über OMG IDL spezifizierte Schnittstelle. Sie übernehmen die Kommunikation und die globale Konferenzverwaltung.

Die dritte Gruppe bilden die gelb hinterlegten "normalen" C++-Klassen, welche die Daten verwalten (Objekte und Teilnehmer) bzw. die Verteilung der Daten und Nachrichten (BC) übernehmen.

Auf eine detailliertere Beschreibung der Objekte und der zwischen ihnen ausgetauschten Ereignisse und Nachrichten wird in Kapitel 7.4 eingegangen.

7 Implementierung

In diesem Kapitel wird das MRT-VR durch die Realisierung der in den vorigen Kapiteln entwickelten Ansätze zum MUI-MRT erweitert. Die Datenhaltungsschicht ist auf den Plattformen Windows 95/98, Windows NT 4.0 implementiert und getestet worden. Die Visualisierung wurde aufbauend auf dem MRTMFC mit Visual C++ 5.0 für Testzwecke erweitert. Für SUN und SGI wurden im Rahmen dieser Arbeit nur Konsolenapplikationen implementiert, die die Lauffähigkeit auf diesen Plattformen überprüfen sollten. Eine Kompilierung der Datenhaltungsschicht in Konsolenapplikationen nach geringfügigen Anpassungen, als Server, Automat oder Monitor ist somit gegeben.

7.1 MRT-VR

Basis für die Implementierung der verteilten Datenhaltung ist das Minimal Rendering Tool (MRT) [Fel96] mit den Erweiterungen zur interaktiven Betrachtung virtueller Welten (MRT-VR) [HA97a]. Es bietet mittels objektorientiertem Design, strukturierte abstrakte Konzepte zur Visualisierung dreidimensionaler Szenen an. Dazu stellt das in C++ programmierte MRT eine Hierarchie von Klassen und Methoden zur Verfügung, um u.a. fotorealistische Bilder mittels Ray Tracing oder durch Hierarchical Radiosity zu berechnen oder interaktive virtuelle Welten mit schnellen teilweise hardwareunterstützten Darstellungen (Wireframe, Flat-, Gouroud- oder Phong-Shading) zu erzeugen.

Aus einem MSD- oder VRML-Skript wird eine Szenenrepräsentation durch eine Hierarchie von Klassen erzeugt, auf der die verschiedenen Visualisierungen und Berechnungen durchgeführt werden. Diese Szenendarstellung wird konvertiert und übertragen (Kapitel 2.2 und 6.1). Alle Objekte mußten um eine gewisse Grundfunktionalität erweitert werden, die eine Übertragung und identische Rekonstruktion ermöglichen. Um die Konstruktionsparameter zu erhalten, wurden alle für den Versand bestimmten Objekte und deren unmittelbare Basisklassen mit der Funktion `writeObject()` ausgestattet und erhielten eine eindeutige Objekt ID, die OID.

7.2 Interfacebeschreibung

Das nach außen hin sichtbare Interface wird durch die zwei Objekte `t_MUI` und `t_ConfInterface` gebildet.

- `t_MUI` bildet das generelle Administrationsinterface, welches der Erzeugung von Konferenzobjekten dient, die dann für die verschiedenen Teilnahmemöglichkeiten konfiguriert werden können. Es ist möglich mehrere Konferenzen parallel zu führen.
- `t_ConfInterface` bietet die Steuerung einer Konferenz mit allen Funktionen in einem Objekt vereint an. Die Aufrufe werden an die zuständigen Objekte weitergeleitet. Somit bleibt vor der Anwendung die interne Struktur und Datenverteilung verborgen. (Verteilungstransparenz)

Im folgenden werden diese Schnittstellen ausführlich beschrieben.

7.2.1 Administrationsinterface (`t_MUI`)

Das Administrationsinterface kapselt die gesamten Initialisierungsvorgänge für CORBA und startet den Konferenzadministrator und die Threads, welche im Hintergrund für die Datenübertragung zuständig sind. Abbildung 7-1 zeigt die Klassendefinition von `t_MUI`.

Mit dem Konstuktor kann über den Parameter `_remoteServer` bestimmt werden, ob es anderen Teilnehmern erlaubt ist über das Netz in dieser Anwendung eine Konferenz als Server zu starten. Auf diese Weise läßt sich ein dedizierter Konferenzserver mit minimalem Programmieraufwand erstellen (Abbildung 7-2).

```
class t_MUI
{
	t_MUI(bool _remoteServer = false);
	~t_MUI();

	t_ConfInterface_ptr createConference();
	void destroyConference(t_ConfInterface_ptr _conf);
	t_StringSeq* getConfNames(const char* _iiop = NULL);

	const char* loadIIOP(const char* _filename);
	void dumpIIOP(const char* _filename);
	const char* getIIOP();
	const char* hostName();
	unsigned int localPort();
};
```

Abbildung 7-1: Definition der Klasse `t_MUI`

Das Erzeugen und Auflösen von Konferenzobjekten geschieht über die Funktionen `createConference()` und `destroyConference()`. Es wird jeweils das Konferenzinterface erzeugt bzw. aufgelöst.

Über `getConfNames()` können die Namen, der auf einem über seine IIOP-Adresse spezifizierten Konferenzserver laufenden Konferenzen, abgefragt werden. Wird keine IIOP-Adresse angegeben, erhält man die Namen der lokal laufenden Konferenzen.

Die übrigen Funktionen erlauben ein einfaches Handling der für eine Kontaktaufnahme erforderlichen Parameter. So kann die IIOP des lokalen Systems erfragt werden, um sie beispielsweise via eMail an andere Teilnehmer zu versenden, die zu ihrer Konferenz eingeladen sind.

```
int main (int argc, char* argv[])
{
	t_MUI_ptr mui = new t_MUI(true);
	cerr << mui -> getIIOP() << endl;
	char c;
	cin >> c;
	free(mui);
	return 0;
}
```

Abbildung 7-2: Minimaler Code eines dedizierten Konferenzservers

7.2.2 Konferenzinterface (`t_ConfInterface`)

Eine Konferenz wird gesteuert über das Konferenzinterface. Eine vollständige Übersicht aller Operationen zeigt Abbildung 0-1 im Anhang. Hier werden die Funktionen, nach den in Kapitel 2.8 erstellten Ereignissen gruppiert, kurz beschrieben.

Starten einer Konferenz - Konferenzschnittstellen werden über die Administrationsschnittstelle erzeugt und können über die Operationen `init()`, `initRemote()` und `login()` als Server oder Client an Konferenzen teilnehmen.

Es gibt zwei Arten eine Konferenz zu starten bzw. zu initiieren. Die erste ist die Initiierung der Konferenzschnittstelle mit `init()` als Server. Andere Teilnehmer melden sich dann mittels `login()` (oder `loginIIOP()`) direkt bei diesem Konferenzserver an. Dies ist eine geeignete Vorgehensweise bei kleinen Konferenzen (bis 10 Personen), die ohne einen eigenständigen Server auskommen, da der Kommunikationsaufwand noch "nebenher" realisiert werden kann, ohne die Visualisierung des Teilnehmers, der die Serverfunktion übernommen hat spürbar zu bremsen. Bei größeren Konferenzen macht es sich bemerkbar, daß die Serveradministration der Visualisierung Rechenzeit wegnimmt und umgekehrt. Durch die Verwendung von Threads ergeben sich allerdings nur Einbußen bei der Geschwindigkeit, blockieren wird die Datenhaltung die Visualisierung nicht.

```
t_ConfInterface conf = mui->createConference();
if (!conf->initRemote("Meine Konferenz",
                      "iiop://terra.informatik.uni-bonn.de:1034/",
                      _fullScene, _baseScene, _avatarScene,
                      _avatar, R_FULL)
{
   cerr << "ERROR: " << conf->getLastError() << endl;
}
```

Abbildung 7-3: Initiierung einer Konferenz auf einem dedizierten Server

Für umfangreichere Konferenzen ist die Initiierung einer neuen Konferenz auf einem dedizierten Konferenzserver mittels `initRemote()` zu bevorzugen. Der dedizierte Server kann sich einzig um die Konferenzadministration kümmern, und wird nicht durch z.B. `updateWorld()` zu einer Synchronisation der Threads gezwungen. Mit `initRemote()` wird die Erzeugung einer neuen Konferenz auf dem dedizierten Server angestoßen und die Konferenzschnittstelle wird als teilnehmender Client mit Administratorrechten angemeldet. Abbildung 7-3 zeigt die erforderlichen Codezeilen, um eine Konferenz zu starten. Neben der Angabe des Konferenznamens und der IIOP-Adresse des dedizierten Servers können noch folgende Parameter angegeben werden:

`t_FullScenePtr`: Es werden die globalen Parameter der Szenenbeschreibung aus der übergebenen `t_FullScene` übernommen (z.B. die Hintergrundfarbe und die Lichtquellen) die Szenenbeschreibung wird ignoriert, um der Anwendung mehr Flexibilität bei der Wahl der von der Datenhaltungsschicht verwalteten Szene zu geben. Es wird ausgegangen von einer Anordnung der Elemente in der Art, wie sie Abbildung 7-4 zeigt. BasisSzene und AvatarSzene können sich aber auch an anderen Stellen der Szenenhierarchie befinden.

BasisSzene: Der Pointer auf die Wurzel der Szenenhierarchie, die der Datenhaltung übergeben werden soll. Die Basisszene und alle über sie rekursiv erreichbaren Objekte werden registriert.

AvatarSzene: Der Pointer auf eine Szene, in die die Avatare eingetragen werden. Die Szene wird zu Beginn der Konferenz geleert.

Avatar: Hier kann direkt eine über ein Referenzobjekt referenzierte AvatarSzene angegeben werden.

Zugriffsrecht: Dieser Parameter bestimmt das Zugriffsrecht aller sich neu anmeldenden Teilnehmer.

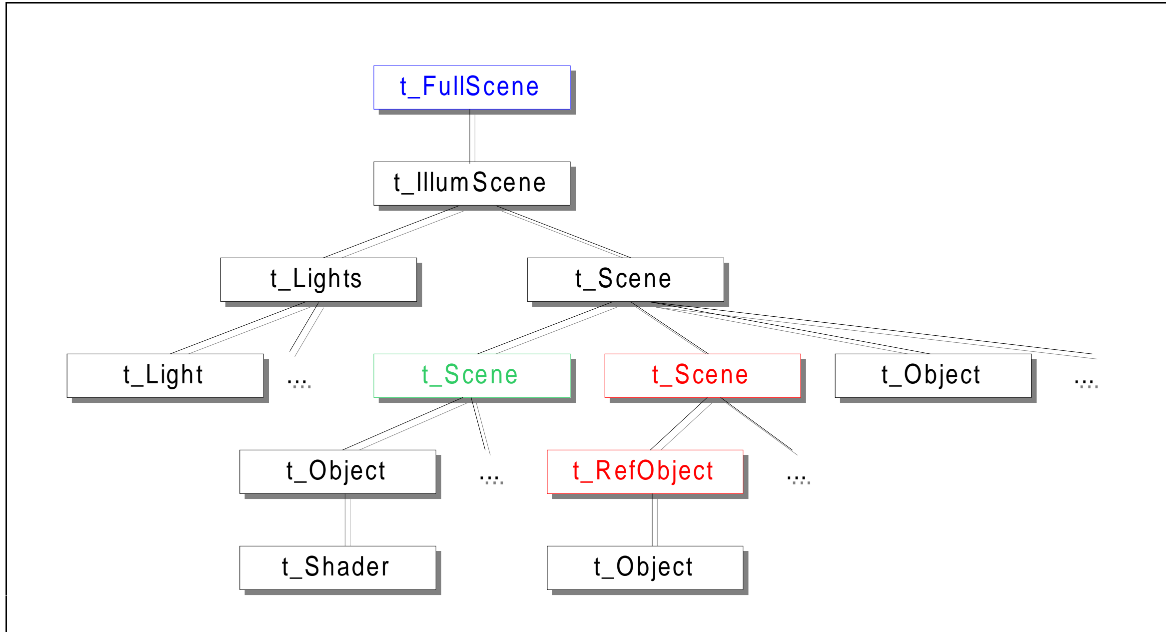


Abbildung 7-4: MUI-MRT erwartet diese Szenenhierarchie

Das **Beenden einer Konferenz** durch den Initiator der Konferenz hat je nach der Art des Konferenzstarts andere Folgen. Läuft die Konferenz auf dem lokalen Client des Teilnehmers, so wird die Konferenz für alle Teilnehmer mit dessen Ausstieg beendet. Alternativ, aber in dieser Arbeit nicht behandelt, wäre die Migration des Serverdienstes auf den Rechner eines anderen Teilnehmers möglich. Wurde die Konferenz auf einem dedizierten Konferenzserver initiiert, läuft die Konferenz weiter, bis sich der letzte Teilnehmer abgemeldet hat.

Anmeldung von Teilnehmern - Nach der Initialisierung des MUI und der Erzeugung einer Konferenzschnittstelle (Kapitel 7.2.1) sind die wichtigsten Informationen, für eine erfolgreiche Anmeldung bei einer laufenden Konferenz, die IIOP-Adresse (oder ersatzweise der Hostname und die Portadresse) des Servers oder eines beliebigen Teilnehmers der Konferenz, sowie der Konferenzname, für den Fall, daß mehrere Konferenzen parallel auf einem Server geführt werden. Über die Funktion `loginIIOP()` (oder `login()` mit Host und Port) werden die Verbindung hergestellt und die gesamten Szenendaten sowie die Teilnehmerdaten übertragen. Anschließend kehrt die Funktion mit dem Rückgabeparameter `true` zurück. Sollte ein Fehler aufgetreten sein, kann über die Funktion `getLastError()` der Grund in einem String abgefragt werden. (Dies gilt im übrigen für die meisten Funktionen des Interfaces)

Zusätzlich können die Parameter, `FullScene`, `BasisScene` und `AvatarScene` wie beim Start einer Konferenz mit `init()` oder `initRemote()` angegeben werden. Bei einem Login werden die Inhalte dieser Szenen jedoch gelöscht und durch die Daten der Konferenzszene ersetzt. Der letzte Parameter kann einen Avatar übergeben.

Abschließend aktualisiert `updateWorld()` die lokale Szene mit den Objektdaten aus der Objektliste (s.u. Szenendaten aktualisieren).

Auf der Serverseite werden alle Teilnehmer über den neuen Teilnehmer informiert, so daß diese ihn in ihre Teilnehmerliste und in die Kamera-Versandroutine einsetzen können.

Anmeldung von Automaten - Die Anmeldung eines Automaten läuft genauso ab, wie die Anmeldung eines Teilnehmers. Der Unterschied liegt nur darin, daß der Automat die meisten Dienste abschalten kann und keine Synchronisation der Szenendaten durchführen muß. Über die Funktion `getObjByName()` kann ein benanntes Objekt angefordert werden. Die wohl häufiger verwendete Alternative ist, daß der Automat die von ihm manipulierten Objekte selbst erzeugt, modifiziert und wieder löscht.

Abbildung 7-5 zeigt am Beispiel eines Automaten, der eine Kugel abwechselnd rot und blau färbt, wie kompakt die Programmierung eines Automaten ist. Nach der Erzeugung der Konferenz wird zunächst versucht den `t_SurfaceObjectPtr` auf eine Kugel über einen Namen zu erhalten. Ist kein Objekt dieses Namens bekannt, wird ein eigenes Objekt erzeugt und registriert. Die verwendeten Funktionen werden im weiteren Verlauf dieses Kapitels erläutert.

```
t_ConfInterface conf = mui->createConference();
if (!conf->loginIIOP("Konferenzname",
                     "iiop://terra.informatik.uni-bonn.de:1034/")
   cerr << "ERROR: " << conf->getLastError() << endl;

t_ShaderPtr blau  = // irgendwie einen blauen Shader definieren
t_ShaderPtr rot   = // irgendwie einen roten Shader definieren
t_SurfaceObjectPtr kugel =
   t_SurfaceObject::cast(conf->getObjByName("Kugelname"));
if (kugel ==(t_SurfaceObjectPtr)NULL)
{
   kugel = new t_Sphere(t_3DVector(0,0,0), 5, blau);
   conf->registerObject(kugel); // Kugel registrieren
}

conf->setFlags(CONF_NONE);      // keine Dienste nutzen
conf->lockObject(kugel);        // Schreibsperre anfordern
conf->updatesImportant(false);  // Änderungen "unwichtig"

while (!ende) {
   conf->exchangeShader(kugel, rot);
   Sleep(100);
   conf->exchangeShader(kugel, blau);
   Sleep(100);
}

conf->deleteObject(kugel);
```

Abbildung 7-5: Code eines kleinen Automaten

Abmelden - Mit `logout()` verläßt ein Teilnehmer die Konferenz. Alle von ihm gehaltenen Sperren und sein Avatar werden gelöscht. Die übrigen Teilnehmer werden über das Ausscheiden des Teilnehmers informiert und die Teilnehmerliste wird aktualisiert.

Sperren von Elementen - Bevor Elemente geändert oder gelöscht werden können, muß eine Schreibsperre gesetzt werden (Kapitel 6.3). Der Aufruf der Funktion `lockElement()` blockiert bis zum Empfang der Bestätigung vom Server, ob die Sperre gesetzt werden konnte oder nicht. Nachdem Änderungen durchgeführt wurden, müssen gesperrte Objekte mit `unlockElement()` wieder freigegeben werden. Beim Löschen eines Objektes wird automatisch auch die Sperre auf die jetzt wieder freigewordene OID gelöscht. Mit dem Aussteigen eines Teilnehmers aus der Konferenz werden automatisch alle seine Sperren wieder freigegeben.

Markieren von Elementen - Ein Objekt mit `markElement()` zu markieren bedeutet es gleichzeitig auch zu sperren. Eine Markierung wird visualisiert durch den Austausch des Shaders mit einem vom Teilnehmer über die Funktion `setMarkShader()` vorgegebenen Shader, oder, falls dieser keinen gewählt hat, einem roten Standardmarkierungsshader. Der Markierungsshader wird nicht übertragen, so daß jeder Teilnehmer für alle Markierungen seinen eigenen Shader verwenden kann.

Markierungen können nur auf `t_SurfaceObject` gesetzt werden und sie werden nicht rekursiv weitergegeben an eventuelle untergeordnete Elemente. Die Markierung wird entweder durch ihre Löschung mit `unmarkElement()` oder durch Lösen der Sperre auf das Objekt wieder entfernt. Verläßt der Teilnehmer die Konferenz, werden automatisch alle seine Markierungen und Sperren wieder freigegeben.

Einfügen und Löschen von Elementen - Um ein Objekt in eine Szene einzufügen muß es bei der Datenhaltung registriert werden. Dies geschieht mit den Funktionen `registerObject()`, `registerScene()`, `registerShader()` und `registerLight()`. Für die ersten 3 Funktionen ist neben dem Pointer auf das zu registrierende Objekt auch das Containerobjekt anzugeben, welches zuvor schon registriert worden sein muß. Soll ein Objekt in die BasisSzene eingefügt werden, kann der Container entfallen. Alle Objekte, auf die über Parameter verwiesen wird, und die noch nicht registriert sind, werden rekursiv mitregistriert. Um eine SubSzene zu registrieren, reicht also der Aufruf von `registerScene()` für die Wurzelszene. Abbildung 7-6 verdeutlicht den Ablauf der Registrierung. Die roten Pfeile zeigen den Verlauf auf, den die Rekursion nimmt. Die blauen Zahlen geben die Reihenfolge wieder, in der die Objekte in die Objektliste übertragen werden. Bei der Registrierung des t_RefObjekt in Schritt 9 sind alle anhängenden Objekte bereits registriert, so daß die Rekursion direkt wieder zurückkehren kann.

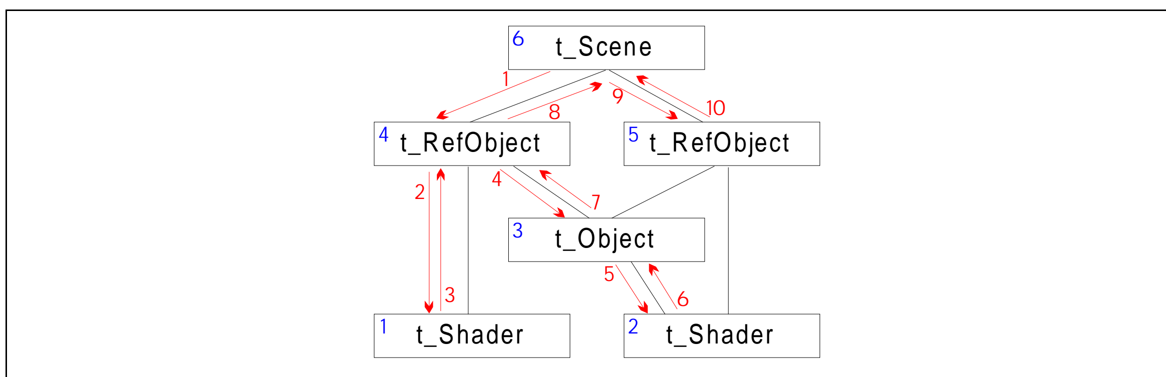


Abbildung 7-6: Rekursiver Registrierungsverlauf eine neuen Szene

Das Löschen von Objekten wird über die vier entsprechenden `delete`-Funktionen durchgeführt. Es wird aber nur das angegebene Objekt gelöscht und nicht alle rekursiv erreichbaren Objekte.

Ändern von Elementen - Die Änderungsoperation `transformObject()` modifiziert die Transformationsmatrix T eines beliebigen `t_SurfaceObject` in der Szene, das bereits registriert und gesperrt ist. Die übergebene Transformationsmatrix S kann entweder relativ oder absolut angewendet werden. Bei der relativen Variante werden T und S multipliziert, bei der absoluten Variante wird T durch S ersetzt.

Analog funktioniert die Methode `translateObjekt()`, welche nur die Translationskomponente der Transformationsmatrix, repräsentiert durch einen `t_3DVector`, benötigt.

`exchangeShader()` wechselt den Shader eines übergeben Objektes durch den übergeben Shader aus. Der Shader wird automatisch registriert, sollte dies noch nicht geschehen sein.

Einige Änderungen werden regelmäßig und in kurzen Abständen vorgenommen, z.B. das Setzen der Uhrzeiger, Bewegen von Regentropfen, etc. . Dies wird meist von Automaten durchgeführt. Bei diesen Änderungen ist es nicht wesentlich, daß alle Änderungen ankommen. In Kapitel 7.4.6 wird dies näher erläutert. Über die Methode `updatesImportant()` kann für die zuvor erwähnten Änderungsoperationen bestimmt werden, ob sie wichtig sind, oder nicht.

Es ist auch möglich Objekte, die direkt in der Szenenhierarchie von der Anwendung geändert wurden, mit den register-Funktionen neu zu registrieren und die Änderung dadurch zu publizieren. Es werden jedoch nur das angegebene Objekt und alle neuen anhängenden Objekte registriert. Für die Implementierung eines Editorsystems ist diese Art der Änderung erforderlich, um objektspezifische Änderungen, wie z.B. die direkte Manipulation der Reflektionseigenschaft eines Shaders, übertragen zu können, ohne die Datenhaltungsschicht modifizieren oder erweitern zu müssen.

Transaktionen werden über die Funktionen `openTransaction()` und `closeTransaction()` gestartet und beendet. Alles was zwischen diesen Aufrufen an der Szene über entsprechende Funktionen geändert wird, ist in einer Transaktion gebündelt. Das heißt alle Operationen werden auch in einem Paket übertragen und bei den Empfängern in einem Vorgang aktualisiert. Die ACID-Eigenschaften bleiben erhalten. Die folgenden Operationen können in einer Transaktion eingeschlossen werden:

- `open-` und `closeTransaction()`
- alle register Funktionen (außer `registerAvatar()`)
- alle deleteFunktionen
- alle lock und mark Funktionen
- `translateObject()`, und `transformObject()`
- `exchangeShader()`

Eine Sonderrolle spielen die Operationen `lockObject()` und `markObject()`. Diese werden sofort durchgeführt und blockieren bis eine Bestätigung oder Ablehnung vom Server erhalten wurde. Kann eine Sperre nicht gesetzt werden, muß die Transaktion

mittels `abortTransaction()` abgebrochen werden. Alle seit dem Start der Transaktion vorgenommenen Änderungsoperationen werden verworfen (Rollback) und alle gesetzten Sperren werden wieder freigegeben, intern wird dazu `reconstructWorld()` aufgerufen (s.u. Szenendaten aktualisieren). Die `unlock()` und `unmark()` Funktionen werden am Ende der Transaktion ausgeführt.

Es ist möglich Transaktionen zu schachteln. Erst beim Schließen der äußersten Transaktionshülle werden die Daten versandt und die Transaktion durchgeführt. Ein Transaktionsabbruch gilt unabhängig von der Schachtelungstiefe immer für die äußere Transaktionshülle und somit alle in ihr enthaltenen Subtransaktionen.

Kamerapositionen der Teilnehmer
Mit der Funktion `setCamera()` aktualisiert jeder Teilnehmer seine Kamera indem der Pointer auf die aktuell verwendete Kamera übergeben wird. Bevor diese weitergeleitet wird, überprüft die Datenhaltung, ob sich die Kamera verändert hat. Danach werden über den Kameraverteiler (Kapitel 7.4.7) die Änderung an alle Teilnehmer direkt weitergeleitet. Der Server ist nicht beteiligt.

Die Kameradaten der anderen Teilnehmer können mit `getCameraIDs()` und `getCamera()` abgefragt werden. `getCameraIDs()` liefert eine Liste aller UIDs, für die neue Kameradaten eingetroffen sind, seit dem letzten Aufruf. Mit `getCamera()` wird die Kamera eines Teilnehmers anhand seiner UID aus der Userliste angefordert.

Haben die Teilnehmer Avatare, werden diese mit den Kameradaten automatisch und ohne Mitwirkung der Anwendung aktualisiert, das heißt der Avatar wird auf die Position der Kamera gesetzt und entsprechend ausgerichtet.

Avatar anmelden - Jeder Teilnehmer kann eine Szene oder ein Objekt für seine Repräsentation in der virtuellen Welt einladen, die automatisch mit der Kameraposition verschoben wird. Beim Registrieren des Avatar werden automatisch die Elemente gesperrt, da ein Avatar nur der Besitzer selbst ändern kann. Mit der Funktion `registerAvatar()` wird ein Pointer auf ein Referenzobjekt übergeben, das den Avatar referenziert. Durch die erzwungene Verwendung eines `t_RefObject` können Avatare von der Datenhaltungsschicht leicht transformiert und ein- bzw. ausgeschaltet werden. Alle rekursiv erreichbaren Objekte werden als Elemente des Avatar mitregistriert. Es kann immer nur ein Avatar angemeldet werden. Dies ist keine Einschränkung, da die Anwendung bzw. der Teilnehmer eine Szene mit mehreren über Referenzobjekte referenzierte Avatare anmelden und über die Funktion `visible()` der Referenzobjekte den gewünschten Avatar an- oder abschalten kann. Beim Registrieren eines neuen Avatar wird automatisch der alte mit allen Objekten gelöscht.

Szenendaten aktualisieren - Die Datenhaltungsschicht kann bei eintreffenden Daten nicht direkt die Szenen- und Objektdaten modifizieren. Es könnte zu Zugriffskonflikten mit der Anwendung kommen. Beispielsweise könnte gerade die Darstellung einer Szene aktualisiert werden, während eines der Objekte gelöscht wird. Was nun im Falle eines gleichzeitigen Zugriffs passiert, ist nicht vorhersehbar und kann im äußersten Fall zum Absturz der Anwendung führen. Deshalb müssen die Zugriffe synchronisiert werden. Die Anwendung ruft immer dann, wenn sie Zeit dazu hat `updateWorld()` auf. Damit werden alle seit dem letzten Aufruf aufgelaufenen Änderungen auf die Szene übertragen. Es werden nur Änderungen aktualisiert, die von anderen Teilnehmern initiiert

wurden, da die Datenhaltungsschicht davon ausgeht, daß der Teilnehmer seine eigenen Änderungen direkt in der Szene umsetzt. Er muß jedoch zuvor eine Sperre anfordern und seine Änderungen registrieren. Sonst kann es vorkommen, daß Änderungen durch ein `updateWorld()` von einem anderen Anwender überschrieben werden.

Zusätzlich gibt es noch die Funktion `reconstructWorld()`, welche die gesamte lokale Szenenhierarchie löscht und aus den Daten der lokalen Objektliste neu generiert. Die Anwendung setzt so die Szene in einen definierten konsistenten Zustand zurück. Dies könnte erforderlich werden, wenn z.B. Änderungen vorgenommen werden, ohne diese zu registrieren, vielleicht um Varianten durchzuprobieren (ähnlich einem Variantenbrett beim Schach, das mit der aktuellen Partie wieder synchronisiert werden soll). Der Hauptzweck ist jedoch die Rekonstruktion der Szenenhierarchie nach einer gescheiterten Transaktion. Mit dem Aufruf von `abortTransaktion()` (s.o. Transaktionen) wird automatisch eine Rekonstruktion der Szene durchgeführt.

Auswahl von Diensten - Um den Kommunikationsaufwand zu reduzieren kann sich jeder Teilnehmer einer Konferenz für bestimmte Dienste anmelden. Er bekommt nur nach dem Setzen eines entsprechenden Flags die korrespondierenden Nachrichten zugestellt. Das Versenden von Nachrichten kann jedoch zu allen Diensten erfolgen, auch wenn der Client selbst nicht daran interessiert ist. Über die Funktion `updates()` kann die Anwendung erfahren, zu welchen Diensten neue Daten eingetroffen sind und die entsprechenden Funktionen zum Bearbeiten dieser Daten aufrufen. Über die Funktionen `getFlags()`, `setFlags()`, `addFlags()` und `removeFlags()` werden diese Dienste an- und abgemeldet. Folgende Dienste stehen zur Verfügung:

- `CONF_CAMERA`: Die Kamerapositionen aller Teilnehmer werden kontinuierlich aktualisiert.
- `CONF_AVATAR`: Mit den eintreffenden Kameradaten der Teilnehmer werden die zugehörigen Avatare automatisch bewegt. Mit jedem Aufruf der Funktion `updateWorld()` werden die Avatarpositionen aktualisiert. (implizit wird `CONF_CAMERA` aktiviert)
- `CONF_USER`: Änderungen von Benutzerdaten werden übermittelt. Das Hinzufügen und Löschen von Teilnehmern wird dadurch nicht beeinflußt.
- `CONF_CHAT`: Chat-Nachrichten werden in einem Puffer zwischengespeichert und können abgerufen werden.
- `CONF_WORLD`: Sicher zu übertragende Änderungen der Szenendaten werden übermittelt und kontinuierlich aktualisiert.
- `CONF_UNSEC`: Empfang von unsicheren Szenendaten. Hierbei handelt es sich hauptsächlich um durch Automaten gesteuerte Objekte, die kontinuierlich modifiziert werden.
- `CONF_STAT`: Statistikdaten werden gesammelt. Dieser Dienst ist zwar nicht in die Kommunikation involviert, es erspart aber Rechenzeit, wenn er nicht genutzt wird, weshalb diese Funktionalität auch abgeschaltet werden kann.
- `CONF_ALL`
- `CONF_NONE`: Setzen oder Löschen aller Dienste

Das **Lesen und Ändern der Benutzerdaten** Name, Nickname, eMail-Adresse und Kommentar erfolgt über entsprechende überladene Funktionen zum Setzen und Lesen

der Attributwerte. Abbildung 7-7 zeigt dies am Beispiel des Benutzernamens. Die Attribute UID, Hostname, und iiop-Adresse können nur gelesen werden. Alle Funktionen kehren direkt nach ihrem Aufruf zurück, da durch die impliziten Sperren keine Sperren angefordert werden müssen und die Daten somit definitiv geändert werden können.

```
char* name();
void name(const char* _name);
```

Abbildung 7-7: Funktionen zum Lesen und Schreiben des Benutzernamens

Die Daten aller Anwender können mit `getUserList()` angefordert werden. Mit `getWithNick()` können die Daten eines über den Nicknamen spezifizierten Teilnehmers geholt werden.

7.3 Datenkatalog

In diesem Abschnitt werden die Attribute und Methoden der wichtigsten Objekte und Datenstrukturen beschrieben.

7.3.1 Konferenzteilnehmerdaten (`t_UserDescription` und `t_FullUserDescription`)

Der Konferenzteilnehmerdatensatz besteht aus einem öffentlichen Teil, der allen Konferenzteilnehmern zu Informationszwecken offensteht, sowie einem internen Teil, der nur innerhalb der Datenhaltung genutzt wird und sichtbar ist. Die öffentlichen Daten sind zusammengefaßt im Struct `t_UserDescription`. Alle Informationen können lediglich vom zugehörigen Teilnehmer selbst geändert werden. Die UID wird vom Server zugewiesen und identifiziert den Teilnehmer eindeutig, auch außerhalb der Datenhaltungsschicht. Ebenfalls eindeutig muß auch der Nickname des Teilnehmers sein. Im Konfliktfall wird der Nickname durch das Anhängen einer Nummer eindeutig gemacht. Ist z.B. der Nickname "schneemann" vergeben, wird der nächste ungenutzte Name z.B. "schneemann4" erzeugt. Dieses automatische Vorgehen reduziert den Kommunikationsaufwand auf ein Minimum, und ist insbesondere für Automaten etc. sinnvoll, deren Nickname ohnehin irrelevant ist.

Bezeichner	**Definition (OMG IDL Typen)**	**benutzt für/von**
`uid`	Eindeutige UserID (UID) Typ: `t_ID`	Schlüssel
`name`	Name des Teilnehmers Typ: `String`	Eigene Benutzerinformationen (s.o.)
`nick`	Eindeutiger Teilnehmername Typ: `String`	
`host`	Name des vom Teilnehmer verwendeten Rechners Typ: `String`	
`email`	Mailadresse des Teilnehmers Typ: `String`	
`comment`	Beliebige Zusatzinformation Typ: `String`	
`avatarOID`	OID des Avatarbasisobjektes Typ: `t_ID`	Wird automatisch mit der Registrierung eines Avatars gesetzt

Tabelle 7-1: Datenverzeichnis der öffentlichen Teilnehmerdaten

Der Hostname wird automatisch auf den Rechnernamen des lokalen Rechners gesetzt und kann nicht geändert werden.

Die übrigen Informationen sind optional, und werden bei Nichtangabe durch das System mit Standardinformationen gefüllt. (Der Name wird auf den Loginnamen gesetzt, wenn einer gefunden wird, die Emailadresse wird aus name@host gebildet.) Optional können auch die Environmentvariablen `MUIMRT_NICK`, `MUIMRT_NAME`, `MUIMRT_EMAIL` und `MUIMRT_COMMENT` gesetzt werden.

Die internen Daten sind im Struct `t_FullUserDescription` zusammengefaßt. Die übrigen Informationen werden für die Sammlung der teilnehmerspezifischen Daten genutzt. Das erste Element ist das Struct `User` der öffentlichen Daten, das letzte Element das `protokoll` Strukt, welches vom Broadcaster verwaltet wird.

Bezeichner	**Definition (OMG IDL Typen)**	**Benutzt für/von**
`user`	Öffentliche Userdaten Typ: `t_UserDescription`	Siehe Tabelle 7-1
`conference`	Objektreferenz des CORBA-Konferenz-Objektes Typ: `t_Conference`	direktes Ansprechen der Konferenz und des Messagehandlers des Teilnehmers.
`msgHandler`	Objektreferenz des CORBA-Messagehandler-Objektes Typ: `t_MsgHandler`	
`camera`	Liste der Kameraparameter Typ: `t_ParamList`	Bewegen des Avatars und Übernahme durch andere Teilnehmer.
`camChanged`	Änderungsmarkierung Typ: `boolean`	zeigt ob sich die Kamera verändert hat, seit dem letzten Auslesen
`lockList`	OID Liste Typ: `Sequence < t_ID >`	Liste aller selbst gesperrten Objekte
`accessRight`	Zugriffsrechte Typ: `t_Rights`	Konferenz entscheidet welche Aktionen ausgeführt werden dürfen
`services`	Flags für abonnierte Dienst Typ: `t_Flags`	Broadcaster entscheidet welche Nachrichten zugestellt werden sollen
`protokoll`	Protokoll der anstehenden Übertragungen Typ: `t_Protokoll`	siehe Tabelle 7-6 Broadcaster

Tabelle 7-2: Datenverzeichnis der internen Teilnehmerdaten

7.3.2 Szenenobjektdatensatz (`t_RefPtr_ID`)

In einer Sequenz von Szenenobjektdatensätzen speichert das MUI-MRT die beiden Versionen der Szenendarstellung. Über einen Datensatz sind beide Versionen eines Objektes erreichbar. Das Änderungsflag `change` hält die Art des Versionsunterschiedes fest. Das Versionsmanagement wird in Kapitel 6.2 ausführlich beschrieben.

Bezeichner	**Definition**	**Benutzt für/von**
`change`	Änderungsflag für `updateworld()` Typ: `t_Changes`	Synchronisationsflag
`objPtr`	Pointer auf das Objekt in der Szene Typ: `t_RtiRefPtr`	Leseversion als Pointer auf das Objekt
`params`	Rekonstruktionsparameter Typ: `t_ParamList`	Schreibversion als Parameterliste des Objektes. (Tabelle 7-4)

Tabelle 7-3: Datenverzeichnis des 2-Versions-Datensatzes

7.3.3 Parameterliste (`t_ParamList`)

Die Parameterliste beinhaltet den aktuellen Status eines Objektes. Die Gesamtheit aller Parameterlisten entspricht der aktuellen Version der Szenenbeschreibung. Neben den benötigten Rekonstruktionsparametern sind noch einige weitere Daten gespeichert, die zu Beschleunigungs- und Verwaltungszwecken verwendet werden. Da sie objektspezifisch sind, stellt die gesamte Parameterliste ein Objekt dar und wird vollständig übertragen.

Bezeichner	**Definition (OMG IDL Typen)**	**Benutzt für/von**
`objID`	eindeutige ObjektID (OID) Typ: `t_ID`	Schlüssel
`disID`	Eindeutige ID spezifiziert den Constructor über den Parser Typ: `t_ID`	Objektliste legt die Rückgabeparameter der Funktion `writeObject()` hier ab, um das Objekt über den mit der `disID` spezifizierten Constructor zu rekonstruieren.
`mat`	`t_4x3Matrix` als CORBA-Strukt Typ: `t_4x3Matrix_Data`	
`param`	Parameter für den Constructor Typ: `sequence < any >`	
`shader`	ID eines zu einzuwechselnden Shaders Typ: `t_ID`	Shader zwischenspeichern, entweder um nach Zurücksetzung einer Markierung den ursprünglichen Shader zu setzen oder einen Austauschshader einzutragen
`marked`	Markierung eines Objektes Typ: `boolean`	
`container`	OIDs der Containerobjekte (es können mehrere sein z.B. RefObjekte) Typ: `sequence < t_ID >`	zum Aktualisieren der Containerobjekte bei einer Änderung durch Neuerzeugung (Kapitel 6.1)
`isAvatar`	Element ist Teil des Avatars des Benutzers mit der UID `creatorID` Typ: `boolean`	zum Löschen aller Avatarobjekte eines abgemeldeten Teilnehmers.
`creatorID`	UID des Users, der dieses Objekt erzeugt hat Typ: `t_ID`	

Tabelle 7-4: Datenverzeichnis der Parameterliste eines MRT-Objektes

7.3.4 Protokoll (t_Protokoll)

Das Protokoll nimmt alle Nachrichten auf, die semisynchron versendet werden sollen. Eine genaue Beschreibung des Vorgangs wird in Kapitel 7.4.6 gegeben. Ein einzelner Aufruf wird in einem CallStruct gespeichert. Die Inhalte seiner Datenfelder werden anhand der in Abbildung 7-8 dargestellten Methode verdeutlicht.

```
      void cmdTransform(in t_ID _oid,
                        in t_4x3Matrix_Data _mat);
```

Abbildung 7-8: Definition einer Nachricht des MessageHandlers

Bezeichner	Definition (OMG IDL Typen)	Benutzt für/von
functionName	Bezeichner der aufzurufenden Funktion (z.B. "cmdTransform") Typ: string	Der BroadCaster erstellt aus diesen Angaben eine Anfragestruktur (Request), die über das DII versandt wird. (Kapitel 4.5, 7.4.6)
names	Liste der Parameternamen (z.B. "_oid", "_mat") Typ: sequence < string >	
values	Liste der Parameterwerte (z.B. 1, (1,0,0, 0,1,0, 0,0,1, 0,0,0)) Typ: sequence < any >	
important	Flag für wichtige oder unwichtige Nachrichten Typ: boolean	Unwichtige Nachrichten kann der BroadCaster ignorieren.

Tabelle 7-5: Datenverzeichnis eines CallList struct

Das Protokoll besteht aus einer Liste von CallStructs und einigen Zusatzinformationen über die Historie.

Bezeichner	Definition (OMG IDL Typen)	Benutzt für/von
callList	Liste der Funktionen im Protokoll Typ: t_CallList	Funktionsaufruf via DII durch BroadCaster (Tabelle 7-5)
count	Anzahl der fehlgeschlagenen Übertragungen Typ: unsigned long	Für die Statistik im BroadCaster
time	Zeit seit Beginn eines Verbindungsverlustes Typ: unsigned long	Timeoutberechnung im BroadCaster

Tabelle 7-6: Datenverzeichnis des Protokolls

7.4 Beschreibung der Objektimplementierungen

7.4.1 Interface Objekte `t_MUI` und `t_ConfInterface`

Die Interfaceobjekte dienen hauptsächlich der Zusammenfassung aller Operationen, die die Schnittstelle zur Datenhaltungsschicht bilden. Die Aufrufe werden weitergereicht an das (oder die) jeweils zuständige(n) Objekt(e). Exceptions der CORBA-Objekte werden in Fehlermeldungen umgewandelt, die mittels `getLastError()` als String abgefragt werden können. Die Funktionen wurden bereits in Kapitel 7.2 beschrieben.

7.4.2 Konferenzen - Administration (`t_ConfAdmin`)

Der Konferenzadministrator verwaltet alle in einer Anwendung laufenden Konferenzen, beziehungsweise er führt eine Liste mit den Objektreferenzen der `t_Conference`-Objekte und den dazugehörenden Konferenzschnittstellen. Über ihn werden innerhalb der Anwendung Konferenzen und ihren Schnittstellen erzeugt oder aufgelöst (Fabrikobjekt für Konferenzen).

Als CORBA-Objekt bietet er das in Abbildung 7-9 dargestellte Interface an und ermöglicht somit den Zugriff auf alle von ihm verwalteten Konferenzen. Mit `getConferenceNames()` kann eine Liste der Konferenznamen angefordert werden und mit einem Konfernznamen über `getConference()` die zugehörige Objektreferenz. Als Konferenzadministrator ist er selbst immer unter der IIOP-Adresse "iiop://hostname:port/Conference-Admin" beim ORB angemeldet.

Ist die Anwendung als dedizierter Konferenzserver initialisiert worden, erlaubt der Server das Erzeugen von Konferenzen auch von außen.

```
interface t_ConfAdmin
{
   t_Conference getConference(in string _confname)
      raises(UnknownServer);
   t_StringSeq getConferenceNames();
   t_Conference createConferenceServer(in string _confname)
      raises(AlreadyRegistered, NotAllowed);
 };
```

Abbildung 7-9: OMG IDL Definition der Schnittstelle von t_ConfAdmin

7.4.3 Konferenz (`t_Conference`)

Das Konferenzobjekt ist das zentrale Kontrollorgan einer Konferenz. Es kann sowohl die Rolle des Servers übernehmen, als auch die eines Client, der alle Informationen an den Server weiterleitet.

Im **Client-Modus** werden folgende Aufgaben erfüllt:

- **Anmeldung:** Die wichtigste Aufgabe eines Clients ist es, sich bei einem Server anzumelden. Mit dem Login bei einem Server wartet das Konferenzobjekt, bis es alle der Replikation unterliegenden Daten in der Version erhalten hat, die im Moment des Logins beim Server bestand.
- **Transaktionsmanagement:** Das Transaktionsmanagement findet vollständig innerhalb des Konferenzobjektes statt. Eingehende Operationen werden nach den

Kriterien der Deadlock-Verhütung und den ACID-Eigenschaften sortiert und gesammelt. Die Objektliste wird angewiesen alle Operationen zu protokollieren, um sie gegebenenfalls zurücksetzen zu können. Mit der Beendigung der Transaktion wird sie an den Server weitergeleitet und lokal umgesetzt. Die Objektlist kann ihr Protokoll verwerfen. Schlägt eine der Operationen fehl, wird die Transaktion abgebrochen. Die Objektliste muß anhand des Protokolls alle Operationen zurücksetzen. Geschachtelte Transaktionen werden intern in eine große Transaktion aufgelöst. Die Transaktionstiefe wird nur als interner Zähler geführt und hat keine Auswirkungen z.B. auf die Protokollierung in der Objektliste. Diese wird beim ersten `openTransaction()` begonnen und beim letzen `closeTransaction()` beendet. Erreicht der Tiefenzähler 0, ist die Transaktion beendet.

- **Chat:** Eingehende Textnachrichten werden bis zu ihrem Abruf durch die Anwendung gespeichert.
- **Zugriffsrechte:** Bevor eine Nachricht an den Server weitergeleitet wird, überprüft das Konferenzobjekt, ob es selbst die erforderlichen Rechte besitzt, um die mit der Nachricht verbundenen Aktionen durchzuführen. So kann in den meisten Fällen eine Kommunikation verhindert werden, die vom Server ohnehin abgelehnt würde.
- **Sperren:** Wie bei den Zugriffsrechten werden auch Sperren anhand der vorliegenden Sperrenliste vorab überprüft, um überflüssige Kommunikation zu vermeiden.
- **Weiterleitung:** Alle Nachrichten von anderen Clienten werden entweder an den Server weitergeleitet oder über die Exception `NotActiveServer()`abgelehnt. Als Parameter der Exception wird die Objektreferenz des Servers übertragen.

Im **Server-Modus** werden zusätzlich folgende Aufgaben erfüllt:

- **Client-Registrierung:** Über die Funktion `registerMe()` meldet sich ein Client beim Server an. Daraufhin fügt der Server den neuen Teilnehmer der Userliste mit `add()` hinzu, holt von der Objektliste mit `getWorld()` die gesamte Szenenbeschreibung in Form einer Sequenz der Parameterlisten (Kapitel 7.3.3) und von der Userliste mit `getPublicUserList()` den öffentlichen Teil der Teilnehmerliste. Diese Daten werden über den Broadcaster an den neuen Teilnehmer gesendet. Die bisherigen Konferenzteilnehmer werden abschließend über den neuen Client informiert, der noch die `cmdUpToDate()`-Nachricht über den Broadcaster erhält, um ihm anzuzeigen, daß alle Daten vollständig übertragen wurden.
- **OID-Vergabe:** Clienten können über `getNewOIDs()` eine beliebige Anzahl von neuen OIDs für die von ihnen neu einzubringenden Objekte anfordern. So ist die Eindeutigkeit der OIDs durch die zentrale Vergabe gewährleistet.
- **Sperren:** Anhand der vorhandenen Sperren in der Userliste wird eine angeforderte Sperre für ein Objekt geprüft und an den Teilnehmer vergeben.
- **Zugriffsrechte:** Vor der Bearbeitung aller eingehenden Nachrichten wird die Berechtigung des Clients überprüft, diese beabsichtigte Aktion durchzuführen.
- **Verteilung:** Alle Daten verändernden Nachrichten werden über den Broadcaster an alle Teilnehmer verteilt. Dazu wird die Nachricht in ein `t_CallStruct` verpackt und mit einer der `addCall()`-Methoden an den Broadcaster übergeben.

7.4.4 Teilnehmerliste (`t_UserList`)

Das Objekt `t_UserList` kapselt die Tabelle der Teilnehmerdaten, die in einer STL-MAP für effizienten Zugriff über die UID gespeichert sind. Die im Datenkatalog angegebenen Daten werden über Funktionen gesetzt und ausgelesen, die für die Einhaltung

der Integritätsbedingungen sorgen. So wird beim Eintragen eines Nicknamen geprüft, ob dieser bereits vorhanden ist. Durch Hinzufügen einer Nummer wird der Name so geändert, daß er einmalig ist. (Kapitel 7.3.1)

Die Informationen über gesperrte Elemente wird mit dem Teilnehmerdatensatz gespeichert. Die Userliste überprüft beim Setzen einer Sperre, ob diese bereits von einem anderen Teilnehmer gesetzt wurde. Ist dies nicht der Fall wird der Sperreintrag zugelassen, sonst wird er abgelehnt. Durch das Entfernen eines Teilnehmerdatensatzes werden damit automatisch alle von ihm gehaltenen Sperren gelöscht.

Für interne Zwecke werden weitere Informationen mit einem User-Datensatz gespeichert und teilweise an die Clienten übertragen. Sie finden innerhalb der Datenhaltung Verwendung und bleiben nach außen unsichtbar. Tabelle 7-2 im Kapitel 7.3.1 beschreibt diese Daten.

7.4.5 Objektliste (`t_ObjectList`)

Die Bezeichnung Objekt ist in diesem Abschnitt nicht immer eindeutig zuweisbar. Es kann einerseits die Objektliste gemeint sein oder aber auch ein graphisches Objekt. Alle graphischen Objekte werden hier als Elemente bezeichnet.

Hinter der Objektliste verbirgt sich die gesamte Logik der Szenendatenverwaltung. Nach außen hin, bzw. vom Konferenz-Interface aus, werden in allen Operationen `t_RtiRefPtr` als Parameter für die Elemente angeboten (z.B. `exchangeShader(t_SurfaceObjectPtr, t_ShaderPtr)`). Das heißt für die Anwendung bleibt die OID verborgen, und damit die gesamte Konvertierung der Hierarchie in eine Liste und die Umsetzung der MRT-Objekte (Elemente) und Typen in entsprechende CORBA-Typ-Konstrukte. In Abbildung 7-10 wird dieser Anwendungs- und MRT-spezifische Teil mit grün gefärbten Objekten dargestellt. Aus der Sicht der Konferenzverwaltung und aller anderen Objekte ist hingegen nur die Verwaltung von über OIDs adressierten Elementdatensätze sichtbar (z.B. `cmdExchange(t_ID _oid, t_ID _sid)`). Die Szenenhierarchie oder die Elementtypen spielen keine Rolle. Dieser Bereich wird durch die gelben Objekte repräsentiert. Nur innerhalb der Objektliste sind beide Welten vorhanden und werden miteinander synchronisiert.

Die Objektdaten, bestehend aus `t_RefPtr_ID` Datensätzen (Tabelle 7-3), werden in einem binären Suchbaum (STL-MAP) gespeichert. So kann auf einzelne Elemente mit logarithmischem Aufwand anhand der OID zugegriffen werden.

Im Folgenden werden die Ereignisse näher beleuchtet, die die Objektliste und die mit ihr kooperierenden Objekte austauschen.

Schnittstelle zum Konferenzinterface

Aus Abbildung 7-10 bereits ersichtlich, werden die Registrierungs-, Lösch- und Änderungsoperationen (`transformObject()`, `translateObject()` und `exchangeShader()`) sowie die Markierung vom Konferenzinterface an die Objektliste weitergeleitet. Hier wird anhand der in den Elementen gespeicherten OID zunächst festgestellt, ob das Element bereits registriert wurde. Anschließend wird überprüft, ob eine Sperre gesetzt ist, vorausgesetzt es handelt sich nicht um ein neues Element. Nachdem alle Überprüfungen positiv verlaufen sind, wird das Element oder der übergebene Parameter auf die in Kapitel 6.1 beschriebene Weise in einen CORBA-Typ konvertiert. Aus der Objekt-

tabelle wird der entsprechende Datensatz gesucht und die ParameterListe des Elementes auf den Rollbackstack (s.u.) gelegt und nachfolgend die Parameterliste entsprechend den Anforderungen der Operation geändert und zurückgespeichert. Abschließend wird noch das Versionsänderungsflag gesetzt und der neue Datensatz an das Konferenzobjekt übergeben, um es an die anderen Teilnehmer weiterzuleiten. Diesen Vorgang verdeutlicht der rote Pfeil in Abbildung 7-10.

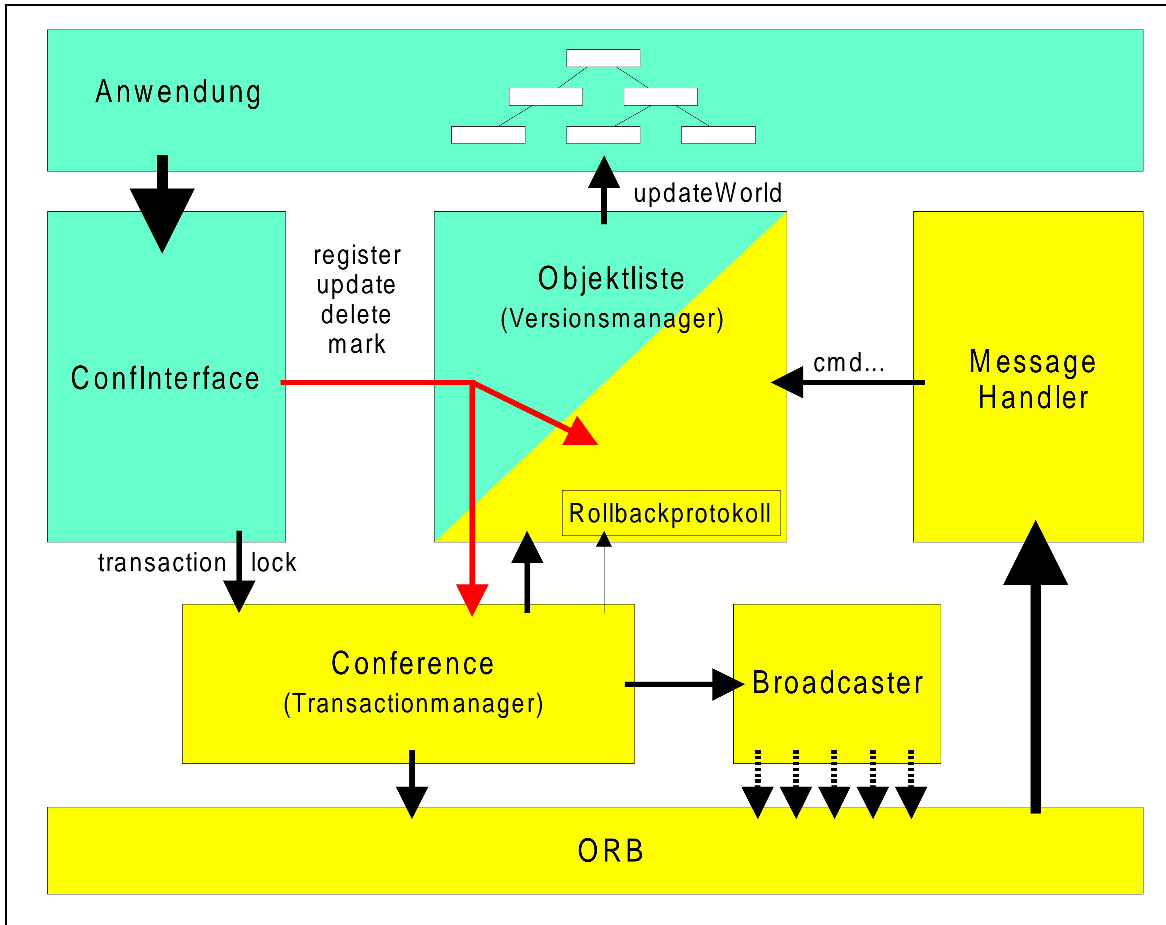


Abbildung 7-10: Ereignisumfeld der Objektliste

Schnittstelle zur Anwendung

Mittels `updateWorld()` initiiert die Anwendung die Versionssynchronisation (Kapitel 6.2). Für jedes geänderte Element wird basierend auf dem gesetzten Versionsänderungsflag die entsprechende Operation durchgeführt.

Schnittstelle zum Konferenzobjekt

Die Schnittstelle zur Konferenz wird bidirektional genutzt. Die Objektliste leitet alle durchgeführten Änderungen an das Konferenzobjekt weiter. Dabei werden lediglich die OID und einige änderungsspezifische Parameter übergeben. Ob diese nun direkt weitergeleitet oder durch das Transaktionsmanagement gesammelt und sortiert werden, spielt keine Rolle. Die gesamte Transaktionskontrolle unterliegt dem Konferenzobjekt. Es steuert lediglich das innerhalb der Objektliste liegende RollbackProtokoll. Nach dem Aufruf von `startRollbackProtokoll()` werden alle Parameterlisten vor der Änderung auf dem Protokollstack abgelegt. Dieser Stack ist als STL VECTOR implementiert. Mit `stopRollbackProtokoll()` wird das Protokollieren beendet und der Stack geleert. Beim Aufruf von `resetRollbackProtokoll()` wird der Stack abgebaut und

die gespeicherten Parameterlisten wieder in die Objektliste zurückgeschrieben. So wird eine Transaktion vollständig zurückgesetzt. Zum Schluß versetzt der Aufruf von `reconstructWorld()` auch die in der Anwendung gehaltene Szenenhierarchie (Leseversion) in den ursprünglichen Zustand zurück.

Schnittstelle zum Messagehandler
Über den Messagehandler eingehende Kommandos, die Replikationsänderungen vom Server darstellen, werden in die Schreibversion der Szene eingefügt und die Versionsänderungsflags aktualisiert.

7.4.6 Protokollierender Nachrichtenverteiler (`t_Broadcaster`)

Das Objekt `t_Broadcaster` übernimmt die Aufgabe des Versandes aller Nachrichten, die sicher ankommen müssen, aber die Ausführung nicht blockieren dürfen. Nur der Server verwendet den Broadcaster um Nachrichten an alle Teilnehmer zu versenden, entsprechend dem in Kapitel 3.5.3 beschriebenen Primary-Copy-Verfahren. Für jeden Teilnehmer wird ein Protokoll der zu sendenden Nachrichten geführt. Die Funktion `addCall()` trägt die Parameter einer Nachricht in die Protokolle der zu benachrichtigenden Teilnehmer ein. An dieser Stelle werden Nachrichten aussortiert, wenn sie einen vom Teilnehmer nicht abonnierten Dienst betreffen.

Ein seperater Thread überprüft die Protokolle kontinuierlich. Nach dem FIFO-Prinzip werden die Nachrichten über das DII (Kapitel 4.5) semisynchron mittels `send_deferred()` versandt. Während eine Nachricht unterwegs ist wird regelmäßig überprüft, ob diese zugestellt wurde. Gelingt dies, wird der Protokolleintrag entfernt und die nächste Nachricht des entsprechenden Teilnehmers abgeschickt. Das Scheitern einer Übertragung kann viele verschiedene Gründe haben (Kapitel 2.4), die der Broadcaster entsprechend zu interpretieren versucht. Wenn aus der Fehlermeldung eindeutig hervorgeht, daß der Client nicht mehr existiert, wird der lokale Konferenz-Server über das Ausscheiden des Teilnehmers informiert und das Protokoll gelöscht. In allen anderen Fällen verschickt der Broadcaster die Nachricht erneut. Dieser Vorgang wiederholt sich, bis eine Wartezeit von mehreren Minuten überschritten wurde. Danach meldet der Broadcaster dem Server ebenfalls das Ausscheiden des Teilnehmers. Es muß sich in der Praxis zeigen, wie groß diese Wartezeit gewählt werden sollte. Kann in der Wartezeit die Verbindung zum Client wiederhergestellt werden, sendet der Broadcaster die gesammelten Nachrichten zu und bringt somit den Client auf den aktuellen Stand.

Ein vom Broadcaster "aussortierter" Teilnehmer erhält bei einer Wiederherstellung der Verbindung eine Fehlermeldung und muß sich als neuer Teilnehmer anmelden.

Um während längerer Wartezeiten das Protokoll nicht übermäßig wachsen zu lassen, entfernt der Broadcaster Nachrichten, die als unwichtig markiert wurden. Beispielsweise könnte ein Automat, der eine Uhr steuert seine Zeigerbewegungen, die jede Sekunde übertragen werden als unwichtige Nachrichten markieren. Bei einer längeren Wartezeit sind diese ohnehin irrelevant. Zusätzlich werden die Nachrichten zu größeren Bündeln von 50 oder 100 Nachrichten zusammengefaßt, um den Kommunikationsaufwand zu verringern (nicht das Datenvolumen!). Der Client kann bei einem Reconnect diese Nachrichtenbündel, die mit der Nachricht `cmdCallList()` übertragen werden, lokal auspacken und ausführen.

In den Broadcaster ließen sich noch weitere interessante Verfahren einbinden, um die Daten zu reduzieren und die Performance zu steigern. Im Ausblick (Kapitel 9) werden einige davon erwähnt.

Der Broadcaster sorgt dafür, daß die Daten aller Teilnehmer konsistent bleiben. Es gehen keine relevanten Informationen verloren, wenn ein Teilnehmer trotz mehrfacher Versuche nicht erreichbar ist. Er wird ausgeschlossen, und tangiert die Konsistenz nicht mehr. Meldet er sich erneut an, bekommt er den aktuellen Datenzustand übermittelt und die Konsistenz ist wieder hergestellt.

7.4.7 Kameraverteiler (`t_CameraThread`)

Den Kameraverteiler könnte man als kleinen Bruder des Broadcasters bezeichnen. Er ist nur für den Versand der Kameradaten zuständig und führt auch kein Protokoll, sondern versucht immer nur die aktuellste Kamera zuzustellen. Jeder Client der Kameradaten versendet, erledigt das über den Kameraverteiler. Es wird im Gegensatz zum Broadcaster auch nicht überprüft, ob die Nachrichten ankommen, da diese über die oneway-Funktion von CORBA direkt und ohne Rückmeldung verschickt werden.

7.4.8 Nachrichtendispatcher (`t_MsgHandler`)

Das Gegenstück der Nachrichtenverteiler ist der Messagehandler oder Nachrichtendispatcher. Er dient einzig dem Empfang der Nachrichten und der Verteilung an die in der Datenhaltungsschicht liegenden Objekte. Er bündelt ähnlich dem Konferenzinterface alle Funktionen in einem Objekt und dient lediglich der intelligenten Weiterleitung.

Der Messagehandler ersetzt den Dispatcher des MRT-VR, d.h. eigentlich ersetzt der BOA die Funktionalität des Dispatchers, indem die eintreffenden Nachrichten in Methodenaufrufe des Messagehandlers und der anderen CORBA-Objekte umgesetzt werden.

8 Laufzeitverhalten des MUI-MRT

In diesem Kapitel wird die Implementierung der Datenhaltungsschicht auf ihr Laufzeitverhalten hin überprüft. Dabei sind die entscheidenden Punkte:

1. Der erreichbar Datendurchsatz und somit die Anzahl der durchführbaren parallelen Änderungen.
2. Die Reaktionszeit auf Änderungen und die Verzögerung bis zur Aktualisierung bei allen Teilnehmern.
3. Stabilität bei auftretenden Netzwerkfehlern oder ausfallenden Clients.

Die durch das Netzwerk bedingten Verzögerungen und Bandbreiten haben entscheidenden Einfluß auf die maximal erreichbare Performance. Um aber die Möglichkeiten und Grenzwerte der Datenhaltung unabhängig davon beurteilen zu können, werden die folgenden Testprogramme auf einem einzelnen Rechner, also ohne Netzwerk durchgeführt. Der Testrechner ist ein Doppelprozessor Pentium 100 mit 64 MB Hauptspeicher unter Windows NT 4.0.

Die angesprochenen Punkte 1 und 2 wurden durch zwei Meßreihen näher untersucht. Die erste analysiert den Datendurchsatz in Abhängigkeit von der Szenengröße, indem die Zeit des Anmeldens bei einer Konferenz und die Übertragung der gesamten Szenendaten gemessen wurde. Beim zweiten Test wurde die Verzögerung der Übertragung von Nachrichten über den Broadcaster an alle Teilnehmer einer Konferenz gemessen, in Abhängigkeit von der Teilnehmeranzahl.

Zu Punkt 3 werden die Erfahrungen bei der Entwicklung des Systems und einige Experimente in (Kapitel 8.3) näher erläutert.

8.1 Abhängigkeit von der Szenengröße

Untersucht wurde, in welchem Rahmen die Datenübertragungsleistung von der Anzahl der Operationen abhängt. Da alle Operationen jeweils in einer Transaktion gebündelt werden, wird hier die Auswirkung der Größe von Transaktionen auf die Übertragungszeit gemessen. Zu einer Übertragung gehört immer das Zusammenfassen der Daten in einer Transaktion, der Versand über den Broadcaster, der Empfang durch den Messagehandler sowie die Aktualisierung der Daten in der Schreibversion. Die Synchronisierung zwischen Schreib- und Leseversion wurde nicht berücksichtigt, da sie über den Parser alle neuen Objekte erzeugen muß. Dieser Zeitaufwand ist MRT-spezifisch, und entsteht somit auch beim Einladen einer Szene von der lokalen Festplatte (ohne Datenhaltung).

Mit der Anmeldung bei einem Server erhält der Teilnehmer alle Daten. Die Zeitspanne zwischen der Bestätigung des Logins und dem Abschluß der Datenübertragung wird innerhalb der Datenhaltung im Konferenzinterface gemessen. Für diesen Test meldet sich ein minimaler Monitorclient ohne Visualisierung beim Server an, der sich nur anmeldet, auf den Empfang der Daten wartet und sich danach wieder abmeldet. Dieser Vorgang wurde 100 mal wiederholt und die Meßergebnisse, bestehend aus Mittelwert, sowie minimaler und maximaler Übertragungszeit, in Millisekunden ausgegeben. Diese Testläufe wurden für Szenen mit 10, 20, 50, 100, 200, 500 und 1000 Kugeln durchge-

führt. Tabelle 8-1 zeigt die erzielten Ergebnisse und Abbildung 8-1 stellt diese graphisch dar.

Anzahl der Szenenelemente	10	20	50	100	200	500	1000
Minimum (ms)	15	15	62	171	391	906	1938
Maximum (ms)	63	78	125	219	578	1531	2890
Mittelwert (ms)	33	52	97	190	487	1087	2223
Linear (ms)	22	44	110	220	440	1100	2200

Tabelle 8-1: Meßergebnisse für Konferenzanmeldung in Millisekunden

Aus den Meßwerten wird ersichtlich, daß die Größe der Szene auf den Durchsatz keinen Einfluß hat. Die Übertragungszeit steigt im Mittel proportional zur Anzahl der Objekte. Auf dem Testsystem beträgt der Faktor 2,2 also ca. 455 Kugel-Objekte pro Sekunde. Das heißt also, daß die Verwaltung der Daten und das Einfügen in die Schreibversion nur einen konstanten Zeitaufwand erfordern. Für die Transaktionen im laufenden Betrieb ist entsprechend das gleiche Verhalten zu erwarten.

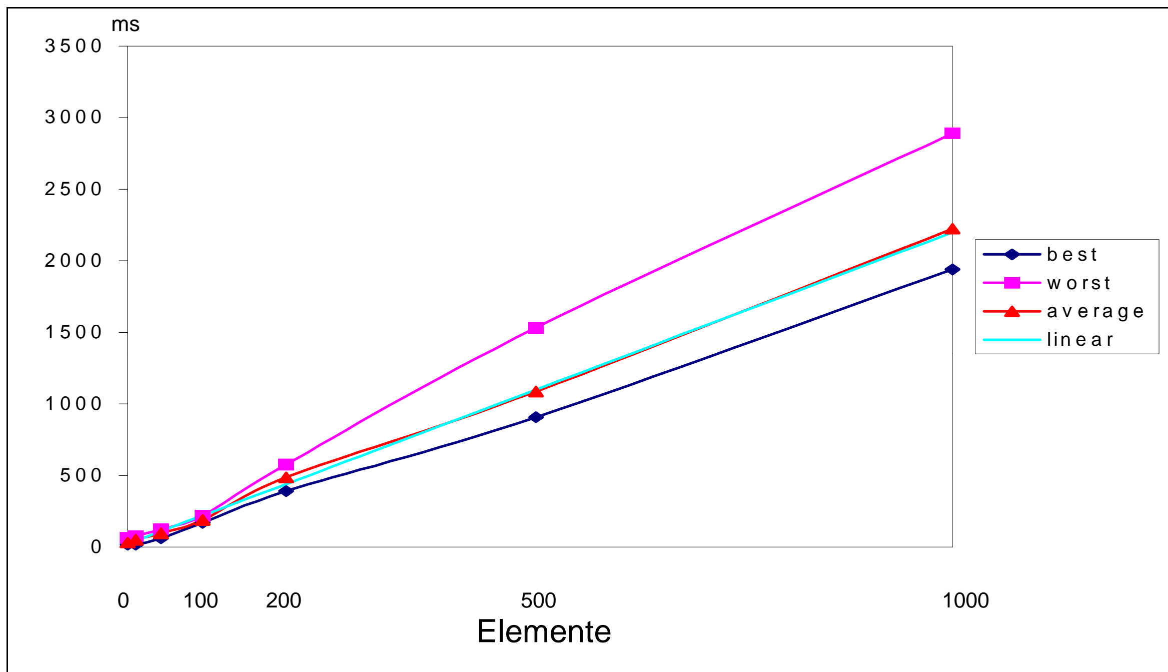


Abbildung 8-1: Diagramm der Performancemessung *Konferenzanmeldung*

8.2 Abhängigkeit von der Teilnehmeranzahl

Nun wurde überprüft, wie die Zahl der Teilnehmer die Aktualisierungsgeschwindigkeit beeinflußt. Dazu wurde eine neue Nachricht (`cmdPing()`) integriert, die den Zeitpunkt des Versandes überträgt und somit die Berechnung der Verzögerung bei ihrem Eintreffen ermöglicht. Diese Nachricht wird von einem Automaten an einen dedizierten Konferenzserver gesendet und von ihm an alle Teilnehmer weitergeleitet. Die Teilnehmer bestehen aus reinen Monitorprogrammen, die nur die Nachricht empfangen und die Verzögerung ausgeben. Dabei wurden die Daten zur Berechnung der maximalen und der minimalen Verzögerung jeweils über einen Zeitraum von 30 Sekunden betrachtet. Nach und nach wurden zusätzliche Monitorprogramme angemeldet und die Werte überprüft. 20 mal pro Sekunde wurde die Testnachricht versandt.

Clients	1	2	3	4	5	6	7	8	9	10	15	20
Minimum (ms)	0	0	0	14	15	18	19	22	28	30	46	61
Maximum (ms)	9	14	20	22	25	31	38	42	47	51	70	95

Tabelle 8-2: Verzögerung von Nachrichten für steigende Teilnehmerzahlen

Es stellte sich heraus, daß die Verzögerung der Übertragung für einen Teilnehmer nahezu konstant bleibt, wenn weitere Teilnehmer hinzukommen. Die Werte in Tabelle 8-2 zeigen die Ergebnisse für den jeweils x-ten Teilnehmer der Konferenz. Der Teilnehmer, der sich als letzter anmeldet, erhält auch als letzter die Daten. Dieses Verhalten spiegelt genau die Arbeitsweise des Broadcasters wieder, alle Teilnehmer nacheinander abzuarbeiten und jeweils eine Nachricht zu verschicken. Das Diagramm in Abbildung 8-2 zeigt den linearen Anstieg der Verzögerung, proportional zur Teilnehmerzahl.

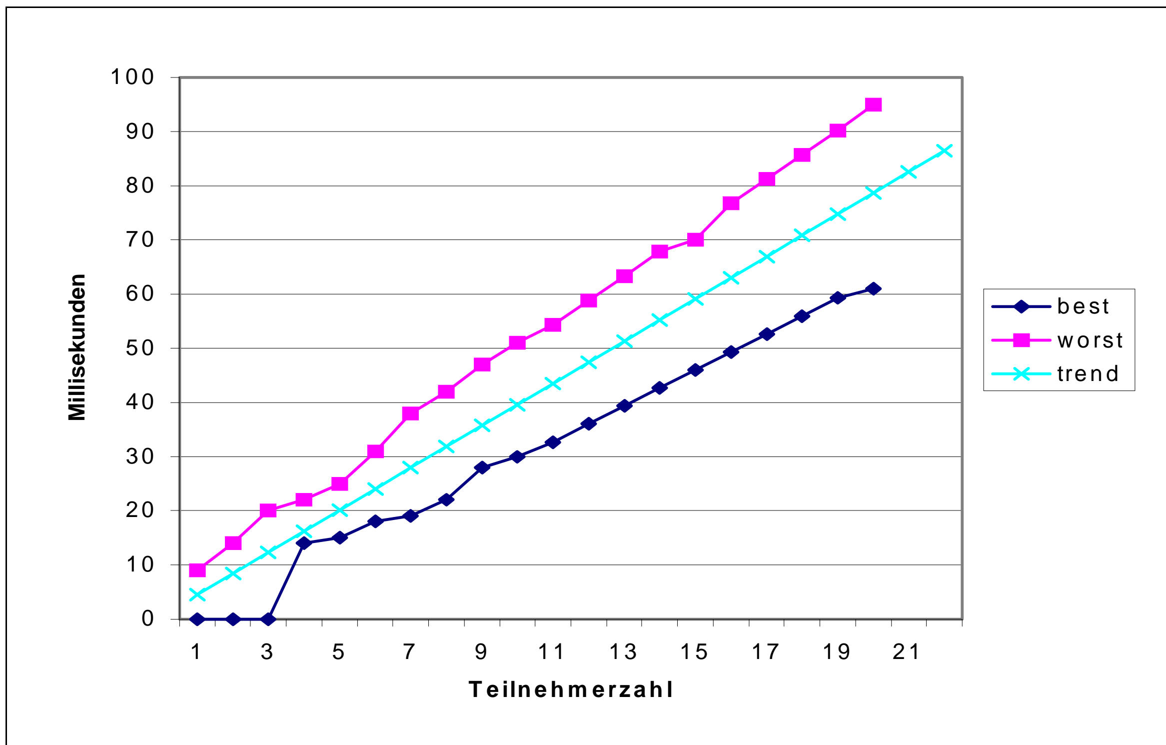


Abbildung 8-2: Diagramm der Performancemessung *Verzögerung*

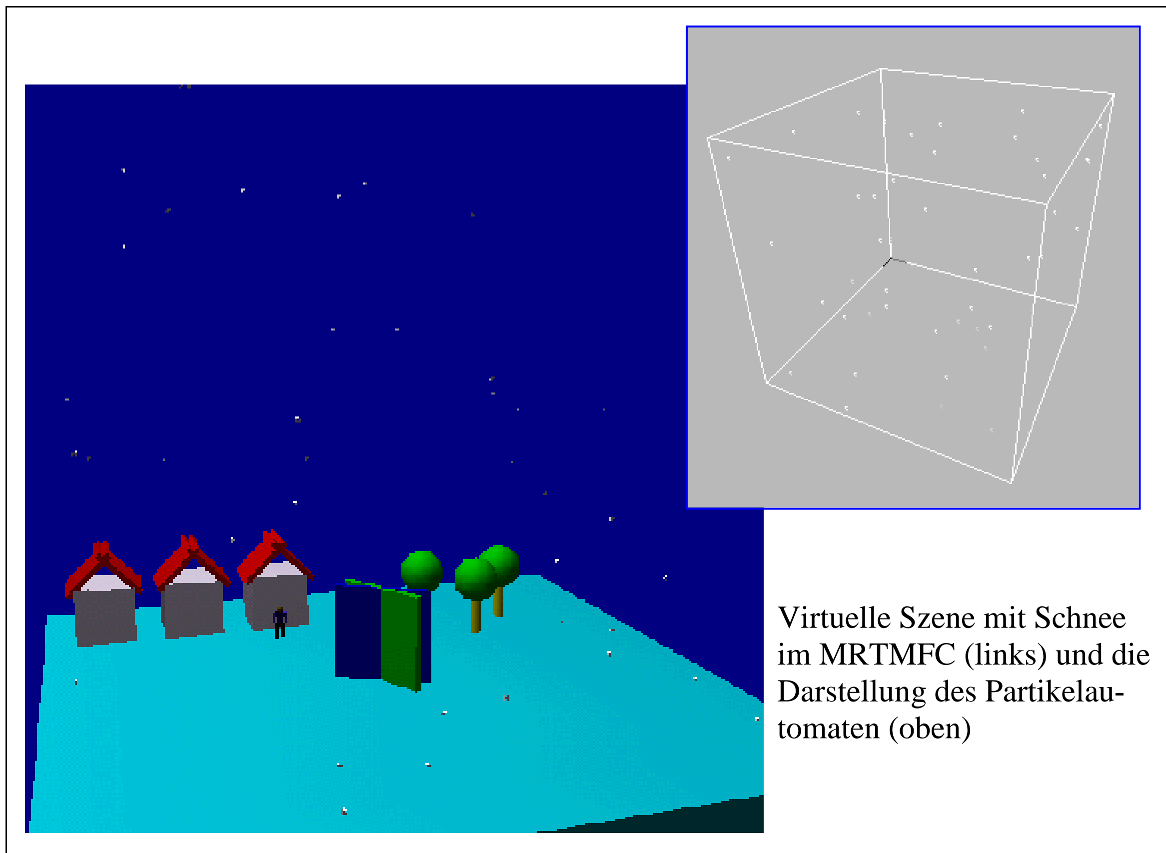
Für den Versand der Kameradaten hat sich gezeigt, daß die Anzahl der Teilnehmer im Grunde beliebig groß werden kann, ohne daß der Versand der Kameraposition einen Einfluß auf die Performance der übrigen Datenhaltung und der Anwendung nimmt. Dadurch, daß immer nur der letzte Kameradatensatz Verwendung findet und dieser auch mitten in einem Durchlauf geändert werden kann, ist ein Überlaufen nicht möglich. Es kommt zwar vor, daß innerhalb eines Durchlaufes mehrfach die Kameradaten wechseln, aber stoppt der Teilnehmer seine Bewegung, erhalten alle Teilnehmer diese letzte Kameraposition. Da die einzelnen Zwischenpositionen nicht unbedingt interessant sind, ist dies ein annehmbarer Kompromiß.

8.3 Praxistest

Während der Entwicklungs- und Testphase wurde die Fehlertoleranz der Datenhaltung erfolgreich belegt. Abstürzende Clientprogramme sorgten immer wieder dafür, daß der Konferenzserver diese Teilnehmer nach einem Timeout ausschließen mußte. Durch das Unterbrechen der Netzwerkverbindung zwischen den Rechnern wurde die Funktionalität des Protokolls getestet. Auf der partitionierten Seite, die den Server enthielt, war ein weiteres Ändern von Objekten möglich. Nach der Wiederherstellung der Verbindung wurden die abgeschnittenen Clienten wieder auf den aktuellen Stand gebracht.

Für eine Präsentation der Datenhaltungsschicht wurde eine kleine Demo entwickelt. Ein für die 2D-Visualisierung entwickeltes Partikelsystem [WS99] wurde zu einem Automaten erweitert. Dazu mußte es nur um wenig Zeilen Quellcode erweitert werden. Zunächst werden entsprechend der Anzahl der Partikel `t_Box`-Objekte erzeugt und über Referenzobjekte in einer Szene bei der Konferenz angemeldet. Danach sendet der Partikelautomat kontinuierlich für jedes Objekt eine `translateObject()`-Nachricht mit der aktuellen Partikelposition. Für dieses Testprogramm stellte der Partikelautomat eine Art Partikelregen her. Abbildung 8-3 zeigt die Visualisierung, in der die Partikel als weiße Boxen sichtbar auf eine Beispielszene regnen.

Es konnten 500 Partikel mit einer konstanten Visualisierung von 20 Bildern pro Sekunde zwischen zwei Pentium II 333 MHz Rechnern unter Windows NT 4.0 übertragen werden.



Virtuelle Szene mit Schnee im MRTMFC (links) und die Darstellung des Partikelautomaten (oben)

Abbildung 8-3: Praxistest mit Partikelautomat

8.4 Zusammenfassung der Meßergebnisse

Aus den Meßergebnissen läßt sich ableiten, daß für die Performance weniger die Größe der Szene oder die Anzahl der Teilnehmer entscheidend ist, sondern das Zusammentreffen vieler Teilnehmer, die viele wichtige Änderungen durchführen. Für die direkte Verbindung zwischen zwei Kommunikationspartnern ohne Server oder das CSCW mit kleinen Gruppen, kann es viele Änderungen bei wenigen Teilnehmern geben. Bei Präsentationen und virtuellen Vorlesungen gibt es viele Teilnehmer, aber nur wenige ändernde Moderatoren. Der MUI-MRT ist somit für den Einsatz der in Kapitel 2.5 spezifizierten Szenarien mit ausreichender Performance und guter Stabilität ausgestattet.

Für ORBacus ist in der nächsten Version ein Multicast-Plugin angekündigt. Dann könnten die unwichtigen Daten über Multicast versendet werden und nur noch die wichtigen protokolliert über den Broadcaster. Durch eine Aufteilung der Nachrichten und eine konsequente Trennung wichtiger und unwichtiger Nachrichten, dürfte sich die Skalierbarkeit weiter verbessern.

9 Ergebnisse und Ausblick

Im Rahmen dieser Arbeit wurde ausgehend vom MRT-VR, das eine Speziallösung für den Datentransport von MRT-Objekten über verschiedene leicht austauschbare Netzwerkprotokolle darstellt, eine Lösung für die generellen Herausforderungen und Probleme verteilter Anwendungen und Mehrbenutzersysteme entwickelt. Dabei ergaben sich Parallelen zu verteilten Datenbankmanagmentsystemen, die zur Übernahme von Konzepten und interessanten Ansätzen führten. Der Entwurf einer transparenten Datenhaltungsschicht, die alle aufgeworfenen Probleme mit der direkten Programmierung von Netzwerkprotokollen hätte lösen wollen, wäre nicht in dieser Arbeit umsetzbar gewesen.

Durch die Verwendung von CORBA bei der Entwicklung dieses verteilten Systems ergaben sich erhebliche Vereinfachungen. Insbesondere bleibt die Programmierung der Netzwerkschnittstellen für den Programmierer verborgen. Mit wenigen Funktionen kann ein Objekt im Netz anhand eines Namens gefunden und dann auf die gleiche Art genutzt werden, wie ein lokal vorhandenes Objekt.

Durch die Unterstützung heterogener Umgebungen mit der Definition von Objektschnittstellen über die systemunabhängige Sprache OMG IDL kann eine Kommunikation zwischen Anwendungen erstellt werden, für die es völlig transparent ist, welche Hardwareplattformen, Betriebssysteme oder Programmiersprachen verwendet werden.

Mit Hilfe der auf CORBA basierenden Datenhaltung ist es also möglich Programme zur Berechnung von Daten auf einer speziellen Plattform laufen zu lassen, die entweder besonders optimiert ist, oder aber nicht ohne weiteres portierbar wäre. Über die Datenhaltungsschicht kann die Visualisierung der Daten dann auf jedem beliebigen Rechner einem oder mehreren Anwendern präsentiert werden.

Somit stellt diese Architektur ein sehr leistungsfähiges System zur Entwicklung verteilter Systeme dar, das auf zukunftsweisende Techniken setzt.

Die Verwendung der aus der Datenbanktheorie übernommenen Konzepte haben die Datenhaltungsschicht zusätzlich einfach, robust und zuverlässig werden lassen. So wurde die Datenhaltung tolerant gegenüber Fehlern der Benutzer oder der darüberliegenden Anwendungsprogramme. Es kam in der Entwicklungsphase häufiger vor, daß ein noch nicht ausgereifter Automat fehlerhaft war und abstürzte, ohne daß die restliche Konferenz davon beeinträchtigt worden wäre. Spätestens bei Überschreitung der Timeoutzeit wurde der Automat vom Server aus der Teilnehmerliste entfernt.

In kleinen Gruppen wurde eine akzeptable Performance bei großer Stabilität und Sicherheit erreicht. Das Verhalten in großen Gruppen ist noch zu erproben.

Ein weiteres gestecktes Ziel war die einfache Handhabung der Schnittstelle und die flexible Steuerung einer Konferenz. Sieht man sich die Codes der Automaten an, zeigt sich, daß diese alle sehr kompakt und kurz gehalten werden konnten. Mit wenigen Zeilen kann ein Programm seine Daten in eine Konferenzszene einbringen und steuern.

Zusammengefaßt ergeben sich diese Ergebnisse:

- stabile und robuste Datenhaltung
- minimale Änderungen an MRT-Basisklassen
- kompakte und einfach zu handhabende Schnittstelle
- flexible Konfiguration der Konferenz-Clients
- hoher Grad an Transparenz
- vernünftige Performance bei kleinen Konferenzen

Im Verlauf der Entwicklung und Implementierung der Datenhaltungsschicht ergaben sich eine Reihe von Erkenntnissen, die für den weiteren Ausbau interessante und erweiterungswürdige Ansätze bieten. Im Folgenden werden diese kurz diskutiert:

- Da basierend auf dem MRT kein Editor existiert, konnten die Änderungsoperationen nur mittels Testprogrammen und Automaten genutzt werden. Um eine echte Interaktion mit anderen Teilnehmern zu ermöglichen, ist die Entwicklung eines Editors, der zunächst nur wenige rudimentäre Operationen, wie das Transformieren von Objekten und die Shaderänderung analog zu den von der Datenhaltung explizit angebotenen Funktionen, zu unterstützen braucht, eine sinnvolle Erweiterung.

- Um mit größeren Teilnehmerzahlen arbeiten zu können und die Netzbelastung zu reduzieren, bietet sich in einer verteilten Umgebung an, mehrere Server zu verwenden, die untereinander größere Datenmengen zusammenfassen und weiterleiten. Jeder Server bedient dann jeweils eine disjunkte Gruppe von Teilnehmern, die in seiner Nähe liegen. Die Nähe eines Rechners zum anderen könnte anhand von Übertragungsverzögerungen gemessen werden, das heißt je nach Datentransferzeiten, wird die Verknüpfung der Clienten so ausbalanciert, daß die effizienteste Netzwerkstruktur aufgebaut wird. Durch das Migrieren des Serverdienstes kann dann auch die Anzahl der Server angepaßt werden.

- Ein Bereich, der bei weiterer und häufigerer Nutzung nicht außer Acht gelassen werden darf, ist die Sicherheit. Wie lassen sich Übertragungen und Konferenz vor unberechtigten Zugriffen schützen? Mit einem entsprechenden CORBA-Browser lassen sich die Schnittstellen von CORBA-Objekten ausspionieren und dann kann jeder die Methoden aufrufen. Da es keine Möglichkeit gibt zu überprüfen, von wem der Aufruf stammt, müssen eigene Lösungen oder PlugIns der Hersteller gefunden werden, die dem einen Riegel vorschieben.

- Einige der Funktionen sind mit naiven Algorithmen implementiert worden und bieten noch Optimierungsmöglichkeiten. Insbesondere der Broadcaster erlaubt viele Ansatzpunkte für Beschleunigungen. Beispielsweise könnten mehrfache Änderungen des gleichen Objektes innerhalb des Protokolls zusammengefaßt werden.

- Der 2-Versionen-Ansatz braucht nur bei Clienten vollständig aufrechterhalten zu werden, die selber Änderungen tätigen können. Teilnehmer, die entweder keine ausreichenden Rechte besitzen oder keine Änderungen durchführen wollen, können nach der Synchronisierung der beiden Versionen die Parameterlisten aus der Objektliste enfernen und sie lediglich als Puffer für eingehende Änderungen nutzen. Das sollte den Speicherverbrauch der doppelten Szenenhaltung nahezu halbieren.

- In die gleiche Richtung geht die Verwendung von Sichtbereichen oder Auren um einen Avatar herum. Nur Objekte innerhalb einer Aura kann der Teilnehmer sehen, alles andere wird ohnehin durch das Clipping entfernt. Somit sind Änderungen an Objekten ausserhalb der Aura nicht interessant und müssen nicht übertragen werden. Der Server könnte leicht anhand der Kamerapositionen der Teilnehmer die Aura berechnen, es müssen allerdings passende Strategien für das Nachladen von Szenenbereichen entwickelt werden, wenn die Aura sich verschiebt. Auch die Konsistenz der Daten muß gesondert beachtet werden. Es muß immer garantiert sein, daß die gesamte Szene erhalten bleibt. Beispielsweise darf beim Gang durch ein virtuelles Gebäude, bei dem alle Teilnehmer in einem Raum sind, der Rest des Gebäudes nicht verschwinden. Hierfür bietet sich eine Erweiterung des Mehrversionenkonzeptes an.
- Um den Verwaltungsaufwand zu verringern, könnten unwichtige Daten über Multicast verteilt werden, z.B. der Versand der Kamerapositionen. Es ist ohnehin nicht wichtig, genau zu wissen, wo sich die anderen Teilnehmer aufhalten. Ein in unregelmäßigen Abständen stattfindende Aktualisierung reicht hierfür völlig aus. Auch der Verlust einiger dieser Informationen ist nicht relevant. So braucht der Server nur die relevanten und wichtigen Datenströme, wie z.B. die Verteilung von Zugriffsrechten, oder die Übermittlung von signifikanten Änderungen, zu verteilen.

Anhang

```
class t_ConfInterface
{
   char* name();
   void name(const char* _name);
   char* nick();
   void nick(const char* _nick);
   char* email();
   void email(const char* _email);
   char* comment();
   void comment(const char* _email);
   t_ID uid();
   const char* host();
   const char* iiop();

   t_Flags updates();

   t_Flags getFlags();
   void setFlags(t_Flags _flags);
   void addFlags(t_Flags _flags);
   void removeFlags(t_Flags _flags);

   char* confname();

   bool init(const char* _confname,
             t_FullScenePtr _fullScene,
             t_ScenePtr _baseScene,
             t_ScenePtr _avatarScene,
             t_RefObjectPtr _avatar = NULL,
             t_Rights _generalRight = R_FULL);
   bool initRemote(const char* _confname,
                   const char* _iiop,
                   t_FullScenePtr _fullScene,
                   t_ScenePtr _baseScene,
                   t_ScenePtr _avatarScene,
                   t_RefObjectPtr _avatar = NULL,
                   t_Rights _generalRight = R_FULL);
   bool loginIIOP(const char* _iiop,
                  const char* _name,
                  t_FullScenePtr _fullScene = NULL,
                  t_ScenePtr _baseScene = NULL,
                  t_ScenePtr _avatarScene = NULL,
                  t_RefObjectPtr _avatar = NULL);

   bool logout();
   bool is_running();
   t_UserDescList* getUserList();
   t_UserDescription* getWithNick(const char* _nick);
   t_MUIStatistic getStatistic();

   void say(const char* _text);
   t_StringSeq* getChatBuffer();

   void setCamera(t_CameraPtr& _cam);
   t_IDList* getCameraIDs();
   t_CameraPtr& getCamera(t_ID _uid);
```

```
   t_RefObjectPtr getAvatar(t_ID _uid);
   t_ID getAvatarUser(t_ObjectPtr _obj);

   t_RtiRefPtr getObjByName(const char* _objName);
   char* getLastError();
   bool updateWorld();
   bool reconstructWorld();

   bool openTransaction();
   bool closeTransaction();
   bool abortTransaction();

   bool registerObject(t_SurfaceObjectPtr& _obj,
                       t_ObjectPtr _container = NULL);
   bool registerScene (t_ScenePtr&         _scene,
                       t_ObjectPtr _container = NULL);
   bool registerLight (t_LightPtr&         _light);
   bool registerShader(t_ShaderPtr&        _shader,
                       t_SurfaceObjectPtr _container = NULL);
   bool registerAvatar(t_RefObjectPtr& _avatar);
   bool deleteObject(t_SurfaceObjectPtr& _obj);
   bool deleteScene (t_ScenePtr&  _scene);
   bool deleteLight (t_LightPtr&  _light);
   bool deleteShader(t_ShaderPtr& _shader);

   bool lockElement(t_RtiRefPtr& _rrp);
   bool unlockElement(t_RtiRefPtr& _rrp);
   bool setMarkShader(t_SurfaceShaderPtr _shader);
   bool markElement(t_SurfaceObjectPtr& _sop);
   bool unmarkElement(t_SurfaceObjectPtr& _sop);
   bool updatesImportant(bool _important = true);
   bool translateObject(t_SurfaceObjectPtr& _obj,
                        t_3DVector& _mat,
                        bool _absolut = false);
   bool transformObject(t_SurfaceObjectPtr& _obj,
                        t_4x3Matrix& _mat,
                        bool _absolut = false);
   bool exchangeShader (t_SurfaceObjectPtr& _obj,
                        t_ShaderPtr& _shader);
}
```

Abbildung 0-1: Definition der Klasse `t_ConfInterface`

```
interface t_Conference
{
   void updateUser(in t_UserDescription _userDesc);
   void setGenRight(in t_ID _uid,
                    in t_Rights _generalRight)
                   raises(AccessDenied);
   void setUserRight(in t_ID _uid,
                     in t_ID _tid,
                     in t_Rights _right)
                    raises(AccessDenied);
   void say(in t_ID _uid, in string _text);
   boolean lockElement(in t_ID _uid,
                       in t_ID _oid,
                       in boolean mark);
   void unlockElement(in t_ID _uid,
                      in t_ID _oid,
                      in boolean mark);
   void sendWorld(in t_ID _uid,
                  in t_ParamListSeq _world);
   void sendFullScene(in t_ID _uid,
                      in t_AnySeq _fscene);
   void registerElement(in t_ID _uid,
                        in t_ParamList _object);
   void deleteElement(in t_ID _uid,
                      in t_ID _oid);
   void translateElement(in t_ID _uid,
                         in t_ID _oid,
                         in t_3DVector_Data _v);
   void transformElement(in t_ID _uid,
                         in t_ID _oid,
                         in t_4x3Matrix_Data _mat);
   void exchangeShader(in t_ID _uid,
                       in t_ID _oid,
                       in t_ID _sid);
   t_IDList getNewOIDs(in short _num);

   t_Conference getActiveServer();
   t_FullUserList registerMe(in t_Conference _user,
                             in string _name,
                             in string _nick,
                             in string _host,
                             in string _email)
                            raises(AlreadyRegistered,
                                   NotActiveServer,
                                   AccessDenied);
   void unregisterMe(in t_ID _uid)
                    raises(NotActiveServer);
}
```

Abbildung 0-2: OMG IDL Definition der Schnittstelle des Konferenzobjektes

```
class t_Conference_impl : public t_Conference_skel
{
   virtual char* name();
   virtual void  name(const char* _name);
   virtual char* nick();
   virtual void  nick(const char* _nick);
   virtual char* email();
   virtual void  email(const char* _email);
   virtual char* comment();
   virtual void  comment(const char* _comment);
   virtual t_ID  uid();
   virtual char* host();
   virtual char* iiop();
   virtual t_ID  avatarOID();
   virtual void  avatarOID(t_ID _oid);
   virtual void setFlags(t_Flags _flags);
   virtual t_Flags getFlags();
   virtual void addFlags(t_Flags _flags);
   virtual void removeFlags(t_Flags _flags);
   virtual t_Flags updates();
   virtual char* confname();
   virtual void init(const char* _confname,
                     t_Rights _generalRight);
   virtual void initRemote(const char* _confname,
                           const char* _iiop,
                           t_Rights _generalRight);
   virtual void loginIIOP(const char* _iiop,
                          const char* _name);
   virtual void logout();
   virtual CORBA_Boolean is_running();
   virtual CORBA_Boolean waitUntilReady();
   virtual t_UserDescList* getUserList();
   virtual t_UserDescription*
              getWithNick(const char* _nick);
   virtual t_UserDescription* getUserDesc(t_ID _uid);
   virtual void camera(const t_ParamList& _camera,
                       CORBA_UShort _counter);
   virtual t_IDList* getCameraIDs();
   virtual t_ParamList* getCamera(t_ID _uid);
   virtual t_StringSeq* getChatBuffer();
   virtual CORBA_Boolean locked(t_ID _oid);
   virtual CORBA_Boolean openTransaction();
   virtual CORBA_Boolean closeTransaction();
   virtual CORBA_Boolean abortTransaction();
   virtual t_MUIStatistic getStatistic();
}
```

Abbildung 0-3: intern Konferenz Objekt

```
class t_ObjectList : public JTCMonitor {
   void cmdInsertElement(const t_ParamList& _params);
   void cmdTransformElement(t_ID _oid,
                            const t_4x3Matrix_Data& _mat);
   void cmdExchangeShader(t_ID _oid, t_ID _sid);
   void cmdTransformAvatar(t_ID _avatarOID,
                           const t_ParamList& _camera);
   void cmdDeleteElement(t_ID _oid);
   void cmdMarkElement  (t_ID _oid);
   void cmdUnmarkElement(t_ID _oid);
   void cmdWorld(const t_ParamListSeq& _world);
   void cmdFullScene(const t_AnySeq& _fscene);
   t_ParamListSeq* getWorld();
   t_ParamList* getFullScene();
   t_Flags updates();
   bool initWorld(t_Conference_impl_ptr _conf,
                  t_FullScenePtr _fs,
                  t_ScenePtr _baseScene,
                  t_ScenePtr _avatarScene);
   bool prepareWorld(t_Conference_impl_ptr _conf,
                     t_FullScenePtr _fs,
                     t_ScenePtr _baseScene = NULL,
                     t_ScenePtr _avatarScene = NULL);
   bool updateWorld();
   bool reconstructWorld();
   bool registerObject(t_SurfaceObjectPtr& _obj,
                       t_ObjectPtr _container,
                       const bool  _updateElement,
                       const bool  _avatar = false);
   bool registerScene (t_ScenePtr& _scene,
                       t_ObjectPtr _container,
                       const bool  _updateElement,
                       const bool  _avatar = false);
   bool registerAvatar(t_RefObjectPtr&     _refObj);
   bool registerLights(t_LightsPtr&        _lights);
   bool registerLight (t_LightPtr&         _light);
   int registerShader (t_ShaderPtr& _shader,
                       t_SurfaceObjectPtr _container,
                       const bool _updateElement,
                       const bool _avatar = false);
   bool deleteShader(t_ShaderPtr& _shader,
                     const bool _updateElement);
   bool deleteObject(t_SurfaceObjectPtr& _obj,
                     const bool _updateElement);
   bool deleteScene (t_ScenePtr&  _scene,
                     const bool _updateElement);
   bool deleteLight (t_LightPtr&  _light);
   bool translateObject(t_SurfaceObjectPtr& _obj,
                        t_3DVector& _v, bool absolut);
   bool transformObject(t_SurfaceObjectPtr& _obj,
                        t_4x3Matrix& _mat, bool absolut);
   bool exchangeShader (t_SurfaceObjectPtr& _obj,
                        t_ShaderPtr& _shader);
   bool setMarkShader  (t_SurfaceShaderPtr _shader);
   bool markObject     (t_SurfaceObjectPtr& _obj);
   bool unmarkObject   (t_SurfaceObjectPtr& _obj);
   t_ID getID(t_RtiRefPtr _objPtr);
   t_RtiRefPtr getObj(t_ID _oid);
   t_ID getCreator(t_RtiRefPtr _objPtr, bool _avatar); }
```

Abbildung 0-4: `t_ObjectList` Header

```
interface t_MsgHandler
{
      void cmdUpToDate();
      void cmdJoin(in t_Conference _user,
                   in t_ID _uid,
                   in string _name,
                   in string _nick,
                   in string _host,
                   in string _email);
      void cmdLeave(in t_ID _uid);
      void cmdUpdate(in t_UserDescription _userDesc);
      void cmdPrint(in string _text);
      void cmdCamera(in t_ID _uid,
                     in t_ParamList _camera,
                     in unsigned short _counter);
      void cmdElement(in t_ParamList _object);
      void cmdTranslate(in t_ID _oid,
                        in t_3Dvector_Data _v);
      void cmdTransform(in t_ID _oid,
                        in t_4x3Matrix_Data _mat);
      void cmdExchangeShader(in t_ID _oid, in t_ID _sid);
      void cmdWorld(in t_ParamListSeq _world);
      void cmdFullScene(in t_AnySeq _fscene);
      void cmdDelete(in t_ID _oid);
      void cmdLock(in t_ID _uid,
                   in t_ID _oid,
                   in boolean _mark);
      void cmdUnlock(in t_ID _uid,
                     in t_ID _oid,
                     in boolean _mark);
      void cmdCallList(in t_CallList _callList);
      void cmdQuit();
}
```

Abbildung 0-5: MessageHandler OMG IDL Interface

Abbildungsverzeichnis

Abbildung 3-1: Eine Datenbank mit den Relationen Kunde und Auftrag 11
Abbildung 3-2: ER-Diagramm der Beispieldatenbank aus Abbildung 3-1 12
Abbildung 3-3: ER-Diagramm der MUI-MRT Daten 12
Abbildung 4-1: Das OMA Referenz-Modell (Quelle OMG)... 21
Abbildung 4-2: CORBA ORB Architektur 22
Abbildung 4-3: OMG IDL Interface des math-Objektes 24
Abbildung 4-4: Beispiel einer OMG IDL union-Definition 25
Abbildung 4-5: Client Anwendung für das math-Objekt... 26
Abbildung 4-6: Server für das math-Objekt... 26
Abbildung 4-7: Aufruf von `seta()` über das DII 27
Abbildung 6-1: Erweiterung der MRT-Basisklassen 31
Abbildung 6-2: Die wichtigsten Klassen der Szenenhierarchie des MRT...................... 32
Abbildung 6-3: Schichtenmodell der MUI-MRT Datenhaltung 36
Abbildung 7-1: Definition der Klasse `t_MUI` 38
Abbildung 7-2: Minimaler Code eines dedizierten Konferenzservers............................ 38
Abbildung 7-3: Initiierung einer Konferenz auf einem dedizierten Server 39
Abbildung 7-4: MUI-MRT erwartet diese Szenenhierarchie.. 40
Abbildung 7-5: Code eines kleinen Automaten 41
Abbildung 7-6: Rekursiver Registrierungsverlauf eine neuen Szene 42
Abbildung 7-7: Funktionen zum Lesen und Schreiben des Benutzernamens................ 46
Abbildung 7-8: Definition einer Nachricht des MessageHandlers 49
Abbildung 7-9: OMG IDL Definition der Schnittstelle von t_ConfAdmin.................... 50
Abbildung 7-10: Ereignisumfeld der Objektliste 53
Abbildung 8-1: Diagramm der Performancemessung *Konferenzanmeldung* 57
Abbildung 8-2: Diagramm der Performancemessung *Verzögerung* 58
Abbildung 8-3: Praxistest mit Partikelautomat 59
Abbildung 0-1: Definition der Klasse `t_ConfInterface` 65
Abbildung 0-2: OMG IDL Definition der Schnittstelle des Konferenzobjektes 66
Abbildung 0-3: intern Konferenz Objekt 67
Abbildung 0-4: `t_ObjectList` Header 68
Abbildung 0-5: MessageHandler OMG IDL Interface 69

Tabellenverzeichnis

Tabelle 7-1: Datenverzeichnis der öffentlichen Teilnehmerdaten 46
Tabelle 7-2: Datenverzeichnis der internen Teilnehmerdaten 47
Tabelle 7-3: Datenverzeichnis des 2-Versions-Datensatzes 48
Tabelle 7-4: Datenverzeichnis der Parameterliste eines MRT-Objektes 48
Tabelle 7-5: Datenverzeichnis eines CallList struct... 49
Tabelle 7-6: Datenverzeichnis des Protokolls.. 49
Tabelle 8-1: Meßergebnisse für Konferenzanmeldung in Millisekunden....................... 57
Tabelle 8-2: Verzögerung von Nachrichten für steigende Teilnehmerzahlen 58

Literaturverzeichnis

- [AD76] Alsberg, P.A., Day, J.D.: "A Principle of Resilient Sharing of Alternative Strategies for Dealing Deadlocks in Database Management Systems." IEEE Trans. on Software Engineering 13 (12), 1348-1363, 1987
- [AS97] Sayegh, A.M.: "CORBA: Standard, Spezifikationen, Entwicklung". 1. Auflage - Köln: O'Reilly Verlag, 1997
- [BHG87] Bernstein, P.A., Hadzilacos, V. Goodman, N.: "Concurrency Control and Recovery in Database Systems." Addison Wesley, 1987
- [BFS97] Broll, W., Fechter, J., Schick, D.: "Unterstützung von Multiuser-VR mit VRML", 1997
- [CP76] Chen P P, "The Entity-Relationship Model - Toward a Unified View of Data", ACM Transactions on Database Systems, Vol. 1, No 1, March 1976, pp 9-36
- [Da90] Date, C.J.: "An Introduction to Database Systems." 5th edition. Addison Wesley 1990
- [DI] Duden "Informatik": "Ein Sachlexikon für Studium und Praxis" hrsg. vom Lektorrat d. BI-Wiss.-Verl. Endesser H., Dudenverlag 1988
- [EGLT76] Eswaran, K.P., Gray, J.N., Lorie, R.A., Traiger, I.L.: "The notions of consistency and predicate locks in a database system." Comm. ACM 19 (11), 624-633, 1976
- [Fel96] Fellner, D.W.: "MRT - a teaching and research plattform for 3D image synthesis", IEEE CG&A 16(3), May 1996
- [Gr78] Gray, J.: "Notes on Data Base Operation Systems." in: "Operating Systems - An Advanced Course", Springer-Verlag, Lecture Notes in Computer Science 60, 393-481, 1978
- [Gr81] Gray, J.: "The Transaction Concept: Virtues and Limitations." Proc. 7th Int. Conf. on Very Large Data Bases, 144-154, 1981
- [Gr86] Gray, J.N.: "An Approach to Decentralized Computer Systems." IEEE Trans. on Software Engineering 12 (6) 684-692, 1986
- [GW90] Gleißner, W., Grimm, R., Herda, S., Isselhorst, H.: "Manipulation in Rechnern und Netzen" Addison-Wesley, Bonn, 1990
- [HA97a] Hopp, A.: "Zwischenbericht zum MRT-VR", AAA'97 Workshop, Darmstadt

- [HA97b] Hopp. A.: "MRT-VR Verteilte dynamische Multi-User-3D-Umgebung"
- [Hä88a] Härder, T.: "Transaktionskonzept und Fehlertoleranz in DB/DC-Systemen", Handbuch der modernen Datenverarbeitung, Heft 144, Forkel-Verlag, Nov. 1988
- [HR83] Härder, T. Reuter, A.: "Principles of Transaction-Oriented Database Recovery." ACM Computing Surveys 15 (4), 287-317, 1983
- [ICQ] Mirabilis ICQ ("I Seek You") WebSite: http://www.icq.com/
- [iX1] Artikel "Objektiv betrachtet" in der iX 08/1997 S. 44-51:
  Verteilte Objekte: DCOM versus CORBA
- [iX2] Artikel "Teile und herrsche" in der iX 08/1997 S. 52-57:
  Microsoft DCOM für Unix
- [iX3] Artikel: "Schritt aus dem Abgrund" in der iX 9/97 S. 102-103[MC]
  McCall Unternehmensberatung: "DCOM vs. CORBA"
  http://www.mcgrp.com/dcomvscorba.htm
- [MJ95] McCane, S., Jacobson, V.: vic: A Flexible Framework for Packet Video, ACM Multimedia '95: http://www-nrg.ee.lbl.gov/vic
- [MS1] Microsoft Netmeeting WebSite: http://www.microsoft.com/netmeeting/
- [MS2] Microsoft DCOM WebSite: http://www.microsoft.com/dcom/
- [OB] Laukien, M.: "Dokumentation ORBacus Version 3.1.1" Object-Oriented Concepts, Inc. http://www.ooc.com/
- [OMG94] "The Common Object Request Broker: Architecture and Specification" Object Management Group - Document 1994 http://www.omg.com/
- [OMG95a] "Common Object Request Broker Architecture", OMG, July, 1995
- [OMG95b] "Common Object Services Specification", OMG 95-3-31, 1995
- [Re87] Reuter, A.: "Maßnahmen zur Wahrung von Sicherheits- und Integritätsbedingungen". in Lockermann, P.C. Schmidt, J.W. (Hrsg.): Datenbank-Handbuch. Springerverlag, 1987
- [RE94] Rahm, E.: "Mehrrechner-Datenbanksysteme: Grundlagen der verteilten und parallelen Datenbankverarbeitung", Bonn; Paris; Reading, Mass. [u.a.o]: Addison-Wesley, 1994
- [St79] Stonebraker, M.: "Concurrency Control and Consistency of Multiple Copies" in Distributed Ingres. IEEE Trans. on Software Engineering 5 (3), 188-194, 1979

- [SW85] Steinbauer, D., Wedekind, H.: "Integritätsaspekte in Datenbanksystemen." Informatik-Spektrum 8 (2), 60-68, 1985
- [VD] Meyer-Wegener, K.: Skript zur Vorlesung „Verteilte Datenhaltung“ Technischen Universität Dresden
- [We88] Weikum, G.: "Transaktionen in Datenbanksystemen". Addison-Wesley, Bonn, 1988
- [WS99] Wolfrum, S.: geplante Diplomarbeit über Partikelautomaten am Institut für Computergrafik der TU Braunschweig