JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis

Monitoring von Anwendungsservern mit Java Management Extensions (JMX)

MAGISTERARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

in der Studienrichtung

INFORMATIK

Eingereicht von:
Johannes Hölzl, 9855117

Angefertigt am:
Institut für Informationsverarbeitung und Mikroprozessortechnik

Betreuung:
o.Prof. Dr. Jörg R. Mühlbacher
Dipl.-Ing. Rudolf Hörmanseder

Linz, Mai 2007

Danksagung

An dieser Stelle möchte ich meinen herzlichen Dank an all jene richten, die mich bei dieser Magisterarbeit unterstützt haben.

Herzlichen Dank an o. Prof. Dr. Jörg R. Mühlbacher und Dipl.-Ing. Rudolf Hörmanseder sowie an die Mitarbeiter des Instituts für Informationsverarbeitung und Mikroprozessortechnik für die Betreuung im Studium und bei dieser Magisterarbeit.

Weiters danke ich sehr herzlich meinen Kollegen bei Borland Software Entwicklung in Linz, die eine praktische Umsetzung dieser Arbeit in einem Produkt ermöglicht haben.

Vielen Dank an Karina Lengauer, Udo Kernecker und Markus Lenger für das Korrekturlesen dieser Arbeit und die wertvollen Hinweise zur äußeren Form und Strukturierung.

Zuletzt vielen Dank an meine Familie und an meine Freundin Pamela Kargl für die liebevolle Unterstützung.

Zusammenfassung

Diese Arbeit beschäftigt sich damit, mit Hilfe der Technologie Java Management Extensions (JMX) und der damit verbundenen Standards, auf Performanz-Daten von Java Anwendungsservern und Applikationen zuzugreifen.
Einen wesentlichen Kernpunkt dieser Arbeit stellt die Entwicklung einer offenen und erweiterbaren Architektur, die das gleichzeitige Monitoring von JMX Daten verschiedener Hersteller ermöglicht, dar.
Ein weiterer wichtiger Bereich ist es, große Datenmengen für den Benutzer auf übersichtliche Weise zu strukturieren und den Zugriff auf Detaildaten aus komplexen Datenstrukturen wie zum Beispiel Tabellen zu ermöglichen.
Schließlich zeigt diese Arbeit, wie die JMX Daten auch für Monitoring Tools, die auf der Programmiersprache C/C++ basieren, verfügbar gemacht werden können.

Abstract

This work focuses on using the Java Management Extensions (JMX) technology and its related standards to access performance data of Java application servers and applications.
A core part of this work is to design and implement an open and flexible architecture that allows monitoring of JMX data of various vendors simultaneously.
Another important aspect is to structure the huge amounts of data in a clearly arranged way and to provide a way to access details of complex data structures like tables.
Finally, this work shows how to make all JMX data available for C/C++ based monitoring tools.

Danksagung....2
Zusammenfassung....3
1. Einleitung....9
1.1. Vorgehensweise und Gliederung....9
1.1.1. Verwendung von Anglizismen....9
1.1.2. Änderungen des Schriftbildes....9
1.1.3. Gliederung....10
1.2. Problemstellung und Motivation....11
1.3. Begriffe....14
1.3.1. Java Management Extensions (JMX)....14
1.3.2. MBean....14
1.3.3. MBean Server....14
1.3.4. ObjectName....15
1.3.5. JMX Agent....15
1.3.6. JSR-003....15
1.3.7. JSR-77....15
1.3.8. JSR-160....16
1.3.9. JMXServiceURL....16
1.3.10. Native Code....16
1.3.11. Java Native Interface (JNI)....17
1.3.12. Java Naming and Directory Interface (JNDI)....17
1.3.13. Remote Method Invocation (RMI)....17
1.4. Anwendungsbeispiel aus Benutzersicht....18
1.5. Ziele und Forschungsfragen....25
2. Grundlagen....26
2.1. Grundbegriffe des JNDI....26
2.1.1. Binding, Bindname....26
2.1.2. Context....26
2.1.3. Initial Context....26
2.2. Die JMX Architektur....28
2.2.1. Allgemeine JMX Architektur....28
Manageable Ressource....28
MBean....28

ObjectName 31
MBean Server 31
JMX Agent 33
2.2.2. Einteilung in Schichten 34
2.2.3. Architektur bei JSR-77 34
2.3. JMX Query Mechanismus 36
2.3.1. Query Scope 36
2.3.2. Query Expression 37
3. JMX Monitoring Client API 38
3.1. Begriffsdefinition und Abgrenzung 38
3.2. Architektur 38
3.2.1. Geschlossene Client Architektur 38
3.2.2. Verteilte Client Architektur mit Verwaltung über RMI Registry 39
3.2.3. Verwendung einer ORB Architektur 42
3.3. Implementierung einer verteilten JMX Client Architektur 43
3.3.1. Starten des RMI Server 44
Starten des Java Prozesses 44
Starten und Wiederverwendung der RMI Registry 45
Registrieren des RMI Server in der RMI Registry 46
Verbinden zum RMI Server 46
3.3.2. Verbinden zum MBean Server 47
3.3.3. Abfragen von Monitoring Daten 48
3.3.4. Terminierung des RMI Servers Prozesses 49
Mehrere Clients pro Server 50
Mehrere RMI Server pro RMI Registry 51
3.3.5. Zusammenfassung der RMI Client Schnittstelle 54
4. JMX Browser API 56
4.1. Begriffsdefinition und Abgrenzung 56
4.2. Problemstellungen 56
4.2.1. Strukturierung der Information 56
4.2.2. Heterogenität der Implementierungen der verschiedenen Hersteller 56
4.2.3. Flexibilität, Erweiterbarkeit, Generizität 57
4.2.4. Darstellung komplexer Attribute 57
4.2.5. Auswahl der Attribute 57

4.2.6. Aufbereitung der Attributwerte 57
4.3. Lösungsansätze zur Strukturierung der MBeans 59
4.3.1. Strukturierung der Daten nach dem MBean Typ 59
4.3.2. Strukturierung der Daten in Baumform 61
4.3.3. Baumdarstellung bei Weblogic 8.x und 9.0 61
4.3.4. Baumdarstellung bei JSR-77 kompatiblen Applikations-Servern 63
4.3.5. Baumdarstellung nach Reihenfolge der Properties 65
4.4. Lösungsansätze zur Strukturierung der MBean Attribute 66
4.4.1. Abbildung komplexer Attribute nach JSR-77 66
4.4.2. Komplexe Attribute nach JSR-003: CompositeData 68
4.4.3. Komplexe Attribute nach JSR-003: TabularData 69
4.5. Lösungsansätze zur Flexibilität und Erweiterbarkeit 72
4.5.1. Auslagerung der Algorithmen auf den RMI Server 72
4.5.2. Abbildung der MBean Namen in einer herstellerunabhängigen Datenstruktur 74
4.5.3. Abbildung der MBean Attribute in einer herstellerunabhängigen Datenstruktur 76
4.6. Behandlung kumulierter Attributwerte 79
5. Konfigurationsdaten und Bootstrapping 83
5.1. Herstellerspezifisches Bootstrapping 83
5.1.1. Allgemeines 83
5.1.2. SUN JVM 1.5, 1.6 83
5.1.3. JBoss 4.0 84
5.1.4. BEA WebLogic Application Server 84
5.1.5. Oracle Application Server 85
5.1.6. IBM WebSphere Application Server 85
5.1.7. Borland Application Server 85
5.2. Konfigurationsdaten 87
5.2.1. Kriterien bei der Wahl der Konfigurationsdaten 87
5.2.2. Arten der Konfigurationsdaten 87
Host Name, Port 87
Protokoll 87
URL Postfix 88
MEJB Name 88

Sicherheitseinstellungen 88
Klassenpfad 89
Java Version 89
5.2.3. Überlegungen zur Konfiguration des Klassenpfades 89
5.2.4. Verwaltung der Standardeinstellungen 90
6. JMX API für C++ basierendes Monitoring 93
6.1. Problemsstellungen 93
6.1.1. Verwenden einer JVM in einem C++ Prozess 93
6.1.2. Abbildung von Java Datentypen in C++ 93
6.2. Lösungsansätze 93
6.2.1. Laden der JVM 93
6.2.2. Bereitstellen des JNIEnv Interface Zeigers 96
6.2.3. Konvertierung von Datentypen 96
Konvertierung von primitiven Datentypen 96
Konvertierung von Strings 96
Konvertierung von komplexen Datentypen 97
6.2.4. Zugriff von Java Methoden und Variablen 98
6.2.5. Caching von Java Referenzen 98
6.2.6. Fehlerbehandlung 99
6.3. C++ Schnittstellen Beschreibung JMX Monitoring API 100
6.3.1. Allgemeines 100
6.3.2. Einfache C++ Schnittstelle unter Verwendung eines Konfigurationsstrings 100
6.3.3. Beispiel eines Konfigurationsstrings 102
6.3.4. C++ Schnittstellenbeschreibung der JMX Monitoring API 102
7. Resümee 104
Historischer Rückblick auf JMX 104
Bootstrapping und Konfiguration 104
Interne Architektur 105
Strukturierte Darstellung 106
Brücke in die C++ Welt 106
8. Literatur 108
A. Anhang 111
WebSphere 6.0.xml 111

WebLogic 9.0.xml 112
WebLogic 9.0 JSR-160.xml 113
WebLogic 8.x.xml 114
SUN JVM 1.6.xml 115
SUN JVM 1.5.xml 116
JBoss 4.0.xml 117
dynaTrace Diagnostics 2.0.xml 118
dynaTrace Diagnostics 1.6.xml 119
BMC TM ART 3.0 Application.xml 120
BMC TM ART 3.0 Execution.xml 121
BMC TM ART 3.0 Frontend.xml 122
Borland Application Server 6.6.xml 123
Borland SC Test Manager 8.5 Application.xml 126
Borland SC Test Manager 8.5 Execution.xml 127
Borland SC Test Manager 8.5 Frontend.xml 128
Curriculum Vitae des Autors 129
Allgemeines 129
Ausbildung 129
Berufserfahrung 129
Eidesstattliche Erklärung 130

1. Einleitung

1.1. Vorgehensweise und Gliederung

1.1.1. Verwendung von Anglizismen

Im Rahmen dieser Arbeit wurde versucht, die Verwendung von Anglizismen auf ein Minimum zu beschränken. Da jedoch Programmiersprachen stark an die englische Sprache angelehnt sind und es von manchen Fachbegriffen nur wenig geläufige Übersetzungen in der deutschen Sprache gibt, lässt sich die Verwendung von Anglizismen nicht völlig vermeiden, ohne die Lesbarkeit dieser Arbeit zu beeinträchtigen. Fachbegriffe die in englischsprachigen Standards oder Fachliteratur definiert wurden, werden in dieser Arbeit generell nicht übersetzt.
Ebenso wird, falls es der Textfluss erfordert, des öfteren auf eine wortwörtliche Übersetzung von englischsprachigen Zitaten verzichtet.
Im Speziellen werden die weitgehend eingedeutschten Fachbegriffe Client, Server, Property und Monitoring anstelle sinngemäßer, deutscher Wörter verwendet.

1.1.2. Änderungen des Schriftbildes

Änderungen des Schriftbildes werden in Rahmen dieser Arbeit bewusst eingesetzt, um Folgendes auszudrücken:

Beispiel	**Beschreibung**	**Bedeutung**
AaBbCc123	Kursiv	• nicht übersetzte Fachbegriffe • Zitate
`AaBbCc123`	Schriftart Courier	• Klassen, Symbole und andere Bezeichner der Programmiersprachen Java oder C++ • Syntaxbeschreibungen in EBNF[1] • Beispiele aus dem Quellcode
AABBCC123	Großschreibung	• Abkürzungen von Fachbegriffen, die schon

[1] EBNF: Erweiterte Backus-Naur-Form

zuvor eingeführt wurden

1.1.3. Gliederung

Diese Arbeit ist in vier Hauptkapitel unterteilt. Kapitel 2 erklärt die theoretischen Grundlagen, die für das Verständnis der weiteren Kapitel notwendig sind. Kapitel 3 erklärt Probleme und Lösungen, die sich auf das Monitoring von Daten von verschiedenen Datenquellen via JMX beziehen. Kapitel 4 beschäftigt sich mit der Struktur der JMX Daten einer einzelnen Datenquelle und den damit verbundenen Darstellungsmöglichkeiten. Kapitel 5 erläutert die herstellerspezifischen Besonderheiten der JMX Datenquellen und die damit verbundene Vielzahl an Konfigurationen. Schließlich beschäftigt sich Kapitel 6 damit, wie die gewonnenen Daten in einer C++ Umgebung verfügbar gemacht werden können.

1.2. Problemstellung und Motivation

Die Idee zu dieser Arbeit ist im direkten Zusammenhang mit meinem beruflichen Umfeld entstanden. Ich arbeite seit 1998 für Segue Software [WIK02, Segue Software], einem renommierten Anbieter von Qualitätssicherungslösungen und -produkten rund um den Softwareentwicklungs- Lebenszyklus.

Zentrale Anwendungsbereiche der von Segue Software entwickelten Produkte sind funktionales Testen, Test und Requirements Management, sowie Loadtesting und Monitoring.

Im April 2006 wurde Segue Software von Borland Software Corporation [BOR01; WIK02, Borland Software Corporation] über die Börse aufgekauft. Die Produkte von Segue Software wurden gleichnamig in die Lifecycle Quality Management Sparte (LQM) von Borland Software Corp. übernommen.

Ein Anwendungsszenario aus dem Bereich des Loadtesting und Monitoring möchte ich nun kurz skizzieren.

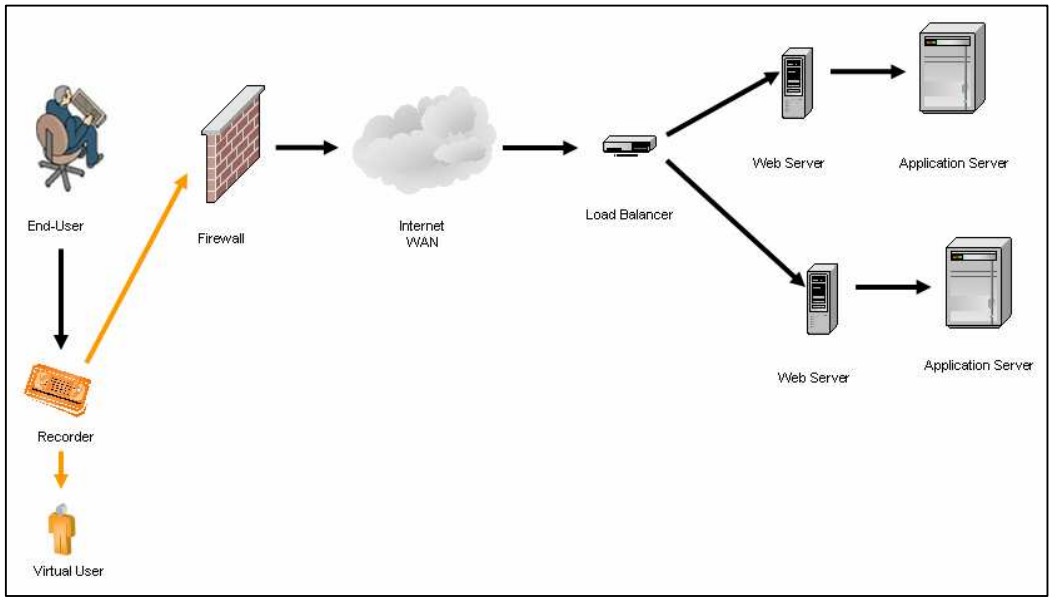

Abbildung 1 – Anwendungsszenario Loadtesting

Ein Konsument sitzt vor seinem Heim-PC und kauft, geschützt durch eine Firewall, im Internet bei einem großen Webshop ein. Der Betreiber des Webshops ist besorgt über die Performanz seines Systems und verwendet eine Loadtesting Lösung um die

Performanz seines Systems zu testen. Das zu testende System besteht aus mehreren Webservern, auf die über einen Load Balancer die Anfragen der Konsumenten verteilt werden. Die Webserver beziehen Daten aus mehreren Applikationsservern und Datenbanken.

In einem ersten Schritt wird der Internetverkehr einer einzelnen, abgeschlossenen Transaktion eines Konsumenten abgefangen und über ein Rekorder Programm, das als Proxy fungiert, umgeleitet. Das Rekorder Programm erkennt die Applikationslogik der Transaktion des Konsumenten und generiert ein Skript, dass das Verhalten des Konsumenten gegenüber dem Webshop repräsentiert. Man spricht von einem virtuellen Benutzer (*Virtual User*).

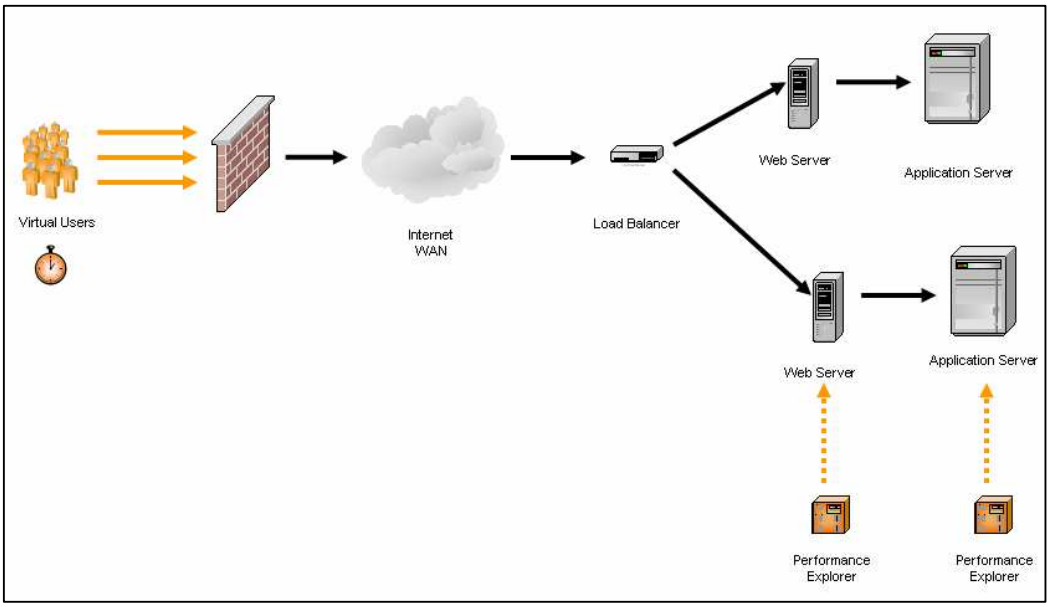

Abbildung 2 – Anwendungsszenario Loadtesting und Monitoring

Bei einem Loadtest wird nun eine beliebig große Anzahl von *Virtual Users* durch ein Programm simuliert. Für jeden *Virtual User* wird die gesamte Transaktionsdauer sowie die Antwortzeiten von speziellen Teilbereichen (z.B. Login, Suche, Checkout) gemessen. Zusätzlich wird die Gesundheit der Webserver und Applikationsserver durch ein Monitoring Tool, z.B. Borlands Silk Performance Explorer, gemessen. Ziel ist es, die auf der Client-Seite gemessenen Antwortzeiten mit den Daten über die Performanz des Webservers und Applikationsservers zu korrelieren, um somit Rückschlüsse auf Engstellen in der Architektur der Web-Anwendung ziehen zu können.

Für das Erfassen der Performanz und Gesundheit der Web-, Applikations- und des Datenbankserver gibt es eine große Anzahl von Möglichkeiten. Sehr weit verbreitet sind das Simple Network Management Protokoll (SNMP) und der Zugriff auf die Windows Performance Monitor Daten (PerfMon), letzteres allerdings nur auf Windows Systemen.

Der Fokus von SNMP und PerfMon Daten liegt oft auf der Hardware (CPU, Speicher, IO). Viele Server bieten auch noch herstellerspezifische Protokolle an, die tiefere Einblicke in die Applikation zulassen.

Die Java Management Extensions (JMX) ist ein neuer Standard, der den Zugriff auf Performanz- und Konfigurationsdaten von Java Applikationen ermöglicht. Die Idee dieser Arbeit besteht darin, mit Hilfe von JMX das Monitoring von spezifischen Informationen über den Zustand einer Java Applikation zu ermöglichen.

In Kapitel 1.4 folgt nun ein ausführliches Anwendungsbeispiel zur Verwendung von Borland Silk Performance Explorer, dass den Inhalt und die Problemstellungen dieser Arbeit verdeutlichen soll. Zuvor werden noch in Kapitel 1.3 kurz die wichtigsten Begriffe, die zum Verständnis von JMX unbedingt notwendig sind, erklärt.

1.3. Begriffe

1.3.1. Java Management Extensions (JMX)

JMX entstand 1998 als Java Specification Request 3 (JSR-003), damals jedoch noch unter dem Namen Java Management API 2.0. Mittlerweile tragen auch die Spezifikationen in JSR-160 und JSR-77 wesentlich zum Begriff JMX bei.
Der JSR-003 Standard schreibt zum Begriff JMX: „*The Java™ Management extensions (also called the JMX™ specification) define an architecture, the design patterns, the APIs, and the services for application and network management in the Java programming language*“ [SUN01, S.17]. Das bedeutet übersetzt, dass JMX eine Architektur, das Design, die Schnittstellen und die Dienste für Applikations- und Netzwerkmanagement in der Programmiersprache Java definiert.
Oder anders formuliert, unter JMX versteht man die Schnittstelle zwischen Ressourcen und Management Systemen in der Java Welt. So schreibt Kreger: „*JMX is an isolation and mediation layer between manageable resources and management systems*“ [KRE03, S. 3f].

1.3.2. MBean

MBeans sind Java Objekte, die die Namens- und Vererbungskonventionen des JMX Standards JSR-003 erfüllen und Zugriff auf eine durch Attribute beschriebene Ressource ermöglichen. Weiters ist es möglich Operationen auf dem *MBean* auszuführen, welche die Ressource beeinflussen. Nach Art der Schnittstelle, die das *MBean* beschreibt, unterscheidet man *Standard MBeans*, *Dynamic MBeans*, *Model MBeans* und *Open MBeans*.

1.3.3. MBean Server

Der *MBean Server* ist ein Java Objekt, das als Verzeichnisdienst für eine Gruppe von MBeans dient [vgl. SUL03, S.12]. Über den *MBean Server* kann nach *MBeans* gesucht werden, und es kann auf Attribute und Methoden von *MBeans* zugegriffen werden.

1.3.4. ObjectName

Der *ObjectName* ist der eindeutige Bezeichner eines *MBeans*. Mittels des *ObjectName* werden *MBeans* auf dem *MBean Server* referenziert oder Suchabfragen abgesetzt. Der *ObjectName* besteht aus einem Bezeichner für die *Domain* und einer Liste von Name/Wert Paaren.

1.3.5. JMX Agent

Der JMX Agent dient als Kontainer für einen *MBean Server*. Der JMX Agent definiert üblicherweise Konnektoren, über die von Remote Clients auf den MBean Server zugegriffen werden kann. Typische Konnektoren basieren auf Remote Method Invocation (RMI) oder auf dem Java Naming and Directory Interface (JNDI). Weiters kann ein JMX Agent Adaptoren beinhalten, die den Zugriff auf JMX Daten über JMX fremde Protokolle wie dem Simple Network Management Protocol (SNMP) oder dem Hypertext Transfer Protocol (HTTP) ermöglichen.

1.3.6. JSR-003

JSR-003 ist die Abkürzung für Java Specification Request 003 und ist der Standard, der die Grundlage von JMX darstellt. Er definiert grundlegende Begriffe wie *MBean*, Attribute, *MBean Server* und JMX Agent.

1.3.7. JSR-77

JSR-77 ist die Abkürzung für Java Specification Request 77 und definiert einen Standard zur Verwaltung von Java Plattform, Enterprise Edition (J2EE) kompatiblen Produkten [siehe WIK01, J2EE]. Der Standard definiert, welche Daten via JMX bereitgestellt werden sollen. Weiters definiert der Standard ein Management Enterprise Java Bean (MEJB), das einen einfach Zugriff auf die MBeans ermöglicht.

1.3.8. JSR-160

Der JSR-160 Standard definiert unter anderem verschiedene Konnektoren, die einen Remote Zugriff auf einen *MBean Server* ermöglichen. JSR-160 dient als Ergänzung zum JSR-003 und wird oft in eigenen Bibliotheken implementiert. Der Zugriff auf den *MBean Server* erfolgt je nach Protokoll über einen eigenen *Connector Server*, welcher über eine *JMXServiceURL* adressiert wird.

1.3.9. JMXServiceURL

Die JMXServiceURL definiert die Adresse eines *Connector Servers*, welcher wiederum einen Remote Zugriff auf einen *MBean Server* ermöglicht. Die Adresse entspricht der Syntax einer Abstract Service URL nach dem RFC 2609 Standard [siehe RFC01] für das Service Location Protocol (SLP) [vgl. SUN03, S. 62].
Die konkrete Syntax für das JMX Service lautet:

```
<JMXServiceURL> = „service:jmx:“ <protocol> „:“ <sap> ;
```

Dabei ist *protocol* der Platzhalter für das verwendete Transportprotokoll, also überlicherweise RMI oder IIOP.
Der Platzhalter *sap* definiert die Adresse des *Connector Servers*. Die Adresse des *Connector Servers* kann entweder direkt angegeben werden, oder es wird eine Referenz auf ein *Lookup Service*, z.B. JNDI, verwendet, das die tatsächliche Adresse des *Connector Servers* liefert.

1.3.10. Native Code

Unter Native Code versteht man in der Softwareentwicklung[2] betriebssystemspezifischen Quellcode, zumeist in den Programmiersprachen C und C++.

[2] Im Zusammenhang von Übersetzerbau versteht man unter Native Code auch Maschinencode.

1.3.11. Java Native Interface (JNI)

Das Java Native Interface ist eine Schnittstelle der Programmiersprache Java, die es erlaubt, C/C++ Code aus einer *Java Virtual Machine* aufzurufen und umgekehrt, eine *Java Virtual Machine* in C/C++ zu starten und Java Code in C/C++ auszuführen. So schreibt Liang: „*Applications that use the JNI can incorporate* `native code` *written in programming languages such C and C++, as well as code written in the Java programming language*“ [LIA99, S.3].

1.3.12. Java Naming and Directory Interface (JNDI)

Das Java Naming and Directory Interface ist eine Programmierschnittstelle und Teil der Programmiersprache Java, der sich mit dem Zugriff auf Namensdienste und Verzeichnisdienste[3] beschäftigt und eine einheitliche Schnittstelle für den Zugriff auf verschiedene Verzeichnisdienste bietet. So definiert die JNDI Dokumentation von SUN: „*Java Naming and Directory Interface™ (JNDI) is an API that provides directory and naming functionality to Java applications. It is defined to be independent of any specific directory service implementation. Thus, a variety of directories can be accessed in a common way*“ [SUN04, S. 1].

1.3.13. Remote Method Invocation (RMI)

Remote Method Invocation (RMI) ist ein *Remote Procedure Call* (RPC) Mechanismus der Programmiersprache Java zum Ausführen von Methoden entfernter Java Objekte. Unter entfernten Java Objekten werden Objekte verstanden, die in einer anderen *Java Virtual Machine* existieren [vgl. WIK01, Remote Method Invocation].
Der RMI Client ruft dabei sogenannte *Stub* Klassen auf, die die Netzwerkkommunikation zum RMI Server übernehmen. Die Verbindungsaufnahme des RMI Client zum RMI Server erfolgt über die RMI *Registry*, einem Verzeichnisdienst in dem alle Objekte, die für den entfernten Zugriff zur Verfügung stehen, registriert sind.

[3] engl.: Naming and Directory Services

1.4. Anwendungsbeispiel aus Benutzersicht

Die im Rahmen dieser Arbeit entwickelte API findet in dem Produkt Borland Silk Performance Explorer praktische Anwendung. Um die Problemstellungen und den Inhalt dieser Arbeit zu verdeutlichen, folgt nun ein Anwendungsbeispiel aus der Sicht eines Benutzers dieses Produkts.

Borland Silk Performance Explorer ermöglicht die Überwachung und Analyse von server- und clientseitigen Performanz-Daten von Server Systemen. Ein typisches Anwendungsszenario ist, wie im einleitenden Kapitel 1.2 beschrieben, das Messen der Performanz-Daten eines Servers während eines Lasttests.

Der erste Schritt ist die Auswahl der Art der Datenquelle. Typische Arten von Datenquellen sind SNMP, Windows Performance Monitor Daten, Rexec für die entfernte Abfrage der Daten von Unix Systemen und JMX für Daten von Java basierten Systemen.

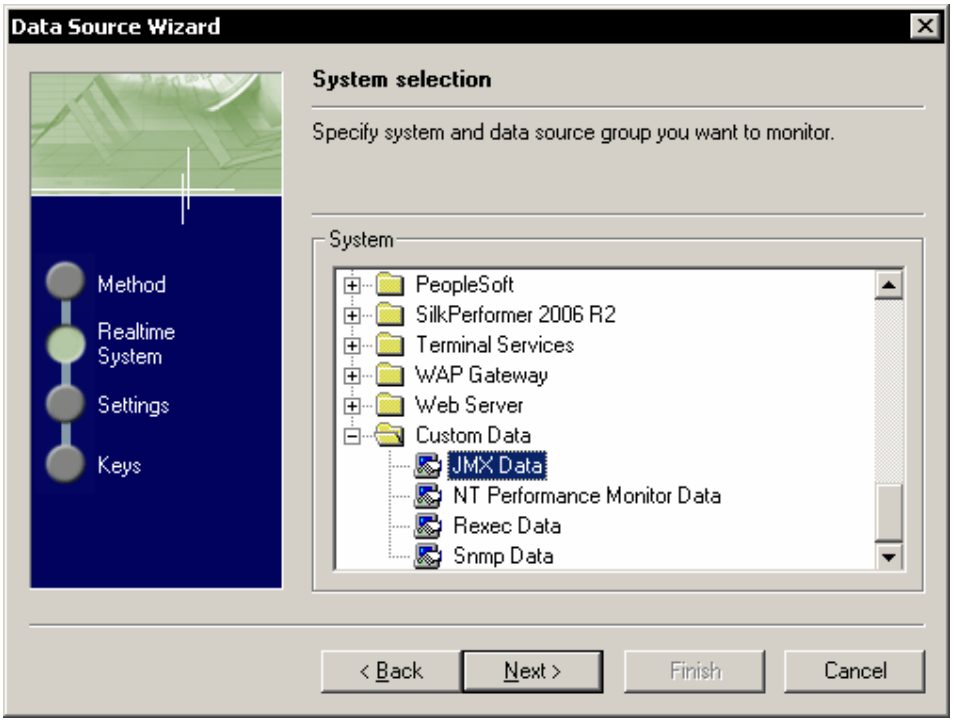

Abbildung 3 - Auswahl der Art der Datenquelle

Für weit verbreitete Server Systeme wie SAP, Oracle oder WebLogic gibt es auch vorkonfigurierte Datenquellen, die dem Benutzer Hilfestellung bieten sollen. Eine Auswahl von vorkonfigurierten Datenquellen aus dem Bereich des Application Server Monitoring zeigt Abbildung 4. Die Art der Datenquelle, also ob zum Beispiel SNMP

oder JMX für das Monitoring verwendet werden soll, ist dabei durch die Namensgebung der Datenquelle oft schon implizit vorweggenommen.

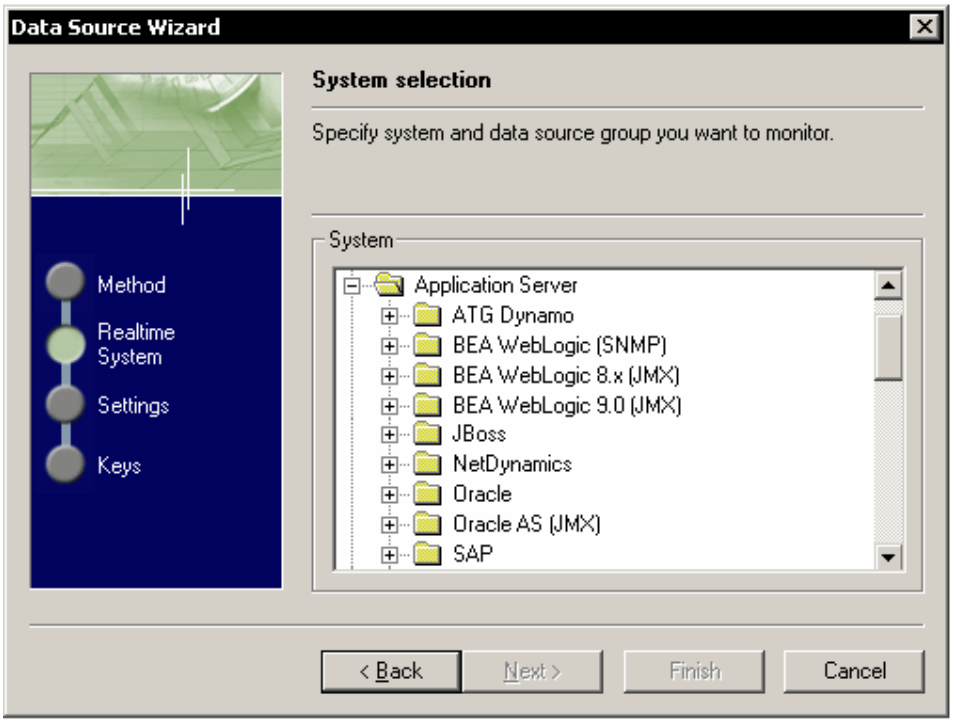

Abbildung 4 - Vorkonfigurierte Datenquellen

Im nächsten Schritt werden die Verbindungsparameter wie Host Name, Port, Benutzername und Passwort abgefragt. Abbildung 5 zeigt den Dialog zur Konfiguration der Verbindungsparameter. In diesem Beispiel verbindet sich der Benutzer Johannes auf den Rechner atlid-jhoelzl mit einer *Java Virtual Machine* von SUN, die JMX Performanz Daten auf dem Port 9997 anbietet.

Details zu den Verbindungsparametern sind durch eine Konfigurationsdatei voreingestellt und für den Benutzer auf den ersten Blick verborgen. Die Schaltfläche „Server Configuration ...“ öffnet einen Optionendialog, in dem weitere Einstellungen vorgenommen werden können.

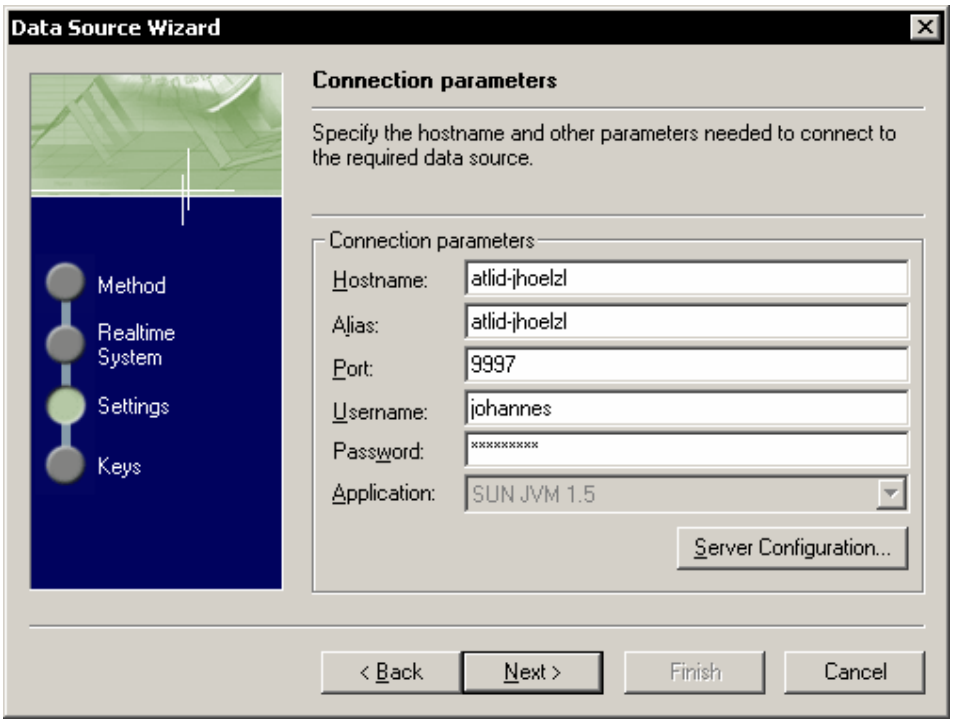

Abbildung 5 - Konfiguration der Verbindungsparameter

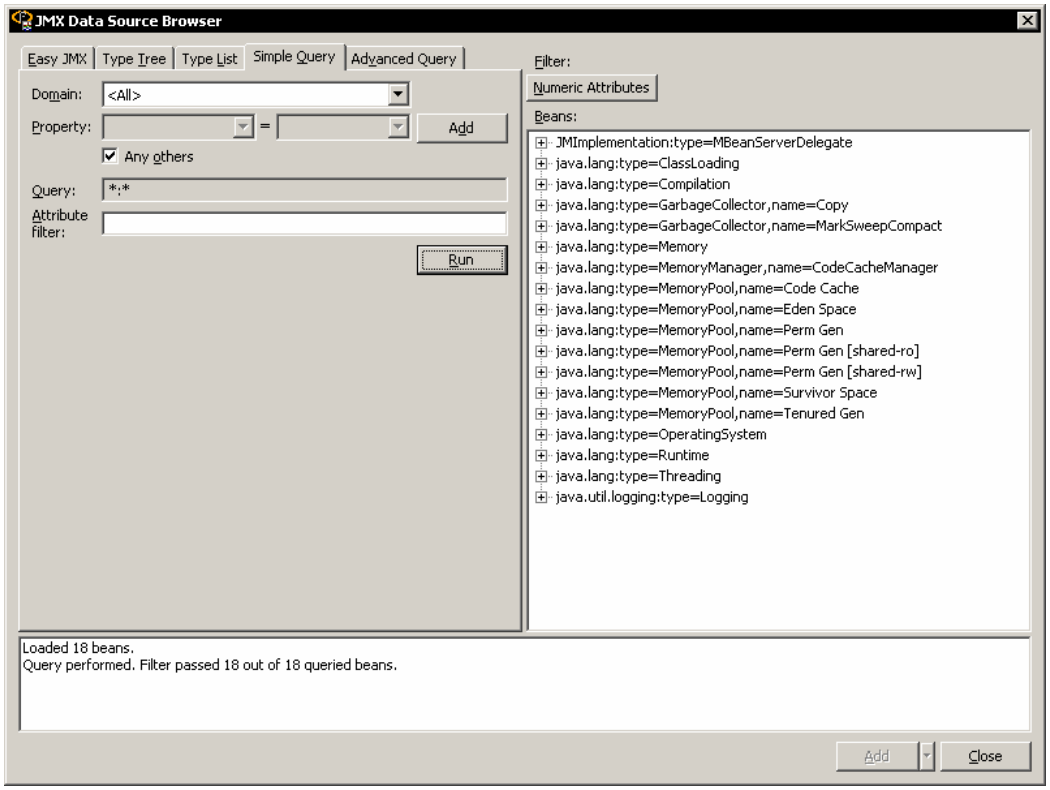

Abbildung 6 - Verfügbare MBeans

Im folgenden Schritt wird der JMX Data Source Browser geöffnet. Abbildung 6 zeigt alle verfügbaren *MBeans*, die im rechten Teil der Abbildung, nach dem *ObjectName* sortiert, aufgelistet sind. Im linken Teil der Abbildung sieht man die Suchabfrage, die zum Auflisten der *MBeans* verwendet wurde. Die Abfrage aller *MBeans* wird hier durch die Zeichenkette „*:*“ ausgedrückt.

Die *Java Virtual Machine* von SUN offeriert nur eine sehr geringe Anzahl von *MBeans*, in diesem Beispiel 18, daher liefert eine einfache Suchabfrage nach allen verfügbaren *MBeans* ein überschaubares Ergebnis.

Applikationsserver wie zum Beispiel BEA WebLogic 9.0 liefern schon alleine für die vorinstallierten Beispielanwendungen mehrere hundert *MBeans*. Da jedes *MBean* mit einem eindeutigen *ObjectName* registriert sein muss, werden mit zunehmender Anzahl von *MBeans* auch deren Bezeichner länger und unüberschaubarer. In Abbildung 7 ist zu sehen, dass der Name des selektierten *MBeans* eines WebLogic Servers bereits die Fensterbreite überragt. Eine Suche in allen verfügbaren *MBeans* ist somit nicht unbedingt zielführend.

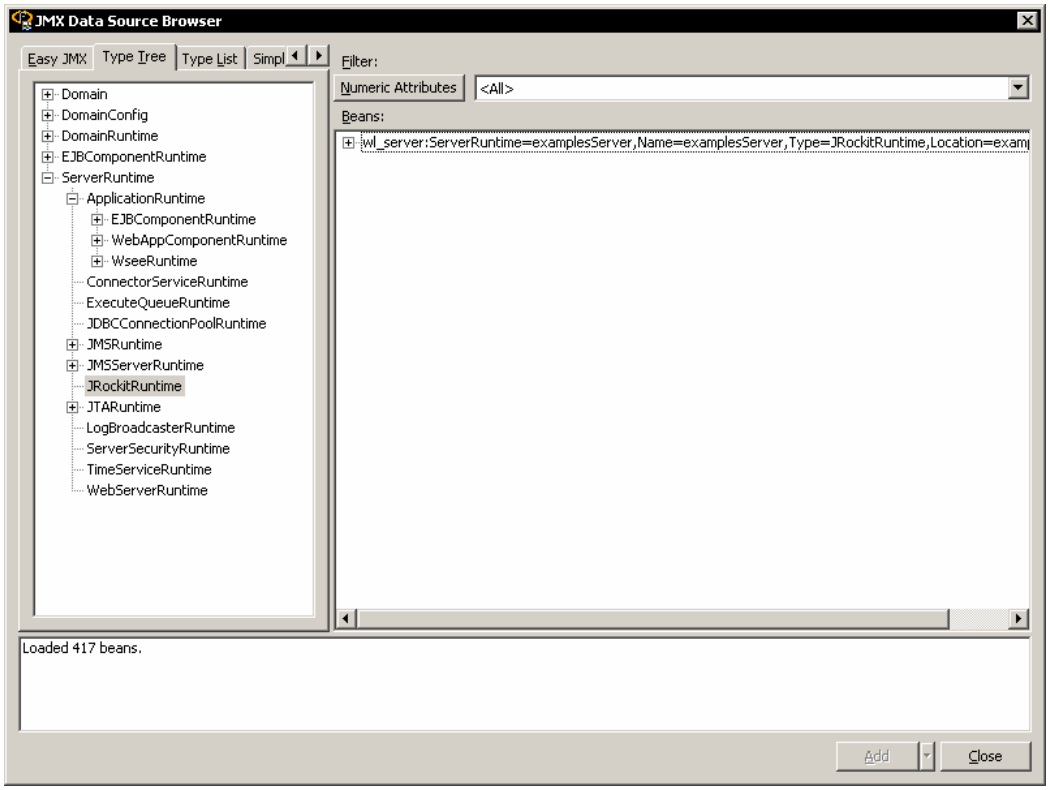

Abbildung 7 - *MBeans* strukturiert nach *MBean* Typ in Baumform

Der JMX Data Source Browser bietet für dieses Problem Hilfestellung durch alternative Darstellungen der *MBeans*, zum Beispiel wie in der linken Bildhälfte von

Abbildung 7 zu sehen, mittels Strukturierung nach dem *MBean* Typ in Baumform. Dies bedeutet, dass die verfügbaren *MBeans* nach Typen kategorisiert und diese *MBean* Typen nach verschiedenen Algorithmen wiederum in Hierarchie zueinander gesetzt werden. Weiters werden erweiterte Such und Filtermöglichkeiten geboten.
Doch zurück zu den überschaubaren 18 *MBeans* einer *Java Virtual Machine*.

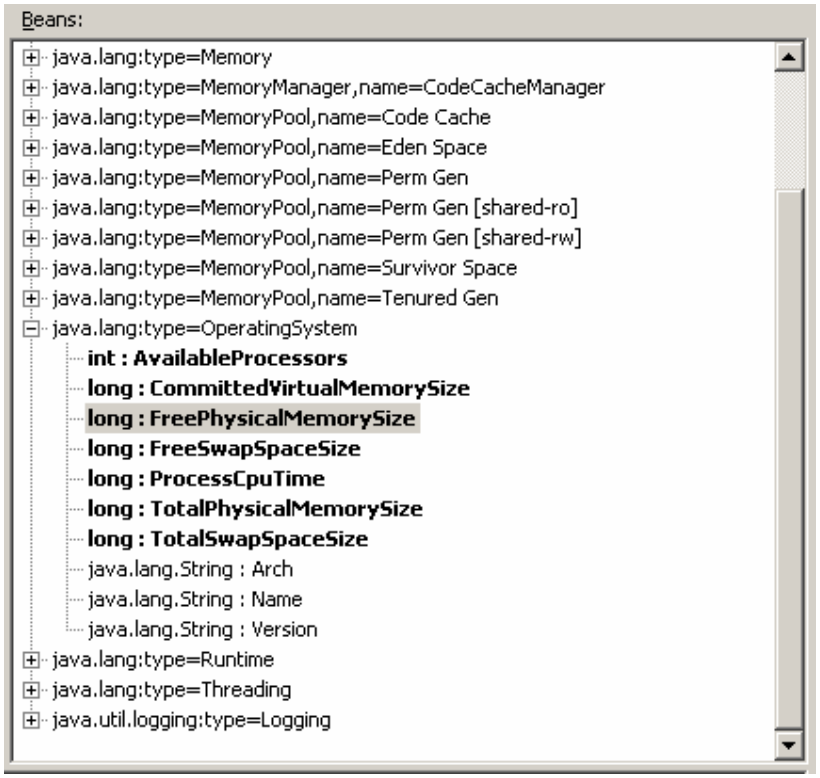

Abbildung 8 - MBean mit Attributen

Abbildung 8 zeigt in einem Teilauschnitt von Abbildung 6, dass beim Öffnen eines *MBean* Knotens automatisch alle Attribute des *MBeans* abgefragt und angezeigt werden. So bietet das MBean `java.lang:type=OperatingSystem` interessante Kennzahlen wie CPU Auslastung (`ProcessCpuTime`) und freier Arbeitsspeicher (`FreePhysicalMemorySize`). Das *MBean* in diesem Beispiel liefert nur primitive Datentypen (int, long, String). Bei komplexen Attributen können auch mehrstufige Hierarchien aufgebaut werden.
So zeigt Abbildung 9 das *MBean* `java.lang:type:Memory`, das neben zwei primitiven Attributen auch zwei komplexe Attribute, `HeapMemoryUsage` und `NonHeapMemoryUsage` enthält, die jeweils aus vier Komponenten bestehen.

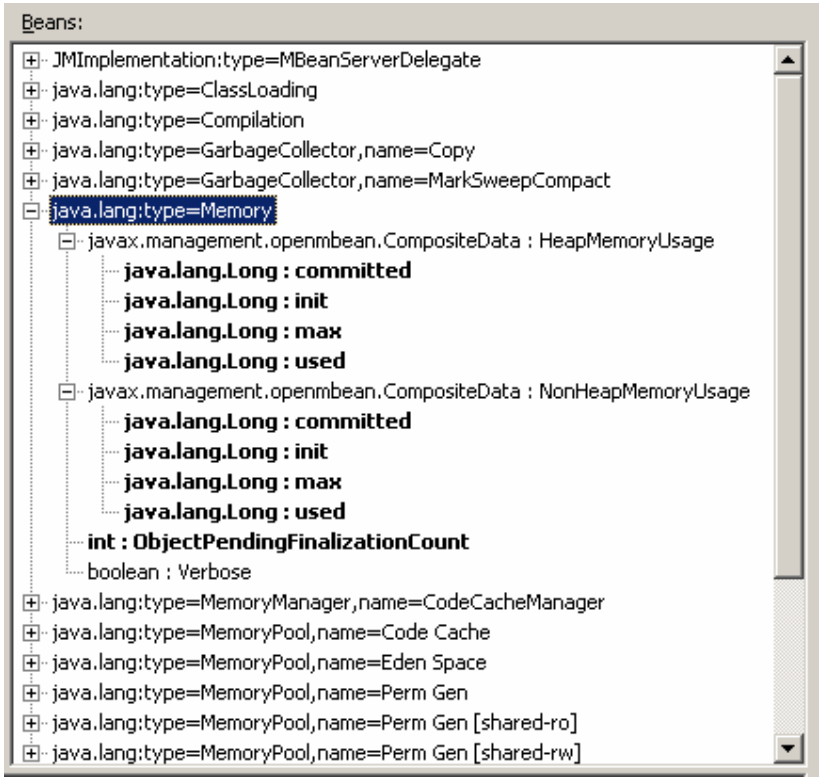

Abbildung 9 - MBean mit komplexen Attributen

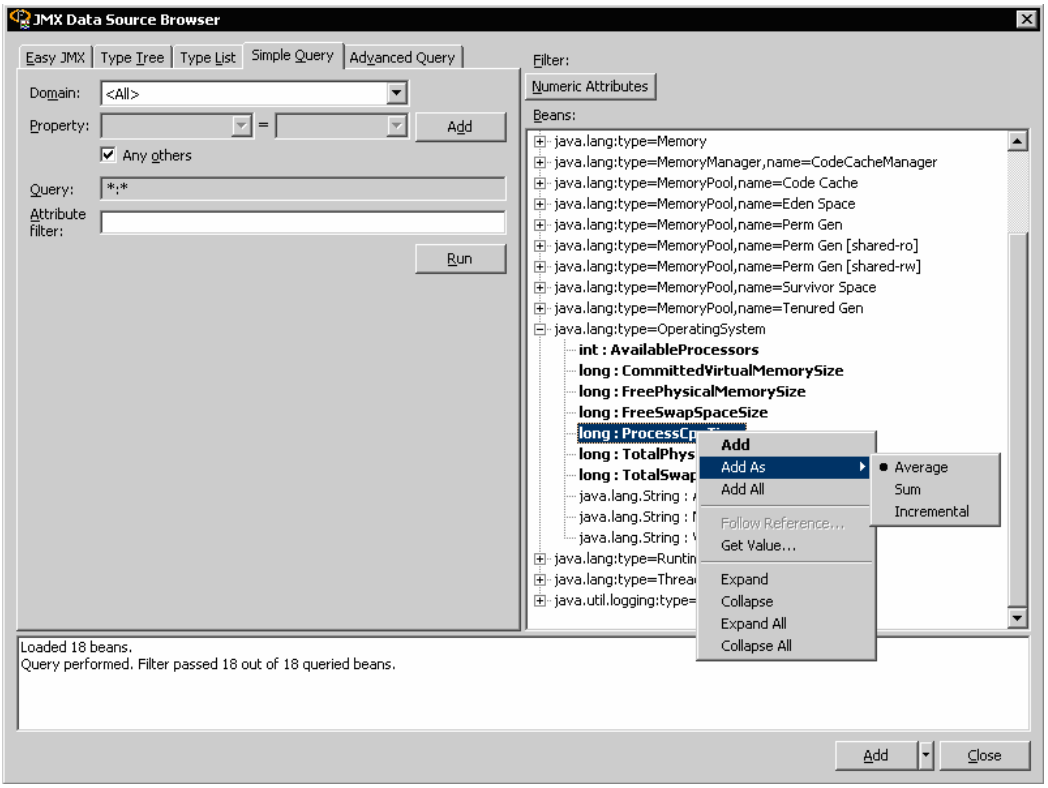

Abbildung 10 - Attribut Auswahl

Als letzter Schritt werden über das Kontext Menü oder die Schaltfläche „Add“ die für den Benutzer interessanten Attribute ausgewählt und der Data Source Wizard geschlossen. Man beachte dabei in Abbildung 10 die notwendige Unterscheidung ob die Messdaten als kumulative (Incremental), Durchschnitts- (Average) oder Summenwerte (Sum) vorliegen. Silk Performance Explorer beginnt mit dem Monitoring der ausgewählten *MBean* Attribute (siehe Abbildung 11).

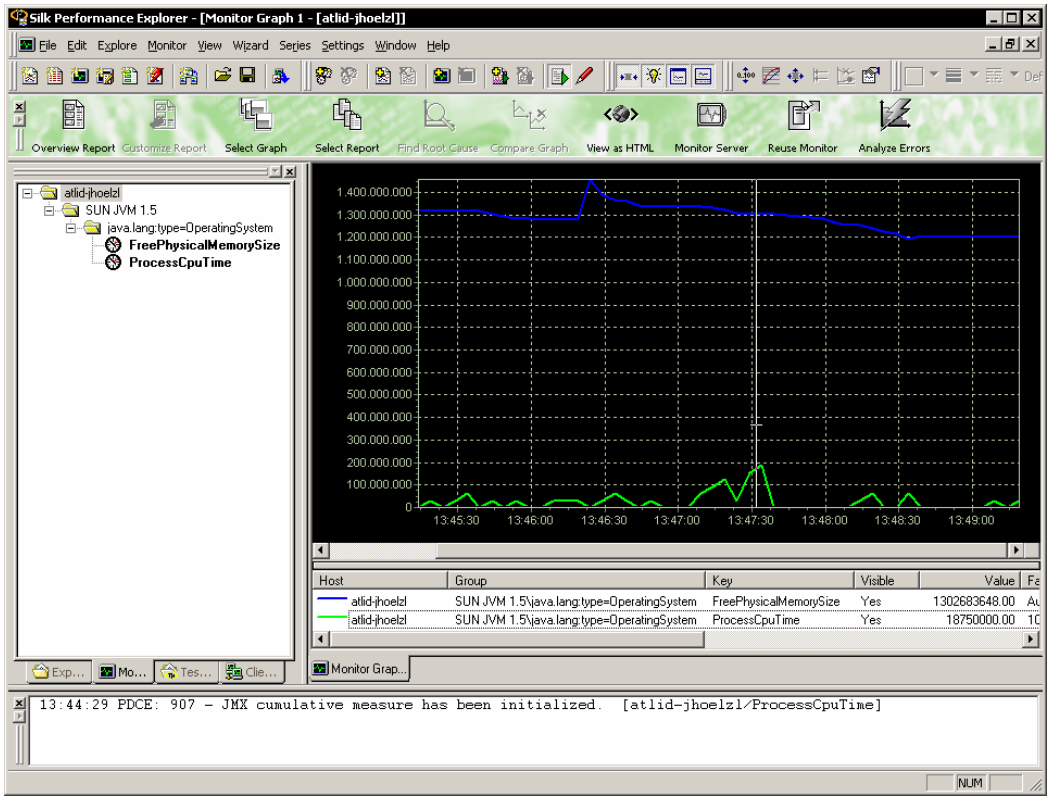

Abbildung 11 - Monitor Graph

1.5. Ziele und Forschungsfragen

Ziel dieser Arbeit ist es, eine Bibliothek für einen JMX Browser und Monitoring Client zu definieren und zu implementieren, die die aktuellen Versionen der marktführenden Java Applikationsserver, sowie auf Java 2 Plattform 5.0 basierenden Anwendungen unterstützt. Die Bibliothek muss einfache Schnittstellen für die Verwendung in Java und C++ basierenden Produkten aufweisen. Weiters soll die Erweiterbarkeit hinsichtlich zukünftiger Produkt Versionen und Anbieter gegeben sein.

Eine wesentliche Anforderung an den JMX Monitoring Client ist, dass dieser Datenquellen verschiedener Hersteller gleichzeitig abfragen kann.

Eine zentrale Frage bei der Entwicklung des JMX Browsers ist, wie man die vorhandenen Daten in geeigneter strukturierter Form darstellen kann, so dass die Darstellung für Monitoring Datenquellen verschiedenster Produkte einheitlich und schlüssig ist. Dabei muss man besonders berücksichtigen, dass Hersteller oft proprietäre Methoden verwenden um die Datenquellen zu implementieren.

2. Grundlagen

2.1. Grundbegriffe des JNDI

2.1.1. Binding, Bindname

Unter *Binding*[4] versteht man in Zusammenhang von JNDI das Referenzieren eines Objektes im Verzeichnis durch einen Bezeichner, dem so genannten *Bindname*. Man unterscheidet dabei zusammengesetzte Bezeichner (*compound names*) und unteilbare Bezeichner (*atomic names*).

2.1.2. Context

In der JNDI Programmierschnittstelle gibt es keine absolute Namensgebung für die Objekte des Verzeichnisses. Die Namen werden relativ zu einem *Context* Objekt vergeben. Nur das *Context* Objekt selbst besteht aus einer Menge von *Bindings* zu atomaren Bezeichnern, die anderen Objekte werden relativ adressiert.
Das *Context* Objekt bietet Methoden, um neue Verknüpfungen zwischen Objekten im Verzeichnis und Bezeichnern zu erzeugen (*Bind* Methoden), wieder zu löschen (*Unbind* Methoden) [vgl. SUN04, S. 5] oder aufzulisten. Weiters bietet es eine Methode, die das von einem *Bindname* referenzierte Objekt retourniert, diese bezeichnet man als *Lookup* Methode.

2.1.3. Initial Context

Ein *Context* Objekt kann wiederum untergeordnete *Context* Objekte, mittels einem atomaren Bezeichner, referenzieren. Man spricht von einem *Subcontext*. Der *Initial Context* stellt für einen Client den Startpunkt für die Namensauflösung dar. So beschreibt die JNDI Dokumentation von SUN: „*The initial context contains a variety of bindings that hook up the client to useful and shared contexts from one or more naming systems, such as the namespace of URLs or the root of DNS*“ [SUN04, S.10]. Das bedeutet, dass die Implementierung des *Initial Context* dafür sorgt, dass der

[4] Deutsch: Verknüpfung, Verbindung

Client auf den *Context* von bekannten Namensdiensten[5] wie URL (Uniform Ressource Locator, [RFC02]) oder DNS (Domain Name Service, [RFC 03]) aufbauen kann.

Oft wird das Entwurfmuster einer *Factory* Klasse verwendet, um das *Context* Objekt für eine Anwendung zur Verfügung zu stellen. Man spricht von der *Initial Context Factory*.

[5] Engl. Naming Systems

2.2. Die JMX Architektur

2.2.1. Allgemeine JMX Architektur

Manageable Ressource

Der Kernpunkt einer allgemeinen JMX Architektur ist eine *Manageable Ressource*. Eine *Manageable Ressource* kann eine beliebige Komponente aus den Bereichen Hardware-, Netzwerk- oder Applikationsmanagement darstellen. Dies gilt sowohl für die Konfiguration als auch das Überwachen von Ressourcen. In der Java Welt stellen unter anderem *Java Virtual Machines* (JVMs), Servlets oder Enterprise Java Beans (EJBs) typische Beispiele von *Manageable Ressources* dar [vgl. KRE03, S. 9].

MBean

Ein *MBean* dient zur Verwaltung einer solchen *Manageable Ressource*. Ein *MBean* implementiert das *Management Interface* einer *Manageable Ressource*. Das *Management Interface* definiert Attribute, die ausgelesen oder gesetzt werden können. Weiters definiert es Konstruktoren zum Erzeugen des *MBeans* und Operationen, die auf einem *MBean* ausgeführt werden können.

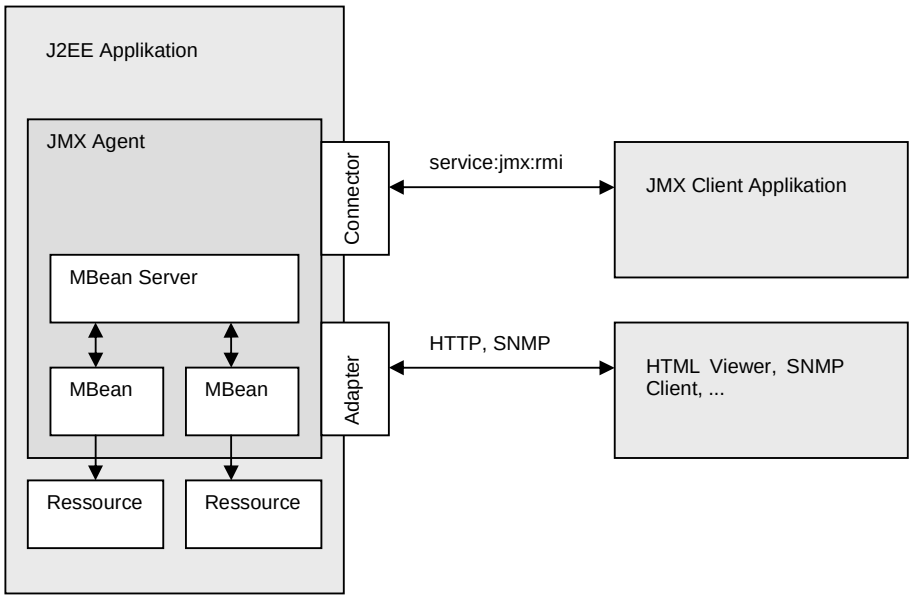

Abbildung 12 - JMX Architektur

Nach dem JSR-003 Standard unterscheidet man 4 Arten von *MBeans*: *Standard MBeans*, *Dynamic MBeans*, *Model MBeans* und *Open MBeans* [vgl. SUN01, S. 26].
Bei *Standard MBeans* definieren *Getter* und *Setter* Methoden den Zugriff auf die Attribute der *Manageable Ressource*. Alle anderen *Public* Methoden des *Management Interface* definieren die Operationen, die auf dem *MBean* ausgeführt werden können [vgl. SUN01, S.36]. Der JSR-003 Standard sieht vor, dass der Name der *Management Interface* Klasse mit dem String „`MBean`“ enden muss.
Bei modernen J2EE Applikationen stellen die *Standard MBeans* sogar einen direkten Bestandteil der Applikation selbst dar. Es muss nicht mehr ein *MBean* implementiert werden, um eine Ressource zu verwalten, sondern jedes relevante Java Objekt implementiert selbst das *MBean* Interface. Der JMX Standard dient somit als Entwurfsmuster bei der Entwicklung von Java Anwendungen. Sullins und Whipple empfehlen: *„If you plan to use an MBean to expose part of a new application in development, you should use a Standard MBean. The MBean is simple to develop, and you can create it concurrently with your application“* [SUL03, S. 67]. Das bedeutet, dass *Standard MBeans* auf einfache Art und Weise bei der Entwicklung neuer Anwendungen eingesetzt werden können.

Im Gegensatz zu Standard MBeans, die bei Sullins und Whipple wie folgt charakterisiert werden: *„... Standard MBeans are perfect for managing new ressources or ressources with well-known, static interfaces“* [SUL03, S. 96], werden bei *Dynamic MBeans* die Informationen über Attribute und Operationen dynamisch zur Laufzeit geliefert. Anstelle einer eigenen *Management Interface* Klasse wird die Interface Klasse `javax.management.DynamicMBean` implementiert. Dies ist von Vorteil, wenn bestehende *Manageable Ressources*, die sich eventuell nicht an die Namenskonvention eines *Standard MBean* halten, instrumentiert werden sollen. [vgl. SUN01, S. 40] Falls sich die *Manageable Ressource* ändert, ist bei *Dynamic MBeans* allerdings erhöhter Programmieraufwand notwendig, um die Attributinformationen und die Information über Operatoren und deren Parameter zu aktualisieren.

Model MBeans sind die völlig generische Variante von *Dynamic MBeans*. Der JMX Agent offeriert die Klasse `javax.management.modelmbean.RequiredModelMBean`. Anhand einer solchen Klasse kann ein JMX Entwickler auf einfache Weise seine Ressourcen instrumentieren [vgl. SUN01, S. 69]. Das `RequiredModelMBean` Interface erweitert das `DynamicMBean` Interface, wird jedoch erst zur Laufzeit dem jeweiligen *MBean* zugewiesen und wird somit nach Sullins und Whipple im Gegensatz zu *Dynamic MBeans* außerhalb des *MBeans* definiert: „*However, unlike usual Dynamic MBeans, the Model MBean's management interface is defined outside the MBean (by a management application or ressource) and inserted into the MBean via a setter method*“ [SUL03, S. 140].

Open MBeans sind *Dynamic MBeans* mit dem Vorteil, dass die verwendeten Datentypen im *Management Interface* über generische Typbeschreibungen aus einem universellen Set von *Basic Data Types* dargestellt werden. Eine Management Applikation erhält somit vollen Zugriff auf alle Attribute und Operationen, ohne dass herstellerspezifische Klassen benötigt werden [vgl. SUN01, S. 60].

Das Set von *Basic Data Types* für *Open MBeans* besteht aus einer Reihe von Wrapper Klassen für die primitiven Datentypen. Zusätzlich wird noch die *ObjectName* Klasse unterstützt. Weiters erlauben *Open MBeans* die Verwendung der beiden Interface Klassen `javax.management.openbean.CompositeData` und `javax.management.openbean.TabularData`. Kreger beschreibt: „*The aggregate types allow the representation of complex data types. Aggregate data types may contain only other basic data types, including other aggregate types*“ [KRE03 S.

127]. Diese Interface Klassen dienen also zur Bildung von aggregierten Datentypen, welchen wiederum *Basic Data Types*, also primitive Datentypen oder aggregierte Datentypen zu Grunde liegen.

ObjectName

Der *ObjectName* ist ein String zur eindeutigen Identifizierung von *MBeans*. Er besteht aus dem *Domain Name* und einer ungeordneten Liste von *Key Properties*. Die *Domain* ist ein Bezeichner, der zur Strukturierung der *MBeans* dient. So definiert der JSR-003 Standard: *„The domain name is a case-sensitive string. It provides a structure for the naming space within a JMX agent or within a global management solution“* [SUN01, S. 106].

Die *Key Properties* Liste ist eine Liste von Name/Wert[6] Paaren, die beliebig lang werden kann, aber mindestens die Länge eins haben muss. Ein *ObjectName* muss sich durch mindestens einen Wert eines *Key Property* von anderen *ObjectNames* derselben *Domain* unterscheiden.

Der *Domain Name* wird durch das ‚:' Symbol von den *Key Properties* getrennt. Die einzelnen Key Properties werden durch das ‚,' Symbol getrennt. Somit ergibt sich nach dem JSR-003 Standard folgende Syntax für einen *ObjectName* [vgl. SUN01, S. 106]:

```
<ObjectName> = [<domainName>] “:” <keyProp> [“,” <keyProp>]* ;
<keyProp> = <property> “=” <value> ;
```

MBean Server

Eine Gruppe von *MBeans* wird durch einen *MBean Server* verwaltet. Die einzelnen *MBeans* werden über ihren *ObjectName* am *MBean Server* registriert. Kreger definiert dies als den Hauptzweck des *MBean Servers*: *„The MBeanServer's primary responsibility is to provide a registry, with a common naming model …“* [KRE03, S. 237].

Nur *MBeans*, die auf dem *MBean Server* registriert sind, sind für Management Applikationen sichtbar. Sichtbar ist jedoch nicht das Objekt selbst, sondern nur sein *Management Interface*. So definiert JSR-003: *„Any object registered with the MBean server becomes visible to management applications. However, the MBean server only*

[6] engl.: property/value

exposes the an MBean's management interface, never its direct object reference" [SUN01, S. 27].

Die Struktur der *MBeans* auf dem *MBean Server* ist flach, d.h. der *MBean Server* kennt keine hierarchischen Beziehungen zwischen *MBeans*. Die einzige logische Gruppierung von *MBeans*, die von der JMX Spezifikation vorgesehen ist, erfolgt über einen bestimmten Teil des *ObjectName*, dem *Domain Name*.

Die Schnittstellenklasse `javax.management.MBeanServer` definiert unter anderem folgende Arten von Operationen:

- Instanzieren, Registrieren und Deregistrieren von *MBeans*
- Abfrage und Setzen von *MBean* Attributen
- Aufruf von *MBean* Methoden
- Abfrage der Anzahl von *MBean* auf dem Server
- Abfrage aller *Domains* auf dem Server, Abfrage der *Default Domain*[7]
- Hinzufügen und Entfernen von *Notification Listeners*
- Abfrage von *MBeans*

Um auf *MBeans* des *MBean Servers* mit einer der aufgelisteten Operationen zuzugreifen, muss der exakte *ObjectName* des *MBeans* angegeben werden oder über einen eigenen Query Mechanismus (siehe auch Kapitel 2.3) nach passenden *MBeans* gesucht werden.

Der *MBean Server* verschickt Notifizierungen über Ereignisse wie das Registrieren oder Deregistrieren von *MBeans*. So definiert der JSR-003: *„The MBean server will always emit notifications when MBeans are registered or deregistered"* [SUN01, S. 121]. Monitoring Clients, die an diesen Ereignissen interessiert sind, müssen die Schnittstellendefinition *Notification Listener* implementieren, um vom *MBean Server* automatisch über solche Ereignisse informiert zu werden.

[7] Die *Default Domain* ist ein String, der beim Registrieren von *MBeans* auf dem *MBeanServer* verwendet wird, wenn kein *Domain* Name explizit angegeben wurde.

JMX Agent

Der JMX Agent ist der Container für den *MBean Server* und dessen *MBeans* (siehe Abbildung 12). Er definiert eine Menge von Konnektoren, die es JMX Clients erlaubt, auch außerhalb der JVM des JMX Agent auf den *MBean Server* zuzugreifen. Ein Konnektor besteht aus einem Konnektor Server, der direkt auf den *MBean Server* zugreifen kann und einem Konnektor Client, der eine mit dem *MBean Server* identische Schnittstelle auf Seite des Clients aufweist [vgl. SUN03, S. 15].

Das dabei verwendete Protokoll zwischen Client und Server wird üblicherweise vom Hersteller des JMX Agent festgelegt. Der Grund dafür ist offensichtlich, so erklärt die Einleitung von JSR-160: *„Although JSR 3 defines terminology for remote access to instrumentation, it does not standardize any particular remote access API or protocol“* [SUN03, S. 11]. Das heißt, mit dem JSR-003 Standard war der entfernte Zugriff auf den *Instrumentation Layer* (siehe das folgende Unterkapitel: Einteilung in Schichten) noch nicht definiert. Erst seit dem JMX Standards JSR-160 wird versucht, den entfernten Zugriff auf den *MBean Server* zu standardisieren.

Doch auch im JSR-160 Standard wird weiterhin die Möglichkeit geboten, generische Konnektoren zu entwickeln, bei denen das Transport Protokoll und das *Object Wrapping* konfigurierbar sind [vgl. SUN03, S. 41]. Unter *Object Wrapping* versteht man das Serialisieren und Deserialisieren von Java Objekten für die Übertragung im Netzwerk. Besonders beim Deserialisieren muss darauf geachtet werden, dass die Klassendefinition der übertragenen Java Objekte, zum Beispiel die Attribute eines *MBeans*, auch für den Konnektor des Clients verfügbar sind [vgl. SUN03, S. 42f].

Eine Beispiel-Implementierung eines generischen Konnektors ist der JMXMP Konnektor. Nur der RMI Konnektor, mit wahlweise Java Remote Method Protocol (JRMP) oder Internet Inter-ORB Protocol (IIOP) als Transport Protokoll, ist im JSR-160 standardisiert [vgl. SUN03, S. 33].

Weiters enthält der JMX Agent Adaptoren, die es ermöglichen, JMX Daten für JMX fremde Protokolle zur Verfügung zu stellen. Typische Beispiele für JMX Adaptoren sind HTTP und SNMP Adaptoren.

2.2.2. Einteilung in Schichten

In der Literatur wird die JMX Architektur häufig in 3 Schichten oder *Layers* unterteilt [vgl. SUL03, S. 14]. Die innerste Schicht ist der *Instrumentation Layer* (siehe Abbildung 13), der die *Manageable Ressources* und die zugehörigen *MBeans* enthält. Darüber befindet sich der *Agent Layer*, der sich mit den *MBeans* aus der Sicht des *MBean Servers* beziehungsweise des *JMX Agents* befasst. Der *Agent Layer* ist die Schnittstelle zu anderen Komponenten innerhalb desselben Java Prozesses. An den *Agent Layer* schließt der *Distributed Layer* als äußerste Schicht der JMX Architektur an. Der *Distributed Layer* ermöglicht den externen Zugriff auf den *JMX Agent*.

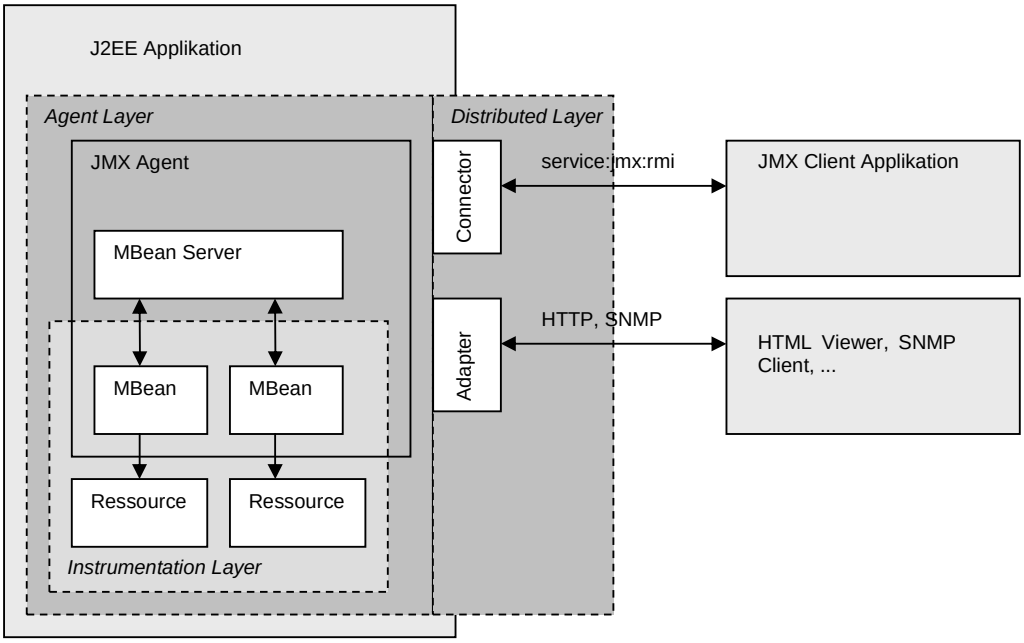

Abbildung 13 – Schichten der JMX Architektur

2.2.3. Architektur bei JSR-77

Der JSR-77 Standard definiert eine Management EJB Komponente oder kurz MEJB, die einen Zugriff auf den *MBean Server* ermöglicht (siehe Abbildung 14). Das MEJB ist üblicherweise wie andere Enterprise Java Beans des jeweiligen Applikationsservers in einem JNDI kompatiblen Verzeichnisdienst registriert.
Die Verbindungsaufnahme zum JNDI Verzeichnisdienst des Applikationsserver ist herstellerspezifisch. Das dabei verwendete Protokoll, der Port und der *Bindname*, also

der Bezeichner unter dem das MEJB im Verzeichnisdienst abgelegt ist, variieren je nach Hersteller des Applikationsservers. Der JSR-77 Standard empfiehlt den Bezeichner „`ejb/mgmt/MEJB`“ als *Bindname*. In der Praxis werden jedoch sehr unterschiedliche Bezeichner verwendet und lediglich die Applikationsserver von Oracle und IBM WebSphere scheinen sich an die Namenskonvention zu halten (siehe Kapitel 5.1).

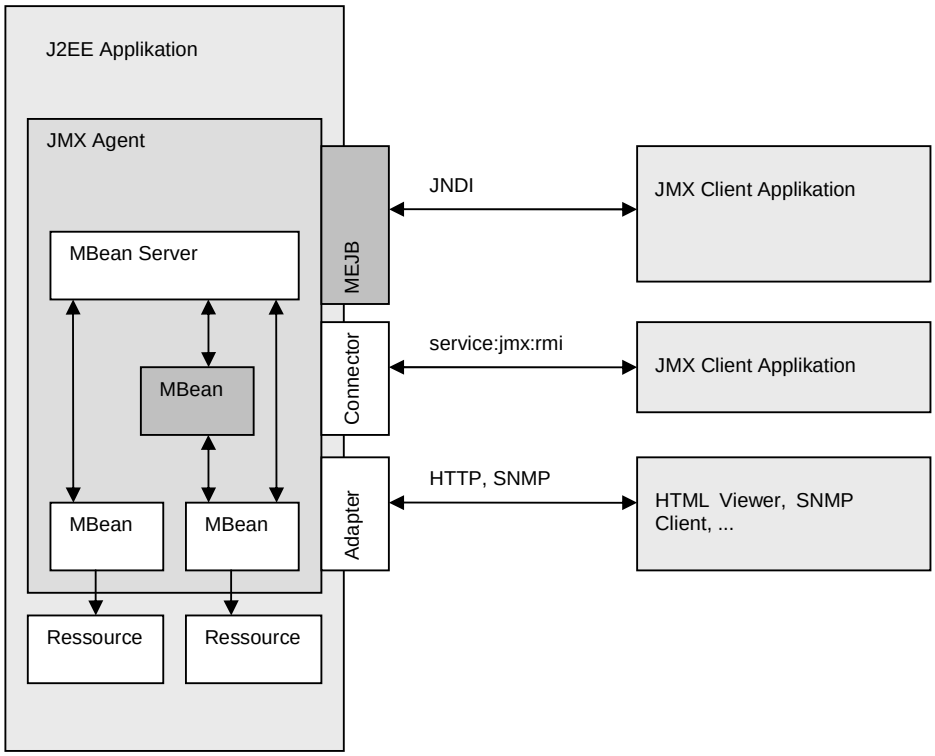

Abbildung 14 - JMX Architektur bei JSR-77

Aus der Sicht des *MBean Servers* sind die *MBeans* auch nach JSR-77 in einer flachen Struktur abgelegt. Allerdings definiert der Standard Typen von *MBeans*, die teilweise in Eltern/Kind Beziehungen zueinander stehen. Somit ist es durch den JSR-77 Standard erstmals die Möglichkeit gegeben, eine Hierarchie zwischen den *MBeans* zu sehen. Diese Möglichkeit wird in Kapitel 4.3.4 ausführlich diskutiert.

2.3. JMX Query Mechanismus

Eine Möglichkeit, eine Auswahl aus dem ungeordneten Haufen von *MBeans* auf dem *MBean Server* zu treffen, ist es, eine Suchabfrage oder Query abzusetzen. Die Definition der gewünschten Auswahl von MBeans erfolgt über einen *ObjectName*, in dem Platzhalter verwendet werden. Man spricht vom *Scope* der Abfrage [vgl. SUN01, S. 111].

Der zweite Teil der Abfrage, die so genannt *Query Expression*, ist optional und kann beliebige Einschränkungen bezüglich Attributwerten beinhalten. Gültige Einschränkungen wären zum Beispiel eine Überprüfung, ob ein numerischer Attributwert größer oder kleiner einer bestimmten Schranke ist oder ob ein String-Attribut einer vorgegebenen Konstante entspricht.

2.3.1. Query Scope

Der den *Scope* einer Abfrage definierende *ObjectName* besteht wie gewohnt aus einem Bezeichner für die Domain, der durch einen Doppelpunkt von einer beliebigen Liste von Name/Wert Paaren getrennt wird (siehe auch Kapitel 2.2.1).

Als Platzhalter werden das ’?’ Symbol, das für ein einzelnes Zeichen steht und das ’*’ Symbol, das für eine beliebe Anzahl von Zeichen steht, verwendet. Das ’*’ Symbol kann nur als Platzhalter für die komplette *Domain* oder als Platzhalter für beliebige Name/Wert Paare verwendet werden. Wenn das ’*’ Symbol als Platzhalter für ein Name/Wert Paar verwendet wird, bedeutet es sinngemäß, dass zusätzlich zu den anderen Bedingungen, beliebig viele weitere Name/Wert Paare folgen können.

Das folgende Beispiel für einen *ObjectName* als Abfrage Scope definiert alle *MBeans* der *Domain* `myDomain`, die das Name/Wert Paar `type=Memory` beinhalten und verwendet das ’*’ Symbol als Platzhalter für beliebige weitere Name/Wert Paare. Die Anzahl der weiteren Name/Wert Paare ist somit beliebig.

```
myDomain:type=Memory,*
```

Ein einzelnes Name/Wert Paar kann erst ab der JMX Implementierung in SUNs JDK 1.6 das ’*’ Symbol als Platzhalter verwenden.

Das folgende Beispiel für einen *ObjectName* definiert alle *MBeans* der *Domain* `myDomain` die das Name/Wert Paar `group=aGroup` sowie den Namen `type` mit einem beliebigen Wert und ansonsten keine weiteren Name/Wert Paare beinhalten.

```
myDomain:type=*,group=aGroup
```

Die *MBean Server* Klasse definiert zwei Arten von Query Methoden. Die Methode `queryMBeans` retourniert ein *Set* von vollständigen MBean Instanzen, während die Methode `queryNames` nur ein *Set* von *ObjectNames*, die das jeweilige *MBean* eindeutig identifizieren, liefert.

2.3.2. Query Expression

Die Klasse *javax.management.Query* enthält zahlreiche statische Methoden zum Erzeugen einer komplexen Query. Teile einer Query Expression können über logische Operatoren wie *'and'*, *'or'* und *'not'* miteinander verknüpft werden [vgl. SUL03, S. 180]. Weiters gibt es statische Methoden für arithmetische Operationen auf Attributwerte sowie statische Methoden für Vergleichsoperationen zwischen Attributwerten und Konstanten [vgl. SUL03, S.181].
Wird keine Einschränkung bezüglich Attributen und Attributwerten gewünscht, so wird der Wert *null* als *Query Expression* übergeben.

3. JMX Monitoring Client API

3.1. Begriffsdefinition und Abgrenzung

JMX behandelt zwei wesentliche Teilaspekte, das Management und das Monitoring von Ressourcen im Allgemeinen. Unter Ressourcen werden sowohl Ressourcen aus dem Bereich des Hardware- und Netzwerkmanagements, als auch Ressourcen im Sinne von Applikationen und Applikationsdaten auf einem Anwendungsserver verstanden.
Die im Rahmen dieser Arbeit entwickelte JMX Monitoring Client API beschäftigt sich mit dem passiven Monitoring von Ressourcen, nicht jedoch mit dem aktiven Eingreifen in die Konfiguration einer Ressource, das in den Bereich des Managements fällt.
Weiters wird im Rahmen dieser Arbeit der Schwerpunkt auf die Entwicklung der API selbst gelegt, nicht jedoch auf die Entwicklung eines Tools, das die API verwendet und die Daten darstellt.
Die Implementierung der JMX Monitoring Client API dient dazu, eine Datenquelle für Monitoring Tools zu schaffen.

3.2. Architektur

Beim Entwurf einer geeigneten Architektur einer JMX Client API muss vor allem beachtet werden, dass jede herstellerspezifische Implementierung einer JMX Datenquelle eigene Konfigurationsanforderungen hat.
Die Datenquellen haben verschiedenste Anforderungen an den Klassenpfad und an die Version der *Java Virtual Machine* (JVM). Daher ist es üblicherweise nicht möglich, dass *MBean Server* verschiedener Hersteller durch denselben Java Prozess abgefragt werden.

3.2.1. Geschlossene Client Architektur

Der Kernpunkt des geschlossenen Architektur-Ansatzes ist es, für jeden *MBean Server* einen eigenen JMX Monitoring Client in einem eigenen Java Prozess zu

starten. Jede Monitoring Datenquelle läuft unter einer eigenen JVM Version und unter einer eigenen Klassenpfad Konfiguration.

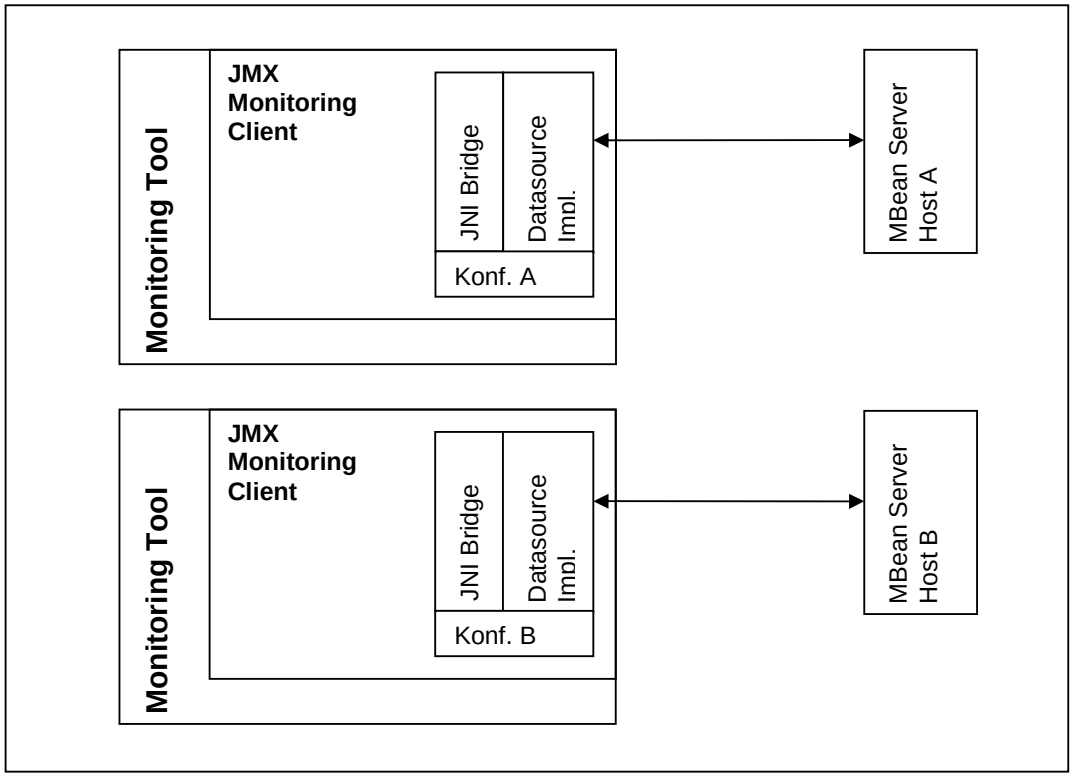

Abbildung 15 – Geschlossene Client Architektur

Der Nachteil dieser Architektur ist, dass bei einer Einbindung des Monitoring Client in ein C++ basierendes Monitoring Tool über das JNI Interface nur eine JVM in den C++ Prozess geladen werden kann. Ein paralleles Monitoren von *MBean Servern* verschiedener Hersteller wäre somit unmöglich.

Als Alternative müssten in einem solchen Fall zwei Instanzen des Monitoring Tools gestartet werden.

Abbildung 15 zeigt zwei unabhängige Instanzen des Monitoring Tools, welche jeweils über die JNI Bridge auf einen entfernten *MBean Server* zugreifen.

3.2.2. Verteilte Client Architektur mit Verwaltung über RMI Registry

Das Charakteristische dieses Lösungsansatzes ist, dass die Verwaltung der Datenquellen über einen externen Prozess, die Remote Method Invocation (RMI)

Registry erfolgt. Der JMX Monitoring Client ist in mehrere Komponenten, die über RMI kommunizieren, aufgesplittert.

Im folgenden, abgebildeten Beispiel werden Daten von drei *MBean Server* gesammelt. Jeder *MBean Server* läuft auf einem eigenen System. *MBean Server* A und B laufen auf ähnlichen Applikationsservern, also auf Applikationsservern vom selben Hersteller und selber Version. *MBean* Server C läuft auf einem Applikationsserver eines anderen Herstellers.

Eine zentrale *Java Virtual Machine*, die in den Prozess des Monitoring Tools geladen wird, startet je nach Bedarf einen oder mehrere RMI Server Prozesse mit entsprechender JVM Version und Klassenpfad Konfiguration (Konfiguration A/B und Konfiguration C). Falls nicht bereits vorhanden, wird die RMI Registry automatisch gestartet. Die RMI Server Prozesse registrieren sich selbstständig in der RMI Registry. Die Schnittstelle zwischen RMI Client und RMI Server ist weitgehend unabhängig von der Konfiguration der Datenquellen.

Laufen auf zwei oder mehreren Hosts ähnliche Versionen von einem Applikationsserver, so kann von einem RMI Server Prozess auf alle diese Datenquellen zugegriffen werden.

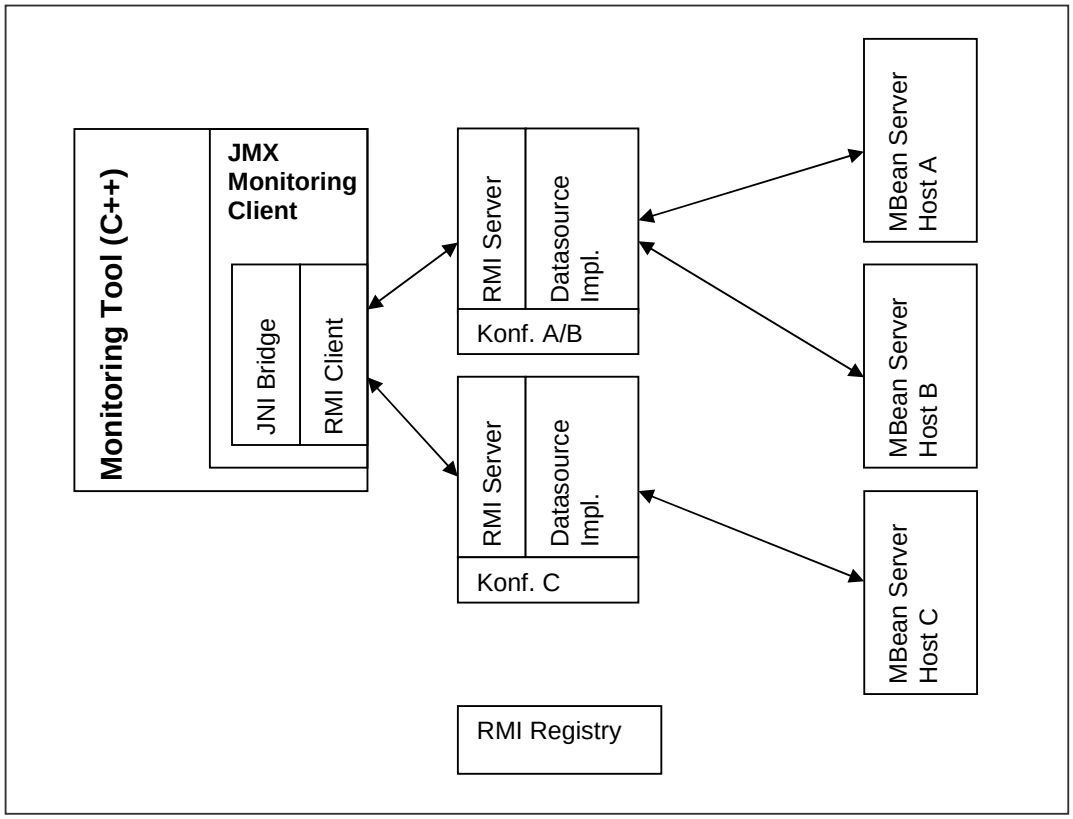

Abbildung 16 – Verteilte Client Architektur

Ein Nachteil dieser Architektur ist, dass der RMI Client nicht unbedingt mit verschiedenen JVM Herstellern kompatibel ist. So können zum Beispiel Kompatibilitäts-Probleme zwischen SUN JVMs und IBM JVMs auftreten. Weiters implementieren und distributieren einige Hersteller eigene Versionen von Standard JMX Klassen wie `javax.management.ObjectName`. RMI Client und Server müssen jedoch identische Versionen der in der RMI Schnittstelle verwendeten Klassen verwenden, ansonsten würden Probleme beim Deserialisieren der Daten auftreten. Somit darf die Schnittstelle zischen RMI Client und Server keine herstellerspezifischen Klassen beinhalten. Siehe auch Kapitel 4.5.2.

3.2.3. Verwendung einer ORB Architektur

Bei diesem Lösungsansatz wird die Konfigurationsverwaltung durch das Monitoring Tool übernommen. Für jeden JMX Monitoring Client wird ein eigener C++ Prozess gestartet, der eine JVM mit entsprechender Version und Klassenpfad Konfiguration lädt.

Da sich diese Arbeit nur auf die Java-seitige Bibliothek und die zugehörige JNI Bridge beschränkt, wird auf das Interface zwischen ORB Client und ORB Server nicht näher eingegangen.

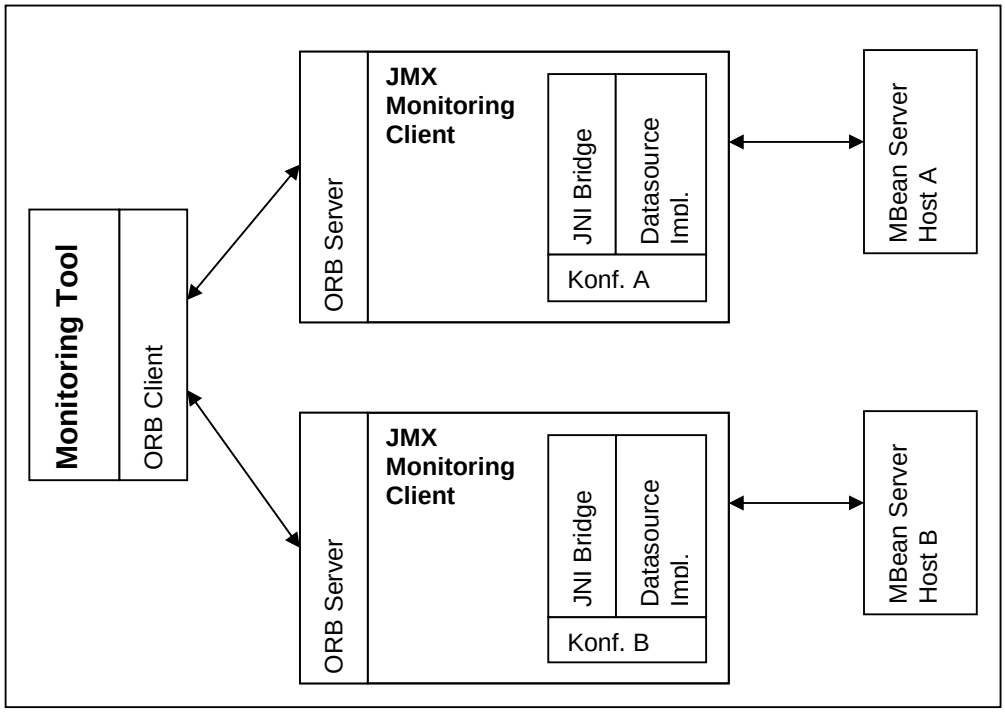

Abbildung 17 - Verteilte ORB Client Architektur

3.3. Implementierung einer verteilten JMX Client Architektur

Im Rahmen dieser Arbeit wurde aus den in Kapitel 3.2 diskutierten Architekturen die verteilte JMX Client Architektur (siehe Kapitel 3.2.2) implementiert. Abbildung 18 zeigt die Kernkomponenten der Architektur. Jeder RMI Server implementiert eine Datenquelle für einen bestimmten Applikationsserver und läuft unter einer spezifischen JVM Version und Klassenpfad Konfiguration. Der Monitoring Client verwendet den RMI Client um über einen oder mehrere RMI Server Monitoring Daten von *MBean Servern* abzufragen.

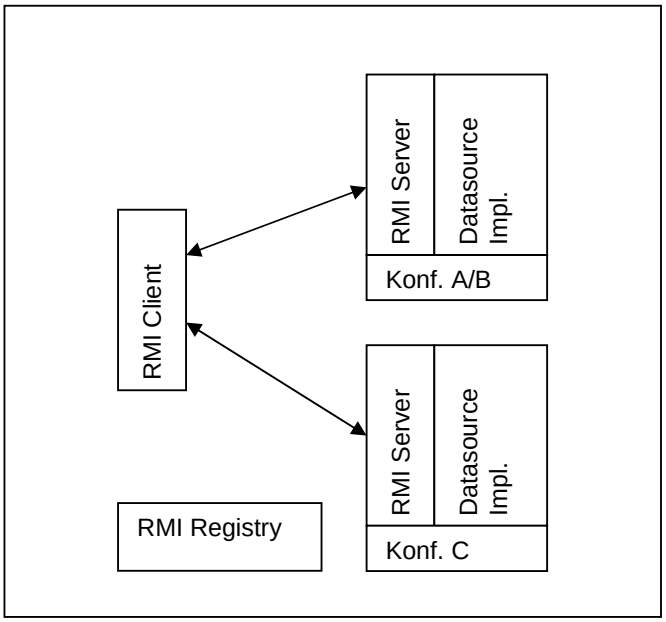

Abbildung 18 – Kernkomponenten der verteilten JMX Client Architektur

Die Schnittstelle zwischen RMI Client und RMI Server untergliedert sich in vier Teilbereiche:

- Starten des RMI Servers (deploy)
- Verbinden des RMI Server zum *MBean Server* (bootstrap)
- Abfragen von Monitoring Daten
- Terminierung des RMI Servers (shutdown)

3.3.1. Starten des RMI Server

Starten des Java Prozesses

Zum Starten des Java Prozesses, in dem der RMI Server läuft, muss die entsprechende JVM und Klassenpfad Konfiguration vorbereitet werden. Ein weiterer Parameter enthält VM Parameter, die beim Starten der JVM berücksichtig werden sollen.
Weiters ist der *Bindname*, also der Name, unter dem der RMI Server in der RMI Registry registriert wird, relevant. Für jede Art von Applikationsserver gibt es eine eigene Implementierung der JMX Datenquelle, welche auf dem RMI Server läuft. Die zur Datenquelle gehörige Klasse, die den Einsprungspunkt für das Hauptprogramm enthält, muss zuletzt spezifiziert werden. Auf der Seite des Clients ergibt sich daher folgende Schnittstelle:

```
public interface IClient
{
  public boolean deploy(String javaHome,
                        String classPath,
                        String vmParams,
                        String bindName,
                        String mainClass);
}
```

Das Starten des Java Prozesses erfolgt über die Methode `exec()` der Klasse `java.lang.Runtime` [siehe SUN05]. Der *Bindname,* der Port unter der die RMI Registry laufen soll und das Logverzeichnis werden als Kommandozeilen-Parameter an den neuen Java Prozess übergeben.
Abbildung 19 zeigt in einem Ablaufdiagramm den ersten Teilschritt, das Starten des Java Prozesses durch die Aktion mit der Bezeichnung „*deploy()*“.

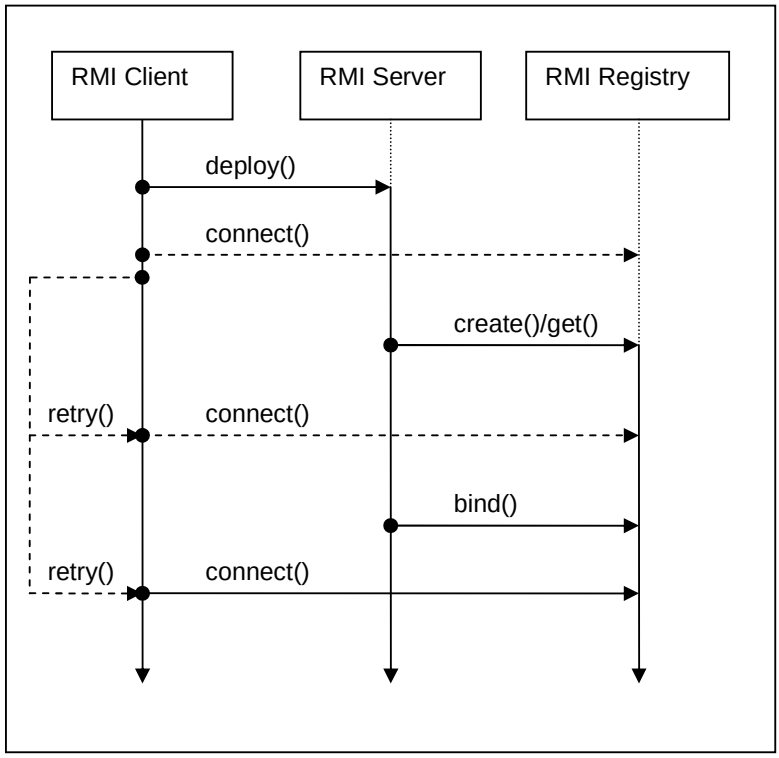

Abbildung 19 – Deployment des RMI Servers und der RMI Registry

Es ist durchaus denkbar, dass sich mehrere RMI Clients einen RMI Server teilen. Die Gründe dafür werden später in Kapitel 4.5.1 beschrieben. (siehe auch Abbildung 31) In einem solchen Fall startet der erste RMI Client wie oben beschrieben den RMI Server. Die weiteren RMI Clients stellen fest, dass der *Bindname* bereits in der RMI Registry vergeben ist und können den bestehenden RMI Server verwenden anstatt ihren eigenen Server zu starten.

Starten und Wiederverwendung der RMI Registry

Der RMI Server versucht gleich nach dem Prozess Start, eine RMI Registry auf dem über einen Kommandozeilen Parameter spezifizierten Port zu kreieren. Falls es noch keine laufende RMI Registry gibt, wird eine RMI Registry im selben Prozess durch den Aufruf der Methode `createRegistry` der Klasse `java.rmi.registry.LocateRegistry` [siehe SUN06] gestartet.

Falls bereits eine RMI Registry für einen anderen RMI Server läuft, schlägt das Starten der RMI Registry mit einer `java.rmi.RemoteException` fehl. In diesem Fall wird die bereits laufende RMI Registry durch Aufruf der Methode `getRegistry` wieder verwendet.

Abbildung 18 zeigt schematisch, wie sich mehrere RMI Server Prozesse eine RMI Registry teilen. Die RMI Registry läuft dabei in Wirklichkeit im Prozess eines der beiden RMI Server, auch wenn in der Abbildung die RMI Server als eigene Komponente dargestellt wird.
Beim Starten der RMI Registry muss beachtet werden, dass diese alle Schnittstellenbeschreibungen der RMI Server[8] im Klassenpfad benötigt, also nicht nur jene des eigenen RMI Servers, durch den die RMI Registry gestartet wurde, sondern auch die Schnittstellenbeschreibungen von all jenen RMI Servern, die sich zukünftig in der selben RMI Registry registrieren.
Abbildung 19 zeigt das Starten bzw. Wiederverwenden der RMI Registry durch die Aktion mit der Bezeichnung *create()*/*get()*.

Registrieren des RMI Server in der RMI Registry

Sobald eine gültige Referenz auf die RMI Registry für den RMI Server verfügbar ist, wird der RMI Server unter dem per Kommandozeilen Parameter spezifizierten *Bindname* in der RMI Registry registriert. Falls der spezifizierte *Bindname* bereits vergeben ist, wird angenommen, dass es sich um einen alten Eintrag handelt und der alte Eintrag durch einen neuen überschrieben.
Abbildung 19 zeigt das Registrieren in der RMI Registry durch die Aktion mit der Bezeichnung *bind()*.

Verbinden zum RMI Server

Nach dem Starten des RMI Server Prozesses verbindet sich der RMI Client über die RMI Registry mit dem neu gestarteten Prozess. Dabei muss berücksichtigt werden, dass das Starten des RMI Servers und der RMI Registry unter Umständen etwas Zeit in Anspruch nimmt und somit die Verbindungsaufnahme erst etwas zeitverzögert erfolgen kann. Schlägt also die Verbindungsaufnahme des Client zum Server oder zur die Registry beim ersten Versuch fehl, werden nach einer kurzen Verzögerung weitere Versuche unternommen.
Abbildung 19 zeigt die Verbindungsaufnahme vom RMI Client zum RMI Server durch die Aktion mit der Bezeichnung *connect()* an. Die zeitversetzte Wiederholung der Verbindungsaufnahme wird durch die Aktion *retry()* dargestellt.

---

[8] Die Schnittstellenbeschreibungen der RMI Server bezeichnet man auch als S*tub Classes*.

3.3.2. Verbinden zum MBean Server

Nachdem, wie in Kapitel 3.3.1 beschrieben, der RMI Server gestartet und eine Verbindung vom RMI Client zum RMI Server aufgebaut wurde, muss im nächsten Schritt die Verbindung zum MBean Server hergestellt werden. Der RMI Server dient dabei als Proxy zwischen RMI Client und MBean Server. Die herstellerspezifischen Anforderungen zur Verbindungsaufnahme mit dem MBean Server werden vom RMI Server übernommen.

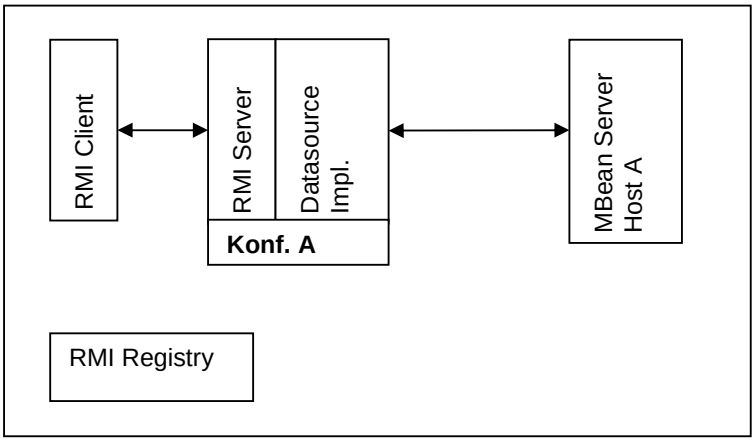

Abbildung 20 - Verbindungsaufnahme zum MBeanServer

Durch den Aufruf der `bootstrap()` Methode teilt der RMI Client dem RMI Server mit, dass dieser die Verbindung zum MBean Server aufnehmen soll. Dabei werden alle relevanten Konfigurationsdaten als Parameter übergeben.

Zu den für die Verbindungsaufnahme notwendigen Konfigurationsdaten zählt die Verbindungsart, also ob per Management Enterprise Java Bean (MEJB) über JNDI oder per *Connector Server* über JSR-160 Standard die Verbindung zum *MBeanServer* aufgenommen wird. Bei Verbindungsaufnahme über den *Connector Server* wird weiters noch das darunterliegende Protokoll unterschieden. Die JMX Monitoring API unterstützt die Protokolle RMI, IIOP und CORBALOC.

Weiters benötigt man die URL des Verzeichnis-Dienstes des Anwendungsservers (*JNDI Provider*) bzw. die *JMXServiceURL* des *Connector Servers* des JSR-160 Standards, bestehend aus Rechnername, Port, Protokoll und optional einem URL Postfix. Der URL Postfix ist ein der URL hinten angestellter String.

Im Falle einer Verbindungsaufnahme per JNDI sind noch der Klassenname der *Initial Context Factory* (siehe Kapitel 2.1.3) und der Name des MEJB notwendig, das die Brücke zum JMX Agent darstellt.
Falls von der Sicherheitskonfiguration des JMX Agent verlangt, müssen noch Benutzername und Passwort übergeben werden.
Die Konfigurationsdaten werden in der Klasse `BootstrapInfo` zusammengefasst. Somit ergibt sich folgende Schnittstelle:

```
public interface IBootstrapInfo
{
  public static final int MODE_JSR160_RMI = 0;
  public static final int MODE_JSR160_IIOP = 1;
  public static final int MODE_JNDI = 2;
  public static final int MODE_JSR160_CORBALOC = 3;
  public static final int MODE_JSR160_RMIIIOP = 4;

  public String getProtocol();
  public String getHost();
  public String getPort();
  public String getContextFactory();
  public int getConnectionMode();
  public String getPrincipal();
  public String getCredentials();
  public String getUrlPostfix();
  public String getCompiledURL();
  public String getMEJBName();
}

public interface IClient
{
  public boolean bootstrap(IBootstrapInfo info);
}
```

3.3.3. Abfragen von Monitoring Daten

Der wesentliche Zweck des JMX Monitoring Client ist es, Attribut-Daten vom *MBean Server* abzufragen. Wie beim Bootstrapping, agiert auch hier der RMI Server als Proxy zwischen RMI Client und MBean Server.

Für die Abfrage der Monitoring Daten ist der Name des *MBeans* und der Bezeichner des *MBean* Attributs notwendig. Bei komplexen Attributen kann zusätzlich die Information, welcher Teilbereich des Attributs abgefragt werden soll, definiert werden.

Der *MBean* Name wird in einer herstellerunabhängigen Klasse `ObjectNameWrp` übertragen (siehe auch Kapitel 4.5.2). Die Beschreibung eines Attributs muss von der Schnittstelle `IAttribute` abgeleitet sein. Für Details zu *MBean* Attributen siehe Kapitel 4.5.3.

Somit ergibt sich folgende Schnittstelle bezüglich Abfrage der Monitoring Daten:

```
public interface IAttribute
{
  public String getAttributeName();
  public ObjectNameWrp getObjectNameWrp();
}

public interface IClient
{
  public Object getAttribute(IAttribute attribute);
}
```

3.3.4. Terminierung des RMI Servers Prozesses

Um einen schonenden Umgang mit den Hardware-Ressourcen zu gewährleisten, wird der RMI Server Prozess nach Ende der Verwendung niedergefahren. Der RMI Client teilt dem RMI Server durch Aufruf der `shutDown()` Methode mit, dass der Client keine weiteren Dienste mehr in Anspruch nimmt.

```
public interface IClient
{
  public void shutDown();
}
```

Bei der Terminierung des RMI Server Prozesses müssen aufgrund der in Kapitel 3.3.1 beschriebenen Besonderheiten beim Starten der RMI Registry zahlreiche Sonderfälle berücksichtigt werden.

Mehrere Clients pro Server

Der erste Sonderfall tritt auf, wenn mehrere RMI Clients gleichzeitig mit dem RMI Server arbeiten. Dies bedeutet, dass der RMI Server erst terminiert werden darf, wenn der letzte verbliebene RMI Client die *shutDown()* Methode aufgerufen hat. Aus diesem Grund wird die Zahl der Referenzen auf den RMI Server mitgezählt. Bei jedem Aufruf der *bootstrap()* Methode wird die Zahl der Referenzen erhöht, durch jede *shutDown()* Methode wird die Zahl der Referenzen verringert. Sind keine Referenzen mehr übrig, kann der RMI Server terminiert werden.

Um sicher zu gehen, dass jeder RMI Client nur seine eigene Referenz freigeben kann, wird jede Referenz durch einen eigenen eindeutigen Bezeichner, in diesem Fall eine fortlaufende Nummer, dargestellt.

Abbildung 21 zeigt in einem Ablaufdiagramm den Lebenszyklus eines RMI Servers, auf den von zwei Clients zugegriffen wird.

Zunächst startet RMI Client A den RMI Server Prozess (*deploy()*). Dieser startet die RMI Registry (*create()*) und registriert sich dort (*bind()*). Sobald der Client mit dem Server zu arbeiten beginnt (*bootstrap(A)*), wird intern die Zahl der Referenzen um eins erhöht (*addRef()*).

Nun verbindet sich RMI Client B über die RMI Registry mit dem RMI Server (*connect()*). Sobald Client B mit dem Server zu arbeiten beginnt (*boostrap(B)*), wird die Zahl der Referenzen wiederum um eins erhöht (*addRef()*).

Schließlich sendet Client A den *shutDown()* Befehl an den RMI Server und dieser entfernt die Referenz zum Client (*removeRef()*). Da jedoch noch eine weitere Referenz vergeben ist, läuft der RMI Server weiter. Erst wenn Client B ebenfalls das Verbindungsende signalisiert (*shutDown(ID B)*), trägt sich der RMI Server aus der RMI Registry aus (*unbind()*) und terminiert (*terminate()*).

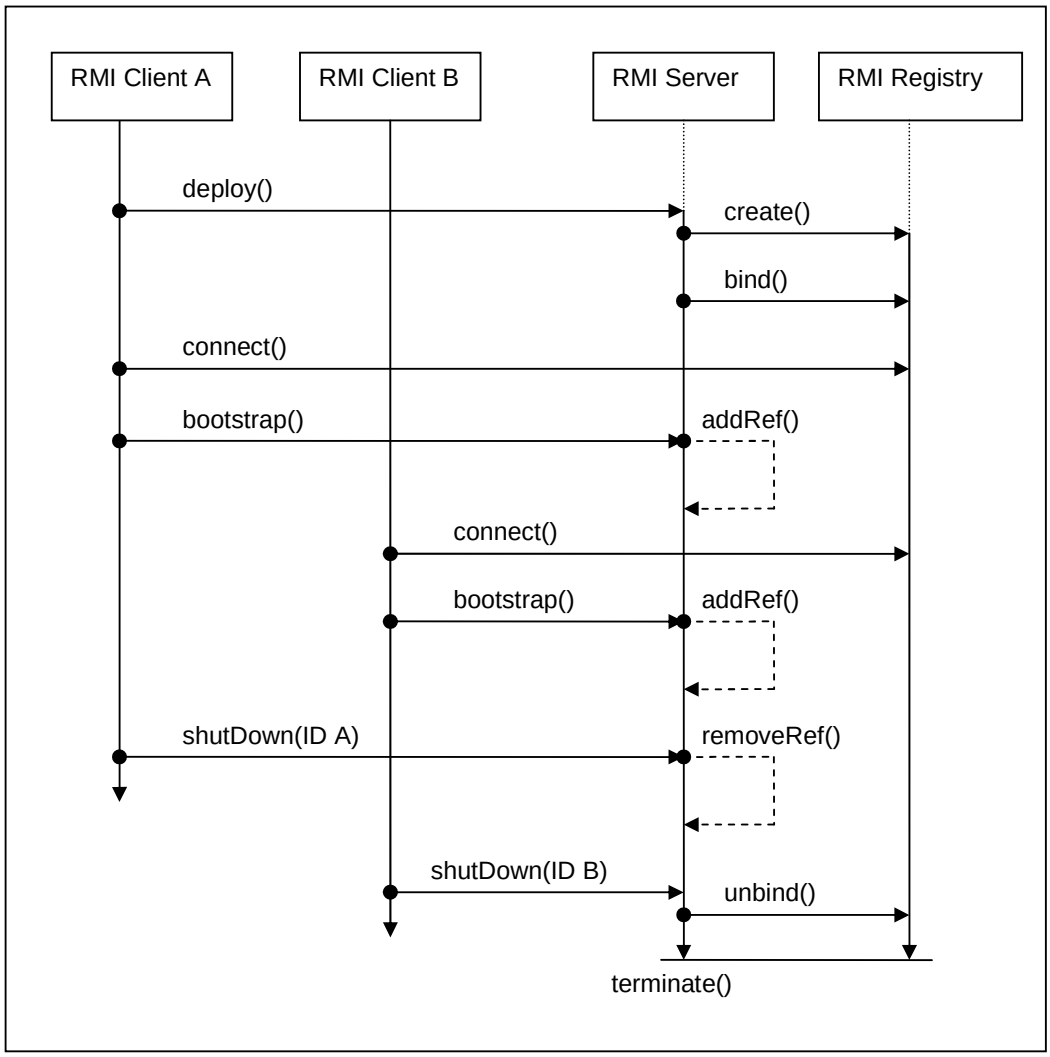

Abbildung 21 – Terminierung des RMI Servers bei mehreren Clients

Mehrere RMI Server pro RMI Registry

Der zweite Sonderfall tritt auf, wenn sich mehrere RMI Server eine RMI Registry teilen. Hierbei muss beachtet werden, dass der Java Prozess der die RMI Registry beinhaltet, unter Umständen selbst dann, wenn die Zahl der Referenzen 0 erreicht hat, nicht beendet werden darf, da die RMI Registry noch von einem anderen RMI Server in Verwendung ist. Dieses Problem kann gelöst werden, indem jeder RMI Server sich vor Beendigung ordnungsgemäß bei der RMI Registry austrägt (unbind).

Der RMI Server, der die RMI Registry im selben Prozess gestartet hat, muss somit so lange laufen, bis die Referenz Zahl 0 erreicht hat und zusätzlich keine anderen RMI Server in der RMI Registry mehr registriert sind.

Abbildung 22 zeigt in einem Ablaufdiagramm den Lebenszyklus von zwei RMI Servern, die sich eine RMI Registry teilen.

Zunächst startet RMI Client A den Prozess von RMI Server A (*deploy(A)*). Dieser wiederum startet die RMI Registry (*create()*) und registriert sich selbst dort (*bind(A)*). Nun kann Client A mit dem Server arbeiten (*connect(), bootstrap()*). Währenddessen startet Client B den Server B (*deploy(B)*). Server B verwendet dieselbe RMI Registry wie Server A (*get()*) und registriert sich dort (*bind(B)*).

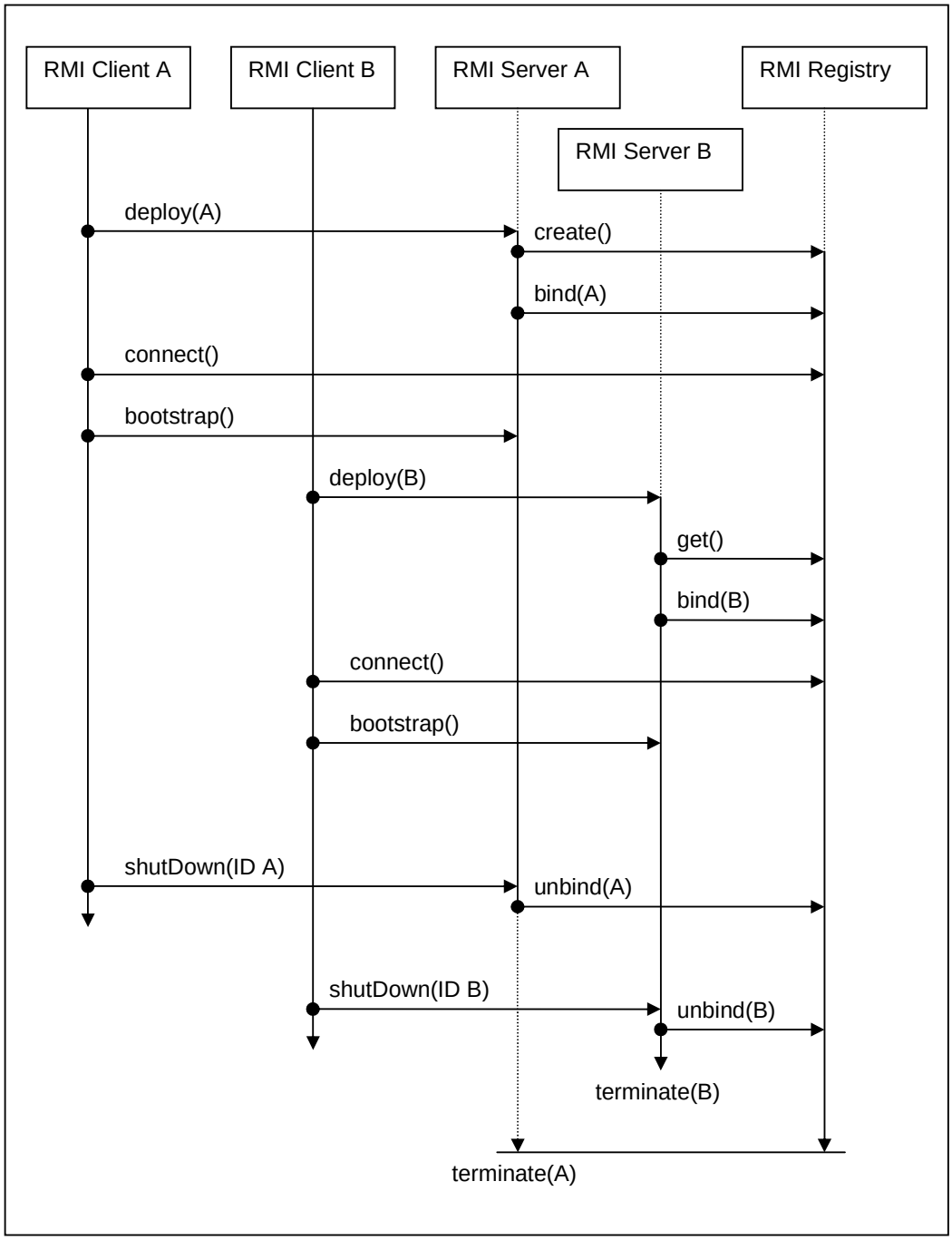

Abbildung 22 - Terminierung der RMI Server Prozesse bei gemeinsamer RMI Registry Nutzung

Wenn nun Client A seinem Server mit der *shutDown()* Methode signalisiert, dass er nicht mehr benötigt wird, so trägt sich der Server A in der RMI Registry aus (*unbind(A)*). Er erkennt jedoch, dass in der Registry immer noch Server B registriert ist und läuft somit weiter. Erst wenn Client B dem Server B signalisiert, dass er

terminieren kann (*shutDown(ID B)*) und dieser sich in der RMI Registry austrägt (*unbind(B)*), erkennt Server A, dass nun auch die RMI Registry nicht mehr benötigt wird und terminiert sich und die RMI Registry (*terminate(A)*).

3.3.5. Zusammenfassung der RMI Client Schnittstelle

```
public interface IAttribute
{
  public String getAttributeName();
  public ObjectNameWrp getObjectNameWrp();
}

public interface IBootstrapInfo
{
  public static final int MODE_JSR160_RMI = 0;
  public static final int MODE_JSR160_IIOP = 1;
  public static final int MODE_JNDI = 2;
  public static final int MODE_JSR160_CORBALOC = 3;
  public static final int MODE_JSR160_RMIIIOP = 4;

  public String getProtocol();
  public String getHost();
  public String getPort();
  public String getContextFactory();
  public int getConnectionMode();
  public String getPrincipal();
  public String getCredentials();
  public String getUrlPostfix();
  public String getCompiledURL();
  public String getMEJBName();
}
```

```
public interface IClient
{
  public boolean deploy(String javaHome,
                        String classPath,
                        String bindName,
                        String mainClass);
  public void connect();
  public boolean bootstrap(IBootstrapInfo info);
  public Object getAttribute(IAttribute attribute);
  public void shutDown();
}
```

4. JMX Browser API

4.1. Begriffsdefinition und Abgrenzung

Der zweite große Teilbereich der Arbeit, die JMX Browser API, beschäftigt sich mit der Auswahl der von einem *MBean Server* angebotenen Daten. Welche Daten stehen auf einem *MBean Server* zur Verfügung, wie sind diese strukturiert und welche sind besonders für das Monitoring geeignet?
Auch hier liegt das Hauptaugenmerk auf der Entwicklung der API selbst, nicht auf deren Verwendung in einem Tool.

4.2. Problemstellungen

4.2.1. Strukturierung der Information

Das zentrale Problem bei der Entwicklung eines JMX Browsers ist es, dass alle *MBeans* auf einem *MBean Server* in einer flachen Struktur abgelegt sind. *MBeans* werden über eine Liste von Name/Wert Paaren identifiziert. Es ist jedoch keine hierarchische Beziehung zwischen den *MBeans* erforderlich. Die einzige Strukturierung ist durch die Zuordnung der *MBeans* zu *Domains* gegeben.
Da ein einzelner *MBean Server* einer Beispielanwendung oft schon tausende von *MBeans* beinhaltet, ist es sicherlich nicht zielführend, diese in einer Liste darzustellen. Ziel ist es somit, die *MBeans* eines *MBean Servers* soweit zu strukturieren, dass eine übersichtliche und zugleich intuitive Darstellung der *MBeans* möglich ist. Zudem darf die Strukturierung keinen Widerspruch zu bestehenden JMX Standards darstellen.

4.2.2. Heterogenität der Implementierungen der verschiedenen Hersteller

Ein Problem ist nicht nur, für einen einzelnen Hersteller eine geeignete Strukturierung der Daten zu finden. Die ausgewählte Darstellungsmethode sollte für möglichst alle Hersteller ähnliche Ergebnisse liefern.

4.2.3. Flexibilität, Erweiterbarkeit, Generizität

Aufgrund der großen Anzahl von Anwendungsserver-Herstellern und der großen Anzahl an Produktversionen, die eingesetzt und neu entwickelt werden, ist es im Rahmen dieser Arbeit unmöglich, für alle möglichen Varianten eine eigene Lösung zu implementieren.
Das Ziel bei der Entwicklung der JMX Browser API ist es, die marktführenden Anwendungsserver zu unterstützen und sich an den von der Java Community definierten Standards zu orientieren.
Der Kerncode der JMX Browser API muss so weit wie möglich frei von herstellerspezifischer Implementierung und vor allem frei von herstellerspezifischen Fremdklassen bleiben.

4.2.4. Darstellung komplexer Attribute

Nicht nur die Strukturierung der *MBeans*, sondern auch die Strukturierung der Attribute eines einzelnen *MBeans* ist ein entscheidendes Thema. Denn nicht alle *MBean* Attribute stellen einfache Datentypen dar. Attribute können Tabellen und Strukturen enthalten. Die API muss eine Möglichkeit geben, solche komplexen Datenstrukturen in geeigneter Form darzustellen und auf deren Detailinhalte zuzugreifen.

4.2.5. Auswahl der Attribute

Nicht alle *MBeans* und nicht alle Attribute eines *MBeans* sind für Monitoring geeignet. JMX wird auch zur Administration von Applikationsservern verwendet. Daher enthalten *MBeans* Konfigurationsdaten in der Form von Strings oder konstanten numerischen Werten, die für den Aspekt des Monitoring nicht berücksichtig werden können und sollen.

4.2.6. Aufbereitung der Attributwerte

Nicht alle Attributwerte sind ohne Aufbereitung für Monitoring geeignet. Man nehme zum Beispiel das MBean “java.lang:type=OperatingSystem“ einer *Java Virtual Machine* von SUN. Das MBean definiert ein Attribut “ProcessCpuTime“, das misst, wieviel CPU-Zeit der Prozess verbraucht hat. Die verbrauchte Zeit wird dabei

aufsummiert. Für das Monitoring dieses Attributs ist es aber interessanter festzustellen, wieviel CPU-Zeit einer bestimmten Zeitspanne verbraucht wurde. Der absolute Wert an verbrauchter CPU-Zeit ist eher nebensächlich.

4.3. Lösungsansätze zur Strukturierung der MBeans

4.3.1. Strukturierung der Daten nach dem MBean Typ

Die Strukturierung der Daten nach dem *MBean* Typ ist ein einfacher Ansatz zu der in Kapitel 4.2.1 geforderten Strukturierung der *MBeans* eines *MBean Servers*.
MBeans enthalten überlicherweise ein *Property*, das den Typ des *MBeans* bezeichnet. Der *MBean* Typ kommt also als Name/Wert Paar im *ObjectName* vor. Oft werden alle *MBeans*, die eine Instanz derselben Java Klasse sind, durch einen *MBean* Typ zusammengefasst.
Der JSR-77 Standard schreibt die Verwendung der Bezeichnung „j2eeType“ für den *MBean* Typ vor. Bei den im Rahmen dieser Arbeit erforschten Applikationsservern werden die Bezeichnungen „type“, „Type“ oder „j2eeType“ verwendet.
Üblicherweise wird von den verschiedenen Herstellern nur eine der 3 verschiedenen Schreibweisen verwendet und kann somit als Standardeinstellung direkt im Quellcode voreingestellt werden. Um eine möglichst große Flexibilität gegenüber Änderungen in künftigen Versionen zu bieten, empfiehlt es sich jedoch, den Namen des *Property*, das den *MBean* Typ definiert, auch über einen Optionendialog änderbar zu machen.
MBeans des Sun JDK 1.5 bzw. 1.6 sowie des Websphere 6.0 Applikationsserver verwenden die Bezeichnung „type“. JBoss 4.0 und Oracle Application Server 10.1.3 halten sich an den JSR-77 Standard.
Es scheint, dass man sich bei WebSphere 6.0 erst sehr spät entschlossen hat, den JSR-77 Standard zu implementieren. *MBeans* des Websphere 6.0 Applikationsservers können noch sowohl ein *Property* „Type“ als auch ein *Property* „j2eeType“ beinhalten.
Die API von WebLogic 8.0 beinhaltet eine eigene Basisklasse für *MBeans*, das *WebLogicMBean*. Über diese Klasse ist ein typsicherer Zugriff auf das *Type* Attribut möglich. Aus Kompatibilitätsgründen wird aber in der JMX Browser API auch bei WebLogic davon ausgegangen, dass ein *Type Property* definiert ist und dieses anstelle der API Funktion verwendet.

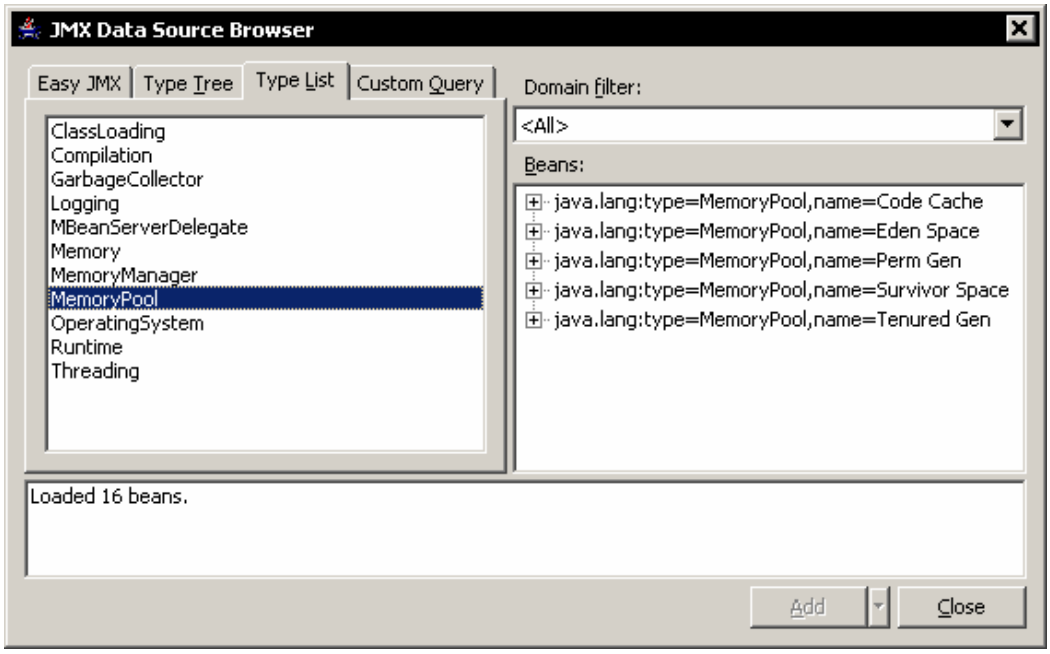

Abbildung 23 - Strukturierung der MBeans nach MBean Typ

Mit dieser nun gewonnen Information des *MBean* Typs, ist eine übersichtlichere Strukturierung der *MBeans* eines Applikationsservers möglich. Die Aufbereitung der Daten erfolgt in der JMX Browser API nach Verbindungsaufnahme zum *MBean Server*. Das Programm iteriert durch alle registrierten *MBeans* und sammelt alle verschiedenen Typbezeichnungen. Dabei wird eine Map angelegt, die als Schlüssel den Typbezeichner und als Wert die Liste von *MBeans* dieses Typs enthält. *MBeans*, die keiner dieser *MBean* Typen zugeordnet werden konnten, kommen in eine eigene Kategorie, die mit „<other>“ bezeichnet wurde.

Das Anwendungsszenario zu dieser Art der Datenstrukturierung sieht wie folgt aus: Der Benutzer des JMX Browser wählt nun zuerst aus einer Liste einen bestimmten *MBean* Typ aus und bekommt alle *MBeans* von diesem Typ dargestellt.

Abbildung 23 zeigt die MBeans eines Sun JDK 1.5 Prozesses strukturiert nach dem *MBean* Typ. Im linken Bereich sieht man die Liste von *MBean* Typen, während im rechten Bereich die einzelnen *MBeans* zum selektierten *MBean* Typ dargestellt werden.

4.3.2. Strukturierung der Daten in Baumform

Für kleine Datenmengen, zum Beispiel die *MBeans* von JDK 1.5, ist die im vorigen Punkt beschriebene Strukturierung in Listenform nach dem Typ Property schon ausreichend, um eine übersichtliche Darstellung der Daten zu gewährleisten. Für die *MBeans* eines Applikationsservers ergeben sich aber teilweise über hundert *MBean* Typen, wodurch die Übersichtlichkeit weiterhin leidet.
Die Idee ist es, Beziehungen zwischen den *MBean* Typen zu finden und somit eine Baumdarstellung zu ermöglichen.

4.3.3. Baumdarstellung bei Weblogic 8.x und 9.0

Wie bereits in Punkt 4.3.1 erwähnt, enthält die WebLogic API mit der *WebLogicMBean* Klasse eine eigene Basisklasse. Dieses bietet die Methode `getParent()`, um das Vater *MBean* eines *MBeans* zu bestimmen.
Mit dieser Vater/Kind Beziehung zwischen den *MBeans* wäre es somit eine Baumdarstellung möglich. Allerdings würde ein Baum aus tausenden *MBeans* nicht unbedingt eine Vereinfachung gegenüber Strukturierung nach *MBean* Typen darstellen.
Eine andere Idee ist es, die Vater/Kind Beziehung zwischen den *MBeans* auch auf die *MBean* Typen abzubilden.

- **Regel 1**

  Sei *MBean* A vom Typ TA und *MBean* B vom Typ TB. Typ TA ist ungleich Typ TB. Weiters wissen wir über das Vater Attribut, dass *MBean* A der Vater von *MBean* B ist. Somit ist anzunehmen, dass auch Typ TA der Vater von Typ TB ist.

Um eine konsistente Darstellung des Baumes zu garantieren, ist noch eine 2. Bedingung notwendig:

- **Regel 2**

  Sei *MBean* A vom Typ TA und *MBean* B vom Typ TB. *MBean* A ist der Vater von *MBean* B. Somit ist nach Regel 1 Typ TA der Vater von Typ TB.

  Es gibt kein *MBean* B' vom Typ TB und kein *MBean* A' vom Typ TA, so dass *MBean* B' der Vater von *MBean* A' ist. Denn dies würde bedeuten, dass nach Regel 1 Typ TB der Vater von Typ TA.

In der Praxis ist mir bei *MBeans* der Standard Installation von WebLogic noch keine Verletzung von Regel 2 aufgefallen. Somit scheint Regel 1 zur Beziehung zwischen *MBean* Typen eine gültige Annahme zu sein. Auch in der Dokumentation der Weblogic API konnte nichts Gegenteiliges gefunden werden.

Da bei Anwendungen eines Kunden jedoch weitere *MBeans* zum *MBeanServer* hinzugefügt werden, die sich eventuell nicht an Regel 2 halten, ist zumindest dafür zu Sorgen, dass beim Generieren des Baumes keine Endlosschleifen oder Abstürze auftreten.

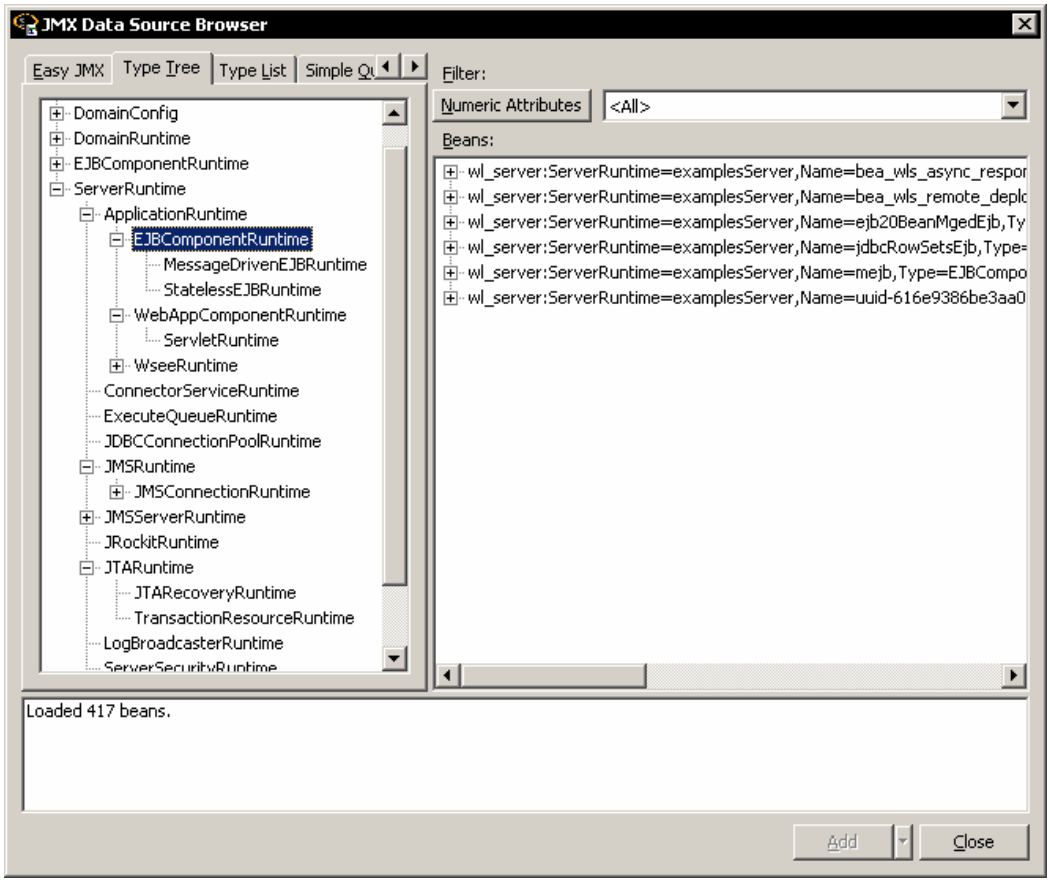

Abbildung 24 – Baumdarstellung bei WebLogic 9.0

Abbildung 24 zeigt im linken Bildteil den aus den *MBean* Typen generierten Baum einer Beispielanwendung auf dem WebLogic 9.0 Application Server. Sechs *MBeans* vom selektierten *MBean* Typ „`EJBComponentRuntime`“ sind im rechten Bildteil ersichtlich. Die MBean Typen „`MessageDrivenEJBRuntime`“ und „`StatelessEJBRuntime`“ stellen Subtypen vom MBean Typ „`EJBComponentRuntime`“ dar.

4.3.4. Baumdarstellung bei JSR-77 kompatiblen Applikations-Servern

Der JSR-77 Standard enthält zahlreiche Regeln, die für die strukturierte Aufbereitung der Daten im JMX Browser hilfreich sind.

Zum Ersten definiert JSR-77 ein Schlüssel Property mit der Bezeichnung ‚j2eeType’, dass für alle *MBeans* eines JSR-77 kompatiblen Applikationsservers vorgeschrieben ist [vgl. SUN02, S. 21]. Durch diese Information ist eine eindeutige Zuordnung von *MBeans* zu *MBean* Typen möglich. Siehe auch Kapitel 4.3.1.

Weiters definiert der Standard weitere Schlüssel Properties für jeden *MBean* Typ, die die übergeordneten *MBean* Typen bestimmen. Die vorgeschriebenen Schlüssel *Properties* werden in Tabelle JSR77.3-1 des Standards aufgelistet [vgl. SUN02, S. 20ff].

Anhand folgender Regeln wird aus den vorgeschriebenen Schlüssel *Properties* ein Baum aus *MBean* Typen aufgebaut:

- **Regel 1**

  Gibt es zu *MBean* Typ TA keine vorgeschriebenen übergeordneten *MBean* Typen, so stellt *MBean* Typ TA ein Wurzelelement im Baum dar.

  $$Vater(TA) = \{\} \rightarrow Kind(Wurzel) = \{TA\}$$

  Eine Ausnahme der Regel stellt *MBean* Typ ‚J2EEServer’ dar. Für *MBeans* vom Typ ‚J2EEServer’ sind zwar keine übergeordnete *MBean* Typen vorgeschrieben, aber da alle *MBeans* einer Domain angehören, macht es Sinn den *MBean* Typ ‚J2EEServer’ dem *MBean* Typ ‚J2EEDomain’ unterzuordnen.

- **Regel 2 – Transitivität**

  Sei *MBean* Typ TA dem *MBean* Typ TB übergeordnet und *MBean* Typ TB dem *MBean* Typ TC übergeordnet. Somit ist anzunehmen, dass auch *MBean*

Typ TA dem *MBean* Typ TC überzuordnen ist, auch wenn dies nicht explizit in Tabelle JSR77.3-1 auf gelistet ist.

Vater(TB) = {TA}

Vater(TC) = {TB}

→ Vater(TC) = {TA, TB}

- **Regel 3**

  Sei *MBean* C vom Typ TC und nicht im JSR 77 Standard explizit als bekannter Typ aufgelistet. *MBean* C enthält die *MBean* Typen TA und TB als Schlüssel *Properties*. Sei *MBean* Typ TB Kind von *MBean* Typ TA. Somit wird *MBean* Typ TC als Kind von *MBean* TB im Baum eingefügt.

  Vater(TC) = {TA, TB}

  Vater(TB) = {TA}

  → Kind(TB) = {TC}

- **Regel 4**

  Sei *MBean* C vom Typ TC und nicht im JSR 77 Standard explizit als bekannter Typ aufgelistet. *MBean* C enthält die *MBean* Typen TA und TB Schlüssel *Properties*. Sei die Beziehung von *MBean* Typ TA und Typ TB unbekannt. Somit kann Typ TC noch keine eindeutige Position im Baum zugeordnet werden und kommt auf die Warteliste.

  Vater(TC) = {TA, TB}

  Vater(TA) = {?}

  Vater(TB) = {?}

  Sollten später weitere *MBeans* verarbeitet werden, die eine Beziehung zwischen Typ TA und Typ TB definieren, so kann auch die Position von Typ TC anhand von Regel 3 bestimmt werden.

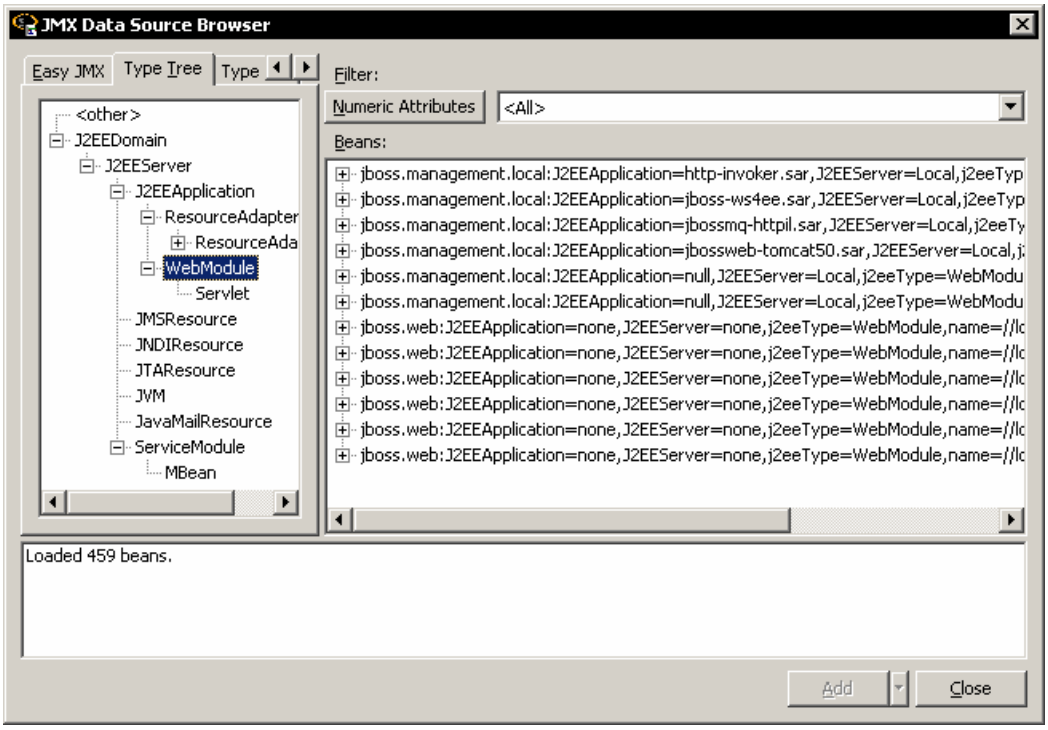

Abbildung 25 – Baumdarstellung bei JBoss 4.0

Abbildung 25 zeigt im linken Bildteil die *MBean* Typen des JBoss 4.0 Application Servers in Baumform. Die Elemente des Baums sind allgemein bekannte Managed Object Typen aus dem JSR-77 Standard. Der rechte Bildteil zeigt zwölf *MBeans* vom Typ „`WebModule`“.

4.3.5. Baumdarstellung nach Reihenfolge der Properties

Eine Alternative zur Baumdarstellung der *MBean* Typen ist eine Baumdarstellung der MBean *Properties*. Dabei wird angenommen, dass die *MBean Properties* von links nach rechts nach Wichtigkeit gereiht sind und die *Property* Namen für eine Kategorisierung geeignet sind.

So könnte man ein *MBean* mit dem Namen “domain:a=x,b=y“ folgendermaßen in einen Baum einordnen, dass der erste Property Name “a“ an der Wurzel eingefügt wird und der zweite *Property* Name “b“ ein Blatt von “a“ darstellt.

Diese Darstellung stellt leider eine proprietäre Lösung dar. Die Reihenfolge *MBean Properties* ist in keinem dem Verfasser dieser Arbeit bekannten JMX Standard festgelegt. So liefert zum Beispiel eine Query nach dem *ObjectName*

`domain:a=x,b=y,*` sowohl *MBeans* mit dem *ObjectName* `domain:a=x,b=y,*` als auch *MBeans* mit dem *ObjectName* `domain:b=y,a=x,*`.
Dennoch scheint diese Art der Datenstrukturierung bei manchen Herstellern kommerzieller J2EE Produkte Anwendung zu finden. Da das von SUN JDKs ab der Version 1.5 mitgelieferte Tool "jconsole" ebenfalls diese Art der Datenstrukturierung unterstützt, ist es überlegenswert diese Art der Darstellung in die JMX Browser API aufzunehmen.

4.4. Lösungsansätze zur Strukturierung der MBean Attribute

4.4.1. Abbildung komplexer Attribute nach JSR-77

JSR-77.6 definiert ein Interface, *javax.management.j2ee.statistics.Stats*, für komplexe Attribute, die überwiegend Performanz relevante Daten liefern [vgl. SUN02, S. 53ff]. Jedes *Stats*-Attribut enthält eine Liste von Statistiken, die von dem Attribut angeboten werden. Von der Basis-Schnittstelle *javax.management.j2ee.statistics.Stats* sind zahlreiche Schnittstellen für verschiedene Teilbereiche eines Applikationsservers abgeleitet und eine Anzahl von Statistiken vorgeschlagen, die vom *MBean* geliefert werden sollen.
So enthält jede JSR-77 kompatible Applikationsserver-Implementierung ein *MBean* vom j2eeType „JVM". Dieses enthält ein Attribut mit dem Namen „stats", dass das *JVMStats* Interface implementiert. Das JVMStats Interface definiert, dass zumindest die Statistiken „UpTime" und „HeapSize" angeboten werden sollen.
Die Implementierung der einzelnen Statistiken ist jedoch nicht verpflichtend. So beschreibt Kreger: *„Note that the defined Stats set of metrics for a particular instance of J2EEManagedObject does not have to be supported completely for the object to be a compliant StatisticsProvider model. The server vendor may elect to provide accessors to a subset of the JSR 77 Stats set for any managed object type"* [KRE03, S. 452].
Darüber hinaus können je nach Hersteller des Applikationsservers noch zusätzliche Statistiken geliefert werden.
Vom *Statistic* Interface gibt es mehrere Unterklassen, was zusätzliche Komplexität beim Monitoren von JSR-77 Statistiken mit sich bringt. So definiert eine *TimeStatistic* Minimum-, Maximum- und Gesamtwert eines Messwertes. Im Rahmen dieser Arbeit

wird angenommen, dass in den meisten Fällen nur der Gesamtwert relevant für Monitoring ist, die übrigen Werte können in der derzeitigen Implementierung nicht abgefragt werden. Auch eine *RangeStatistic* unterteilt sich in mehrere Messwerte. Auch hier werden die von der Statistik zusätzlich angebotenen Minimum- und Maximumwerte in der JMX Browser API ignoriert und nur der aktuelle Wert kann abgefragt werden.

In der JMX Browser Bibliothek werden die einzelnen Statistiken eines Attributes wie eigenständige Attribute behandelt. Die Liste der Attribute eines *MBeans* liefert also einen Listeneintrag für jedes Attribut und einen zusätzlichen Listeneintrag für die einzelnen Statistiken eines Attributes. Dabei wird bei der Darstellung eines komplexen Attributs anstatt des Attributname die Namenskonvention `<Attributname>.<Statistikname>` verwendet. Die interne Abbildung von komplexen Attributen wird später in Kapitel 4.5.3 im Detail beschrieben.

Abbildung 26 zeigt ein *MBean* eines JBoss Applikationsservers vom Typ JVM. Es enthält eine Liste von Attributen von verschiedensten Datentypen wie `int`, `boolean` und `java.lang.String`. Weiters enthält es ein Attribut vom Typ `javax.management.j2ee.statistics.Stats`, welches wiederum die Statistiken `HeapSize` und `UpTime` enthält.

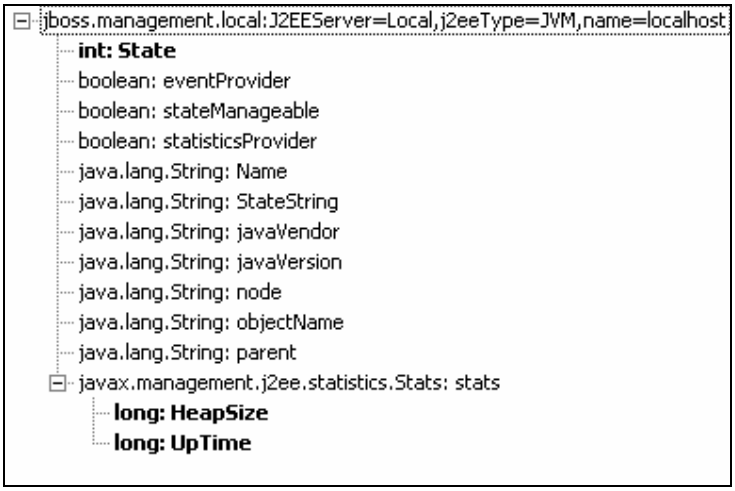

Abbildung 26 - MBean mit JVMStats Attribut

Eine weitere Ausnahme stellen die Interfaces `JDBCStats` und `JCAStats` dar. Diese enthalten nicht eine einfache Liste von Statistiken, sondern Listen von Statistik-Objekten, welche wiederum selbst eine Liste von Statistiken anbieten.

Abbildung 27 zeigt ein `JCAResource` *MBean* auf JBoss 4.0. Das *Stats*-Attribut enthält eine Liste von `JCAConnectionPoolStats` Objekten. Die Länge der Liste ist in diesem Beispiel 1, dargestellt 0-basiert durch den Index in [] Symbolen. Jedes `JCAConnectionPoolStats` Objekt enthält wieder eine Reihe von Substatistiken, in diesem Beispiel 7 Stück.

Abbildung 27 - MBean mit JCAStats Attribut

Bei der JMX Browser API internen Darstellung wird die Namenskonvention `<Attributname>.<Statistikname>[<Index>].<Substatistik Name>` verwendet.

4.4.2. Komplexe Attribute nach JSR-003: CompositeData

Ein weiteres Beispiel eines komplexen Attributes ist *javax.management. openmbean.CompositeData*. Ein *CompositeData* Attribut besteht aus einer Liste von Sub-Objekten, deren Datentyp durch *OpenType* Objekte beschrieben wird. Die Sub-Objekte werden auch als *Data Items* bezeichnet. Jedes *OpenType* Objekt in der Liste beschreibt wiederum entweder einen primitiven Datentyp, ein Array von primitiven Datentypen oder einen komplexen Datentyp, der wiederum vom Typ *Composite* oder *Tabular* sein kann.

Die JMX Browser API liefert bei der Liste von Attributen eines *MBeans* zusätzlich einen Listeneintrag für jedes *Data Item* eines komplexen Attributs. Bei der Darstellung von *CompositeData* Attributen wird in der JMX Browser API die Namenskonvention `<AttributName>.<Data Item>` verwendet. Tiefere Hierarchien,

wo ein *CompositeType* wiederum *CompositeType* Objekte enthält, sind nach dem JSR-003 Standard durchaus möglich, denn dieser definiert *„To comply with the design patterns for open MBeans, all data items must have a type among the set of basic data types. Because this set also includes CompositeData objects, complex hierarchies can be represented by creating composite types that contain other composite types"* [SUN01, S. 64].
Mehrstufige Hierarchien können in der derzeitigen Implementierung der JMX Browser API nicht dargestellt werden.
Abbildung 28 zeigt ein MBean einer SUN JDK 1.6 *Java Virtual Machine*. Das MBean enthält zwei einfache Attribute vom Typ int bzw. boolean. Weiters enthält es zwei komplexe Attribute vom Typ *CompositeData*. Jedes komplexe Attribut besteht aus vier *Data Items*, alle vom Typ *java.lang.Long*.

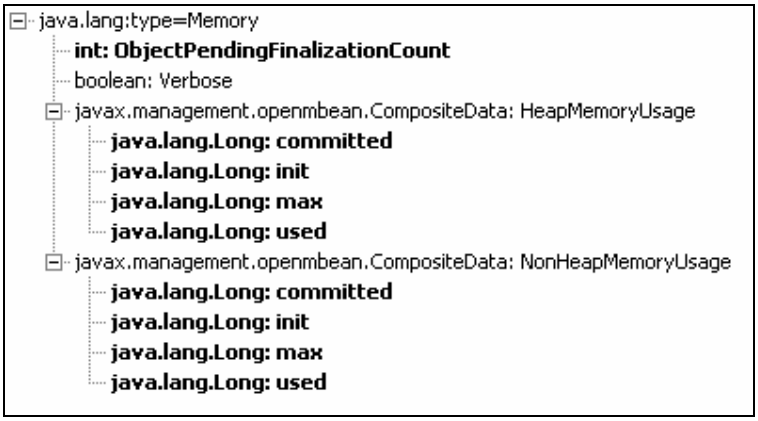

Abbildung 28 - CompositeData Attribut bei SUN JDK 1.6

4.4.3. Komplexe Attribute nach JSR-003: TabularData

Das Interface *javax.management.openmbean.TabularData* ist ähnlich strukturiert wie das *CompositeData* Interface. Auch hier besteht ein Attribut aus einer Liste von *Data Items*. Der wesentliche Unterschied ist, dass die Anzahl der *Data Items* nicht wie bei *CompositeData* fix vorgegeben ist, sondern beliebig Reihen hinzugefügt oder wieder entfernt werden können. Ein weiteres Merkmal von *TabularData* Attributen ist, dass alle Tabellenzeilen dieselbe Datenstruktur aufweisen müssen.
Intern wird die Liste der *Data Items* in einer Collection-Klasse vom Typ Map abgelegt. Jede Tabellenzeile wird von einem Schlüssel referenziert.

Bei primitiven, also einspaltigen Tabelleneinträgen reicht ein einfacher Schlüssel. Das bedeutet, falls die Map einen *SimpleType*, zum Beispiel java.lang.String, als Schlüssel verwendet, so wird dieser auch in der JMX Browser API als Schlüssel verwendet.

Abbildung 29 - TabularData Attribut bei SUN JDK 1.6

Abbildung 29 zeigt die System Properties in einer SUN JDK 1.6 *Java Virtual Machine* als einspaltige Tabelle. Jede Tabellenzeile wird über den Namen des jeweiligen System Property referenziert.

Bei komplexen, also mehrspaltigen Tabelleneinträgen wird die komplette Typbeschreibung der Tabellenzeile als Schlüssel verwendet. Da sich für die Darstellung ein komplexes Objekt schlecht als Schlüssel eignet, wird anstelle dessen die Position des *Data Item* in der Liste als Schlüssel verwendet.

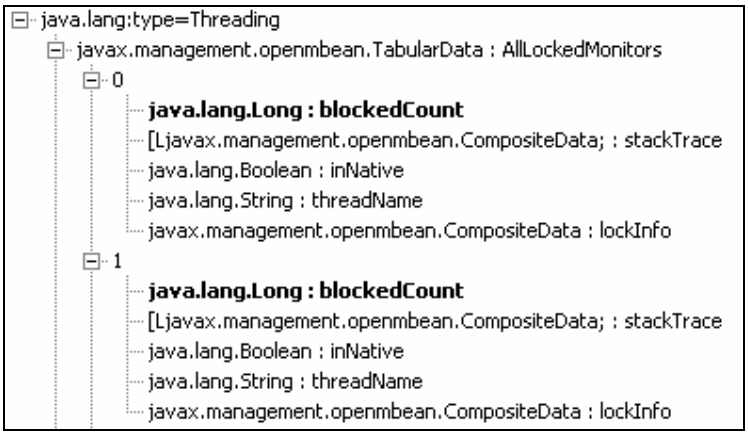

Abbildung 30 - TabularData Attribut bei SUN JDK 1.6 Beta

Abbildung 30 zeigt Informationen über Locked Monitors einer SUN JDK 1.6 Beta[9] *Java Virtual Machine* als mehrspaltige Tabelle. Die Tabellenzeilen werden vom Attribut intern über ein *CompositeData* Objekt referenziert. In der JMX Browser API werden anstelle dessen die Tabellenzeilen fortlaufend nummeriert. Die Tabelle weist fünf Spalten von unterschiedlichen Datentypen auf.

[9] Das Attribut AllLockMonitors ist in der aktuellen Version JDK 1.6 Beta2 nicht mehr verfügbar. Leider ist leider unbekannt, ob das Attribut in der Release Version verfügbar sein wird oder nicht.

4.5. Lösungsansätze zur Flexibilität und Erweiterbarkeit

4.5.1. Auslagerung der Algorithmen auf den RMI Server

Bei den ersten Prototypen einer JMX Browser GUI, wurde die JMX Browser Bibliothek direkt vom Monitoring Tool verwendet. Da dies jedoch herstellerspezifische Konfiguration benötigt, wurde jede GUI Instanz mit herstellerspezifischen Einstellungen und Bibliotheken gestartet. Es war daher nicht möglich, mehrere verschiedene *MBean Server* mit einer GUI Instanz zu überwachen. Bei kommerziellem Einsatz ist es jedoch wünschenswert, mit mehreren MBean Servern gleichzeitig zu arbeiten. Somit ist gefordert, dass die GUI unabhängig von dem zu erforschenden Applikationsserver ist.
Um den JMX Browser GUI frei von herstellerspezifischem Bibliotheken zu halten, macht es Sinn, Teile der JMX Browser Bibliothek über ein RMI Interface anzubieten.
Der in Kapitel 3.2.2 beschriebene RMI Server kann problemlos um die für einen JMX Browser notwendigen Bibliotheksfunktionen erweitert werden.

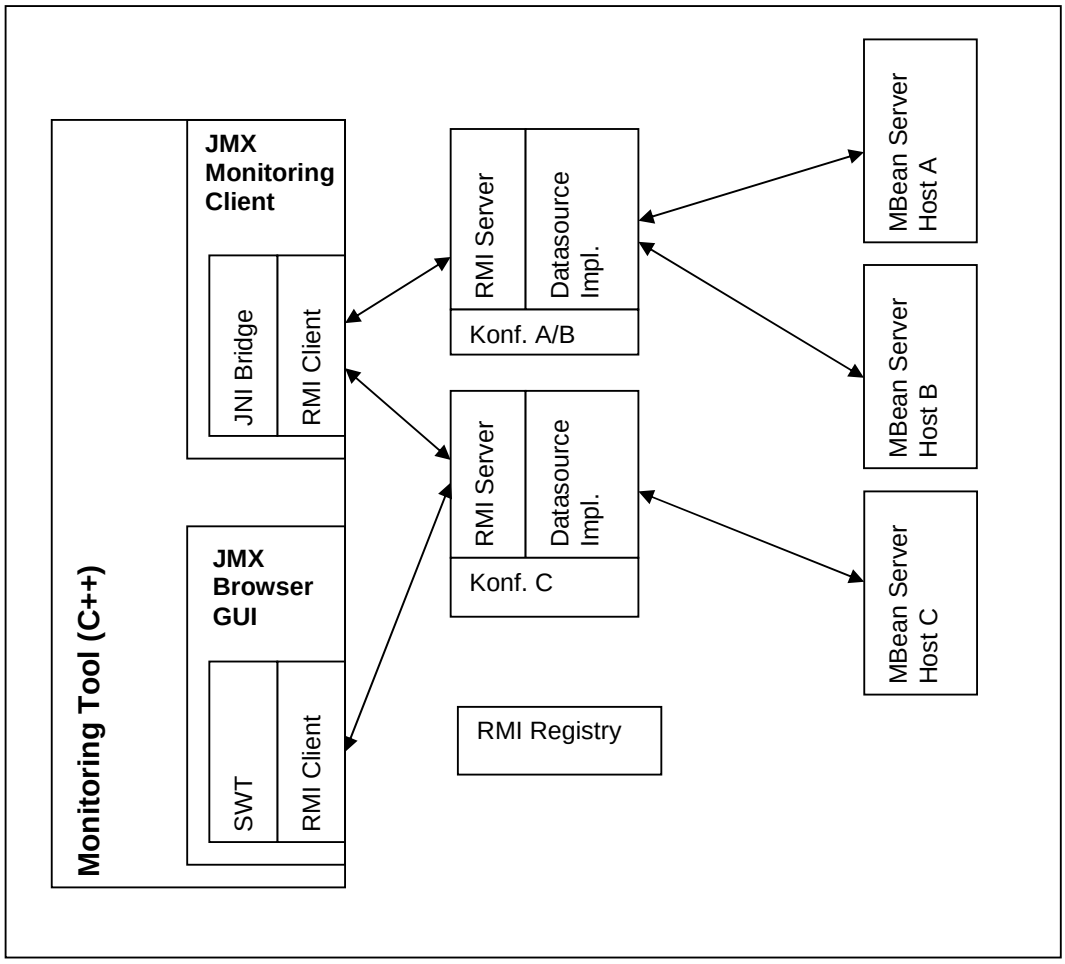

Abbildung 31 – Erweiterte RMI Client/Server Architektur

Die JMX Browser GUI agiert als RMI Client. Code zur Analyse und Aufbereitung der Daten läuft im RMI Server Prozess. Die JMX Browser GUI stellt die aufbereiteten Daten dar. Dadurch wird erreicht, dass nur mehr der RMI Server mit einer herstellerspezifischen Konfiguration laufen muss. Somit ist der erste Schritt der in Kapitel 4.2.3 geforderten Flexibilität der JMX Browser API erreicht. Die JMX Browser GUI kann problemlos mit mehreren RMI Servern gleichzeitig kommunizieren. Somit ist gewährleistet, dass ein GUI Prozess für alle Monitoring Szenarios ausreicht.

4.5.2. Abbildung der MBean Namen in einer herstellerunabhängigen Datenstruktur

Bei der in Kapitel 4.5.1 dargestellten Architektur tritt das Problem auf, dass alle in der RMI Schnittstelle zwischen RMI Server und RMI Client vorkommenden Datentypen herstellerunabhängig sein müssen, ansonsten würden Probleme beim Deserialisieren der Daten auftreten. Grundsätzlich kann man davon ausgehen, dass der Großteil der *MBeans* auf einem *MBeanServer* herstellerabhängig ist. Somit darf keine *MBean* Klasse in der Schnittstelle vorkommen.
Also wird nicht das *MBean* selbst übertragen, sondern nur der Bezeichner des *MBeans*, der durch die *javax.management.ObjectName* Klasse definiert wird. Doch auch hier können noch Probleme auftreten, wenn zum Beispiel ein Hersteller die *ObjectName* Klasse ableitet oder eigens implementiert um noch weitere Informationen in den *ObjectName* zu verpacken. So leitet der Weblogic Applikationsserver von der Klasse *ObjectName* in der Klasse *weblogic.management.WebLogicObjectName* ab, diese kann also von einem herstellerunabhängigen Client nicht deserialisiert werden.
Die Lösung dieses Problems ist jedoch einfach, da ein *ObjectName* problemlos in einen *java.lang.String* und wieder zurück konvertiert werden kann. Der Konstruktor der *ObjectName* Klasse hat nur einen *String* als Parameter und die `toString()` Methode von *ObjectName* liefert einen eindeutigen *String*, aus dem später wieder ein *ObjectName* Objekt erzeugt werden kann. Daher eignet sich für eine herstellerunabhängige Darstellung des *MBean* Bezeichners folgende Klassen-Definition:

```
/**
 * Wrapping all ObjectName instances for RMI communication.
 */
public class ObjectNameWrp implements Serializable, Comparable
{
  private String mName;

  public ObjectNameWrp(String name)
  {
    mName = name;
  }
```

```
  public ObjectName toObjectName()
    throws MalformedObjectNameException
  {
    return new ObjectName(mName);
  }
}
```

Somit kann sowohl der RMI Client, als auch der jeweilige RMI Server mit jeweils seiner eigenen Version der Klasse *ObjectName* arbeiten. Die Methode `toObjectName()` der Klasse `ObjectNameWrp` liefert eine für den jeweiligen Java Prozess gültige Instanz. Für die Serialisierung bei der Interprozess Kommunikation wird jedoch nur die Klassenvariable `mName` benötigt, und diese stellt durch den Datentyp String kein Problem dar.
Der *ObjectName* enthält das Typ *Property* sowie alle Schlüssel *Properties*. Somit kann aus der Liste aller *MBean ObjectNames* problemlos die Strukturierung der *MBeans* in Listen (siehe Kapitel 4.3.1) und Baumform bei JSR-77 kompatiblen MBean Servern (siehe Kapitel 4.3.4) vorgenommen werden.
Die Liste der *MBean ObjectNames* wird also nur einmal vom RMI Server zum Client übertragen. Die Generierung der *MBean* Typ Liste und des *MBean* Typ Baumes kann am Client erfolgen. Für die Strukturierung in Baumform beim Weblogic Applikationsserver (siehe Kapitel 4.3.3) ist zusätzlich pro *MBean* noch die Information über das Vater *MBean* notwendig. Daher wird der Name des Vater *MBeans* als weitere Klassenvariable zu `ObjectNameWrp` hinzugefügt.

```
/**
 * Wrapping all ObjectName instances for RMI communication.
 */
public class ObjectNameWrp implements Serializable, Comparable
{
  private String mName;
  private String mParent;

  /*...*/
}
```

Die für die Strukturierung der *MBeans* Typen in Listen und Baumform notwendigen Daten werden also durch eine einzige Methode der RMI Schnittstelle bereitgestellt:

```
public interface RemoteJmxDataSource extends Remote
{
  public ObjectNameWrp[] getAllBeans() throws RemoteException;
}
```

4.5.3. Abbildung der MBean Attribute in einer herstellerunabhängigen Datenstruktur

Die in Kapitel 4.5.2 dargestellt Methode, anstelle der kompletten *MBeans* nur die *MBean* Namen in herstellerunabhängiger Form an den JMX Browser Client zu liefern, hat zur Folge, dass die Information über die *MBean* Attribute verloren geht. Daher ist zur Abbildung der Attribute eines *MBeans* in der JMX Browser API ein weiterer Aufruf einer RMI Methode notwendig. Ähnlich wie bei den *MBean* Bezeichnern werden auch hier die Attribute in einer herstellerunabhängigen Datenstruktur abgebildet.

```
public class SimpleAttribute implements IAttribute
{
  private String mAttributeName;
  private ObjectNameWrp mObjectNameWrp;
  private String mType;

  public SimpleAttribute(ObjectNameWrp objName, String attrName,
    String type)
  {
    mObjectNameWrp = objName;
    mAttributeName = attrName;
    mType = type;
  }

  /*...*/
}
```

Die Klasse *SimpleAttribute* enthält den Attribut-Namen, den Datentyp des Attributs und eine Referenz auf den *MBean* Bezeichner.
Die Klasse *ComplexAttribute* erweitert die Klasse *SimpleAttribute* enthält eine zusätzliche Klassenvariable, die den Detailbereich des Attributs definiert. Somit ist es, wie in Kapitel 4.4.1 definiert, möglich, Detaildaten eines Attributs zu messen, vorausgesetzt, dass diese einen numerischen Wert darstellen.

```
public class ComplexAttribute extends SimpleAttribute implements
  IAttribute
{
  private String mAttributeDetailName;

  public ComplexAttribute(ObjectNameWrp objNameWrp, String attrName,
    String type, String attributeDetailName)
  {
    super(objNameWrp, attrName, type);
    mAttributeDetailName = attributeDetailName;
  }

  /*...*/
}
```

Von der Klasse *ComplexAttribute* werden noch die Klassen *TabularAttribute* und *StatsAttribute* abgeleitet. Diese dienen im wesentlich nur zur typsicheren Unterscheidung der komplexen Attribute. Die Klasse *StatsAttribute* definiert noch eine zusätzliche Klassenvariable, die die Einheit des Attributs, z.B. Sekunden, definiert. Die Klasse *TabularAttribute* enthält noch Klassenvariablen für Zeilen- und Spaltenadressierung.
Für die RMI Schnittstelle ergeben sich somit 2 weitere Methoden. Eine Methode `getAttributes`, die die Liste der Attribute eines *MBeans* liefert und eine Methode `getAttributeValue`, die den konkreten Wert eines Attributs liefert. Um die Schnittstelle generisch zu halten, wird für alle Attribut Typen noch eine Basis Schnittstelle, `IAttribute`, definiert. `IAttribute` leitet sich von *Serializable* und *Comparable* ab.

```
public interface IAttribute extends Serializable, Comparable
{
  public String getAttributeName();
  public ObjectNameWrp getObjectNameWrp();
  public String getType();
}

public interface RemoteJmxDataSource extends Remote
{
  public ObjectNameWrp[] getAllBeans() throws RemoteException;
  public IAttribute[] getAttributes(ObjectNameWrp objectNameWrp)
    throws RemoteException;
  public Object getAttributeValue(IAttribute attribute)
    throws RemoteException;
}
```

Das *Serializable* Interface ist notwendig, damit die Attribut Klassen vom RMI Server an den RMI Client übertragen werden können. Das *Comparable* Interface ermöglicht, dass die Attribute für die Anzeige in einem Monitoring Tool sortiert werden können.

4.6. Behandlung kumulierter Attributwerte

Bei Attributen, die einen kumulierten numerischen Wert darstellen, ist es oft wünschenswert, anstelle des absoluten Wertes das Delta zum letzten Messzeitpunkt darzustellen. Wie in Kapitel 4.2.6 beschrieben, interessiert den Beobachter z.B. weniger der absolute Wert des Attributs “ProcessCpuTime“, sondern die Steigerung seit dem letzten Messzeitpunkt. Die Berechnung des Delta Wertes erscheint zuerst ein triviales Problem zu sein, bei näherer Betrachtung ergeben sich jedoch Komplikationen.

Abbildung 32 zeigt die Rohdaten eines kumulierten Attributwertes y. Die Daten in diesem Beispiel sind konstruiert um das Problem anschaulicher darzustellen. In unregelmäßigen Abständen zwischen 2 und 17 Sekunden wird der Attributwert y um dy erhöht. Auf den ersten Blick erkennt man einen deutlichen Anstieg der Kurve in der Mitte der Zeitachse. Ansonsten ist die Steigung relativ konstant.

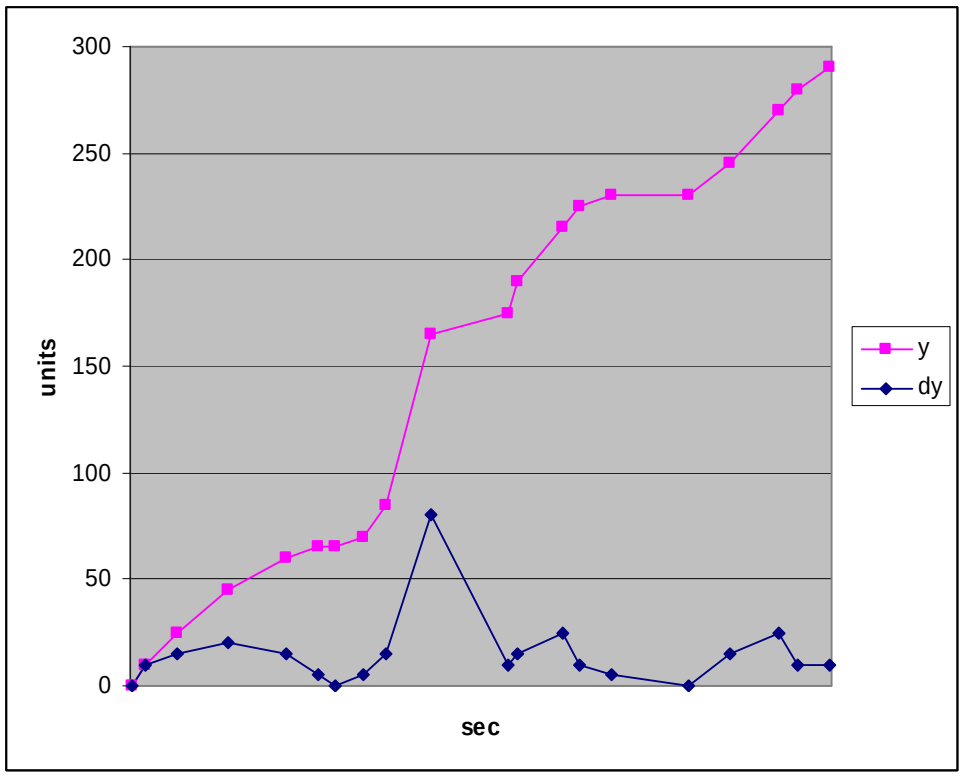

Abbildung 32 - Kumulierter Attributwert y, dy

Abbildung 33 zeigt die Steigerungsrate von dy nach dx, also der Anstieg von y in Einheiten (units) pro Sekunde. Die Berechnung von dy/dx erfolgt ereignisbasiert in unregelmäßigen Abständen bei Eintreffen eines neuen Messwertes. Wie zu erwarten ergibt sich ein starker Anstieg in der Mitte der Zeitachse. Doch zusätzlich gibt es noch eine 2. Spitze zu Beginn der 2. Hälfte der Zeitachse. Diese wird durch 2 sehr nahe beisammen liegende Messwerte verursacht. Durch den geringen Wert von dx an dieser Stelle ist dy/dx überdurchschnittlich hoch, die Interpretation dieser 2. Kurvenspitze obliegt jedoch dem Beobachter.

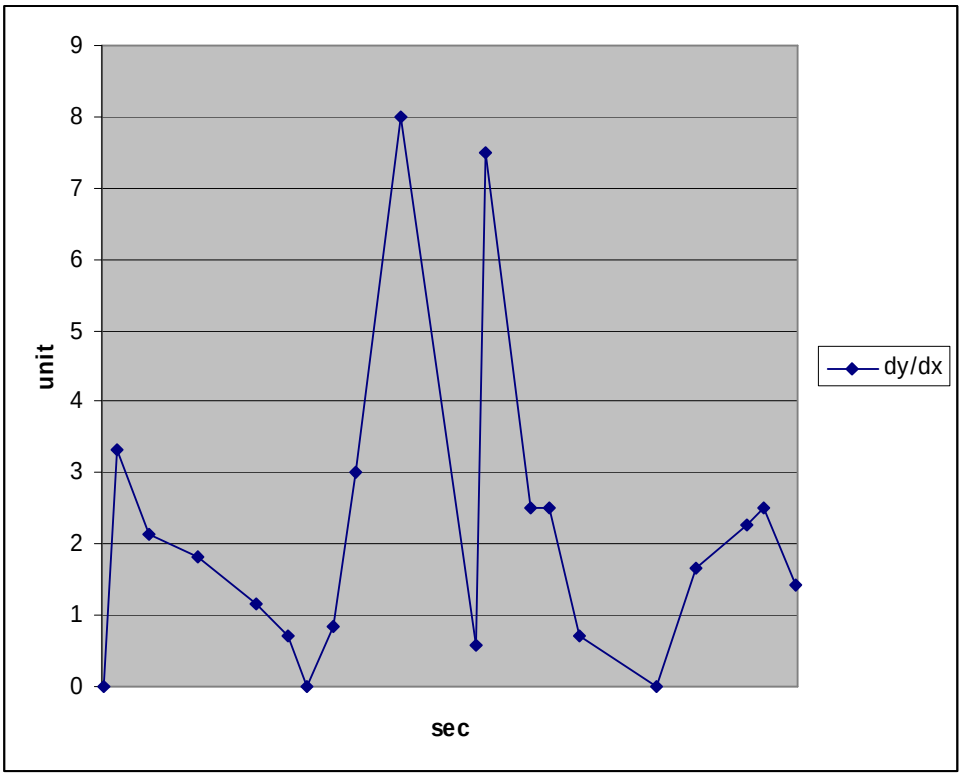

Abbildung 33 - dy/dx; Serverseitige Berechnung

Wird nun derselbe Steigerungswert dy von einem Monitoring Client gemessen, so kann die Berechnung der Steigerungsrate nicht mehr ereignisbasiert erfolgen. Man könnte zwar serverseitig den jeweils letzten dy/dx Wert speichern. Da jedoch der Monitoring Client zu jedem willkürlichen Zeitpunkt Daten vom Server abfragen kann, wäre es dem Zufall überlassen, ob der Monitoring Client einen Zeitpunkt mit einer Kurvenspitze oder einem Kurvental erwischt. Es würden zweifellos einige Details der Kurve für den Beobachter verloren gehen. Würden mehrere unabhängige Clients auf

dieselben Messwerte zugreifen, so können aufgrund leicht unterschiedlicher Messzeitpunkte völlig unterschiedliche Ergebnisse resultieren.

Die Berechnung der Steigerungsrate muss daher über das Monitoring Intervall des Clients gemittelt werden. Abbildung 34 zeigt 2 Kurven die in verschiedenen Intervallen, 5 bzw. 10 Sekunden, die gemittelte Steigerungsrate dy darstellen. Bei einem 5 Sekunden Intervall kommt es häufiger vor, dass sich von einer Messung bis zu nächsten keine Änderung des kumulierten Werts ergeben hat, also die Steigerungsrate gleich 0 ist. Eine 2. Kurve, die mit einem 10 Sekunden Intervall arbeitet, zeigt einen etwas geglätteten Verlauf der Steigerungsrate.

Ein relevantes Detail ist, dass beide Kurven nur mehr eine markante Kurvenspitze aufweisen. Die 2. Kurvenspitze wie sie in Abbildung 33 noch zu sehen war, fällt bei einer gemittelten Steigerungsrate nicht mehr ins Gewicht.

Eine über das Monitoring Intervall gemittelte Steigerungsrate liefert somit ein unter Anführungszeichen besseres Ergebnis. Nun kann der Server jedoch nicht im vornherein wissen, mit welchem Intervall die Clients arbeiten und es wäre auch zuviel Aufwand, für alle möglichen Intervalle die Steigerungsrate voraus zu berechnen.

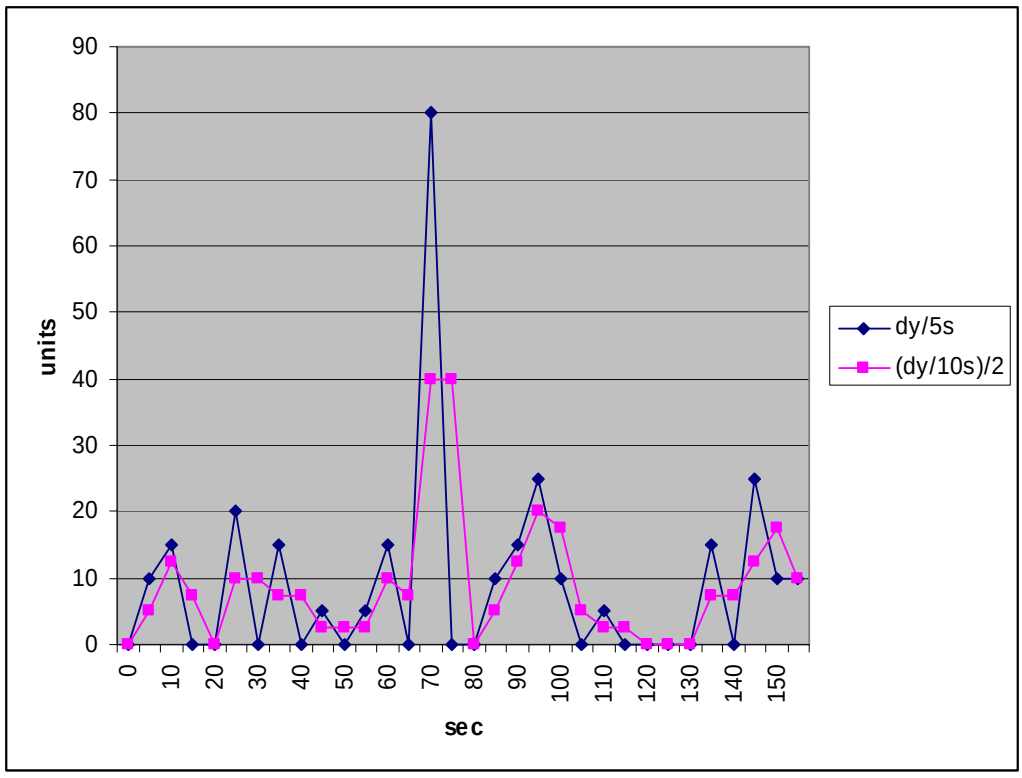

Abbildung 34 – dy/5s, dy/10s; clientseitige Berechnung

Zusammenfassend ist zu sagen, dass eine serverseitige Berechnung der Steigerungsrate zu ungenau wäre, da sie das Intervall des Clients nicht berücksichtigen kann. Daher macht es Sinn, die kumulierten Rohdaten an den Client zu liefern und die Berechnung dem Client zu überlassen. Dieser kann je nach Wahl des Monitoring Intervalls eine detailiertere oder eine mehr geglättete Kurve berechnen.
Für die JMX Browser API bedeutet dies, dass es dem Benutzer bei der Auswahl der Attribute möglich sein muss, einen Attributwert als kumulierten Wert y zu markieren, um den Client dazu veranlassen, die gemittelte Steigerungsrate dy je nach eingestelltem Monitoring Intervall zu berechnen. Für das Interface `IAttribute` ergibt sich folgende Änderung:

```
public interface IAttribute extends Serializable, Comparable
{
  public String getAttributeName();
  public ObjectNameWrp getObjectNameWrp();
  public String getType();
  public boolean isTimebased();
}
```

Jedes Attribut weiß somit über die Methode `isTimebased` selbst, ob es einen kumulierten Wert darstellt, der auf der Client-Seite nachbearbeitet werden muss.

5. Konfigurationsdaten und Bootstrapping

5.1. Herstellerspezifisches Bootstrapping

5.1.1. Allgemeines

Der Begriff Bootstrapping wird in der Informatik in zahlreichen Teilbereichen verwendet. Unter Bootstrapping wird in diesem Zusammenhang, wie bereits in Kapitel 3.3.2 beschrieben, die Verbindungsaufnahme zum *MBean Server* von einem externen Prozess aus verstanden. Während die Funktionsweise und das Interface eines *MBean Servers* innerhalb eines Java Prozesses weitgehend standardisiert sind, so gibt es bei der Verbindungsaufnahme durch externe Prozesse zahlreiche Spezifika, die Beachtung finden müssen. Zudem ist aus Sicherheitsgründen oft der entfernte Zugriff auf den *MBean Server* unterbunden oder durch Passwörter gesichert.

5.1.2. SUN JVM 1.5, 1.6

Wenn eine JVM mit Standardeinstellungen gestartet wurde, ist kein Remote Zugriff auf den *MBean Server* möglich. Um den Remote Zugriff zu ermöglichen, muss folgender Parameter beim Starten der JVM übergeben werden:

```
-Dcom.sun.management.jmxremote.port= <number>
```

Weiters sind optional folgende Startparameter hilfreich, um die Authentisierung und Verschlüsselung auszuschalten:

```
-Dcom.sun.management.jmxremote.ssl=false
-Dcom.sun.management.jmxremote.authenticate=false
```

Wurden diese Parameter beim Starten der JVM gesetzt, so kann über einen JSR-160 kompatiblen Connector Server auf RMI Basis auf den *MBean Server* zugegriffen werden. Dafür muss die beim Prozess-Start verwendet Port-Nummer und die Endung der JMXServiceURL bekannt sein. Standardmäßig endet die URL mit dem String „`jmxrmi`“. Die Verbindung erfolgt über das RMI Protokoll. Somit ergibt sich folgende URL für die Verbindungsaufnahme zum *MBean Server*:

```
service:jmx:rmi:///jndi/rmi://<host>:<port>/jmxrmi
```

5.1.3. JBoss 4.0

Der JBoss Applikationsserver verwendet JMX als Design Pattern quer durch seine interne Implementierung. Daher ist es möglich, ohne Änderung der Konfiguration des Applikationsservers auf eine große Anzahl von *MBeans* zuzugreifen.
Der Remote Zugriff auf den *MBeanServer* erfolgt über das Management EJB nach dem JSR-77 Standard, das unter dem Namen „`jmx/rmi/RMIAdaptor`“ im JNDI Verzeichnisdienst abgelegt ist.
Der Zugriff auf die JNDI Verzeichnisdienst erfolgt über das JNP Protokoll, standardmäßig auf Port 1099. Der Name der *Context Factory* Klasse ist:

```
org.jnp.interfaces.NamingContextFactory
```

Der JBoss Applikationsserver unterstützt auch schon in früheren Versionen JMX, dies wurde jedoch im Rahmen der Arbeit nicht getestet.

5.1.4. BEA WebLogic Application Server

Verbindungsaufnahme zum BEA WebLogic Application Server [siehe BEA02] wurde mit den Versionen 8.0, 8.1 und 9.0 getestet. Aktuell ist bereits Version 9.2 verfügbar.
In den 8.x Versionen erfolgt die Verbindungsaufnahme zum *MBean Server* ausschließlich über das Management EJB. Über das T3 Protokoll und der Klasse `weblogic.jndi.WLInitialContextFactory` als *Context Factory* wird auf das in JNDI Verzeichnisdienst unter dem Bezeichner `weblogic/management/adminhome` abgelegte MBean zugegriffen.
Die 9.x Versionen sind bezüglich JMX Bootstrapping rückwärtskompatibel mit den 8.x Versionen. Zusätzlich wird nun auch der JSR-160 Standard bezüglich Verbindungsaufnahme zum *MBean Server* unterstützt. Das dabei verwendete Protokoll ist, wie schon bei *MBean Servern* von SUN JVMs, RMI. Allerdings wird der JNDI Verzeichnisdienst über das IIOP Protokoll kontaktiert, der per Default auf Port 7001 auf Verbindungen wartet. Somit ergibt sich folgendes Schema für die JMXServiceURL:

```
service:jmx:rmi:///jndi/iiop://<host>:<port>/weblogic.managemen
t.mbeanservers.runtime
```

Allerdings ist vorher eine Änderung der Konfiguration des Applikations- Servers notwendig [vgl. BEA01, S.53]. Ansonsten ist der Zugriff über IIOP für den Standardbenutzer nicht erlaubt.

5.1.5. Oracle Application Server

Der Oracle Application Server [siehe ORA01] wurde mit einer Developer Preview der Version 10.1.3 getestet. Der JNDI Verzeichnisdienst wird über das ORMI Protokoll auf Port 23791 erreicht, dabei wird die Klasse

```
com.evermind.server.ApplicationClientInitialContextFactory
```

als *Context Factory* verwendet. An die JNDI URL muss das Kürzel „default“ als Postfix hinzugefügt werden. Im JNDI Verzeichnisdienst ist unter dem Namen `java:comp/env/ejb/mgmt/MEJB` das Management EJB mit einer Referenz auf den *MBean Server* abgelegt. Für die Verbindungsaufnahme zum *MBean Server* wird üblicherweise ein gültiger Benutzername (admin) mit Kennwort benötigt.

5.1.6. IBM WebSphere Application Server

Der IBM WebSphere Application Server [IBM01] wurde in der Version 6.0 getestet. Aktuell ist die Version 6.1 verfügbar. Die Verbindungsaufnahme zum JNDI Verzeichnisdienst erfolgt bei WebSphere über das IIOP Protokoll. Neben der eigenen *Context Factory*, `com.ibm.websphere.naming.WsnInitialContextFactory`, wird hier sogar ein eigenes JDK von IBM ausgeliefert. Um unangenehmen Problemen, wie Klassenkonflikten zwischen JMX Client und *MBean Server* aus dem Weg zu gehen, empfiehlt es sich, den JMX Client unter dem IBM JDK laufen zu lassen. Der Standard Port ist 2809, und muss im Standardfall in der JNDI URL nicht explizit angegeben werden. Das Management EJB ist unter dem Namen `ejb/mgmt/MEJB` im JNDI Verzeichnisdienst abgelegt.

5.1.7. Borland Application Server

Der Borland Application Server wurde mit der Version 6.6 getestet. In der Dokumentation sind verschiedene Möglichkeiten des Bootstrapping beschrieben. Am einfachsten erwies sich die Möglichkeit, über das proprietäre, jedoch JSR-160 kompatible, CORBALOC Protokoll eine Verbindung zum *MBean Server* zu erstellen. Beim Starten des JMX Client müssen einige VM Parameter gesetzt werden:

```
-Dorg.omg.CORBA.ORBClass=com.inprise.vbroker.orb.ORB
-Dorg.omg.CORBA.ORBSingletonClass=com.inprise.vbroker.orb.ORBSingleton
-Djavax.rmi.CORBA.StubClass=com.inprise.vbroker.rmi.CORBA.StubImpl
-Djavax.rmi.CORBA.UtilClass=com.inprise.vbroker.rmi.CORBA.UtilImpl
```

```
-Djavax.rmi.CORBA.PortableRemoteObjectClass=
    com.inprise.vbroker.rmi.CORBA.PortableRemoteObjectImpl
-Dvbroker.agent.enableLocator=false
```

Diese Parameter bewirken unter anderem, dass der korrekte (herstellerspezifische) CORBA ORB verwendet wird.

Das URL Postfix ist ein variabler String, der den Namen der auf dem Applikationsserver laufenden Applikation beinhaltet. Bei der getesteten Version, lautete das URL Postfix für die J2EE Sample Application die auf einem Rechner mit Namen „Dylan“ läuft: `j2eeSample_DYLAN_MyPartition.`

5.2. Konfigurationsdaten

5.2.1. Kriterien bei der Wahl der Konfigurationsdaten

Die zwei wesentlichen Kriterien bei der Wahl der Konfigurationsdaten sind Vollständigkeit und Einfachheit.
Der JMX Monitoring Client muss einerseits für verschiedenste Anwendungsgebiete konfigurierbar sein und genügend Konfigurationsmöglichkeiten bieten, so dass die API weitgehend frei von hardkodierten herstellerspezifischen Einstellungen bleibt. So bieten die JMX Agents der verschiedenen Hersteller oft mehrere Möglichkeiten bezüglich des Remote-Zugriffs (MEJB über JNDI, Connector Server über JSR-160) und eine Auswahl der darunter liegenden Protokolle (RMI, IIOP, JMXMP). Ziel ist es, ein möglichst großes Gebiet von Anwendungsgebieten abdecken zu können.
Andererseits muss die Konfiguration einfach genug bleiben, so dass auch Einsteiger sich schnell zurechtfinden können. Es wird versucht, in möglichst vielen Bereichen der Konfiguration Standardeinstellungen vorzuschlagen, so dass bei einer Standardinstallation eines Applikationsservers nur mehr ein Minimum von Einstellungen geändert werden muss, um ein erstes Monitoring zu ermöglichen.

5.2.2. Arten der Konfigurationsdaten

Host Name, Port

Rechnername und Port Konfiguration sind die grundlegendsten Konfigurationsmöglichkeiten des Monitoring Clients. Egal welche Art der Verbindungsaufnahme zum Applikationsserver gewählt wird, Host Name und Port Nummer müssen immer konfiguriert werden.

Protokoll

Beim Großteil der vom JMX Monitoring Client API unterstützten Anwendungsserver, erfolgt das Bootstrapping über JNDI. In einem JNDI kompatiblen Verzeichnisdienst ist ein EJB registriert, das eine Referenz auf den *MBean Server* zurückliefern kann. Häufig wird der Name Management EJB oder kurz MEJB verwendet. Das für das JNDI Bootstrapping verwendete Protokoll und die *InitialContextFactory* sind herstellerspezifisch.

Seit der Standardisierung durch die JMX Remote API im JSR-160, wie in Kapitel 1.3.8 beschrieben, ist eine alternative Möglichkeit des Bootstrapping gegeben. Der JMX Monitoring Client unterstützt RMI, IIOP und JMXMP Konnektoren. Die Service URL wird entweder direkt angegeben, oder wiederum über spezielle Protokolle wie JNDI oder CORBALOC referenziert. Der JSR-160 Standard wird vor allem zum Verbindungsaufbau mit SUNs *Java Virtual Machines* ab Version 1.5 verwendet. Der JSR-160 Standard ermöglicht eine einfachere Verbindungsaufnahme zum *MBean Server*. So sind zum Beispiel keine herstellerspezifischen Bibliotheken für die *Initial Context Provider* Klasse mehr notwendig.
Zukünftig sollte dieser Standard auch bei den verschiedenen Anwendungsservern eine größere Rolle spielen und das Bootstrapping über JNDI nach und nach verdrängen. Dieser Trend ist daran ersichtlich, dass neuere Versionen verschiedener Applikationsserver immer mehr auf JSR-160 setzen (siehe zum Beispiel BEA WebLogic Applikation Server, Kapitel 5.1.4).

URL Postfix

Bei manchen Anwendungsservern (z.B. Oracle Application Server) wird pro laufenden Server-Prozess ein eigener *MBean Server* zur Verfügung gestellt. Der Name des Prozesses ist bei der Verbindungsaufnahme via JNDI als letzter Teil der JNDI URL hinzuzufügen.
Auch bei Verbindungsaufnahme mittels JSR-160 Standard muss an die JMXServiceURL häufig noch ein Postfix hinzugefügt werden. So muss beim Monitoring von SUN JVMs der Versionen 1.5 oder 1.6 via JSR-160 die JMXServiceURL mit dem Postfix “jmxrmi“ terminiert werden.

MEJB Name

Bei Verbindungsaufnahme via JNDI ist im JNDI Verzeichnisdienst des Applikationsservers ein spezielles EJB registriert, das Zugriff auf den *MBean Server* ermöglicht. Der Name dieses MEJB unterscheidet sich von Installation zu Installation und muss somit konfigurierbar sein.

Sicherheitseinstellungen

Üblicherweise sind Benutzername und Passwort für den Verbindungsaufbau zu einem Java Naming Service (JNDI) notwendig.

Klassenpfad

Die Konfiguration des Klassenpfads gehört zu den aufwendigsten Tätigkeiten beim J2EE Bootstrapping. Je nach Hersteller sind teilweise zahlreiche Java Archive aus verschiedensten Verzeichnissen im Klassenpfad nötig. Zusätzlich müssen oft noch Einstellungsdateien[10] über den Klassenpfad gelesen werden.

Java Version

Oft ist es notwendig, dass der RMI Server, der sich zum JMX Agent verbindet, unter einer bestimmten Java Version läuft. Grundsätzlich wird von der JMX Client API zumindest JDK 1.4 vorausgesetzt. Um JMX Agents vom JDK 1.5 bzw. 1.6 monitoren zu können, macht es Sinn, ebenso den RMI Server unter derselben Version laufen zu lassen. Fürs das Monitoring des IBM Websphere Applikationsservers empfiehlt sich, anstatt des JDK von SUN die IBM Version von Java zu verwenden.

5.2.3. Überlegungen zur Konfiguration des Klassenpfades

Grundsätzlich stellt sich bei der Konfiguration des Klassenpfades die Frage, ob die Java Archive auf dem Rechner mit Applikationsserver bleiben oder ob sie auf den Client Rechner kopiert werden müssen. Bei den in den Punkten 3.2.2 und 3.2.3 dargestellten Architekturen ist es möglich, dass nur das Monitoring Tool auf dem Client Rechner läuft. Der RMI Server bzw. der ORB Server laufen auf demselben Rechner wie der Applikationsserver. Somit ist kein Kopieren der Java Archive auf den Client Rechner notwendig. Allerdings müssen bei diesem Ansatz Programmteile des Monitoring Tools auf dem Rechner des Applikationsservers installiert werden. Da nicht auszuschließen ist, dass bei kommerziellen Systemen dies aus Sicherheitsgründen untersagt oder stark eingeschränkt ist, ist von diesem Ansatz für ein kommerzielles Monitoring Tool wie SilkPerformer Performance Explorer abzuraten.

Ein einfacherer Ansatz ist es, die Java Archive des Applikationsservers auf den Client Rechner zu kopieren. Dies kann manuell oder, falls vorhanden, über ein Sourcecontrol-System passieren. Meistens ist es ausreichend, den Inhalt des Lib-Verzeichnisses zu kopieren. In extremen Fällen, wo Java Archive aus verschiedensten

[10] engl.: properties files

Verzeichnissen benötigt werden, wie zum Beispiel beim JBoss 4.0 Applikationsserver, macht es Sinn, gleich den gesamten Installationsordner zu kopieren.

Ein sehr eleganter Lösungsansatz ist es, nicht das Lib-Verzeichnis zu kopieren sondern über eine UNC-Freigabe[11] zur Verfügung zu stellen. Somit lässt sich die Monitoring Konfiguration problemlos von einem Rechner auf den anderen übertragen. Allerdings muss dafür gesorgt werden, dass die notwendigen Rechte und technischen Vorraussetzungen vorhanden sind, auf die UNC-Freigabe zuzugreifen.

Zuletzt muss entschieden werden, wie die einzelnen Java Archive zum Klassenpfad hinzugefügt werden. Die einfachste, wenn auch unkomfortabelste Lösung ist es, es dem Anwender zu überlassen, welche Archive zum Klassenpfad hinzugefügt werden sollen und welche nicht.

Mehr Komfort ist geboten, wenn vorkonfigurierte Profile für die verschiedenen Applikationsserver angeboten werden. Für jede Hauptversion[12] eines Applikationsserver wird ein Konfigurationsfile im XML Format bereit gestellt, das die Liste der benötigten Klassenpfad Einträge relativ zum Installations- bzw. Lib-Verzeichnis des Applikationsservers beinhaltet. Im Monitoring Client muss dann nur mehr das passende Konfigurationsfile geladen und das Installationsverzeichnis konfiguriert werden. Der Klassenpfad wird aus den obig genannten Informationen automatisch generiert.

5.2.4. Verwaltung der Standardeinstellungen

Die Standardeinstellungen für die Verbindungsaufnahme zu einem bestimmten JMX Agent sind in Konfigurationsdateien im XML Format abgelegt. Referenziert wird die Konfigurationsdatei über einen Konfigurationsstring, der Anwendungsserver und Versionsnummer beinhaltet.

Zusätzlich zu der Konfigurationsdatei für jede JMX Agent Konfiguration gibt es noch eine allgemeine Konfigurationsdatei, `jmx-config.xml`, die für alle Konfigurationen gültige Einstellungen verwaltet.

---

[11] engl.: UNC share

[12] engl. major release

Die jmx-config.xml Datei beinhaltet eine Liste von JDK Installationen, eine Liste von für alle Konfigurationen gemeinsamen Klassenpfadeinträgen und eine Liste von so genannten `BaseConfigurations`.
Die Liste der JDK Installationen enthält das Installationsverzeichnis, ein Kürzel für den Hersteller und Versionsnummer der Java Installation.

```
<JvmReference>
  <MajorVersion>1.5</MajorVersion>
  <FullVersion>1.5.0_01</FullVersion>
  <Vendor>SUN</Vendor>
  <Homepath>C:\jdk1.5.0_01</Homepath>
</JvmReference>
```

Zu den gemeinsamen Klassenpfadeinträgen zählen die Java Archive, die die RMI Stubs enthalten, die Implementierung der Client API selbst sowie die JMX Referenz-Implementierungen von SUN.

```
<CommonConfiguration>
  <Classpath>
    <Entry>T:\bin\ClassFiles\silk-jmx-common.jar</Entry>
    <Entry>T:\bin\ClassFiles\silk-jmx-jboss_stubs.jar</Entry>
    <Entry>T:\bin\ClassFiles\silk-jmx-oc4j_stubs.jar</Entry>
    <Entry>T:\bin\ClassFiles\silk-jmx-weblogic_stubs.jar</Entry>
    <Entry>T:\bin\ClassFiles\silk-jmx-websphere_stubs.jar</Entry>
    <Entry>T:\BuildUtilities\Lib\Java\JMX_Remote-101\jmxremote.jar</Entry>
    <Entry>T:\BuildUtilities\Lib\Java\JMX-121\jmxtools.jar</Entry>
    <Entry>T:\BuildUtilities\Lib\Java\JMX-121\jmxri.jar</Entry>
    <Entry>T:\BuildUtilities\Lib\Java\j2ee_1.4.jar</Entry>
  </Classpath>
</CommonConfiguration>
```

Die `BaseConfiguration` definiert den Namen der Klasse, mit der der RMI Server gestartet wird und das Java Archiv, das diese Klasse enthält.

```
<BaseConfiguration TypeDescription="Base Configuration Websphere 6.0">
  <MainClass>com.segue.monitoring.jmx.websphere.datasource.RemoteDataSource
  </MainClass>
  <Classpath>T:\bin\ClassFiles\silk-jmx-websphere.jar</Classpath>
</BaseConfiguration>
```

Jede einzelne JMX Agent Konfigurationsdatei enthält Referenzen auf eine JDK Konfiguration und auf eine BaseConfiguration aus der Datei jmx-config.xml. Weiters gibt es in jeder Konfigurationsdatei Standardwerte für Host, Port, Protokoll, URL Postfix bzw. MEJB Name, Sicherheitseinstellungen und Klassenpfad. Falls die

tatsächliche Konfiguration zur Verbindungsaufnahme zum JMX Agent von den Standardwerten abweicht, so müssen diese explizit überschrieben werden.

```
<AgentConfiguration Name="SUN JVM 1.5" DataSourceType="Base Configuration Jvm 1.5">
  <Host>localhost</Host>
  <Port>9998</Port>
  ...
  <JvmInfo>
    <JvmHomePath RecommendedVendor="SUN" RecommendedMajorVersion="1.5" />
    <Classpath />
  </JvmInfo>
</AgentConfiguration>
```

Für eine komplette Auflistung der Standardwerte für die verschiedenen Anwendungsserver, siehe Anhang.
Zurzeit sind Standardeinstellungen für folgende JMX Agents vordefiniert:

- JBoss 4.0
- BEA WebLogic 8.0, 8.1
- BEA WebLogic 9.0 (JNDI)
- BEA WebLogic 9.0 (JSR-160)
- IBM WebSphere 6.0
- Borland Application Server 6.6
- Oracle Application Server 10.1.3
- SUN JVM 1.5, 1.6
- Borland SilkCentral Test Manager 8.5
- BMC Transaction Management Application Response Time 3.0
- dynaTrace Diagnostics 1.6
- dynaTrace Diagnostics 2.0

6. JMX API für C++ basierendes Monitoring

6.1. Problemsstellungen

Sowohl die API für den JMX Monitoring Client, als auch die API für den JMX Browser sollen für C++ basierende Tools zur Verfügung stehen. Durch Verwendung des *Java Native Interface* (JNI) wird eine C++ Schnittstelle über die bestehende Java API gestellt. Bei Verwendung des *Java Native Interface* müssen einige wichtige Gesichtspunkte beachtet werden, die in diesem Kapitel behandelt werden.

6.1.1. Verwenden einer JVM in einem C++ Prozess

Um Java Methoden in einem C++ Prozess aufrufen zu können, müssen zwei Bedingungen erfüllt sein. Erstens muss eine JVM in den C++ Prozess geladen werden. Zweitens muss ein Zeiger auf die Java Umgebung (*JNIEnv Interface* Pointer) verfügbar sein. Die Begründung liefert *„... the JNIEnv Interface pointer, points to a location that contains a pointer to a function table. Each entry in the function table points to a JNI function. Native methods always access data structures in the Java virtual machine through one of the JNI functions“* [LIA99, S. 22].

6.1.2. Abbildung von Java Datentypen in C++

Ein komplexes Java Interface, wie es die im Rahmen dieser Arbeit entwickelte JMX API darstellt, kann nicht so einfach in C++ dargestellt werden. Es muss dafür gesorgt werden, dass die Parameter und Rückgabe Werte korrekt von C++ nach Java und umgekehrt übergeben werden. Weiters muss der Fehlerbehandlung besondere Aufmerksamkeit geschenkt werden.

6.2. Lösungsansätze

6.2.1. Laden der JVM

Fürs das Laden der *Java Virtual Machine* in einen C++ Prozess sind zumindest zwei Parameter notwendig, die Version der JVM und das Installationsverzeichnis der Java Entwicklungsumgebung. Anhand der Java Version wird definiert, wie die Parameter

zum Laden der JVM strukturiert sein müssen. Anhand des Installationsverzeichnisses kann der Pfad zur *jvm.dll* erstellt werden. Die *jvm.dll* enthält unter anderem die Methode `JNI_CreateJavaVM`, welche, wie in Kapitel 6.2.2 beschrieben, den *JNIEnv Interface* Zeiger bereitstellt.
Weiters müssen alle wichtigen Einstellungen wie die Parameter der *Virtual Machine* und der Klassenpfad definiert werden.
Ist damit zu rechnen, dass die JVM wiederum C++ Methoden aus Dynamic Link Libraries aufruft, so muss der Java Library Path korrekt gesetzt werden.
In der JMX Monitoring API wird nur mit Java Versionen größer gleich 1.4 gearbeitet. Daher können problemlos die mit JDK 1.2 eingeführten JNI Vereinfachungen verwendet werden. Somit werden die Parameter zum Laden der JVM über die Struktur `JavaVMInitArgs` gesetzt, das ein Array von `JavaVMOption` Strukturen enthält [vgl. GOR98, S. 333]. Gordon beschreibt: *„The version field of the JavaVMInitArgs must be set to 0x00010002 to use the new options format"* [GOR98, S.333]. Die einzelnen `JavaVMOptions` beschreiben zum Beispiel den Klassenpfad, mit dem die JVM initialisiert werden soll und zusätzliche *System Properties*, die für den JVM Prozess gelten sollen.
Die beiden Strukturen sowie andere wichtige Konstanten des *Java Native Interface* sind in der Datei `jni.h` definiert, das bei jeder JDK Installation im Include-Verzeichnis mitgeliefert wird. Hier ein Auszug dem `jni.h` von SUN JDK 1.4.2_12:

```
typedef struct JavaVMInitArgs {
    jint version;

    jint nOptions;
    JavaVMOption *options;
    jboolean ignoreUnrecognized;
} JavaVMInitArgs;

typedef struct JavaVMAttachArgs {
    jint version;

    char *name;
    jobject group;
} JavaVMAttachArgs;
```

Um die Konfiguration komfortabel zu halten, sind die Konfigurationsparameter zum Starten der JVM in einer Konfigurationsdatei unter dem Namen `jmxDataSourceBridge.xml` abgelegt. Die Konfigurationsdatei enthält den Pfad zum Installationsverzeichnis und die Versionsnummer des JDK, Klassenpfad Einträge und Optionen der *Virtual Machine.*

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<JvmConfigfile javahome=" C:\Java\jdk1.5.0_07 " version="150">
  <Classpath>
      <Item value="T:\bin\release\ClassFiles\silk-jmx-common.jar"/>
      <Item value="T:\bin\release\ClassFiles\silk-jmx-jboss_stubs.jar"/>
      <Item value="T:\bin\release\ClassFiles\silk-jmx-oc4j_stubs.jar"/>
      <Item value="T:\bin\release\ClassFiles\silk-jmx-weblogic_stubs.jar"/>
      <Item value="T:\bin\release\ClassFiles\silk-jmx-websphere_stubs.jar"/>
      <Item value="T:\bin\release\ClassFiles\jmx\jmxremote.jar"/>
      <Item value="T:\bin\release\ClassFiles\jmx\jmxtools.jar"/>
      <Item value="T:\bin\release\ClassFiles\jmx\jmxri.jar"/>
      <Item value="T:\bin\release\ClassFiles\j2ee\j2ee_1.4.jar"/>
      <Item value="T:\bin\release\ClassFiles\Apache-Commons\commons-lang-2.0.jar"/>
      <Item value="T:\bin\release\ClassFiles\Apache-Commons\commons-logging-
        1.0.3.jar"/>
      <Item value="T:\bin\release\ClassFiles\jdom.jar"/>
  </Classpath>
  <Options>
    <!-- Enable following options for debugging -->
    <!--Option value="-Xdebug"/>
    <Option value="-Djava.compiler=NONE"/>
    <Option value="-
      Xrunjdwp:transport=dt_socket,address=falco:4910,server=y,suspend=n"/-->
  </Options>
</JvmConfigfile>
```

6.2.2. Bereitstellen des JNIEnv Interface Zeigers

Der *JNIEnv* Zeiger wird beim Aufruf der Funktion zum Laden der JVM, `JNI_CreateJavaVM`, retourniert. Der *JNIEnv* Zeiger ist nur für den jeweiligen Thread gültig und kann nicht an andere Threads übergeben werden. So schreibt Liang: „*A JNIEnv pointer is only valid in the thread associated with it. You must not pass this pointer from one thread to another ...*“ [LIA99, S. 93]. Die JVM kann nur einmal pro Prozess geladen werden, der *JNIEnv* Zeiger wird jedoch für jeden Thread separat benötigt. Also muss jeder weitere Java Thread den *JNIEnv* Zeiger von der bereits geladenen JVM abfragen.
Dies bedeutet für die C++ Implementierung der JMX API, dass zunächst einmal festgestellt werden muss, ob schon eine JVM in den C++ Prozess geladen wurde. Wenn durch einen vorhergegangene Benutzung der JMX API bereits eine JVM geladen wurde, so muss mittels der `AttachCurrentThread()` Methode ein gültiger *JNIEnv* Zeiger erzeugt wird.

6.2.3. Konvertierung von Datentypen

Konvertierung von primitiven Datentypen

Das *Java Native Interface* definiert einige primitive Datentypen, welche es erlauben, Werte direkt von C++ Variablen auf Java Variablen und umgekehrt zuzuweisen. So schreibt Liang: „*Argument types in the native method declaration have corresponding types in the native programming languages. The JNI defines a set of C and C++ types that correspond to types in the Java programming language*“ [LIA99, S. 23]. Dies bedeutet, dass der Java Datentyp `int` mit dem C/C++ Datentyp `jint` (und somit einer 32 Bit Integer Zahl) oder der Java Datentyp `float` mit dem C/C++ Datentyp `jfloat` (und somit einer 32 Bit Fließkommazahl) kompatibel ist [vgl. LIA99, S. 23 u. S. 165].

Konvertierung von Strings

Die Abbildung von Strings unterscheidet sich in Java fundamental von C++. Eine Konvertierung von Java Strings (und deren Pendant im native Code, `jstring`) in C/C++ Strings und umgekehrt ist absolut notwendig. So erklärt Liang: „*You cannot use a jstring as a normal C string.*“ [LIA99, S. 24].

Das Java Native Interface bietet die Methode `GetStringUTFChars`, um einen Unicode String in einen C/C++ String ins UTF-8 Format zu konvertieren. So schreibt Liang über die Methode `GetStringUTFChars`: „*It converts the `jstring` reference, typically represented by the Java virtual machine implementation as a Unicode sequence, into a C String represented in the UTF-8 format*“ [LIA99, S. 25]. Die Methode `GetStringChars` liefert die Unicode Zeichenkette zurück des Java Strings zurück.
Um Internationalisierung, das bedeutet eine länderspezifisches Kodierung von Zeichenketten, zu unterstützen, muss die Konvertierung von Strings selbst implementiert werden. Die Methode `getBytes` der Klasse `java.lang.String` liefert die Zeichen eines Strings als Native String, das heißt codiert für die vom Betriebssystem festgelegten Ländereinstellungen. So schreibt Liang: „*Use the `String.getBytes` method to convert a `jstring` to the appropriate native encoding*“ [LIA99, S. 100]. Die Methode muss über Reflexion im Native Code aufgerufen werden.
In der umgekehrten Richtung, also das zum Erzeugen eines Java Strings im Unicode Format aus einem Native String, wird der String Konstruktor mit einem Byte Array als Parameter, ebenfalls über Reflexion, verwendet. So beschreibt Liang: „*Use the `String(byte[] bytes)` constructor to convert a native string into a jstring*“ [LIA99, S. 99].

Konvertierung von komplexen Datentypen

Komplexe Java Objekte müssen in C++ über einen Introspektions-Mechanismus erzeugt werden. Zunächst muss mit der `FindClass()` Methode eine Referenz auf die Java Klasse angelegt werden. Der Datentyp für Referenzen auf Java Klassen ist `jclass`. Die Referenz auf die Java Klasse wird benötigt, um die Referenz auf den Konstruktor der Klasse vom Datentyp `jmethodID` zu erhalten. Dazu wird die JNI Methode `GetMethodID` verwendet. Als Name der Konstruktor Methode, die als Parameter für die `GetMethodID()` Methode benötigt wird, ist der konstante String „<init>“ vorgeschrieben. Als weiterer Parameter ist, zur Unterscheidung von mehreren Konstruktoren derselben Klasse, die Signatur der Konstruktor Methode notwendig. Schließlich erlaubt die JNI Methode `NewObject` das Erzeugen eines komplexen Java Objektes. Als Parameter werden Referenzen auf die Klasse, der

Konstruktor und eventuelle Parameter des Konstruktors benötigt [vgl. LIA99, S. 51ff].

6.2.4. Zugriff von Java Methoden und Variablen

Zugriff auf Java Methoden, Instanz und Klassenvariablen im C/C++ Code ist ebenfalls nur mittels Reflexion möglich.
Beim Aufruf von Instanz Methoden muss, ähnlich wie beim Aufruf eines Konstruktors, zuerst die Referenz auf die Java Klasse und die Referenz auf die Methode durch die JNI Methoden `FindClass` und `GetMethodID` erzeugt werden. Anschließend wird mit der JNI Methode `Call<Type>Method` die gewünschte Java Methode aufgerufen, wobei der Platzhalter `<Type>` im Methodennamen für den retournierten Datentyp der Methode, also zum Beispiel `void`, `int` oder `float`, steht [vgl. LIA99, S. 46ff u. 184].
Die Parameter der aufgerufenen Methode werden als Argumentliste beim Befehl `Call<Type>Method` hinten angefügt. Dabei muss dafür gesorgt, dass die einzelnen Elemente der Argumentliste im korrekten Java Datentyp vorliegen.
Der Aufruf von statischen Methoden funktioniert ähnlich. Die JNI Methoden `GetStaticMethodID` und `CallStatic<Type>Method` weisen dieselben Parameter wie ihre nicht statischen Pendants auf.
Zugriff auf Instanzvariablen bzw. Klassenvariablen ist über das *Java Native Interface* ebenfall möglich. Die Methoden `GetFieldID` bzw. `GetStaticFieldID` liefern eine Referenz auf die Variable, deren Inhalt dann über die Methode `Get<Type>Field` bzw. `GetStatic<Type>Field` verfügbar ist. In der Praxis ist bei der korrekten Verwendung von Getter und Setter Methoden üblicherweise der direkte Zugriff auf Variablen nicht relevant, so auch in dieser Arbeit.

6.2.5. Caching von Java Referenzen

Das Erzeugen von Referenzen auf Java Klassen und Methodenbeschreibungen im native Code ist rechenintensiv. So schreibt Liang: „*Obtaining field and method Ids requires symbolic lookups based on the name and descriptor of the file or method. Symbolic lookups are relatively expensive*“ [LIA99, S.53]. Die Referenzen selbst sind jedoch standardmäßig nur für den aktuellen Codeblock gültig. Danach wird der Speicherbereich wieder freigegeben. Man spricht von lokalen Referenzen.

Daher empfiehlt es sich beim Arbeiten mit JNI, besonders häufig verwendete Methoden- und Klassenreferenzen zu cachen.
Um eine Referenz auf ein Java Objekt außerhalb des aktuellen Codeblocks verfügbar zu machen, um zum Beispiel die Referenz in einer globalen Variable zu cachen, muss die JNI Funktion `NewGlobalRef` aufgerufen werden. Ab diesem Zeitpunkt ist der Entwickler selbst für die spätere Speicherfreigabe durch die JNI Funktion `DeleteGlobalRef` verantwortlich.
Sowohl lokale als auch globale Referenzen auf Java Objekte verhindern, dass das referenzierte Java Objekt durch die Java Garbage-Collection[13] freigegeben wird. So schreibt Liang: „*Like a local reference, a global reference ensures that the referenced object will not be garbage collected* " [LIA99, S. 64].

6.2.6. Fehlerbehandlung

Die Fehlerbehandlung beim, wie in Kapitel 6.2.4 beschriebenen, Aufruf von komplexen Java Methoden aus dem C/C++ Code ist ein wichtiges Thema um die Stabilität des Programms zu gewährleisten. So schreibt Liang: „*Proper exception handling is sometimes tedious but is necessary in order to produce robust applications*" [LIA99, S.76]. Wenn beim Aufruf einer Java Methode über das *Java Native Interface* eine Ausnahme geworfen wird, so ist die in den nativen Prozess geladene JVM in einem instabilen Zustand. Es kann erst dann wieder mit der Verwendung von Methoden des JNI fortgefahren werden, wenn die JVM durch Aufruf von der Methode `ExceptionClear` wieder in einen stabilen Zustand gebracht wurde. So warnt Liang: „*Calling most JNI functions with a pending exception – with an exception you have not explicitly cleared – may lead to unexpected results*" [LIA99, S. 78].
Gordon beschreibt ebenfalls in einem detaillierten Beispiel die Behandlung von Java Exceptions im C/C++ Code [vgl. GOR98, S. 128ff]. Allerdings missachtet er die von Liang aufgestellten Warnungen und versucht den Text der Ausnahme zu erhalten, bevor er die JNI Methode `ExceptionClear` aufruft [vgl. GOR98, S. 130]. Dies hat sich in der praktischen Anwendung im Rahmen dieser Arbeit als Fehler herausgestellt und es kam zu unberechenbaren Programmabstürzen. Erst nach Umstrukturierung des

[13] Dt.: Freispeichersammlung

Codes nach den Regeln von Liang konnte die gewünschte Stabilität des Codes erreicht werden.

6.3. C++ Schnittstellen Beschreibung JMX Monitoring API

6.3.1. Allgemeines

Beim Design der C++ Schnittstelle der JMX Monitoring API gibt es verschiedene Strategien. Die intuitivste Lösung ist es, die Java Schnittstelle möglichst identisch in C++ abzubilden. Da jedoch die in Kapitel 3.3.5 beschriebene Java Schnittstelle komplexe Java Objekt als Parameter enthält und die Konvertierung von komplexen Datentypen, wie in Kapitel 6.2.3 beschrieben, nur über Introspektions-Mechanismen möglich ist, wäre eine solche Abbildung der Schnittstelle sehr aufwendig.
Eine alternative Lösung ist es, alle Konfigurationsparameter in einem einzigen String abzubilden und diese in einem Aufruf an den Java Client zu übertragen. Der Java Teil der Schnittstelle sorgt dafür, dass der Konfigurationsstring wieder in die einzelnen Teile zerlegt wird. Alle weiteren Methodenaufrufe würden auf diese Konfigurationsparameter zurückgreifen.
Eine weitere Vereinfachung der Schnittstelle wird dadurch erreicht, dass Parameter, die von einem Benutzer üblicherweise nicht geändert werden, in einer Konfigurationdatei ausgelagert werden. Das folgende Kapitel 6.3.2 zeigt den Aufbau des Konfigurationsstrings.

6.3.2. Einfache C++ Schnittstelle unter Verwendung eines Konfigurationsstrings

Im Produkt SilkPerformer Performance Explorer werden Datenquellen durch einen einzigen String, dem Performance Data Colletion Engine (PDCE) String beschrieben. In Anlehnung an die Grammatik für PDCE Strings für andere Protokolle wie SNMP entstand die Grammatik für den JMX PDCE String.
Der JMX PDCE String wird, notiert in Erweiterter Backus-Naur Form (EBNF), wie folgt definiert:

```
<PDCE String> = "JMX:" <properties> "@" <host> ":" <port> ;
<properties> = {<name> "=" <value> ";"} ;
```

```
<value> = "'" [<text>] "'" ;

<name> = <required attribute> | <optional attribute> ;

<required attribute> = "AgentConfigurationName" | "Url" | "MBeanName"
         | "MBeanAttributeName" ;

<optional attribute> = "MEJBName" | "JNDIContextFactory" | "Username"
         | "Password" | "MBeanAttributeDetail" | "ComplexStatsIndex"
         | "ComplexStatsMode" | "TableRowKey" | "TableColumnName" ;

<text> = {<letter> | <special> | "$(host)" | "$(port)"} ;

<host> = <letter> {<letter>} ;

<port> = <number> {<number>} ;

<letter> = "a".."z" | "A".."Z" | <number> ;

<number> = "0".."9" ;

<special> = " " | ":" | "/" | "_" | "-" ;
```

Der PDCE String beginnt also mit dem Präfix „JMX:“. Danach folgt eine Liste von Properties in der Form von Name/Wert Paaren. Abgeschlossen wird der String durch Rechnername und Port, jeweils durch eigene Trennzeichen separiert.

Die Properties sind in der EBNF als optional gekennzeichnet, um zum Ausdruck zu bringen, dass für Properties, die im PDCE String fehlenden Standardwerte aus dem Konfigurationsfile gelesen werden können. Es ist aber grundsätzlich notwendig, zumindest die Parameter `AgentConfigurationName`, `Url`, `MBeanName` und `MBeanAttributeName` zu definieren. Da es mehrere Konfigurationsfiles für die verschiedenen JMX Agent Konfigurationen vorgesehen sind, spezifiziert das Attribut `AgentConfigurationName` die jeweilige Konfigurationsdatei, die verwendet werden soll.

Weiters sind zahlreiche Properties, die nur bestimmte komplexe Attribute betreffen, zum Beispiel das Property `MBeanAttributeDetail`, optional. Für Details zu den verschiedenen komplexen Attributen siehe Kapitel 4.4.

Eine weitere Besonderheit stellt die Abbildung der URL im PDCE String dar. Je nach Art der Verbindungsaufnahme zum *MBean Server* wird eine *JMX Service URL* bei JSR-160 bzw. eine *Naming Provider URL* bei JNDI benötigt. Weiters ist es aus Gründen der Rückwärtskompatibilität notwendig, Rechnername und Port explizit im PDCE String anzuführen. Um eine redundante Speicherung von Rechnername und Port zu vermeiden, können in der URL auch die Platzhalter `$(host)` und `$(port)`

verwendet werden. Diese werden dann im Java Teil der JMX Monitoring API automatisch ersetzt.

6.3.3. Beispiel eines Konfigurationsstrings

Folgender Konfigurationsstring definiert eine JMX Monitor, der zum *MBean Server* einer SUN JVM auf dem lokalen Rechner auf Port 9997 verbindet und vom MBean „java.lang:type=Memory“ das Attribut „HeapMemoryUsage“ abfragt. Das Attribut stellt einen komplexen Datentyp dar. In diesem Beispiel wird nur der Wert „used“ des Attributs abgefragt.

```
JMX:AgentConfigurationName='SUN JVM 1.5';
Url='service:jmx:rmi:///jndi/rmi://$(host):$(port)/jmxrmi';
MEJBName=''; JNDIContextFactory=''; Username=''; Password='';
MBeanName='java.lang:type=Memory';
MBeanAttributeName='HeapMemoryUsage'; MBeanAttributeDetail='used';
@localhost:9997
```

6.3.4. C++ Schnittstellenbeschreibung der JMX Monitoring API

Die `Init()` Methode lädt, wie in Kapitel 6.2.1 beschrieben, eine JVM in den Prozess, bzw. falls schon ein JVM im Prozess vorhanden ist, wird der *JNIEnv* Zeiger bereitgestellt (siehe Kapitel 6.2.2). Danach werden die Referenzen auf die `java.lang.String` Klasse sowie die Basis Klasse der Monitoring Client API, `RemoteDataSourceClient`, erzeugt und gecached. In weiterer Folge wird über den JNI Inspektions-Mechanismus die Referenz auf die `getInstance()` Methode der `RemoteDataSourceClient` Klasse erzeugt. Beim anschließenden Aufruf der Methode werden, wie in Kapitel 6.3.2 beschrieben, die Konfigurationsdaten in einem einzigen String an die Java API übergeben. Dazu muss davor der Konfigurationsstring in eine Java String umgewandelt werden. Der Methodenaufruf retourniert eine Referenz auf das erzeugt Java Objekt der Klasse `RemoteDataSourceClient`, die für die weitere Verwendung, um eine Speicherfreigabe zu vermeiden, in eine globale Referenz umgewandelt wird.

Die Methoden `Deploy()` zum Starten des Java Prozesses des RMI Server (siehe Kapitel 3.3.1), `Connect()` zum Verbinden zum RMI Server, `Bootstrap()` zum Verbinden zum *MBean Server* (siehe Kapitel 3.3.2) sind somit auf der C++ Seite

parameterlos und verwenden auf der Java Seite die Parameter aus der `Init()` Methode bzw. aus den Konfigurationsdateien. Über Reflexion werden die Referenzen auf die entsprechenden Java Instanz Methoden erzeugt und auf die globale Referenz der `RemoteDataSourceClient` Klasse angewendet.

Die Method `GetAttribute()` ist in zwei Varianten verfügbar. Die erste Variante ist parameterlos und liefert das vom PDCE String definierte *MBean* Attribut. Die zweite Variante übergibt den *ObjectName* des *MBeans* und den Attributnamen, um die Voreinstellung durch den PDCE String überschreiben zu können. Beide Methoden übergeben eine Referenz auf eine String Container Klasse, in der im Erfolgsfall der Attribut Wert als String retourniert wird.

Die Methode `IsAlive()` dient zum überprüfen des RMI Server Status.

Die Methode `ShutDown()` wird für das in Kapitel 3.3.4 beschriebene Verwaltung der Referenzen auf den RMI Server benötigt.

```
class CJmxDataSourceBridge
{
public:
  CJmxDataSourceBridge();
  ~CJmxDataSourceBridge();

  void Init(const char* csPdceString);
  void Deploy();
  void Connect();
  void Boostrap();
  bool GetAttribute(const char* sObjectName,
                    const char* sAttributeName,
                    CStr& rsMeasure);
  bool GetAttribute(CStr& rsMeasure);
  bool IsAlive();
  void ShutDown();
};
```

7. Resümee

Historischer Rückblick auf JMX

Die Java Bibliothek JMX spielt eine immer wichtigere Rolle für das Management und Monitoring von Applikationsservern und Java Anwendungen. Der Großteil der Hersteller von Applikationsservern unterstützt mittlerweile zumindest in den Grundzügen JMX.

Mit der Definition des JSR-003 Standards im Jahr 1998, begannen Hersteller Daten über JMX Adaptoren für SNMP oder HTML für den Zugriff über das Netzwerk zur Verfügung zu stellen. Reine Java Lösungen für einen entfernten Zugriff auf die Daten, die darauf verzichten, die Daten für ein anderes Protokoll zu konvertieren, gab es anfangs zumeist nur durch proprietäre Implementierungen.

Schon bald erkannte man den Bedarf nach einheitlich definierten Schnittstellen für den Remote Zugriff auf das Herzstück einer JMX Implementierung, den *MBean Server*, was durch den JSR-160 Standard erreicht wurde. Doch dieser Standard wurde von vielen Herstellern nur zögerlich umgesetzt. Der Grund dafür lässt sich nur vermuten, aber Kompatibilitätsprobleme zu den existierenden proprietären Lösungen werden sicherlich auch einen Grund gespielt haben.

Einen enormen Aufschwung erlebte JMX mit der Veröffentlichung von SUNs Java Development Kit 1.5. Der in die Java Runtime inkludierte JMX *MBean Server*, der als Container für die vom Benutzer entwickelten *MBeans* dienen kann und über das JSR-160 Protokoll einfachen Remote Zugriff bietet, scheint einen großen Anreiz zu bieten, auf JMX umzusteigen.

Bootstrapping und Konfiguration

Die eben geschilderte, historisch bedingte Vielzahl an Möglichkeiten, wie ein Remote Zugriff auf die *MBean Server* verschiedener Herstellern von Applikationsservern ermöglicht wird, brachte große Komplexität in die zentralen Themen dieser Arbeit, das Bootstrapping und die Wahl der Konfigurationsdaten. Durch Auslagerung von Konfigurationsparameter in XML Dateien, wurden wichtige Ziele dieser Arbeit, die Einfachheit der Benutzung und Erweiterbarkeit bezüglich weiterer Applikationsserver und Anwendungen erreicht. Die voreingestellten Parameter für einige bedeutende Applikationsserver sollen die Bedienung für Erstbenutzer der JMX Monitoring API

vereinfachen, zugleich aber auch den Wartungsaufwand zur Unterstützung künftiger Versionen von Applikationsserver gering halten.
Die meiner Meinung nach bedeutendsten Applikationsserver am Markt, JBoss, BEA WebLogic, IBM WebSphere und Oracle Application Server verlangen jeweils eine völlig eigenständige Konfiguration für das Bootstrapping, und deren Unterstützung durch diese Arbeit war somit mit hohem Forschungsaufwand verbunden.
Als eine große Hilfe erwies sich dabei der JSR-77 Standard, der den Herstellern von Applikationsservern die Bereitstellung eines speziellen Enterprise Java Beans, das *MEJB*, für den entfernten Zugriff den *MBean Server* vorschreibt. Herstellerspezifisch erfolgt somit nur mehr die Verbindungsaufnahme zum JNDI Verzeichnisdienstes des Applikationsservers. Mit Hilfe dieses Standards kann nun, abgesehen von JDK Kompatibilitätsproblemen[14] bei IBM WebSphere, das *MEJB* als gemeinsame Ausgangsbasis für die JMX Monitoring API verwendet werden.

Interne Architektur

Ein weiteres wichtiges Thema war, eine optimale interne Architektur für die Komponenten der JMX Monitoring API zu finden. Zu den Bedingungen zählte, dass trotz einer Vielzahl an Datenquellen, gleichzeitiges Monitoring und Browsing von JMX Daten verschiedener *MBean Server* in einem einzigen, sofern möglich C++ basierten Tool, möglich sein soll.
Insbesondere die unterschiedlichen Konfigurationen des Klassenpfades machten es notwendig, die Kernbereiche der JMX Monitoring API auf einen oder mehrere dynamisch gestartete externe Prozesse auszulagern. Somit entstand eine über eine RMI Registry verwaltete RMI Client-Server Architektur.
Ein Monitoring Tool, das die in dieser Arbeit entwickelte JMX Monitoring API verwendet, agierte somit als RMI Client und bedient sich eines oder mehrerer RMI Server, die als Proxy zu den einzelnen Datenquellen agieren. Um eine korrekte Freigabe der allokierten Ressourcen auch in Sonderfällen zu gewährleisten, wurden aufwändige Modelle zum Lebenszyklus von RMI Server und Registry entworfen und ein Mechanismus zum Zählen der Referenzen auf den RMI Server implementiert.

[14] IBM WebSphere distributiert ein eigenes Java Development Kit, dass mit dem JDK von SUN nur bedingt kompatibel ist.

Strukturierte Darstellung

Die nächste große Schwierigkeit bestand darin, die Vielzahl der *MBeans*, die in einer völlig flachen Struktur auf dem *MBeans Server* abgelegt sind, in geeigneter Weise zu strukturieren, so dass eine übersichtliche Darstellung in einer graphischen Benutzerschnittstelle möglich ist.
Hier war es leider nicht möglich, eine einheitliche Lösung zu finden, die für alle Applikationsserver ein zufrieden stellendes Ergebnis liefert. Eine Strukturierung nach dem *MBean* Typ und *Domain* erwies sich oft als nicht ausreichend. Daher wurde ein proprietärer Algorithmus für den BEA WebLogic Applikationsserver entwickelt, der eine Baumdarstellung der *MBean* Typen formuliert. Dabei musste jedoch auf die proprietäre WebLogic API zurückgegriffen werden. Für die meisten anderen Applikationsserver erwies sich wiederum der JSR-77 Standard als Basis für eine Baumdarstellung der *MBean* Typen als hilfreich.
Zusätzlich macht es Sinn, erfahrenen JMX Benutzern die *MBeans* über Abfragen auswählen zu lassen. Der JMX Query Mechanismus bietet großartige Suchmöglichkeiten, sofern man bereits eine Vorstellung hat wonach man suchen will.
Aber nicht nur die strukturierte Darstellung der *MBeans*, sondern auch der *MBean* Attribute weist einiges an Komplexität auf. Sowohl der JSR-77 als auch der JSR-003 Standard erlauben die Darstellung von komplexen Attributen, die zum Beispiel Daten in tabellarischer Form enthalten. Die JMX Monitoring API erlaubt den Zugriff auf einzelne Zellen der Tabellen oder einzelne Komponenten von Verbund-Datentypen.

Brücke in die C++ Welt

Für den Bereich des Monitorings spielen oft noch C++ basierende Produkte die entscheidende Rolle. So ist auch in Borlands SilkPerformer Performance Explorer, die Anwendung in der die in dieser Arbeit verfasst JMX Monitoring API eingesetzt wird, die Sprache C++ vorherrschend. Dies muss jedoch keineswegs einen Konflikt dazu bedeuten, dass JMX bekanntlich eine Java Spracherweiterung ist. Das *Java Native Interface (JNI)* bietet die Möglichkeit, eine *Java Virtual Machine (JVM)* in einen C++ Prozess einzubinden, Datentypen zu konvertieren und Java Methoden über Reflexionsmechanismen aus C++ aufzurufen.
Somit entstand eine C++ Schnittstelle für die JMX Monitoring API. Anfang 2007 wurde die erste kommerzielle Version der API als Bestandteil von Borlands

Loadtesting und Monitoring Lösung SilkPerformer 2006 R2 zum Verkauf freigegeben [vgl. BOR02].

8. Literatur

[SUL03] Ben G. Sullins, Mark B. Whipple: JMX in Action, Manninger Pulications Co. 2003, ISBN 1930110561

[KRE03] Heather Kreger, Ward Harold, Leigh Williamson: JAVA™ and JMX, Building Manageable Systems, Addison-Wesley 2003, ISBN 0672324083

[LIA99] Sheng Liang: The Java Native Interface, Programmer's Guide and Specification, Addison-Wesley 1999, ISBN 0201325772

[GOR98] Rob Gordon: Essential JNI: Java Native Interface, Prentice Hall PTR 1998, ISBN 0136798950

[SUN01] SUN Microsystems, Inc.: Java™ Management Extensions Instrumentation and Agent Specification, v1.2, Maintenance Release October 2002 (JSR-003)

[SUN02] SUN Microsystems, Inc., Hans Hrasna: Java™ 2 Platform, Enterprise Edition
Management Specification, JSR-77, Final Release v1.0, June 18, 2002

[SUN03] SUN Microsystems, Inc.: JavaTM Management Extensions (JMXTM) Remote API 1.0 Specification, Final Release October 2003 (JSR-160)

[SUN04] SUN Microsystems, Inc.: Java Naming and Directory Interface™ Application Programming Interface (JNDI API), JNDI 1.2/Java™ 2 Platform, Standard Edition v 1.3, July 14, 1999
(http://java.sun.com/j2se/1.5/pdf/jndi.pdf)

[SUN05] SUN Microsystems, Inc.: J2SE 1.5.0 API Documentation, Class java.lang.Runtime (http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Runtime.html)

[SUN06] SUN Microsystems, Inc.: J2SE 1.5.0 API Documentation, Class java.rmi.registry.LocateRegistry (http://java.sun.com/j2se/1.5.0/docs/api/java/rmi/registry/LocateRegistry.html)

[RFC01] Internet Engineering Task Force: Request for Comments: 2609: Service Templates and Service: Schemes (RFC 2609), June 1999 (http://www.ietf.org/rfc/rfc2609.txt)

[RFC02] Internet Engineering Task Force: Request for Comments: 1630: Universal Resource Identifiers in WWW (RFC 1630), June 1994 (http://www.ietf.org/rfc/rfc1630.txt)

[RFC03] Internet Engineering Task Force: Request for Comments: 1034: DOMAIN NAMES - CONCEPTS AND FACILITIES (RFC 1034), November 1987 (http://www.ietf.org/rfc/rfc1034.txt)

[BEA01] BEA WebLogic Server: Developing Manageable Applications with JMX, Version 9.0, September 7, 2005

[BEA02] BEA Systems Inc., BEA WebLogic® Product Family (http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/weblogic)

[WIK01] Wikipedia, Die freie Enzyklopädie, Deutsche Edition (http://de.wikipedia.org)

[WIK02] Wikipedia, The Free Encyclopedia, Englische Edition (http://en.wikipedia.org)

[BOR01] Borland Software Corp.®, The Open ALM Company (http://www.borland.com)

[BOR02] Borland Software Corp.®, The Open ALM Company, Press Release, February 2007 (http://www.borland.com/us/company/news/press_releases/2007/02_13_07_borland_extends_lqm_solution.html)

[IBM01] International Business Machines Corp.® (IBM), WebSphere Software (www.ibm.com/software/websphere)

[ORA01] Oracle Corp. ®, Oracle Application Server (http://www.oracle.com/appserver/index.html)

A. Anhang

WebSphere 6.0.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<AgentConfiguration Name="WebSphere 6.0" DataSourceType="Base
Configuration WebSphere 6.0">
  <Host>localhost</Host>
  <Port></Port>
  <Protocol>iiop</Protocol>
  <!--0 .. JSR160/RMI-->
  <!--1 .. JSR160/IIOP-->
  <!--2 .. JNDI-->
  <!--3 .. JSR160/CORBALOC-->
  <ConnectionMode>2</ConnectionMode>
  <MEJBName>ejb/mgmt/MEJB</MEJBName>
  <UrlPostfix />

<ContextFactory>com.ibm.websphere.naming.WsnInitialContextFactory</Co
ntextFactory>
  <DefaultUser />
  <DefaultPassword />
  <TypePropertyString>j2eeType</TypePropertyString>
  <IsJsr77>true</IsJsr77>
  <JvmInfo>
    <JvmHomePath RecommendedVendor="IBM"
RecommendedMajorVersion="1.4" />
    <Classpath
ClasspathRootDir="T:\BuildUtilities\Lib\Java\Websphere6\">
      <Entry>.\admin.jar</Entry>
      <Entry>.\bootstrap.jar</Entry>
      <Entry>.\ecutils.jar</Entry>
      <Entry>.\emf.jar</Entry>
      <Entry>.\ffdc.jar</Entry>
      <Entry>.\idl.jar</Entry>
      <Entry>.\iwsorb.jar</Entry>
      <Entry>.\j2ee.jar</Entry>
      <Entry>.\management.jar</Entry>
      <Entry>.\mejb.jar</Entry>
      <Entry>.\naming.jar</Entry>
      <Entry>.\namingclient.jar</Entry>
      <Entry>.\ras.jar</Entry>
      <Entry>.\sas.jar</Entry>
      <Entry>.\utils.jar</Entry>
      <Entry>.\wsexception.jar</Entry>
    </Classpath>
    <AdditionalClasspath />
    <VmParameters />
  </JvmInfo>
</AgentConfiguration>
```

WebLogic 9.0.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<AgentConfiguration Name="WebLogic 9.0" DataSourceType="Base
Configuration WebLogic 8.x">
  <Host>lab50</Host>
  <Port>7001</Port>
  <Protocol>t3</Protocol>
  <!--0 .. JSR160/RMI-->
  <!--1 .. JSR160/IIOP-->
  <!--2 .. JNDI-->
  <!--3 .. JSR160/CORBALOC-->
  <ConnectionMode>2</ConnectionMode>
  <MEJBName>weblogic/management/adminhome</MEJBName>
  <UrlPostfix />

<ContextFactory>weblogic.jndi.WLInitialContextFactory</ContextFactory
>
  <DefaultUser>weblogic</DefaultUser>
  <DefaultPassword>weblogic</DefaultPassword>
  <TypePropertyString>Type</TypePropertyString>
  <IsJsr77>false</IsJsr77>
  <JvmInfo>
    <JvmHomePath RecommendedVendor="SUN"
RecommendedMajorVersion="1.4" />
    <Classpath
ClasspathRootDir="T:\BuildUtilities\Lib\Java\Weblogic9">
      <Entry>.\weblogic.jar</Entry>
    </Classpath>
    <AdditionalClasspath />
    <VmParameters />
  </JvmInfo>
</AgentConfiguration>
```

WebLogic 9.0 JSR-160.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<AgentConfiguration Name="WebLogic 9.0 (JSR-160)"
DataSourceType="Base Configuration WebLogic 8.x">
  <Host>10.5.3.196</Host>
  <Port>7001</Port>
  <Protocol>t3</Protocol>
  <!--0 .. JSR160/RMI-->
  <!--1 .. JSR160/IIOP-->
  <!--2 .. JNDI-->
  <!--3 .. JSR160/CORBALOC-->
  <!--4 .. JSR160/RMIIIOP-->
  <ConnectionMode>4</ConnectionMode>
  <MEJBName />
  <UrlPostfix>weblogic.management.mbeanservers.runtime</UrlPostfix>

<ContextFactory>weblogic.jndi.WLInitialContextFactory</ContextFactory
>
  <DefaultUser>weblogic</DefaultUser>
  <DefaultPassword>weblogic</DefaultPassword>
  <TypePropertyString>Type</TypePropertyString>
  <IsJsr77>false</IsJsr77>
  <JvmInfo>
    <JvmHomePath RecommendedVendor="SUN"
RecommendedMajorVersion="1.4" />
    <Classpath
ClasspathRootDir="T:\BuildUtilities\Lib\Java\Weblogic9">
      <Entry>.\weblogic.jar</Entry>
    </Classpath>
    <AdditionalClasspath />
    <VmParameters />
  </JvmInfo>
</AgentConfiguration>
```

WebLogic 8.x.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<AgentConfiguration Name="WebLogic 8.x" DataSourceType="Base
Configuration WebLogic 8.x">
  <Host>lab53</Host>
  <Port>7001</Port>
  <Protocol>t3</Protocol>
  <!--0 .. JSR160/RMI-->
  <!--1 .. JSR160/IIOP-->
  <!--2 .. JNDI-->
  <!--3 .. JSR160/CORBALOC-->
  <ConnectionMode>2</ConnectionMode>
  <MEJBName>weblogic/management/adminhome</MEJBName>
  <UrlPostfix />

<ContextFactory>weblogic.jndi.WLInitialContextFactory</ContextFactory
>
  <DefaultUser />
  <DefaultPassword />
  <TypePropertyString>Type</TypePropertyString>
  <IsJsr77>false</IsJsr77>
  <JvmInfo>
    <JvmHomePath RecommendedVendor="SUN"
RecommendedMajorVersion="1.4" />
    <Classpath
ClasspathRootDir="T:\BuildUtilities\Lib\Java\Weblogic8">
      <Entry>.\weblogic.jar</Entry>
    </Classpath>
    <AdditionalClasspath />
    <VmParameters />
  </JvmInfo>
</AgentConfiguration>
```

SUN JVM 1.6.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<AgentConfiguration Name="SUN JVM 1.6" DataSourceType="Base
Configuration Jvm 1.6">
  <Host>localhost</Host>
  <Port>9998</Port>
  <Protocol />
  <!--0 .. JSR160/RMI-->
  <!--1 .. JSR160/IIOP-->
  <!--2 .. JNDI-->
  <!--3 .. JSR160/CORBALOC-->
  <ConnectionMode>0</ConnectionMode>
  <MEJBName />
  <UrlPostfix>jmxrmi</UrlPostfix>
  <ContextFactory />
  <DefaultUser />
  <DefaultPassword />
  <TypePropertyString>type</TypePropertyString>
  <IsJsr77>false</IsJsr77>
  <JvmInfo>
    <JvmHomePath RecommendedVendor="SUN"
RecommendedMajorVersion="1.6" />
    <Classpath />
    <AdditionalClasspath />
    <VmParameters>-Dcom.sun.management.jmxremote.ssl=false -
Dcom.sun.management.jmxremote.authenticate=false -
Dcom.sun.management.jmxremote.port=9996</VmParameters>
  </JvmInfo>
</AgentConfiguration>
```

SUN JVM 1.5.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<AgentConfiguration Name="SUN JVM 1.5" DataSourceType="Base
Configuration Jvm 1.5">
  <Host>localhost</Host>
  <Port>9998</Port>
  <Protocol />
  <!--0 .. JSR160/RMI-->
  <!--1 .. JSR160/IIOP-->
  <!--2 .. JNDI-->
  <!--3 .. JSR160/CORBALOC-->
  <ConnectionMode>0</ConnectionMode>
  <MEJBName />
  <UrlPostfix>jmxrmi</UrlPostfix>
  <ContextFactory />
  <DefaultUser />
  <DefaultPassword />
  <TypePropertyString>type</TypePropertyString>
  <IsJsr77>false</IsJsr77>
  <JvmInfo>
    <JvmHomePath RecommendedVendor="SUN"
RecommendedMajorVersion="1.5" />
    <Classpath />
    <AdditionalClasspath />
    <VmParameters>-Dcom.sun.management.jmxremote.ssl=false -
Dcom.sun.management.jmxremote.authenticate=false -
Dcom.sun.management.jmxremote.port=9997</VmParameters>
  </JvmInfo>
</AgentConfiguration>
```

JBoss 4.0.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<AgentConfiguration Name="JBoss 4.0" DataSourceType="Base
Configuration JBoss 4.x">
  <Host>lab50</Host>
  <Port>1099</Port>
  <Protocol>jnp</Protocol>
  <!--0 .. JSR160/RMI-->
  <!--1 .. JSR160/IIOP-->
  <!--2 .. JNDI-->
  <!--3 .. JSR160/CORBALOC-->
  <ConnectionMode>2</ConnectionMode>
  <MEJBName>jmx/rmi/RMIAdaptor</MEJBName>
  <UrlPostfix />

<ContextFactory>org.jnp.interfaces.NamingContextFactory</ContextFacto
ry>
  <DefaultUser />
  <DefaultPassword />
  <TypePropertyString>j2eeType</TypePropertyString>
  <IsJsr77>true</IsJsr77>
  <JvmInfo>
    <JvmHomePath RecommendedVendor="SUN" RecommendedMajorVersion="2"
/>
    <Classpath ClasspathRootDir="C:\jboss-4.0.1sp1">
      <Entry>.\server\all\lib\jboss-management.jar</Entry>
      <Entry>.\client\jbossall-client.jar</Entry>
      <Entry>.\lib\dom4j.jar</Entry>
    </Classpath>
    <AdditionalClasspath />
    <VmParameters />
  </JvmInfo>
</AgentConfiguration>
```

dynaTrace Diagnostics 2.0.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<AgentConfiguration Name="dynaTrace Diagnostics 2.0"
DataSourceType="Base Configuration Jvm 1.5">
  <Host>localhost</Host>
  <Port>2020</Port>
  <Protocol />
  <!--0 .. JSR160/RMI-->
  <!--1 .. JSR160/IIOP-->
  <!--2 .. JNDI-->
  <!--3 .. JSR160/CORBALOC-->
  <ConnectionMode>0</ConnectionMode>
  <MEJBName />
  <UrlPostfix>com.dynatrace.diagnostics.management.20</UrlPostfix>
  <ContextFactory />
  <DefaultUser />
  <DefaultPassword />
  <TypePropertyString>type</TypePropertyString>
  <IsJsr77>false</IsJsr77>
  <JvmInfo>
    <JvmHomePath RecommendedVendor="SUN"
RecommendedMajorVersion="1.5" />
    <Classpath />
    <AdditionalClasspath />
    <VmParameters />
  </JvmInfo>
</AgentConfiguration>
```

dynaTrace Diagnostics 1.6.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<AgentConfiguration Name="dynaTrace Diagnostics 1.6"
DataSourceType="Base Configuration Jvm 1.5">
  <Host>localhost</Host>
  <Port>2020</Port>
  <Protocol />
  <!--0 .. JSR160/RMI-->
  <!--1 .. JSR160/IIOP-->
  <!--2 .. JNDI-->
  <!--3 .. JSR160/CORBALOC-->
  <ConnectionMode>0</ConnectionMode>
  <MEJBName />
  <UrlPostfix>com.dynatrace.diagnostics.management.10</UrlPostfix>
  <ContextFactory />
  <DefaultUser />
  <DefaultPassword />
  <TypePropertyString>type</TypePropertyString>
  <IsJsr77>false</IsJsr77>
  <JvmInfo>
    <JvmHomePath RecommendedVendor="SUN"
RecommendedMajorVersion="1.5" />
    <Classpath />
    <AdditionalClasspath />
    <VmParameters />
  </JvmInfo>
</AgentConfiguration>
```

BMC TM ART 3.0 Application.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<AgentConfiguration Name="BMC TM ART 3.0 - Application Svr."
DataSourceType="Base Configuration Jvm 1.5">
  <Host>localhost</Host>
  <Port>19142</Port>
  <Protocol />
  <!--0 .. JSR160/RMI-->
  <!--1 .. JSR160/IIOP-->
  <!--2 .. JNDI-->
  <!--3 .. JSR160/CORBALOC-->
  <ConnectionMode>0</ConnectionMode>
  <MEJBName />
  <UrlPostfix>jmxrmi</UrlPostfix>

<ContextFactory>com.sun.jndi.cosnaming.CNCtxFactory</ContextFactory>
  <DefaultUser />
  <DefaultPassword />
  <TypePropertyString>type</TypePropertyString>
  <IsJsr77>false</IsJsr77>
  <JvmInfo>
    <JvmHomePath RecommendedVendor="SUN"
RecommendedMajorVersion="1.5" />
    <Classpath />
    <AdditionalClasspath />
    <!--VmParameters>-Dcom.sun.management.jmxremote.ssl=false -
Dcom.sun.management.jmxremote.authenticate=false -
Dcom.sun.management.jmxremote.port=9981</VmParameters-->
  </JvmInfo>
</AgentConfiguration>
```

BMC TM ART 3.0 Execution.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<AgentConfiguration Name="BMC TM ART 3.0 - Execution Svr."
DataSourceType="Base Configuration Jvm 1.5">
  <Host>localhost</Host>
  <Port>19144</Port>
  <Protocol />
  <!--0 .. JSR160/RMI-->
  <!--1 .. JSR160/IIOP-->
  <!--2 .. JNDI-->
  <!--3 .. JSR160/CORBALOC-->
  <ConnectionMode>0</ConnectionMode>
  <MEJBName />
  <UrlPostfix>jmxrmi</UrlPostfix>

<ContextFactory>com.sun.jndi.cosnaming.CNCtxFactory</ContextFactory>
  <DefaultUser />
  <DefaultPassword />
  <TypePropertyString>type</TypePropertyString>
  <IsJsr77>false</IsJsr77>
  <JvmInfo>
    <JvmHomePath RecommendedVendor="SUN"
RecommendedMajorVersion="1.5" />
    <Classpath />
    <AdditionalClasspath />
    <!--VmParameters>-Dcom.sun.management.jmxremote.ssl=false -
Dcom.sun.management.jmxremote.authenticate=false -
Dcom.sun.management.jmxremote.port=9982</VmParameters-->
  </JvmInfo>
</AgentConfiguration>
```

BMC TM ART 3.0 Frontend.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<AgentConfiguration Name="BMC TM ART 3.0 - Frontend Svr."
DataSourceType="Base Configuration Jvm 1.5">
  <Host>localhost</Host>
  <Port>19140</Port>
  <Protocol />
  <!--0 .. JSR160/RMI-->
  <!--1 .. JSR160/IIOP-->
  <!--2 .. JNDI-->
  <!--3 .. JSR160/CORBALOC-->
  <ConnectionMode>0</ConnectionMode>
  <MEJBName />
  <UrlPostfix>jmxrmi</UrlPostfix>

<ContextFactory>com.sun.jndi.cosnaming.CNCtxFactory</ContextFactory>
  <DefaultUser />
  <DefaultPassword />
  <TypePropertyString>type</TypePropertyString>
  <IsJsr77>false</IsJsr77>
  <JvmInfo>
    <JvmHomePath RecommendedVendor="SUN"
RecommendedMajorVersion="1.5" />
    <Classpath />
    <AdditionalClasspath />
    <!--VmParameters>-Dcom.sun.management.jmxremote.ssl=false -
Dcom.sun.management.jmxremote.authenticate=false -
Dcom.sun.management.jmxremote.port=9983</VmParameters-->
  </JvmInfo>
</AgentConfiguration>
```

Borland Application Server 6.6.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<AgentConfiguration Name="Borland Application Server 6.6"
DataSourceType="Base Configuration Jvm 1.5">
  <Host>DYLAN</Host>
  <Port>42222</Port>
  <Protocol />
  <!--0 .. JSR160/RMI-->
  <!--1 .. JSR160/IIOP-->
  <!--2 .. JNDI-->
  <!--3 .. JSR160/CORBALOC-->
  <ConnectionMode>3</ConnectionMode>
  <MEJBName />
  <UrlPostfix>j2eeSample_DYLAN_MyPartition</UrlPostfix>
  <ContextFactory />
  <DefaultUser />
  <DefaultPassword />
  <TypePropertyString>type</TypePropertyString>
  <IsJsr77>false</IsJsr77>
  <JvmInfo>
    <JvmHomePath RecommendedVendor="SUN"
RecommendedMajorVersion="1.4" />
    <Classpath ClasspathRootDir="T:\BuildUtilities\Lib\Java\BAS66">
      <Entry>.\lib\patches\7697.jar</Entry>
      <Entry>.\lib\anttasks.jar</Entry>
      <Entry>.\lib\asrt.jar</Entry>
      <Entry>.\lib\beandt.jar</Entry>
      <Entry>.\lib\bes-jbsp.jar</Entry>
      <Entry>.\lib\bes-jmx.jar</Entry>
      <Entry>.\lib\borland-commons-logging.jar</Entry>
      <Entry>.\lib\common.jar</Entry>
      <Entry>.\lib\dbswing.jar</Entry>
      <Entry>.\lib\dbtools.jar</Entry>
      <Entry>.\lib\ddeditor.jar</Entry>
      <Entry>.\lib\deployment.jar</Entry>
      <Entry>.\lib\dns.jar</Entry>
      <Entry>.\lib\dom4j.jar</Entry>
      <Entry>.\lib\dx.jar</Entry>
      <Entry>.\lib\dxdebug.jar</Entry>
      <Entry>.\lib\ejbdesigner.jar</Entry>
      <Entry>.\lib\gnuregexp.jar</Entry>
      <Entry>.\lib\help.jar</Entry>
      <Entry>.\lib\itsadmin.jar</Entry>
      <Entry>.\lib\jaas.jar</Entry>
      <Entry>.\lib\jafa.jar</Entry>
      <Entry>.\lib\jaxen.jar</Entry>
      <Entry>.\lib\jbcl-awt.jar</Entry>
      <Entry>.\lib\jbcl.jar</Entry>
      <Entry>.\lib\jbuilder.jar</Entry>
      <Entry>.\lib\jce1_2_1.jar</Entry>
      <Entry>.\lib\jcert.jar</Entry>
      <Entry>.\lib\jcommon.jar</Entry>
      <Entry>.\lib\jdbcx.jar</Entry>
      <Entry>.\lib\jdom.jar</Entry>
      <Entry>.\lib\jds.jar</Entry>
      <Entry>.\lib\jdshelp.jar</Entry>
      <Entry>.\lib\jdsmobile.jar</Entry>
```

```
<Entry>.\lib\jdsremote.jar</Entry>
<Entry>.\lib\jdsserver.jar</Entry>
<Entry>.\lib\jfreechart.jar</Entry>
<Entry>.\lib\jlayout20.jar</Entry>
<Entry>.\lib\jloox20.jar</Entry>
<Entry>.\lib\jnet.jar</Entry>
<Entry>.\lib\js.jar</Entry>
<Entry>.\lib\jsse.jar</Entry>
<Entry>.\lib\juddi.jar</Entry>
<Entry>.\lib\laf.jar</Entry>
<Entry>.\lib\lch.jar</Entry>
<Entry>.\lib\lm.jar</Entry>
<Entry>.\lib\local_policy.jar</Entry>
<Entry>.\lib\log4j.jar</Entry>
<Entry>.\lib\mail.jar</Entry>
<Entry>.\lib\migrate.jar</Entry>
<Entry>.\lib\migration.jar</Entry>
<Entry>.\lib\mx4j-impl.jar</Entry>
<Entry>.\lib\mx4j-jmx.jar</Entry>
<Entry>.\lib\mx4j-remote.jar</Entry>
<Entry>.\lib\mx4j-rimpl.jar</Entry>
<Entry>.\lib\mx4j-rjmx.jar</Entry>
<Entry>.\lib\mx4j-tools.jar</Entry>
<Entry>.\lib\mx4j.jar</Entry>
<Entry>.\lib\orgomg.jar</Entry>
<Entry>.\lib\partition.jar</Entry>
<Entry>.\lib\partition_client.jar</Entry>
<Entry>.\lib\primetime.ddeditor.jar</Entry>
<Entry>.\lib\primetime.jar</Entry>
<Entry>.\lib\providerutil.jar</Entry>
<Entry>.\lib\ResIndex.jar</Entry>
<Entry>.\lib\sanct6.jar</Entry>
<Entry>.\lib\sanctuary.jar</Entry>
<Entry>.\lib\scout.jar</Entry>
<Entry>.\lib\script.jar</Entry>
<Entry>.\lib\scu.jar</Entry>
<Entry>.\lib\scu_client.jar</Entry>
<Entry>.\lib\scu_common.jar</Entry>
<Entry>.\lib\snmp4_13.jar</Entry>
<Entry>.\lib\sunjce_provider.jar</Entry>
<Entry>.\lib\tailwind.jar</Entry>
<Entry>.\lib\tibjmslic.jar</Entry>
<Entry>.\lib\tomcat4.jar</Entry>
<Entry>.\lib\uddi4j.jar</Entry>
<Entry>.\lib\US_export_policy.jar</Entry>
<Entry>.\lib\vbcdev.jar</Entry>
<Entry>.\lib\vbdev.jar</Entry>
<Entry>.\lib\vbejb.jar</Entry>
<Entry>.\lib\vbjclientorb.jar</Entry>
<Entry>.\lib\vbjdev.jar</Entry>
<Entry>.\lib\vbjorb.jar</Entry>
<Entry>.\lib\vbsec.jar</Entry>
<Entry>.\lib\vbsysmib.jar</Entry>
<Entry>.\lib\vbwsdev.jar</Entry>
<Entry>.\lib\vbwsjdev.jar</Entry>
<Entry>.\lib\visiconnect.jar</Entry>
<Entry>.\lib\visijmx.jar</Entry>
<Entry>.\lib\wcch.jar</Entry>
<Entry>.\lib\xalan.jar</Entry>
<Entry>.\lib\xercesImpl.jar</Entry>
<Entry>.\lib\xmlParserAPIs.jar</Entry>
```

```
      <Entry>.\lib\xmlrt.jar</Entry>
    </Classpath>
    <AdditionalClasspath>
      <Entry>C:\Program Files\Borland\SilkPerformer 2006
R2\JMX\jmxri.jar</Entry>
    </AdditionalClasspath>
    <VmParameters>-
Dorg.omg.CORBA.ORBClass=com.inprise.vbroker.orb.ORB -
Dorg.omg.CORBA.ORBSingletonClass=com.inprise.vbroker.orb.ORBSingleton
-Djavax.rmi.CORBA.StubClass=com.inprise.vbroker.rmi.CORBA.StubImpl -
Djavax.rmi.CORBA.UtilClass=com.inprise.vbroker.rmi.CORBA.UtilImpl -
Djavax.rmi.CORBA.PortableRemoteObjectClass=com.inprise.vbroker.rmi.CO
RBA.PortableRemoteObjectImpl -
Dvbroker.agent.enableLocator=false</VmParameters>
  </JvmInfo>
</AgentConfiguration>
```

Borland SC Test Manager 8.5 Application.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<AgentConfiguration Name="Borland SC Test Manager 8.5 - Application
Svr." DataSourceType="Base Configuration Jvm 1.5">
  <Host>localhost</Host>
  <Port>19142</Port>
  <Protocol />
  <!--0 .. JSR160/RMI-->
  <!--1 .. JSR160/IIOP-->
  <!--2 .. JNDI-->
  <!--3 .. JSR160/CORBALOC-->
  <ConnectionMode>0</ConnectionMode>
  <MEJBName />
  <UrlPostfix>jmxrmi</UrlPostfix>

<ContextFactory>com.sun.jndi.cosnaming.CNCtxFactory</ContextFactory>
  <DefaultUser />
  <DefaultPassword />
  <TypePropertyString>type</TypePropertyString>
  <IsJsr77>false</IsJsr77>
  <JvmInfo>
    <JvmHomePath RecommendedVendor="SUN"
RecommendedMajorVersion="1.5" />
    <Classpath />
    <AdditionalClasspath />
    <!--VmParameters>-Dcom.sun.management.jmxremote.ssl=false -
Dcom.sun.management.jmxremote.authenticate=false -
Dcom.sun.management.jmxremote.port=9971</VmParameters-->
  </JvmInfo>
</AgentConfiguration>
```

Borland SC Test Manager 8.5 Execution.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<AgentConfiguration Name="Borland SC Test Manager 8.5 - Execution
Svr." DataSourceType="Base Configuration Jvm 1.5">
  <Host>localhost</Host>
  <Port>19144</Port>
  <Protocol />
  <!--0 .. JSR160/RMI-->
  <!--1 .. JSR160/IIOP-->
  <!--2 .. JNDI-->
  <!--3 .. JSR160/CORBALOC-->
  <ConnectionMode>0</ConnectionMode>
  <MEJBName />
  <UrlPostfix>jmxrmi</UrlPostfix>

<ContextFactory>com.sun.jndi.cosnaming.CNCtxFactory</ContextFactory>
  <DefaultUser />
  <DefaultPassword />
  <TypePropertyString>type</TypePropertyString>
  <IsJsr77>false</IsJsr77>
  <JvmInfo>
    <JvmHomePath RecommendedVendor="SUN"
RecommendedMajorVersion="1.5" />
    <Classpath />
    <AdditionalClasspath />
    <!--VmParameters>-Dcom.sun.management.jmxremote.ssl=false -
Dcom.sun.management.jmxremote.authenticate=false -
Dcom.sun.management.jmxremote.port=9972</VmParameters-->
  </JvmInfo>
</AgentConfiguration>
```

Borland SC Test Manager 8.5 Frontend.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<AgentConfiguration Name="Borland SC Test Manager 8.5 - Frontend
Svr." DataSourceType="Base Configuration Jvm 1.5">
  <Host>localhost</Host>
  <Port>19140</Port>
  <Protocol />
  <!--0 .. JSR160/RMI-->
  <!--1 .. JSR160/IIOP-->
  <!--2 .. JNDI-->
  <!--3 .. JSR160/CORBALOC-->
  <ConnectionMode>0</ConnectionMode>
  <MEJBName />
  <UrlPostfix>jmxrmi</UrlPostfix>

<ContextFactory>com.sun.jndi.cosnaming.CNCtxFactory</ContextFactory>
  <DefaultUser />
  <DefaultPassword />
  <TypePropertyString>type</TypePropertyString>
  <IsJsr77>false</IsJsr77>
  <JvmInfo>
    <JvmHomePath RecommendedVendor="SUN"
RecommendedMajorVersion="1.5" />
    <Classpath />
    <AdditionalClasspath />
    <!--VmParameters>-Dcom.sun.management.jmxremote.ssl=false -
Dcom.sun.management.jmxremote.authenticate=false -
Dcom.sun.management.jmxremote.port=9973</VmParameters-->
  </JvmInfo>
</AgentConfiguration>
```

Curriculum Vitae des Autors

Allgemeines

Johannes Hölzl

Rosenstrasse 12

4040 Linz

email: johannes.hoelzl@borland.com

Ausbildung

1985-89	Volksschule Schweinbach
1989-97	BRG Auhof
1997-98	Grundwehrdienst
seit 1998	Studium an der Johannes Kepler Universität Linz
	Studienrichtung Informatik (881, 880, 921)

Berufserfahrung

seit 1998	Softwareentwickler bei: Borland Entwicklung GmbH Linz Freistädterstrasse 400
	ehemals (vor April 2006): Segue Software Entwicklungs GmbH Wienerstrasse 131 / Freistädterstrasse 400

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Magisterrbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wortwörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Des weiteren versichere ich, dass ich diese Magisterarbeit weder im In- noch im Ausland in irgendeiner Form als Prüfungsarbeit vorgelegt habe.