Universität Leipzig
Fakultät für Mathematik und
Informatik
Institut für Informatik

Thema: Entwurf, Implementierung und Bewertung eines Dienstes für die verteilte Bearbeitung von Dokumenten unter Verwendung von CORBA und XML

Diplomarbeit

Uwe Voigt

09. Februar 2000

Die Diplomarbeit untersucht die Möglichkeiten eines Architekturentwurfs und Implementierungen von verteilten Software-Komponenten zum gemeinsamen Bearbeiten von Dokumenten. Beim gemeinsamen Zugriff auf Ressourcen sind vielfältige Problematiken wie Zugriffsrechte, gemeinsame Lese- bzw. Schreibzugriffe, die Auswahl einer geeigneten Datenstruktur und Datenbeschreibung, Konsistenz der Daten und nicht zuletzt die Leistungsfähigkeit der verwendeten Übertragungsprotokolle zu beachten. Diesen Problematiken sind in der heutigen Praxis eine Anzahl von Strategien gewidmet. Von der **O**bject **M**anagement **G**roup (OMG) sind eine Reihe von Spezifikationen erarbeitet worden. Eine von ihnen ist die **C**ommon **O**bject **R**equest **B**roker **A**rchitecture (CORBA), eine Architektur für Software, die zwischen Anwendungs- und Systemsoftware angesiedelt ist, die sogenannte Middleware. CORBA spezifiziert eine objektorientierten Architektur für die Kommunikation von Komponenten in heterogenen Netzwerken. Der Beschreibung von Datenstrukturen, die über Netze verbreitet werden sollen, widmet sich ein weiteres Konsortium, das **W**orld **W**ide **W**eb **C**onsortium (W3C). Fußend auf den Bemühungen dieses Interessenzusammenschlusses ist die mächtige Metasprache E**x**tensible **M**arkup **L**anguage (XML) entstanden. Das Dokumentenbeschreibungsmodell, auf dem XML basiert ist das **D**ocument **O**bject **M**odel (DOM). Diese neuen Spezifikationen sind die Werkzeuge, um leistungsstarke und skalierbare Implementierungen zu ermöglichen. In der Arbeit werden die Stärken der genannten Technologien herausgearbeitet und verwendet, um die Skalierbarkeit und Performanz der bereitgestellten verteilten Anwendung zu erhöhen.

This thesis examines the possibilities of an architectural design and the implementation of distributed software components for shared processing of documents. Common use of resources contains various problems as rights for read or write accesses, the selection of a suitable structure and description of the data to consider consistency and the efficiency of the used transmission protocols. In today's practice a number of strategies are dedicated to these problems. A set of specifications was compiled by the **O**bject **M**anagement **G**roup (OMG). One of them is the **C**ommon **O**bject **R**equest **B**roker **A**rchitecture (CORBA), an architecture for software which is settled between application and operational software, the so-called middleware. CORBA is an object-oriented architecture for communication of components in heterogeneous networks. A further consortium, the **W**orld **W**ide **W**eb **C**onsortium (W3C) dedicates itself to the description of data structures which are to be spread over networks. Being based on the efforts of this interest union the powerful meta language E**x**tensible **M**arkup **L**anguage (XML) was developed. The document descriptive model based on the XML is the **D**ocument **O**bject **M**odel (DOM). These new specifications are the tools in order to enable high performance and scalable implementations. The thesis focuses on the strength of the mentioned technologies. They will be used to increase scalability and performance of the distributed application that was developed for the duration of the thesis.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einführung

Das Schreiben von Dokumenten durch mehrere Autoren, das Publizieren auf verschiedenen Medien oder die Verwaltung und Koordination von Projekten mit vielen Mitarbeitern sind komplexe Aufgabenbereiche, die auf Seiten von Rechnersystemen und Kommunikationsnetzwerken einen hohen Grad an Inhomogenität beinhalten können. Unterschiedliche Hardware- und Softwareprodukte sind oft durch spezifische Datenformate oder Kommunikationsprotokolle in ihrer Integrierbarkeit in größere Systemverbunde beschränkt. Vor allem die Realisierung von verteilten Systemen, in denen Clients die Leistungsangebote von Servern anderer Systemarchitektur, Implementierungssprache oder Kommunikationsumgebung nutzen, erfordert offene und systemübergreifende Standards und Technologien. Die Beschränkung auf Industriestandards spezifischer Hersteller bedeutet hierbei allzu oft eine Einschränkung der Interoperabilität, also der Möglichkeiten, Produkte unterschiedlicher Herkunft im Einklang zu verwenden.

CORBA etabliert einen offenen Standard, der sich gegenüber vergleichbaren Client/Server-Architekturen durch eine Vielzahl von Eigenschaften hervorhebt. Eine Gegenüberstellung erfolgt in Abschnitt 2.7.

(1) CORBA integriert eine große Anzahl an standardisierten Diensten, wie den Naming-, Trading-, Event-, Security- oder Transaction-Service. [OMG98d]

(2) Die Funktionalität eines Servers wird durch ein Interface beschrieben, das in der von Programmiersprachen abgekoppelten, ausdrucksstarken **I**nterface **D**efinition **L**anguage (IDL) (siehe Abschnitt 2.1) verfaßt wird.

(3) CORBA folgt dem Designprinzip der Objektorientierung. IDL verfügt demgemäß über die Möglichkeit der Vererbung, der Bestim-

mung von Attributen sowie der Deklaration nutzerspezifischer Exceptions.

(4) CORBA spezifiziert des Interface Repository (siehe Abschnitt 2.3.2), das es ermöglicht, Typüberprüfungen dynamisch, d.h. zur Laufzeit vorzunehmen.

(5) CORBA verfügt über standardisierte Abbildungen von IDL in mehrere Programmiersprachen, sogenannte Language-Mappings.

(6) Das grundlegende Kommunikationsprotokoll IIOP (siehe Abschnitt 2.5), das Bestandteil der CORBA-Spezifikation ist, garantiert die Interoperabilität der Kommunikation.

Generell begründet sich der Vorteil höherer Architekturen in der Abstraktion von zugrundeliegenden Schichten, wobei jede Schicht ihre technischen Details weitestgehend verbirgt, man spricht hier von der Transparenz dieser Details für den Anwender. Viele Anbieter von Produkten der CORBA-Technologie implementieren die grundlegenden Kommunikationsfunktionen mit Hilfe von TCP/IP-Kommunikationsendpunkten, den Sockets. Für den Anwendungsprogrammierer ergibt sich aber über System- und Sprachgrenzen hinweg eine einheitlichen Sicht auf Daten und Operationen. Er wird zu keinem Zeitpunkt gezwungen sein, über Socketprogrammierung nachzudenken. Das Designparadigma Objektorientierung bietet neben Vererbung auch die Eigenschaften der Polymorphie und des Überladens von Funktionen[1]. Objektorientierte Entwürfe bzw. Programme zeichnen sich durch geringere Fehleranfälligkeit, Wiederverwendbarkeit und dadurch bessere Wartbarkeit aus. Diese Eigenschaften resultieren oft in einer Reduktion von Design- und Entwicklungskosten.

Der Entwurf einer Architektur zur gemeinsamen Bearbeitung von Dokumenten birgt neben den oben genannten auch das Problem der Serverbelastung. Das Anliegen von Client/Server-Systemen ist die Dezentralisierung und die Verteilung der Aufgaben von Servern bzw. deren Delegierung an Clients. Durch CORBA ist es darüber hinaus möglich,

[1] Polymorphie umschreibt die Möglichkeit, gleiche Funktionen auf verschiedene Implementierungen von Objekten anzuwenden. Überladen beschreibt die Definition verschiedener gleicher Bezeichnung.

diese Aufgaben auf verschiedene Serverinstanzen, die auf unterschiedlichen Maschinen residieren, zu verteilen. Diese Verteilung kann dynamisch erfolgen und bleibt vom Client unbemerkt.

Weiterhin ist die Wahl des Datenrepräsentationformats von großer Bedeutung. Die Speicherung von Dokument-Komponenten empfiehlt sich aus Speicherplatzgründen eher in der Datensemantik angepaßten Formaten. Es ist nicht sinnvoll, Grafikobjekte, die Teil von Dokumenten sein können, von ihrem spezifischen Format in ein generalisiertes Format umzuwandeln. Wenn menschenlesbare - also Text - Daten, die außerdem maschinell manipulierbar sein müssen verarbeitet werden sollen, dann stellen universelle Formate eine bessere Wahl dar. Die Sprache XML ist hierfür vor allem ihrer verhältnismäßig einfachen und mächtigen Struktur wegen besonders geeignet. Mit Hilfe des **A**pplication **P**rogramming **I**nterfaces (API), das durch das **D**ocument **O**bject **M**odel (DOM) bereitgestellt wird, ist der Zugriff auf Dokumente bzw. die Manipulation von Dokumenten standardisiert. In der Arbeit wird auf die durch die Komplexität des DOM auftretenden Probleme eingegangen und ein Lösungsansatz diskutiert, der auf einer Reduktion durch die Zusammenfassung von Objekten zu Objektgruppen basiert.

Für die Implementierung von CORBA-Applikationen stehen grundsätzlich nur Programmiersprachen zur Wahl, für die ein Language-Mapping existiert. Das sind zur Zeit Ada, C, C++, COBOL, Java und Smalltalk. Ein objektorientierter Ansatz liegt mehreren dieser Sprachen zugrunde. Die noch relativ junge Sprache Java wurde für die Implementierung gewählt, da sie durch ihre konzeptuelle Plattformunabhängigkeit und der im Vergleich zu C++ größeren Typensicherheit und transparenten Speicherverwaltung vorteilhaft ist. Zielstellung dieser Arbeit ist die prototypische Entwicklung eines CORBA-Dienstes, der es verschiedenen Clients ermöglicht, gemeinsam auf Dokumente zuzugreifen und sie zu modifizieren. Dabei wird im Besonderen auf neue Bestandteile der CORBA-Spezifikation eingegangen, die die Interoperabilität, Skalierbarkeit und Performanz der Anwendung erhöhen. Die dafür notwendigen Grundlagen zu CORBA und XML werden in den folgenden beiden Kapiteln näher erläutert.

2. CORBA

2.1. Grundlagen

Inhomogenität ist ein Hauptmerkmal von Rechnernetzwerken. Je größer ein Netzwerk, desto komplexer ist es normalerweise und desto ausgeprägter ist auch die Verschiedenartigkeit seiner Komponenten. Große Netzwerke bestehen meist über einen langen Zeitraum und werden durch Hinzufügen von Subnetzen und Rechnern weiterentwickelt. Dadurch wächst der Aufwand für das Entwickeln von Programmsystemen für das Management dieser Netzwerke extrem an. Da die meisten Netzwerkprogramme unter Verwendung von Bibliotheken für die verschiedenen Abstraktionsebenen arbeiten, ergibt sich vor allem für verteilte Anwendungen ein hoher Grad an Heterogenität. Diese Problematik läßt sich nur durch das Erarbeiten einer weiteren Abstraktionsebene entschärfen, die die Komplexität der niederen Schichten verbirgt und durch ihre Homogenität dem Anwendungsentwickler eine solide Basis für die Lösung anwendungsorientierter Probleme schafft.

CORBA bietet eine solche systemunabhängige Schnittstelle für verteilte, objektorientierte Anwendungen. Die Architektur ist wie herkömmliche Client/Server-Architekturen aufgebaut, in denen Server-Anwendungen Dienste bereitstellen, die von Client-Anwendungen nachgefragt werden. CORBA generalisiert diese Bestandteile eines Client-/Server-Systems als *Objekte*, die durch den **O**bject **R**equest **B**roker (ORB) lokalisiert werden und das Ziel von Operationsaufrufen (*Requests*) durch Clients sein können. Das Auffinden von Objekten durch den ORB wird mit Hilfe von *Objektreferenzen* realisiert, die eine Art von Zeiger darstellen, um Objekte zu adressieren. Im Sinne der Bereitstellung oder Nachfrage von Diensten, können Objekte gleichzeitig als Server und als Clients auftreten. Repräsentiert ein Objekt das Ziel einer Client-Anforderung, spricht man von einer *Objektimplementierung*. Die

Objektimplementierung kann in einem völlig anderen Adreßraum als die Implementierung des aufrufenden Objekts existieren. Die konkrete programmiersprachliche Repräsentation einer oder mehrerer Objektimplementierungen heißt *Servant* und ist Bestandteil der Server-Anwendung. In der Programmiersprache Java stellt sich ein Servant als das Objekt einer Klasse dar. Die Instanzierung eines Objektes durch einen Servant beinhaltet die Assoziation dieses Servant mit dem Objekt für spätere Operationsaufrufe. Um Verwechslungen der Begriffe zu vermeiden, wird sie im Folgenden durch den Begriff *Inkarnation* beschrieben. Der Aufruf von Operationen auf Objekten sowie der Transport von Aufrufargumenten bzw. Rückgabewerten oder Exceptions geschieht mittels Senden von Nachrichten durch den ORB. Für die erfolgreiche Zustellung dieser Nachrichten, muß der aufrufende Client im Besitz der Objektreferenz dieses speziellen Objekts sein. Der Ablauf des Operationsaufrufs ist für den Client vollständig transparent. Im Quelltext des Programmes liest sich ein Request auf einem eventuell entfernten Objekt wie ein lokaler Funktionsaufruf.

Folgende Charakteristika sind, wie in [Vin99b] dargestellt Bestandteil des Operationsaufrufes auf einem CORBA-Objekt:

- Location-Transparenz

  Der Clientaufruf wird vollkommen unabhängig davon behandelt, ob sich das Objekt auf dem gleichen Rechner oder irgendwo im Netzwerk befindet. Objekte können außerdem ihren Aufenthaltsort verändern, ohne das der Client irgendeine Veränderung registrieren muß.

- Server-Transparenz

  Es ist nicht notwendig, daß der Client irgendeine Information darüber hat, in welchem Server welches Objekt implementiert ist.

- Sprach-Transparenz

  Die Implementierungssprache des Objektes ist unerheblich für den Client.

- Implementierungsunabhängigkeit

  Die Semantik von CORBA-Objekten ist hunterprozentig objektorientiert. Auch wenn das Objekt mit Hilfe von nicht objektorientier-

ten Techniken implemetiert ist[1], ist die Sicht von der Clientseite aus vollkommen konsistent.

- Architekturunabhängigkeit

  Einerlei welche Systemarchitektur dem Server zugrundeliegt, (z.B. ob die Byteanordnung dem Little-Endian oder Big-Endian folgt), für den Client ist es das gleiche Objekt.

- Unabhängigkeit vom Betriebssystem

  Auf die gleiche Art und Weise ist es unnötig für den Client zu wissen, auf welchem Betriebssystem der Server für das nachgefragte Objekt läuft.

- Protokollunabhängigkeit

  Unabhängig von der Verwendung eines eigenen, auf transportorientierten Protokollen aufsetzenden Nachrichtenprotokolls, ist es für den ORB gleich, auf welchem Transportprotokoll er operiert.

- Transportunabhängigkeit

  Protokolle sowohl der Transport-, als auch der Verbindungsschicht sind nicht Gegenstand der ORB-Spezifikation. Der ORB kann auf verschiedenen Technologien der Transportschicht (z.B. Ethernet, ATM, ISDN) arbeiten.

Abbildung 2.1 zeigt die verschiedenen Zugriffsmöglichkeiten von Clients auf Objektimplementierungen über den Object Request Broker. In den folgenden Abschnitten werden die einzelnen Techniken kurz erläutert.

Interface Definition Language (IDL)

Die Funktionalität von CORBA-Objekten, die Clientanfragen entgegennehmen um Operationen auszuführen, wird anhand von Interface Definitionen beschrieben. Sie umfassen die ausführbaren Operationen und die Parameter, die mit den Request übermittelt werden. Zur Beschreibung der Objektschnittstellen wird in CORBA die OMG Interface Definition Language verwendet [OMG98c]. IDL ermöglicht die Definition von

[1] So existieren auch Language-Mappings für die nicht objektorientierten Sprachen Ada, C und COBOL.

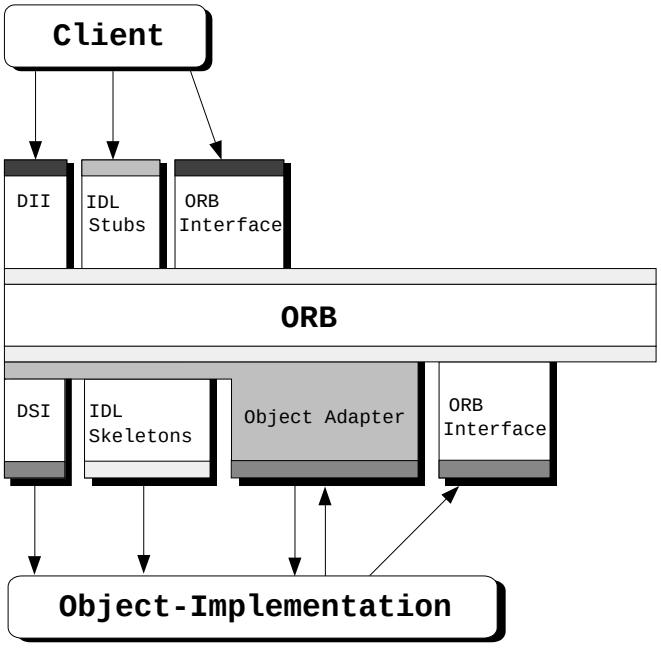


Abbildung 2.1.: Struktur eines Client-Request über den Object Request Broker

Interfaces unabhängig von Programmiersprachen. Erst die Anwendung eines speziellen *Language-Mappings*, der Vorschrift, wie IDL in die entsprechende Programmiersprache zu übersetzen ist, bildet die Grundlage für die Implementierung des Objekts. Dadurch bildet CORBA eine standardisierte Abstraktion, mit Hilfe derer Objekte verschiedener programmiersprachlicher Umgebungen miteinander kommunizieren können. Die Arbeit der Übersetzung von IDL in die jeweilige Sprache übernimmt ein IDL Compiler. Anhang B zeigt ein Beispiel der Anwendung der IDL-Spezifikation mit der Interace-Definition des in der Arbeit entwickelten CORBA-Dienstes.

Eine besondere Art von Interfaces stellen die Pseudo-Interfaces dar. Sie werden nicht als CORBA-Objekte, sondern normalerweise innerhalb der ORB-Distribution (z.B. in Form von Bibliotheks-Code) implementiert. Sie können nicht Argumente von Operationen auf gewöhnliche CORBA-Objekte sein und sind in dem zentralen Verzeichnisdienst für Schnittstelleninformationen, dem Interface Repository (siehe Abschnitt 2.3.2) nicht definiert. Mit Ausnahme von Pseudo-Interfaces erben alle Objektschnittstellen implizit von `CORBA::Object`.

Object Request Broker (ORB)

CORBA basiert auf dem Object Request Broker, einem Software-Bus, der die Kommunikation zwischen verteilten Objekten implementiert und vor dem Anwendungsprogramm verbirgt. Der ORB wird durch das Pseudo-Objekt `ORB` in IDL repräsentiert. Sowohl Objekte, die ihre Operationen remote zur Verfügung stellen, als auch Clients, die diese Operationen aufrufen, benötigen eine Referenz auf dieses Pseudo-Objekt. Diese Referenz liefert der Aufruf der Funktion `ORB_init(inout arg_listargv, in ORBid orb_identifier)`.

Der Vorgang des Operationsaufrufs eines Clients auf einem Objekt besteht für den ORB aus mehreren Arbeitsschritten :

- Lokalisieren des Zielobjektes
- Aktivieren der Server-Anwendung, falls der Server nicht aktiv ist
- Transfer der Aufrufargumente zum Objekt
- Aktivieren eines Servant für das Objekt, falls notwendig
- Warten auf die Antwort vom Objekt
- Rückgabe der `out`- oder `inout`-Parameter und des Rückgabewertes bei Erfolg
- Rückgabe einer Exception bei Auftreten eines Fehlers

Objektreferenzen

CORBA-Objekte werden durch Objektreferenzen identifiziert und lokalisiert. Die Objektreferenz besitzt dadurch eine Semantik, die der von Adreßzeigern in lokalen Speichern nahekommt. Der höhere Abstraktionsebene erlaubt es aber, Objekte auch in anderen Speichersystemen adressieren zu können. Anhand der Objektreferenz bestimmt der ORB, ob das referenzierte Objekt nur *remote* erreichbar ist. In dem seltenen Fall der Prozeßidentität von Objekt und Client kann sogar ein lokaler Funktionsaufruf realisiert werden. Clients sind nur dann in der Lage, Operationen auf Objekten auszuführen, wenn sie deren Referenzen besitzen. Es bestehen verschiedene Möglichkeiten für einen Client, die Objektreferenz zu erhalten:

- in Form einer Zeichenkette aus einer Datei, einer Webseite oder ähnlichen Quellen
- mit Hilfe eines Namensservice, der Namen in Referenzen auflöst
- als Rückgabewert der Operation eines anderen Objektes

Das Pseudo-Objekt `ORB` stellt zwei grundlegende Operationen zur Verfügung, die für die Konvertierung von Objektreferenzen in Zeichenketten und von Zeichenketten in Objektreferenzen verwendet werden. Das ORB-Interface enthält weiterhin die Operation `resolve_initial_references`, deren Funktionalität als *bootstrapping* bekannt ist. Sie liefert die Referenzen auf grundlegende CORBA-Dienste, wie den in Abschnitt 2.2 beschriebenen RootPOA oder den Namensdienst. Objektreferenzen weisen folgende Eigenschaften auf:

- sie sind eindeutig, d.h. sie verweisen auf genau eine Instanz eines Objektes
- mehrere Objektreferenzen können sich auf ein und dasselbe Objekt beziehen
- sie können Null sein
- sie können auf nicht existente Objekte verweisen
- sie repräsentieren sich dem Client als Zeichenkette, ihr wirklicher Inhalt bleibt also verborgen
- sie sind stark typisiert
- sie können persistent sein
- sie können interoperabel sein

Zur Gewährleistung der Interoperabilität[2], wird durch CORBA die **I**nteroperable **O**bject **R**eference (IOR) als standardisiertes Objektreferenzformat definiert. Die Spezifikation der IOR ist sehr flexibel, so daß auch zukünftige Netzwerkprotokolle zum Einsatz kommen können, ohne bereits existierende Client/Server-Systeme in ihrer Funktionalität zu

[2] Wie in Abschnitt 2.5 beschrieben, die Fähigkeit von ORBs verschiedener Hersteller, unabhängig von ihren System- oder Netzwerkumgebungen miteinander kommunizieren zu können.

beeinträchtigen. Wie in Abbildung 2.2 dargestellt, enthalten Objektreferenzen die Repository-ID, die notwendig ist, um die Beschreibung des Interfaces eines Objektes im Interface Repository (siehe Abschnitt 2.3.2) zuordnen zu können. Außerdem werden sie vom ORB für die Deduktion (das Eingrenzen eines Basistyps auf einen mehr abgeleiteten) von Objekttypen verwendet. Das zweite standardisierte Feld enthält Informationen über den Aufenthaltsort des Objektes, also über den Verbindungsendpunkt. Dazu gehören das verwendete Protokoll, die physikalische Adresse und die Portnummer (bei IIOP). Eine IOR kann Informationen für die Verbindung mit mehreren Protokollen parallel enthalten. So können alle Clients die gleiche IOR verwenden, auch wenn ihre Netzwerkumgebung nur eines der enthaltenen Protokolle unterstützt. Ein drittes Feld der Objektreferenz enthält den Objektschlüssel, der herstellerabhängige Informationen darüber enthält, auf welche Weise der Server das Objekt zu lokalisieren hat. Dieses Schlüsselfeld ist nur soweit standardisiert, daß es von jedem ORB übertragen wird. Es wird nur von dem ORB, für den der Inhalt bestimmt ist verwendet.

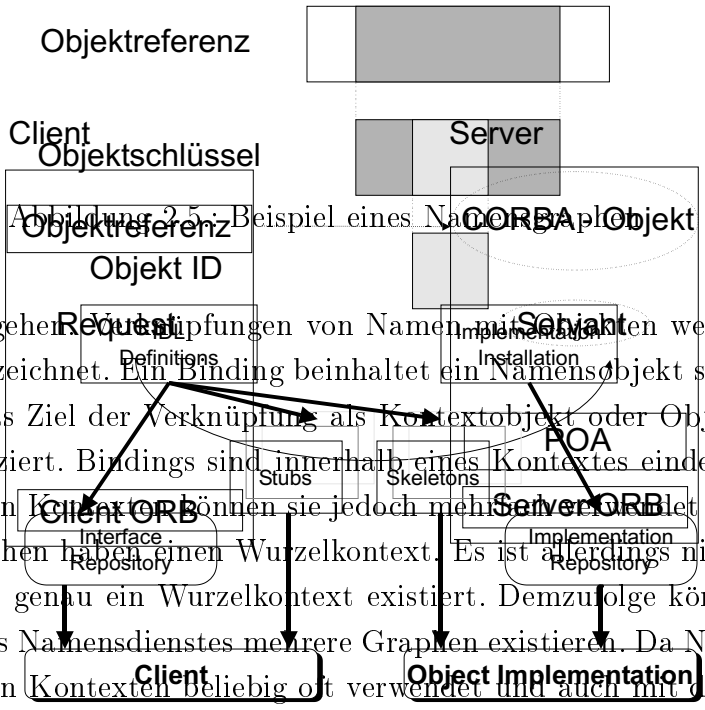
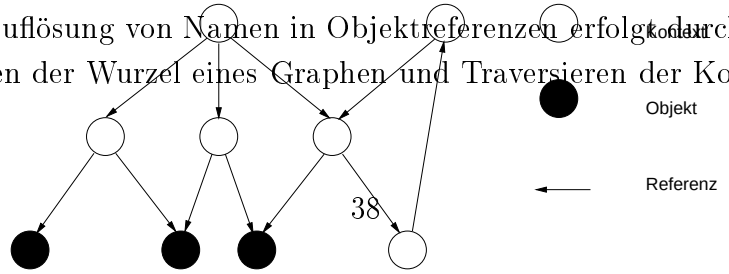


Abbildung 2.5: Beispiel eines Namensgraphen

Knoten ausgehenden Verknüpfungen von Namen mit Kontexten werden als *Bindings* bezeichnet. Ein Binding beinhaltet ein Namensobjekt sowie ein Flag, das das Ziel der Verknüpfung als Kontextobjekt oder Objektreferenz identifiziert. Bindings sind innerhalb eines Kontextes eindeutig. In verschiedenen Kontexten können sie jedoch mehrfach verwendet werden. Namensgraphen haben einen Wurzelkontext. Es ist allerdings nicht spezifiziert, daß genau ein Wurzelkontext existiert. Demzufolge können innerhalb eines Namensdienstes mehrere Graphen existieren. Da Namen in verschiedenen Kontexten beliebig oft verwendet und auch mit dem gleichen Objekt verbunden werden können, spricht man von der Möglichkeit des Namensgraphen, Schleifen zu enthalten.

Die Auflösung von Namen in Objektreferenzen erfolgt durch das Referenzieren der Wurzel eines Graphen und Traversieren der Kontextbin-



dings bis zum gewünschten Objektbinding. Die Angabe von voll qualifizierten Namen, was einer absoluten Pfadangabe in einem Dateisystem entsprechen würde, wird vom CosNaming nicht unterstützt. Aufgrund mehrerer möglicher Wurzeln eines Graphen werden Bindings immer relativ zu einem Kontext betrachtet.

Die Objektreferenz des Namensservice erhält das Anwendungsprogramm mit Hilfe der ORB-Operation `resolve_initial_references`. Beispiel für Java:

```
CORBA.ORB.orb = CORBA.ORB.init(args);
CORBA.Object obj =
  orb.resolve_initial_references("NameService");
CosNaming.NamingContext inc =
  CosNaming.NamingContextHelper.narrow(obj);
```

Der erste im Beispiel gezeigte Funktionaufruf beinhaltet die obligatorische Initialisierung des ORB, bei der Konfigurationsdetails und Properties eingestellt werden. Der nächste Funktionsaufruf liefert die Objektreferenz des initialen Namensdienstes, der durch den dritten Aufruf auf ein Namenskontext-Objekt eingegrenzt wird.

CosEvent Service

Innerhalb von verteilten Umgebungen müssen Objekte in der Lage sein, das Auftreten von Ereignissen wie z.B. das Aktualisieren eines Dokumentes, den Status zeitaufwendiger Berechnungen oder ähnliches anderen Objekten auf asynchrone Weise zu signalisieren. Die OMG Event Service Spezifikation geht in ihrem Modell von Ereignisproduzenten (Suppliers) und Ereigniskonsumenten (Consumers) aus. Als Bindeglied wird ein CORBA-Objekt, der Event Channel eingesetzt, über den Ereignisse unter Verwendung verschiedener Übertragungsmodalitäten versandt werden können. Es stehen vier verschiedene Typen zur Verfügung:

(1) **PushSupplier/PushConsumer Modell oder kanonisches Push-Modell**
Ereignisse werden vom PushSupplier erzeugt und durch Aufruf der `push`-Operation bei allen registrierten PushConsumern angezeigt. Dieses Modell ist das am häufigsten verwendete Modell.

(2) **PullSupplier/PullConsumer Modell oder kanonisches Pull-Modell**
Ereignisse werden von PullConsumern vom EventChannel abgefragt, welcher wiederum durch Aufruf der `pull`-Operation beim PullSupplier die Ereignisse anfordert.

(3) **PushSupplier/PullConsumer Modell oder hybrides Push/Pull-Modell**
Ereignisse werden vom PushSupplier erzeugt und PullConsumern angefordert, wobei der EventChannel die Rolle einer Warteschlange übernimmt.

(4) **PullSupplier/PushConsumer Modell oder hybrides Pull/Push-Modell**
Ereignisse werden vom EventChannel bei den PullSuppliern angefordert und an PushConsumer weiter verteilt, wobei der EventChannel einen Agenten darstellt, der alle Ereignisse selbsttätig übermittelt.

Zur transparenten Weiterleitung an eine Vielzahl von Objekten existieren Proxy-Interfaces, die verschiedenen Kombinationen von Push/Pull Kombinationen ermöglichen und die für die jeweiligen Produzenten bzw. Konsumenten von Ereignissen die direkte Verbindung zu ihren Pendants darstellen. Ein Objekt, das Ereignisse erzeugen und auf den Kanal legen möchte, alloziert zuerst ein `ProxyConsumerAdmin`-Interface und stellt über die Implementierung der `push`- bzw. `pull`-Operation die Verbindung zum Konsumenten her, der diese Operationen seinerseits über das von ihm allozierte `ProxySupplierAdmin`-Interface aufruft. Für die Signalisierung der im Beispiel angegebenen Aktualisierung eines Dokumentes, fällt die Wahl auf eine PushSupplier/PushConsumer Variante. Ein an dem Aktualisierungsereignis interessiertes Client-Objekt, würde also die `push`-Operation implementieren und erhält so vom Dokument-Objekt (als Objektimplementierung) das entsprechende Ereignis signalisiert.

Concurrency Control Service

Wenn ein Objekt von mehreren konkurrierenden Zugriffen belegt ist, wird eine Zugriffskontrolle bzw. ein Scheduling der Zugriffe notwendig. Der

Concurrency Control Service bietet die Möglichkeit, Objekte für den Zugriff zu sperren und auf diese Weise die Integrität von Änderungen des Status von Objekten zu gewährleisten. Es stehen die drei Zugriffsmodi

gewährter Modus	**angefragter Modus**				
Intention Read (IR)					*
Read (R)				*	*
Upgrade (U)			*	*	*
Intention Write (IW)		*	*		*
Write (W)	*	*	*	*	*

Tabelle 2.4.: Sperrungs-Modi des Concurrency Control Service

`Read`, `Write` und `Upgrade` zur Verfügung. Für Objekte variabler Granularität sind zusätzlich die Modi *Intention Read* und *Intention Write* hinzugefügt worden. Der gegenseitige Ausschluß der verschiedenen Zugriffsarten ist in Tabelle 2.4 dargestellt.
Für einen lesenden Zugriff auf ein Objekt innerhalb eines Objektgraphen, muß zuerst die Absicht (intention), das Objekt zu sperren durch eine intention read Anfrage an das übergeordnete Objekt im Graphen und dann an das Objekt selbst gestellt werden. Lesende Zugriffe dieser Art lassen keine anderen Schreibzugriffe, dafür aber Lesezugriffe zu. Schreibende Zugriffe mit dem Modus intention write lassen werder andere Lese- noch Schreibzugriffe zu. Durch diese Technik können Clients simultan lesenden Zugriff ausführen.

Property Service

Dieser Service ermöglicht die dynamische Generierung, Haltung und Löschung von Attributinformationen. Attribute werden in Form von Properties, ihre Definitionen in PropertyDefs gehalten. Ein Property ist ein Tupel `<property_name, property_value>`. Sie können in den Modi `normal`, `Readonly`, `Fixed_normal`, `Fixed_readonly` und `Undefined` verwendet werden. Der Property Service definiert im Weiteren ein Iterator-Interface, das für das schrittweise Durchsuchen großer Mengen von Name-

Wert Paaren eingesetzt wird

Obwohl die Attribute des Property Service ähnlich den Attributen sind, die durch IDL definiert werden, bestehen wichtige Unterschiede. Da bei dem Abfragen oder Modifizieren von IDL-Attributen keine Möglichkeit der Indikation von Fehlern durch das Auslösen von Ausnahmen (Exceptions) besteht, ist ihnen generell die Verwendung von Operationen auf Objekten vorzuziehen. Der Property Service kombiniert die Sicherheit von Interface-Operationen mit der Einfachheit der Semantik von Attributen.

Security Service

Das Modell der CORBA Security betrachtet Einheiten, die im System agieren als *Principals*, die über Sicherheitsattribute verfügen. Principals sind entweder menschliche Nutzer oder Systemeinheiten, die durch das System registriert worden sind [OMG98f]. Dem CORBA Security Service liegt eine umfangreiche Spezifikation von objektbezogenen Sicherheitsaspekten zugrunde, wie

- Identifikation und Authentifikation
  - Überprüfung der Identität von Principals
- Autorisierung und Zugriffskontrolle
  - Überprüfung der Identität von Principals und ihrer Zugriffsrechte beim Zugriff bzw. bei der Modifikation von Objekten
- Überprüfung der Verantwortlichkeit
  - Aktionen von Prinzipals müssen zuordenbar sein
  - Sicherung der Datenherkunft und des Datenerhalts
- sichere Objektkommunikation
  - Abgleich der Zugriffsrechte von Clients und Objekten beim Operationsaufruf
  - Schutz der Kommunikationsverbindung vor Abhören oder Modifikation
- Datenüberprüfung

- Administration
  - Definition und Management der Sicherheitsaspekte von Objektgruppen

Aufgrund seiner Modularität ist CORBA Security an verschiedenste Anforderungen anzupassen. Die Spezifikation erlaubt das Wechseln zwischen verschiedenen Security Diensten, die vom ORB verwendet werden. Unter dem übergeordneten Secure Inter-ORB Protocol (SECIOP) können die Protokolle Simple Public-Key GSS-API Mechanism (SPKM), Kerberos GSS und CSI-ECMA zum Einsatz kommen. Weiterhin sind das Distributed Computing Environment Common Inter-ORB Protocol (DCE-CIOP) oder Secure Sockets Layer (SSL) möglich. Die Security Service Spezifikation ist in numerierte Schichten aufgeteilt.

(1)
- Unterstützung von unsicheren Anwendungen
- ORB-gesteuerte Authentifikation, sichere Objektkommunikation, Autorisierung
- einfache Delegation

(2)
- Unterstützung von sicheren Anwendungen
- Möglichkeiten der Auswahl von Sicherheitsstufen, Veränderung von Delegationseinstellungen, Verwendung von Revisionsdiensten
- Unterstützung von Administrationsschnittstellen unter Verwendung von Sicherheitsdomänen

Die OMG hat eine Sicherheitsspezifikation entwickelt, die den Anforderungen einer Vielzahl kommerzieller Märkte gerecht wird. Kein anderer Sicherheitsstandard stellt eine solche komplette Interoperabilität zur Verfügung.

2.7. CORBA im Vergleich

Die Philosophie von CORBA ist keine Neuheit. Der Aufruf von auf entfernten Rechnern laufenden Programmen ist eine grundlegende Notwendigkeit funktionierender Netzwerke, und so ist eine Vielzahl konkurrierender Technologien verfügbar. Durch CORBA werden existierende Konzepte integriert und um die Eigenschaft der Interoperabilität erweitert.

Maßgebliche Punkte, die CORBA von anderen Architekturen hervorheben sind:

(1) konsequente Objektorientierung

(2) Offenheit und Systemunabhängigkeit des Standards

(3) Transparenz

(4) kurze Entwicklungszeiten (Wiederverwendbarkeit durch Objektorientierung)

(5) die ständige Weiterentwicklung durch die Mitglieder der OMG

Um die Wahl von CORBA als Ansatz für die Objektverteilung zu begründen, werden im Folgenden kurz drei wichtige Technologien, die auf dem Gebiet des entfernten Funktionsaufruf mit CORBA konkurrieren aufgeführt und ihre Nachteile genannt.

Distributed Component Object Model (DCOM)

Microsofts DCOM ist eine Technologie für die Kommunikation verteilter Komponenten. Es verfügt über eine Sprache zur Interface-Definition unabhängig von der Programmiersprache, die für die Implementierung verwendet wird und es bietet Möglichkeiten der Gestaltung von Server- und Clientanwendungen auf einer CORBA-gleichen Abstraktionsebene. Weiterhin existieren integrierte Standards für

- Authentisierung
- Autorisierung
- sichere Operationsaufrufe nach DCE (Distributed Computing Environment) RPC

Für die Kommunikation mit CORBA-Objekten über IIOP sind sogenannte Bridges spezifiziert worden. DCOM unterstützt keine Definition von User-Exceptions, seine Verfügbarkeit hat durch vorrangig kommerzielle Produkte nicht den Stand von CORBA mit seiner Vielzahl von freien Implementierungen.

Remote Method Invocation (RMI)

RMI ist seit der Java-Version 1.1 Bestandteil von Java. Wenn in einer Architektur die Beschränkung auf *eine* Implementierungssprache kein Problem darstellt, d.h. eine reine Java-Umgebung besteht, dann stellt das RMI die geeignete Wahl dar. Eigenschaften, die CORBA noch nicht ausreichend spezifiziert, sind integraler Bestandteil von RMI:

- Es unterstützt die Übertragung komplexer Datenstrukturen in Form von Java-Objekten.
- Es unterstützt die Freigabe des von entfernten Objekten belegten Speicherplatzes mit Freigabe der letzten Referenz auf diese Objekte (*Remote Garbage Collection*).

Durch die OMG wurde eine Spezifikation von RMI über IIOP vorgelegt, mit Hilfe derer Java-RMI-Objekte mit CORBA-Objekten anderer Implementierungssprachen kommunizieren können. Für die Java 2 Plattform ist eine Implementierung unter [Mic99] frei verfügbar. Damit ist die Einschränkung von RMI auf Java-Anwendungen verringert worden. Die Übertragung von Java-Objekten zu CORBA-Anwendungen anderer Implementierungssprachen ist aber nach wie vor wenig sinnvoll, da diese meist nur Strukturen (C) oder eigene Klassen (C++/Smalltalk) kennen.

Remote Procedure Call (RPC)

Das Remote Procedure Call Message Protocol (RPC) [Mic80] wurde von Sun Microsystems als Protokoll für entfernte Funktionsaufrufe entwickelt. Das Designprinzip von RPC ist nicht objektorientiert. Seine Abstraktionsgrad von Systemdetails ist geringer als der von CORBA. Obwohl die Verfügbarkeit von RPC durch viele verschiedene Implementierungen (es existiert auch eine Implementierung für Java [Net99]) groß ist, ist eine solche Einfachheit, wie sie durch die Schnittstellenbeschreibung durch OMG-IDL möglich ist mit RPC nicht zu erreichen.

2.8. Zusammenfassung

In diesem Kapitel wurde CORBA mit dem Schwerpunkt auf den letzten Erweiterungen des Standards und den Services vorgestellt. Die POA-Spezifikation und das Object-by-Value-Konzept werden eingesetzt, um

deren Auswirkungen auf die Leistungsfähigkeit und die Skalierbarkeit der CORBA-Anwendung, die Gegenstand dieser Arbeit ist zu untersuchen. Die strikte Objektorientierung von CORBA soll verwendet werden, um XML-Dokumente effektiv verteilt zu Verfügung zu stellen und nutzbar zu machen.

3. XML

Da sich XML[W3C98b] aufgrund seiner vielfältigen Möglichkeiten als Dokumentensprache im World Wide Web immer mehr durchsetzt, ist es als Format für die Speicherung von Dokumenten gewählt worden. Eine Konvertierung von anderen Dokumentformaten, wie zum Beispiel dem Rich Text Format (RTF) oder anderen herstellerspezifischen Formaten nach XML ist als Dienst einer Mittelschicht, die zwischen anfragenden Clients und dem XML-Daten exportierenden Dokumentendienst vermittelt, einsetzbar. Dies ist allerdings von der Existenz von Konvertierungsprogrammen für die jeweiligen Formate abhängig.

3.1. Grundlagen

Die Extensible Markup Language ist ein vergleichsweise einfacher Dialekt und eine Untermenge der **S**tandard **G**eneralized **M**arkup **L**anguage (SGML). Sie beschreibt eine Klasse von Datenobjekten - die XML Dokumente - und spezifiziert teilweise das Verhalten von Programmen, die XML-Dokumente verarbeiten. XML ist entwickelt worden, um den wachsenden Anforderungen von Web-Inhalten bezüglich Markup, herstellerunabhängigen Datenübertragung, systemunabhängigen Publizieren, kollaborativen Bearbeiten von Dokumenten und durch die Möglichkeit der maschinellen Verarbeitung auf Clientseite gerecht zu werden. XML ist voll internationalisiert. Es unterstützt europäische und asiatische Sprachen basierend auf dem Unicode Zeichensatz [W3C99].

XML-Dokumente bestehen aus Einheiten (Entities), die durch einem Parser verarbeitbare Daten enthalten. Ein Parser erwartet Zeichen-Daten und Markup-Daten, welche die Beschreibung der logischen Dokumentenstruktur und Speicherorganisation vermitteln. Der Zugriff von Anwendungen auf die vom Parser bereitgestellten Informationen über XML-Dokumente kann mit Hilfe verschiedener standardisierter Application

Programming Interfaces (APIs) erfolgen. Zwei grundlegende APIs werden durch das Document Object Model (DOM) und durch das Simple API for XML (SAX) spezifiziert. Während das DOM-API die Parserinformationen in einer Baumstruktur aufbereitet, stellt das SAX einen ereignisorientierten Zugriff mit Hilfe von Callback-Methoden bereit.

3.2. Document Object Model (DOM)

Das Document Object Model (DOM) ermöglicht es Anwendungen, anhand von Interface-Beschreibungen den Inhalt und die Struktur von Dokumenten zu lesen und zu modifizieren. Es repräsentiert das der Sprache XML zugrundeliegende objektorientierte Datenmodell [W3C98a]. Das DOM ist in OMG IDL beschrieben und bietet dadurch eine system- und sprachunabhängige Spezifikation. Durch dieses Modell ist eine äußerst feinkörnige Abbildung von XML-Daten auf informationstragende Objekte, zum Beispiel CORBA-Objekte möglich.

DOM Interfaces

Die Komponenten von Dokumenten werden durch eine Anzahl von Interfaces beschrieben. Durch die Verkettung der Komponenten ergibt sich ein Dokumentgraph, der von Parsern traversiert werden kann. Fundamentale Interfaces sind

- **Node**
  ist das grundlegende Interface des DOM. Node stellt einen Knoten im Dokumentengraph dar, der Nachfolger besitzen kann, aber nicht muß.

- **Document**
  stellt den Wurzel-Kontext des Dokumentgraphen dar, über den auf die weiteren Bestandteile des Dokuments zugegriffen werden kann. Außerhalb dieses Kontextes existieren keine Knoten, deshalb sind alle Factory-Funktionen für die Erstellung der verschiedenen Knoten in diesem Interface deklariert.

- **EntityReference**
  beinhaltet die Referenz auf eine Entity. Während des Parsevor-

ganges wird die Referenz durch den Zeichencode ihres Äquivalents ersetzt.

- **Element**
  ist der Typ, der den logischen Aufbau eines XML-Dokuments maßgeblich bestimmt.

- **Attr**
  repräsentiert ein Attribut eines Element-Objektes, in der Form wie es in der Document Type Definition (DTD) festgelegt ist. Attribute können in XML Referenzen enthalten, die auf komplexe externe Strukturen verweisen können.

- **ProcessingInstruction**
  kann Anweisungen für Anwendungsprogramme enthalten.

- **Text**
  enthält textuelle Daten und Markup.

Erweiterte Interfaces sind

- **CDATASection**
  enthält textuelle Daten mit Zeichencodes, die fälschlicherweise als Markup interpretiert werden könnten. Der maßgebliche Zweck dieses Typs ist das Einfügen von XML-Fragmenten.

- **DocumentType**
  repräsentiert den Dokumententyp eines Dokuments.

- **Notation**
  stellt eine Notation innerhalb einer DTD dar. Dieses Objekt enthält entweder das Format einer ungeparsten Entity, d.h. die ExternalEntity-Deklaration enthält den Eintrag NDataDecl='NDATA' oder es enthält die Deklaration von Zielen für ProcessingInstructions.

- **Entity**
  repräsentiert eine entweder vorgeparste oder ungeparst vorliegende physikalische Einheit eines XML-Dokuments.

Die Erstellung von Knoten des Dokumentgraphs wird per Spezifikation nicht anhand von Konstruktoren (wie in objektorientierten Programmiersprachen üblich) erreicht, sondern implementierungsabhängig durch Funktionen des Document-Interfaces. Eine derartige Vorgehensweise entspricht dem *Factory*-Prinzip bzw. dem *Factory*-Designmuster [Gam95]. Eine Factory vereint Funktionen für die Erstellung gleichwertiger oder untergeordneter Objekte in sich.

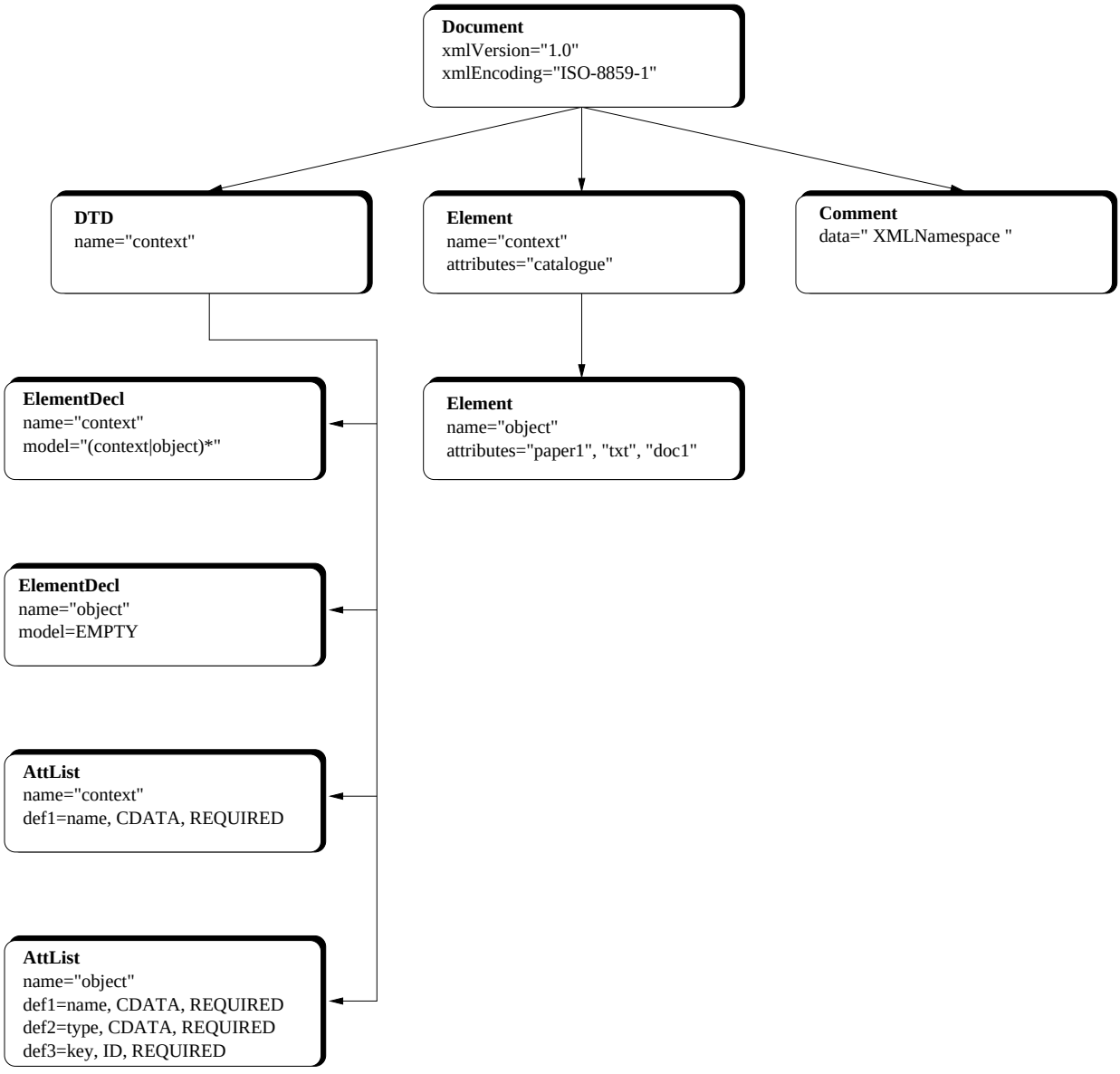


Abbildung 3.1.: Beispiel eines DOM-Tree

Das folgende gültige XML-Dokument ist in der Form seiner DOM-Datenstruktur in Abbildung 3.1 visualisiert. Der hierarchische Aufbau des Dokuments ist durch die Abstufung vom Wurzelknoten an der obersten Position zu dem ihm untergeordneten DTD-, Comment- und Elementknoten ersichtlich.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<!-- XMLNamespace -->
<!DOCTYPE context [
<!ELEMENT context (context|object)*>
<!ELEMENT object EMPTY>
<!ATTLIST context name CDATA #REQUIRED>
<!ATTLIST object
     name CDATA #REQUIRED
     type CDATA #REQUIRED
     key ID #REQUIRED>]>
<context name="catalogue">
<object name="paper1" type="txt" key="doc1"/>
</context>
```

External Entities

XML-Dokumente sind modular aufgebaut. Ihre physikalischen Einheiten (Entities) können innerhalb der Dokumentstruktur vorliegen. In diesem Fall werden sie beim Parse-Vorgang in jedem Fall erfaßt und bei Anwendung eines validierenden Parsers auf Gültigkeit untersucht. Sie können aber auch als externe Referenzen auftreten, wobei die Möglichkeit besteht, diese Einheiten als ungeparst in die DOM-Struktur zu übernehmen. Ungeparste Entities müssen in ihrer Deklaration den Eintrag NDataDecl='NDATA' enthalten. Dieses Schlüsselwort weist außerdem auf einen Inhalt der Entity hin, der im binären Format vorliegt und nicht XML-kodiert ist. Wenn der Parser während des Iterationsvorganges externe Entities überprüfen soll, dann ist die Angabe eines Identifiers *External-ID* erforderlich. Die External-ID enthält entweder eine *System-ID*, die einen Universal Ressource Identifier (URI) darstellt oder eine einen alternativen URI enthaltende *Public-ID*. Dieser Public-Identifier verweist zum

Beispiel in einen Katalog von Document Type Definitions (DTD), die die für die Validierung von XML-Dokumenten notwendigen Informationen enthalten. Dadurch kann die modulare Struktur von XML-Dokumente zu einer verteilten Struktur erweitert werden. Wie in Abschnitt 5.4 auf Seite 74 beschrieben wird, besteht mit Hilfe der Public-ID die Möglichkeit, während des Parse-Vorganges externe Entities zu verwenden, die mit verteilten Objekten verknüpft sind.

3.3. Simple API for XML (SAX)

Das SAX-API stellt zwar keinen formalen Standard dar, ist aber eine gewissermaßen nicht formale Spezifikation von David Megginson [Meg98]. Es ist eine ereignisbasierte Programmierschnittstelle für den Zugriff auf XML-Dokumente. Die Definition von Ereignisbehandlungsroutinen für spezielle Knotentypen der Baumstruktur eines XML-Dokumentes ermöglicht eine an die jeweiligen Anforderungen angepaßte Arbeitsweise der SAX-Anwendung. Ein Parser ruft nach dem Callback-Prinzip die entsprechenden Methoden während des Iterationsvorganges beim Auftreten der zugehörigen Elemente auf. Die Anwendung wird dadurch in die Lage versetzt, spezielle, Dokumente repräsentierende Datenstrukturen zu erzeugen oder andere Dokumentinformationen zu sammeln. Das SAX-API enthält Schnittstellenbeschreibungen für

- die Iteration über die Attributliste eines Elements
- typbedingte Ereignisse, wie die Deklaration von Entities
- das Auflösen von Entities entsprechend ihrer System-ID bzw. Public-ID
- das Liefern von Positionsinformationen für das Auftreten von Ereignissen in XML-Dokumenten
- die Fehlerindikation

Das SAX-API ist weniger für die Modifikation von Dokumentstrukturen als für die Änderung von Inhalten wie Text von Textelementen oder Attributen geeignet. Es lassen sich zum Beispiel Anfragen formulieren, wie „vermindere den Inhalt aller Elemente des Namens `preis` und einem

Wert des Attributes `kaufdatum` kleiner als `01.01.1999` um 100“. Neben der Einfachheit, die eine hohe Verarbeitungsgeschwindigkeit ermöglicht, ist ein Vorteil des SAX, daß Anwendungen nicht gezwungen sind, Speicherplatz für die Datenstruktur des gesamten XML-Dokumentes über den vollen Verarbeitungszeitraum bereitzustellen.

3.4. Zusammenfassung

XML wurde in diesem Kapitel vorgestellt, um einen Einblick in die Struktur und Mächtigkeit der Sprache zu schaffen. Die CORBA-Anwendung, die Gegenstand dieser Arbeit ist, verwendet XML als Beschreibungssprache für Dokumente, die durch CORBA-Elemente verteilt werden und so eine kollaborative Modifikation ermöglichen. Das DOM-API beinhaltet alle notwendigen Operationen, um Dokumentknoten zu manipulieren. Es wird durch eine Implementierung abgeschätzt, inwieweit die vollstandige Kapselung der DOM-Interfaces durch entsprechende CORBA-Objekte sinnvoll und praktikabel ist.

4. Entwurf des Dienstes

4.1. Überblick

Die Verteilung von Dokumenten über Netzwerke mit der Möglichkeit, Modifikationen vorzunehmen stellt einige Anforderungen an die Architektur. Dazu gehören

- die Wahl eines geeigneten Datenmodells
- die Wahl einer geeigneten Objekttechnologie
- die persistente Speicherung des Status von Dokumenten
- die transiente Haltung von Informationen, wie die Anzahl der gegenwärtig in Bearbeitung befindlichen Dokumente
- die Möglichkeit, Nutzer über erfolgte Änderungen an Dokumenten zu informieren
- sicherheitsrelevante Eigenschaften, wie Nutzerregistrierung und -Authentifikation oder Lese- und Schreibrechte

4.2. Das Datenmodell

Die Wahl des Datenmodells bzw. dessen Granularität bestimmt zu einem großen Teil die Leistungsfähigkeit der Anwendungen. Je kleiner die kleinste durch ein CORBA-Objekt repräsentierte Einheit eines Dokumentes ist, desto größer ist die Anzahl der bei der rekursiven Iteration dieses Dokumentes notwendigen Operationsaufrufe. Durch eine Abbildung des Document Object Models auf CORBA-Objekte ist die Fähigkeit von XML-Dokumenten erreichbar, „sich selbst“ remote abrufbar zur Verfügung zu stellen. Dafür sind zwei maßgebliche Ansätze überdacht worden, die im Folgenden dargestellt werden.

4.2.1. Die vollständige Abbildung des DOM

Die erste Variante bildet das Dokument vollständig durch das DOM ab, wobei jedes DOM-Objekt gleichzeitig ein CORBA-Objekt ist. Jeder Bestandteil eines XML-Dokumentes, auf den ein Zugriff mit Hilfe des Interfaces `Node` aus dem Document Object Model gestattet ist, wird durch eine Schnittstelle mit Hilfe des ORB zur Verfügung gestellt. Daraus ergibt sich ein äußerst feinkörniges Modell. Die IDL-Spezifikation in Anhang A enthält die Beschreibung eines CORBA-Objektes, das das grundlegende DOM-Interface `Node` kapselt.

Geht man davon aus, daß ein CORBA-Objekt, welches ein Dokument repräsentiert für seine Aktivierung (welche auch die Aktivierung aller seiner Kindobjekte beinhaltet) rekursiv iteriert werden muß, dann wird ersichtlich, daß die konsequente Objektorientierung einer erheblichen Belastung der Infrastruktur gegenübersteht. Davon sind die Server, die Kommunikationmedien und auch die Clients betroffen. Schon die Übermittlung eines kleinen Teils eines Dokumentes (z.B. ein einzelnes „tag“ in einem HTML- bzw. XML-Dokument) erfordert die Aktivierung aller übergeordneten Objekte. Eine Ausnahme bildet die Situation, daß der anfragende Client schon im Besitz der persistenten Referenz dieses Dokumententeiles bzw. des umgebenden Objektes ist. In diesem Fall würde die Aktivierung dieses einzelnen Objektes ausreichend sein.

Die Repräsentation jedes Teilstückes eines Dokumentes durch ein CORBA-Objekt ist vor allem dann sinnvoll, wenn eine Bearbeitung dieses Teilstückes unabhängig von anderen Teilen des Dokumentes erfolgen kann. Es wird aber ersichtlich, daß eine Modifikation eines XML-Unterbaumes ohne Sperrung der übergeordneten Elternknoten nicht erfolgen kann. Ohne diese Sperrung wäre eine Änderung verketteter Dokumententeile durch verschiedene Nutzer möglich, die das Dokument in einen inkonsistenten Zustand versetzen würde. Einen Sperrungsmechanismus, der diesen Anforderungen genügt bieten die Sperrungsmodi Intention Read bzw. Intention Write (siehe Tabelle 2.4), die der Concurrency Control Service definiert. In Abbildung 4.1 ist dargestellt, wie zwei Clients konkurrierend auf verknüpfte Objekte zugreifen. Ein lesender oder schreibender Zugriff von Client B auf den Teilknoten des Dokumentes muß in jedem Fall einen schreibenden Zugriff von Client A auf den übergeordneten Teilknoten verhindern. Client A kann sonst den Status

des Knotens in einer Weise verändern, die dessen Unterbaum beeinflußt. Umgekehrt bedeutet ein schreibender Zugriff von Client A implizit die Sperrung aller Unterbäume dieses Knotens.

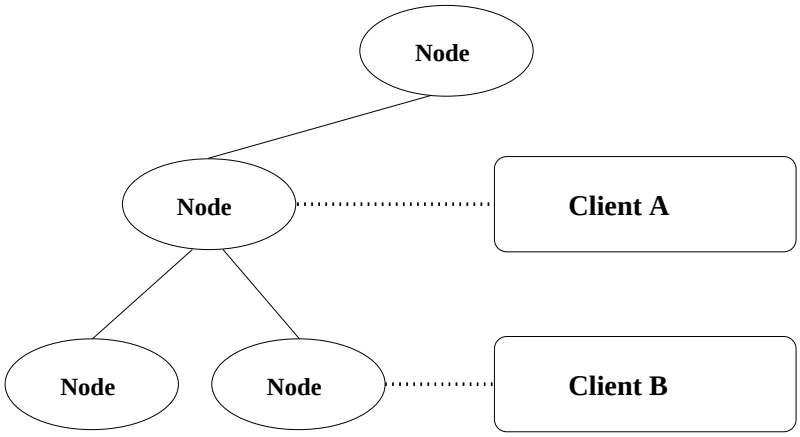


Abbildung 4.1.: gleichzeitiger Client-Zugriff auf Dokumententeile

Unabhängig davon, ob beim Einsatz des Portable Object Adapters für die Weiterleitung von Client-Anfragen ein Default-Servant verwendet wird, oder nicht, der Speicher sowie die Serverkapazität, die für die Inkarnation der Objekte durch Servants aufgewandt werden, könnte ein durchschnittliches Rechnersystem schnell an seine Grenzen bringen. Clientseitig kann die Anfrage nach dem Wurzelknoten eines großen Dokumentes zu langen Wartezeiten durch die serverseitige Objektaktivierung und Sperrung kommen.

4.2.2. Die teilweise Abbildung des DOM

In der zweiten Variante werden nur ausgewählte Bestandteile des DOM durch CORBA-Objekte dargestellt. Enstprechend dem DOM existiert nur *ein* übergeordnetes Factory-Objekt, das Document-Objekt, mit dessen Hilfe untergeordnete Bestandteile des Dokumentes erzeugt werden können. Wird diese Eigenschaft auf CORBA-Objekte übertragen, dann ergibt sich ein grobkörniges Datenmodell: Es wird nur das DOM-Interface `Document` durch ein korrespondierendes CORBA-Interface abgebildet. Die IDL-Spezifikation ist in Anhang B enthalten. Dadurch wird die Anzahl der remote-Operationsaufrufe bei der Iteration über ein Dokument minimiert, wobei die Menge der in Form von Parametern bei jedem Auf-

ruf zu übertragenden Daten entsprechend wächst. Diese Variante resultiert letztendlich in einer Operation, die komplette Dateien entweder in Form eines Byte-Arrays oder als Iterator über ein Byte-Array verschickt. Das Interface `DocNode` kann die Funktionalität eines Iterator-Objektes enthalten. Die Realisierung dieser Möglichkeit wird in Abschnitt 5.3.2 beschrieben. Dadurch wird das Senden von riesigen Datenmengen innerhalb eines Aufrufs vermieden, was einerseits anderen Clients die Möglichkeit zum Agieren gibt und andererseits einen Speicherüberlauf beim Marshaling verhindert.

Die Möglichkeit, ein einzelnes Dokumentteil separat zu bearbeiten läßt sich unter Verwendung dieses Datenmodells nur mit großem Aufwand realisieren, da das Modell keine Teilung von Dokumenten vorsieht. Angesichts der Restriktionen, die sich durch die Verwaltung konkurrierender Zugriffe und der hohen Rechnerlast durch rekursive Operationsaufrufe beim Liefern eines Dokumentunterbaumes aus dem erstgenannten Modell ergeben, ist diese zweite Variante leistungsfähiger hinsichtlich Geschwindigkeit und Speicheraufwand.
Zusammengefaßt stehen zwei Darstellungsformen zur Verfügung, die sich durch erhebliche Geschwindigkeitsunterschiede auszeichnen. Eine Aggregation der beiden Modelle ließe sich zur nutzbringenden Verbindung der positiven Eigenschaften in der Form einsetzen, daß auf einzelne Dokumentknoten separat zugegriffen werden kann, ganze Unterbäume aber immer als komplette Struktur gesendet werden und nicht durch rekursive Iteration vom Wurzelknoten an.

Ein gemeinsamer Zugriff auf Dokumente erfordert die Möglichkeit der Überprüfung von Zugriffsberechtigungen. Zusammen mit spezifischen XML-Strukturinformation, wie Elternknoten und Kindknotenliste muß das Interface, das eine übertragbare Einheit (normalerweise ein Dokumentknoten) repräsentiert zusätzliche Attribute erhalten, die Besitzerinformationen und Zugriffsrechte speichern. Die Struktur eines Dokumentknotens in der Darstellungsform als Value-Type ist wie folgt gestaltet.

```
valuetype NodeValue
{
    private NodeType nodeState;
    private DocNode parent;
    private DocNodeList nodeList;
```

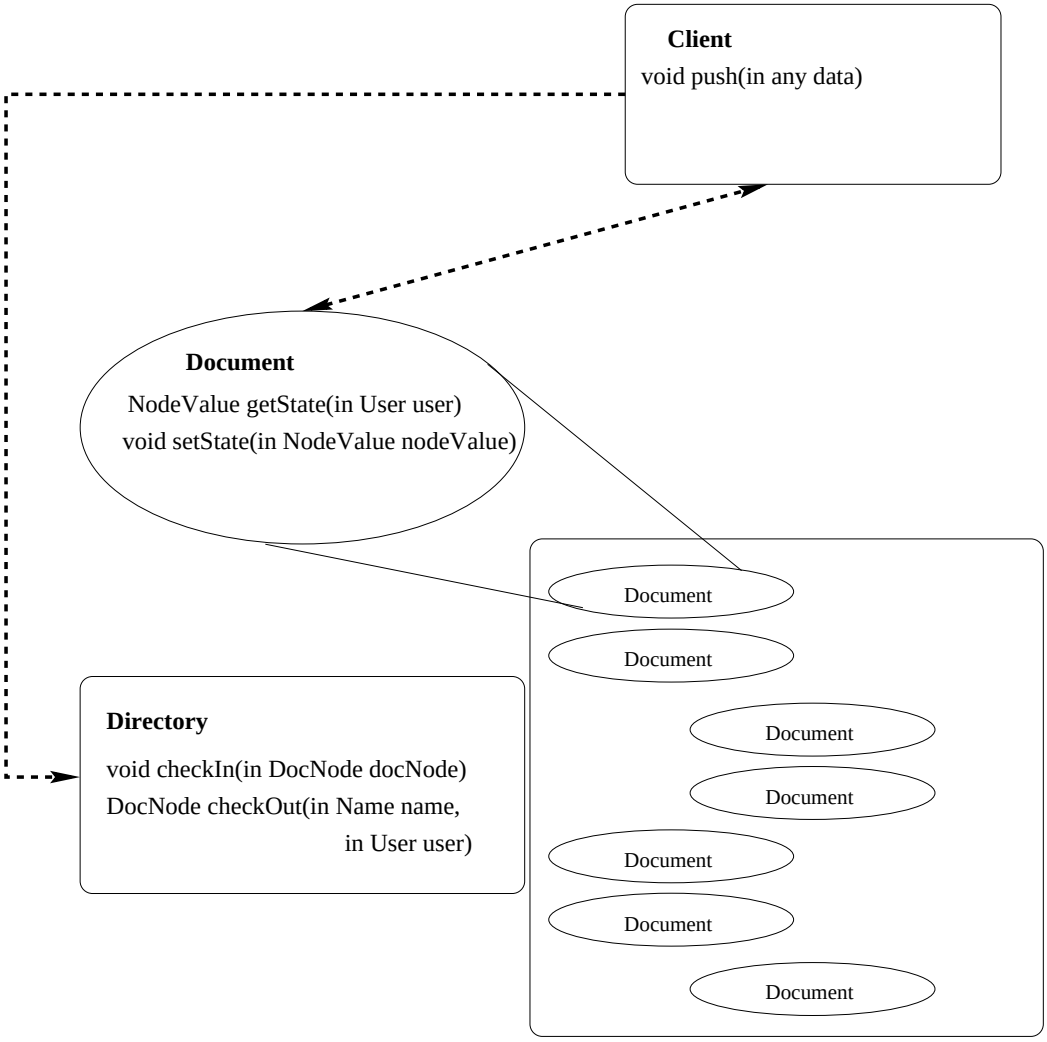


Abbildung 4.2.: Darstellung des Zugriffs auf ein Dokument

```
    private Owner owner;
    private Permissions permissions;

    factory initNode(in NodeType defaultNode);
};
```

Die Strukturdefinitionen für Namen oder Zugriffsrechte sind entsprechend den Anforderungen zu wählen. Im Normalfall entsprechen Namen gewöhnlichen IDL Zeichenketten (Typ `string`). Zugriffsrechte können durch den Aufzählungstyp (Typ `enum`) dargestellt werden.

In Abbildung 4.2 wird sichtbar, wie Clients über einen Verzeich-

nisdienst auf den Wurzelknoten von Dokumenten zugreifen können, über den dann die Möglichkeit besteht, das gesamte Dokument iterativ anzufordern. Die in der Darstellung vorhanden Elemente sind die in den Ablauf involvierten CORBA-Objekte mit den wichtigsten Operationen ihrer Schnittstellen. Mit Hilfe der Inhaltsdarstellung des Verzeichnisdienstes, die durch die Operation `DocNode list()` zurückgegeben wird, kann die Clientanwendung ein bestimmtes Dokument durch Aufruf der Operation `DocNode checkOut(in Name name, in User user)` anfordern. Da der Typ `Document` von `DocNode` abgeleitet ist, verfügt er ebenso über die Operation `NodeValue getState(in User user)`. Deren Aufruf liefert den Status des Dokumentes bzw. Dokumentknotens. Nach erfolgten Änderungen an den Daten des Dokumentes ruft die Clientanwendung auf Nutzeranforderung hin die Operation `void setState(in NodeValue nodeValue)` auf. Die Implementierung dieser Operation beinhaltet den Aufruf eines Basisdienstes für die Verbindung mit einer Datenbank, um die Dokumentdaten zu speichern. Nach erfolgreicher Speicherung wird unter Verwendung des CosEvent Services (siehe Abschnitt 2.6) ein Ereignis für die Notifikation der Clientanwendungen generiert. Dabei wird die Operation `void push(in any data)` aufgerufen, die dem Client entsprechende Reaktionen ermöglicht.

4.3. Die Speicherung von Dokumentinformationen

Eine Menge von Dokumenten kann durch eine Menge von Verweisen dargestellt werden. Der Ort der persistenten Speicherung muß daher nicht zwingend zentralisiert sein. Es sind zentrale oder verteilte Datenbanken, oder auch der Verbleib von Dokumenten auf den Rechnern der Autoren denkbar. Die Dokumentverweise selbst werden durch einen Verzeichnisdienst verwaltet, der aus der Sicht von CORBA eine dem CosNaming ähnliche Funktionalität bereitstellt. Dieser Dienst muß grundlegende Funktionen für das *Check-In* bzw. *Check-Out* von Objekten, sowie die Möglichkeit, seinen gesamten Inhalt iterativ zu durchsuchen bereitstellen. Das Ausführen einer Check-In Funktion beinhaltet den Eintrag des Dokumentnamens und -Typs in die Verzeichnishierarchie und das Zurückschreiben des Dokumentstatus auf das Medium für die persisten-

te Speicherung. Die Auflistung des Verzeichnisses selbst wird im XML-Format exportiert und muß von der aufrufenden Anwendung selbständig iteriert werden. Wie jedes andere Dokument auch, kann der Verzeichnisdienst seinen Inhalt auf das Medium für die persistente Speicherung mittels der Check-In Operation eintragen.

4.3.1. Die persistente Speicherung des Dokumentstatus

Der Status eines Dokumentknotens besteht aus den typischen im Dokument enthaltenen Daten, für XML-Dokumente die Textdaten, sowie Daten, die den Besitzer identifizieren. Informationen über Nutzer, die zum gegenwärtigen Zeitpunkt lesend oder schreibend auf das Dokument zugreifen, gehören zu transienten Daten, deren Speicherung in Abschnitt 4.3.2 beschrieben wird. Jedes CORBA-Objekt, das einen Dokumentknoten darstellt, beinhaltet die Operation `setState`, die der Eigenverantwortlichkeit des Objekts für die persistente Speicherung gerecht wird. Diese Operation und ihr Gegenstück `getState` stellen den leistungskritischen Kern des Entwurfs dar. Die Geschwindigkeit, mit der der Status eines gesamtes Dokumentes gespeichert bzw. zum Client gesendet werden kann wird von den Aktionen bestimmt, die von dieser Operation ausgeführt werden. Mit dem Basic Object Adapter muß jedes Objekt einer programmiersprachlichen Repräsentation, einem Servant entsprechen. Erst der POA ermöglicht uns die Registrierung eines *Default Servant* (siehe Seite 24), der für die Inkarnation jedes Objekts eines Basistyps verantwortlich ist. Durch die Umsetzung des Flywight (Fliegengewicht)-Designmusters [Gam95] bei der Konzeption des Default Servant ist es möglich, eine große Menge von Objekten durch eine Standardschnittstelle zu behandeln. Die Anwendung kann eine theoretisch unbegrenzte Anzahl an Objekten für Dokumentknoten erstellen und verwendet dafür eine konstante Anzahl an Servantinstanzen, die mit der Anzahl der verwendeten Dokumente korrespondiert. Die korrekte Referenzierung der Knoten erfolgt durch die Vergabe einer Objekt-ID durch die Anwendung. Die Skalierbarkeit dieser Lösung ist augenscheinlich durch eine Implementierung unter Verwendung des Basic Object Adapters nicht zu erreichen. Der Speicherbedarf einer Implementierung mit dem BOA wächst proportional zur Anzahl der erstellten Objekte. Abhängig von der Größe

des Servants, der ein Dokumentknoten-Objekt inkarniert, kann schon ein einziges Dokument den Speicherbedarf der Anwendung vervielfachen.

4.3.2. Die Speicherung transienter Daten

Dokumentbezogene Informationen, wie

- die Anzahl der in Bearbeitung befindlichen Kopien
- wie viele und welche Nutzer eine Kopie in Benutzung haben

sind dafür geeignet, in der Form von objektbezogenen Eigenschaften (Properties) gespeichert zu werden. Die Verwaltung dieser Informationen wird von einem Dienst ausgeführt, der den gesamten Bestand an Dokumenten überwacht. Ein unabhängiger Property-Service (siehe Abschnitt 2.6) bietet eine standardisierte Möglichkeit, diese Informationen zu halten und auf Anfrage zur Verfügung zu stellen.

Durch die Definition eines Referenzzählers innerhalb des Property-Sets, ist die Beschränkung der Anzahl der Nutzer, die gleichzeitig schreibend auf ein Dokument zugreifen können möglich. Aus Gründen der Konsistenzerhaltung sollte normalerweise nur ein Client in der Lage sein, ein Objekt zu modifizieren. Anderenfalls wäre auf der Seite des CORBA-Servers eine umfangreiche Versionskontrolle notwendig, bei der mehrere Kopien pro Dokument existieren können. Diese Vorgehensweise kann soweit erweitert werden, daß Dokumente nicht ersetzt, sondern generell unter einer höheren Versionsnummer zurückgeschrieben werden. Der Speicheraufwand auf dem Medium für die persistente Speicherung aber auch der Kontrollaufwand durch die Serveranwendung würde sich vervielfachen. Ein gleichzeitiges Bearbeiten eines Dokumentknotens durch mehrere Benutzer soll aus diesen Gründen nicht möglich sein. Für das gleichzeitige Lesen kann aber ein Referenzzähler genauso notwendig werden. Es kann dadurch die Möglichkeit geschaffen werden, die CORBA-Objektimplementierungen inaktiver Objekte aus dem Speicher zu entfernen und ihren POA zu zerstören. Dadurch kann der Ressourcenbedarf der Serveranwendung auf einem gleichmäßigen Niveau gehalten werden.

4.4. Die Notifikation von Clients

CosEvent, der Ereignis-Notifikations-Service von CORBA dient dem Propagieren von Ereignissen an interessierte Clients. Clients, die sich für die Notifikations registrieren, müssen nicht notwendigerweise CORBA-Objekte sein, die Implementierung des dem Event-Modell entsprechenden Interfaces ist Voraussetzung für die Fähigkeit, Nachrichten zu empfangen.

Die Übermittlung von Nachrichten wird durch den Aufruf einer Operation auf einem Proxy-Objekt (für eine Beschreibung des Proxy-Designmusters siehe [Gam95]) realisiert, das stellvertretend für alle an der Ereignisnotifikation interessierten Objekte verwendet wird. Auf diese Weise wird eine asynchrone Kommunikation zur Verfügung gestellt. Der Signalisierungsmechanismus von CosEvent erlaubt weder Rückmeldungen an den Ereignisproduzenten, noch ist das Senden von Ereignissen an genau adressierte Clients möglich. Wenn erweiterte *Qualtiy of Service*-Anforderungen z.B. die Sicherheit des Erhalts von Ereignisbenachrichtigungen, die Definition von Zeitfenstern für die Nachrichtenlieferung oder *time-to-live*-Attribute von Nachrichten notwendig sind, dann sollte auf die CORBA Messaging Spezifikation [OMG98b] bzw. die Implementierung eines Messaging Services zurückgegriffen werden.

Für diese Arbeit wird das in Abschnitt 2.6 auf Seite 39 beschriebene PushSupplier/PushConsumer Modell für die Benachrichtigung von Clients über erfolgte Änderungen an Dokument-Knoten eingesetzt. Die Auswahl einer der verfügbaren Strategien der Ereigniserzeugung bzw. -verwertung richtet sich nach dem Enstehungsort des zu propagierenden Ereignisses, sowie der Notwendigkeit von asynchroner Verarbeitung des Ereignisses. Änderungsnotifikationen entstehen zwar prinzipiell durch den Client, dessen aktuelle Kopie eines Dokumentes modifiziert worden ist. Die Modifikation wird aber erst sichtbar für andere Clients, wenn der Status des entsprechenden Dokumentknotens erfolgreich auf dem Speichermedium eingetragen worden ist. Demzufolge wird durch die Implementierung des Verzeichnisdienstes ein Ereignis erzeugt, das im Normalfall bei allen registrierten Clients zum Aufruf der Operation `void push(any data)` führt. Im `any`-Argument dieser Operation ist eine Referenz auf das modifizierte Objekt enthalten, die Clients die Unterscheidung zwischen dem modifizierten Ob-

jekt und der eigenen aktuellen Kopie ermöglicht. Die Clientanwendung muß eine private Klasse (`_EventPushConsumer`) enthalten, die von der Klasse `_PushConsumerImplBase` abgeleitet ist, da die Sprache Java ohne Mehrfachvererbung spezifiziert ist. Diese Basisklasse wird durch das Java Language-Mapping definiert. Sie implementiert das Interface `PushConsumer` aus der CosEvent-Spezifikation.

```
private class _EventPushConsumer
           extends _PushConsumerImplBase
{
  public void push(org.omg.CORBA.Any data)
    throws Disconnected
  {
    ...
  }
  public void disconnect_push_consumer()
  {
    ...
  }
} // class _EventPushConsumer
```

Der Verzeichnisdienst verwendet für die Generierung des Ereignisses eine Klasse mit entgegengesetzter Funktionalität. Diese Klasse ist vom Typ `PushSupplier`.

Es sind verschiedene Aktionen, die durch Clients bei Erhalt des Ereignissignals in der `push`-Operation auszuführen sind denkbar.

(1) automatisches Neuladen des Dokumentknotens

(2) Anzeigen der Notifikationssignals und eventuelles Neuladen des Dokumentknotens durch Nutzerinteraktion

Eine automatische Anfrage an die Serveranwendung zur Aktualisierung des Status ist aus dem folgenden Grund ungünstig. Bei vielen involvierten Clients kann es zu einem plötzlichen Anwachsen der Anfragenanzahl

an die Serveranwendung kommen, die zu Verzögerungen führt. Das Anzeigen der erfolgten Modifikation durch die Clientanwendung überläßt die weiteren Entscheidungen dem Benutzer und führt dadurch zu einer zufälligen Verzögerung der neuen Anfrage.

Die EventService-Implementierungen des C++ ORBs OmniORB2 und der Java ORBs JavaORB und Visibroker sind für die Bereitstellung eines EventChannels eingesetzt worden. Dem Vergleich des Leistungsverhaltens widmet sich der Abschnitt 6.

4.5. Zusammenfassung

Wie in Abschnitt 4.3 ausgeführt wurde, müssen Dokumente nicht zwingend zentral gehalten werden. Die Semantik von CORBA-Objekten erlaubt beliebige Verteilungen. Da im Verlauf der Arbeit mit den beiden in Abschnitt 4.2 dargestellten Abbildungsmöglichkeiten des DOM experimentiert worden ist, verwendet die Architektur grundlegend *eine* zentrale Datenbank. Die wurde notwendig, da Dokumente, die zusammenhängend übertragend werden sollen auch zusammenhängend gespeichert werden müssen. Die Sicht auf die gespeicherten Dokumente wird durch einen Verzeichnisdienst verwaltet, der auch die Funktionen für das Check-In und Check-Out von Dokumenten beinhaltet (siehe Anhang B).

5. Implementierung

5.1. Der Datenbankzugriff

Die Verwendung einer objektorientierten Datenbank bietet den Vorteil, Java-Objekte als solche zu speichern, ohne daß diese vorher durch die Anwendung serialisiert werden müssen. Dieser Vorteil kommt aber nur dann zum Tragen, wenn serverseitig vorgeparste XML-Strukturen für die Übertragung zu den Clients verwendet werden. Das Übertragen von XML-Daten als kodierte Zeichenkette ist aufgrund der geringeren Größe und der Interoperabiltät vorzuziehen. Deshalb ist es unerheblich, welche Art von Datenbank für die Speicherung zum Einsatz kommt. Eine geeignete Wahl stellt eine relationale Datenbank dar, auf die mit Hilfe eines JDBC-Treibers (JDBC - Java Database Connectivity) zugegriffen wird. JDBC ist ein dem ODBC ähnliches API, das das Verbindungsmanagement mit der Datenbank regelt bzw. Anfragen der Anwendung an die Datenbank vermittelt. JDBC-Treiber können in vier verschiedenen Kategorien eingeteilt werden [Mic98]:

(1) JDBC-ODBC Bridge-Treiber ermöglichen den Zugriff auf ODBC-gesteuerte Datenbanken durch Übersetzung der JDBC-API Aufrufe in solche für ODBC.

(2) Native API-Treiber mit teilweiser Java-Technologie Unterstützung übersetzen die Aufrufe der JDBC-API in die entsprechenden Funktionsaufrufe der Datenbank bzw. des Programms, das die Datenbank verwaltet. Das erfordert, ähnlich wie beim Bridge-Treiber die Installation eines entsprechenden Programms auf der Clientseite.

(3) Netzprotokoll-Treiber mit vollständiger Java-Technologie Unterstützung übersetzen JDBC-API Aufrufe in vom Datenbanksystem unabhängige Protokoll-Aufrufe, die von einem Server-

Programm in native Funktionsaufrufe der Datenbank gewandelt werden.

(4) Treiber mit vollständiger Java-Technologie Unterstützung, die JDBC-API Aufrufe in das datenbankspezifische Protokoll übersetzen und so das DBMS direkt kontaktieren. Dadurch sind Treiber dieser Kategorie proprietär.

Die Datenbank wird durch eine repräsentative Zeichenkette beschrieben und durch Übergabe dieser Zeichenkette an die Funktion `getConnection` des Treibers eine Verbindung initiiert. Ihr Aufbau ist ähnlich dem einer URL:

```
jdbc:freetds:sqlserver://foo.com/database
```

Anhand von Anfragen, die in der Structered Query Language (SQL) formuliert werden, kann der Status der Datenbank abgefragt bzw. manipuliert werden.

Dokument-Objekte können entweder in Form von ungeparsten XML-Dateien oder als serialisierte Java-Objekte in der Datenbank abgespeichert werden. Aus Gründen der Interoperabilität werden die Objekte als XML-Dateien gespeichert, in einer relationalen Datenbank kann demzufolge der Zell-Datentyp `Text` verwendet werden. Demgegenüber verlangt ein serialisiertes Java-Objekt ein binäres Datenformat und benötigt im Schnitt mehr als 10 mal so viel Speicherplatz wie das ungeparste XML-Format. Der Grund dafür ist die Technik der Objektserialisierung von Java. Dabei wird rekursiv in der Datenstruktur jedes Objekt absteigend der voll qualifizierte Name und der Status in den Serialisierungs-Stream geschrieben[1].

5.2. Der Einsatz des CORBA Naming Service

Der Namensservice von CORBA stellt eine Abbildung von Namen auf Objektreferenzen bereit und verfügt somit über eine Funktionalität, die dem Domain Name Service ähnlich ist. Da durch die Spezifikation keine Beschränkung der Anzahl von Objekten, die innerhalb eines Kontextes gebunden sind vorgesehen ist, ist die Abbildung von Dokumenten bzw.

[1] Das grundlegenste Java-Objekt benötigt in serialisierter Form 256 Bytes.

Komponenten von Dokumenten mittels CosNaming überlegenswert. Auf diese Weise können z.B. Entities aus XML-Dokumenten anhand ihrer Namen identifiziert, d.h. als CORBA-Objekte mittels Namensauflösung referenziert werden. Dieser Ansatz beinhaltet jedoch zwei Schwachpunkte:

(1) Die Referenzierung jeder Komponente eines Dokuments erfordert die zusätzliche Kommunikation mit dem Naming Service. Das bedeutet, daß sich selbst bei einer optimalen Implementierung durch die zusätzliche Kosten Einbußen in der Gesamtleistungsfähigkeit ergeben.

(2) Die Implementierungsstrategie des CosNaming ist stark herstellerabhängig, da sie in der Spezifikation nicht festgeschrieben wird. Einige Implementierungen unterliegen Einschränkungen in der Länge von Namen oder der Anzahl von möglichen Name-Bindings innerhalb eines Kontextes.

Aus diesen Gründen beschränkt sich der Einsatz von CosNaming ausschließlich auf die Abbildung der Namen von grundlegenden Diensten. Da das CosNaming den grundlegensten CORBA-Dienst darstellt, wird er von den meisten ORB-Herstellern implementiert. Im Verlauf dieser Arbeit sind die Naming Services des C++ ORB OmniORB2 von Olivetti (kompiliert unter Linux 2.0) und die Naming Services des freien JavaORB2 sowie des Visibroker for Java 3.4 getestet worden. Damit der Dienst des C++ ORBs von beliebigen Systemen aus mit Java verwendet werden kann, mußte eine Java-Klasse implementiert werden. Diese Klasse akzeptiert als Eingabeparameter eine Konfigurationsdatei oder eine Property-Umgebungsvariable, die die IOR des Namensservice beinhaltet und den aufzulösenden Namen einer CORBA-Objektimplementierung. Die Objektreferenz wird verwendet, um den Namensdienst zu kontaktieren und den spezifizierten Dienst zu erreichen. Die IOR des Dienstes bildet den Rückgabewert.

5.3. Das Marshaling von Dokumentknoten

Unter dem Marshaling wird das Kopieren der zu übertragenden Daten in den Übertragungspuffer sowie das anschließende Senden verstanden.

Gleichermaßen gilt der Begriff Unmarshaling für das Empfangen und Auslesen der Daten aus dem Übertragungspuffer. Diese Aufgaben werden vom ORB-Kern ausgeführt. Das Leistungsverhalten dieser Komponente des entfernten Operationsaufrufes hängt entscheidend von der Wahl der zu übertragenden Datentypen ab. Dazu gehört die Betrachtung der Typen von Funktionsargumenten und Rückgabewerten, sowie die zu erwartende Menge der zu übertragenden Daten. Bei der Bewertung des Leistungsverhaltens gilt normalerweise, daß bei großen zu übertragenden Datenmengen die für das Weiterleiten der Anfrage an das entsprechende Objekt anfallende Zeit pro Aufruf gegenüber der reinen Übertragungszeit vernachlässigbar gering ausfällt. Andererseits führt ein sehr feinkörniges Datenmodell, das durch häufige Operationsaufrufe mit geringen zu übertragenden Datenmengen gekennzeichnet ist, zu kurzen reinen Übertragungszeiten pro Aufruf, die insgesamt durch eine große Summe von Zeiteinheiten, die für das Weiterleiten der Anfragen anfallen, vergrößert werden. Das Resultat dieser Überlegungen ist ein eher grobkörniges Datenmodell, das wenige Operationsaufrufe mit entsprechend großen in Argumenten zu übertragenden Datenmengen bewirkt.

Um „schlanke“ Clients zu ermöglichen, wurde die Überlegung angestellt, das Parsing von Dokumentknoten serverseitig auszuführen. Im Folgenden werden die beiden Möglichkeiten für das Verteilen von Parsing-Aufgaben gegenübergestellt, um zu zeigen, daß der serverseitige Ansatz von Nachteil ist.

(1) **Parsing auf Serverseite**

Das abhängig von der Dokumentgröße mitunter erheblich zeitaufwendige Parsen wird einmal von der Serverseite vorgenommen und der vorgeparste XML-Graph in einer Datenbank gespeichert. Die zu übertragenden Daten bestehen aus komplexen benutzerdefinierten Datentypen, die als solche nicht in IDL-Typen überführbar sind. Deshalb müssen sie vor dem Marshaling durch den ORB von der Anwendung serialisiert werden.

Vorteile:

a) Die Clients können mit höherer Leistungsfähigkeit agieren, da die Dokumentknoten als vorgeparste Datenstruktur geliefert werden.

b) Es ist eine API konstruierbar, mit Hilfe derer für die Modifizierung anstelle von Dokumenten bzw. Dokumentknoten Anfragen übertragen werden können, die serverseitig in die Dokumentstruktur eingetragen werden.

Nachteile:

a) Die Datenstruktur, die den XML-Graphen repräsentiert liefert nicht die gleiche Unabhängigkeit, wie ein XML-Dokument. Datenstrukturen sind oft von Programmiersprachen abhängig.

b) Da die serialisierte Struktur um den Faktor 10 mal größer ist als das XML-Dokument, tritt eine ebenso größere Netzbelastung auf[2].

(2) **Parsing auf Clientseite**

Die Wahl der zu übertragenden Daten beschränkt sich bei diesem Ansatz ausschließlich auf XML-kodierte Zeichenketten. Die Zeichenketten sind einfacherweise in `octet`-Arrays abzubilden oder als `string` darstellbar. Das Prinzip der Verteilung liegt hier auch der Serverseite zugrunde. Das bedeutet, daß im Zuge der Verteilung die Aufgaben nicht unbedingt zu den Clients migrieren müssen, sondern auch auf mehrere Server verteilt werden können.

Vorteile:

a) Der Server wird vom Parsing der Dokumente entlastet und kann die Rechenzeit für die eigentliche Aufgabe, das Versenden von Dokumentknoten verwenden.

b) Es tritt eine geringere Netzbelastung auf, da ungeparste XML-Daten weniger Platz beanspruchen als geparste XML-Daten.

c) Das Marshaling der Dokumentknoten durch den ORB wird aufgrund des einfachen Datentyps mit höchster Geschwindigkeit erledigt.

Nachteile:

[2] Eine simple XML-Datei von 200 Bytes Länge belegt vorgeparst und serialisiert mehr als 2 KB.

a) Die Clientanwendung wird durch den Parsercode um einige hundert KB vergrößert[3].

Dieser einzige Nachteil ist nur von geringer Bedeutung, da bei einer Anwendung in der Umgebung eines Web-Browsers auf den eventuell integrierten Parser zurückgegriffen werden kann und der Code des Applets nur um Wrapper-Klassen für die Verwendung dieses Parsers erweitert wird. Im Fall einer Applikation müßte der Parser in der Laufzeitumgebung installiert sein.

5.3.1. Die Wahl des Datentyps für die Übertragung

Als grundlegender IDL-Datentyp für die Kapselung des Status von Dokumentknoten kommen einige Möglichkeiten in Frage. Ein Vergleich der Datentypen ist in Tabelle 5.1 zusammengefaßt. Die einfachsten geeigneten Datentypen stellen die Zeichenkettentypen `string` bzw. `wstring` und das Byte-Array `sequence<octet>` dar. Ein von `InputStream` erbender Objekt-Typ ist eine Möglichkeit der Übertragung von XML-Daten zum Java-Parser. Die Handhabung von Strings oder Byte-Arrays bei der Implementierung von Anwendungen ist bei der Verwendung von Java vor allem hinsichtlich der Verfügbarkeit von Stream-Klassen einfach. Ein weiterer geeigneter Datentyp ist der Container-Typ `any`, der beliebige IDL-Typen enthalten kann. Sowohl das Einfügen als auch das Extrahieren von Typen in den `any` kostet allerdings zusätzliche Rechenzeit, die durch den Abstraktionsgewinn nicht gerechtfertigt wird. Dokumente und Bestandteile von Dokumenten sind Objekte, deren hauptsächliche Aufgabe im Kapseln von Daten besteht. Der Aufruf von Operationen auf Objekten dieser Art bzw. die Übertragung dieser Objekte als Argumente von Operationen ist geeignet für den Einsatz der Objects-by-Value Technik [OMG98e], die mit Verwendung des IDL-Typs `valuetype` angewandt wird.

Auf diese Weise werden bei den Client-Anwendungen unabhängige Instanzen der Dokumente erzeugt, deren Manipulation erst zu einem geeigneten Zeitpunkt, zum Beispiel auf Wunsch des Bearbeiters, publiziert wird. Der semantische Unterschied zwischen Value-Types und „gewöhnlichen“ CORBA-Objekten, die immer als Referenz übertragen werden ist

[3] Der „XML4J Version 2.0“ von IBM belegt mehr als 600 KB.

Typ	Vorteile	Nachteile
`any`	alle IDL-Typen lassen sich in diesen Typ einbetten	laufzeitintensiv
`sequence<octet>`	schnelles Marshaling, einfache Implementierung	geringe Abstraktion
`valuetype`	Objekte können *by-value* übertragen werden	noch von wenigen ORBs unterstützt

Tabelle 5.1.: Vor- und Nachteile verschiedener Datentypen

mit dem Unterschied zwischen der meist verwendeten Parameterübergabe als Wert und der als Referenz bei einem Funktionsaufruf der Sprache C vergleichbar. Da die standardisierte Objects-by-Value Variante (viele Hersteller hatten vor CORBA 2.3 ihre eigenen Erweiterungen des Standards implementiert) bisher nur mit wenigen ORBs verfügbar ist, sind Anwendungsarchitekturen, die die Value-Types verwenden zur Zeit nicht vollständig interoperabel[4]. Da Operationen auf „normalen“ CORBA Objekten, die Änderungen am Objektstatus vornehmen aufgrund des Referenzcharakters immer zu „sichtbaren“ Ergebnissen führen, kommt es aufgrund der hohen Anzahl an Aufrufen von remote-Operationen zu starker Server- und Netzbelastung. Außerdem wird die Semantik einer Operation, die Dokumentknoten zur Bearbeitung nach dem *check out*-Prinzip liefert, durch das bloße Übertragen einer Objektreferenz nicht genügend erfüllt. Modifikationen sollen erst auf Benutzerinteraktion hin publiziert werden. Value-Type-Objekte ([OMG98e]), wie sie durch CORBA 2.3 eingeführt wurden ermöglichen das Kopieren von Objekten (*pass-by-value* bzw. *object-by-value*).
Die IDL-Schreibweise eines Value-Types, der den Status eines Dokumentknotens darstellt ist wie folgt:

[4]Der mit dem Netscape Communicator Version 4.5 ausgelieferte Visibroker ist nicht gemäß CORBA 2.3 implementiert.

```
typedef sequence<octet> nodeType;

valuetype nodeValue
{
  private nodeType nodeState;

  factory initNode(in nodeType defaultNode);
};
interface Node
{
  nodeValue getNode(in string name);
};
```

Im Kontext des Clients wird von einem solchen Objekt eine neue Instanz angelegt und deren Status mit dem des Originals initialisiert. Das bedeutet, daß alle weiteren Operationen auf dem Value-Type-Objekt ausschließlich lokale Auswirkungen haben. Erst durch das Zurückschreiben des Objektstatus werden die Änderungen sichtbar. Für die Generierung von Ereignissen, die anderen Clients bzw. interessierten Objekten Änderungen an den Daten anzeigen ist ein separates CORBA-Objekt zuständig.

Herstellerspezifische Erweiterungen

ORB-Implementierungen, die sich an den CORBA 2.1 Standard halten, übermitteln Argumente und Rückgabewerte von Operationen als Objektreferenzen (*pass-by-reference*). Das wirkliche Kopieren von Objekten wird durch diese Technik nicht unterstützt. Dokumentknoten, die *by-reference* übertragen werden, müssen auf Clientseite mit Hilfe einer Caching-Strategie dupliziert werden. Dadurch wird eine ständige „Sichtbarkeit“ der Modifikationen verhindert. Verschiedene Hersteller haben unabhängig von den Standardisierungsbemühungen der OMG eigene Erweiterungen des Standards in ihren Produkten etabliert. Ein Beispiel ist Visigenics Visibroker. Mit den Versionen 3.4 des Visigenic ORB wird ein Werkzeug ausgeliefert, mit dem Java-Interfaces in IDL-Interfaces übersetzt werden können. Dabei werden Java-Datentypen, die in IDL eben-

falls verfügbar sind in die selben übertragen, wohingegen Arrays durch unlimitierte IDL-`sequences` und Java-Objekte durch den neudefinierten Typ `extensible struct` dargestellt werden. Visigenics ORB wendet bei Argumenten, die das Schlüsselwort `extensible` kennzeichnet das *pass-by-value*-Konzept an, wobei Objekte dieser Art beim Marshaling für die Übertragung mit allen Sub-Objekten rekursiv serialisiert werden. Die aus der Übersetzung von Java-Interfaces mit Hilfe des Visibroker-Werkzeugs resultierenden IDL-Interfaces sind Visibroker-spezifisch und für den Fall, daß sie das Schlüsselwort `extensible struct` enthalten nicht mit ORBs anderer Hersteller verwendbar.

5.3.2. Große Datenmengen

Obwohl ein Dokumentknoten im Allgemeinen die Größe von einigen hundert Byte nicht überschreitet, sollte das Interface mit dem Vielfachen arbeiten können. Das heißt, daß weder das sendende, noch das empfangende Objekt bei einer Knotengröße in einem Bereich, der die Java-Laufzeit-Umgebung überfordert durch einen Speicherüberlauf ausfallen darf. Um das Eintreten eines solchen Falles zu vermeiden, kann ein Iterator-Objekt eingesetzt werden, mit Hilfe dessen übergroße Dokumentknoten in Form von kleineren Segmenten übertragen werden. Das Iterator-Muster wird in [Gam95] eingehend beschrieben. Dafür muß die Implementierung des `DocNode`-Interfaces eine private Liste verwalten, die die Operationen `NodeValue getState(in User user)` und `void setState(in NodeValue nodeValue)` wie die `Enumeration`-Methode `nextElement()` agieren läßt, die bei jedem Aufruf das nächste Listenelement referenziert. Dadurch ist eine maximale Größe für die zu übertragenden Segmente einstellbar, deren Wert sich an in Tests gewonnenen Erfahrungen orientiert. Das folgende Code-Segment zeigt zuerst Beispiel-Implementierung unter Verwendung einer Liste:

```
class DocNodeImpl
{
  private Vector docNodeList;

  NodeValue getState(User user)
  {
    ...
```

```
    return (NodeValue)
      (docNodeList.elements.nextElement());
  }
} // class DocNodeImpl
```

und eine Performance-orientierte Variante:

```
class DocNodeImpl
{
  NodeValue getState(User user)
  {
    NodeValue tempNode;
    ...
    System.arraycopy(nodeValue, index, tempNode, 0, length)
    if((index += length) > nodeValue.length)
      index = 0;
    return tempNode;
  }
} // class DocNodeImpl
```

Bei dieser Realisierung ist die Darstellung des `NodeValue` durch ein Byte-Array Voraussetzung. Da das ursprüngliche Array lokal weiterverwendet werden kann, müssen durch die Iteration lediglich Teilstücke des Arrays produziert werden. Eine Verwendung von Value-Types gestaltet diesen Vorgang abstrakter, da hierbei ein Zugriff auf den Type-Code des Value-Types notwendig ist.

Dokumentknoten, deren Größe das definierte Limit eines Elements des Iterator-Objekts nicht überschreitet werden dementsprechend mit nur einem einzigen Operationsaufruf übertragen. Die Iterationsmethode `Object nextElement()` liefert in diesem Fall das vollständige Array.

5.4. External Entities als CORBA-Objekte

Externe Referenzen innerhalb von XML-Dokumenten können in verschieden Formen vorliegen. Normalerweise wird ein Universal Ressource Identifier (URI) [BL94] als System-ID verwendet, der im allgemeinen auf eine Datei im lokalen Dateisystem oder eine über HTTP erreichbare Datei verweist. Alternativ kann eine Public-ID zusätzlich zum

System-ID angegeben werden, die auf einen Eintrag in einem Katalog verweist oder mit Hilfe von speziellen Diensten aufgelöst werden kann. Im Fall von XML-Dokumenten, die durch CORBA-Dienste geliefert werden, kann die Public-ID eine Objektreferenz in einer der Form von URLs ähnlichen Schreibweise enthalten. Diese Objektreferenz könnte auf ein eigenständiges Dokument, das zum Beispiel eine DTD enthält verweisen. Eine zweite Möglichkeit besteht in der Implementierung eines eigenständigen CORBA-Dienstes, der es ermöglicht, DTDs oder andere externe Entities zu registrieren bzw. durch Angabe des Namens anzufordern. Dieser Dienst kann mit der Registrierung einer externen Entity eine Validierung durchführen und sie über ihren Namen mit einem CORBA-Objekte verknüpfen. Das bedeutet, daß auf einen durch externe Referenz verwiesenen Dokumentbestandteil durch eine CORBA-Objektimplementierung zugegriffen werden kann.

XML-Parser, wie der „XML for Java“ von IBM die auf dem SAX-API basieren, stellen eine grundlegende Schnittstelle für das Auflösen von Entities zur Verfügung. Ein CORBA Dienst, der diese Schnittstelle implementiert, muß beim Parser als Dienst registriert werden. Das geschieht mit Hilfe der Funktion,

```
void setEntityResolver(EntityResolver resolver);
```

die im Parser-Interface des SAX definiert ist. Der Aufruf des Dienstes durch den Parser erfolgt dann immer, wenn in einer Referenz eine Public-ID vor einer System-ID angegeben wird. In der Abbildung 5.1 wird der Vorgang des Auflösens einer externen Entity unter Verwendung des CORBA-Dienstes für die Dokumentverteilung dargestellt.

Die Klasse `ObjectResolver` implementiert das abstrakte Interface `EntityResolver`, in dem die Methode

```
InputSource resolveEntity(java.lang.String publicId,
                          java.lang.String systemId);
```

definiert ist. Die Klasse verwaltet eine Datei, in der Objektbezeichner Objektreferenzen zugeordnet sind. Alternativ kann der Inhalt der Datei in einer Datenbank eingetragen werden. Wenn die Parserinstanz, bei der Iteration eines Dokumentes auf eine Public-ID innerhalb einer Entity trifft, wird die Methode `resolveEntity` aufgerufen. Die Methode verwendet

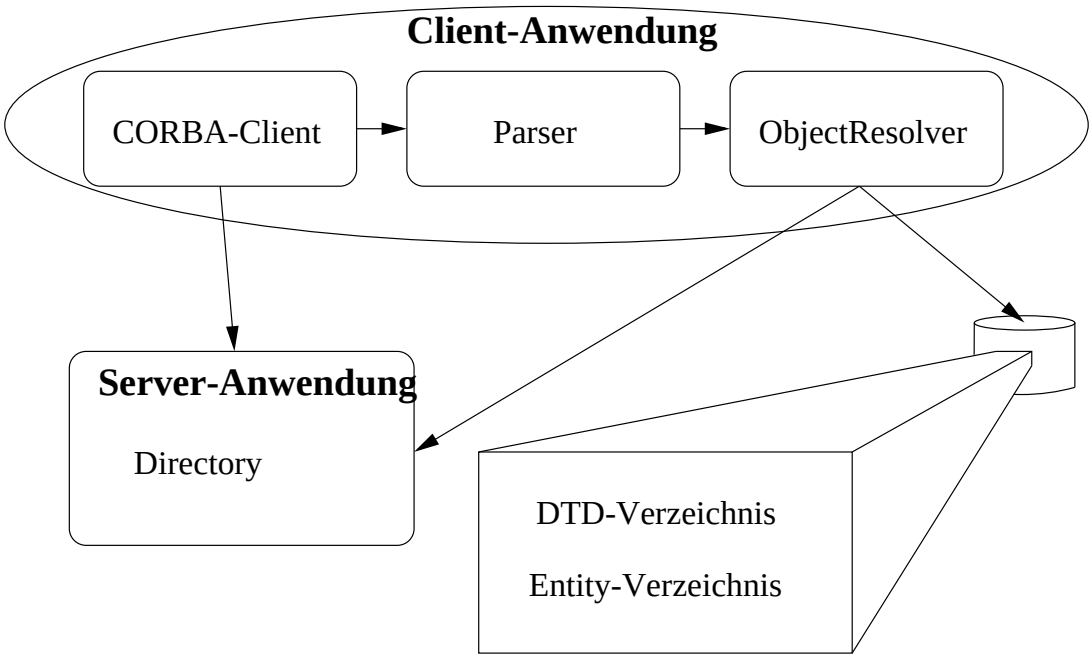


Abbildung 5.1.: Vorgang der Auflösung einer externen Entity

das Argument `publicId` als Bezeichner für das referenzierte Objekt und tätigt selbständig einen Operationsaufruf auf dem CORBA-Objekt. Dieser Aufruf liefert die gewünschte Entity in Form eines Byte-Streams. Falls das CORBA-Objekt nicht über einen aktiven Servant verfügt bzw. nicht existiert, wird versucht, den Byte-Stream aus einer Datei im lokalen Dateisystem des `ObjectResolver`-Objektes zu erzeugen. Abhängig von der Anzahl der externen Entities mit Referenz zu einem CORBA-Objekt kann ein Parse-Vorgang sehr lange dauern (es können Rekursionen auftreten) oder durch das Eintreten einer CORBA-System-Exception unterbrochen werden. Diese Ausnahme tritt im Kontext des Objektes auf, das den Parser instanziert hat und für das Übertragen des Dokumentknotens verantwortlich ist.

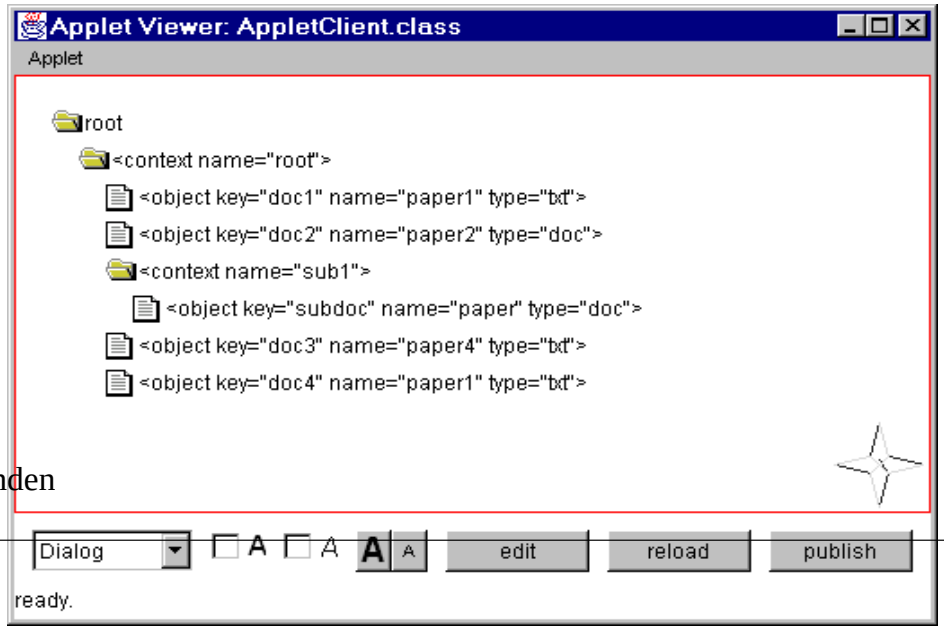
5.5. Konfiguration

Serverdienst

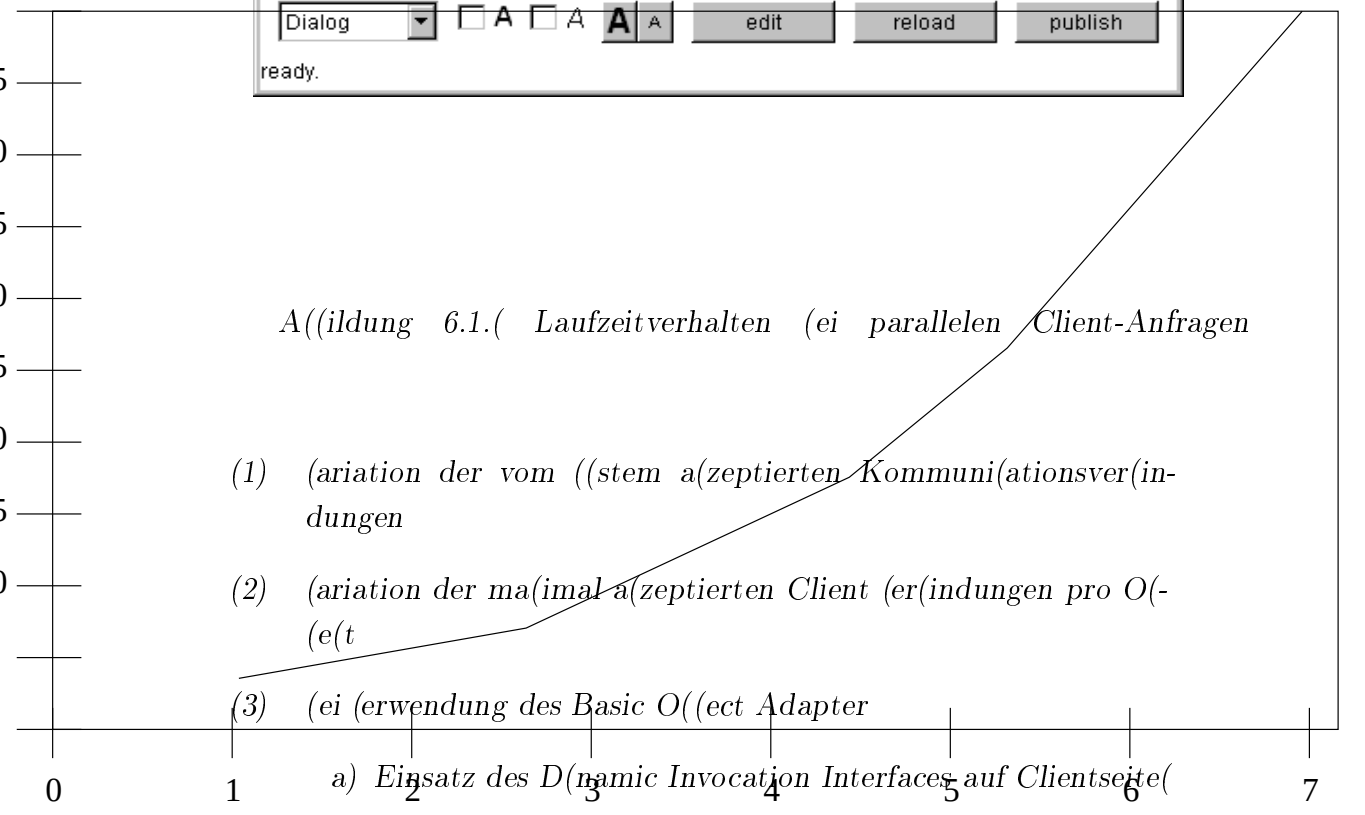
Die Serveranwendung wird von einem Singleton-Objekt (siehe [Gam95]) aus verwaltet, das das Bootstrapping des Namens- und Nachrichtendienstes ausführt. Danach wird jeweils eine Instanz des Dienstes für die Datenbankanbindung (`BaseConnector`) und des Verzeichnisdienstes (`Directory`) erstellt. Der Verzeichnisdienst liest sofort nach seinem Start den Inhalt des Dokumentverzeichnisses in Form eines XML-Dokumentes aus der Datenbank. Die Operationen des `Directorys` erzeugen eine User-Exception vom Typ `TimeOut`, wenn das Dokumentverzeichnis nicht existiert. Dies kann der Fall sein, wenn es beim Verbindungsaufbau zur Datenbank zu Verzögerungen kommt. Der Datenbank und der Verzeichnisdienst registrieren sich bei dem Namensdienst. Ihre Operationen können von da an durch Clients aufgerufen werden. Der Aufruf der Check-Out-Operation veranlaßt den Verzeichnisdienst, den in den Argumenten enthaltenen Dokumentnamen mit den Daten im aktuellen Verzeichniskontext zu vergleichen und bei Übereinstimmung die Objektreferenz dieses Dokumentes zu liefern.

Client

Für die Ausführung von Tests des Laufzeitverhaltens wurde ein Client in der Form eines Applets implementiert. Er beinhaltet ein Anzeigemodul für die Darstellung der Baumstruktur eines XML-Dokumentes und ermöglicht das Editieren von Dokumenten, die durch den zentralen CORBA-Dienst verwaltet werden. In der Abbildung 5.2 ist der Inhalt des internen Verzeichnisses des Dokumentenverteilungsdienstes dargestellt. Nach dem Start erfragt der Client die Objektreferenz des Verzeichnisdienstes beim Namensdienst und ruft danach erstmalig dessen `DocNode list()`-Operation auf, um den Inhalt des Verzeichnisses zu erhalten. Die gelieferte DOM-Datenstruktur wird verwendet, um den Verzeichnisinhalt durch das Anzeigemodul sichtbar werden zu lassen. Innerhalb eines Verzeichniskontextes selektierte Einträge von Dokumenten können durch Aufruf der Check-Out-Operation des Verzeichnisdienstes geladen und in einem separaten Anzeigemodul dargestellt werden. Dies geschieht durch das Betätigen des „edit“-Buttons. Das Zurückschreiben



ntwortzeit in Sekunden



A((ildung 6.1.(Laufzeitverhalten (ei parallelen Client-Anfragen

(1) (ariation der vom ((stem a(zeptierten Kommuni(ationsver(indungen

(2) (ariation der ma(imal a(zeptierten Client (er(indungen pro O(-(e(t

(3) (ei (erwendung des Basic O((ect Adapter

a) *Einsatz des D(namic Invocation Interfaces auf Clientseite(*

(4) (ei (erwendung des (orta(le O((ect Adapter

a) *Einsatz von Multithread-(ervern*

() *Einsatz multipler (OAs*

c) *Einsatz von Default (ervants*

Die mit einer Implementierung unter Verwendung des POA und dem Einsatz von Default Servants durchgeführten Tests ergaben die in Abbildung 6.1 zusammengefaßten durchschnittlichen Antwortzeiten. Der extreme Anstieg der Kurve schon bei einer geringen Anzahl von fünf gleichzeitig anfragenden Clients ist mit dem Umstand zu erklären, daß das CORBA-Object `DocNode` für jede `getState`-Operation eine Datenbankabfrage generiert. Dieses Verhalten läßt sich durch eine Caching-Strategie verbessern, die mehrfach nachgefragte Objekte im Speicher der Anwendung beläßt.

6.3. Zusammenfassung

Die während der Implementierung gesammelten Erfahrungen zeigen außerdem, daß die Verbesserung von häufig durchlaufenen Programmteilen hinsichtlich ihrer Performanz eine deutliche Veränderung des Gesamtverhaltens bewirkt. Obwohl der Fall des zeitlich genauen Übereinstimmens von Clientanfragen in der Praxis selten ist, wird die für diese Arbeit durch die Dokumentspeicherung in *einer* zentralen Datenbank getroffene Vereinfachung den Anforderungen einer realen Benutzerumgebung wahrscheinlich nicht gerecht.

Da durch die Architektur keine Einschränkungen hinsichtlich der Dokumentenspeicherung getroffen werden, ist die Skalierbarkeit nur mittelbar durch diese prototypische Implementierung begrenzt. Die Vorteile einer vollständig objektorientierten Architektur zeigen sich erfahrungsgemäß in Eigenschaften wie hoher Abstraktion, Erweiterbarkeit, Wiederverwendbarkeit und einfacherer Wartbarkeit. Laufzeitnachteile, die sich oft aufgrund umfangreicher Objektkommunikation und hohem -speicherbedarf ergeben, können sich durch die Weiterentwicklung der Objektumgebung verbessern. CORBA wird ständig weiterentwickelt. Die zukünftige Integration programmiersprachlicher Details, wie Streams in IDL könnte einigen Entwicklungsaufwand, Fehlerquellen und Performanzdefizite auf der Seite der Anwendung verringern.

7. Zusammenfassung und Ausblick

Im Verlauf der Diplomarbeit ist die Architektur eines Dienstes für die Verteilung und Bearbeitung von Dokumenten erarbeitet worden. Verschiedene Möglichkeiten der serverseitigen und clientseitigen Implementierung wurden prototypisch entwickelt und getestet. Dabei wurde vor allem die geringe Granularität des Document Object Model als problematisch für eine skalierbare Architektur eingeschätzt. Die Schwierigkeiten, die sich indirekt durch dieses Modell ergeben sind aber eher mit der zusätzlichen Objektfunktionalität verknüpft, die durch die Verfügbarkeit in einer CORBA-Umgebung notwendigerweise hinzukommen. Die Eignung von XML als Beschreibungssprache für verteilte Dokumente bleibt von dieser Eigenschaft unberührt. Eine Strategie zur Optimierung, die von einer Zusammenfassung mehrerer Knoten zu einem Objekt ausgeht wurde experimentell entwickelt und getestet. Diese Strategie ist aber stark von der Menge von Daten abhängig, die durch Dokumentknoten repräsentiert wird und erfordert deshalb eine dynamische Anpassung, deren Struktur nur schwer in das Datenmodell integrierbar ist. Die Möglichkeit einer Multithread-Implementierung könnte die Problematik der schlechten Skalierbarkeit bei parallelen Clientzugriffen entschärfen. Diese zukünftige Arbeit ist stark von der voranschreitenden Entwicklung der CORBA-Spezifikation und ihrer Umsetzung in den Produkten der Hersteller abhängig. Die Realisierung von in der Architektur integrierten Sicherheitsmechanismen unter Nutzung des CORBA Security Service stellt ebenfalls eine sinnvolle weitere Arbeit dar.

A. IDL für die vollständige DOM-Kapselung (Auszug)

Aus Platzgründen enthält dieser Anhang nur die Abbildung des `Node`-Interfaces. Die anderen 17 DOM-Interfaces[W3C98a] sind entsprechend aufgebaut.

```
interface DocNode
{
    string getNodeName();
    NodeValue getNodeValue();
    void setNodeValue(NodeValue nodeValue)
      raises DOMException;
    short getNodeType();
    DocNode getParentNode();
    DocNodeList getChildNodes();
    DocNode getFirstChild();
    DocNode getLastChild();
    DocNode getPreviousSibling();
    DocNode getNextSibling();
    NamedDocNodeMap getAttributes();
    DocNode getOwnerDocument();
    DocNode insertBefore(DocNode newChild,
                         DocNode refChild)
      raises DOMException;
    DocNode replaceChild(DocNode newChild,
                         DocNode oldChild)
      raises DOMException;
    DocNode removeChild(DocNode oldChild)
      raises DOMException;
    DocNode appendChild(DocNode newChild)
```

```
      raises DOMException;
    boolean hasChildNodes();
    DocNode cloneNode(boolean deep);
};
```

B. IDL des CORBA-Dienstes

```
module Manager
{
    typedef string Owner;
    typedef string User;
    typedef sequence<octet> NodeType;

    enum Permissions { READ, WRITE };

    valuetype NodeValue
    {
        private NodeType nodeState;
        private DocNode parent;
        private DocNodeList nodeList;
        private Owner owner;
        private Permissions permissions;

        factory initNode(in NodeType defaultNode);
    };

    interface DocNode
    {
        exception NotAllowed{};

/* The getState operation return this nodes state.     */
        NodeValue getState(in User user);

/* The setState operation sets this nodes state.       */
        void setState(in NodeValue nodeValue)
            raises (NotAllowed);
```

```
    };

    interface Directory
    {
        struct NameComponent
        {
            string name;
            string type;
        };

        typedef sequence <NameComponent> Name;

        exception NotEmpty{};
        exception NotFound{};
        exception AlreadyBound{};
        exception InvalidName{};
        exception InUse{};
        exception TimeOut{};

/* The following operations perform a service similar  */
/* to the CosNaming Service.                           */
        void bind(in Name name, in DocNode node)
            raises (NotFound, AlreadyBound, TimeOut);
        void unbind(in Name name)
            raises (NotFound, InUse, TimeOut);
        DocNode resolve(in Name name)
            raises (NotFound, TimeOut);
        void createContext(in Name name)
            raises (NotFound, AlreadyBound, InvalidName, TimeOut);
        void destroy(in Name name)
            raises (NotFound, NotEmpty, TimeOut);
        DocNode list()
            raises (TimeOut);

/* The checkIn operation activates all objects ser-    */
/* ving the subtree and recursively sets their state.  */
        void checkIn(in DocNode docNode)
```

```
            raises (InUse, TimeOut);

/* The checkOut operation activates all objects serving*/
/* the document recursively and returns the root node. */
        DocNode checkOut(in Name name, in User user)
            raises (NotFound, InUse, TimeOut);
    };

    interface BaseConnector
    {
        exception AlreadyExists{};
        exception NotFound{};

/* The following operations perform database           */
/* connection services.                                */
        void createObject(in NodeValue nodeValue)
            raises (AlreadyExists);
        void updateObject(in NodeValue nodeValue)
            raises (NotFound);
        void dropObject(in string name, in string type)
            raises (NotFound);
        NodeValue getObject(in string name, in string type)
            raises (NotFound);
  };
};
```

Literaturverzeichnis

[A.97] Thomas J. Mowbray; William A. *Inside CORBA Distributed Object Standards.* Addison-Wesley, 1997.

[BL94] T. Berners-Lee. *Universal Resource Identifiers in WWW - A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World-Wide Web.* http://rfc.fh-koeln.de/rfc/html/rfc1630.html, 1994.

[CG99] CORBA and Distributed Systems Research Group. *CORBA Comparison Project.* http://nenya.ms.mff.cuni.cz/thegroup/COMP/Report_0899.pdf, 1999.

[Dow98] Troy Bryan Downing. *Java RMI, Remote Method Invocation, Distributed Databases, Peer-To-Peer, Object Serialization, RMI Security Issues, Java RMI APIs.* IDG Books Worldwide, 1998.

[Fow97] Martin Fowler. *Analysis Patterns, Reusable Object Models.* Addison-Wesley, 1997.

[Gam95] E. Gamma. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.

[Har98] Robert Orfali; Dan Harkey. *Client/Server Programming with Java & CORBA.* Jon Wiley & Sons, Inc., 2nd edition, 1998.

[Lor91] J. Rumbaugh; M. Blaha; W.Premerlani; F. Eddy; W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall International Editions, 1991.

[May95] Peter Coad; David North; Mark Mayfield. *Object Models, Strategies, Patterns & Applications.* Prentice Hall, 1995.

[Meg98] David Megginson. *The Simple API for XML.* http://www.megginson.com/SAX/, 1998.

[Mic80] Sun Microsystems. *Remote Procedure Call Protocol Specification.* http://rfc.fh-koeln.de/rfc/html/rfc1057.html, 2nd edition, 1980.

[Mic98] Sun Microsystems. *JDBC(TM) Technology.* http://java.sun.com/products/jdbc/index.html, 1998.

[Mic99] Sun Microsystems. *RMI over IIOP.* http://java.sun.com/products/rmi-iiop/index.html, 1.01st edition, 1999.

[Net99] Netbula. *Java RPC Internet enable RPC applications.* http://netbula.com/javarpc/, 1999.

[NWbI98] Inc. NC World by ITWorld.com. *Just In Time for Java vs. C++.* http://www.ncworldmag.com/ncworld/ncw-01-1998/ncw-01-jperf.html, 1998.

[OMG97] Object Management Group OMG. *Persistent Object State Service.* ftp://ftp.omg.org/pub/docs/formal/97-12-12.pdf, 1997.

[OMG98a] Object Management Group OMG. *The Common Object Request Broker: Architecture and Specification.* ftp://ftp.omg.org/pub/docs/formal/98-12-01.pdf, 2.3rd edition, 1998.

[OMG98b] Object Management Group OMG. *CORBA Messaging, JointRevisedSubmission.* ftp://ftp.omg.org/pub/docs/orbos/98-05-05.pdf, 1998.

[OMG98c] Object Management Group OMG. *CORBA OMG IDL.* http://www.omg.org/library/corbidl.html, 1998.

[OMG98d] Object Management Group OMG. *CORBA-Services Specification.* ftp://ftp.omg.org/pub/docs/formal/98-07-05.pdf, 1.2nd edition, 1998.

[OMG98e] Object Management Group OMG. *Objects By Value, JointRevisedSubmission.* ftp://ftp.omg.org/pub/docs/orbos/98-01-18.pdf, 1998.

[OMG98f] Object Management Group OMG. *Security Service Specification.* ftp://ftp.omg.org/pub/docs/formal/98-12-21.pdf, 1.2nd edition, 1998.

[OMG99a] Object Management Group OMG. *CORBA 2.3 - Mapping of OMG IDL to C++. Preliminary Specification.* ftp://ftp.omg.org/pub/docs/formal/99-07-45.pdf, 1999.

[OMG99b] Object Management Group OMG. *CORBA 2.3 - Mapping of OMG IDL to Java. Preliminary Specification.* ftp://ftp.omg.org/pub/docs/formal/99-07-57.pdf, 1999.

[Pos81] J. Postel. *Transmission Control Protocol - DARPA Internet Program Protocol Specification.* http://rfc.fh-koeln.de/rfc/html/rfc0793.html, 1981.

[Rum97] James Rumbaugh. *Object-Oriented Modeling & Design.* Prentice Hall, 1997.

[Sie98] Geoffrey Lewis; Steven Barber; Ellen Siegel. *Programming with Java IDL.* Macmillan, 3rd edition, 1998.

[Ste98] James Gosling; Bill Joy; Guy Steele. *The Java Language Specification.* http://java.sun.com/docs/books/jls/html/index.html, 1998.

[Vin97] Douglas C. Schmidt; Steve Vinoski. *Object Adapters: Concepts and Terminology. SIGS, Vol. 9, No. 11.* http://www.cs.wustl.edu/

[Vin99a] Douglas C. Schmidt; Steve Vinoski. *Programming Asynchronous Method Invocations with CORBA Messaging. Vol. 11, No. 2.* http://www.cs.wustl.edu/

[Vin99b] M. Henning; S. Vinosky. *Advanced CORBA Programming with C++.* Addison-Wesley, 1999.

[W3C98a] World Wide Web Consortium W3C. *Document Object Model (DOM) Level 1 Specification.* http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/DOM.pdf, 1st edition, 1998.

[W3C98b] World Wide Web Consortium W3C. *Extensible Markup Language (XML) 1.0 Specification.* http://www.w3.org/TR/1998/REC-xml-19980210.pdf, 1st edition, 1998.

[W3C99] World Wide Web Consortium W3C. *Character Model for the World Wide Web.* http://www.w3.org/TR/1999/WD-charmod-19990225.html, 1999.

[Yel99] Tim Lindholm; Frank Yellin. *The Java Virtual Machine Specification.* http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html, 2nd edition, 1999.

Erklärung

Ich versichere, daß ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Leipzig, Februar 2000