

Yves Langisch

**Anbindung eines
Link-Management-Systems an einen
Web-Server**

*Diplomarbeit DA-2001.13
Wintersemester 2000/2001*

*Betreuer: Dr. Erik Wilde,
Co-Betreuer: Prof. Dr. Bernhard Plattner*

*Verantwortlicher:
Prof. Dr. Bernhard Plattner*

*Departement of Electrical Engineering
Computer Engineering and Networks Laboratory
Prof. Dr. Bernhard Plattner*

Diplomarbeit

Anbindung eines Link-Management-Systems an einen Web-Server

Wintersemester 2000/2001

Professor:

Prof. Dr. Bernhard Plattner plattner@tik.ee.ethz.ch

Betreuer:

Dr. Erik Wilde net.dret@dret.net

Student:

Yves Langisch yves@langisch.ch DS D-INFK

Vorwort

Ich erinnere mich noch genau an jenen Tag als ich den für mich neuen Begriff der Linkdatenbank zum erstenmal gehört habe. Es war dies beim Durchlesen der ausgeschriebenen Diplomarbeiten am TIK. Als ich schliesslich noch den Begriff XML entdeckte, war das Interesse bei mir definitiv geweckt, da ich mich sehr für neue Internet-Technologien interessiere. Ich wusste zu diesem Zeitpunkt zwar in groben Zügen was XML ist, aber ich hatte weder jemals damit gearbeitet noch eine konkrete Anwendung davon gesehen. So hoffte ich mich während der Diplomarbeit mit Themen auseinandersetzen zu können, mit denen ich mich nicht schon während des ganzen Studiums beschäftigt habe. Ich machte also mit Erik Wilde einen Termin ab, um mehr über diese Arbeiten zu erfahren. Während dieses Gespräches bestätigte es sich, dass die Basistechnologie des XLinkbase Projektes XML ist. Zugegebenermassen konnte ich mir nach dem Gespräch noch nicht sehr viel unter einer XLinkbase vorstellen. Ich wusste aber, dass viel Konzeptarbeit gefragt war, da das Projekt noch ganz am Anfang stand. Dies war ein weiterer Punkt, der mich reizte. Während meiner Semesterarbeiten hatte ich nie die Möglichkeit etwas ganz neues zu entwickeln. Meine Arbeit war immer Teil eines bereits bestehenden Ganzen. So entschied ich mich einige Tage später dafür, den XLinkbase Server zu konzipieren und darauf basierend einen Prototypen zu erstellen.

Rückblickend kann ich sagen, dass es sehr interessant war mit den eingesetzten Technologien (XML und J2EE) zu arbeiten. Die 4 Monate waren sehr lehrreich und ich konnte viel praktisches Know-How im Bereich Middleware mitnehmen. Die Zusammenarbeit mit dem Team des XLinkbase Client habe ich als sehr angenehm empfunden. Ich war sozusagen in ein Team involviert, konnte aber trotzdem immer selbständig arbeiten.

Mein Bericht soll es ermöglichen, möglichst rasch ein Verständnis für diese relativ komplexe Umgebung zu bekommen um so eine Weiterentwicklung des XLinkbase Servers in Angriff nehmen zu können.

An dieser Stelle möchte ich mich noch herzlich bei Erik Wilde für seine stete Hilfsbereitschaft bedanken. In diesem Sinne wünsche ich der Weiterführung des XLinkbase Projektes gutes Gelingen.

Yves Langisch
Zürich, im März 2001

Zürich, 1.3.01

Herrn
Yves Langisch

Aufgabenstellung Diplomarbeit WS 2000/2001 für Yves Langisch

Im XLinkbase-Projekt entstehen ein Modell und eine Implementierung einer Datenbank, die spezifisch dafür ausgelegt ist, beliebige Ressourcen-Informationen zu sammeln (meist Referenzen auf Internet-basierte Ressourcen in Form von HTTP- oder FTP-URIs) und in komplexe inhaltliche Zusammenhänge zu bringen. Das zugrundeliegende Modell verwendet an vielen Stellen XML-basierte Technologien, um die verschiedenen Komponenten des Gesamtsystems miteinander zu verbinden. Die Grundarchitektur des Systems ist ein Client/Server-Modell, in dem der Client entweder ein normaler HTML-basierter Web-Browser, ein XML/XLink-Browser der nächsten Generation, oder auch ein spezialisierter Client sein kann, der spezifisch für den XLinkbase-Server programmiert ist und die Informationen in der XLinkbase in anschaulicher Weise graphisch modelliert. Der XLinkbase Server ist ein durchgängig XML-basiertes System, das über HTTP mit dem Client kommuniziert und die Requests und Responses über eine flexibel konfigurierbare Menge von XSLT Style Sheets verarbeitet. Die eigentliche Speicherkomponente ist ebenfalls austauschbar, im Normalfall wird dies jedoch entweder ein relationales Datenbanksystem oder eine XML-Lösung sein.

Im Rahmen der Diplomarbeit sollen an Design und Implementierung des XLinkbase-Servers sowie des Zugriffsprotokolls mitgewirkt werden. Dabei geht es im speziellen um die möglichst flexible und effiziente Integration verschiedener XML-basierter Komponenten in die Gesamtarchitektur des Servers. Vieles der Funktionalität der XLinkbase wird in XSLT Style Sheets programmiert werden, deshalb wird besonderes Augenmerk darauf gelegt, eine Architektur zu verwenden, die diesem Umstand Rechnung trägt (z.B. ein auf EJB oder einfachen JavaBeans aufsetzendes komponentenbasiertes Modell, das die XSLT Style Sheets in Translets übersetzt und in Komponenten kapselt). Ziel der Diplomarbeit ist es, einen funktionsfähigen Prototypen zu konzipieren und zu erstellen, der im Zusammenhang mit dem in einer anderen Diplomarbeit erstellen Client zu ersten Tests und Demonstrationen verwendet werden kann.

Kurzfassung

Das Ziel dieser Diplomarbeit war die Konzeptionierung einer Linkdatenbank, die XLinkbase genannt wird, und anschliessender Implementierung eines darauf basierenden Prototypen um erste Demonstrationen zu ermöglichen. Die Idee dahinter, die von Erik Wilde stammt, ist, den unzähligen existierenden URIs einen Kontext zu geben, indem man sie in ein semantisches Netz einordnet und sie in Beziehung zueinander stellt. Der XLinkbase Server soll dann sozusagen in der Lage sein, dieses Netz zu verwalten und Anfragen von XLinkbase Clients sowie externen Clients wie z.B. einer Suchmaschine darauf zu erlauben.

Nach der Einarbeitung in das Thema von XML und Topic Maps, habe ich ein Grobkonzept aufgestellt, das einerseits die Anforderungen bzw. Vorgaben an den Server, andererseits bereits eine Framework-unabhängige Architektur beinhaltet. Wichtige Aspekte waren unter anderem, dass ein komponentenbasierter Ansatz, der Flexibilität, Skalierbarkeit und Erweiterbarkeit mit sich bringen soll, gewählt wird. Um den heutigen Anforderungen von Enterprise Applikationen gerecht zu werden, wurde auch noch Verteilung mit in das Pflichtenheft aufgenommen. Die Grobarchitektur zeigt bereits eine vollständig komponentenbasierte Architektur. Die Clients kommunizieren ausschliesslich über HTTP mit dem Server oder greifen über eine eigene Komponente im XLinkbase Server direkt auf die Daten zu. HTTP wurde gewählt, weil der typische Client ein Browser sein wird. Dieser erste Entwurf einer Architektur des XLinkbase Servers zeigte rasch, dass ein traditioneller Client/Server-Ansatz den Anforderungen nicht genügen würde. Es war also eine Middleware Technologie gefragt.

Zur Auswahl standen grundsätzlich die drei populärsten Frameworks für Multi-tier Applikationen: CORBA, die Java 2 Platform / Enterprise Edition und die von Microsoft vielfach propagierte .NET Platform. Da die .NET Platform bis zum heutigen Zeitpunkt noch sehr stark Plattform-abhängig ist, schied diese relativ früh aus. Microsoft will die neue Programmiersprache C# zwar auch auf anderen Plattformen etablieren, bis dies aber der Fall

sein wird, wird noch einige Zeit vergehen, ganz zu schweigen davon, ob man dann tatsächlich auch COM-Komponenten einsetzen kann. In der engeren Auswahl standen schliesslich noch CORBA und J2EE. Die zwei Standards sind sich eigentlich relativ ähnlich, wobei CORBA eher in Richtung Verteilung und J2EE eher Richtung Webtechnologien zielt. Zudem ist CORBA vergleichsweise um einiges komplexer als J2EE. Ein Punkt, der für CORBA gesprochen hätte, wäre die höhere Performanz. Dies alleine hätte den Einsatz aber noch nicht gerechtfertigt. Somit habe ich mich für die J2EE Plattform entschieden und kann im Nachhinein sagen, dass dies eine gute Entscheidung war.

Nachdem ich nun wusste, dass J2EE zum Einsatz kommt, habe ich mich an die Konzeptionierung der Details gemacht. Zunächst stand aber wieder ein Entscheid bevor. Ich war mir uneinig, ob ich das Web Publishing Framework Cocoon einsetzen oder das Ganze mit Enterprise Java Beans (EJBs) neu designen soll. Ich entschied mich schliesslich gegen Cocoon, da dieses, unter anderem, Verteilung nur sehr schlecht unterstützt. Das Detailkonzept geht davon aus, dass jeder Funktionalität eine eigene Komponente zugeordnet wird (z.B. eine Logger- oder Parser-Komponente) und der Request, abhängig vom Inhalt, durch die verschiedenen Komponenten geroutet wird, bis am Ende der Kette die eigentlichen Daten aus einer Datenquelle gelesen werden. Dabei sollen vorwiegend sogenannte Message Driven Beans (MDBs), die asynchrones Messaging über den Java Messaging Service erlauben, zum Einsatz kommen. Dadurch wird es elegant möglich, die Response durch andere Komponenten laufen zu lassen, als der ursprüngliche Request. Am Anfang der Kette ist ein Servlet positioniert, das für die HTTP-Kommunikation mit den Clients zuständig ist. Die Servlet Komponente empfängt also HTTP-Requests und leitet diese an das erste Bean in der Kette weiter und empfängt dann schliesslich die Responses wieder um sie an die Clients zurückzugeben. Für das Parsing und die XLST-Transformationen von XML-Dokumenten, sind zwei zentrale Komponenten vorgesehen: eine Parser- und XSLT-Prozessor Komponente in der Form von Stateless Session Beans. Eine Komponente, die für die Wegleitung der Requests zuständig ist, wäre eine weitere zentrale Komponente.

Nach den diversen Tests von Applikationsservern und Entwicklungsumgebungen, die zum Teil leider noch sehr fehlerhaft sind, war die eigentliche Implementierung des Prototypen mehr oder weniger 'straight forward'. Im Prototypen sind das Servlet und drei EJBs implementiert, wobei das letzte Bean in der Kette die angeforderten Daten nicht aus einer Datenbank sondern vom Dateisystem holt. Auch ist eine zentrale Parser-Komponente vorhanden, die dazu eingesetzt wird den XML-Request zu parsen.

Abschliessend kann ich sagen, dass am Schluss meiner Arbeit ein Konzept und ein Prototyp, der erfolgreich mit dem XLinkbase Prototypen zusammenarbeitet, vorlagen. Die diversen Tests, die wir gemacht haben, zeigten, dass basierend auf dem entwickelten Konzept, die fehlenden Teile (z.B. das Routing und die XLinkbase Query Language) implementiert werden können.

Inhaltsverzeichnis

Vorwort	iii
Aufgabenstellung	v
Kurzfassung	vii
1 Einführung	3
2 Grobkonzept	5
2.1 Anforderungen/Vorgaben	5
2.2 Grobarchitektur	7
3 Technologien/Standards	9
3.1 CORBA	11
3.1.1 Common Object Services (CORBAservices)	11
3.1.2 Common Facilities (CORBAfacilities)	14
3.2 J2EE	14
3.2.1 EJB	16
3.3 Microsoft's .NET Platform	18
4 Evaluation	21
4.1 XML-Unterstützung	21
4.2 Verteilung	22
4.3 Plattformunabhängigkeit	22
4.4 Produkteunabhängigkeit	23
4.5 Performanz	23
4.6 Verbreitung	23
4.7 Komplexität	23
4.8 Zusammenfassung	24
5 Detailkonzept	27
5.1 Architektur	28

5.1.1	Clients	28
5.1.2	Application Server und Container	29
5.1.3	EJBs	30
5.1.4	Servlets	31
5.2	Kommunikation	32
5.2.1	Client/Server Kommunikation	32
5.2.2	Serverinterne Kommunikation	33
6	Implementierung	35
6.1	Entwicklungsumgebung	36
6.2	Das Kicker Servlet	38
6.3	Das Controller Bean (MDB)	41
6.4	Das RequestHandler Bean	43
6.5	Der Parser	45
7	Ausblick	49
A	Document Type Definitions	55
A.1	Request DTD	55
A.2	XLinkbase DTD Version 0.4	55
B	Java Source Code	61
B.1	Kicker Servlet	61
B.2	Controller Bean	66
B.3	RequestHandler Bean	69
B.4	Parser (Stateless Session Bean)	73
B.4.1	Home Interface	73
B.4.2	Remote Interface	73
B.4.3	Bean Implementierung	73
C	CD-ROM	77
	Literaturverzeichnis	79

Tabellenverzeichnis

4.1 Vergleich der Technologien	24
--	----

Abbildungsverzeichnis

2.1	Eine komponentenbasierte Server-Architektur	7
3.1	Die Common Object Request Broker Architecture	11
3.2	Die J2EE Architektur	15
3.3	Evolution von Windows DNA nach .NET	18
5.1	Die Cocoon Architektur	27
5.2	Die XLinkbase Architektur	28
6.1	Die Architektur des Prototypen	36
6.2	Die Weblogic Management Konsole	37

Kapitel 1

Einführung

Das Internet wächst von Tag zu Tag mit einer atemberaubenden Geschwindigkeit. So wurden die Anzahl Webseiten im November 2000 auf rund 500 Millionen geschätzt und jeden Tag sollen weitere 2 Millionen dazukommen [1]. Aufgrund dieser Tatsache ist es offensichtlich, dass Links zwischen Ressourcen immer wichtiger werden. Es drängt sich aber die Frage auf, was die vielen Ressourcen bedeuten und wie sie zusammenhängen. Ohne, kann diese Frage aber nicht beantwortet werden. Man müsste also die URIs sozusagen in einen Kontext einordnen können. Genau an diesem Punkt setzt die Idee der Linkdatenbank XLinkbase an.

Die Idee der XLinkbase ist, die Ressourcen in eine Art semantisches Netz, das Beziehungen zwischen den Knoten erlaubt, einzuordnen. Das semantische Netz basiert auf einer Erweiterung des Topic Map Standard [3]. Hat man einmal ein semantisches Netz aufgebaut, könnte man sich eine Abfragesprache für komplexe Queries vorstellen, um die Links wieder aus der Datenbank holen zu können. In einer XLinkbase werden somit nur Metainformationen gespeichert. Eine solche Funktionalität kann z.B. zur Ergänzung zu einem bestehenden Content Management System (CMS), zur Generierung von gutem Hypermedia oder für Knowledge Management, wo oft viele heterogene Datenquellen auftreten, eingesetzt werden.

Meine Aufgabe war es, diese Idee in ein Konzept umzusetzen und einen Prototypen der Kernkomponente, des XLinkbase Servers, zu erstellen. Die Reihenfolge der Kapitel entspricht der chronologischen Reihenfolge in der ich gearbeitet habe. Es folgt zuerst ein Kapitel über die Vorgaben bzw. Anforderungen, die an die Serverkomponente gestellt wurden, und eine erste grobe Architektur. Ausgehend von diesem Grobkonzept werden drei Frameworks/Technologien vorgestellt, die sich für die Architektur und Implementierung der XLinkbase eignen würden. Diese drei Technologien werden

schliesslich im darauffolgenden Kapitel evaluiert und in Bezug zum Projekt auf ihre Tauglichkeit hin geprüft. Mit dem gewählten Framework als Grundlage liess sich dann ein detaillierteres Konzept erstellen, das in einem separaten Kapitel beschrieben ist. Schliesslich wird im nächsten Kapitel auf die Details der Implementierung eingegangen. Die beiden letzten Kapitel beschreiben die Implementierungsdetails des Prototypen und Möglichkeiten zur Weiterentwicklung des XLinkbase Servers.

Kapitel 2

Grobkonzept

2.1 Anforderungen/Vorgaben

Bereits vor Beginn der Diplomarbeit war klar, dass ein grosser Teil der Zeit für die Kozeptionierung des Servers verwendet werden musste, da die XLinkbase mehr eine Idee war als ein funktionierendes System. Einzig das Glossary zu Erik Wilde's Buch basierte auf der XLinkbase Idee [4]. Als weiterer Ausgangspunkt der Arbeit diente eine DTD, die das XLinkbase Superschema definiert Anhang A.2 auf Seite 55, auf welchem auch genanntes Glossary basiert.

Die Aufgabenstellung selbst definiert sozusagen den Rahmen oder die Eckpunkte der Anforderungen. Wie aber die Anforderungen zwischen diesen Eckpunkten aussehen, musste zuerst noch definiert werden. Zu diesem Punkt habe ich mir einige Gedanken gemacht. Ich versuchte diese Anforderungen nicht im Kontext von produktespezifischen Möglichkeiten zu definieren, sondern losgelöst davon und irgendwelchen technischen Möglichkeiten.

Aus der Aufgabenstellung heraus stellen sich folgende Grundanforderungen:

XML Die Architektur des Gesamtsystems soll auf XML basieren.

N-tier Architektur Das Ganze soll auf einer mehrstufigen Architektur basieren um Business Logic serverseitig implementieren zu können und um somit einen schlanken Client zu erhalten.

HTTP Der Server muss die Fähigkeit besitzen mit Clients über HTTP zu kommunizieren.

XSLT Stylesheets Es muss möglich sein, XSLT Stylesheets im XLinkbase Server zu verarbeiten.

Speicherkomponenten Austauschbare Speicherkomponenten ermöglichen die Daten der XLinkbase persistent zu machen.

Komponentenbasiertes Modell Ein komponentebasiertes Modell soll es ermöglichen, dass Funktionalität gekapselt und wiederverwendet werden kann.

Aufgrund verschiedener Diskussionen mit Erik und den Entwicklern des Clients, Simon Künzli und Peter Zberg, habe ich die Anforderungsliste ergänzt und zum Teil konkretisiert:

Query Language Als Transportmittel für die Requests/Responses haben wir die HTTP-Kommunikation. Damit aber der Client dem Server mitteilen kann was für Daten er gerne von ihm hätte muss der XLinkbase Server eine Query Language anbieten. Der Client benutzt diese Query Language um Anfragen an die XLinkbase zu machen, aber auch um z.B. Daten einzufügen oder Zustandsinformationen auszutauschen.

Skalierbarkeit Die XLinkbase Architektur soll gut skalieren um sowohl komplexe und umfangreiche als auch kleine XLinkbases gut zu unterstützen.

Erweiterbarkeit Die Architektur und das gewählte Framework sollen es möglich machen, dass die XLinkbase ohne grossen Aufwand individuell an die Bedürfnisse der Benutzer angepasst werden kann. Zum Beispiel sollte das Hinzufügen eines neuen Stylesheets oder einer neuen Komponente problemlos möglich sein.

Verteilung Immer mehr Unternehmen kommen weg von einer zentralisierten Architektur und verteilen ihre Applikationen auf verschiedene Server, die an verschiedenen Standorten laufen. Sie erreichen damit eine viel höhere Verfügbarkeit des Systems (kein Single Point of Failure mehr). Die Verteilung ermöglicht es z.B. auch ein XLinkbase System als Ganzes wieder als eine Komponente anzusehen, das mit einem anderen XLinkbase System verbunden wird und so dessen Dienste/Komponenten brauchen kann.

Routing Mit dem Einsatz eines Komponenten-Frameworks kann man sich vorstellen, Requests und Responses abhängig von ihrem Inhalt durch

die verschiedenen Komponenten zu schicken. Dies bedeutet natürlich, dass so etwas wie ein Routing-Mechanismus vorhanden sein muss.

Produkte- und Herstellerunabhängigkeit Heutzutage haben viele Unternehmen Verträge und Lizenzvereinbarungen mit einigen wenigen Softwareherstellern, die das ganze Unternehmen mit Software ausstatten, um so Support- und Anschaffungskosten zu reduzieren. Um eine XLinkbase in eine bestehende Umgebung integrieren zu können, wäre es sicherlich von Vorteil, wenn man nicht abhängig wäre von bestimmten Produkten oder gar einem Hersteller.

2.2 Grobarchitektur

Ausgehend von den Anforderungen, die im letzten Abschnitt beschrieben wurden, soll in diesem Abschnitt bereits ein erster Ansatz einer Architektur aufgezeigt werden. Nachfolgend eine grobe Architektur, die den Anforderungen gerecht wird.

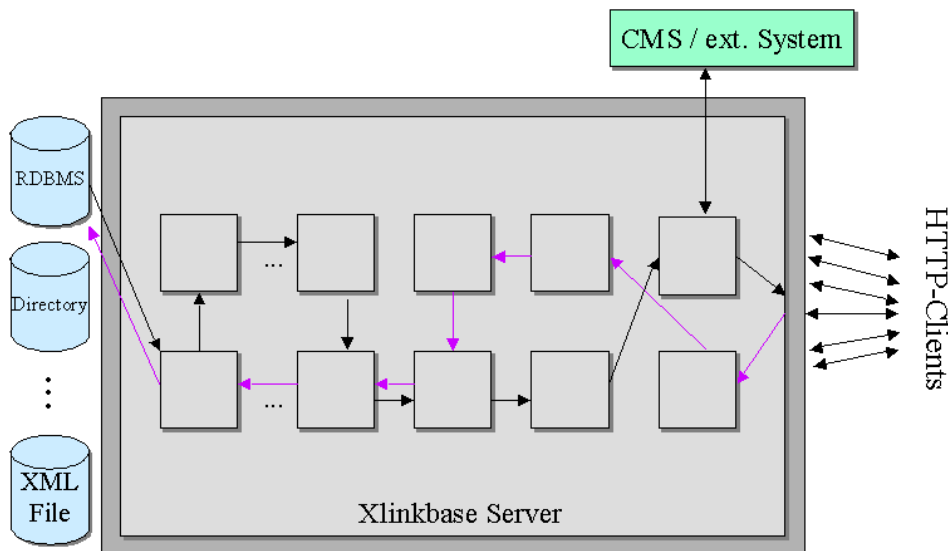


Abbildung 2.1: Eine komponentenbasierte Server-Architektur

Auf obiger Abbildung sind zwei Arten von Clients eingezeichnet. Zum einen HTTP-basierte Clients und zum anderen externe Systeme wie z.B. ein CMS oder eine Search Engine. Eine Anfrage wird typischerweise im XML-Format abgesetzt. Die erste Komponente im System muss den Request entsprechend interpretieren und ihn an die nächste Komponente weiterleiten.

Interpretieren heisst in diesem Zusammenhang den erhaltenen XML-Request zu parsen und die Wegleitung festzulegen. Um ein XML-Dokument parsen zu können muss ein XML-Parser vorhanden sein. Das heisst, es muss idealerweise eine Komponente existieren, die diese Funktionalität den anderen Komponenten anbietet. Dasselbe gilt natürlich für die Wegleitungs-Funktionalität. Da das Parsen von XML Dokumenten, abhängig von Komplexität und Umfang, zeitaufwendig ist, würde es Sinn machen, wenn zwischen den Komponenten direkt DOM-Bäume ausgetauscht werden könnten.

Um externen Systemen Zugriff auf die XLinkbase zu ermöglichen, kann eine spezielle Komponente integriert werden, die die Anfragen, die nicht zwingendermassen in der XLinkbase Query Language formuliert sein müssen, umsetzt und weiterleitet. Diese Umsetzung könnte über ein Stylesheet erfolgen. Es muss also noch eine Komponente vorhanden sein, die einen XSLT-Prozessor implementiert.

Durch diese Architektur sollte es möglich sein beliebige Datenquellen gleichzeitig zu verwenden. Das heisst, am Ende dieser Pipeline hätte man für jede Datenquelle eine eigene Komponente, die den Requests in ein Abfrageformat bringt, das die Datenquelle versteht. Dies kann beispielweise SQL-92 sein. Über die Wegleitung kann dann gesteuert werden, welche Datenquelle schliesslich die Daten liefern soll. Mit dieser Architektur ist es auch vorstellbar die Daten für einen Request aus mehr als nur einer Datenquelle zu beziehen. Im Extremfall kann so sogar eine andere XLinkbase als Datenquelle verwendet werden.

Wie sich eben vorgestellte Architektur im Detail umsetzen lässt, wird sicherlich noch vom gewählten Framework abhängen. Die in Frage kommenden Technologien werden im nächsten Kapitel näher beschrieben.

Kapitel 3

Technologien/Standards

Traditionell haben viele Systeme eine zweistufige Architektur (Client/Server). Das heisst, clientseitig befindet sich der Presentation Layer und optional ein Business Logic Layer. Serverseitig befindet sich der Data Layer und optional ein Business Logic Layer. Diese Architektur führt vielfach zu einem Thick-Client was einige offensichtliche Nachteile mit sich bringt. Auch wenn die Business Logic vollständig in den Data Layer verschoben wird, leidet unter anderem die Wiederverwendbarkeit der implementierten Logik.

Das Grobkonzept aus vorigem Kapitel zeigt bereits ziemlich klar, dass eine reine Client/Server Architektur nicht ausreichen wird. Die Gründe dafür liegen auf der Hand. Wäre es doch nötig, ein DBMS einzusetzen, welches XML-Parsing und XSLT-Transformationen ermöglicht und zudem offen genug wäre um mit anderen System (CMS), z.B. über Sockets, zu kommunizieren. Das Ziel ist aber eine möglichst offene und flexible Architektur zu haben, die nicht von einem bestimmten DBMS abhängig ist. Es macht auch keinen Sinn, einen Thick-Client zu implementieren, da das Ziel ist, einen möglichst schlanken Client zu haben, der sich einfach und schnell beim Anwender installieren lässt.

Seit einigen Jahren basieren immer mehr Applikationen auf einer N-tier bzw. Multi-tier Architektur. In einer N-tier Architektur sind Presentation-, Business- und Data Layer eigenständige Layer, die vielfach auch physikalisch getrennt sind. Man kann sich auch vorstellen einen Layer weiter aufzuteilen um eine unabhängige Skalierung zu erreichen. N-tier Architekturen haben folgende Charakteristiken:

Verteilungskosten sind tief Datenbanktreiber müssen nur serverseitig installiert und konfiguriert werden. Es ist viel günstiger Software nur ein-

mal in einer kontrollierten Serverumgebung zu konfigurieren, als diese tausenden von Usern zu verteilen.

Kosten für einen DBMS-Wechsel sind tief Clients greifen nicht direkt auf die Datenbank zu, sondern gehen immer über die Middleware. Dies erlaubt DB-Schemas zu migrieren, das Wechseln von DB-Treibern, oder sogar den Wechsel auf einen anderen Persistenzmechanismus.

Migrationskosten von Business Logic sind tief Ein Wechsel der Business Logic Schicht impliziert nicht zwingendermassen eine Neucompilation und -verteilung der Clients.

Effiziente Verwaltung und Wiederverwendbarkeit von Ressourcen

In einer N-tier Architektur können Verbindungen zu externen Ressourcen in einem Pool verwaltet werden, so dass nicht jedesmal zeitaufwendig eine neue Verbindung hergestellt werden muss (shared connections). Somit kann eine Ressource für mehrere Clients wiederverwendet werden. Dies ist in einer Client/Server Umgebung mit Thick-Clients nicht möglich.

Lokalität von Fehlern Falls ein kritischer Fehler auftritt, betrifft sie nur eine Schicht. Die anderen Schichten funktionieren weiterhin sauber und können den Fehler entsprechend behandeln. Zum Beispiel könnte der Webserver einem Anwender eine 'site down' Seite präsentieren.

Kommunikations-Performanz leidet Weil die verschiedenen Schichten meistens physikalisch getrennt sind, muss die Kommunikation über Prozessgrenzen hinweg erfolgen. Daraus resultiert ein Kommunikations-Overhead.

Seit der Idee des Multi-tier Konzeptes, sind viele sogenannte Applikationsserver auf dem Markt erschienen. Diese stellen eine Laufzeitumgebung zur Verfügung, in der Komponenten die benötigten Middleware-Services bereitstellen und bereits vorhandene nutzen können. Leider gab es bis vor kurzem keine standardisierte Definition was genau eine Middleware-Komponente ist. Aus diesem Grunde stellte jeder Applikations-Server seine Dienste in einer unheimlichen und proprietären Art zur Verfügung. Konkret heisst dies, dass der für einen bestimmten Server geschriebenen Code nicht auf einen anderen portierbar war.

Aus diesem Dilemma heraus entstand das Bedürfnis nach einer standardisierten Architektur für serverseitige Komponenten. Inzwischen hat sich

in dieser Hinsicht auch einiges getan. Die am weitesten verbreiteten Standards stammen von Microsoft, Sun Microsystems und der Object Management Group (OMG). Im nachfolgenden möchte ich ein bisschen näher auf diese drei Standards eingehen.

3.1 CORBA

Die Common Object Request Broker Architecture, oder kurz CORBA, ist eine Spezifikation, welche durch die Object Management Group (OMG) [2] definiert wurde. Die OMG ist bestrebt Interoperabilität zwischen verteilten, heterogenen Umgebungen zu erreichen. Der CORBA Standard ist ein Konsens von über 800 Firmen (z.B. Netscape, Oracle, IBM, Sun, HP, Compaq). Er definiert einen Object Request Broker (ORB), der sowohl transparentes Verwenden von Remote-Objekten als auch Objekt Dienste auf System-Level und höherliegenden liegenden Schichten (Common Facilities) anbietet. Durch CORBA wird es möglich, dass Objekte, die in verschiedenen Sprachen implementiert wurden und auf verschiedenen Plattformen laufen, interoperieren. Inzwischen gibt es viele Implementationen der CORBA Spezifikation.

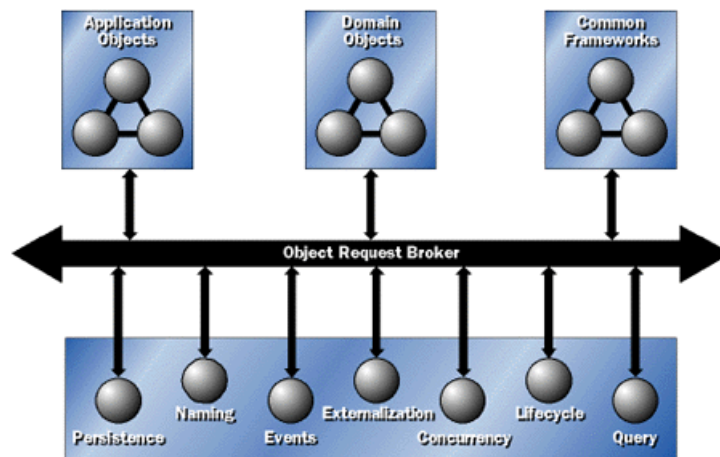


Abbildung 3.1: Die Common Object Request Broker Architecture

3.1.1 Common Object Services (CORBAservices)

Die Object Services unterstützen CORBA-basierende Applikationen auf eine anwendungsunabhängige Art. Sie konzentrieren sich auf fundamentale

Elemente verteilter Anwendungen, wie z.B. Transaktionen oder Persistenz.

CORBA services spezifiziert zur Zeit mehr als 16 Dienste, die nachfolgend kurz vorgestellt werden.

Naming service Objekte haben intern immer eine eindeutige OID. Dieser Service bindet einen Namen an ein Objekt. Namen sind bezüglich eines Namenskontext eindeutig und die verschiedenen Namenskontexte bilden eine Hierarchie ähnlich der Verzeichnisstruktur eines Dateisystems.

Object security service Viele heutige ORB's unterstützen diesen Service noch nicht. Es muss also auf platform-spezifische Dienste, die durch das darunterliegende Betriebssystem zur Verfügung gestellt werden, zurückgegriffen werden.

Object trader service Dieser Service kann als die 'Yellow Pages' eines CORBA Systems betrachtet werden. Dienstanbieter haben die Möglichkeit ihren Service registrieren zu lassen. Das heisst, er beschreibt seine Dienst mit Hilfe von Keywords nach denen Clients den Trader abfragen kann. Ein neuer Client kann so herausfinden, welche Dienste vom ORB bereitgestellt werden.

Object transaction service (OTS) Gemäss der Spezifikation muss eine OTS Implementierung sowohl flache als auch verschachtelte Transaktionen unterstützen. Es ist zudem möglich non-CORBA Transaktionen, die dem X/Open Standard gehorchen, zu integrieren.

Change management service Die Idee hier ist Versionskontrolle von Objekten, Diensten und Schnittstellen zu unterstützen. So soll zum Beispiel verhindert werden, dass ein Client, der verschiedene Interfaces benutzt, auf eine veraltete Schnittstelle zugreift.

Concurrency service Dieser Service unterstützt Locking auf Ressourcen. Locks können entweder in einem transaktionellen oder nicht-transaktionellen Kontext initiiert werden.

Event notification service Dieser Dienst entspricht einem Messaging service, wobei sowohl das Push- als auch das Pull-Modell unterstützt werden.

Externalization service Externalisierung heisst, Objekte in einen Stream zu serialisieren und entsprechend wieder zu deserialisieren. Bei der Serialisierung bleiben Referenzen zwischen Objekten erhalten. Deshalb wird dieser Dienst auch dazu verwendet Objekte by-value zu kopieren.

Licensing service Sobald Komponenten eingekauft und zu einer Gesamtapplikation verbunden werden, stellt sich die Frage der Lizenzen. Der Licensing Dienst ermöglicht, dass das System jederzeit informiert ist, wieviele Lizenzen Gleichzeitig im Einsatz sind und kann bei einer Überschreitung der Lizenzvereinbarungen entsprechende Massnahmen ergreifen.

Lifecycle service Dieser Dienst kann dazu benutzt werden um die Instanziierung, das Kopieren, Verschieben und Löschen von Komponenten zu unterstützen. Er verwaltet eine Menge von Factory-Objekten und kann diese bei Bedarf Clients zur Erzeugung von neuen Instanzen zur Verfügung stellen.

Object collections service Der Collection Service gehört zur Basisspezifikation von CORBA und bietet abstrakte Datentypen wie Mengen, Schlangen, Listen oder Bäume an. Es ist aber fraglich, ob sich dieser Dienst etablieren wird, ist doch eine native Collection-Bibliothek um einiges leichter und sicher auch schneller.

Object query service Ähnlich wie der Trader Service bietet dieser Dienst einen deklarativen Zugriff über OQL (Object Query Language) und SQL-92 an. Im Unterschied zum Trader Dienst (Lokalisierung von Servern) werden direkt Objektinstanzen lokalisiert.

Persistent object service (POS) Die fundamentale Idee hinter POS ist, eine Abstraktionsschicht anzubieten, die persistente Objekte vom eigentlichen Persistenzmechanismus kapselt. Objekte sollen mit den gleichen Operationen zum Beispiel in Dateien, relationale oder Objektdatenbanken gespeichert werden können. Es werden nur zwei Operationen unterstützt: Speichern und Laden von Objekten.

Properties service Dieser Dienst erlaubt es Objekten, die mindestens das PropertySet Interface implementieren, beliebige Felder (properties) 'anzuhängen', die jedoch vom Dienst in keiner Weise interpretiert werden. Sie dienen zum Beispiel für Administrations-Tools, um Objekten Zustandsinformationen anzuhängen um sie so effizient überwachen zu können.

Relationship service Erlaubt die Definition von Beziehungen zwischen Objekten.

Time service Um ein verteiltes System mit asynchronen Uhren dauerhaft konsistent halten zu können, bietet der Time service Zeitsynchronisation an.

3.1.2 Common Facilities (CORBAfacilities)

Die CORBAfacilities stellen die oberste Abstraktionsebene einer CORBA Implementation dar. Es wird zwischen Horizontal und Vertical facilities unterschieden. Die Vertical facilities stellen anwendungsabhängige Business-Objekte dar, wobei die Horizontal facilities anwendungsunabhängige Dienste anbieten. Nachfolgend einige Beispiele von Facilities:

Horizontal common facilities

- User Interface
- Information Management
- Task Management

Vertical market facilities

- CORBAfinance
- CORBAmanufacturing
- CORBAecommerce
- CORBAtelecom
- CORBAMED
- CORBAtransport

3.2 J2EE

Die Java 2 Platform, Enterprise Edition Spezifikation besteht aus vier Teilen: Der Platform Spezifikation, die die Anforderungen definiert, der Kompatibilitäts-Test-Suite, die die J2EE Platform Kompatibilität validiert, der J2EE Referenz-Implementation und des Application Programming Model, das beschreibt wie J2EE Applikationen aufgebaut sein müssen.

In der Platform Spezifikation werden verschiedene Dienste beschrieben, die eine J2EE konforme Implementierung bieten muss. Auf diese Services möchte ich ein bisschen näher eingehen:



Abbildung 3.2: Die J2EE Architektur

Servlets Ein Servlet ist eine Java-Klasse, die direkt über das HTTP-Protokoll angesprochen werden kann. Eine Servlet Engine dient dabei als Host-Objekt für Servlets. Eine solche Engine ist im Grunde nichts anderes als eine Java Virtual Machine (JVM), die auf einem bestimmten Port (meist 80) auf HTTP-Requests wartet und diese dem entsprechenden Servlet weiterleitet. Viele Webserver, zum Beispiel der Apache, bieten als Plug-in Modul eine Servlet Engine an. Wird ein Applikationsserver eingesetzt, bietet dieser meistens diese Funktionalität an. Bei der Konzeptionierung einer Applikationsarchitektur ist diesbezüglich zu beachten, dass wenn ein Applikationsserver und eine separate Servlet Engine eingesetzt werden (EJB- und Servlet-Komponenten), über Prozessgrenzen hinweg kommuniziert werden muss. Dies bedeutet ein entsprechender Overhead bei der Kommunikation zwischen den beiden JVMs.

Java Server Pages (JSP) JSP erlauben, Java Code direkt in eine normale HTML-Seite zu integrieren. Dieser Code wird bei einem Request ausgeführt und das Resultat schliesslich in die HTML-Seite anstelle des Java-Codes gesetzt.

JavaMail Dies ist ein API, das den Entwickler bezüglich Mailversand und -empfang unterstützt.

JDBC Dieses API wird dazu benutzt um auf eine einheitliche Art und Weise auf relationale Datenbanksysteme zuzugreifen. Für praktisch alle gängigen RDBMS existieren entsprechende Treiber.

JNDI Dieser Dienst wird im Zusammenhang mit der Verteilung wichtig, wo es darum geht Ressourcen lokalisieren zu können.

RMI RMI ist sozusagen die objektorientierte Version von Remote Procedure Call (RPC). Diese Möglichkeit ist vor allem im Zusammenhang mit einer verteilten Anwendung wichtig.

Enterprise Java Beans (EJB) Die EJBs sind der Eckpfeiler der J2EE Spezifikation. Erst durch EJB Komponenten wird serverseitige Business Logic im grösseren Stile möglich. Mehr Informationen dazu sind im Kapitel 3.2.1 auf Seite 16 zu finden.

Java Messaging Service (JMS) Dieser Dienst stellt sowohl synchrones als auch asynchrones Messaging zur Verfügung. Der JMS wird leider nicht von allen Applikationsserver implementiert, da die J2EE Spezifikation dies nicht zwingend vorschreibt. Dies wird sich aber mit der nächsten Version der Spezifikation ändern.

Java Transaction Service (JTS) Um Ressourcen-Zugriffe von verschiedenen EJBs zu synchronisieren, wird dieser Transaction Service benötigt.

Connectors Durch Connectors wird es möglich sich auf einfache Weise in ein Enterprise Information System einzuklinken. Es ist zu erwarten, dass in nächster Zeit Connectors für fast alle gängigen ERPs (z.B. SAP R/3) verfügbar sein werden.

Da die Enterprise Java Beans ein integraler Bestandteil von J2EE sind, ist eine nähere Betrachtung gerechtfertigt. Auch erfreuen sie sich immer grösserer Popularität.

3.2.1 EJB

Die Enterprise Java Bean Spezifikation ist wie bereits angetönt ein Eckpfeiler von J2EE. Die Publikation der EJB Spezifikation Version 1.0 war das bedeutendste Ereignis seit Einführung von JDK 1.1. Durch das JDK 1.1 wurden erstmals Multi-tier Applikationen mit Java möglich. Mit JDBC wurde der Datenzugriff auf ein DBMS möglich, RMI als objektorientierte Variante von RPC machte Verteilung möglich und die Java Beans waren die Basis für ein Java-basiertes Komponentenmodell. Dies waren wichtige Erweiterungen in Richtung Enterprise Computing. Dennoch fehlte die Unterstützung um robuste und skalierbare Serverkomponenten zu entwickeln. Dieses Dilemma wurde durch die EJBs behoben.

EJB wurde auf Robustheit und Skalierbarkeit ausgelegt und stellt weitere, nötige Funktionalität für serverseitige Komponenten wie Naming Service, verteilte Transaktionen, transparente Persistenz, Messaging und Sicherheitsmechanismen zur Verfügung, die weiter oben bereits kurz skizziert wurden. EJB ist aber mehr als nur eine Ansammlung genannter Dienste. Es ist ein komplettes Framework zur Erstellung von Serverkomponenten, das dem Entwickler viele Aufgaben abnimmt, so dass sich dieser hauptsächlich auf die Implementierung der Business Logic konzentrieren kann.

3.2.1.1 Session Beans

Ein Session Bean ist ein nicht-persistentes Objekt, das serverseitig Business Logik implementiert. Session Beans sind sozusagen private Ressourcen, die nur vom Client benutzt werden können, der sie erzeugt hat. Somit wird ein Session Bean nicht mit anderen Clients geteilt und erscheint anonym. Ein Session Objekt muss nicht für jede Benutzung neu erzeugt werden. Einmal von einem Client erzeugt, kann das Objekt im Container bleiben bis es entweder vom Client entfernt wird oder vom Container aus speichertechnischen Gründen in einen sleep-Zustand gebracht wird.

3.2.1.2 Entity Beans

Ein Entity Bean ist ein persistentes Objekt, das eine Objektsicht einer Datenbank-Entität repräsentiert. Im Gegensatz zu den Session Beans können mehrere Clients ein Entity Bean gleichzeitig benutzen. Der Container synchronisiert die Zugriffe auf ein Entity Bean mit Hilfe von Transaktionen. Jedes Bean hat eine Identität, seinen Primärschlüssel, die einen Container-Crash übersteht.

3.2.1.3 Message Driven Beans (MDB)

Die EJB 2.0 Spezifikation führt eine neue Art von Beans ein, die Message Driven Beans. Sie agieren als Message Konsumenten und Produzenten des Java Messaging Services (JMS). MDBs erhalten ihre Nachrichten über Queues oder Topics, bei denen sie sich registrieren und führen, je nach Nachrichtinhalt, Business Logic aus. Sie werden vollständig vom Applikationsserver verwaltet und können somit nicht explizit durch einen Client instanziiert werden. Der Server erzeugt automatisch die nötige Anzahl von Instanzen um die gegebene Nachrichtenlast bewältigen zu können. Clients könne also einzig über Nachrichten mit MDBs kommunizieren.

3.3 Microsoft's .NET Platform

Microsoft hat im Jahre 1999 einige ihrer Produkte genommen und sie zu Windows Distributed interNet Applications Architecture (DNA) geschnürt. Im Gegensatz zu CORBA und der J2EE Plattform ist Windows DNA eine Menge von Produkten und keine Spezifikation. Mit der kürzlichen Vorstellung der .NET Vision hat Microsoft die Gelegenheit genutzt und Windows DNA zu Web Solution Platform umgetauft. Folgende Abbildung soll die neue Architektur und die Unterschiede zum ursprünglichen Windows DNA zeigen:

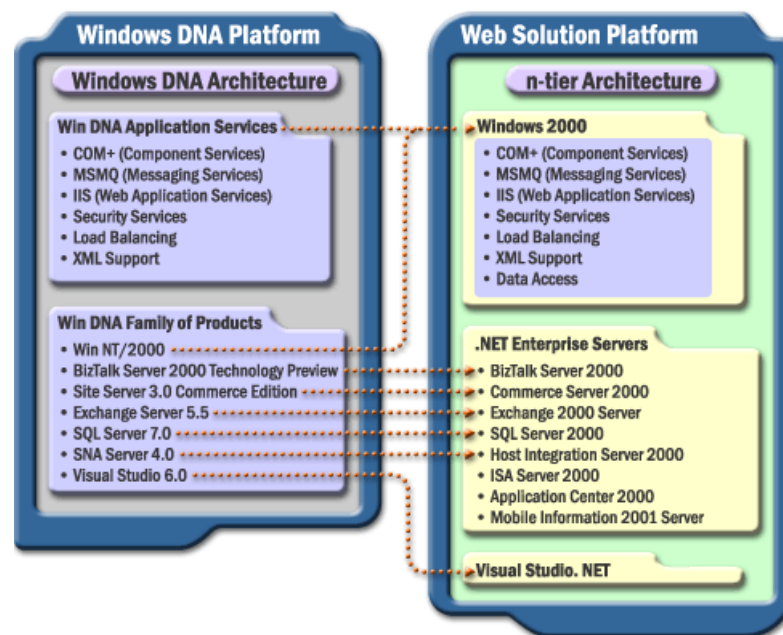


Abbildung 3.3: Evolution von Windows DNA nach .NET

Grosse Teile der Plattform sind als COM-Komponenten implementiert. Lediglich der Kern von Windows basiert nicht auf COM. Es erstaunt dann auch nicht, dass sich praktisch alle angebotenen Services über COM ansprechen lassen, da sie selbst als COM-Komponente implementiert wurden. Ich gebe nachfolgend eine kurze Beschreibung der Komponenten und Services, die die Web Solution Plattform ausmachen:

Windows 2000 Das darunterliegende Betriebssystem, das eine Laufzeitumgebung für alle Microsoft Technologien bereitstellt.

Internet Information Services 5.0 Darunter wird der eigentliche Webserver mit allen unterstützten Technologien wie Active Server Pages (ASP),

Common Gateway Interface (CGI), Internet Server API (ISAPI), COM Komponenten verstanden.

Message Queueing (MSMQ) Die Messaging Services sind als eine Menge von COM-Komponenten implementiert. Somit lässt sich Messaging Funktionalität überall dort verwenden, wo COM-Komponenten eingesetzt werden können. Es wird sowohl synchrones als auch asynchrones Messaging unterstützt.

COM+ COM+ ist eine Erweiterung des Komponentenstandards COM. Grob gesagt wurde COM einfach um Transaction Services (MTS) erweitert.

Security Services Um sensitive Daten einer Applikation zu schützen unterstützt Microsoft 56-bit und 128-bit SSL/TLS, IPsec, Server Gated Cryptography, Digest Authentication, Kerberos 5 und Fortezza. Diese Dienste stellen einen Kernpunkt einer solchen Architektur dar. Werden sie doch mit der steigenden Zahl von Intranets immer wichtiger um firmeneigene Daten gegen aussen zu schützen.

Load Balancing In der Advanced und Datacenter Version von Windows 2000 Server ist es möglich, mehrere Rechner zu einem Cluster zusammenzuschliessen um die Last aufteilen zu können. Dieser Services wird vielfach in Umgebungen eingesetzt wo eine hohe Verfügbarkeit und Belastbarkeit erwartet wird.

XML Support Windows 2000 unterstützt XML nach dem W3C 1.0 Standard auf Betriebssystem-Ebene. Das heisst, der MSXML Parser ist als COM Komponente aus verschiedenen Anwendungen, insbesondere aus dem Internet Explorer heraus, ansprechbar. Windows soll das erste Betriebssystem sein, das eine XML Unterstützung auf dieser Ebene anbietet.

Data Access Um den Zugriff auf heterogene Datenquellen auf einfache und einheitliche Weise zu unterstützen, werden COM-Komponenten in Form von Data Access Components (DAO) für verschiedene Schnittstellen (OLE DB, ODBC, ADO, ...) angeboten.

Für weitergehende Informationen stellt Microsoft auf ihrer Website viel zusätzliches Informationsmaterial und Links bezüglich der neuen .NET Plattform zur Verfügung [5].

Kapitel 4

Evaluation

Bei der Evaluation galt es herauszufinden, welche der drei vorgestellten Technologien den Anforderungen einer Linkdatenbank am besten genügt. Stützt man sich auf die Bedürfnisse, lässt sich folgender Vergleich der Technologien bezüglich der Kriterien anstellen.

4.1 XML-Unterstützung

Die .NET Plattform hat bereits Unterstützung für die Verarbeitung von XML eingebaut. Diese Funktionalität liegt als COM-Komponente vor, die Microsoft MSXML nennt. Diese Komponente hat einen XML-Parser und einen XSLT-Stylesheet Prozessor eingebaut. Wobei hier zu sagen ist, dass der MSXML in der zur Zeit aktuellsten Version 3.0 nur die DOM Level 1 Spezifikation unterstützt. Will man nicht Microsoft's Parser und XSLT-Prozessor verwenden muss eine neue Komponente entwickelt werden.

Die J2EE und CORBA Architektur haben keine vorgefertigte XML-Komponente anzubieten. Eine solche Komponente muss also neu entwickelt werden. Schaut man sich aber ein bisschen um, existieren auf der Basis von Java schon fast unüberblickbar viele Parser und XSLT-Prozessoren. Als Beispiel sei hier nur das Apache XML Projekt genannt, das mit Xerces und Xalan einen sehr verbreiteten DOM Level 2 Parser und Prozessor anbietet [6]. Nimmt man C++ als Referenzsprache für CORBA, existieren z.B. auch C++ Versionen des eben genannten Parsers und Prozessors, aber viele weitere Vertreter sucht man vergebens. Auch existieren für eine J2EE Umgebung ein vollständiges Web Publishing Framework namens Cocoon, das ebenfalls aus dem Apache Projekt heraus entstanden ist. Im Zusammenhang mit Java ist es sicher-

lich erwähnenswert, dass seit kurzem eine Preview Version eines sogenannten XSLT Compilers, der als Eingabe ein Stylesheet nach 1.0 Spezifikation nimmt und eine Java Klasse, Translet genannt, als Resultat zurückgibt [9]. Diese Klasse übernimmt dann die Funktionalität der XSLT Transformation von XML Dokumenten mit dem ursprünglichen Stylesheet.

Bezüglich CORBA ist noch zu erwähnen, dass selbstverständlich alle Java Parser und Prozessoren auch in eine CORBA Komponente integriert werden können, entwickelt man diese mit Java. Der grosse Vorteil von CORBA liegt aber meiner Meinung nach in der Performanz und da würde es keinen Sinn machen eine Java Komponente im System zu haben, die das ganze System 'ausbremst'.

4.2 Verteilung

CORBA und J2EE wurden im Gegensatz zu .NET von Anfang so konzipiert, dass Verteilung durchgängig unterstützt wird. Die Idee von CORBA basiert ja sogar auf dem Konzept der Verteilung von Objekten und Diensten und ist somit ein integraler Bestandteil eines CORBA Systems. Microsoft hat die Möglichkeit der Verteilung von Objekten erst nachträglich in ihr Framework integriert. Das heisst zuerst war COM da und erst einige Zeit später wurde die sozusagen verteilte Version von COM, Distributed COM (DCOM), eingeführt.

4.3 Plattformunabhängigkeit

Hier liegen ganz klar die Stärken von J2EE. Eine Java Virtual Machine ist heute beinahe für jede erdenkliche Plattform verfügbar. Zudem sind praktisch alle Application Server in Java selbst geschrieben und sind somit auch Unabhängig von der Plattform. Ähnlich präsentiert sich das Bild bei CORBA. Inzwischen gibt es ORB Implementierungen für die gängigsten Plattformen. Ganz anders sieht es bei .NET aus. Microsoft verspricht zwar C# Compiler für verschiedene Plattformen anbieten zu wollen. Aber ob eine .NET Applikation mit unzähligen COM-Komponenten jemals auf einem UNIX Rechner zum Laufen kommen wird, ist für mich zum heutigen Zeitpunkt fragwürdig.

4.4 Produkteunabhängigkeit

Es gibt zur Zeit nur sehr wenige Application Server, ausgenommen MS-eigene, die COM-Komponenten hosten können. Ein Beispiel hierfür wäre der Enterprise Application Server von Sybase [8]. Schaut man sich die EJB-Komponenten an, lassen sich diese in jeden J2EE zertifizierten Applikationsserver ohne grossen Aufwand integrieren. Einzig der produkteabhängige Deployment Descriptor muss angepasst werden. Ziemlich ähnlich sieht es mit CORBA Komponenten aus.

4.5 Performanz

Da compilierte Java Applikationen als Byte-Code vorliegen und auf einer JVM laufen müssen, wird durch die zusätzliche Zwischenschicht Performanz eingebüsst. CORBA und .NET Applikationen liegen im Gegensatz dazu in Maschinensprache vor und werden ohne Umweg über eine Virtual Machine ausgeführt. Serverseitig fällt dies nicht sonderlich ins Gewicht, da bei Multi-tier Applikationen erfahrungsgemäss bis zu 80% der Rechenzeit mit Datenbankzugriffen zugebracht wird.

4.6 Verbreitung

Am populärsten sind sicher .NET und J2EE Applikationen. Es gibt inzwischen unzählige Webseiten, die von ASP oder JSP und Servlets Gebrauch machen. CORBA bietet keine ähnliche Funktionalität und ist zudem komplexer.

4.7 Komplexität

Ein Ziel der J2EE Spezifikation war es, den Entwickler nicht mit unnötigen Sachen zu belasten, sondern ihm ein Framework zur Verfügung zu stellen, das die Konzentration auf die Business Logic ermöglicht. Ich denke, dass dies SUN ganz gut gelungen ist, wenn man z.B. schaut, wie einfach es im Grunde ist, ein Servlet zu implementieren. CORBA hingegen wird als ziemlich komplexe und umfangreiche Umgebung, die viel Erfahrung und Know-How erfordert, eingestuft. Microsoft's .NET Plattform wird irgendwo zwischen

J2EE und CORBA angesiedelt.

4.8 Zusammenfassung

Ausgehend von obigen Ausführungen, habe ich Tabelle 4.1 erstellt, die eine Übersicht wiedergibt.

Während der Evaluation war für mich eigentlich relativ rasch klar, dass die .NET Plattform keine ernst zu nehmende Alternative zu J2EE und CORBA war. Im Kontext dieser Arbeit zumindest. Ausschlaggebend dafür war sicherlich die starke Plattform- und Produkteabhängigkeit. Ich sehe das Einsatzgebiet der .NET Plattform deshalb viel eher dort, wo es um spezifische Lösungen in Unternehmen geht und nicht als Basis für ein Produkt, das in verschiedensten Umgebungen eingesetzt werden soll. Somit blieben nur noch CORBA und J2EE zur Auswahl.

	Microsoft .NET	J2EE	CORBA
XML-Unterstützung	++	++	0
Verteilung	+	++	++
Plattformunabhängigkeit	-	++	+
Produkteunabhängigkeit	-	+	++
Performanz	+	0	+
Verbreitung	++	++	+
Komplexität	0	+	-

Tabelle 4.1: Vergleich der Technologien

CORBA schneidet im Vergleich zu J2EE nur in den Rubriken Performanz und Produkteunabhängigkeit besser ab. CORBA wird vielfach in extrem umfangreichen Umgebungen (die UBS AG z.B. hat mit 40'000 PC's und 400 UNIX Servern wahrscheinlich eines der grössten CORBA Systeme weltweit) eingesetzt. In einer solchen Umgebung hat hohe Performanz sicher einen höheren Stellenwert als dies bei einer XLinkbase der Fall ist. Die Pluspunkte in der Produkteunabhängigkeit rühren daher, dass es beinahe mit jeder Programmiersprache möglich ist eine CORBA Komponente zu schreiben. In einer J2EE Umgebung ist dies grundsätzlich nur mit Java möglich, obwohl es über einen ORB, der RMI-IIOP unterstützt möglich ist, EJB Komponenten in CORBA Clients und umgekehrt aufzurufen. Somit wird auch dieses Argument relativiert.

Bei J2EE gefiel mir vor allem die relativ einfache Implementierung von EJB Komponenten und der extrem vielfältige XML-Support. Zudem bietet J2EE im Vergleich zu CORBA eine vollständige HTTP Kommunikationsschnittstelle in Form von Servlets.

So folgte schliesslich, dass ich mich für J2EE als Basis für das Konzept des XLinkbase Servers entschieden habe.

Kapitel 5

Detailkonzept

Nach der Wahl von J2EE als Framework, war ich mir lange Zeit uneinig, ob ich als Basis das Web Publishing Framework Cocoon [6] [7] nehmen oder das Ganze neu designen soll. Ich habe dann die Architektur von Cocoon ein bisschen näher angeschaut, ob damit eine Architektur gemäss dem Grobkonzept überhaupt möglich wäre.

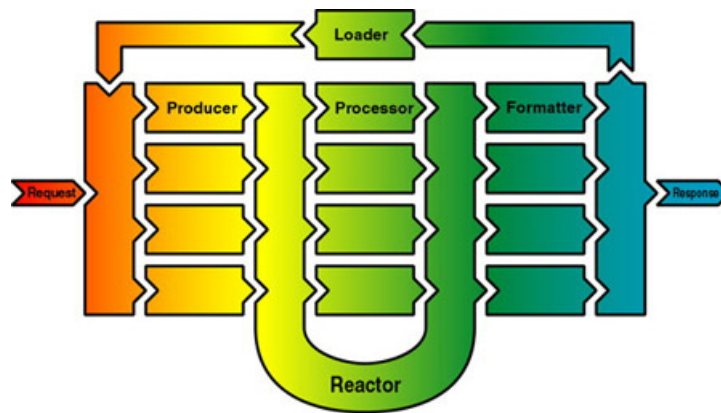


Abbildung 5.1: Die Cocoon Architektur

Die Architektur basiert auf dem Reactor Design Pattern [10] und besteht aus den Komponenten Producer, Reactor, Formatter und dem Loader. Der Loader ermöglicht es die Pipeline mehrmals zu durchlaufen. Abhängig von Cocoon-eigenen XML Processing Instructions wird der Pfad durch die Komponenten gewählt. Auf den ersten Blick scheint es, als würde Cocoon unseren Anforderungen (Komponenten, Routing, etc.) gerecht. Erst bei längerem Hinschauen wird klar, dass einige Änderungen nötig wären:

Verteilung Da Cocoon im Grunde als ein grosses Servlet konzipiert wurde, würde es einiger Arbeit bedürfen die einzelnen Teile in völlig eigenständige Komponenten zu kapseln.

Routing Wegleitung ist zwar vorhanden, aber kann nur über die Processing Instructions in einem XML Dokument gesteuert werden. Dies ist für unsere Anwendung unzureichend. Zudem müsste auch bedingtes Routing unterstützt werden.

Diese Gründe haben es gerechtfertigt auf eine eigene, auf EJBs basierende Komponenten-Architektur zu setzen.

5.1 Architektur

Die Detailarchitektur, die ich entworfen habe, basiert bereits auf der EJB 2.0 Spezifikation, die asynchrones Messaging für EJBs zulässt. Die Version 1.0 der Spezifikation liess nur synchrones Messaging für Beans zu. Mit Version 2.0 wurde auch eine neue Art EJBs eingeführt, die Message Driven Beans (siehe auch Kapitel 3.2.1.3 auf Seite 17).

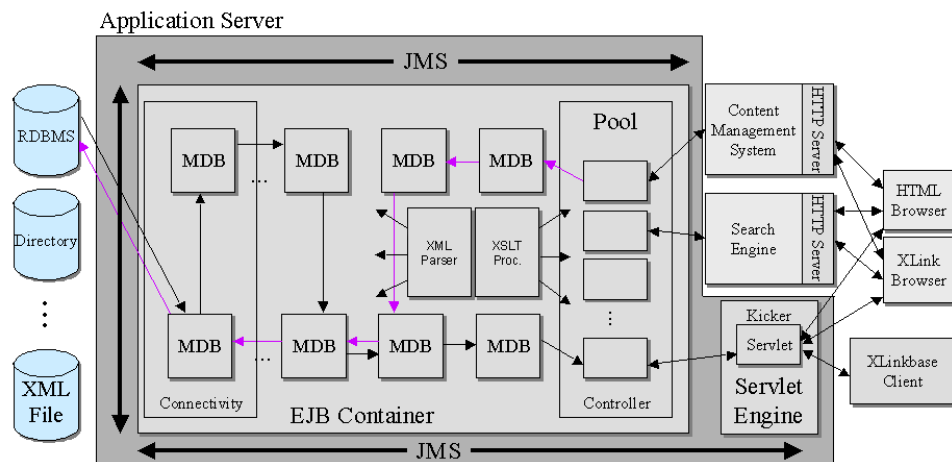


Abbildung 5.2: Die XLinkbase Architektur

5.1.1 Clients

Wir haben grundsätzlich zwei Arten von Clients. Zum einen sind dies die XLinkbase Clients und zum anderen Systeme wie Suchmaschinen oder Con-

tent Management Systeme. Ein XLinkbase Client kommuniziert immer über HTTP mit dem XLinkbase Server. Theoretisch wäre dies auch für externe Systeme möglich. Um aber eine bessere Integration in das System zu erreichen würde man idealerweise in das Fremdsystem eine Java Komponente (eine normale Java Klasse, ein (Enterprise) Java Bean oder ein Servlet) integrieren, die über den JNDI Dienst eine Verbindung zum EJB Server machen kann. Manch einer fragt sich vielleicht wieso dieser direktere Weg nicht auch für den XLinkbase Client gewählt wurde. Diese Frage ist durchaus berechtigt, kann aber relativ einfach beantwortet werden. Der XLinkbase Client basiert im Moment auf einer ActiveX Komponente, die in den Internet Explorer integriert wurde. Da es über eine solche Komponente nicht möglich ist, einen Kontext auf den JNDI Dienst des EJB Servers zu erhalten, wurde eine Servlet Engine, die die HTTP Anfragen umsetzt und weiterleitet, dazwischengeschaltet. Würde es einmal ein XLinkbase Client in Form einer Java Applikation oder Applet geben, könnte man, wie gesagt, den direkten Weg gehen.

5.1.2 Application Server und Container

Sehen wir uns die Bestandteile des XLinkbase Servers ein bisschen genauer an, haben wir als eigentliche Laufzeitumgebung einen Application Server der J2EE compliant ist. Um die EJBs hosten zu können, muss ein sogenannter Container vorhanden sein. Die J2EE Spezifikation zieht leider keine klare Linie zwischen EJB Server und Container, so dass ich hier einfach den Begriff Container verwende. Die Aufgaben eines EJB Containers sind sehr vielfältig. Containers sind verantwortlich für das Managing der Beans. So interagieren sie bei Bedarf mit den Beans, indem sie Management Methoden der Beans aufrufen. Dabei handelt es sich um Callback Methoden die nur der Container aufruft. Diese Methoden erlauben es einem Container beispielsweise, gewisse Beans zu notifizieren, dass sie in einen persistenten Zustand übergehen sollen. Die wichtigste Eigenschaft eines Containers bleibt aber, den Beans eine Laufzeitumgebung zur Verfügung zu stellen. Erst durch EJB Container werden Beans für Clients erreichbar. Man kann auch sagen, dass sie eine Art transparente Mittelfunktion zwischen Beans und Clients übernehmen. Beans werden nie direkt instanziiert oder aufgerufen. Obwohl dies für den Client transparent ist, laufen alle Interaktionen hinter der Bühne über den Container. Sie übernehmen deshalb auch weitere Funktionen wie z.B. Transaktionskoordination, Bereitstellung von Persistenzmechanismen (Container Managed Persistence für Entity Beans) oder Verwaltung der Lebenszyklen der Beans. Für weitergehende Informationen ist das Buch über EJBs von Ed

Roman eine gute Quelle [11].

5.1.3 EJBs

Die EJBs stellen den Kern des ganzen Systems dar, da sie die Business Logic des XLinkbase Servers implementieren. Es werden vorwiegend MDBs eingesetzt.

Die eigentliche Verarbeitung eines Requests beginnt in einem Controller Bean. Wie man in Abbildung 5.2 sehen kann, kann es einen ganzen Pool solcher Controller Beans geben. Sicher gibt es mal für jeden unterschiedlichen Client (XLinkbase, CMS, Search Engine) mindestens eine solche Komponente. Der Pool dient dazu, dass der EJB Server automatisch neue Instanzen generieren kann, wenn die Last auf die vorhandenen Komponenten zu gross wird. Im Deployment Descriptor einer EJB kann zusätzlich angegeben werden wieviele Instanzen der Server beim Starten erzeugen soll. Das Pooling ist also nichts anderes als eine Art Load Balancing auf Komponentenbasis.

Die Aufgabe eines Controller Beans ist nun, den erhaltenen Request zu parsen und ihn mit zusätzlicher Wegleitungsinformation weiterzuschicken. Es macht Sinn, dass eine zentrale Routingkomponente in Form einer Session Bean vorhanden ist, die einen Request von einem Controller Bean als Parameter erhält, und als Rückgabewert den mit Routinginformation angereicherten Request zurückgibt. Eine solche zentrale Routingkomponente könnte dann auch über eine Management Konsole konfiguriert werden.

Handelt es sich um ein Controller Bean, das ein externes System mit der XLinkbase verbindet, muss noch weitere Funktionalität integriert werden. Es könnte möglich sein, dass die Query noch in die XLinkbase Query Language transformiert werden muss. Diese Transformation kann im Idealfall über ein Stylesheet erfolgen. Wird das externe System asynchron mit der XLinkbase verbunden (über ein MDB), muss sichergestellt werden, dass die Requests und Responses eindeutig identifizierbar sind. Es könnte sein, dass ein CMS mehrere Requests absetzt bevor es überhaupt einen ersten Response zurückbekommt. In einem solchen Falle ist es notwendig in der entsprechenden Controller Komponente Requestinformationen zu speichern um sicherzustellen, dass wenn die Response zurückkommt, diese dem richtigen Request zugeordnet und zurückgegeben wird. Wie dies funktionieren könnte zeigt die Implementierung des Kicker Servlets.

Ich habe für das System auch eine zentrale XML-Parser und XSLT-Prozessor Komponente vorgesehen. Man kann sich diese als eine Art Dienste

vorstellen. Muss an irgendeiner Stelle im Server XML geparkt oder ein Stylesheet angewendet werden, kann man diese Aufgabe diesen Komponenten übertragen. Hier muss aber betont werden, dass es keinen Sinn macht, wenn die Parser Komponente eine komplette API zur Manipulation von DOM-Trees bereitstellt. Eine Manipulation eines Baumes über eine andere Komponente würde einen viel zu grossen Kommunikationsoverhead produzieren, da bei jedem Methodenaufruf der Baum serialisiert und auf der anderen Seite wieder deserialisiert werden muss. Es macht mehr Sinn die entsprechende Bibliothek direkt in das Bean einzubinden und die Manipulation über diese vorzunehmen. Der Preis dafür ist aber eine stärkere Abhängigkeit vom gewählten Parser. Vergleichsweise lässt sich der eingesetzte Parser in der Parser Komponente nach Belieben austauschen, ohne dass die anderen Komponente etwas davon mitbekommen oder gar neu übersetzt werden müssten. Diese Abhängigkeit lässt sich aber bis zu einem gewissen Grade durch den Einsatz eines API, das vom Parser API abstrahiert, auflösen. Auf dieses API [12] wird im Implementierungsteil (siehe Kapitel 6 auf Seite 35) näher eingegangen.

Beim Back-End des Servers findet man auch eine Art Pool. Im Grunde soll nur gezeigt werden, dass man am Ende der Kette eine Gruppe von Komponenten hat, die für die Konnektivität zu Datenquellen verantwortlich ist. Diese Gruppe könnte beispielsweise aus einer Komponente bestehen, die über JDBC an eine Oracle Datenbank angeschlossen ist, eine die über LDAP auf eine Directory zugreifen kann und eine weitere die mit einer XML Datenbank umgehen kann.

5.1.4 Servlets

Um mit dem XLinkbase Client kommunizieren zu können, muss serverseitig ein Servlet implementiert sein. Für die anderen Clients ist dies, wie bereits gesagt, nicht zwingend notwendig. Dieses Kicker Servlet muss in einer Servlet Engine laufen. Dies ist der Grund wieso nebst dem EJB Server noch eine Servlet Engine benötigt wird.

Die Aufgabe des Servlets besteht darin, die Requests über einen bestimmten Port, aus Firewall-technischen Gründen vorzugsweise 80, zu empfangen und an ein Controller Bean weiterzuleiten. Wie oben bereits angesprochen, ist aber das Weiterleiten nicht die einzige Aufgabe. Da das Servlet asynchron mit den anderen Komponenten kommuniziert, ist es dafür verantwortlich, den Requests eine eindeutige ID zuzuordnen, diese abzulegen, damit zurückkommende Responses wieder dem richtigen Request, und damit der richtigen

Session, zugeordnet werden können. Bezüglich Implementierungsdetails verweise ich auf Kapitel 6 auf Seite 35.

5.2 Kommunikation

Die Kommunikation spielt in einer Umgebung mit vielen Teilnehmern eine besonders wichtige Rolle. Einerseits muss sie effizient stattfinden können, andererseits muss sie flexibel genug sein um auch zukünftigen Anforderungen gewachsen zu sein. Das Ziel war ganz klar möglichst durchgängig XML, als Sprache sozusagen, zu verwenden.

5.2.1 Client/Server Kommunikation

Der XLinkbase Client kommuniziert mit dem Server über HTTP 1.1. Um Daten an den Server zu übertragen, verwenden wir die POST Methode und schicken den eigentlichen Request im Body an die XLinkbase mit. Bezüglich Protokolldetails sei auf den HTTP Standard verwiesen [13]. Im Client wird also auf Interaktion eines Benutzers ein XML-Dokument zusammengestellt, das die eigentliche Query an die XLinkbase repräsentiert, und in den Body eines POST Requests abgefüllt wird. Wie eine solche XML Query aussehen darf, beschreibt eine DTD, die im Anhang A.1 auf Seite 55 zu finden ist. Dies ist natürlich eine sehr rudimentäre DTD, die nur dazu gebraucht werden kann ein Topic anzugeben, das dann vom Server zurückgeliefert wird. Diese DTD wurde für den Prototypen eingesetzt um die Basiskommunikation testen zu können. Nachfolgend ein Beispielrequest des XLinkbase Clients wie er im Servlet empfangen wird.

┌

```
POST /Kicker HTTP/1.1
Accept-Encoding: gzip, deflate
Accept: */*
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 4.0)
Host: 129.132.57.104
Content-Length: 173
Connection: Keep-Alive
Cache-Control: no-cache

<?xml version="1.0" "encoding="UTF-8"?>
```

```
<!DOCTYPE xlbrequest SYSTEM "http://localhost/xlbrequest.dtd">
<xlbrequest xlbql="1.0">
  <stmt>
    SELECT topic[t] FROM XLinkbase WHERE t="html"
  <\stmt>
</xlbrequest>
```

└

Eine ausgereifte Query Language für die XLinkbase würde eine viel komplexere DTD verlangen, die zudem in die XLinkbase DTD integriert würde, so dass am Schluss nur noch eine grosse DTD, die das XLinkbase Schema (siehe Anhang A.2 auf Seite 55), die Requests und Responses beschreiben würde, vorhanden wäre.

5.2.2 Serverinterne Kommunikation

Die serverinterne Kommunikation ist vollständig XML-basiert. Das heisst, die Komponenten sprechen miteinander in XML. Diese Spezifikation alleine reicht natürlich noch nicht aus. Um XML Daten austauschen zu können muss noch ein Kommunikationsmechanismus vorhanden sein. Wie wir bereits gesehen haben, sind alle Komponenten ausser das Servlet und eventuelle Komponenten zur Anbindung eines externen Systems als MDB implementiert. Da MDBs nur über den Messaging Service kommunizieren können, müssen zur Kommunikation Queues und Topics verwendet werden. Das Prinzip von Queues und Topics' ist relativ einfach. Im Prinzip registriert man sich bei einer Queue oder einem Topic und erhält dann im asynchronen Fall Messages, die in eine Queue eingereicht oder als Topic publiziert wurden. Im synchronen Falle muss der Interessent selbst nachfragen, ob eine neue Message vorhanden ist (Polling). Einen grossen Unterschied gibt es aber zwischen Queues und Topics'. Queues sind nur dafür gedacht eine Point-to-Point Kommunikation zu unterstützen. Das heisst, wird eine Meldung konsumiert, wird sie aus der Queue gelöscht. Ein Topic dagegen ist sozusagen für Multicast-Kommunikation gedacht. Das heisst, bevor nicht alle, die sich für ein bestimmtes Topic registriert haben, die Message konsumiert haben, wird sie nicht gelöscht.

Es stellte sich dann die Frage, ob ich nun Queues, Topics oder beides zusammen zur Kommunikation verwenden soll. Nach ein paar Überlegungen kam ich zum Schluss, dass eine Kombination von beiden Konzepten

wahrscheinlich die beste Lösung ist. Die Idee dabei ist jeder Komponente eine eigene Queue zu geben über die sie die Requests von anderen Komponenten erhält. Zusätzlich gibt es ein Topic 'Management' für das sich jede Komponente im Server registriert. Denkt man an eine Management Konsole, kann dieses Topic beispielsweise dazu verwendet werden, einen Request an alle Komponenten zu schicken um herauszufinden welche Komponenten überhaupt im System vorhanden sind. Man müsste sich natürlich über ein spezielles Format für Management-Messages einig werden, das dann auch von allen Komponenten verstanden und entsprechend beantwortet wird.

Kapitel 6

Implementierung

Nach dem Entwurf des Detailskonzepts galt es einen Prototypen des XLink-base Servers zu implementieren. Während der Konzeptionierung habe ich viel Zeit damit zugebracht überhaupt mal einen Applikationsserver zum Laufen zu bringen und so zu konfigurieren, dass er eine Testkomponente und ein Servlet aufnehmen kann. Um eine kleine Test-Komponente implementieren zu können, musste ich mich deshalb bereits in dieser Phase mit den EJB spezifischen Interfaces und Deployment Descriptors auseinandersetzen. Ein Komponentengerüst zu erstellen oder eine Komponente in einen Applikationsserver zu integrieren, Deployment genannt, war also bereits zu Beginn der Implementierung schon fast eine Routineangelegenheit.

Zu Beginn der Implementierungsphase waren bereits zwei Applikationsserver auf meinem System lauffähig installiert. Zum einen war dies der EA-Server von Sybase [8] und zum anderen der Open Source Server jBoss [14] mit Jetty [15] als Servlet Engine. Es verging jedoch einige Zeit bis ich festgestellt habe, dass die Implementierungen des JMS, besonders die des EAServers, ziemlich unausgereift und fehlerhaft waren. So verbrachte ich einmal mehrere Tage damit, einen vermeintlichen Fehler in der Implementierung meines Beans zu finden, bevor mir schliesslich von Sybase selbst bestätigt wurde, dass es sich dabei um einen Bug in ihrem Server handle. Ähnlich erging es mir, als ich versuchte den Messaging Service auf dem jBoss Server sauber zum Laufen zu bringen. Auch dieser war durchsät von Fehlern und Unsauberkeiten. Schliesslich entschied ich mich zur Installation des BEA Weblogic 6.0 Server [16]. Die Firma BEA war mir durch ihren TP-Monitor Tuxedo bekannt und ich konnte nur hoffen, dass der Weblogic Server ähnlich gut funktioniert. Glücklicherweise wurde ich nicht enttäuscht und der Weblogic Server präsentierte sich als stabilster und ausgereiftester der bisher getesteten

Application Server. So hatte ich eine gute Grundlage für die Implementierung des Servlets und der Beans.

Die Architektur des Prototypen ist ziemlich einfach, hält sich aber an das besprochene Konzept. Die Idee war nicht einen hochkomplexen Prototypen mit unzähligen Features zu implementieren, dafür hätte die Zeit auch nicht gereicht, sondern vielmehr das entwickelte Konzept auf dessen Tauglichkeit hin zu testen.

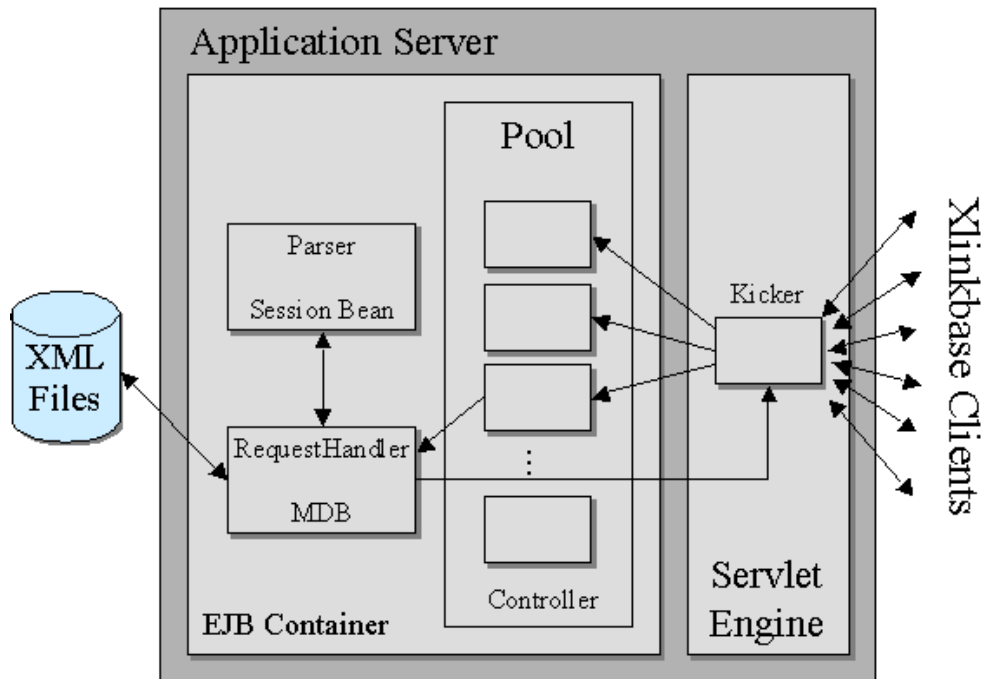


Abbildung 6.1: Die Architektur des Prototypen

6.1 Entwicklungsumgebung

Im Gegensatz zum EAServer, den es mit kompletter Entwicklungsumgebung (PowerJ) und Deployment Unterstützung gibt, kommt der Weblogic Server als reiner Application Server daher. Das heisst, make- oder Batch-Files um einfach compilieren zu können, müssen selbst geschrieben werden. Erst empfand ich dies als grossen Nachteil, bis ich es schätzen lernte, wirklich auch selbst kontrollieren zu können was hinter den Kulissen abgeht. Hat man einmal ein Batch- oder make-File für eine Komponente erstellt, lässt sich dieses mit sehr geringem Aufwand für eine neue Komponente anpassen. Der Nach-

teil des Ganzen ist, dass so kein inkrementelles Kompilieren unterstützt wird. Bei jedem Kompilationsthrough wird jede Klasse neu übersetzt, auch wenn keine Änderungen gemacht wurden. Deshalb lohnt sich der Einsatz eines entsprechenden Tools. In der Java-Welt ist Ant [17] stark verbreitet. Ich habe Ant auch selbst ausprobiert und empfinde es als ein sehr hilfreiches und gutes Tool.

Der Weblogic 6.0 Server wird mit dem kompletten JDK 1.3 von Sun ausgeliefert, also mit der zur Zeit aktuellsten Version. Somit ist es nicht mehr nötig irgendetwas von den Sun Seiten herunterzuladen. Da der Weblogic Server selbst eine Java Applikation ist, spielt die Plattform auf der er installiert wird keine Rolle. Administriert wird er direkt über den Browser. Um einen Eindruck davon zu bekommen zeigt die Abbildung 6.2 auf Seite 37 wie die Management Konsole des Weblogic Servers aussieht. Dargestellt ist ein Teil einer Queue-Konfiguration.

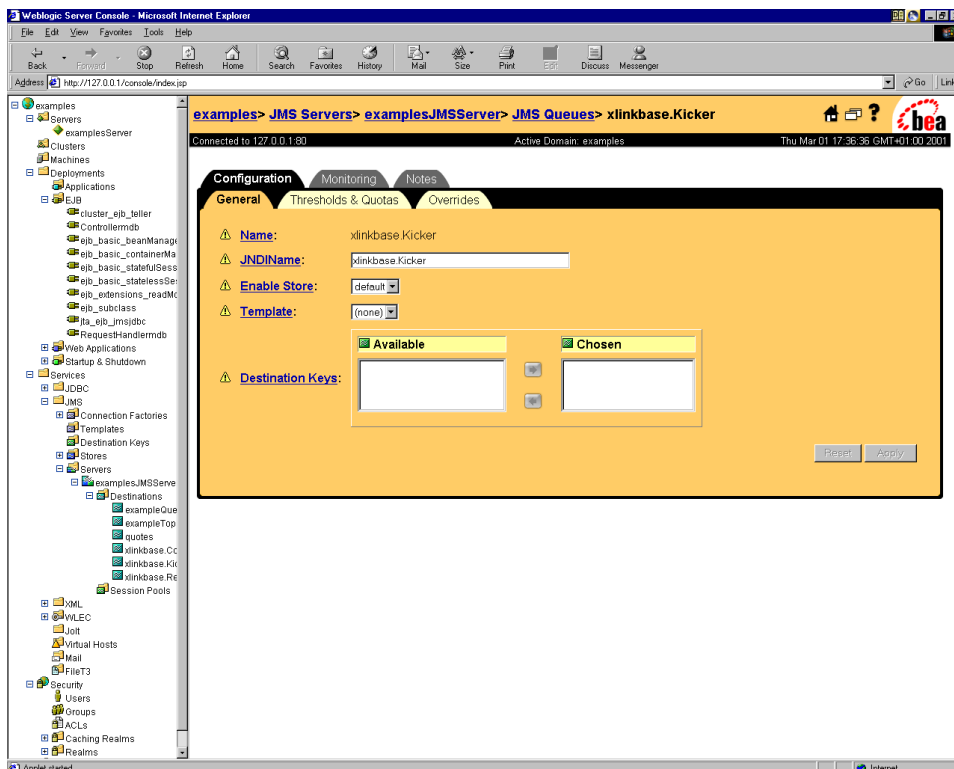


Abbildung 6.2: Die Weblogic Management Konsole

6.2 Das Kicker Servlet

Um eine Klasse zu einem Servlet zu machen, muss diese die abstrakte Klasse `javax.servlet.http.HttpServlet` erweitern und je nachdem, ob man GET oder POST Requests verarbeiten will, die Methode `doGet()` bzw. `doPost()` überschreiben. Eine vollständige JavaDoc Dokumentation der J2EE Klassen und Interfaces findet man übrigens bei Sun unter [18]. Die beiden Methoden `doGet()` und `doPost()` haben zwei Argumente: einen `HttpServletRequest` und einen `HttpServletResponse`. Die `HttpServletRequest` Klasse bietet verschiedene Methoden an um z.B. Header-, MIME-Type- oder Protokoll-Information zu erhalten. Die `HttpServletResponse` Klasse stellt Methoden zur Verfügung um z.B. den Rückgabe MIME-Type, Header-Felder oder den Statuscode zu setzen. Am allerwichtigsten ist aber, dass man einen `PrintWriter` zur Verfügung hat, um über die aktuelle Session zurück zum Client schreiben zu können.

Nachfolgend ein Skeleton eines Servlets, um einen besseren Eindruck von der Funktionsweise zu bekommen:

┌

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SomeServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {

        // Use "request" to read incoming HTTP headers (e.g. cookies)
        // and HTML form data (e.g. data the user entered and submitted)

        // Use "response" to specify the HTTP response line and headers
        // (e.g. specifying the content type, setting cookies).

        PrintWriter out = response.getWriter();
        // Use "out" to send content to browser
    }
}
```

└

Schauen wir uns nun das konkrete Kicker Servlet (Sourcecode siehe Anhang B.1 auf Seite 61) an, implementiert dieses noch das Interface `MessageListener`. Dieses Interface wird implementiert damit das Servlet asynchron Messages empfangen kann. Die einzige Methode, die das Interface deklariert ist die `onMessage()` Methode. Diese wird vom Messaging Service aufgerufen um dem Servlet eine neue Message abzuliefern.

Wird eine Instanz des Servlets vom Servlet Container erzeugt, wird zuerst die `init()` Methode aufgerufen. Diese Methode wird nur einmal aufgerufen und kann z.B. dazu benutzt werden Klassenvariablen zu initialisieren. Das Kicker Servlet nutzt diese um sich über den Container-Kontext (`getServletContext()`) bei der `xlinkbase.Kicker Queue` als Empfänger und bei der `xlinkbase.Controller Queue` als Produzent von Nachrichten zu registrieren. Nach Abarbeitung der `init()` Methode ist das Servlet bereit Requests von einem Client zu empfangen. In unserem Falle sind dies wie schon gesagt immer POST Requests. Schauen wir uns also die `doPost()` Methode ein bisschen genauer an.

Ein Servlet ist von Natur aus reentrant. Das heisst, während ein Request noch abgearbeitet wird, kann schon ein nächster kommen und die `doPost()` Methode aufrufen. Es können sich also mehrere Threads gleichzeitig in der gleichen Servletinstanz befinden. Deshalb ist ein besonderes Augenmerk auf die Synchronisierung von kritischen Ressourcen zu legen, damit das Servlet auch wirklich Threadsafe arbeitet. Im Kicker Servlet gibt es keine kritischen Abschnitte die synchronisiert werden müssten. Einzig der Zugriff auf die Hashtabelle wäre kritisch, wenn sie nicht schon in der Klasse selbst synchronisiert würde.

Ganz ohne Synchronisierung kam ich dann aber doch nicht aus. Der springende Punkt war, dass das Servlet die Verbindung bzw. Session zum Client schliesst, wenn die `doPost()` Methode verlassen wird. Stellt man sich nun vor, dass ich eine Message in die `xlinkbase.Controller Queue` schreibe (nicht-blockierender Aufruf), die `doPost()` Methode weiter abarbeite, in der Hoffnung, die Response kommt zurück bevor ich die Methode verlasse, ist das Problem offensichtlich. Wird der Request rasch genug von allen Komponenten bearbeitet und kommt rechtzeitig zurück, funktioniert alles einwandfrei. Im umgekehrten Falle wird, wie gesagt, die Verbindung geschlossen und der Client bekommt seine Response nicht mehr. Die Lösung ist, den Thread am Ende der `doPost()` Methode schlafen zu legen bis die Response eintrifft. Es macht Sinn über den Response Parameter zu synchronisieren. Trifft die Response ein, wird der schlafende Thread in der `onMessage()`, unmittelbar nach Auslieferung der Response an den Client, wieder aufgeweckt, so dass er die

doPost() Methode verlassen kann und die Verbindung abgebaut wird. Was in der doPost() Methode passiert zeigt folgender Auszug aus dem Code:

┌

```
synchronized(response) {
    while (requests.get(new Integer(requestID))!=null) {
        try {
            log2("Go sleeping");
            response.wait();
            log2("Has been waken up");
        } catch (InterruptedException ie) {}
    }
}
} catch (Exception e) {
    System.err.println("Exception: " + e.getMessage());
    e.printStackTrace();
}
```

└

Um eine Response der MDBs wieder dem richtigen Request zuzuordnen ist ein zusätzlicher Mechanismus nötig, um die Requests bzw. Responses eindeutig indentifizieren zu können. Ich hab dies so gelöst, dass bei Ankunft eines Requests in der doPost() Methode ein neues Object erzeugt wird und ich dessen Hashcode als ID für den Request nehme. Das erzeugte Objekt wird sogleich vom Garbage Collector wieder abgeräumt, da es keinerlei Referenzen auf sich oder andere gibt. Diese ID ist mit grosser Wahrscheinlichkeit eindeutig, da Java den Hashcode aus der OID erzeugt, die ebenfalls mit grosser Wahrscheinlichkeit eindeutig ist, da sie unter anderem aus der Systemzeit generiert wird. Diese ID wird dann zusammen mit dem response-Objekt in eine Hashtabelle abgelegt, so dass die onMessage() Methode bei Ankunft des Responses aufgrund der ID in der Hashtabelle nachschlagen kann, um das entsprechende response-Objekt wieder zurückzubekommen und dem Client anschliessend die Antwort schicken kann. Folgender Codeabschnitt zeigt das Generieren einer ID und das anschliessende Ablegen in der Hashtabelle:

┌

```
int requestID = (new Object()).hashCode();
requests.put(new Integer(requestID), response);
```

Diese ID wird dann mit jeder Nachricht als Property 'RequestID' mitgeschickt. Auch jede Response-Message enthält dieses Feld.

6.3 Das Controller Bean (MDB)

Das Controller Bean übernimmt im Prototypen keine grosse Funktionalität. Ich habe es nur gebraucht um die Performanz der JMS-Kommunikation über mehrere Komponenten hinweg zu testen. Es macht nichts anderes als die Request Message vom Kicker Servlet zu empfangen, sie auszupacken und schliesslich an das RequestHandler Bean weiterzuschicken. An dieser Stelle möchte ich aber näher auf die Message Driven Beans eingehen und zeigen wie man solche implementiert.

Ein MDB besteht immer aus mindestens drei Teilen. Der erste und grösste Teil macht die Java-Klasse selbst aus. Damit aber ein Applikationsserver ein Bean hosten kann, braucht er noch zusätzliche Angaben wie z.B. der (JNDI-)Name unter dem das Bean dem Naming Service bekannt ist oder ob es sich um ein Statefull oder Stateless Bean handelt. Diese Angaben werden in zwei verschiedenen XML-Files gemacht: `ejb-jar.xml` und ein serverspezifisches Deployment Description File (im Falle von Weblogic heisst dieses `weblogic-ejb-jar.xml`). Schauen wir uns ersteres im Falle des Controller Beans an:

┌

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD  
Enterprise JavaBeans 2.0//EN" "http://java.sun.com/dtd/  
ejb-jar_2_0.dtd">
```

```
<ejb-jar>  
  <enterprise-beans>  
    <message-driven>  
      <ejb-name>Controller</ejb-name>  
      <ejb-class>xlinkbase.ejb.Controller</ejb-class>  
      <transaction-type>Container</transaction-type>  
      <message-driven-destination>  
        <jms-destination-type>  
          javax.jms.Queue
```

```

        </jms-destination-type>
    </message-driven-destination>
    <security-identity>
        <run-as-specified-identity>
            <role-name>foo</role-name>
        </run-as-specified-identity>
    </security-identity>
</message-driven>
</enterprise-beans>
</ejb-jar>

```

┘

Mit dieser XML-Datei beschreibt der Entwickler die allgemeinen Eigenschaften eines Beans gemäss der Spezifikation. In obigem Beispiel wird deklariert, dass es sich um ein Message Driven Bean mit dem Namen Controller handelt, das als Nachrichtenquelle eine Queue hat. Die konkrete Queue wird, wie wir sehen werden, im Deployment File spezifiziert. Weiter ist es möglich im Application Server Zugriffsberechtigungen für die Beans festzulegen, so dass nicht jeder Client jedes Bean 'benutzen' kann. Deshalb wird dem Bean eine Security Identity verpasst. Eine gute Übersicht aller möglichen Properties inkl. Beschreibung findet man bei [19].

Das zweite Deployment Description File wird gebraucht um serverspezifische Angaben zu machen. Nachfolgend ist das weblogic-ejb-jar.xml File des Controller Beans abgedruckt.

┌

```

<?xml version="1.0"?>
<!DOCTYPE weblogic-ejb-jar PUBLIC "-//BEA Systems, Inc.//DTD
  WebLogic 6.0.0 EJB//EN" "http://www.bea.com/servers/wls600/
  dtd/weblogic-ejb-jar.dtd">

<!-- Sample MessageDriven bean Weblogic deployment descriptor -->

<weblogic-ejb-jar>
  <weblogic-enterprise-bean>
    <ejb-name>Controller</ejb-name>
    <message-driven-descriptor>
      <pool>

```

```
<max-beans-in-free-pool>200</max-beans-in-free-pool>
  <initial-beans-in-free-pool>10</initial-beans-in-free-pool>
</pool>
  <destination-jndi-name>
    xlinkbase.Controller
  </destination-jndi-name>
</message-driven-descriptor>
  <jndi-name>Controller</jndi-name>
</weblogic-enterprise-bean>
</weblogic-ejb-jar>
```

└

Wie man sieht, muss ich auch hier wieder den Namen des Beans angeben. Als nächstes weise ich den Server an, beim Starten 10 Instanzen des Beans zu erzeugen. Bei hoher Last darf die Anzahl bis auf 200 erhöht werden. Desweiteren wird hier jetzt die Registrierung bei einer konkreten Queue (xlinkbase.Controller) spezifiziert. Es ist zu beachten, dass dieser Descriptor abhängig ist vom eingesetzten Server. Will man also ein MDB auf einem anderen Server einsetzen, muss dieser entsprechend angepasst werden. Hat man einmal diese drei Teile eines Beans beisammen, kompiliert man die Java-Klasse und schnürt alle Files zu einem JAR-Archiv, das dann vom Server verarbeitet werden kann.

Wie sieht es nun aber mit der Implementierung der Klasse aus. Im Vergleich zu Session- oder Entity Beans ist dies ziemlich einfach. Einzig das **MessageDrivenBean** und **MessageListener** Interface müssen implementiert werden. Die Implementierung des MessageDrivenBean Interface ist in fast allen Fällen immer dieselbe und kann dem Source im Anhang [B.2](#) auf Seite [66](#) entnommen werden. Auf das MessageListener Interface wurde bereits im Kapitel [6.2](#) auf Seite [38](#) eingegangen.

6.4 Das RequestHandler Bean

Das RequestHandler Bean ist am Ende der Kette und ist dafür zuständig die gewünschten Informationen zu holen und zurückzuschicken. Als Datenbasis haben wir die xlx-Files des Glossary's von Erik Wilde genommen [\[4\]](#). Jedes Topic ist dort in einem separaten File abgelegt. So war es für den Prototypen nicht nötig eine Komponente für eine Datenbankanbindung zu implementieren.

Als Input bekommt das RequestHandler Bean den weitergeleiteten und unverarbeiteten Request eines Controller Beans. In dieser Komponente wird der Request zum ersten Mal geparkt und zugleich auch der Request erzeugt. Das Parsing des XML-Strings wird durch die zentrale Parser Komponente (siehe Kapitel 6.5 auf Seite 45) vorgenommen. Das Traversieren des Baumes um den Request interpretieren zu können, wird lokal über das JDOM API [12] gemacht. Ich möchte an dieser Stelle noch ein bisschen näher auf das JDOM API eingehen.

Ich habe das JDOM API aus zwei Gründen gewählt. Zum einen ist die Bearbeitung von DOM-Bäumen über ein Standard DOM API, wie sie von Parsern wie Xerces angeboten wird, alles andere als einfach und intuitiv, und der andere, noch viel ausschlaggebendere Punkt ist, dass mit JDOM eine zusätzliche Abstraktionsschicht eingeführt werden kann. Das heisst, dass ich Dokumente unabhängig vom darunterliegenden Parser über das JDOM API parsen und bearbeiten kann. Wenn ich den verwendeten Parser austauschen möchte kann ich dies problemlos tun, ohne dass ich irgendwelche Klassen neu kompilieren muss. Um jegliche Neukompilierung zu verhindern, muss die zu verwendende Parserklasse in einer Umgebungsvariable gespeichert werden, die dann vor der Instanziierung des Parser eingelesen wird. Die Deployment Description Files, die ich in Kapitel 6.3 auf Seite 41 angesprochen habe, bieten diese Möglichkeit. Konkret würde dies dann heissen, die Parserklasse im `weblogic-ejb-jar.xml` zu spezifizieren, anschliessend das JAR-Archiv neu zu erstellen und schliesslich dem Applikationsserver mitzuteilen, das Bean zu aktualisieren. Wird nun über das JDOM API ein Parser instanziiert, muss nur noch die Umgebungsvariable über JNDI ausgelesen und als Argument übergeben werden. Die aktuelle Version des RequestHandlers hat die Parserklasse aber noch fix im Sourcecode codiert.

Nach diesem kleinen Abstecher möchte ich wieder auf die Funktionsweise des RequestHandlers zurückkommen. Nachdem der Request also von der zentralen Parser Komponente geparkt worden ist, muss die eigentliche Query abgearbeitet werden. Der erhaltene JDOM-Tree wird nun in der `getState-``ment()` Methode nach einem Tag mit der Bezeichnung `'stmt'` durchsucht, da dieser das gewünschte Topic enthält. Mit dem JDOM API kann dieses Element mit einem Methodenaufruf gefunden werden. Mit einem 'normalen' Parser API wäre dies viel aufwendiger. Der nachfolgende Codeausschnitt soll veranschaulichen, wie einfach die Suche nach einem bestimmten Element mit JDOM ist:

▮

```
private String getStatement(Document doc) throws IOException
{
    Element root = doc.getRootElement();

    try
    {
        String stmt = root.getChild("stmt").getText();
        return stmt;
    } catch (NullPointerException nlp) {
        log("Not a valid statement!");
        return null;
    }
}
```

┘

Der gefundene String wird dann der aufrufenden `processRequest()` Methode zurückgegeben. Da noch keine eigentliche Query Sprache existiert, gibt der XLinkbase Client das gewünschte Topic einfach in Anführungszeichen an. Die `processRequest()` Methode sucht also nach dem gewünschten Topic, fügt die File-Extension `.xlx` dazu, liest die XML-Datei ein und schickt diese als Response-Message zurück an das Servlet. Normalerweise würde die Response wieder über das Controller Bean zurückgeschickt, damit dieses die Kontrolle über das System bewahren kann. Ich habe aber darauf verzichtet, da dieses im Moment noch keine grossartige Funktionalität besitzt.

6.5 Der Parser

Die zentrale Parser Komponente ist als Stateless Session Bean implementiert. Ein Session Bean ist im Aufbau komplizierter als ein MDB, da es anstatt eine Klasse zwei Klassen plus zwei Interfaces besitzt (Sourcecode siehe Anhang B.4 auf Seite 73). Dasselbe gilt übrigens für Entity Beans auch. Ein Session Bean kann nur über dessen sogenanntes Home Interface, das eine Erweiterung von `EJBHome` sein muss und zudem eine Methode `create()` deklarieren muss, erzeugt werden. Sowohl der EJB Container selbst, als auch Clients erzeugen Instanzen über diese `create()` Methode. Das zweite Interface ist das Remote-Interface, das die `EJBObject` Schnittstelle erweitern muss. In dieser Schnittstelle werden alle Methoden deklariert, die einem Client des Beans zur Verfügung stehen sollen. Implementiert werden diese

Methoden schliesslich in der Implementierungsklasse, die somit die eigentlich Business Logic repräsentiert. Analog zu den MDBs werden die Klassen- und Schnittstellendateien und die zwei Deployment Description Files zu einem JAR-Archiv geschnürt, bevor sie vom Application Server akzeptiert werden. Im Falle des Parser Beans sieht das ejb-xml.xml Description File folgendermassen aus:

┌

```

<?xml version="1.0"?>

<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD
Enterprise JavaBeans 1.1//EN" 'http://java.sun.com/j2ee
/dtds/ejb-jar_1_1.dtd'>

<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>statelessSession</ejb-name>
      <home>xlinkbase.ejb.ParserHome</home>
      <remote>xlinkbase.ejb.Parser</remote>
      <ejb-class>xlinkbase.ejb.ParserBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>statelessSession</ejb-name>
        <method-intf>Remote</method-intf>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>

```

└

Auffallend ist die Spezifikation des Home- und Remote-Interfaces. Diese müssen hier angegeben werden, damit der Server weiss, über welche Interfa-

ces auf die Beans zugegriffen werden kann und er die entsprechenden Stub- und Skeleton Klassen erzeugen kann. Weiter unten folgen Angaben zu den Transaktionseigenschaften, auf die ich aber nicht näher eingehen möchte, da sie für den Prototypen nicht relevant waren.

Schauen wir uns die Implementierungsklasse noch ein bisschen genauer an. Diese muss die Methoden des `SessionBean` und des angesprochenen Remote Interfaces implementieren. In den meisten Fällen hat nur die Methode `setSessionContext()` keine leere Implementierung. Die `setSessionContext()` Methode ist dafür zuständig, die vom Container erhaltene Referenz auf den Session-Kontext in einer Instanzvariable zu speichern. Dies ist schon alles was diese Methode zu tun hat. Eine wichtige Methode ist `ejbCreate()`. Diese wird aufgerufen, wenn eine neue Instanz des Beans erzeugt wird. Sie kann also dazu verwendet werden um Instanzvariablen zu initialisieren. Im Parser Bean wird sie benutzt um ein Parser-Objekt zu erzeugen, so dass dieses nicht jedesmal beim Aufruf von `parse()` erzeugt werden muss. Die Implementierung der `parse()` Methode ist sehr einfach. Der String, der ihr als Argument übergeben wurde, wird an das Parser übergeben, der dann den eigentlichen Parsingprozess startet und ein JDOM Dokument zurückgibt. Ein JDOM Dokument ist ein JDOM spezifischer DOM-Baum. Das JDOM API stellt Methoden zur Transformation eines JDOM-Baumes nach einem DOM-Baum und umgekehrt zur Verfügung.

Kapitel 7

Ausblick

Die Aufgabe des Prototyps war es einerseits zu zeigen, dass das entworfene Konzept für eine Weiterentwicklung geeignet ist und andererseits eine Serverkomponente zur Verfügung zu haben um erste Demonstrationen der XLinkbase mit Clients machen zu können. Ich denke, im Verlaufe der Arbeit und der vielen Tests mit dem Prototypen des XLinkbase Client hat sich gezeigt, dass das gewählte Konzept gut zur Idee der XLinkbase passt. Das XLinkbase Projekt steht aber noch ganz am Anfang und es gibt noch viel Arbeit bis man eine ausgereifte und stabile Serverkomponente haben wird. Ich will nachfolgend ansatzweise aufzeigen an welchen Stellen angesetzt werden kann. An dieser Stelle sei noch angemerkt, dass man noch unzählige Punkte mehr niederschreiben könnte. Es seien hier einfach die wichtigsten genannt.

XLinkbase Superschema

Das XLinkbase Superschema ist ein sehr wichtiger Aspekt des ganzen Projektes, da jede konkrete XLinkbase auf diesem basiert. Dieses Datenmodell entscheidet also, wie einfach und elegant sich Topics und die Beziehungen modellieren lassen. Während der Diplomarbeit haben wir uns, Erik Wilde und Ich, einige Gedanken zum XLinkbase Superschema gemacht. Die im Anhang [A.2](#) auf Seite [55](#) abgedruckte Version 0.4 besitzt sozusagen noch zwei Ebenen. Zum einen die Typ- und zum anderen die Instanzebene. Erfahrungen haben gezeigt, dass es Sinn machen würde, wenn man eine Instanz gleichzeitig wieder als Typ verwenden könnte. Dies würde natürlich eine Verschmelzung der beiden Schichten bedeuten. Erik ist im momentan daran diese Idee in einer nächsten Version der DTD umzusetzen. Ob dies der Weisheit letzter Schluss ist, wird sich zeigen. Aber ich denke, dass dies ein

ganz vernünftiger Ansatz ist.

Abbildung der XLinkbase Daten auf ein RDBMS Schema

Der Prototyp besitzt in der jetzigen Version noch keine Anbindung an eine Datenbank. Die XML-Daten sind in Form von Dateien im Filesystem abgelegt. Für grössere Datenmengen ist diese Lösung natürlich ungeeignet. Sobald die neue DTD verfügbar ist, würde es also Sinn machen eine Abbildung auf ein RDBMS zu finden. Ronald Bourret hat zum Thema Abbildung von XML-Daten auf relationale Datenbanken einige Papers geschrieben, und stellt sogar eine API namens XML-DBMS zur Verfügung (siehe [20]). Im Zusammenhang mit der Abbildung von DTDs sind [21] [22] gute Informationsquellen. Letztere zeigen interessante Konzepte und Ideen von denen man sich bei der Implementierung inspirieren lassen kann.

XSLT-Prozessor Komponente

Eine XSLT-Prozessor Komponente ist im Prototypen noch nicht enthalten. Diese wird aber in kommenden Version des XLinkbase Servers unverzichtbar sein. Sie kann z.B. dazu benutzt werden um die XLinkbase Query Language nach SQL abzubilden oder um die Daten für den Client in ein unterstütztes Format (z.B. HTML oder PDF) zu transformieren, falls der Client keine Stylesheets verarbeiten kann. Da der Prototyp des XLinkbase Clients auf Basis des Internet Explorers und einer ActiveX Komponente Stylesheets verarbeiten kann, stand die Implementierung einer Parser-Komponente im Vordergrund. Die XSLT-Komponente kann analog zum Parser als Stateless Session Bean implementiert werden. Dies würde meiner Meinung nach am meisten Sinn machen.

Translets

Der Einsatz von Translets, die ich bereits im Kapitel 4.1 auf Seite 21 angesprochen habe, wäre eine gute Alternative zur XSLT-Prozessor Komponente. Zum einen wäre der Transformationsprozess um einiges schneller und zum anderen würde man sich an die EJB Spezifikation halten (siehe Abschnitt EJB Restriktionen weiter unten). Wie die Implementierung aussehen würde, müsste man sich genau überlegen. Von mir aus könnte man zwei Ansätze

verfolgen. Eine Idee wäre, die compilierten Stylesheets als normale Java-Klasse zu belassen und sie jeweils bei Bedarf über den Classloader zu laden und instanziiieren. Dies wäre zwar ein sehr performanter Ansatz, würde aber einerseits nicht sehr gut skalieren, da sich normale Klassen nicht ohne weiteres verteilen lassen, und andererseits würde die Komponentenarchitektur nicht konsequent durchgezogen. Ein eleganterer Ansatz wäre, die Translets als MDBs zu kapseln. Das heisst, jedes Translet würde durch ein separates Bean repräsentiert. Dies würde aber nach sich ziehen, dass man den XSLT-Compiler so modifizieren müsste, dass er als Output nicht eine normale Java-Klasse, sondern eine MDB ausgibt, die dann direkt in den XLinkbase Server integriert und über eine Management Konsole konfiguriert werden könnte. Ich denke, der Aufwand für die Modifikation würde sich in Grenzen halten. Den modifizierten XSLT-Compiler würde ich schliesslich in einer 'Translet-Creator' Komponente kapseln.

XLinkbase Query Language

Die Query Language wurde an vielen Stellen angesprochen. Sie stellt sozusagen den Schlüssel zu einer XLinkbase dar und ist deshalb auch sehr wichtig. Das Thema der Query Languages ist zu komplex, als dass ich hier einen Ansatz dafür geben könnte. Im Sommersemester 2001 wird sich aber eine Diplomarbeit damit beschäftigen. Trotzdem möchte ich ganz kurz eine Idee davon vermitteln. Die Query Language soll einerseits ermöglichen Abfragen zum semantischen Netz zu machen, andererseits über entsprechende Konstrukte auch ermöglichen Daten hinzuzufügen, zu editieren und zu löschen. Diese Abfragen beziehen sich aber nicht nur auf die XLinkbase Daten an sich, sondern können sich auch auf Sekundärdaten wie Benutzer, Zugriffsrechte oder verfügbare Stylesheets beziehen.

EJB Restriktionen

Die EJB Spezifikation beschreibt Vorgaben, damit Beans zuverlässig, sicher und vor allem portabel sind. So sollte man es z.B. vermeiden Socketverbindungen aufzubauen. Eine gute Zusammenfassung über alle Restriktionen gibt [23]. In einer zukünftigen Version des Servers sollte darauf geachtet werden alle Vorgaben einzuhalten. Mein Prototyp verletzt nur die Vorgabe, Dateizugriffe zu vermeiden. Die RequestHandler Komponente z.B. liest das angeforderte File vom Dateisystem. Da aber fast ausnahmslos alle Application Server diese Restriktionen nicht konsequent beachten, funktionieren diese

Zugriffe trotzdem. Als Lösung dieses Problems könnte man die XML-Dateien als BLOB-Stream über eine JDBC Komponente in einer Datenbank ablegen.

Object- vs. TextMessages

Im Prototypen werden alle Messages als TextMessages verschickt. Der XML-Request liegt dabei in String Form vor und wird als Property mitgeschickt. Stellt man sich vor, dass viele Komponente, die nacheinander die Meldung bekommen, den Request bearbeiten wollen, macht der Einsatz von Text Messages keinen grossen Sinn, da jede Komponente den Request parsen, bearbeiten und zum Weitersenden wieder in Stringform bringen muss. Viel eleganter wäre es, wenn man direkt den JDOM-Baum als Message verschicken könnte. Diese Funktionalität wird vom JMS in Form von **Object** Messages, die es erlauben serialisierbare Objekte zu verschicken, unterstützt. Leider wird diese Funktion zur Zeit noch von keinem Application Server unterstützt. In diesem Zusammenhang würde es Sinn machen, dass jede Komponente beide Arten von Messages versteht und entsprechend reagiert. Denn je nach Funktionalität der Quell- und Zielkomponente, macht die eine oder andere Art mehr Sinn. Auf welche Art eine Komponente den Request weiterleiten soll, könnte z.B. in den Routinginformationen stehen.

Routing

Das Routing der Request und Responses ist für eine grosse XLinkbase mit vielen Komponenten sehr wichtig um Anfragen effizient abarbeiten zu können und um flexibel zu bleiben. Im Moment, wo nur ganz wenige Komponenten vorhanden sind, ist dieser Aspekt sicherlich zweitrangig. Je mehr Komponenten dazukommen umso wichtiger wird die Möglichkeit des Routing. Das Thema des bedingten Routing ist aber nicht ganz trivial und will sorgfältig überlegt sein. Schaut man sich z.B. Workflow Systeme an, wurde in diesem Bereich bereits viel Vorarbeit geleistet. Als guter Ausgangspunkt könnte die Website der Workflow Management Coalition dienen [24].

Management Tools

Im Moment müssen alle Einstellungen für die Komponenten von Hand gemacht werden. Für zukünftige Versionen des XLinkbase Servers wäre es wünschenswert eine zentrale Management Konsole zur Verfügung zu haben,

die Einstellungen für das Routing, für Benutzer(-rechte), usw. zulässt. Zudem könnte sie Statusinformationen zum Systemzustand anbieten.

XLinkbase Light

Man kann sich Szenarien vorstellen, wo einerseits nicht extrem viele Daten verwaltet werden sollen und wo andererseits keine grosse Netzwerkinfrastruktur vorhanden ist. Trotzdem könnte es in einer solchen Umgebung wünschenswert sein, eine XLinkbase zu haben. Der Einsatz der gezeigten Lösung mit einem Application Server wäre aber ein glatter Overkill. Angebracht wäre viel eher ein abgespeckte Version, die ohne die komplexe Laufzeitumgebung auskommt. Eine Variante zur Lösung wäre der Einsatz eines kleinen Webservers mit integrierter Servlet Engine, der lokal betrieben wird. Der grösste Aufwand läge darin, die wichtigste Funktionalität der XLinkbase auf Servlets zu portieren. Auf einige Funktionalität wie dynamisches Routing würde man aber sicher verzichten. Hier könnte ich mir den Einsatz von Cocoon vorstellen.

Anhang A

Document Type Definitions

A.1 Request DTD

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT xlbrequest (stmt?)>
<!ATTLIST xlbrequest
  xlbql CDATA #REQUIRED
>
<!ELEMENT stmt ANY>
```

A.2 XLinkbase DTD Version 0.4

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- xlb schema version 0.4, dret, 12/13/2000 -->

<!ENTITY % facet-types "
  type (simple | complex | external) #REQUIRED
">

<!-- an occurrences model equivalent to that of xml itself (" ", "+", "?", "*")
<!-- this attribute determines whether an xlb facet is fixed for a type, fixed
for an instance, or may modified in instance references -->
<!ENTITY % fixed "
  fixed (no | instance | type)
">
```

```

<!-- an occurrences model equivalent to that of xml itself (" ", "+", "?", "*")
<!ENTITY % occurrences "
    occurrences (once | repeat | optional | optionalrepeat)
">

<!-- each type has a name, abstract types may not be instantiated,
    final types may not be used as supertypes -->
<!ENTITY % type-attributes "
    name ID #REQUIRED
    abstract ( yes | no ) #IMPLIED
    final (yes | no) #IMPLIED
">

<!-- the default for 'abstract' and 'final' is no', but for some strange
    reason xml spy does not accept a default value to be specified -->

<!-- an instance may have multiple types, for no supertype pair x and y
    of an instance x is allowed to be a supertype of y -->
<!ENTITY % instance-attributes "
    type NMTOKENS #REQUIRED
">

<!-- in xml schema terms, the 'name' attribute has to be a qname,
    i.e. is has to be valid with respect to a superschema of the schema
    as identified in superschema element(s). strictly speaking, it must a
    sequence of qnames, but i don't know whether xml schema supports a
    qnames type analog to the NMTOKENS type... -->

<!-- a reference specifies the type that it is referencing -->
<!ENTITY % ref-attributes "
    name NMTOKEN #REQUIRED
">

<!-- in xml schema terms, the 'name' attribute has to be a qname,
    i.e. is has to be valid with respect to a superschema of the schema as
    identified in superschema element(s) -->

<!-- this is the root element, defining the two main building blocks
    of an xlinkbase, the types and the instances -->
<!ELEMENT xlb (annotation?, xlb-schema, xlb-instances) >
<!ATTLIST xlb
    version CDATA #REQUIRED
>

<!-- the 'name' attribute defines the preferred prefix for this
    xlb schema (which may or may not be used by xlb schemas using this

```

schema), and the namespace-attribute defines the namespace

```

<!ELEMENT xlb-schema (superschema*, (association-type | role-type |
  topic-type | facet-type)*) >
<!ATTLIST xlb-schema
  name NMTOKEN #REQUIRED
  namespace CDATA #REQUIRED
>

<!ELEMENT xlb-instances (association | topic)* >

<!-- this annotation element is used for documentation purposes,
  and different kind of descriptions can be used, differentiated by
  the type attribute -->
<!ELEMENT annotation (description+) >
<!ELEMENT description ANY >
<!ATTLIST description
  type CDATA #IMPLIED
>

<!-- a superschema is identified by its namespace and the prefix associated
  with it. types from this schema can then be referenced by a qname using
  the superschema namespace prefix. -->
<!ELEMENT superschema (annotation?) >
<!ATTLIST superschema
  ns-prefix NMTOKEN #REQUIRED
  namespace CDATA #REQUIRED
>

<!ELEMENT acl ANY >
<!ATTLIST acl
  user CDATA #REQUIRED
  group CDATA #REQUIRED
  created CDATA #REQUIRED
  modified CDATA #REQUIRED
  permissions CDATA #REQUIRED
>

<!ELEMENT association-type (acl, annotation?, supertype*, facet-type-ref*,
  role-type-ref+) >
<!ATTLIST association-type
  %type-attributes;
>

```

```
<!-- an association is instantiated by defining its facets and roles,
it must contain at least one role -->
<!ELEMENT association (acl, facet*, role+) >
<!ATTLIST association
    name ID #REQUIRED
    %instance-attributes;
>
```

```
<!-- a role is instantiated by defining its facets and references to
topic instances, it must contain at least one topic reference -->
<!ELEMENT role (facet*, topic-ref+) >
<!ATTLIST role
    %instance-attributes;
>
```

```
<!ELEMENT role-type (acl, annotation?, supertype*, facet-type-ref*) >
<!ATTLIST role-type
    %type-attributes;
>
```

```
<!ELEMENT role-type-ref (annotation?, topic-type-ref+) >
<!ATTLIST role-type-ref
    %ref-attributes;
    %occurrences; #REQUIRED
>
```

```
<!-- role instances may be repeated within one association, because
different role instances may have different facets -->
```

```
<!ELEMENT topic-type (acl, annotation?, supertype*, facet-type-ref*) >
<!ATTLIST topic-type
    %type-attributes;
>
```

```
<!ELEMENT topic-type-ref (annotation?) >
<!ATTLIST topic-type-ref
    %ref-attributes;
    %occurrences; #REQUIRED
>
```

```
<!ELEMENT topic-ref (facet*) >
<!ATTLIST topic-ref
    %ref-attributes;
```

```
    weight CDATA #REQUIRED
>
<!-- the weight is a real number which is used to specify how "good"
a topic fits into an association (or the role), ranging from 0.0
(not at all) to 1.0 (fits perfectly) -->

<!-- a topic is instantiated by defining its facets -->
<!ELEMENT topic (acl, facet+) >
<!ATTLIST topic
    name ID #REQUIRED
    %instance-attributes;
>

<!ELEMENT facet-type-ref (annotation?, facet?) >
<!ATTLIST facet-type-ref
    %ref-attributes;
    %occurrences; #REQUIRED
    %fixed; #IMPLIED
>
<!-- the 'fixed' attribute is only allowed for topic facet-type-refs,
allowing non-fixed facets to be changed in topic references within
role references in association instances -->
<!-- the default for 'fixed' is 'instance', but for some strange reason
xml spy does not accept a default value to be specified -->
<!-- the facet element is not a good solution of the problem that we
would like to be able to specify facet values in types, and not only
in instances -->

<!ELEMENT facet-type (acl, annotation?) >
<!ATTLIST facet-type
    name ID #REQUIRED
    type NMTOKEN #REQUIRED
    %facet-types;
>
<!-- simple types use a reference to a predefined set of facet types,
such as some or all of the types provided by xml schema datatypes -->
<!-- complex facets a reference to a type from a non-standard namespace,
which can or cannot be interpreted, depending on whether the types defined
by the namespace-uri are known -->
<!-- this type references a type described by a non-standard namespace
and present external to the xlinkbase. the external-facet-type contains
any information that is necessary to retrieve attributes of that type from
the external source -->
```

```
<!ELEMENT facet ANY >
<!ATTLIST facet
  name NMTOKEN #IMPLIED
>
<!-- the name attribute must be used for facets occurring in instances,
and must be omitted for facets occurring in the schema -->

<!ELEMENT supertype (annotation?) >
<!ATTLIST supertype
  %ref-attributes;
>
```

Anhang B

Java Source Code

B.1 Kicker Servlet

```
package xlinkbase.servlet;

import javax.servlet.*;
import javax.servlet.http.*;

import java.rmi.RemoteException;

import javax.jms.*;
import java.io.*;
import java.util.*;

import javax.ejb.CreateException;
import javax.ejb.RemoveException;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class Kicker extends HttpServlet implements MessageListener
{
    final static String JMS_FACTORY="weblogic.jms.ConnectionFactory";

    long servletID;
    Context ctx;
    Hashtable requests = new Hashtable();
```

```
QueueConnectionFactory qconFactory;
QueueConnection qcon;
QueueSession qsession;
QueueSender qsender;
QueueReceiver qreceiver;
Queue queueInput;
Queue queueOutput;

private void log2 (String msg)
{
    System.out.println("Kicker Servlet: " + msg);
}

public void onMessage(Message msg)
{
    try {
        String msgText;

        if (msg instanceof TextMessage) {
            msgText = ((TextMessage)msg).getText();
        } else {
            msgText = msg.toString();
        }

        log2("Response for request " + msg.getIntProperty("RequestID") +
            " arrived");

        HttpServletResponse response =
            (HttpServletResponse)requests.get(
                new Integer(msg.getIntProperty("RequestID")));

        if (response != null) {

            String resp = new String(msg.getStringProperty("Response"));

            response.setContentType("text/html");
            response.setHeader("Content-Length",
                String.valueOf(resp.length()));
            // doesn't work properly
            // response.setContentLength(resp.length())
        }
    }
}
```

```
        PrintWriter out = new PrintWriter(
            response.getOutputStream(), true);

        out.println(resp);
        out.close();

        synchronized(response) {
            requests.remove(new Integer(msg.getIntProperty("RequestID")));
            response.notifyAll(); // Notify request thread to quit
        }

        log2("Stream is written");
    }
} catch (JMSEException jmse) {
    System.err.println("JMSEException: " + jmse.getMessage());
    jmse.printStackTrace();
} catch (IOException ioe) {
    System.err.println("I/O exception: " + ioe.getMessage());
    ioe.printStackTrace();
}
}

public void doGet (HttpServletRequest request,
                  HttpServletResponse response)
    throws ServletException, IOException
{
    response.setContentType("text/html");
    response.setContentLength(11);
    Writer out = response.getWriter();

    try
    {
        out.write("Kicker Test");
        out.flush();
    } catch (Exception e) {}

    out.close();
}
```

```
public void doPost (HttpServletRequest request,
                   HttpServletResponse response)
    throws ServletException, IOException
{
    BufferedReader str = request.getReader();
    int len = request.getContentLength();
    char[] buffer = new char[len];
    str.read(buffer, 0, len);

    try
    {
        int requestID = (new Object()).hashCode();
        requests.put(new Integer(requestID), response);

        log2("Request " + requestID + " arrived");
        log2("Request " + new String(buffer));

        TextMessage msg = qsession.createTextMessage();
        msg.setJMSReplyTo(queueInput);
        msg.setText("Message from Kicker servlet");
        msg.setStringProperty("Target", "Controller");
        msg.setIntProperty("RequestID", requestID);
        msg.setStringProperty("Request", new String(buffer));

        qsender.send(msg);

        synchronized(response) {
            while (requests.get(new Integer(requestID))!=null) {
                try {
                    log2("Go sleeping");
                    response.wait();
                    log2("Has been waken up");
                } catch (InterruptedException ie) {}
            }
        }
        } catch (Exception e) {
            System.err.println("Exception: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

```
public void cleanUp()
{
    try
    {
        qsender.close();
        qsession.close();
        qcon.close();
    } catch (JMSEException jmse) {
        System.err.println("JMS exception: " + jmse.getMessage());
        jmse.printStackTrace();
    }
}

private static Context getInitialContext() throws NamingException
{
    return new InitialContext();
}

public void init (ServletConfig config) throws ServletException
{
    super.init(config);

    try
    {
        servletID = System.currentTimeMillis();

        ctx = new InitialContext();
        qconFactory = (QueueConnectionFactory) ctx.lookup(JMS_FACTORY);
        qcon = qconFactory.createQueueConnection();
        qsession = qcon.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
        queueInput = (Queue) ctx.lookup("xlinkbase.Kicker");
        queueOutput = (Queue) ctx.lookup("xlinkbase.Controller");
        qsender = qsession.createSender(queueOutput);
        qreceiver = qsession.createReceiver(queueInput);
        qreceiver.setMessageListener(this);

        qcon.start();

    } catch (NamingException ne) {
        System.err.println("Naming exception: " + ne.getMessage());
        ne.printStackTrace();
    }
}
```

```
    } catch (JMSEException jmse) {
        System.err.println("JMS exception: " + jmse.getMessage());
        jmse.printStackTrace();
    }
}

public void destroy()
{
    cleanUp();
}
}
```

B.2 Controller Bean

```
package xlinkbase.ejb;

import weblogic.rmi.RemoteException;

import java.io.*;
import java.util.*;

import javax.ejb.*;
import javax.jms.*;
import javax.naming.*;

public class Controller implements MessageDrivenBean, MessageListener
{

    private static final boolean VERBOSE = true;
    private MessageDrivenContext m_ctx;
    private Context ctx;

    final static private String QUEUE_NAME = "xlinkbase.RequestHandler";
    final static String JMS_FACTORY="weblogic.jms.ConnectionFactory";
    final static String JNDI_FACTORY="weblogic.jndi.WLInitialContextFactory";

    QueueConnectionFactory qconFactory;
    QueueConnection qcon;
    QueueSession qsession;
```

```
QueueSender qsender;
QueueReceiver qreceiver;
Queue queue;

private void log(String s)
{
    if (VERBOSE) System.out.println("Controller MDB: " + s);
}

public void ejbActivate() {}
public void ejbRemove() {}
public void ejbPassivate() {}
public void ejbCreate () throws CreateException
{
    try
    {
        log("Controller instance created");
        ctx = new InitialContext();
        qconFactory = (QueueConnectionFactory) ctx.lookup(JMS_FACTORY);
        qcon = qconFactory.createQueueConnection();
        qsession = qcon.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
        queue = (Queue) ctx.lookup(QueueName);
        qsender = qsession.createSender(queue);

        qcon.start();

    } catch (NamingException ne) {
        System.err.println("Naming exception: " + ne.getMessage());
        ne.printStackTrace();
    } catch (JMSEException jmse) {
        System.err.println("JMS exception: " + jmse.getMessage());
        jmse.printStackTrace();
    }
}

public void setMessageDrivenContext(MessageDrivenContext mctx)
{
    log("setMessageDrivenContext called");
    m_ctx = mctx;
}
```

```
public void onMessage(Message msg)
{
    TextMessage tm = (TextMessage) msg;

    try {
        log("Request " + msg.getStringProperty("RequestID") + " arrived");

        // process request (routing, ...)
        // ...

        // forward request incl. routing info to the first mdb

        sendMessage(msg.getIntProperty("RequestID"),
                    msg.getStringProperty("Request"));

    } catch(Exception ex) {
        ex.printStackTrace();
    }
}

public void sendMessage(int requestID, String request)
{
    try
    {
        TextMessage msg = qsession.createTextMessage();
        msg.setJMSReplyTo(queue);
        msg.setText("Message from ControllerMDB");
        msg.setIntProperty("RequestID",requestID);
        msg.setStringProperty("Request",request);

        qsender.send(msg);
    } catch (Exception e) {
        System.err.println("Exception: " + e.getMessage());
        e.printStackTrace();
    }
}

private static InitialContext getInitialContext()
    throws NamingException
{
    Hashtable env = new Hashtable();
```

```
    env.put(Context.INITIAL_CONTEXT_FACTORY, JNDI_FACTORY);
    env.put(Context.PROVIDER_URL, "t3://localhost:7001");
    return new InitialContext(env);
}
}
```

B.3 RequestHandler Bean

```
package xlinkbase.ejb;

import weblogic.rmi.RemoteException;

import java.util.*;
import java.io.*;

import javax.ejb.*;
import javax.jms.*;
import javax.naming.*;

import org.jdom.*;
import org.jdom.input.*;
import org.jdom.output.*;

public class RequestHandler implements MessageDrivenBean,
    MessageListener
{
    private static final boolean VERBOSE = true;
    private MessageDrivenContext m_ctx;
    private Context ctx;

    final static private String QUEUE_NAME = "xlinkbase.Kicker";
    final static String JMS_FACTORY="weblogic.jms.ConnectionFactory";
    final static String JNDI_FACTORY="weblogic.jndi.WLInitialContextFactory";

    Parser parser;
    QueueConnectionFactory qconFactory;
    QueueConnection qcon;
    QueueSession qsession;
    QueueSender qsender;
```

```
QueueReceiver qreceiver;
Queue queue;

private void log(String s)
{
    if (VERBOSE) System.out.println("RequestHandler MDB: " + s);
}

public void ejbActivate() {}
public void ejbRemove() {}
public void ejbPassivate() {}
public void ejbCreate () throws CreateException
{
    try
    {
        log("RequestHandler instance created");
        ctx = new InitialContext();
        qconFactory = (QueueConnectionFactory) ctx.lookup(JMS_FACTORY);
        qcon = qconFactory.createQueueConnection();
        qsession = qcon.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
        queue = (Queue) ctx.lookup(Queue_NAME);
        qsender = qsession.createSender(queue);

        qcon.start();

        ParserHome parserHome =
            (ParserHome)ctx.lookup("xlinkbase/Parser");
        parser = parserHome.create();

    } catch (NamingException ne) {
        System.err.println("Naming exception: " + ne.getMessage());
        ne.printStackTrace();
    } catch (JMSException jmse) {
        System.err.println("JMS exception: " + jmse.getMessage());
        jmse.printStackTrace();
    } catch (java.rmi.RemoteException re) {
        System.err.println("Remote exception: " + re.getMessage());
        re.printStackTrace();
    }
}
```

```
public void setMessageDrivenContext(MessageDrivenContext mctx)
{
    log("setMessageDrivenContext called");
    m_ctx = mctx;
}

private String getStatement(Document doc) throws IOException
{
    Element root = doc.getRootElement();

    try
    {
        String stmt = root.getChild("stmt").getText();
        return stmt;
    } catch (NullPointerException nlp) {
        log("Not a valid statement!");
        return null;
    }
}

private String processRequest(String request)
{
    Document doc;
    StringWriter str = new StringWriter();
    char[] buffer = null;

    log("Process request");

    try
    {
        doc = parser.parse(request);

        String stmt = getStatement(doc);
        String xlx = stmt.substring(stmt.indexOf('')+1,
                                   stmt.lastIndexOf(''));

        log("Requested file: " + xlx + ".xlx");
        File xlxfile = new File("/Xlinkbase/xlx/" + xlx + ".xlx");
        int len = (new Long(xlxfile.length())).intValue();
        buffer = new char[len];
        FileReader fr = new FileReader(xlxfile);
```

```
        fr.read(buffer, 0, len);
        fr.close();

    } catch (Exception e) {
        log("Error in parsing: " + e.getMessage());
    }

    return new String(buffer);
}

public void onMessage(Message msg)
{
    TextMessage tm = (TextMessage) msg;
    try {
        String text = tm.getText();
        log("Request " + msg.getStringProperty("RequestID") + " arrived");
        sendMessage(msg.getIntProperty("RequestID"),
                    processRequest(msg.getStringProperty("Request")));
    } catch (JMSEException ex) {
        ex.printStackTrace();
    }
}

public void sendMessage(int requestID, String response)
{
    try
    {
        TextMessage msg = qsession.createTextMessage();
        msg.setJMSReplyTo(queue);
        msg.setText("Message from RequestHandlerMDB");
        msg.setIntProperty("RequestID", requestID);
        msg.setStringProperty("Response", response);

        qsender.send(msg);
    } catch (Exception e) {
        System.err.println("Exception: " + e.getMessage());
        e.printStackTrace();
    }
}

private static InitialContext getInitialContext()
```

```
    throws NamingException
  {
    Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY, JNDI_FACTORY);
    env.put(Context.PROVIDER_URL, "t3://localhost:7001");
    return new InitialContext(env);
  }
}
```

B.4 Parser (Stateless Session Bean)

B.4.1 Home Interface

```
package xlinkbase.ejb;

public interface ParserHome extends javax.ejb.EJBHome
{
    public Parser create()
        throws java.rmi.RemoteException, javax.ejb.CreateException;
}
```

B.4.2 Remote Interface

```
package xlinkbase.ejb;

public interface Parser extends javax.ejb.EJBObject
{
    public org.jdom.Document parse(String xmlstring)
        throws java.rmi.RemoteException;
}
```

B.4.3 Bean Implementierung

```
package xlinkbase.ejb;

import javax.naming.*;
import java.util.*;
```

```
import org.jdom.*;
import org.jdom.input.*;

public class ParserBean implements javax.ejb.SessionBean
{
    private javax.ejb.SessionContext _sessionContext;
    private Context ctx;

    final static String SAX_DRIVER_CLASS =
        "org.apache.xerces.parsers.SAXParser";

    SAXBuilder builder;

    public ParserBean()
    { // EJB constructors don't have a server context.
    }

    public void ejbActivate()
        throws javax.ejb.EJBException, java.rmi.RemoteException{}
    public void ejbPassivate()
        throws javax.ejb.EJBException, java.rmi.RemoteException{}
    public void ejbRemove()
        throws javax.ejb.EJBException, java.rmi.RemoteException{}

    public void setSessionContext( javax.ejb.SessionContext parm0 )
        throws javax.ejb.EJBException, java.rmi.RemoteException
    {
        this._sessionContext = parm0;
    }

    public void ejbCreate()
        throws javax.ejb.EJBException, javax.ejb.CreateException
    {
        try
        {
            builder = new SAXBuilder(SAX_DRIVER_CLASS);
        } catch (Exception e)
        {
            System.err.println("Unexcepted exception: "+e.getMessage());
            e.printStackTrace();
        }
    }
}
```

```
}

public Document parse(String xmlstring)
    throws java.rmi.RemoteException
{
    Document doc = null;
    try
    {
        doc = builder.build(new java.io.StringReader(xmlstring));
    } catch (Exception e) {
        System.err.println("Exception while parsing: " +
            e.getMessage());
        e.printStackTrace();
    }

    return doc;
}
}
```


Anhang C

CD-ROM

Ich habe eine CD-ROM mit allen relevanten Dateien gebrannt und beigelegt:

/Bea Weblogic/ Dieses Verzeichnis enthält ein 30 Tage Trial Version des BEA Weblogic 6.0 Servers. Das Konfigurationsfile config.xml muss nach der Installation ins Verzeichnis /bea/wlserver6.0/config/examples kopiert werden. Die Datei ejb20.zip muss entsprechend dem Readme File installiert werden.

/Bericht/ Hier findet man diesen Bericht im pdf-Format.

/Source/ Der komplette Source der Komponenten. Die Files sind in zwei Pakete geschnürt: xlinkbase.ejb und xlinkbase.servlet.

Literaturverzeichnis

- [1] Der Speicherbedarf explodiert
<http://www.computerworld.ch/domino/CWArchiv.nsf/378dd4e611038e3a412565b2005c1621/52687092eb2c3296412569d9004d426e?OpenDocument> 3
- [2] Object Management Group
<http://www.omg.org> 11
- [3] Der Topic Map Standard (ISO13250)
<http://www.infoloom.com/tnm/draft27.htm> 3
- [4] Glossary zu Erik Wilde's Buch 'Wilde's WWW'
<http://wildesweb.com/glossary/> 5, 43
- [5] Microsoft's .NET Platform
<http://microsoft.com/business/products/webplatform/default.asp> 19
- [6] The Apache XML Project
<http://xml.apache.org/> 21, 27
- [7] Brett McLaughlin
JAVA and XML
O'Reilly, ISBN 0-596-00016-2 27
- [8] Sybase's Enterprise Application Server
<http://www.sybase.com/products/applicationservers/easerver/> 23, 35
- [9] Ein XSLT Compiler für Java
<http://www.sun.com/software/xml/developers/xsltc/index.html;sessionidDFWQSUIAAAJYZAMTA1LU5YQ> 22
- [10] Reactor – An Object Behavioral Pattern for Event Demultiplexing and Event Handler Dispatching.
<http://www.cs.wustl.edu/~schmidt/PDF/reactor-siemens.pdf> 27

- [11] Ed Roman
Mastering Enterprise JavaBeans and the Java 2 Platform, Enterprise Edition
Wiley Computer Publishing, ISBN 0-471-33229-1
[30](#)
- [12] Die JDOM API
<http://www.jdom.org> [31](#), [44](#)
- [13] HTTP - Hypertext Transfer Protocol
<http://www.w3.org/Protocols/> [32](#)
- [14] jBoss - an Open Source Application Server
<http://www.jboss.org> [35](#)
- [15] Jetty - an Open Source HTTP Servlet Server
<http://www.jboss.org/newsite/business/jboss-jetty.html> [35](#)
- [16] BEA Weblogic 6.0 Server
<http://www.bea.com/products/weblogic/server/index.shtml> [35](#)
- [17] Ant - A Java based build tool
<http://www.itpath.com/userGuide.htm> [37](#)
- [18] J2SDKEE JavaDoc
<http://java.sun.com/j2ee/j2sdkee/techdocs/api/index.html> [38](#)
- [19] EJB Deployment Properties
<http://edocs.bea.com/wls/docs60/ejb/reference.html#1048022> [42](#)
- [20] XML-DBMS - Middleware for Transferring Data between XML Documents and Relational Databases
<http://www.rpbouret.com/xmldbms/index.htm> [50](#)
- [21] Mapping DTDs to Databases
<http://www.rpbouret.com/xml/DTDToDatabase/index.htm> [50](#)
- [22] Mapping Objects To Relational Databases
<http://www.ambysoft.com/mappingObjects.pdf> [50](#)
- [23] Programming restrictions on EJB
<http://www.javaworld.com/javaworld/jw-08-2000/jw-0825-ejbrestrict.html> [51](#)
- [24] Mapping Objects To Relational Databases
<http://www.aiim.org/wfmc/> [52](#)