

Technische Universität Darmstadt



# Verwaltung eines verteilten Systems mit Java und Servlets

Wintersemester 2001/2002  
Christoph Ender – Matrikel Nummer 723772  
20. März 2002

Betreuer: Markus Lippert

---

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>5</b>
1.1	Gliederung . . . . .	6
1.2	Typographische Konventionen . . . . .	6
<b>2</b>	<b>Grundlegende Annahmen</b>	<b>7</b>
2.1	Rahmenbedingungen . . . . .	7
2.1.1	Unterteilung der zu verwaltenden Bereiche . . . . .	7
2.1.2	Verwaltung von Properties . . . . .	8
2.1.3	Funktionen . . . . .	8
2.1.4	Starten und Stoppen von Modulen . . . . .	8
2.1.5	Servlets und Servlet-Container . . . . .	8
<b>3</b>	<b>Entwurf</b>	<b>10</b>
3.1	Konzeptionelle Fragen . . . . .	10
3.2	Umsetzung . . . . .	12
3.2.1	Die Verwaltung der Administrationsumgebung . . . . .	12
3.2.2	Kommunikationsendpunkte . . . . .	13
3.2.3	Zu verwaltende Objekte . . . . .	13
3.2.4	Logging . . . . .	13
<b>4</b>	<b>Erste Realisierung</b>	<b>16</b>
4.1	Bestehende Komponenten . . . . .	16
4.1.1	flexiTrust . . . . .	16
4.1.2	Java Servlets . . . . .	17
4.1.3	Java Virtual Machines . . . . .	18
4.1.4	Tomcat Servlet-Container . . . . .	18
4.1.5	RMI . . . . .	19
4.2	Konzept . . . . .	20
4.2.1	Der AdminNode . . . . .	20
4.2.2	Module . . . . .	20
4.2.3	Komponenten . . . . .	20
4.2.4	Verwalten von Properties . . . . .	21
4.2.5	Authentifizierung via X.509 Zertifikat . . . . .	21

---

---

4.2.6	SSL-verschlüsselte RMI-Kommunikation . . . . .	22
4.3	Umsetzung . . . . .	22
4.3.1	Arbeitsweise . . . . .	22
4.3.2	Klassen . . . . .	26
4.4	Ende der Weiterführung . . . . .	33
<b>5</b>	<b>Zweite Realisierung</b>	<b>34</b>
5.1	Bestehende Komponenten . . . . .	34
5.1.1	JMX . . . . .	34
5.2	Konzept . . . . .	35
5.2.1	Warum JMX? . . . . .	36
5.2.2	JMX-RMI . . . . .	36
5.2.3	JmxAgent . . . . .	38
5.2.4	ClassInvoker . . . . .	38
5.2.5	AdminServlet . . . . .	40
5.3	Umsetzung . . . . .	40
5.3.1	Arbeitsweise . . . . .	40
5.3.2	Klassen . . . . .	42
<b>6</b>	<b>Zusammenfassung</b>	<b>49</b>
<b>A</b>	<b>Admin-HowTo</b>	<b>50</b>
A.1	Overview . . . . .	50
A.2	Java Management Extensions . . . . .	50
A.2.1	Static MBeans . . . . .	51
A.2.2	Dynamic MBeans . . . . .	52
A.3	flexiPKI-specific JMX-Enhancements . . . . .	54
A.3.1	RmiAdaptorClient and JmxAgent . . . . .	54
A.4	Understanding the JMX Management . . . . .	55
A.5	Setting up the JmxAgent with the PrinterHandler . . . . .	57
A.5.1	Registering PrinterHandler in the same virtual machine . . . . .	57
A.5.2	Registering PrinterHandler in a different virtual machine . . . . .	59
A.5.3	Setting up the Administration Servlet . . . . .	59
A.6	Dynamic MBean PrinterHandler . . . . .	60
A.7	Installing the Administration Package . . . . .	65
A.7.1	Build the JmxAgent . . . . .	65
A.7.2	Setting up Tomcat for the Servlet . . . . .	65
<b>B</b>	<b>Tomcat Installation</b>	<b>66</b>
B.1	Requirements . . . . .	66
B.2	Tomcat Installation . . . . .	66
B.3	Tomcat Servlet Setup . . . . .	68
B.4	Setup Apache server and personal certificates . . . . .	69
B.5	Start all Components and test the Installation . . . . .	69

---

<b>C Extracting MIME-Encoded Parameters</b>	<b>70</b>
<b>Literaturverzeichnis</b>	<b>73</b>

---

# Kapitel 1

## Einleitung

Für komplexere Programme und Anwendungen werden zu Konfigurations- und Wartungszwecken eigene Tools entwickelt. Diese Werkzeuge verschaffen einen Überblick über den Status des Systems und erlauben es Zustandsinformationen nicht nur abzufragen sondern auch zu manipulieren. Entsprechend komfortable und intuitiv zu erfassende Verwaltungsutilities entlasten den Administrator deutlich und verkürzen die Einarbeitungszeit.

In vielen Fällen sind Tools oder Frontends speziell zur Verwaltung eines Systems unverzichtbar. Die Fähigkeit, das Verhalten bestimmter Teilbereiche eines größeren Systems gezielt zu bearbeiten und die daraus resultierenden Veränderungen beobachten zu können sind nicht nur bei der Entwicklung und bei der Fehlersuche förderlich, vielmehr erlauben sie auch, bei unvorhergesehenen Ereignissen zur eigentlichen Laufzeit reagieren zu können indem an den notwendigen Stellen gezielt in den Programmablauf eingegriffen werden kann, ohne den Quellcode manipulieren zu müssen. Auch bei gravierenden Änderungen im System kann durch ein entsprechendes Hilfsmittel zur Verwaltung die neue Funktionalität schneller, einfacher und damit sicherer genutzt werden.

Obwohl das Verwaltungswerkzeug im Zusammenhang mit der flexiTrust-PKI entwickelt wurde kann es durchaus auch in einer anderen Umgebung verwendet werden. Die zugrundeliegenden Eigenschaften des zu verwaltenden Systems wurden so allgemein wie möglich betrachtet, um ein möglichst universell zu verwendbares Tool zu erhalten. Durch die Implementierung des von Sun vorgestellten Standards zum Applikationsmanagement kann das entstandene Werkzeug ohne größere Änderungen auch für andere Systeme verwendet werden, und da lediglich ein Webbrowser zum Arbeiten benötigt wird kann das System über sehr viele Plattformen genutzt werden.

---

## 1.1 Gliederung

Die Kapitel der Arbeit sind wie folgt gegliedert:

Kapitel 2 gibt einen kurzen Überblick über die für diese Arbeit angenommenen Rahmenbedingungen.

Kapitel 3 beschreibt den für beide Realisierungen relevanten Entwurf. Es werden die allgemeinen Grundgedanken zur Administration verteilter Systeme dargestellt und auf die Umsetzung im flexiTrust-Bereich präzisiert.

Kapitel 4 beschreibt die erste Realisierung. Diese wurde ohne JMX konzipiert und nach dem Erstellen des ersten Prototypen nicht weiter entwickelt, da sich der JMX-Ansatz als der wünschenswertere herausgestellt hat.

Kapitel 5 beschreibt die zweite Realisierung. Im Gegensatz zum ersten Ansatz wird hier JMX verwendet. Dieser Entwurf wurde vollständig implementiert.

Kapitel 6 faßt die im Verlauf dieser Arbeit erworbenen Kenntnisse zusammen.

Im Anhang sind die Einführung für das Administrationstool und Hinweise zur Installation von Tomcat zusammen mit Hinweisen zum Vorgehen beim Extrahieren von MIME-kodierten Inhalten aus Browser-Requests aufgeführt.

## 1.2 Typographische Konventionen

*Kursivschrift* wird für Datei- und Verzeichnisnamen sowie für Pfadnamen verwendet. **Nichtproportionalschrift** – „Typewriter“ – wird für Codepassagen, Datentypen und Klassennamen verwendet.

---

## Kapitel 2

# Grundlegende Annahmen

Im Rahmen dieser Diplomarbeit ist am Beispiel des flexiTrust[1]-Administrationstools ein allgemein zu verwendendes Verwaltungswerkzeug konzipiert und implementiert worden. Dabei soll der Administrator mit dem Verwaltungswerkzeug via HTML über einen Webbrowser kommunizieren können. Durch die generelle Verfügbarkeit von Webbrowsern wird Plattformunabhängigkeit gewährleistet, außerdem sind die meisten Anwender bereits mit einem Webbrowser vertraut, was die Einarbeitungszeit entsprechend verkürzt.

Da im Rahmen dieser Arbeit ein möglichst universell verwendbares Werkzeug entstehen sollte, werden in Bezug auf die zu verwaltenden Eigenschaften nur einige allgemeine Vorgaben angenommen. Es werden mehrere Prozesse bzw. Threads verwaltet, welche entweder alleine oder mit anderen zusammen in einer Java Virtual Machine laufen. Jeder Prozeß besitzt eine oder mehrere Eigenschaften, im folgenden Properties genannt, die gelesen oder auch geschrieben werden können. Des weiteren soll es möglich sein Funktionen aufzurufen, diesen dabei einen oder mehrere Parameter mitzugeben und einen Rückgabewert zu erhalten – ähnlich einem Funktionsaufruf in einer Programmiersprache. Als weiteres Feature sollen einige Loggingfunktionen konzipiert werden, zusammen mit einer Möglichkeit, die Logs – auch im verteilten System – auslesen und analysieren zu können. Die Berücksichtigung verteilter Systeme ist zur heutigen Zeit und grade im Bereich der Programmiersprache Java ein wichtiger Bestandteil der Annahmen, da hier durch die schon vorhandenen Tools verteiltes Rechnen einfach realisiert werden kann und häufiger genutzt wird.

## 2.1 Rahmenbedingungen

### 2.1.1 Unterteilung der zu verwaltenden Bereiche

Die zu administrierenden Teilbereiche des Systems werden in Module eingeteilt. Diese Einteilung ist mehr logischer als technischer Natur. So könnte ein Modul z.B. ein Sammlung von Konfigurationseinstellungen, einen einzelnen Prozeß oder eine Menge von Prozessen beinhalten. Es wäre denkbar, ein zu verwaltendes Computersystem als Modul

---

zu betrachten oder die Bestandteile wie Drucker oder Rechner in einzelne Module einzuteilen. Die Einteilung sollte so erfolgen, daß der Anwender sich gut orientieren kann und die gewünschte Funktion im System möglichst einfach findet.

### 2.1.2 Verwaltung von Properties

Die Vorgabe verlangt, daß die Properties in den zu verwaltenden Modulen ausgelesen und manipuliert werden können. Hier soll der Begriff „Properties“ genauer definiert werden:

Die zu bearbeitenden Properties haben je einen Namen und einen Wert. Der Name ist ein gültiger Java-String, der Wert ein primitiver Java-Datentyp. Von der aktuellen Implementierung unterstützte Datentypen sind `int`, `float`, `long`, `boolean` und `String`. Diese Properties können entweder nur gelesen, nur geschrieben oder sowohl gelesen als auch geschrieben werden. Das Administrationstool soll eine einfache und komfortable Methode unterstützen, mit der die in den verschiedenen zu verwaltenden Modulen vorhandenen Properties entsprechend ausgelesen und/oder manipuliert werden können.

### 2.1.3 Funktionen

Jedes zu verwaltende Modul kann eine oder mehrere Funktionen beinhalten. Das Administrationstool soll in der Lage sein, diese aufzurufen und einen Funktionswert zurückzugeben. Eine Funktion kann – wie in einer Programmiersprache üblich – einen oder mehrere Parameter besitzen und einen Rückgabewert liefern. Die gültigen Übergabeparameter sind, wie schon bei den Properties erwähnt, `int`, `float`, `long`, `boolean` und `String`. Zusätzlich existiert noch die Klasse `de.tud.cdc.flexiTrust.ad.ByteArray`, mit der es möglich ist eine Datei über den Webbrowser zum Administrationstool zu übertragen. Die gültigen Rückgabeparameter sind wiederum `int`, `float`, `long`, `boolean` und `String`. Das Administrationstool soll in der Lage sein eine einfach zu bedienende Maske zur Verfügung zu stellen mit der man die Parameter für die Funktion eingeben kann, nach einer entsprechenden Bestätigung vom Administrator die Funktion aufzurufen und danach den Rückgabewert anzuzeigen. Ein Beispiel für eine Anwendung wäre etwa das erneute Ausdrucken eines PIN-Briefes.

### 2.1.4 Starten und Stoppen von Modulen

Das Administrationstool soll es ermöglichen auf den verschiedenen Rechnern innerhalb des Netzwerks bestimmte Module zu starten. Dabei werden von der Anwendung die Namen von Klassen oder Ant-Targets zur Verfügung gestellt, die das Verwaltungstool dann in der eigenen oder auch in einer neuen Java Virtual Machine startet.

### 2.1.5 Servlets und Servlet-Container

Um die Integration des Administrationstools mit dem zu verwaltenden System zu vereinfachen, werden bevorzugt bereits in der Programmiersprache Java verfügbare Werkzeuge benutzt. So soll die Interaktion zwischen Administrator und Verwaltungswerkzeug mit Hilfe von Java Servlets[2] realisiert werden. Der bevorzugte Servlet-Container ist

---

Tomcat[3]. Mit Tomcat ist es möglich, X.509-Zertifikate aus HTTPS-Anfragen zu extrahieren, wodurch eine Authentifizierung des Administrators möglich wird. Eine genauere Erläuterung dieser Komponenten folgt in Kapitel 3 auf den Seiten 17 bzw. 18.

---

## Kapitel 3

# Entwurf

In diesem Kapitel wird der grundlegende Entwurf, der beiden in diesem Text geschilderten Realisierungsmöglichkeiten zugrunde liegt, erläutert. Ein Großteil des Konzepts ergibt sich direkt aus den zugrundeliegenden Begebenheiten.

### 3.1 Konzeptionelle Fragen

Im Bereich dieser Arbeit wird die Administration eines verteilten Systems betrachtet. In diesem Zusammenhang ist im Bezug auf den Begriff „Administration“ vor allem der Aspekt der Fernwartung eines gegebenen System interessant. Das laufende System soll über eine einheitliche Schnittstelle so beeinflusst werden können, daß die gesamten Möglichkeiten und Fähigkeiten des Systems derart ausgeschöpft werden können, so daß nur noch ein minimaler Teil der Verwaltungsaufgaben „von Hand“ abgearbeitet werden muß, wie z.B. das Starten des Administrationssystems an sich.

Die Bereiche der Administration können dabei durchaus mit der eigentlichen Programmbedienung verschmelzen. Im Bereich dieser Arbeit ist es z.B. möglich über die Administrationsfunktionen bestimmte Funktionalität innerhalb der zu verwaltenden Prozesse zu aktivieren, die auf keine andere Weise aufgerufen werden können. In diesem Fall übernimmt das Administrationstool teilweise die Anwendungsbereiche eines Frontends. Ein Beispiel dafür ist im Bereich dieser Diplomarbeit der Aufruf von JMX-Funktionen oder das Durchsuchen der Logfiles.

Unter dem Begriff „Administration“ wird gängigerweise die Verwaltung eines bereits laufenden Systems verstanden. Administration kann jedoch auch die Konfiguration eines noch nicht vollständig aktiven Systems beinhalten, um den lauffähigen Status des verwalteten Systems zu erreichen. Werden Grundeinstellungen am noch inaktiven System vorgenommen, so spricht im allgemeinen von „Konfiguration“, bzw. „Wartung“, wenn das System bereits aktiv war und bestimmte Eigenschaften verändert werden sollen.

#### **Konfiguration**

Das eigentliche zu verwaltende System soll über das Administrationswerkzeug gestartet werden können. Vor allem innerhalb eines verteilten Systems ist eine einfache Einbindung

---

eines neuen Rechners wünschenswert. Damit ein neuer Knoten eingebunden werden kann, sind im Bereich dieses Verwaltungswerkzeuges folgende Minimalanforderungen an einen neuen Rechner zu erfüllen:

- **Betriebssystem**

Ein Betriebssystem, für welches eine Java-Umgebung existiert, wird benötigt.

- **Java**

Es muß eine Java-Laufzeitumgebung installiert sein.

- **Klassenrepository**

Die Klassen des Administrationstools und die der zu verwaltenden Applikation müssen zugänglich sein.

- **Aktiver Admin-Kommunikationsendpunkt**

Das Administrationstool benötigt auf jedem zu verwaltenden Rechner einen Kommunikationspartner. Falls das Administrationstool bereits aktiv ist, genügt es, auf dem neuen, hinzuzufügenden Rechner einen neuen Kommunikationsendpunkt zu starten – je nach Realisierung einen AdminNode bzw. einen JmxAgent.

Sobald diese Voraussetzungen geschaffen wurden ist das Verwaltungstool in der Lage mit dem neuen Rechner zu kommunizieren. Dort können dann die Komponenten der Applikation vollständig über das Administrationsutility verwaltet werden. Im allgemeinen wird es sinnvoll sein, den Kommunikationspartner direkt automatisch beim Initialisieren des Betriebssystems starten zu lassen. In diesem Fall muß einfach nur noch der Rechner eingeschaltet und die Initialisierung des Betriebssystems abgewartet werden um den neuen Knoten im Netzwerk verfügbar zu machen.

## **Administration**

Sobald das gesamte System im lauffähigen Zustand ist können im laufenden Betrieb Änderungen an den aktiven Komponenten vorgenommen werden. Die einfachsten Manipulationen bestehen im Hinzufügen neuer und Stoppen oder Entfernen bereits laufender Prozesse. Alle anderen Manipulation können im Prinzip durch die Änderung von Attributen vorgenommen werden. Eine bestimmte Menge von (primitiven) Java-Datentypen sind von außen über das Administrationsutility zu manipulieren und werden von dem zu verwaltenden Prozeß entweder bei Bedarf verwendet oder es werden dann Aktionen ausgelöst, wenn ein bestimmtes Attribut einen bestimmten Wert angenommen hat. Dazu muß ein solches Attribut allerdings in regelmäßigen Zeitabständen abgefragt werden (sog. „Polling“), was einen unverhältnismäßig hohen Rechenaufwand bedeutet. Außerdem muß bei dieser Art von „Funktionsaufruf“ der Rückgabewert über ein eigenes Attribut realisiert werden.

Besser ist es einen Funktionsaufruf neben der Attributsmanipulation als eigenes Konzept zu realisieren. Ein Funktionsaufruf wird wie das entsprechende Konstrukt innerhalb einer Programmiersprache gestaltet: Eine Funktion besitzt einen Namen, kann ein oder mehrere Argumente haben und optional einen Wert zurückliefern.

---

## Logging

Zu den bisher erwähnten Aspekten der Applikationsverwaltung kommt noch das Logging von Ablaufinformationen hinzu. Obwohl Logging im Prinzip nicht direkt für das Verwalten von Anwendungen benötigt wird bietet es viele Vorteile. Von einer einheitlichen Loggingschnittstelle, die von allen Bereichen der Anwendung verwendet wird, profitieren vor allem die Fehlersuche und eine eventuelle Ablaufkontrolle. Durch eindeutige zeitliche Markierung eines Logeintrages kann ein Überblick über die Aktivitäten in einem bestimmten Zeitraum gewonnen werden. Zur vereinfachten Fehlersuche können die Logeinträge außerdem mit dem Namen der erzeugenden Klassen versehen werden. Des weiteren ist eine Einstufung der „Wichtigkeit“ eines Logeintrags nützlich.

## 3.2 Umsetzung

### 3.2.1 Die Verwaltung der Administrationsumgebung

Gegeben sind mehrere Rechner, die durch das Administrationstool verwaltet werden sollen. Die technisch einfachste Realisierung würde darin bestehen, auf jedem Rechner im flexiTrust-Netzwerk einen Kommunikationsendpunkt für den Webbrowser des Administrators einzurichten, welcher Zugriff auf die verschiedenen flexiTrust-Komponenten des Servers gewährt. Dieser Ansatz ist jedoch für den Administrator keineswegs der wünschenswerteste, da auf diese Weise jeder Rechner nur für sich, aber nicht im Zusammenhang mit anderen verwaltet werden kann. Des weiteren muß er die Netzwerkadressen oder die Namen jedes einzelnen Rechners im Netzwerk kennen um auf diese zugreifen zu können. Man möchte also lieber eine Realisierung, die es erlaubt, über einen einzelnen Rechner auf alle weiteren zugreifen zu können.

Für diesen Ansatz wird ein Kommunikationspartner für den Webbrowser benötigt, welcher seinerseits mit den anderen Rechnern im Netzwerk kommunizieren kann. Der Administrator bekommt nach erfolgreicher Anmeldung in diesem Fall eine Liste aller Rechner präsentiert und kann so jeden einzelnen Rechner gezielt ansprechen. Durch die zentrale Verwaltung aller Server lassen sich jedoch auch Aufgaben bewältigen, die nicht nur einen sondern auch mehrere Rechner betreffen. Es ist also z.B. möglich eine Anfrage nach Logeinträgen, deren Eintragsdatum sich in einem fest definierten Interval befinden muß, an alle Rechner zu stellen.

Realisiert wird dieses Verwaltungskonzept in der Praxis wie folgt: Ein Rechner, der nicht unbedingt zum eigentlichen flexiTrust-Netzwerk gehören muß, wird für die Installation des Programmteils zwecks Kommunikation mit dem Webbrowser des Administrators ausgewählt. Diese Komponente bildet quasi ein eigenes Management für die Erreichbarkeit aller anderen Bereiche. Der Name oder die Netzwerkadresse des Managementrechners muß dem Administrator bekannt sein. Zur Initalisierung wird diesem Programmteils dabei eine Liste der eigentlichen Rechner, auf denen flexiTrust-Komponenten installiert sind, vorgegeben. Die in der Liste benannten Rechner müssen nicht unbedingt aktiv sein wenn

---

das System gestartet wird. Das Management versucht nun seinerseits auf bestimmte, für die Administrationsumgebung erstellte Kommunikationsendpunkte auf jedem flexiTrust-Rechner zuzugreifen. Dadurch kann es die Eigenschaften aller laufenden Prozesse auf diesen Rechnern auslesen und dem Anwender einen Überblick über die verfügbaren Elemente geben. Es kann die Rechnernamen bzw. -adressen verwalten, evtl. Anfragen an die Logs verschicken und ansonsten alle eingehenden Anfragen an die entsprechenden Empfänger weiterleiten.

### 3.2.2 Kommunikationsendpunkte

Damit das Managementsystem – oder eine beliebige andere Komponente – mit den zu verwaltenden Prozessen Kontakt aufnehmen kann, benötigt es auf jedem im Netzwerk zu administrierenden Rechner einen „Ansprechpartner“, einen fest definierten Verbindungsendpunkt. Innerhalb der ersten Realisierung werden diese Kommunikationsendpunkte „AdminNode“, innerhalb der zweiten „JmxAgent“ genannt. Beide erfüllen im Prinzip die gleiche Aufgabe, auch wenn die Implementierungen grundsätzlich verschieden sind.

Um von außerhalb ansprechbar zu sein, wird der AdminNode bzw. JmxAgent unter einem vorher festgelegten Namen auf dem eigenen Rechner registriert. Dieser Name ist dem Managementsystem bekannt, so daß unabhängig von der Art und Menge der zu verwaltenden Objekte jederzeit eine Verbindung zu jedem Rechner aufgebaut werden kann. Damit nun auch die zu administrierenden Objekte angesprochen werden können, müssen diese sich ihrerseits wiederum beim AdminNode bzw. JmxAgent registrieren.

### 3.2.3 Zu verwaltende Objekte

Die zu verwaltenden Prozesse werden an den AdminNode bzw. JmxAgent der aktuellen virtuellen Maschine angebunden. Je nach Realisierung unterscheiden sich die möglichen Eigenschaften der zu verwaltenden Objekte, grundsätzlich ist jedoch zu sagen, daß ein Objekt eine Menge von zu manipulierenden Eigenschaften und aufzurufenden Funktionen nach außen zur Verfügung stellt. Auf der Basis der Eigenschaften lassen sich Properties realisieren, auf Basis der Funktionen das Starten und Stoppen von Diensten sowie der explizite Aufruf einzelner Funktionalitäten.

### 3.2.4 Logging

Es soll Funktionalität geschaffen werden mit der Loggingnachrichten in einem einheitlichen Format in eine Datei geschrieben und später analysiert werden können. Ein Logeintrag hat dabei das folgende Format:

- Uhrzeit  
Eine Zeitangabe, die die Zeit des Eintrags in GMT angibt.
  - Loglevel  
Eine ganze Zahl größer null, die angibt, wie wichtig der Eintrag ist. Je niedriger die Angabe, desto wichtiger ist der Eintrag.
-

- Modulbezeichnung  
Der Name des Moduls welches den Eintrag geschrieben hat, z.B. „RA“ oder „IS“.
- Klassenname  
Der Name der Klasse, die den Eintrag geschrieben hat.
- Nachricht  
Der eigentliche Logeintrag.

Die Logdateien sollen dabei vom Administrationstool auch analysiert werden können. So soll es möglich sein, alle Einträge von allen im Netzwerk erkannten Rechnern nach bestimmten Kriterien zu sortieren und die gewünschten auszugeben. Das Verwaltungstool soll in der Lage sein, alle Einträge, die innerhalb eines bestimmten Zeitintervalls erstellt wurden und ein Loglevel größer oder kleiner als eine bestimmte Vorgabe haben auszugeben. Um dies zu erreichen werden alle Angaben bis auf den eigentlichen Logeintrag in eckige Klammern gesetzt. Beispiel:

```
[Jul 6, 2001 4:39:43 PM] [994430383181] [16] [Admin] [LogInspector] "mylog" has been opened.
```

Das erste Feld enthält die lokalisierte Zeit, das zweite die Zeitangabe als Anzahl von Sekunden seit 1970 in GMT und das dritte Feld enthält das Loglevel. In der Klasse `de.tud.cdc.flexiTrust.Admin.Logfile` werden die folgenden globalen Konstanten für Logfiles vorgegeben:

```
public static final int FATAL = 0;
public static final int ERROR = 8;
public static final int EXCEPTION = 16;
public static final int WARNING = 24;
public static final int COMMENT = 32;
public static final int DEBUG = 64;
public static final int DEBUG2 = 128;
```

Feld Nummer vier ist der Name des Moduls, welches in diesem Fall die Exception ausgelöst hat, das nächste Feld enthält den Klassennamen. Zuletzt folgt, ohne eckige Klammern, der Logeintrag.

Probleme ergeben sich lediglich dann, wenn ein Logeintrag mehrere Zeilen umfaßt. Für solche Fälle wird hinter dem Zeilenumbruchszeichen ein Leerzeichen eingefügt. Jede umgebrochene Zeile eines Eintrags beginnt also mit einem Leerzeichen. Beginnt eine neue Zeile direkt mit einer öffnenden linken, eckigen Klammer, so beginnt ein neuer Logeintrag. Ein einzelner Eintrag kann leicht dadurch rekonstruiert werden, daß alle Leerzeichen nach einem Zeilenumbruch entfernt werden.

Beispiel für einen mehrere Zeilen umfassenden Logeintrag:

```
[Jul 6, 2001 4:42:25 PM] [994430545387] [32] [Admin] [LogInspector] Trying to open "mylog".
[Jul 6, 2001 4:42:25 PM] [994430545452] [16] [Admin] [LogInspector] Exception: mylog
java.io.FileNotFoundException: mylog (No such file or directory)
    at java.io.FileInputStream.open(Native Method)
```

---

```
at java.io.FileInputStream.<init>(FileInputStream.java:68)
at de.tud.cdc.flexiTrust.Admin.LogInspector.openLogfile(LogInspector.java:73)
at de.tud.cdc.flexiTrust.Admin.LogInspector.init(LogInspector.java:94)
at de.tud.cdc.flexiTrust.Admin.LogInspector.<init>(LogInspector.java:58)
```

## Kapitel 4

# Erste Realisierung

Die erste Realisierung umfaßt einen einfachen Ansatz auf Basis von Servlets und RMI (Erläuterung dieser Begriffe folge im nächsten Abschnitt). Diese Umsetzung wurde nicht vervollständigt und es wurde nur ein Prototyp entwickelt, da sich später herausstellte, daß ein Entwurf auf Basis von JMX, den Java Management Extensions, eine leistungsfähigere und auch standatisiertere Alternative darstellt.

### 4.1 Bestehende Komponenten

In diesem Abschnitt werden die Komponenten beschrieben auf denen das System aufbaut und über welche infolgedessen eine gewisse Grundkenntnis vorhanden sein muß um die später getroffenen Entscheidungen und Umsetzungen verstehen können. Es wird kein umfassender Überblick gegeben, das Gewicht wird dagegen auf das Verständnis der für die Diplomarbeit wichtigen Punkte der Komponenten gelegt. Für weiterführende Erklärungen in Bezug auf die für diese Arbeit nicht relevanten Teile der Systeme sei auf die entsprechende Dokumentation verwiesen.

#### 4.1.1 flexiTrust

flexiTrust ist eine vom Bereich für theoretische Informatik an der Technischen Universität Darmstadt entwickelte Public-Key-Infrastruktur („PKI“). Dieses System soll nach Abschluß der Diplomarbeit vom Administrationstool verwaltet werden können. flexiTrust besteht aus mehreren Modulen und kann als verteiltes System laufen. Die einzelnen Komponenten innerhalb dieser Public-Key-Infrastruktur sind die Registration Authority, kurz RA, die Certification Authority, abgekürzt CA sowie die Infrastructural Services, kurz IS. Innerhalb der Registration Authority können vom Administrator Anträge an die PKI gestellt werden. So kann mit Hilfe der RA z.B. ein neues Zertifikat angefordert oder die Revokation eines bestehenden initiiert werden. Die RA speichert diese und alle anderen Anfragen in speziellen Dateien ab, die dann von der Certification Authority weiterverarbeitet werden können. Die CA muß dabei nicht auf demselben Rechner installiert sein wie das RA-Modul. Es ist auch möglich, daß der Administrator die von der RA generierten

---

Anfragenfiles über ein Netzwerk oder auch von Hand mit Hilfe einer Diskette zu einem isolierten System transferiert, auf dem dann die CA arbeitet. Von der CA fertiggestellte Produkte können dann mit Hilfe der IS weiter bearbeitet werden: So werden neue Zertifikate z.B. mit Unterstützung der IS in einem LDAP-Verzeichnis abgelegt oder PIN-Briefe ausgedruckt.

Es ist möglich, daß in Zukunft noch weitere Funktionalität implementiert wird. In diesem Fall soll es möglich sein, das Administrationstool mit so wenig Aufwand wie möglich zu erweitern, so daß auch neue Module entsprechend verwaltet werden können.

Um die PKI so flexibel wie möglich zu gestalten kann sie, wie bereits erwähnt, als verteiltes System arbeiten. So kann die Certification Authority mehrere Rechnersysteme beanspruchen um Anträge entsprechend schneller abzuwickeln. Verteiltes System kann dabei auch bedeuten, daß auf einem Rechner mehrere Java Virtual Machines laufen, als auch die Beteiligung mehrerer Rechner mit wiederum mehreren Java Virtual Machines.

flexiTrust ist in der Version 1.0 bereits im Einsatz. Für Version 2.0 wird die Integration mit dem Administrationstool angestrebt. Zu diesem Zweck soll ein entsprechendes Servlet zusammen mit der Verwaltungslogik zum Einsatz kommen.

#### 4.1.2 Java Servlets

Eine zentrale Vorgabe für den Entwurf des Administrationstools ist die Verwendung von Java Servlets. Die Servlettechnologie stellt dabei die Schnittstelle zwischen statischen HTML-Dokumenten und der Java Programmiersprache dar, mit Hilfe von Servlets ist es also möglich dynamisch HTML-konforme Seiten zu erzeugen. Dies soll an einem kurzen Beispiel deutlich gemacht werden.

Wir nehmen eine einfache Anfrage eines Webbrowsers an einen Webserver an, in unserem Fall `http://www.informatik.tu-darmstadt.de/TT/Welcome.html`. Der Client fragt in diesem Fall beim Webserver „www.informatik.tu-darmstadt.de“ nach einem HTML-Dokument mit dem Namen „Welcome.html“ im Verzeichnis „TT“. Wenn ein solches Dokument existiert, wird es als Antwort vom Webserver zum Client gesandt. In diesem Fall liegt das fertige Dokument mit dem Namen „Welcome.html“ bereits abrufbereit auf dem Massenspeicher des Webserver und kann direkt versandt werden. Für das Verwaltungstool ist dieser Weg jedoch nicht geeignet, da hier Kommunikation mit einer Programmlogik stattfinden soll. Ohne eine Schnittstelle vom Webbrowser zu einem Programm wäre das Verwaltungstool darauf angewiesen in regelmäßigen Abständen HTML-Dokumente auf die Festplatte zu schreiben, welche dann vom Client abgerufen werden könnten. Mit Hilfe von Servlets ist es jedoch möglich, die Anfrage eines Clients an ein Java-Programm weiterzuleiten. Dort kann je nach Inhalt der Anfrage im Speicher eine entsprechende Antwort im HTML-Format kodiert werden, die dann direkt ohne Umweg über den Massenspeicher zurück an den Webbrowser gesandt wird.

---

Mit Hilfe von sog. CGI-Skripte oder -Programme war es auch bisher schon möglich Antworten auf Anfragen von Programmen direkt über Programmlogik konstruieren zu lassen. Dabei wird jedoch für jede Anfrage ein neuer Prozeß gestartet, was insbesondere bei Javaprogrammen mit einer Startzeit von fünf bis zehn Sekunden bis zur ersten Instruktionsausführung erhebliche Verzögerungen bedeutet. Das Servletkonzept dagegen basiert auf der Methode mit bereits im Speicher befindlichen Objekten zu kommunizieren. Dazu werden beim Laden des Webservers bereits die entsprechenden Klassen instanziiert, so daß diese dann ohne Verzögerung beim Eintreffen von Anfragen aktiv werden können.

Der Vorteil von Servlets besteht natürlich insbesondere darin, daß man durch diese Schnittstelle alle von Java zur Verfügung gestellten Features nutzen kann. Dies ist insbesondere deswegen von Bedeutung, da das Administrationstool mit Java-Objekten kommunizieren soll. Diese Verständigung ist natürlich mit viel geringerem Aufwand zu realisieren wenn der Kommunikationspartner ebenfalls in Java konstruiert werden kann. Es wäre zwar auch möglich, eine Verbindung zwischen in C und Java geschriebenem Programmcode zu realisieren, dieser Ansatz ist aber deutlich aufwendiger. Desweiteren sind zuverlässige Servlet-Container-Implementationen (siehe auch im folgenden Abschnitt) kostenlos verfügbar und werden bereits in vielen kritischen Bereichen (Bankwesen etc.) eingesetzt. Servlets beherrschen praktisch alle relevanten Formen der Kommunikation und Anfragen, so können neben dem gängigen HTTP-Protokoll Verbindungen auch SSL-gesichert (über HTTPS) hergestellt werden.

### 4.1.3 Java Virtual Machines

Eine der Schlüsselkomponenten von Java ist die virtuelle Maschine. Es handelt sich dabei um einen „virtuellen Computer“, der in Software implementiert ist auf einem „echten“ Rechner und seinem Betriebssystem läuft. Eine virtuelle Maschine interpretiert den Java-Bytecode und ermöglicht es damit den Java-Programmen auf beliebiger Hardware zu laufen. Durch die komplette Abstraktion aller Hardwarezugriffe kann das „Sandkasten“-Prinzip verwirklicht werden: Es kann von Java-Programmen nur auf die virtuelle Maschine zugegriffen werden, nicht auf die Hardware selbst. Dadurch ist das System vor eventuellen illegalen Zugriffen von Java-Code geschützt.

### 4.1.4 Tomcat Servlet-Container

Servlets werden von einem sog. „Servlet-Container“ verwaltet. Der Servlet-Container ist für das Starten der nötigen Servlets verantwortlich, er nimmt Requests von Webclients entgegen und gibt diese, verarbeitet und gekapselt in ein entsprechendes Request-Objekt, an das betreffende Servlet weiter.

Der Tomcat Servlet-Container ist die Referenzimplementierung der Java Servlet Technologie. Dieser Servlet-Container ist inzwischen in der Version 4.0 verfügbar und wird bereits in vielen Umgebungen eingesetzt. Tomcat ist ein Teil des Jakarta-Projektes [4]

---

und als solches eine Open-Source-Entwicklung, also frei verfügbar. Aufgrund dieser Eigenschaften und der Tatsache, daß Tomcat bereits in anderen Projekten im Rahmen der flexiTrust-Entwicklung benutzt wurde ist Tomcat als Servlet-Container für das Administrationswerkzeug gewählt worden.

Tomcat kann in Verbindung mit einem herkömmlichen Webserver (z.B. Apache) eingesetzt werden. In einem solchen Fall übernimmt Apache die Auslieferung der statischen Webseiten und leitet bestimmte Anfragen an Tomcat weiter. Die Kooperation kann auch daraufhin ausgeweitet werden, daß Apache die Authentifizierung des anfragenden Benutzers bewerkstelligt und nur überprüfte Anfragen an den Servlet-Container weitergereicht werden. In einem solchen Fall kann das Servlet durch eine entsprechende Anfrage auf ein evtl. vorhandenes X.509-Zertifikat zugreifen, um interne Abläufe den Berechtigungen des Anwenders anzupassen. Tomcat kann aber auch stand-alone eingesetzt werden: In diesem Fall kommuniziert der Webbrowser direkt mit Tomcat. Wenn also nur Servlets eingesetzt werden und kein Rückgriff auf evtl. in der Verbindung übermittelte X.509-Zertifikate nötig ist kann man auf Apache komplett verzichten und Tomcat sämtliche Anfragen beantworten lassen.

#### 4.1.5 RMI

Die „Remote Method Invocation“-Erweiterung ermöglicht es, Methoden in anderen Java Virtual Machines aufzurufen. Diese Virtual Machines können sowohl auf dem gleichen Rechner wie die eigene Maschine laufen, es ist aber auch möglich, daß sie sich auf einem anderen Rechner befinden. Um die von außen aufrufbaren Objekte zu verwalten werden alle in eine zentrale Registry eingetragen. Die Verwaltung der Registry übernimmt das Programm *rmiregistry*. Jedes Objekt wird durch einen eindeutigen Namen identifiziert und kann über diesen Namen, zusammen mit dem Namen oder der IP-Adresse des Rechners auf dem sie sich befindet, aufgerufen werden. So kann z.B. ein Objekt, welches in der Registry unter dem Namen `getStatus` auf dem lokalen Rechner installiert ist, über den Deskriptor `//localhost/getStatus` erreicht werden.

Diese besondere Art des Methodenaufrufes macht es nötig, daß weitere Exceptions abgefangen werden und daß die Parameter und Rückgabewerte besonderen Anforderungen genügen. Da die Kommunikation mit der aufzurufenden Methode über das Netzwerk erfolgt können hier Fehler auftreten, die beim Methodenaufruf innerhalb der eigenen Java Virtual Machine nicht möglich waren. Bei jeder „Remote Method Invocation“ müssen also entsprechende, RMI-spezifische Exceptions abgefangen werden. Da die Parameter und Rückgabewerte nun außerdem nicht mehr als Referenz übergeben werden können, müssen diese nun serialisiert werden damit sie via Netzwerk verschickt werden können. Alle zu übertragenden Klassen müssen dabei das Interface `java.io.Serializable` implementieren.

Obwohl über RMI auch Verbindungen auf demselben Rechner – oder innerhalb der gleichen Java Virtual Machine – realisiert werden ist dies nicht mit größeren Performanceeinbußen verbunden. Diese „Sonderfälle“ werden von RMI erkannt und entsprechend effizienter behandelt, so daß z.B. Methodenaufrufe in für denselben Rechner nicht über

---

das Netzwerkinterface geleitet werden.

## 4.2 Konzept

Die erste Realisierung verlagert den größten Teil der Funktionalität in die Komponenten und hält die „Intelligenz“ des Managementsystems, welches über ein Servlet realisiert wird, so gering wie möglich. Der Grund für diese Maßnahme ist die angestrebte möglichst große Flexibilität und Erweiterbarkeit des Administrationstools. Der Grundgedanke besteht darin, daß das Servlet nicht erweitert werden soll, sondern je nach gewünschter Zusatzfunktionalität lediglich neue Komponenten hinzugefügt werden müssen. Vorgefertigte Komponenten besitzen Funktionalität zum Bearbeiten von Properties, Archivieren von Logeinträgen und ähnliches. Das Servlet selbst verwaltet dabei nur die Liste aller im Netzwerk vorhandenen und zur Administration angemeldeten Rechner und leitet alle Anfragen einfach nur an diese weiter.

### 4.2.1 Der AdminNode

Der Aufbau eines AdminNodes gestaltet sich wie folgt: Auf jedem Rechner, der im Rahmen von flexiTrust verwaltet werden soll, muß genau ein AdminNode instanziiert werden. Dieser AdminNode verwaltet dann die verschiedenen Module die auf dem entsprechenden Rechner laufen. Der AdminNode selbst stellt den Endpunkt für die Kommunikation mit dem Administrationservlet dar, die Kommunikation wird auf Basis von (verschlüsseltem) RMI realisiert. Der AdminNode selbst kann die Anzahl der bei ihm registrierten Module zurückmelden, eine Liste der Namen der Module, eine Liste von Komponenten (Beschreibung einer Komponente siehe im entsprechenden Abschnitt) in einem benannten Modul, die registrierten Lognachrichten ab einem bestimmten Zeitpunkt sowie eine (HTTP-GET-)Anfrage vom Servlet an eine entsprechende Komponente weiterleiten.

### 4.2.2 Module

Ein Modul kennzeichnet einen logischen Teilbereich innerhalb des flexiTrust-Systems. Ein Modul wäre z.B. die Registration- oder die Certification Authority. Jedes Modul kann verschiedene Komponenten enthalten, so z.B. eine Komponente zur Verwaltung der Eigenschaften und eine zur Überwachung der Logfiles. Jedes Modul muß die abstrakte Klasse `de.tud.cdc.flexiTrust.Admin.Module` erweitern.

### 4.2.3 Komponenten

Ein Modul kann eine oder mehrere Komponenten beinhalten. Die Komponenten enthalten die eigentliche Funktionalität der Administrationsumgebung. Für die erste Version wurden eine Komponente zur Verwaltung der Properties, eine Komponente zur Logverwaltung sowie eine Komponente zum Zugriff auf Funktionen geplant. Die Namen der entsprechenden Klassen lauten „PropertyManager“, „LogInspector“ und „ServiceManager“. Die

---

---

Komponenten kommunizieren praktisch direkt mit dem Webbrowser, sie erhalten die GET-Anfrage vom Browser durch eine Weiterleitung über das Servlet und geben als Antwort fertiges HTML zurück. Die einzigen Methoden, die vom Servlet intern verwendet werden sind die Abfrage nach dem Namen der Komponente und eine Abfrage nach Lognachrichten. Jede Komponente muß die abstrakte Klasse `de.tud.cdc.flexiTrust.Admin.Component` erweitern.

#### 4.2.4 Verwalten von Properties

Das Verwalten von Properties wird über die Klasse „PropertyManager“ realisiert. Jede „Property“ hat einen Namen, einen Wert, einen Typ und eine Minimal- und Maximallänge für Strings. Mögliche Typen sind z.B. `Integer`, `Float` oder `String`. Die Definitionen für die einzelnen Properties werden in einem eigenen File als Properties gespeichert. Diese Properties werden dann dem Konstruktor der `PropertyManager`-Klasse übergeben. Ein mögliches Definitionsfile sähe z.B. folgendermaßen aus:

```
PropertyFile1=/usr/local/cdc/RAServlet.properties
PropertyFile1-Group1=Database Definitions
PropertyFile1-Group1-Element1=Servername, String, 128
PropertyFile1-Group1-Element2=Serverport, Integer, 5
PropertyFile1-Group1-Element3=Databasename, String, 128
PropertyFile1-Group1-Element4=Username, String, 32
PropertyFile1-Group1-Element5=Password, String, 32
PropertyFile1-Group2=Tomcat Settings
PropertyFile1-Group2-Element1=Verbositylevel, Integer, 2
PropertyFile1-Group2-Element2=Tomcatport, Integer, 5
PropertyFile2=/usr/local/cdc/BTD.properties
PropertyFile2-Group1=Batch Transfer Demon
PropertyFile2-Group1-Element1=Request Interval, Integer, 3
```

Dieses Definition deklariert zwei Property-Files. In dem ersten File, welches den Dateinamen `/usr/local/cdc/RAServlet.properties` besitzt, existieren zwei Gruppen, die Gruppe „Database Definitions“ mit den Elementen „Servername“, „Serverport“, „Databasename“, „Username“ und „Password“, die zweite Gruppe mit dem Namen „Tomcat Settings“ enthält die Elemente „Verbositylevel“ und „Tomcatport“. Das zweite File `/usr/local/cdc/BTD.properties` enthält nur die Gruppe „Batch Transfer Demon“ mit dem Element „Request Interval“.

#### 4.2.5 Authentifizierung via X.509 Zertifikat

Damit ein Administrator sich am System anmelden kann, muß er ein von der PKI ausgestelltes Zertifikat vorweisen können. Diese Zertifikate sind in der Praxis auf Chipkarten oder auch direkt auf der Festplatte gespeichert und werden vom Webbrowser an den Webserver – in diesem Fall Tomcat bzw. Apache – übermittelt. Dieser prüft die Gültigkeit und läßt den Anwender je nach Ergebnis entweder zugreifen oder weist ihn ab.

Die Umsetzung dieser Theorie in der Praxis ist mit einigem Aufwand verbunden. Der erste Schritt besteht in der Installation des Tomcat-Servlet-Containers. Im Bereich dieser

---

Arbeit wurde die Tomcat-Version 3.2.4 verwendet. Da Tomcat selbst wie bereits erwähnt nicht in der Lage ist Zertifikate aus SSL-geschützten Anfragen zu extrahieren, wird dafür der Apache Webserver eingesetzt, der zu diesem Zweck „zwischen“ Tomcat und dem Webbrowser plziert wird. Eingehende Anfragen werden somit zuerst von Apache empfangen. Sobald das Zertifikat überprüft und für gültig befunden wurde wird die Anfrage an Tomcat weitergeleitet. Mit Hilfe eines speziellen „Connectors“ kann Tomcat von Apache das bei der Anfrage verwendete Zertifikat erfragen und für weitere Authentifikationen und Sicherheitsmaßnahmen verwenden.

Aufgrund der verschiedenen möglichen Zertifikatskodierungen und der vielen verschiedenen zu installierenden Komponenten gestaltet sich die Installation eines solchen Setups vor allem für Anwender ohne Vorwissen relativ kompliziert. Zu diesem Zweck wurde ein entsprechender Einführungstext erstellt, der die Vorgehensweise beim Setup in allen Details aufzeigt (siehe Anhang).

#### **4.2.6 SSL-verschlüsselte RMI-Kommunikation**

RMI-Kommunikation zwischen dem Servlet und den einzelnen Komponenten kann im Prinzip einfach abgehört werden. Um dies zu verhindern soll die RMI-Kommunikation mit Hilfe von SSL abgesichert werden. Seit Veröffentlichung des Java Developer Kits Version 1.2 ist es möglich, eigene Socket-Factories für RMI vorzugeben und damit den gewünschten Effekt zu erreichen. Die Vorgehensweise wird von Sun in einem entsprechenden Tutorial [7] beschrieben.

### **4.3 Umsetzung**

Der erste Abschnitt beschreibt eine Beispielanwendung. Im zweiten Abschnitt werden die Klassen und ihre Funktionalität im Detail beschrieben.

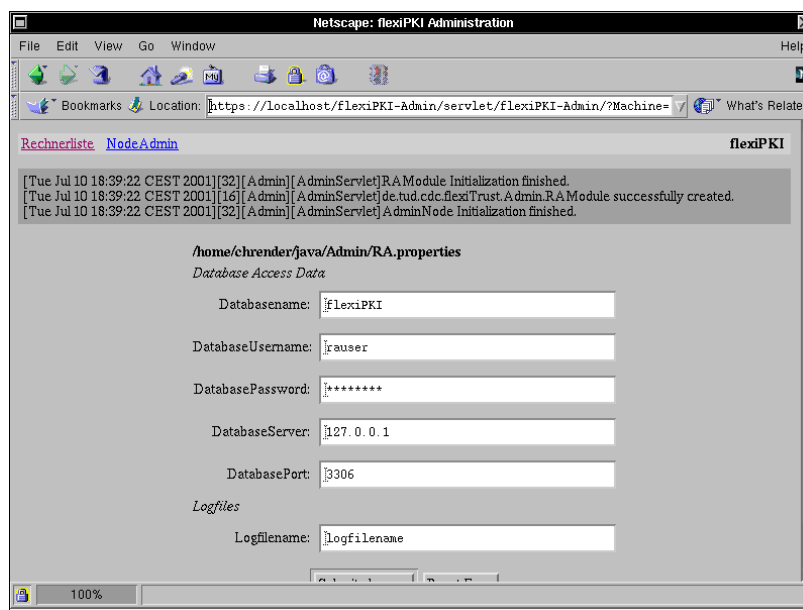
#### **4.3.1 Arbeitsweise**

Der Abschnitt „Konzept“ hat bereits die Komponenten des Administrationstools einzeln vorgestellt. Zum besseren Verständnis soll hier nun ein Überblick über die eigentliche Arbeitsweise mit dem Tool erfolgen.

Um überhaupt Anfragen an das Administrationstool richten zu können, ist ein entsprechend gültiges Zertifikat nötig. Der Prototyp läßt allerdings alle Zertifikate zu und gibt zu Testzwecken lediglich die wichtigsten Daten aus. Das Hauptmenü des Servlets zeigt eine Liste aller registrierten Rechner, ihrer Module und Komponenten an. Eine Navigationszeile am oberen Bildschirmrand erlaubt es jederzeit hierher zurückzukehren oder den „NodeAdmin“ aufzurufen, welcher das Hinzufügen oder Löschen einzelner Nodes erlaubt. Vom Hauptmenü aus lassen sich nun die einzelnen Module bzw. Komponenten anwählen. Die einzige gut zu verwendende Komponente innerhalb der ersten Realisierung stellt der PropertyManager dar. Es wird eine Liste aller Properties mit Markierungen für die Zugehörigkeit zu den entsprechenden Files und Property-Gruppen angezeigt. Der Administrator ändert die Werte je nach Bedarf und bestätigt über den „Submit“-Button.

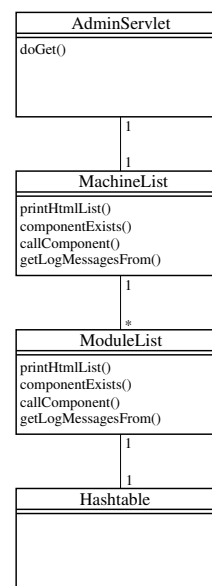
---

Sind die Werte gültig, so wird die Property-Liste erneut aufgebaut, falls nicht werden entsprechend Fehlermeldungen ausgegeben. Zur besseren Veranschaulichung sei hier eine Abbildung des PropertyManagers aufgeführt, danach wird die Arbeitsweise der einzelnen Komponenten im Detail beschrieben.



## AdminServlet

Das AdminServlet nimmt, wie schon im Kapitel „Konzept“ beschrieben, praktisch nur Verwaltungsaufgaben wahr. Es verwaltet eine Liste der zur Verfügung stehenden Rechner, Module und Komponenten. Dazu werden verschiedene Klassen verwendet, die das nebenstehende Diagramm zeigt. Zur Initialisierung liest das AdminServlet zunächst das File *NodeList* aus. Es wird jeweils ein **Vector** für die Namen der Rechner und einer für Benennung und/oder Kommentare belegt. Bei jeder **GET**-Anfrage des Browsers wird ein neues **MachineList**-Objekt erzeugt. Der Konstruktor der **MachineList**-Klasse benutzt dabei die Liste der Rechnernamen, um zu jedem AdminNode eine Verbindung via RMI aufzubauen. Wenn dies geglückt ist, wird für jeden Rechner ein Objekt vom Typ **ModuleList** angelegt. Dieses Objekt nimmt alle Informationen in Bezug auf die verfügbaren Module und Komponenten aus dem entsprechenden Rechner auf. Die Komponenten werden dabei als Array von Strings in eine Hashtabelle gespeichert, wobei der Name des Moduls den Schlüssel darstellt.



Neben der verwaltungstechnischen Funktionalität besitzen die Klassen `MachineList` und `ModuleList` noch Unterstützung zur Ausgabe der HTML-Formulare des Hauptmenüs. Das `AdminServlet` ruft zu diesem Zweck die entsprechenden Methode in `MaschineList` auf, welche dann die Aufgabe der Darstellung übernimmt.

### AdminNode

Wie im Abschnitt „Konzept“ bereits beschrieben wurde, stellt der `AdminNode` den Kommunikationsendpunkt für die Verbindung zum `AdminServlet` dar. Auf jedem Rechner muß ein eigener `AdminNode` gestartet werden. Dies kann z.B. durch folgende Kommandozeileneingabe geschehen:

```
java
  -Djava.rmi.server.codebase=file:/usr/local/cdc/Admin
  -Djava.security.policy=/usr/local/cdc/Admin/mypolicy
  -DModule1=de.tud.cdc.flexiTrust.Admin.RAModule
  de.tud.cdc.flexiTrust.Admin.AdminNodeImpl
```

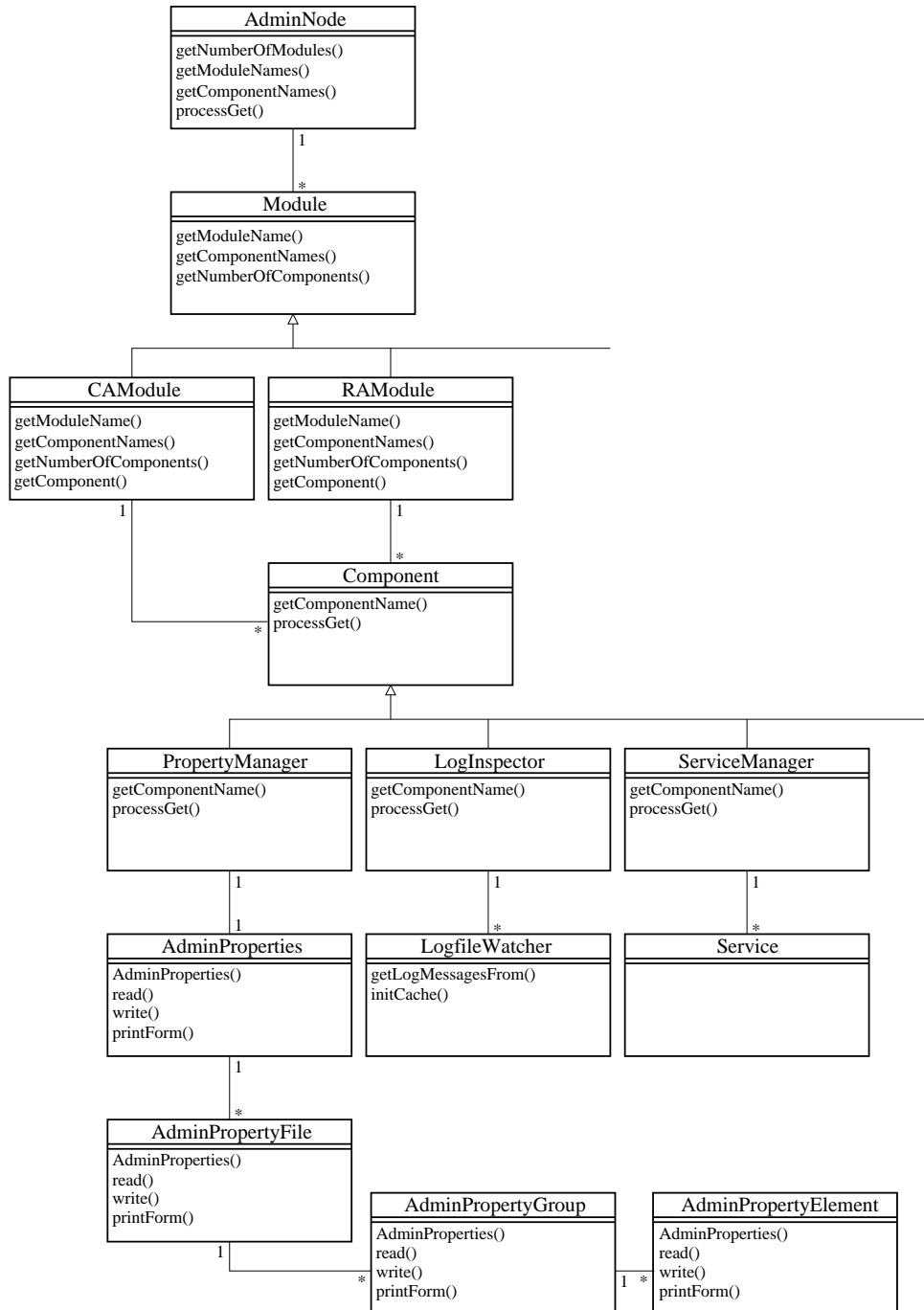
Die „codebase“-Option teilt dem Java-Interpreter dabei den Ablageort der Klassen mit und die Security-Policy stellt die nötigen Sicherheitseinstellungen her. Die Kommandozeilenswitches `-DModule<n>` geben dem `AdminNode` die Klassennamen der zu startenden Module an. In diesem Fall wird vom `AdminNode` also nur ein Modul instanziiert: Die Klasse `de.tud.cdc.flexiTrust.Admin.RAModule`. Diese Klasse erweitert wie alle anderen Module die Klasse `Module`. Jedes Modul instanziiert seine Komponenten im Konstruktor selbstständig. Im Falle der Klasse `RAModule` wird zunächst ein Property-File ausgelesen, dessen Inhalt dann von den Komponenten `PropertyManager`, `ServiceManager` und `LogInspector` ausgewertet werden. Diese Konfigurationsdatei könnte z.B. folgendermaßen aussehen:

```
PropertyFile1 = /usr/local/cdc/BTD.properties
PropertyFile1-Group1 = Batch Transfer Demon
PropertyFile1-Group1-Element1 = Request Interval, Integer, 3
LogFile1 = /home/chrenderer/java/Admin/mylog
LogFile1-Cache = 256 Entries
Service1 = Echo
Service1-Element1 = Filename, String, 48
Service1-Element2 = TextToEcho, String, 32
```

Die Komponenten nutzen diese Informationen um sich ebenfalls selbstständig zu initialisieren. Sobald alle Module und deren Komponenten gestartet wurden und die SSL-Kommunikation für die RMI-Verbindung vom `AdminNode` fertig initialisiert wurde, ist der Startvorgang für diesen `AdminNode` abgeschlossen. Seine weitere Aufgabe besteht nun darin, die Anzahl und Namen der von ihm zur Verfügung gestellten Module und Komponenten an das `AdminServlet` weiterzureichen sowie Requests vom Browser des Administrators an die Komponenten weiterzugeben.

Die folgende Seite zeigt die Verwendung von Klassen auf der Seite des `AdminNodes`.

---



## Der Propertymanager

Die Funktionsweise der eigentlichen `PropertyManager`-Klasse ist sehr einfach. Die bei der Initialisierung des Moduls eingelesenen Konfigurationsproperties – ein Beispiel wurde im vorangehenden Abschnitt gezeigt – werden vom `PropertyManager` direkt an den Konstruktor des `AdminProperties`-Objektes weitergegeben. Dieses Objekt wiederum legt für jedes Propertyfile ein eigenes Objekt vom Typ `AdminPropertyFile` an. Diese werden in einem `Vector` verwaltet. Jedes dieser Objekte erhält die Konfigurationsproperties, liest den Filenamen daraus und untersucht die Properties nach Gruppen. Für jede Gruppe wird jeweils eine Instanz der Klasse `AdminPropertyGroup` erstellt. Dieser parsen erneut die Konfigurationsangaben, speichern den Namen der Gruppe und erstellen wiederum Objekte vom Typ `AdminPropertyElement`, die dann die eigentlichen Properties mit Namen und Wert kapseln.

Wie schon die Klassen `MachineList` und `ModuleList` ist auch jede einzelne dieser Klassen dafür ausgelegt ihren Inhalt passend zum Einfügen in ein HTML-Formular auszugeben. Des weiteren ist die Klasse `AdminPropertyElement` in der Lage, aufgrund der Konfigurationsproperties die Gültigkeit der vom Administrator eingegebenen neuen Werte zu speichern.

### 4.3.2 Klassen

Dieser Abschnitt gibt einen Überblick über die verschiedenen Klassen die im Prototyp für den ersten Entwurf benutzt wurden. Es wird ein sehr kurzer Abriß über die Rolle der Klasse im System gegeben, gefolgt von einer Liste der verfügbaren Methoden.

#### AdminHTML

Die Klasse `de.tud.cdc.flexiTrust.Admin.AdminHTML` stellt Funktionen für die Ausgabe von HTML zur Verfügung. Diese Funktionen können von allen anderen Klassen genutzt werden und sollen dabei helfen, ein einheitliches Erscheinungsbild zu gewährleisten.

1. **printHeader** Ausgabe des HTML-Headers, der für alle vom Admin-Tool ausgelieferten Seiten gleich ist. Es sind alle Headerinformationen inklusive eines öffnenden `<body>`-Tags vorhanden.
  2. **printFooter** schreibt einen `</body>` und `</html>` Tag in die Ausgabe.
  3. **printNavigationBar** liefert die HTML-Navigationsleiste, die Links für die Rechnerliste und NodeAdmin enthält.
  4. **printLogWarningHeader** schreibt einen besonderen Header an den Beginn der HTML-Ausgabe, falls während der Arbeit oder seit dem letzten Ausloggen ein Fehler aufgetreten ist, der gleich angezeigt werden soll.
  5. **printLogWarningFooter** schreibt den abschließenden Teil der Warnung in die HTML-Ausgabe.
-

## AdminNode

Die Klasse `de.tud.cdc.flexiTrust.Admin.AdminNode` besteht aus einem Interface, welches nur zu RMI-Zwecken verwendet wird. Alle in diesem Interface definierten Methoden werden in der Klasse `AdminNodeImpl` implementiert, für eine genauere Beschreibung der Funktionen dieses Interfaces sei deswegen auf die Dokumentation der `AdminNodeImpl`-Klasse verwiesen.

## AdminNodeImpl

Die Klasse `de.tud.cdc.flexiTrust.Admin.AdminNodeImpl` implementiert das Interface `AdminNode`. Sie stellt den RMI-Verbindungspunkt mit dem `AdminServlet` dar. Das `AdminNode(-Impl)` Objekt verwaltet eine Liste aller auf dem eigenen Rechner instanziierten Module und Komponenten und stellt diese Information dem Administrationservlet zur Verfügung. Desweiteren können vom Administrator empfangene `HTTP-GET-Requests` an die entsprechende Komponente weitergeleitet werden.

Der Konstruktor initialisiert zunächst die SSL-verschlüsselte RMI-Kommunikation mit Hilfe der `RMISSLClientSocketFactory` und `RMISSLServerSocketFactory` Objekte. Danach wird versucht, mit Hilfe der Liste der zur Verfügung gestellten Modulnamen die entsprechenden Modulobjekte zu instanziiieren. Danach können Anfragen von außen entgegengenommen werden.

1. **getModuleNames** Diese Methode gibt in einem `String`-Array die Namen aller in diesem `AdminNode` registrierten Module zurück.
2. **getComponentNames** Diese Funktion gibt in einem `String`-Array die Namen aller Komponenten zurück, die zu einem benannten Modul gehören.
3. **getNumberOfModules** liefert die Anzahl aller registrierten Module zurück.
4. **getModule** liefert eine Referenz für ein benanntes Modul.
5. **processGet** Diese Funktion gibt eine vom Browser des Administrators empfangene und für eine Komponente bestimmte `GET`-Anfrage an die entsprechend benannte Komponente weiter.
6. **getLogMessagesFrom** Diese Methode liefert Lognachrichten von allen in diesem `AdminNode` registrierten Modulen zurück.

## AdminRes

Die Klasse `de.tud.cdc.flexiTrust.Admin.AdminRes` stellt vereinfachte Funktionalität bereit, um auf lokalisierte Strings zugreifen zu können, zusätzlich bietet sie die Möglichkeit, ein Property-File direkt in ein entsprechendes Objekt einzulesen. Der (statische) Initialisierer orientiert sich an der gesetzten „DefaultLocale“ und versucht die entsprechend lokalisierten Strings zu laden.

---

1. **getString** liefert den mit dem als Argument übergebenen Schlüssel zusammenhängenden lokalisierten String zurück.
2. **readPropertiesFromFile** benötigt als Argument den Namen des Files, aus dem die Properties gelesen werden. Die Funktion gibt das entsprechend aus dem File geparste `Properties`-Objekt zurück.

### AdminServlet

Die Klasse `de.tud.cdc.flexiTrust.Admin.AdminServlet` ist der primäre Verbindungspunkt zum Browser des Benutzers. Das `AdminServlet` initialisiert nach der Instanziierung zunächst die Liste aller Nodes, die kontaktiert werden sollen. Diese wird aus einem entsprechenden File gelesen.

1. **doGet** Diese Methode behandelt jede `GET`-Anfrage, welche vom Browser des Administrators eintrifft. In Abhängigkeit von der Anfrage wird entweder die Liste von Rechnern, Modulen und Komponenten ausgegeben, die Liste der Nodes kann verwaltet werden oder die Anfrage wird direkt via RMI an die entsprechende Komponente weitergeleitet.

### Component

Jede Komponente muß das Interface `de.tud.cdc.flexiTrust.Admin.Component` implementieren. Das Interface wird momentan von den Klassen `PropertyManager`, `LogInspector` und `ServiceManager` implementiert.

1. **getComponentName** Diese Funktion liefert den Namen der Komponente in einem String zurück.
2. **processGet** nimmt als Argument einen `HttpRequestContainer` entgegen und bearbeitet die darin enthaltene Anfrage vom Browser des Administrators. Das Ergebnis wird als HTML-Ausgabe in einem String zurückgegeben.
3. **getLogMessagesFrom** liefert die entsprechenden Einträge aus dem Log zurück, sofern diese Komponente die entsprechende Funktionalität aufweist.

### HttpRequestContainer

Eine Instanz der Klasse `de.tud.cdc.flexiTrust.Admin.HttpRequestContainer` kapselt die wichtigsten Informationen aus einer vom Browser versendeten Anfrage. Sinn und Zweck dieser Klasse besteht in ihrer Serialisierbarkeit, so daß im Gegensatz zur Klasse `HttpServletRequest`, welche nicht serialisierbar ist, Informationen via RMI vom `AdminServlet`-Objekt an die Komponenten Übergeben werden können. Zugriff auf die einzelnen Elemente geschieht über die public-Elemente „parameterlist“ und „requesturi“.

1. **public String getUrlParameterList** Dieser Funktion gibt einen URL-kodierten String zurück, welcher die in der Parameterliste enthaltenen Argumente zum Inhalt hat.
-

## LogEntry

Ein Objekt der Klasse `de.tud.cdc.flexiTrust.Admin.LogEntry` besitzt fünf public-Elemente: `timestamp`, `loglevel`, `module`, `theclass` und `message`. Ein neuer `LogEntry` kann entweder durch explizite Angabe jeder der fünf Eigenschaften erfolgen, oder er kann aus einem String heraus erzeugt werden.

## LogInspector

Die Klasse `de.tud.cdc.flexiTrust.Admin.LogInspector` erweitert die Klasse mit dem Namen `Component`. Der `LogInspector` erzeugt anhand der Property-Definitionen, die ihm über den Aufruf des Konstruktors übergeben werden, für jedes File einen `LogWatcher`. Die Properties zur Definition können z.B. folgendermaßen aussehen:

```
LogFile1 = /home/chrender/java/Admin/de/tud/cdc/flexiTrust/Admin/Test/mylog
LogFile1-Cache = 4 Entries
LogFile2 = /home/chrender/java/Admin/de/tud/cdc/flexiTrust/Admin/mylogfile
LogFile2-Cache = 8 Entries
```

Für jedes Logfile wird also der Filename zuzüglich einer Anzahl von zu puffernden Zeilen angegeben.

1. `getComponentName` liefert den String „LogInspector“ zurück.
2. `processGet` zeigt den Inhalt des momentan gespeicherten Caches für alle Files an.
3. `getLogMessagesFrom` liefert alle Lognachrichten ab einem bestimmten Zeitpunkt zurück.

## Logfile

Die `Logfile`-Klasse repräsentiert ein einzelnes Logfile, in das geschrieben werden soll. Um eine Instanz dieser Klasse zu erhalten, muß mindestens der Name des Logfiles angegeben werden. Optional können auch der Name des Moduls und/oder der Klassenname angegeben werden, die als Vorgabe bei Logeinträgen benutzt werden sollen. Da beim Loggen keine weitere Exception mehr auftreten darf, um Endlosschleifen zu vermeiden, muß der jeweilige Returncode analysiert werden. Liefert eine `write`-Methode nicht 0 zurück, so kann mit Hilfe der „`getLastError`“-Methode die genaue Art des Fehlers abgefragt werden.

1. `setDefaultClassName` setzt eine neue Vorgabe für den Klassennamen.
  2. `getDefaultClassName` liefert die Vorgabe für den Klassennamen in einem String zurück.
  3. `setDefaultModuleName` setzt die Vorgabe für den Modulnamen.
  4. `getDefaultModuleName` liefert die Vorgabe für den Modulnamen in einem String zurück.
-

5. **writeException** schreibt eine Exception in das Logfile. Dabei müssen zumindest ein Loglevel und die Exception selbst übergeben werden, zusätzlich können die Vorgaben für Klassen- und Modulname ersetzt werden.
6. **write** schreibt einen Logeintrag in das Logfile. Minimal müssen der Eintrag selbst und ein Loglevel angegeben werden, wie bei `writeException` können auch die Vorgaben für Klassen- und Modulnamen durch eigene Angaben ersetzt werden.
7. **getLastError** liefert eine Beschreibung für den zuletzt aufgetretenen Fehler.

### LogFileWatcher

Diese Klasse liefert Informationen über ein einzelnes Logfile. Zur Instanziierung werden der Name des Logfiles und die Menge der zu puffernden Zeilen angegeben. Diese werden dann im Konstruktor eingelesen.

1. **printCache** gibt den Inhalt des Caches in einen bereitgestellten StringBuffer aus.
2. **getLogMessagesFrom** liefert alle Lognachrichten ab einem bestimmten Zeitpunkt zurück.

### MachineList

Ein `MachineList`-Objekt wird vom `AdminServlet` verwendet, um Zugriff auf die verschiedenen `AdminNodes` zu erhalten. Dazu wird dem Konstruktor der `MachineList`-Klasse ein Vector mit Namen der zu kontaktierenden `AdminNodes` übergeben. Der Konstruktor versucht nun eine Verbindung zu jedem Rechner in der Liste herzustellen. Sobald dies gelungen ist, wird für jeden Rechner eine `ModuleList` angelegt, in der die Informationen zu den auf diesem Rechner vorhandenen Modulen gespeichert sind.

1. **printHtmlList** gibt eine Liste aller vorhandenen Rechner und ihrer Module bzw. Komponenten aus. Für die Ausgabe der Module wird die entsprechende Funktion der Klasse `ModuleList` verwendet.
  2. **componentExists** akzeptiert als Parameter einen Rechner- Modul- und Komponentennamen. Falls die benannte Komponente im entsprechenden Modul auf dem angegebenen Rechner existiert, wird `true` als Funktionswert zurückgeliefert, ansonsten ist das Ergebnis `false`.
  3. **callComponent** dient dazu, ein Objekt vom Typ `HttpRequestContainer` an eine Komponente via RMI zu übergeben. Benötigte Parameter sind der Rechner- Modul- und Komponentename sowie das `HttpRequestContainer`-Objekt.
  4. **getLogMessagesFrom** liefert alle Lognachrichten von allen registrierten Maschinen ab einem bestimmten Zeitpunkt zurück.
-

## Module

Jedes Modul muß die Klasse `de.tud.cdc.flexiTrust.Admin.Module` erweitern. Vorgegeben ist Funktionalität zum Verwalten der Komponenten, so daß die erweiternde Klasse nur noch Komponenten im Konstruktor instanziiieren und „components“-`Vector` hinzufügen muß, sowie die Methode `getModuleName` zu implementieren ist.

1. **`getModuleName`** liefert den Namen des Moduls als `String` zurück.
2. **`getComponentNames`** gibt die Namen aller Komponenten in einem `String-Array` zurück.
3. **`getNumberOfComponents`** gibt die Anzahl der im Modul registrierten Komponenten zurück.
4. **`getComponent`** liefert eine Referenz auf eine benannte Komponente.
5. **`getLogMessagesFrom`** liefert Lognachrichten aus allen registrierten Komponenten.

## ModuleList

Die Klasse `de.tud.cdc.flexiTrust.Admin.ModuleList` verwaltet die Namen von Modulen und Komponenten. Das `MachineList`-Objekt legt für jeden Rechner ein solches Objekt an. Der Konstruktor benötigt eine Referenz auf den `AdminNode`, dessen Module verwaltet werden sollen, und den Namen des Rechners, auf dem der `AdminNode` installiert ist. Mit Hilfe dieser Angaben wird eine Liste der Module und Komponenten des übergebenen `AdminNodes` aufgebaut.

1. **`printHtmlList`** schreibt ein `HTML-Menü` zum Zugriff auf die einzelnen Komponenten in einen der Funktion übergebenen `PrintWriter`.
2. **`doesComponentExist`** akzeptiert einen Modul- und Komponentennamen. Existiert die benannte Komponente innerhalb des entsprechenden Moduls, liefert die Funktion `true`, ansonsten `false` zurück.

## PropertyManager

Ein Objekt der Klasse `de.tud.cdc.flexiTrust.Admin.PropertyManager` verwaltet die `Properties` eines Moduls. Dazu verwendet es `AdminProperties`, `AdminPropertyFile`, `AdminPropertyGroup` und `AdminPropertyElement`.

1. **`processGet`** Ruft die `write`-Methode des `AdminProperties`-Objektes auf, falls Daten aktualisiert werden müssen. Danach werden die `read`- und `printForm`-Methode verwendet, um ein `HTML-Menü` für die `Properties` auszugeben.
  2. **`getComponentName`** liefert den `String` „Propertymanager“ zurück.
  3. **`getLogMessagesFrom`** gibt einen leeren `String` zurück.
-

### **RAModule**

Die Klasse `de.tud.cdc.flexiTrust.Admin.RAModule` ist eine Beispielimplementierung für ein Modul. Die Klasse erweitert die Klasse `Module`, der Konstruktor legt jeweils eine Instanz vom Typ `PropertyManager`, `ServiceManager` und `LogInspector` an.

1. `getModuleName` gibt den String „Registration Authority“ zurück.

### **RMISSLClientSocketFactory**

Die `de.tud.cdc.flexiTrust.Admin.RMISSLClientSocketFactory`-Klasse wird vom Paket „JSSE-Samples“ von SUN zur Verfügung gestellt. Die Klasse wird von einer Instanz der `AdminNodeImpl`-Klasse genutzt, um SSL-gesichert RMI-Verbindungen zu realisieren.

### **RMISSLServerSocketFactory**

Die `de.tud.cdc.flexiTrust.Admin.RMISSLServerSocketFactory`-Klasse wird vom Paket „JSSE-Samples“ von SUN zur Verfügung gestellt. Die Klasse wird von einer Instanz der `AdminNodeImpl`-Klasse genutzt, um SSL-gesichert RMI-Verbindungen zu realisieren.

### **Service**

Die abstrakte Klasse `de.tud.cdc.flexiTrust.Admin.Service` muß von jedem Service implementiert werden.

1. `getServiceName` soll den Namen des Services zurückliefern.
2. `processGet` soll die GET-Anfrage des Browsers behandeln.

### **ServiceManager**

Die Klasse `de.tud.cdc.flexiTrust.Admin.ServiceManager` ist nur ein Rahmenentwurf. Sie enthält keinerlei Funktionalität.

1. `processGet` gibt lediglich einen kurzen Teststring aus.
2. `getComponentName` liefert den String „ServiceManager“ zurück.
3. `getLogMessagesFrom` gibt einen leeren String zurück.

### **StaticLogfile**

Die Klasse `de.tud.cdc.flexiTrust.ad.StaticLogfile` ist der Klasse `Logfile` sehr ähnlich. Der einzige Unterschied besteht darin, daß alle Methoden statisch sind um somit jederzeit ohne Initialisierung aufgerufen werden können. Eine genauere Beschreibung der einzelnen Methoden findet sich unter der Beschreibung der Klasse `Logfile`.

---

## 4.4 Ende der Weiterführung

Die Existenz der Java Management Extensions hat einen komplett neuen Ansatz zur Umsetzung der Vorgaben wünschenswert gemacht. Dadurch diese Art der Verwaltung ist für das Management von Java-Applikationen ein Standard geschaffen worden der es unsinnig macht eigene Methoden zur Verwaltung zu konzipieren. Für Personen mit JMX-Vorkenntnissen ist nur noch eine minimale Einarbeitungszeit notwendig um mit einem auf diesem System basierenden Admin-Tool umzugehen, im Gegensatz zu einem selbstentwickelten, völlig neuen Ansatz. Da außerdem ein Großteil der angestrebten Funktionen durch JMX bereits realisiert werden, ist allein schon aus Stabilitäts- und Zerverlässigkeitsgründen der Einsatz einer erprobten Technik wünschenswert. Aus diesen Gründen wurde die Weiterentwicklung des ersten Ansatzes an dieser Stelle beendet.

---

---

## Kapitel 5

# Zweite Realisierung

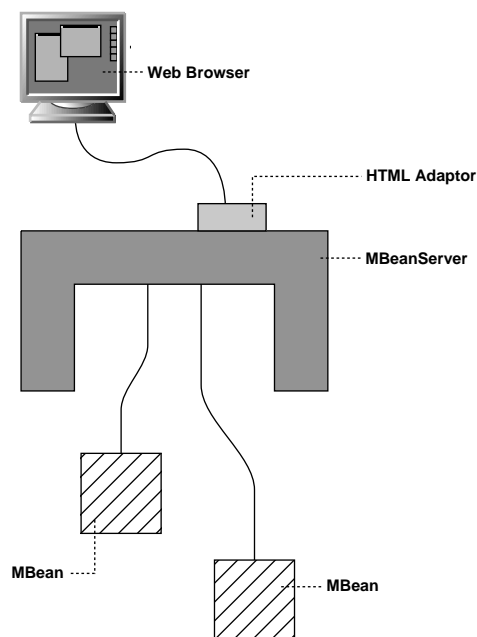
Die zweite Realisierung unterscheidet sich grundlegend von der ersten. Um sie verstehen zu können sei hier zunächst ein kurzer Überblick über die Java Management Extensions gegeben.

### 5.1 Bestehende Komponenten

#### 5.1.1 JMX

Die Java Management Extensions[5] wurden entwickelt um ein einfaches und übersichtliches Verwalten von Applikationen zu ermöglichen. Die Anforderungen, die JMX erfüllt, entsprechen zu einem großen Teil denen, die an das Administrationstool gestellt werden. So sind zwar einige Erweiterungen nötig, insgesamt wird aber ein Großteil der notwendigen Verwaltungsfunktionalität von JMX bereits zur Verfügung gestellt.

Das grundlegende Konzept von JMX besteht darin, daß an einem Server, dem sog. „MBeanServer“, eines oder mehrere der zu verwaltenden Objekte, die sog. „MBeans“, registriert werden. Diese Managable Beans können dann über Adapter, die ebenfalls an den MBeanServer gekoppelt werden, verwaltet werden. Das Bild auf der rechten Seite illustriert ein mögliches JMX-Setup. In diesem Fall sind zwei MBeans am MBeanServer registriert. Zugriff auf die Management-Funktionen ist über einen Webbrowser mit Hilfe des bei JMX mitgelieferten „HTML-Adaptor“ möglich.



Auf eine der zu verwaltenden Komponenten kann auf mehrere Arten Einfluß genommen werden: Erstens können „Attribute“ gelesen und/oder verändert werden. Ein Attribut hat einen Typ, nämlich eine Java-Klasse, einen Wert und es kann lesbar und/oder beschreibbar sein. Attribute sind von außen lesbare und/oder manipulierbare Variablen eines MBean-Objektes. Als zweites können Funktionen aufgerufen werden. Diese „Operationen“ besitzen einen Namen, eine Signatur, also eine Menge von Übergabeparametern, die jeweils einer gültigen Java-Klasse entsprechen müssen, sowie einen Rückgabewert, der ebenfalls eine Klasse sein muß. Zusätzlich zu den grade genannten Eigenschaften kann noch vermerkt werden ob die Operation eher als „Auskunfts“- oder „Manipulations“-Aktion zu verstehen ist. Drittens ist es noch möglich, Nachrichten, sog. „Notifications“ zu versenden. Von diesem Feature wird innerhalb dieser Arbeit allerdings kein Gebrauch gemacht.

Es gibt zwei verschiedene Arten von Manageable Beans: statische und dynamische. Zu jedem statischen MBean existiert muß ein Interface existieren welches die verfügbaren Attribute und Operationen beschreibt. Auf alle durch das Interface beschriebenen Elemente kann via JMX zugegriffen werden. Dadurch kann einfach kontrolliert werden, welche Elemente des MBeans von außen manipulierbar sein sollen: Nur die „öffentlichen“ Attribute und Funktionen werden im Interface beschrieben.

Der dynamische Ansatz verlangt mehr Arbeit vom Programmierer: Hier gibt das MBean selbst ein beschreibendes Objekt zurück, in dem alle nötigen Informationen über Attribute und Operationen enthalten ist. Soll nun ein Attribut manipuliert werden, wird eine entsprechende Funktion mit den nötigen Parametern aufgerufen (Name des Attributes, neuer Wert, etc.), ähnlich verhält es sich mit dem Aufruf einer Operation. Dynamische MBeans haben gegenüber statischen den Vorteil, daß sich die zu verwaltenden Elemente während dem Programmablauf ändern können. Ein weiterer nicht zu unterschätzender Vorteil besteht darin, daß zu den dynamischen MBeans nicht nur zum Bean an sich, sondern auch zu den Attributen und Operationen textuelle Beschreibungen mitangegeben werden können, welche dem Anwender die Orientierung im zu verwaltenden System stark erleichtern.

Das bestehende JMX-System erlaubt das Management von Applikationen innerhalb einer einzigen virtuellen Maschine. So ist es möglich, mit Hilfe des HTML-Adaptors neue MBeans zu instanzieren und im MBeanServer zu registrieren.

## 5.2 Konzept

Wie bereits erwähnt unterscheidet sich die zweite Realisierung grundlegend von der ersten. Während beim ersten Prototyp praktisch die gesamte Funktionalität in den einzelnen Komponenten implementiert wurde, ist sie beim zweiten Ansatz weiter verteilt.

Der zweite Ansatz setzt komplett auf JMX, den Java Management Extensions auf. Jede Komponente, die mit Hilfe des zweiten Ansatzes gesteuert werden soll, wird als

---

dynamisches MBean realisiert.

Im Prinzip ließe sich damit schon praktisch alles mit den beim JMX-Paket mitgelieferten Utilities erledigen. JMX bietet jedoch von sich aus noch keine Möglichkeit, MBeans auf mehreren Rechnern zu verwalten, kann auf keine MBeans innerhalb anderer Java Virtual Machines zuzugreifen und es existiert auch keine Unterstützung für Logging. Aus diesem Grund ist noch eine Erweiterung der Basis-JMX-Funktionalität nötig: das JMX-RMI-Paket.

### 5.2.1 Warum JMX?

Zum Einsatz von JMX haben zwei wesentliche Gründe beigetragen: Zum einen bietet dieses von Sun entwickelte Framework bereits einen Großteil der Funktionalität die beim Application Management verwendet werden soll. Das Verwalten von Eigenschaften erfüllt die Anforderungen an das Property-Management, und mit Hilfe der Funktionsaufrufe und dem Neuanbinden von MBeans im laufenden Betrieb kann der Starten und Stoppen von Diensten realisiert werden. Lediglich in Hinsicht auf das Logging wird keine Unterstützung bereit gestellt, für diesen Zweck können allerdings die Logging-Klassen aus dem ersten Ansatz praktisch unverändert weiterbenutzt werden.

Der zweite Grund ist die die Tatsache, daß durch die Bereitstellung einer Management-API von Sun bereits eine standardisierte Methode zur Applikationsverwaltung existiert. Es ist für neue, in das System noch nicht eingearbeitete Personen wesentlich einfacher den flexiTrust-Management-Prozeß zu verstehen, wenn dieser auf einer bekannten Technologie aufbaut. Da die im nächsten Abschnitt beschriebene JMX-RMI-Erweiterung die einzige grundlegende, aber dennoch sehr einfache Erweiterung am JMX-Framework darstellt, ist es lediglich notwendig, sich mit der Funktionalität im Servlet selbst vertraut machen zu müssen.

### 5.2.2 JMX-RMI

#### Überblick

Die mitgelieferten JMX-Komponenten erlauben es mit Hilfe des MBeanServers auf einem Rechner innerhalb einer virtuellen Maschine beliebige Manageable Beans zu starten, zu verwalten und zu stoppen. Diese Funktionalität wird nun mit Hilfe der Klassen im Pfad `de.tud.cdc.flexiTrust.ad.jmxrmi` so erweitert, daß mit Hilfe von RMI auf MBeans in weiteren virtuellen Maschinen auf dem gleichen Rechner oder auf einem anderen Rechner zugegriffen werden kann. Um die Erweiterungen möglichst kompatibel zum eigentlichen JMX-System zu halten und so viele Bestandteile wie möglich vom originalen JMX-Paket weiter nutzen zu können, werden MBeans in anderen virtuellen Maschinen über ein sog. „Proxy-MBean“ angebunden.

#### Funktionsweise

---

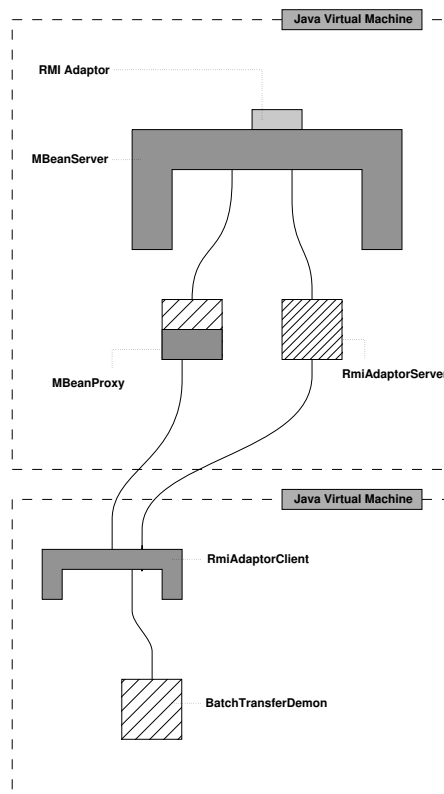
Im folgenden Abschnitt wird der Begriff „RMI-MBean“ für ein MBean verwendet, welches über die vom JMX-RMI-Paket bereitgestellte Funktionalität von einer anderen virtuellen Maschine und/oder einem anderen Rechner angebunden wird.

Ein wichtiger Unterschied zur „normalen“ Anbindung eines MBeans von einem MBeanServer aus besteht darin, daß ein RMI-MBean nicht direkt vom MBeanServer selbst instanziiert wird, sondern wie ein einfaches Java-Programm aufgerufen wird. Die `main`-Methode des entsprechenden MBeans erstellt dann die eigentliche Instanz der Klasse des MBeans und instanziiert weiterhin einen `RmiAdaptorClient`. Dieser `RmiAdaptorClient` bindet sich zunächst selbst in der `RmiRegistry` des aktuellen Rechners. Sobald dies gelungen ist wird versucht, eine Verbindung zum `RmiAdaptorServer` aufzubauen. Bei diesem handelt es sich um ein statisches MBean, welches beim Aktivieren des JMX-RMI-Frameworks direkt an den MBeanServer gebunden wurde. Wenn die Verbindungsaufnahme zu diesem MBean geglückt ist, wird die Methode `registerRmiMBean` aufgerufen, welche dafür zuständig ist, das RMI-MBean beim MBeanServer zu registrieren. Da jedoch keine direkte Verbindung zum angegebenen RMI-MBean besteht, wird nicht dieses direkt, sondern statt dessen ein `MBeanProxy` registriert. Dieser leitet dann alle Anfragen des MBeanServers via RMI wiederum an den `RmiClientAdaptor` weiter, welcher dann seinerseits die Anfragen an das eigentliche RMI-MBean schickt.

Auf diese Weise kommuniziert der MBeanServer mit Hilfe von zwei Proxy-Klassen über RMI, ohne daß am Server oder am dynamischen RMI-MBean selbst Änderungen vorgenommen werden müssen – mit Ausnahme der `main`-Methode im MBean, welche einen `RmiAdaptorClient` instanziiieren muß.

Da nur Funktionsaufrufe weitergeleitet werden ist es nicht möglich statische MBeans anzubinden, es ist nur die Anbindung von dynamischen MBeans möglich. Eine genauere Beschreibung der JMX-RMI-Funktionalität ist zusammen mit einigen Beispielen auf Seite 50 im Abschnitt „Admin-HowTo“ im Anhang enthalten.

Im Prinzip ist es damit möglich, MBeans auf beliebig vielen Rechnern an einen einzigen MBeanServer zu binden. Um die Ausfallsicherheit zu steigern ist es jedoch empfehlenswert, auf jedem Rechner jeweils einen MBeanServer zusammen mit einem `RmiAdaptorServer` zu installieren. Diese Aufgabe wird von der Klasse „`JmxAgent`“, die im nachfolgenden Kapitel beschrieben wird, wahrgenommen. Die eigentliche Verwaltungsfunktionalität – im `AdminServlet` – greift dann auf die einzelnen `JmxAgenten` auf den jeweiligen Rechnern zu.



### 5.2.3 JmxAgent

Die Funktionalität der JmxAgent-Klasse ist teilweise mit der AdminNode-Klasse aus dem ersten Entwurf zu vergleichen. Sie stellt einen Kommunikationsendpunkt für die Kommunikation mit dem Administrationsservlet dar. Desweiteren enthält sie eine Instanz der Klasse MBeanServer und bindet beim Start ein Objekt der Klasse RmiAdaptorServer daran, welches dafür sorgt daß RMI-MBeans an den MBeanServer gebunden werden können. Zum Abschluß der Initalisierung wird noch ein statisches MBean der Klasse „ClassInvoker“, deren Beschreibung im nächsten Abschnitt folgt, angebunden.

Neben den eben erwähnten Komponenten wird außerdem noch ein HTML-Adaptor intanziert und an den MBeanServer gebunden. Es ist also möglich, über den Port 8082 direkt und ohne „Umweg“ über das Administrationsservlet auf den MBeanServer zuzugreifen. Im späteren, endgültigen Betrieb kann diese Funktion deaktiviert werden, zu Testzwecken ist diese Funktionalität allerdings nützlich, da man so auch ohne die Apache/Tomcat/Servlet-Kombination Einfluß auf das Management System nehmen kann.

Wie der AdminNode des ersten Entwurfs muß auf jedem Rechner, auf den mit Hilfe des Administrationstools zugegriffen werden soll, ein JmxAgent laufen.

Der JmxAgent akzeptiert bestimmte Kommandozeilenparameter zum Starten von dynamischen und statischen MBeans – siehe Beschreibung des JmxAgents im Abschnitt „Umsetzung“ auf Seite 40.

### 5.2.4 ClassInvoker

Der ClassInvoker unterstützt den Administrator beim Starten der verschiedenen MBeans. Der ClassInvoker ist ein dynamisches MBean, welches in der Grundeinstellung zwei Eigenschaften und zwei Operationen besitzt: Die Properties „AntExecCommand“ und „JavaRunCommand“ sowie die Methoden „invokeClass“ und „ExecuteSystemCommand“. Mit Hilfe von „invokeClass“ kann eine Klasse über das Java-Systemkommando instanziiert werden, es kann also eine Klasse in einer eigenen virtuellen Maschine gestartet werden. Diese Operation ist sinnvoll für dynamische MBeans, die sich dann mit Hilfe des RmiClientConnector an den JMXAgent des lokalen Rechners binden. Mit Hilfe der zweiten Operation kann ein direktes Systemkommando aufgeführt werden. Diese Methode ist momentan nur für Testzwecke gedacht und kann in der endgültigen Version entfernt werden.

Der ClassInvoker liest beim Start eine Konfigurationsdatei aus. Es handelt sich dabei um ein Property-File, dessen Name in der AdminProperties-Klasse vorgegeben ist und in dem neue Klassen zum Starten vorgegeben werden können. Es existieren zwei Möglichkeiten eine neue Instanz einer Klasse zu starten. Erstens: Mit Hilfe von Ant. Dies wird im Konfigurationsfile durch `Job<n>-InvocationMethod = 1` kenntlich gemacht.

Beispiel für einen Auszug aus der Konfigationsdatei:

```
Job1 = startBatchTransferDemon
Job1-InvocationMethod = 1
Job1-AntArguments = -buildfile /home/chrender/FlexTmp/ra/run.xml btdemon
```

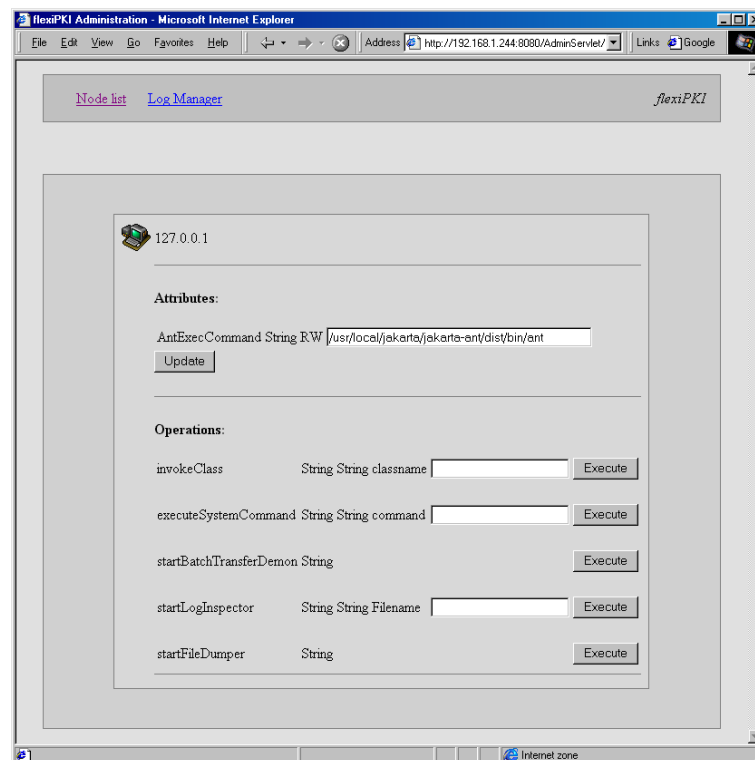
---

Die zweite Startmöglichkeit besteht im direkten Instanzieren der Klasse. Hierbei muß im Gegensatz zur Ant-Methode, bei der sich das Objekt selbst um die Registrierung kümmert, ein Objektname und die nötigen Konstruktor-Parameter mit angegeben werden.

Auch hier ein Beispiel für eine solche Konfiguration:

```
Job2 = startLogInspector
Job2-Name = flexiPKI:name=LogInspector,ViewInServlet=true
Job2-ClassName = de.tud.cdc.flexiTrust.ad.LogInspector
Job2-InvocationMethod = 2
Job2-ConstructorParameterType1 = String
Job2-ConstructorParameterName1 = Filename
```

Es folgt ein Beispiel für die Ausgabe des ClassInvokers mit drei im Konfigurationsfile definierten neuen Jobs:

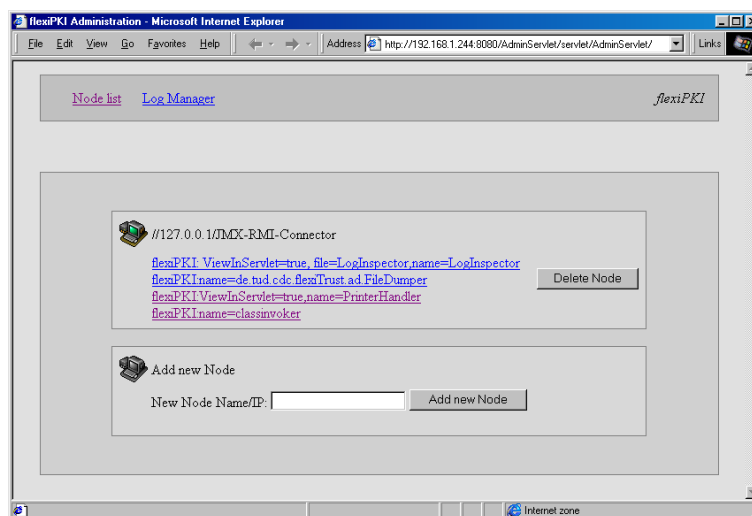


Die neuen in der Konfiguration definierten Jobs werden dem ClassInvoker als Operationen hinzugefügt und können genau wie die standardmäßig implementierten Operationen „invokeClass“ oder „ExecuteSystemCommand“ ausgeführt werden. Sinn und Zweck ist es, für jede neue vom Administrationstool zu startende Klasse einen entsprechenden Eintrag in der Konfiguration hinzuzufügen, so daß damit die neuen Klassen durch einen einzigen Klick im ClassInvoker gestartet werden können.

## 5.2.5 AdminServlet

Das AdminServlet ermöglicht den Zugriff auf die im Netzwerk installierten Managable Beans. Um auf die MBeans zugreifen zu können, ist es notwendig, eine Liste aller im Netzwerk zu verwaltenden Rechner, der sog. „Nodes“, zusammenzustellen. Diese Nodes werden dann im Hauptmenü des Servlets zusammen mit denen auf ihnen registrierten MBeans dargestellt.

Die folgende Abbildung zeigt ein Beispiel für die Node-Liste:



Durch einen Klick auf den entsprechenden Link werden Attribute und Operationen des gewählten MBeans angezeigt (vergleiche Abbildung im Abschnitt „ClassInvoker“). Ein bestehender Rechner kann durch einen Klick auf den „Delete“-Button aus der Liste entfernt werden, ein neuer kann durch Eintrag des Namens bzw. der IP-Adresse im Abschnitt „Add new Node“ hinzugefügt werden.

Der LogManager – erreichbar über die Navigationsleiste am oberen Bildschirmrand – ermöglicht es, alle registrierten LogInspector MBeans nach Einträgen in einem bestimmten Zeitraum suchen zu lassen.

## 5.3 Umsetzung

### 5.3.1 Arbeitsweise

#### JmxAgent

Wie bereits erwähnt stellt der JmxAgent den zentralen Knotenpunkt für die Kommunikation mit dem AdminServlet und den dynamischen MBeans fremder virtueller Maschinen dar. Nach dem Start werden zunächst die Kommandozeilenparameter analysiert. Dabei werden momentan folgende Optionen erkannt:

---

<code>-d &lt;classname&gt; &lt;objectname&gt;</code>	initialisiert ein dynamisches MBean in einer fremden virtuellen Maschine.
<code>-s &lt;classname&gt; &lt;objectname&gt;</code>	initialisiert ein statisches MBean in der virtuellen Maschine des JmxAgents.
<code>-ci &lt;classname&gt; &lt;arg1&gt; ... &lt;argn&gt;</code>	startet einen im Konfigurationsfile des ClassInvokers definierten ClassInvokerJob mit den entsprechend benötigten Argumenten.
<code>-li &lt;filename&gt;</code>	startet einen LogInspector, der auf der mit <code>&lt;filename&gt;</code> angegebenen Datei arbeitet.

Die Komponenten werden dabei genau in der Reihenfolge initialisiert, wie sie in der Kommandozeile angegeben wurden. Es folgt ein Beispiel für einen Aufruf eines JmxAgents:

```
java de.tud.cdc.flexiTrust.ad.JmxAgent
-d de.tud.cdc.flexiTrust.ad.demo.PrinterHandler
  "flexiPKI:name=PrinterHandler,ViewInServlet=true"
-s de.tud.cdc.flexiTrust.ad.FileDumper "flexiPKI:name=FileDumper"
-li /tmp/StaticLogfile
```

```
Starting job "de.tud.cdc.flexiTrust.ad.demo.PrinterHandler"
JmxAgent initialised
Waiting for: [flexiPKI:name=PrinterHandler,ViewInServlet=true]
[flexiPKI:name=PrinterHandler,ViewInServlet=true] has been bound.
Starting job "de.tud.cdc.flexiTrust.ad.FileDumper"
Starting job "/tmp/StaticLogfile"
```

Zunächst wird die Klasse `de.tud.cdc.flexiTrust.ad.demo.PrinterHandler` initialisiert. Dazu wird via `Runtime.getRuntime().exec` der Java-Interpreter aufgerufen und damit die entsprechende Klasse gestartet. Dann wird darauf gewartet, daß eine entsprechende Klasse mit Hilfe der `RmiAdaptorServer/RmiAdaptorClient` Objekte angemeldet wird. Da der Name, unter dem sich das MBean registriert, von diesem selbst festgelegt wird, muß er beim Aufruf des `JmxAgent` auf der Kommandozeile explizit angegeben werden. Sobald ein MBean mit entsprechendem Namen gefunden wurde („has been bound“) werden das statische MBean `de.tud.cdc.flexiTrust.ad.FileDumper` und der `LogInspector` für das File `/tmp/StaticLogfile` gestartet.

Neben dem Abarbeiten der Kommandozeile und der Übergabe der zu startenden Jobs an den `ClassInvoker` erstellt der `JmxAgent` außerdem einen `MBeanServer`, an den alle statischen und dynamischen MBeans dieses Rechners – auch über RMI – angebunden werden. Um die Anbindung über RMI zu ermöglichen wird außerdem noch ein `RmiAdaptorServer` an diesem `MBeanServer` gebunden. Bevor die auf der Kommandozeile

---

definierten, zu startenden Jobs an den `ClassInvoker` weitergegeben werden können, muß auch von diesem noch eine Instanz erstellt werden. Der `ClassInvoker` wird ebenfalls an den `MBeanServer` gebunden und ist damit über JMX-Anfragen zugänglich.

Damit ist die Initialisierung des `JmxAgents` abgeschlossen. Die weitere Bearbeitung des Managements wird von den vom `JmxAgent` erzeugten und verbundenen Klassen `MBeanServer`, `RmiAdaptorServer` und `ClassInvoker` überlassen.

Alle Logging-Ausgaben des `JmxAgents` werden in ein File umgeleitet, dessen Name in der Klasse `AdminProperties` definiert ist, normalerweise *JmxAgent.log*.

### **ClassInvoker**

Wie bereits erwähnt ist der `ClassInvoker` ein dynamisches `MBean`. Jede Instanz eines `JmxAgents` instanziiert immer einen `ClassInvoker`. Beim Aufruf des Konstruktors liest der `ClassInvoker` die Jobliste aus dem File und erstellt eine neue Jobliste aus `ClassInvokerJobs`-Objekten. Nach dem Einlesen wird ein `MBeanInfo`-Objekt erstellt, welches auf Anfrage an das JMX-System geliefert wird und Informationen über die beiden festen Operationen und alle im Konfigurationsfile definierten Jobs enthält.

### **5.3.2 Klassen**

Dieser Abschnitt gibt einen Überblick über die verschiedenen Klassen und die über sie verfügbaren Methoden. Es wird nur grob die Funktionalität beschrieben, für die genauere Erklärung sei auf die entsprechende Java-Dokumentation und die im Code selbst enthaltenen Erklärungen verwiesen.

#### **AdEnumeration**

Die Klasse `de.tud.cdc.flexiTrust.ad.AdEnumeration` implementiert die Klasse `Enumeration`. Ein Objekt dieses Types wird mit einem Array von Objekten initialisiert.

1. `hasMoreElements` gibt `true` zurück, wenn noch Objekte zum Abruf bereit liegen.
2. `nextElement()` liefert das nächste Objekt in der Reihenfolge zurück.

#### **AdHttpRequest**

Die Klasse `de.tud.cdc.flexiTrust.ad.AdHttpRequest` ist ein Wrapper für die Klasse `javax.servlet.http.HttpServletRequest`. Der Nachteil der Klasse im `JavaServlet`-Package besteht darin, daß keine neuen Parameter zum Request hinzugefügt werden können. Dies ist jedoch nötig, da das Servlet Framework nicht in der Lage ist, mit MIME-kodierten Requests umzugehen. Parameter, die auf diese Weise übertragen werden, können dann zwar vom `AdministrationServlet` dekodiert werden, aber nicht mehr zum eigentlichen Request hinzugefügt werden. In diesem Fall wird das eigentliche Request-Objekts durch ein Objekt der Klasse `AdHttpRequest` ersetzt. Der Konstruktor dieser Klasse nimmt ein `HttpServletRequest`-Objekt entgegen und leitet die

---

meisten Funktionsaufrufe intern direkt an dieses weiter. Lediglich `getParameter` und `getParameterNames` werden intern anders behandelt.

1. **`addNewParameter`** erlaubt es, neue Parameter zum Request hinzuzunehmen.

### **AdminProperties**

Diese Klasse enthält die wichtigsten Klassen und Pfadangaben für das Administrationspackage. Die Namen von Klassen, die über den `ClassInvoker` instanziiert werden können, die Kommandozeilen-Syntax oder der Klassenpfad können definiert werden.

### **AdminServlet**

Die Klasse `de.tud.cdc.flexiTrust.ad.AdminServlet` stellt das Frontend für die flexiTrust Administration dar. Das Servlet kommuniziert intern über RMI und JMX mit den eigentlichen zu verwaltenden Prozessen/Threads. Es nimmt Eingaben vom Browser entgegen und liefert Antworten in HTML zurück.

1. **`doPost`** nimmt „Post“-Requests vom Browser des Administrators entgegen. Requests dieser Art werden verschickt, wenn Files vom Browser zum Servlet geschickt werden sollen. Die MIME-kodierten Parameter werden dekodiert, der Request entsprechend modifiziert und an die `doGet`-Methode weitergeleitet. Weitere Details werden im Abschnitt „Upload von Files“ behandelt.
2. **`doGet`** enthält die eigentlich Funktionalität des Servlets. Je nach Anfrage werden Operationen gestartet, der `LogManager` aufgerufen und ähnliches.

### **ByteArray**

`de.tud.cdc.flexiTrust.ad.ByteArray` ist ein Objekt-Wrapper für ein `ByteArray`. Diese Klasse wird verwendet um File-Uploads zu markieren. Siehe Abschnitt „Upload von Files“.

### **ClassInvoker**

Ein Objekt der Klasse `de.tud.cdc.flexiTrust.ad.ClassInvoker` wird vom `JmxAgent` beim Start an den `MBeanServer` gebunden. Es handelt sich bei dieser Klasse um ein statisches `MBean` dessen Zweck darin besteht, andere `MBeans` – evtl. auch in einer eigenen virtuellen Maschine – zu instanziiieren.

1. **`invokeClass`** erzeugt ein neues Objekt anhand des übergebenen Klassennamens.
  2. **`executeSystemCommand`** führt das als String übergebene Systemkommando aus.
  3. **`getAttribute`** liefert auf Anfrage das Kommando zum Aufruf von ant zurück.
  4. **`getAttributes`** gibt ein oder mehrere Attribute zurück. In der Praxis beschränkt auf das Attribut „`AntExecCommand`“.
-

5. **getMBeanInfo** liefert das MBeanInfo-Objekt für den ClassInvoker zurück.
6. **invoke** führt eine der Operationen zum Starten von Klassen durch. Die Operationen „executeSystemCommand“ und „invokeClass“ sind fest vorgegeben, alle weiteren bestimmen sich durch den Inhalt der Daten „ClassInvokerJobs“, die im Abschnitt „ClassInvoker“ genauer beschrieben wird.
7. **setAttribute** erlaubt es, daß Attribut „AntExecCommand“ neu zu setzen.
8. **setAttributes** erlaubt es eines oder mehrere Attribute zu setzten, die Möglichkeiten beschränken sich allerdings lediglich auf das Attribut „AntExecCommand“.

### **ClassInvokerJob**

Ein Objekt der Klasse `de.tud.cdc.flexiTrust.ad.ClassInvokerJob` beinhaltet alle Informationen über einen Job, die über den ClassInvoker gestartet werden können. Der Informationen werden direkt aus den Properties gelesen, die dem ClassInvoker zur Verfügung gestellt werden.

### **FileDumper**

Die Klasse `de.tud.cdc.flexiTrust.ad.FileDumper` ist ein statisches MBean, die es erlaubt, hochgeladene Files als String auszugeben. Sie dient zum Testen der Upload-Funktion.

1. **dumpfile** ist eine Operation, die ein ByteArray als Argument erhält. Die Klasse ByteArray zeigt dem AdminServlet an, daß ein File vom Browser heraufgeladen werden soll. Dieses ByteArray wird intern in einen String gewandelt und direkt ausgegeben.

### **FileDumperMBean**

`de.tud.cdc.flexiTrust.ad.FileDumperMBean` liefert für den MBeanServer Informationen über das statische MBean FileDumper.

### **JmxAgent**

Die Klasse `de.tud.cdc.flexiTrust.ad.JmxAgent` initialisiert einen MBeanServer und bindet einen RmiAdaptorServer, HtmlAdaptor sowie einen ClassInvoker daran. Die Klasse wird genauer im Abschnitt „JmxAgent“ beschrieben.

1. **main** führt die oben beschriebene Initalisierung durch.
-

## LogEntry

Die `de.tud.cdc.flexiTrust.ad.LogEntry`-Klasse stellt einen Container für einen Logeintrag zum Transport vom `LogInspector` zum `AdminServlet` dar. Der Eintrag kann entweder durch direktes Einlesen aus einem File oder durch explizite Angabe aller Parameter erzeugt werden.

## LogInspector

Die Klasse `de.tud.cdc.flexiTrust.ad.LogInspector` überwacht ein einzelnes Logfile. Die Größe wird als Read-Only-Attribut an das Servlet zurückgeliefert, des weiteren kann das Log nach Einträgen durchsucht werden.

1. **getHTMLLogMessages** liefert die Lognachrichten in einem bestimmten Zeitraum als HTML-Output zurück.
2. **getLogMessages** liefert einen Vector aus `LogEntry`-Objekten zurück, die die gesuchten Logeinträge enthalten.

## Logfile

Die `Logfile`-Klasse repräsentiert ein einzelnes Logfile, in das geschrieben werden soll. Um eine Instanz dieser Klasse zu erhalten, muß mindestens der Name des Logfiles angegeben werden. Optional können auch der Name des Moduls und/oder der Klassenname angegeben werden, die als Vorgabe bei Logeinträgen benutzt werden sollen. (`getLasterror`, `flush`, `loglevel`)

1. **setDefaultClassName** setzt eine neue Vorgabe für den Klassennamen.
  2. **getDefaultClassName** liefert die Vorgabe für den Klassennamen in einem String zurück.
  3. **setDefaultModuleName** setzt die Vorgabe für den Modulnamen.
  4. **getDefaultModuleName** liefert die Vorgabe für den Modulnamen in einem String zurück.
  5. **writeException** schreibt eine Exception in das Logfile. Dabei müssen zumindest ein Loglevel und die Exception selbst übergeben werden, zusätzlich können die Vorgaben für Klassen- und Modulname ersetzt werden.
  6. **write** schreibt einen Logeintrag in das Logfile. Minimal müssen der Eintrag selbst und ein Loglevel angegeben werden, wie bei `writeException` können auch die Vorgaben für Klassen- und Modulnamen durch eigene Angaben ersetzt werden.
  7. **getLastError** liefert eine Beschreibung für den zuletzt aufgetretenen Fehler.
-

### StaticLogfile

Die Klasse `de.tud.cdc.flexiTrust.ad.StaticLogfile` ist der Klasse `Logfile` sehr ähnlich. Der einzige Unterschied besteht darin, daß alle Methoden statisch sind um somit jederzeit ohne Initialisierung aufgerufen werden können. Eine genauere Beschreibung der einzelnen Methoden findet sich unter der Beschreibung der Klasse `Logfile`.

### PrinterHandler.java

Die Klasse `de.tud.cdc.flexiTrust.ad.demo.PrinterHandler` wird vom Admin-HowTo im Anhang zur Demonstration der Funktion eines dynamischen MBeans verwendet. Für eine detaillierte Beschreibung sei auf den entsprechenden Abschnitt im Anhang verwiesen.

### MBeanProxy

Die Klasse `de.tud.cdc.flexiTrust.ad.jmxrmi.MBeanProxy` ist ein Interface, welches nur zu RMI-Zwecken verwendet wird. Die in dieser Klasse definierten Methoden „`rmiHandleNotification`“ und „`unregisterRmiMBean`“ werden in der Klasse `MBeanProxyImpl` implementiert. Für eine genauere Beschreibung sei auf diese Klasse verwiesen.

### MBeanProxyImpl

Die Klasse `de.tud.cdc.flexiTrust.ad.MBeanProxyImpl` ist ein dynamisches MBean, welches in seiner Funktion als Proxy alle Anfragen an das eigentliche MBean weiterleitet. Dazu werden die entsprechenden Funktionen im `RmiClientAdaptor` in einer anderen virtuellen Maschine, mit der dieses Objekt via RMI verbunden ist, weitergeleitet.

1. **getAttribute** erlaubt es, ein Attribut im verbundenen MBean auszulesen.
2. **getAttributes** dient dazu, ein oder mehrere Attribut im verbundenen MBean zu lesen.
3. **getMBeanInfo** liefert das zugehörige MBeanInfo-Objekt des via RMI verbundenen Objekts zurück.
4. **invoke** ruft eine Operation im verbundenen dynamischen MBean auf.
5. **setAttribute** dient dazu, ein Attribut im verbundenen MBean zu setzen.
6. **setAttributes** erlaubt es, eines oder mehrere Attribute im verbundenen MBean zu setzen.

### RmiAdaptorClient

`de.tud.cdc.flexiTrust.ad.jmxrmi.RmiAdaptorClient` ist ein Interface, welches für die RMI-Funktionalität gebraucht wird. Das Interface wird in der Klasse `RmiAdaptorClientImpl` implementiert. Für eine genauere Beschreibung sei auf diese Klasse verwiesen.

---

### RmiAdaptorClientImpl

Die Klasse `de.tud.cdc.flexiTrust.ad.jmxrmi.RmiAdaptorClientImpl` dient dazu, ein dynamisches MBean an einen in einer anderen virtuellen Maschine laufenden MBean-Server anzubinden.

1. **getAttribute** liefert ein benanntes Attribut vom verbundenen MBean an den entsprechenden MBeanServer zurück.
2. **getAttributes** dient dazu, ein oder mehrere Attribute im dynamischen MBean zu lesen.
3. **getMBeanInfo** liefert das zugehörige MBeanInfo-Objekt des verbundenen Objekts zurück.
4. **invoke** ruft eine Operation im verbundenen dynamischen MBean auf.
5. **setAttribute** dient dazu, ein Attribut im verbundenen MBean zu setzen.
6. **setAttributes** liefert ein oder mehrere Attribute aus dem dynamischen MBean zurück.

### RmiAdaptorServer

`de.tud.cdc.flexiTrust.ad.jmxrmi.RmiAdaptorServer` ist ein Interface, welches für die RMI-Funktionalität gebraucht wird. Das Interface wird in der Klasse `RmiAdaptorServerImpl` implementiert. Für eine genauere Beschreibung sei auf diese Klasse verwiesen.

### RmiAdaptorServerImpl

Die Klasse `de.tud.cdc.flexiTrust.ad.jmxrmi.RmiAdaptorServerImpl` implementiert `RmiAdaptorServer`, `CommunicatorServerMBean` und `RmiAdaptorServerImplMBean`. Die Klasse `RmiAdaptorServer` erlaubt es, auf die implementierende Klasse mit Hilfe von RMI – von der Klasse `RmiAdaptorClient` aus – zuzugreifen, die Klasse `RmiAdaptorServerImplMBean` ermöglicht es, das Objekt als statisches MBean an den MBeanServer anzubinden. Das Implementieren der Klasse `CommunicatorServerMBean` ermöglicht es der Klasse `RmiAdaptorServerImpl`, neue MBeans selbstständig am MBeanServer zu registrieren.

1. **queryNames** Liefert ein `Set` aller Namen von MBeans, die an den MBeanServer dieses Rechners gebunden sind.
  2. **getProtocol** Gibt RMI als String zurück.
  3. **getState** Liefert den (Online-)Status des `RmiAdaptorServer` als `int`-Konstante zurück. Mögliche Rückgabewerte sind entweder `CommunicatorServer.ONLINE` oder `CommunicatorServer.OFFLINE`.
  4. **getStateString** Liefert den (Online-)Zustand als String zurück. Mögliche Rückgabewerte sind entweder `ONLINE` oder `OFFLINE`.
-

5. **isActive** Liefert **true**, wenn der RmiAdaptorServer in der RMI-Registry gebunden ist und so von außen erreicht werden kann.
6. **waitState** Wartet, bis der (Online-)Status des RmiAdaptorServer den gewünschten Zustand hat oder die angegebene Zeit verstrichen ist.
7. **getHost** Liefert immer einen leeren String zurück.
8. **getPort** Gibt immer -1 zurück, da im RMI-Protokoll keine Ports verwendet werden.
9. **start** Startet den RmiAdaptorServer – bindet ihn in der RMI-Registry, so daß Zugriffe von außen möglich sind.
10. **stop** Stoppt den RmiAdaptorServer, indem die Anbindung an die RMI-Registry entfernt wird.
11. **setPort** Der Aufruf dieser Methode hat keine Wirkung, da Ports im RMI-Protokoll nicht benutzt werden.
12. **getMBeanInfo** liefert das beschreibende MBeanInfo-Objekt für das MBean mit dem angegebenen Namen zurück.
13. **getAttribute** liefert das gewünschte Attribut des angegebenen MBeans zurück.
14. **setAttribute** setzt ein Attribut in dem angegebenen, am MBeanServer angebotenen MBean.
15. **invoke** ruft eine in einem am MBeanServer angebotenen MBean vorhandene Methode mit der entsprechenden Signatur und den übergebenen Parametern auf.
16. **registerRmiMBean** versucht ein MBean mit dem übergebenen **ObjectName** am MBeanServer zu registrieren. Dabei wird ein MBeanProxy eingerichtet, der seinerseits versucht, den ans MBean gebundenen RmiAdaptorClient über den übergebenen **rminame** zu kontaktieren.
17. **getRmiName** liefert `//127.0.0.1/JMX-RMI-Connector` zurück. Der Name ist abhängig von dem in der Klasse `AdminProperties` definierten Defaultwert.
18. **setRmiName** erlaubt es, den RMI-Namen des RmiAdaptorServer zu ändern.

### **RmiAdaptorServerImplMBean**

Die Klasse `de.tud.cdc.flexiTrust.ad.RmiAdaptorServerImplMBean` enthält die Informationen über das statische MBean `RmiAdaptorServerImpl`. Diese Klasse wird vom `MBeanServer` abgefragt, um Informationen über die via JMX verfügbare Funktionalität zu erhalten.

---

## Kapitel 6

# Zusammenfassung

Jede der Realisierungen besitzt ihre eigenen Vor- und Nachteile. So ist der erste Ansatz in Hinsicht auf die Ausbaufähigkeit der flexibelste. Durch die Tatsache, daß das Servlet der ersten Realisierung nur die notwendigsten Maßnahmen selbst durchführt und alle anderen Anfragen an frei definierbare Komponenten weiterleitet kann im Prinzip jede beliebige Funktionalität realisiert werden. Der zweite Ansatz, welcher auf JMX aufbaut, erlaubt es zwar, den größten Teil der nötigen Funktionen mit Hilfe des vorgegebenen Management-Frameworks zu erledigen, bestimmte Funktionen, wie z.B. der Upload von Files, müssen jedoch mit nicht-JMX-konformen – und nicht-Servlet-konformen – Maßnahmen realisiert werden. So ist der Upload beim zweiten Ansatz nur über das Servlet an sich möglich, über den mitgelieferten HTML-Adaptor kann dieser Schritt jedoch nicht realisiert werden.

Die zweite Art der Realisierung bedingt außerdem einen höheren Programmieraufwand am Servlet selber. Dies ist insbesondere bei möglichen Erweiterungen von Bedeutung, da so das Servlet selbst verändert werden muß. Im Gegensatz dazu genügt es bei der ersten Form der Realisierung, eine neue Komponente mit der entsprechenden Funktion zu erstellen und diese dem Framework hinzuzufügen. Das Servlet sowie die anderen Komponenten können ohne Änderung weiterverwendet werden.

Die Anpassung der zweiten Realisierung an vorgegebene – und bereits eingesetzte – Standards rechtfertigt allerdings den etwas höheren Arbeitsaufwand durchaus. Tools anderer Herkunft können im Prinzip mit der nun bestehenden JMX-Infrastruktur zusammenarbeiten. Bestimmte Funktionalität wie z.B. der PropertyManager aus dem ersten Ansatz ist durch JMX bereits vorhanden. Dadurch, daß ein größerer Teil des verwendeten Codes bereits seit längerer Zeit im Einsatz ist, sind weniger Fehler zu erwarten als dies bei einer vollständigen Neuimplementierung der sich überschneidenden Funktionalität zu erwarten wäre. Ein nicht zu unterschätzender Vorteil besteht außerdem darin, daß ein Anwender mit JMX-Kenntnissen beim zweiten Ansatz sehr viel schneller in der Lage ist das bestehende System zu verstehen und zu erweitern.

---

## Anhang A

# Admin-HowTo

### A.1 Overview

This paper is supposed to be a quick overview describing the flexiPKI Administration tools and their setup. It gives a – very short – introduction to JMX, the Java Management Extensions, shows the Enhancements that were necessary to make JMX usable in the flexiPKI Environment and gives a quick tutorial how to make ones applications manageable via JMX and thus the Administration tool. In order to make the explanations as short as possible, this text relies heavily on examples instead of formal definitions.

### A.2 Java Management Extensions

The Java Management Extensions provide a concept to make objects, applications, applets and similar easier manageable. All manageable ressources are named “MBeans” – Manageable Beans. Two types of MBeans are of interest in our context: The Static MBean and the Dynamic MBean. To explain the two different kinds, we take a look at an example: a java object that is handling printers. It might look like this:

```
public class PrinterHandler {
    private int sheetsofpaperleft;           // How much paper is left
    private int sheetsprintedsofar;         // Total number of papersheets printed
    private int maximumnuberofjobsinqueue; // How many jobs we will queue at most
    private Vector jobs;                    // A Vector containing the PrinterJobs

    public int getSheetsOfPaperLeft() { ... }
    public int getSheetsPrintedSoFar() { ... }
    public int getMaximumNumberOfJobsInQueue() { ... }
    public void setMaximumNumberOfJobsInQueue(int numberofjobs) { ... }
    public void setOnline(boolean newmode) {...}
    public void addPrinterJob(String texttoprint) { ... }
    public void removePrinterJob(int jobid) { ... }
    public String getJobList() { ... }
}
```

---

This very simple example printer object can tell you how much paper is left for printing by calling the `getSheetsOfPaperLeft()` function, it can tell you how many sheets have been used by printing so far with the `getSheetsPrintedSoFar()` method, we can read the maximum number of printer jobs the object will queue with `getMaximumNumberOfJobsInQueue()` and set a new value for the maximum number with `setMaximumNumberOfJobsInQueue(int numberOfjobs)`. It is possible to set the printer offline with `setOnline(false)` and online with `setOnline(true)`. We can add new jobs, remove jobs and get a list of the currently queued jobs with `addPrinterJob(String texttoprint)`, `removePrinterJob(int jobid)` and `getJobList()`. Now we want to make this object manageable via JMX.

### A.2.1 Static MBeans

The static approach is the simpler one of the two possible: We just create an interface with the name of the class and “MBean” appended. For our printer example object, the interface would look like this:

```
public interface PrinterHandlerMBean {
    public int getSheetsOfPaperLeft();
    public int getMaximumNumberOfJobsInQueue();
    public void setMaximumNumberOfJobsInQueue(int numberOfjobs);
    public void setOnline(boolean newmode);
    public void addPrinterJob(String texttoprint);
    public void removePrinterJob(int jobid);
    public String getJobList();
}
```

When registering the `PrinterHandler` with the JMX, the system checks for the existence of an interface named “`PrinterHandlerMBean`”. If the search is successful as in our example for the static MBean, JMX “looks into” the interface to determine what functionality can be used for management. In our first example, we only have Attributes and Operations. JMX also supports Notifications and Constructors which are not covered in this introduction.

Attributes can be Read-only, Write-only or Read- and Writeable. JMX searches through all prototypes in your interface and defines an Attribute if it can find a “set” or “get” method. In our example, our interface gives access to three attributes:

1. `SheetsOfPaperLeft`: a read-only attribute, since there is only a “get” method for this attribute, no “set” method.
2. `JobList`: A read-only attribute.
3. `MaximumNumberOfJobsInQueue`: A read/writeable attribute, since there are both a “get” and “set” method.
4. `Online`: A write-only attribute, only a “set”-method in the interface.

Methods that don’t start with “get” or “set” are interpreted as Operations. In our example, there are three operations:

1. `addPrinterJob(String texttoprint)`
2. `removePrinterJob(int jobid)`
3. `getJobList()`

Once registered, the mentioned attributes and operations are accessible via JMX. Please note that `getSheetsPrintedSoFar()` was not included in the interface and thus is not accessible from JMX.

---

## A.2.2 Dynamic MBeans

Dynamic MBeans are more complex. Much more work is needed to gain the advantage that the “JMX-Behaviour” of the MBean can be altered at runtime and the MBean can be registered from a different Java Virtual Machine (more information about accessing MBeans from different Virtual Machines is discussed in a later section).

To model the PrinterHandler into a DynamicMBean, our class has to implement the interface `javax.management.DynamicMBean` and thus six more methods are needed:

- `getAttribute` and `getAttributes` return the values of attributes to JMX.
- `setAttribute` and `setAttributes` allows JMX to set attributes in the object.
- `getMBeanInfo` tells JMX what functionality is available for management from this object.
- `invoke` allows JMX to invoke operations in the MBean.

The following part shows the essential changes that have to be made to transform PrinterHandler into a dynamic MBean. Since the changes are quite complex, just a few key examples are shown. The complete code is supplied in the “Dynamic MBean PrinterHandler” section.

```
public class PrinterHandler implements DynamicMBean {
    private int sheetsofpaperleft;           // How much paper is left
    private int sheetsprintedsofar;         // Total number of papersheets printed
    private int maximumnumberofjobsinqueue; // How many jobs we will queue at most
    private Vector jobs;                    // A Vector containing the PrinterJobs

    // This is the first new part. In order to tell JMX what kind of functionality
    // this object provides, we create an MBeanInfo object that can be requested
    // by JMX from this object.

    private MBeanInfo mbeaninfo

    // ----- First, we provide name and description for the MBean.
    = new MBeanInfo("PrinterHandler",
        "This object manages a printer.",

        // ----- Next, we store information about the attributes.
        new MBeanAttributeInfo[]
        { new MBeanAttributeInfo(
            "SheetsOfPaperLeft", "int",
            "The number of paper sheets left in printer",
            true, // ----- Yes, it's readable.
            false, // ----- No, not writeable.
            false), // ----- No, no "is"-method available.

        // ----- No constructors in our example.
        new MBeanConstructorInfo[0],

        // ----- The following part contains information about
        // the operation the object provides.
        new MBeanOperationInfo[]
        { new MBeanOperationInfo(
```

---

```
        "addPrinterJob",
        "adds a new job to the printer queue.",
        new MBeanParameterInfo[]
            { new MBeanParameterInfo
                ("texttoprint","String","") },
        "",
        MBeanOperationInfo.ACTION),

        // ----- No notifications in our example.
        new MBeanNotificationInfo[0]);

public int getSheetsOfPaperLeft() { ... }
public int getSheetsPrintedSoFar() { ... }
public int getMaximumNumberOfJobsInQueue() { ... }
public void setMaximumNumberOfJobsInQueue(int numberofjobs) { ... }
public void setOnline(boolean newmode) {...}
public void addPrinterJob(String texttoprint) { ... }
public int removePrinterJob(int jobid) { ... }
public String getJobList() { ... }

// ----- The first new method allows JMX to request the value of attributes.
public Object getAttribute(String attribute) throws AttributeNotFoundException
{
    Object result;
    if (attribute.equals("SheetsOfPaperLeft"))
        result = new Integer(sheetsofpaperleft);
    else if (attribute.equals("MaximumNumberOfJobsInQueue"))
        result = new Integer(maximumnumberofjobsinqueue);
    else if (attribute.equals("JobList"))
        result = getJobList();
    else
        throw new AttributeNotFoundException();
    return result;
}

// ----- This method gives JMX the possibility to set attributes.
public void setAttribute(Attribute attribute) {
    if (attribute.getName().equals("MaximumNumberOfJobsInQueue"))
        maximumnumberofjobsinqueue = ((Integer)attribute.getValue()).intValue();
    else if (attribute.getName().equals("Online"))
        setOnlineStatus(((Boolean)attribute.getValue()).booleanValue());
    return;
}

// ----- This method returns information about the MBean. The variable
//         mbeaninfo is defined at the top of the class.
public MBeanInfo getMBeanInfo() {
    return mbeaninfo;
}
```

---

```
// ----- The invoke method allows JMX to invoke operations in this MBean.
public Object invoke(String actionName, Object[] params, String[] signature)
    throws MBeanException, ReflectionException {

    Object result;
    if (actionName.equals("addPrinterJob"))
        addPrinterJob((String)params[0]);
    else if (actionName.equals("removePrintJob"))
        removePrinterJob(((Integer)params[0]).intValue());
    else if (actionName.equals("setOnline"))
        setOnline(((Boolean)params[0]).booleanValue());
    else {
        throw new MBeanException(new Exception(), "Unknown operation");
    }
    return result;
}
}
```

As the length of this – incomplete – example shows, a lot of work that has been done by JMX in the static example has to be done by the object itself when using the dynamic approach.

## A.3 flexiPKI-specific JMX-Enhancements

### A.3.1 RmiAdaptorClient and JmxAgent

The current knowledge is sufficient to set up a fully JMX-compliant working MBean. There are problems with distributed architectures, however. The standard – that means free – JMX package distributed by Sun doesn't allow MBeans on other machines in the network to be used, not even MBeans on the same computer that are running in a different Java Virtual machine (there is an extension that helps with this, but it is only available through the JDMK package which is not cheap). Since flexiPKI uses multiple virtual machines as well as multiple nodes in a network, an extension is needed: The JMX/RMI-Package.

To quickly summarize the functionality: On every machine that should be accessible from the outside there has to be an instance of `de.tud.cdc.flexiTrust.ad.JmxAgent` running. This object will set up a RMI connection point so that MBeans in other virtual machines on the same computer can be registered at one single point. In theory, it would be also possible to register MBeans from other machines, but this approach is discouraged.

One more step is necessary. So far we just modified existing objects so that the JMX system was able to access and manipulate them. That was sufficient in case that our object is instantiated by someone else and bound to the JMX management system – that's the usual way JMX is supposed to work. In case we want to start a program "by hand" and make itself available to the JMX management, we need to bind it with JMX/RMI. This is what the `RmiAdaptorClient` class does.

To further extend our existing `PrinterHandler` example object, we add the main function that will try to bind the object to the central server on the current machine. This will only work if an instance of `JmxAgent` is running. The bad news: Objects that should be bound via the RMI method have to be dynamic MBeans.

---

```
public static void main(String[] args) {
    PrinterHandler printerhandler = new PrinterHandler();
    Hashtable hashes = new Hashtable();
    RmiAdaptorClientImpl raci;

    hashes.put("name", "PrinterHandler");
    hashes.put("ViewInServlet", "true");
    try {
        raci = new RmiAdaptorClientImpl(printerhandler,
                                       new ObjectName("flexiPKI", hashes));
    }
    catch (Exception e) {
        e.printStackTrace();
    }
    return;
}
```

This method will create an instance of `PrinterHandler` and register it with the `JmxAgent`. Now our object can be accessed not only by standard JMX means, but also from components running in different virtual machines, from other computers and from the Administration servlet.

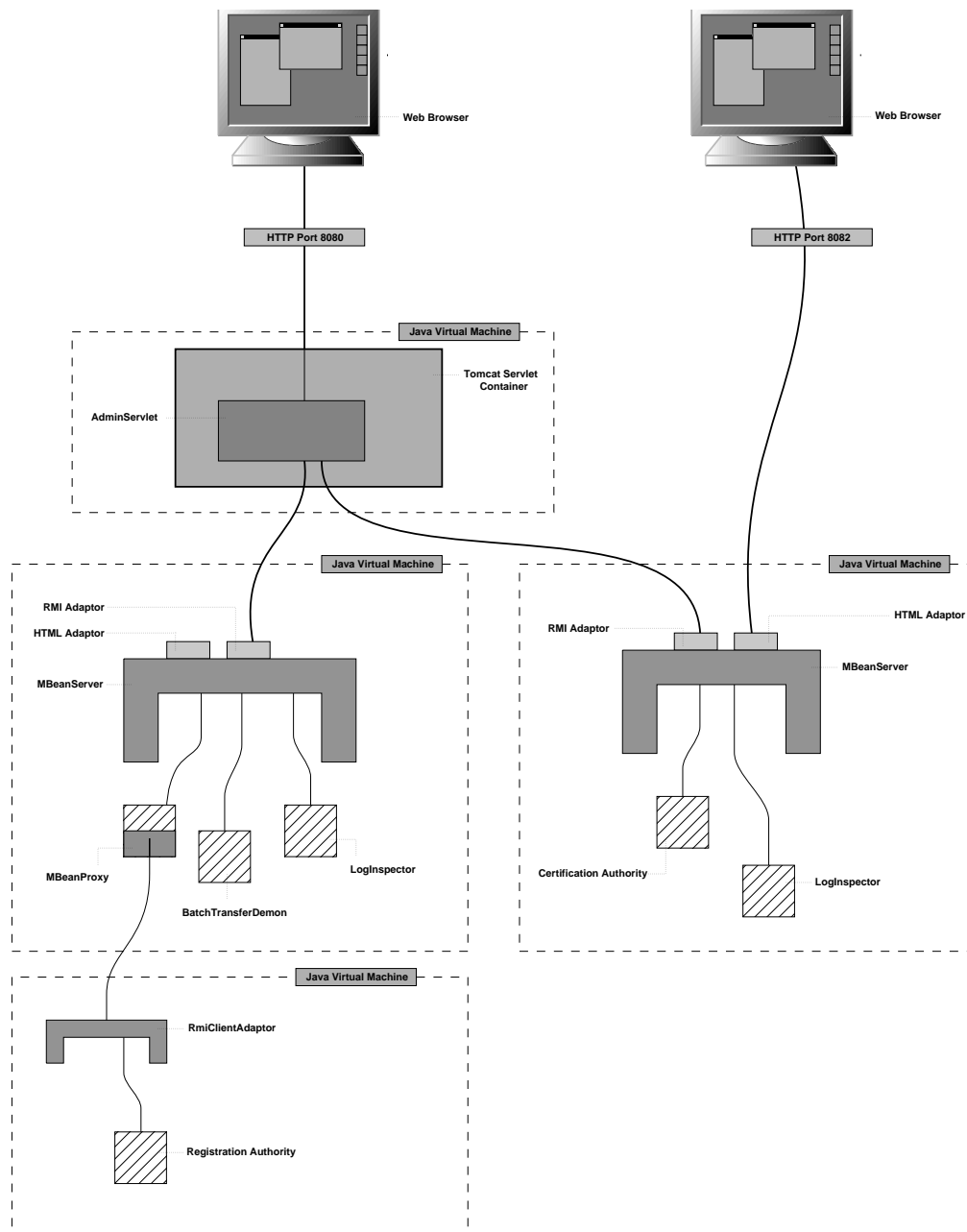
## A.4 Understanding the JMX Management

The previous chapter finished the implementation of our first MBean. To actually use it in an active environment, we have to register it somewhere and access it via some management application. Before we start with the practical part, we'll take a look at a working Administration setup to better understand how JMX is working.

The key JMX component is the MBean Server. This object provides a registry for MBeans and gives access to these via adaptors. The java virtual machine on the right side for example contains one `MBeanServer` with two registered MBeans. Access to these MBeans is provided either via the HTML Adaptor or the RMI Adaptor. To use the HTML Adaptor you have to point your browser at `http://127.0.0.1:8082`. If you use this approach your machine is represented at the top right of the page – you have direct access to the `MBeanServer` in one Java Virtual Machine, but to access the contents of others, you have to use a different URL. If you are using the Administration servlet instead, your machine would be represented at the top left of the scheme. You access the Administration Servlet running inside a Tomcat Servlet Container. The Admin Servlet internally uses the RMI Adaptors to communicate with the various `MBeanServers` on different machines.

The last thing there is to discuss is the usage of the `RmiAdaptorClient`. As mentioned before, the `MBeanServer` cannot access MBeans running inside another java virtual machine. To circumvent this limitation, the MBean – in our example on the lower left of the page the Registration authority – registers itself with a newly instantiated `RmiAdaptorClient`. The `RmiAdaptorClient` contacts the RMI Adaptor which is bound to the `MBeanServer` already running on the local computer, and gets registered via a proxy object that is forwarding all requests to the Registration Authority via RMI and vice versa.

---

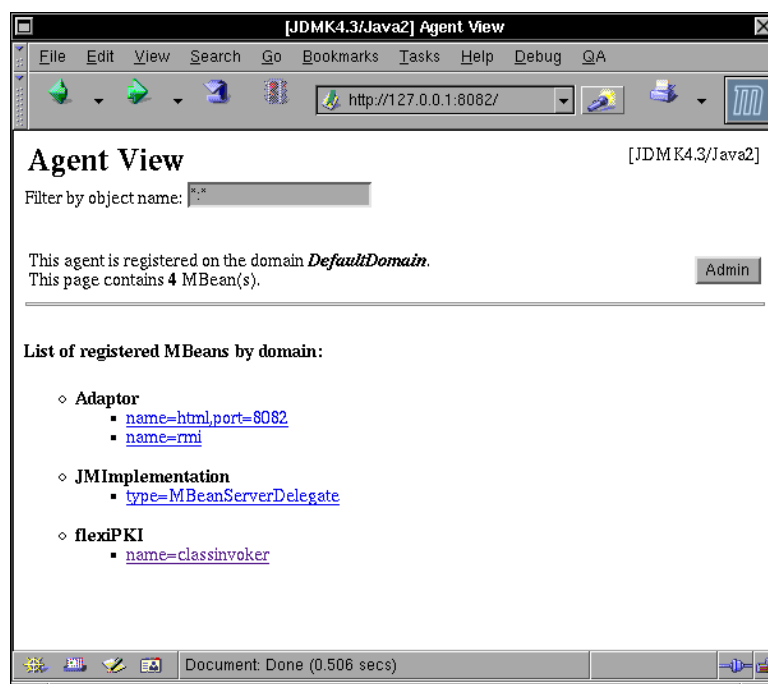


## A.5 Setting up the JmxAgent with the PrinterHandler

### A.5.1 Registering PrinterHandler in the same virtual machine

For the first try, we take the easy approach – we register our MBean at an existing MBeanServer and access it via the HTML Adaptor. This requires only a few steps.

1. First, since we're using RMI, the rmiregistry has to be running. Just type `rmiregistry`.
2. Now we need to start the JmxAgent. Simply invoke this class from the command line: `java de.tud.cdc.flexiTrust.ad.JmxAgent`. The JmxAgent starts up a MBeanServer as well as a HTML- and RMI-Adaptor. It also invokes a ClassInvoker-Object and binds it to the MBeanServer. The ClassInvoker is not needed now and will be discussed later.
3. Start a webbrowser and point it to `http://127.0.0.1:8082`. You should see the following output:



4. Now we register the PrinterHandler object. Click on the “Admin” button on the top right of the page. In the form that appears on the screen enter “name=PrinterHandler” in the “Keys” input field. Type “de.tud.cdc.flexiTrust.ad.demo.PrinterHandler” in the “Java Class” field. Then click “Send Request”.
5. If you get a “Create Successful” response, click on “Back to Agent View” on the top right of the screen. A new MBean is shown under “Default Domain”: Our PrinterHandler. Click once on the corresponding link to enter the PrinterHandler’s MBean view. You should see the following output:

**MBean View** [JDK4.3/Java2]

- MBean Name: DefaultDomain:name=PrinterHandler
- MBean Java Class: PrinterHandler

[Back to Agent View](#)      Reload Period in seconds:       

---

**MBean description:**  
This object manages a printer.

---

**List of MBean attributes:**

Name	Type	Access	Value
<a href="#">JobList</a>	String	RO	(empty)
<a href="#">MaximumNumberOfJobsInQueue</a>	int	RW	<input type="text" value="20"/>
<a href="#">SheetsOfPaperLeft</a>	int	RO	100

---

**List of MBean operations:**

**Description of setOnline**  
 (boolean)newmode  True  False

**Description of removePrinterJob**  
 (int)jobid

**Description of addPrinterJob**  
 (String)texttoprint

Document: Done (0.651 secs)

- Due to the very simple implementation, it is only possible to change the maximum number of jobs in the queue. Simply enter a new value and click the “Apply” button.

This concludes the simple approach. The meaning of simple is that we let the HTML Adaptor create and register a new instance of the PrinterHandler. The new PrinterHandler object is running in the same virtual machine as the MBeanServer and all other components.

### A.5.2 Registering PrinterHandler in a different virtual machine

In the next example, we create a new PrinterHandler running in another virtual machine and use the RmiAdaptorClient to register it.

1. Stop the JmxAgent and restart it. When connecting to `http://127.0.0.1:8082` again, the PrinterHandler link is gone.
2. Execute `java de.tud.cdc.flexiTrust.ad.demo.PrintHandler` to run the PrinterHandler class directly. With the method used in the previous section, the MBeanServer directly instantiated a PrinterHandler with the help of its constructor. Now the `main()`-method is called, which is creating an instance of PrinterHandler and registering it using a RmiAdaptorClient.
3. Reloading the page at `http://127.0.0.1:8082` you can see that the PrinterHandler MBean link is there again. Note that this time it is listed under “flexiPKI”, because the `main()`-method of the PrinterHandler class registers it under this domain.

### A.5.3 Setting up the Administration Servlet

After using the HTML Adaptor to manipulate the MBeans, we’ll now use the Administration Servlet.

1. Install Tomcat.
  2. Change into the Tomcat build directory.
  3. Change into the `webapps` directory.
  4. Copy the `AdminServlet` directory from `ad/etc` into the current directory, including all of its contents.
  5. Change into the `AdminServlet/WEB-INF` directory.
  6. Copy the directory `classes` with all its contents from `ad` to the current directory. If you are using Unix, create a link.
  7. In the file `AdminProperties.java` in `src/de/tud/cdc/flexiTrust/ad`, modify the following lines:
    - The line `NODELISTFILENAME` points to a file where the list of nodes that are registered with the AdminServlet are written to. Example: `/home/myname/java/ad/NodeList`. An empty `NodeList`-file comes along with the `ad`-package, but is not needed. If the file doesn’t exist, it will be created automatically.
    - The line `CLASSINVOKERJOBLIST` points to a property list of jobs that the AdminServlet should know about and thus can start automatically, e.g. the `BatchTransferDemon`. One `ClassInvokerJobs`-file containing examples for the `BatchTransferDemon`, the `LogInspector` and the `FileDumper` comes along with the `ad`-package in the `etc` directory, point `CLASSINVOKERJOBLIST` to the correct location.
    - `ANT_EXECCOMMAND` is needed to start jobs via ant. Set the contents of the String to the ant executable.
    - It’s possible to set `JMXAGENT_STATICLOGFILENAME` to some specific location. That ensures that the JmxAgent’s logfile will always be written to the same location, else it will be written in the current directory.
-

8. To compile the package, you need the libraries already contained in the `all/lib` directory. The only additional jar that is needed is the “Protomatter” class collection, which is freely available at <http://protomatter.sourceforge.net>. These tools provide functionality to parse MIME-encoded POST-Requests from browsers, that means File-Uploads.
9. Compile the changes with `ant publish-jar`. Four deprecation warnings will appear during compile time due to servlet deprecations.
10. Before starting any application, run `rmiregistry`.
11. Start Tomcat. You should see a line containing `Adding context Ctx( /AdminServlet )` in the output.
12. Run the `JmxAgent` with `java de.tud.cdc.flexiTrust.ad.JmxAgent`.
13. Point your browser at `http://127.0.0.1:8080/AdminServlet/servlet/AdminServlet/`.

## A.6 Dynamic MBean PrinterHandler

This section contains the complete `PrinterHandle` dynamic MBean class. The `getAttributes`- and `setAttributes`-method as well as some parts of the `MBeanInfo` were omitted in the text for better readability.

```
// This object doesn't do anything useful. It's just for demonstration
// purposes.

package de.tud.cdc.flexiTrust.ad.demo;

import de.tud.cdc.flexiTrust.ad.jmxrmi.RmiAdaptorClientImpl;
import java.util.Vector;
import java.util.Hashtable;
import javax.management.Attribute;
import javax.management.AttributeList;
import javax.management.DynamicMBean;
import javax.management.AttributeNotFoundException;
import javax.management.InvalidAttributeValueException;
import javax.management.MBeanAttributeInfo;
import javax.management.MBeanConstructorInfo;
import javax.management.MBeanException;
import javax.management.MBeanInfo;
import javax.management.MBeanNotificationInfo;
import javax.management.MBeanParameterInfo;
import javax.management.MBeanOperationInfo;
import javax.management.ObjectName;
import javax.management.ReflectionException;

public class PrinterHandler implements DynamicMBean {
    private int sheetsofpaperleft = 100;    // How much paper is left
    private int sheetsprintedsofar = 0;    // Total number of papersheets printed
```

---

---

```
private int maximumnumberofjobsinqueue
                                = 10; // How many jobs we will queue at most
private Vector jobs;              // A Vector containing the PrinterJobs

// This is the first new part. In order to tell JMX what kind of functionality
// this object provides, we create an MBeanInfo object that can be requested
// by JMX from this object.

private MBeanInfo mbeaninfo

// First, we provide name and description for the MBean.
= new MBeanInfo("PrinterHandler",
                "This object manages a printer.",

                // Next, we store information about the attributes.
                new MBeanAttributeInfo[]
                { new MBeanAttributeInfo(
                  "SheetsOfPaperLeft", "int",
                  "The number of paper sheets left in printer",
                  true, false, false),

                  new MBeanAttributeInfo(
                  "MaximumNumberOfJobsInQueue", "int",
                  "The maximum number of jobs the object will queue",
                  true, true, false),

                  new MBeanAttributeInfo(
                  "JobList", "String",
                  "Show the list of jobs in the printer queue",
                  true, false, false)},

                // No constructors in our example.
                new MBeanConstructorInfo[0],

                // The following part contains information about
                // the operation the object provides.
                new MBeanOperationInfo[]
                { new MBeanOperationInfo(
                  "addPrinterJob",
                  "adds a new job to the printer queue.",
                  new MBeanParameterInfo[]
                  { new MBeanParameterInfo
                    ("textttoprint", "String", "") },
                  "",
                  MBeanOperationInfo.ACTION),
                  new MBeanOperationInfo(
                  "removePrinterJob",
                  "removes a printer job from the queue.",
```

---

---

```
        new MBeanParameterInfo[]
            { new MBeanParameterInfo("jobid","int","") },
        "",
        MBeanOperationInfo.ACTION),
        new MBeanOperationInfo(
            "setOnline",
            "sets the printer online or offline.",
            new MBeanParameterInfo[]
                { new MBeanParameterInfo
                    ("newmode","boolean","") },
            "",
            MBeanOperationInfo.ACTION) },

        // No Notifications in our example.
        new MBeanNotificationInfo[0]);

public int getSheetsOfPaperLeft() { return sheetsofpaperleft; }
public int getSheetsPrintedSoFar() { return 0; }
public int getMaximumNumberOfJobsInQueue()
    { return maximumnumberofjobsinqueue; }
public void setMaximumNumberOfJobsInQueue(int numberofjobs)
    { maximumnumberofjobsinqueue = numberofjobs; }
public void setOnline(boolean newmode) { return;}
public void addPrinterJob(String texttprint) { return; }
public void removePrinterJob(int jobid) { return; }
public String getJobList() { return "(empty)"; }

// The first new method. JMX can request the value of attributes via
// this function.
public Object getAttribute(String attribute) throws AttributeNotFoundException
{
    Object result;
    if (attribute.equals("SheetsOfPaperLeft"))
        result = new Integer(sheetsofpaperleft);
    else if (attribute.equals("MaximumNumberOfJobsInQueue"))
        result = new Integer(maximumnumberofjobsinqueue);
    else if (attribute.equals("JobList"))
        result = getJobList();
    else
        throw new AttributeNotFoundException();
    return result;
}

// This method returns multiple attribute values to JMX.
public AttributeList getAttributes(String[] attributes) {
    AttributeList result = new AttributeList();
    Object attribute;
```

---

---

```
for (int i=0; i<attributes.length; i++) {
    try {
        attribute = getAttribute(attributes[i]);
    }
    catch (AttributeNotFoundException e) {
        attribute = null;
    }
    result.add(new Attribute(attributes[i], attribute));
}

return result;
}

// This method gives JMX the possibility to set attributes.
public void setAttribute(Attribute attribute) {
    if (attribute.getName().equals("MaximumNumberOfJobsInQueue"))
        maximumnumberofjobsinqueue = ((Integer)attribute.getValue()).intValue();
    else if (attribute.getName().equals("Online"))
        setOnline(((Boolean)attribute.getValue()).booleanValue());
    return;
}

// setAttributes() allows JMX to set multiple values with a single
// function call. A list of the new values is returned.
public AttributeList setAttributes(AttributeList attributes) {
    String[] attributenames = new String[attributes.size()];

    for (int i=0; i<attributes.size(); i++) {
        setAttribute((Attribute)attributes.get(i));
        attributenames[i] = ((Attribute)attributes.get(i)).getName();
    }

    return getAttributes(attributenames);
}

// This method returns information about the MBean. The variable
// mbeaninfo is defined at the top of the class.
public MBeanInfo getMBeanInfo() {
    return mbeaninfo;
}

// The invoke method allows JMX to invoke operations in this MBean.
public Object invoke(String actionName, Object[] params, String[] signature)
    throws MBeanException, ReflectionException {
```

---

```
Object result = null;
if (actionName.equals("addPrinterJob"))
    addPrinterJob((String)params[0]);
else if (actionName.equals("removePrintJob"))
    removePrinterJob(((Integer)params[0]).intValue());
else if (actionName.equals("setOnline"))
    setOnline(((Boolean)params[0]).booleanValue());
else {
    throw new MBeanException(new Exception(), "Unknown operation");
}
return result;
}

public static void main(String[] args) {
    PrinterHandler printerhandler = new PrinterHandler();
    Hashtable hashes = new Hashtable();
    RmiAdaptorClientImpl raci;

    hashes.put("name", "PrinterHandler");
    hashes.put("ViewInServlet", "true");
    try {
        raci = new RmiAdaptorClientImpl(printerhandler,
                                       new ObjectName("flexiPKI", hashes));
    }
    catch (Exception e) {
        e.printStackTrace();
    }
    return;
}
}
```

---

## A.7 Installing the Administration Package

The following steps are required to install the Administration Package. The first part describes the steps to build the Admin classes, which are used to make a machine available for management by the servlet. The second part describes the necessary work to get the servlet running. The servlet is only required on a single machine of the network.

### A.7.1 Build the JmxAgent

1. Install a Java Development Kit. The Standard Edition is sufficient. It can be any version from 1.3.1 to 1.4.0. The 1.2.2 version should work too, but has not been tested lately.
2. Install an Ant distribution. The actual version has been build without problems using Ant 1.4.1.
3. Get the JMX-API. Only two of the three supplied library files are needed: The jars named `jmxri.jar` and `jmxtools.jar`. Both should be added to the `CLASSPATH` environment variable. `jmxgrinder.jar` must **not** be included.
4. The Servlet-API jar has to be available und included in the `CLASSPATH` environment variable.
5. The protomatter-package has to be accessible and its location has to be included in the `CLASSPATH` environment variable.
6. The `build.xml` file includes files from `../all/lib` by default. So, to get the build process running without errors, this directory has to be created. It does not have to contain any files.
7. Once the requirements are met, the package can be built with a call to "ant jar". This will create the file "admin.jar", which also has to be added to the `CLASSPATH`.

Now the JmxAgent can be started with

```
java -Djava.security.manager -Djava.security.policy=jmx-agent.policy
  de.tud.cdc.flexiTrust.ad.JmxAgent
```

Please note that the `rmiregistry` has to be running when starting the JmxAgent.

### A.7.2 Setting up Tomcat for the Servlet

1. Install Tomcat. The current version has been tested with Tomcat 3.2.x and 3.3.x versions.
  2. In the tomcat directory tree, create a new folder named `AdminServlet` under the `webapps` directory.
  3. Copy the files `Computer.Gray.gif` and `Computer.gif` together with `style.css` into the new directory.
  4. Within the new directory, create a folder named `WEB-INF`.
  5. Copy the file `web.xml` into the `WEB-INF` directory.
  6. In the `WEB-INF` directory, create a folder named `lib`.
  7. Copy the `jmxri.jar`, `jmxtools.jar`, `protomatter-1.1.6.jar` and the Servlet 2.3 API classfiles into the `lib` directory.
  8. Create a new folder named `classes` and copy the classfiles of the ad-Package into it.
  9. Start Tomcat, for example with `tomcat.bat` or `tomcat.sh`.
-

## Anhang B

# Tomcat Installation

### B.1 Requirements

The following files are required:

1. `jakarta-ant-1.4.1-src.tar.gz`  
from (<http://jakarta.apache.org/builds/jakarta-ant/release/v1.4.1/src/>)
2. `jakarta-servletapi-3.2.4-src.tar.gz`  
from (<http://jakarta.apache.org/builds/jakarta-tomcat/release/v3.2.4/src/>)
3. `jakarta-tomcat-3.2.4-src.tar.gz`  
from (<http://jakarta.apache.org/builds/jakarta-tomcat/release/v3.2.4/src/>)
4. `java_xml_pack-fall01.zip`  
from (<http://java.sun.com/xml/downloads/javaxmlpack.html>)
5. `jsse-1.0.2-g1.zip`  
from (<http://java.sun.com/products/jsse/>)
6. `Christoph Ender_SIGN.12.p12`
7. `caCert.pem.crt`
8. `localhost.11.pem`
9. `localhost.11.pem.crt`
10. `web.xml`
11. `TestServlet.java`

### B.2 Tomcat Installation

First, we install the Tomcat Servlet Container with SSL functionality. To get there, we need XML, the Java Secure Socket Extension, Ant and the ServletAPI.

- You need a JDK installed.
  - Unzip `java_xml_pack-fall01.zip`.
  - Copy `xalan.jar` and `crimson.jar` from `java_xml_pack-fall01/jaxp-1.1.3/` into a separate directory outside of `java_xml_pack-fall01/` and add them to your CLASSPATH.
  - Delete the `java_xml_pack-fall01` directory.
  - Unzip `jsse-1.0.2-g1.zip`.
  - Copy `jcrt.jar`, `jnet.jar` and `jsse.jar` from `jsse1.0.2/lib` into a separate directory outside of `jsse1.0.2/` and add the jars to your CLASSPATH.
-

- Delete the `jsse1.0.2` directory.
- Unpack `jakarta-ant-1.4.1-src.tar.gz` (best in `/usr/local/jakarta`).
- Change in `jakarta-ant-1.4.1/` and execute `./build.sh`.
- Move the `dist` directory out of `jakarta-ant-1.4.1/`, and add the `dist/bin` directory to your `PATH`.
- Execute `ant -verbose`. You should get something like this:

```
Ant version 1.4.1 compiled on November 27 2001
Buildfile: build.xml does not exist!
Build failed
```

- Delete the `jakarta-ant-1.4.1` directory.
- Unpack `jakarta-servletapi-3.2.4-src.tar.gz`.
- Change into the `jakarta-servletapi-3.2.4` directory.
- Execute `./build.sh`.
- Copy the file `lib/servlet.jar` to a directory outside of the directory with the name `jakarta-servletapi-3.2.4/` and add this `servlet.jar` to your `CLASSPATH`.
- Do **NOT** delete the `jakarta-servletapi-3.2.4/` directory
- Unpack `jakarta-tomcat-3.2.4-src.tar.gz` (best in `/usr/local/jakarta`).
- Change to `jakarta-tomcat-3.2.4-src.tar.gz/` and edit the `build.xml` file:

```
In "Initialization properties", change
  <property name="jaxp" value="../jaxp-1.1/jaxp.jar" />
to
  <property name="xalan" value="(yourdirectory)/xalan.jar" />
```

```
In "Initialization properties", change
  <property name="parser" value="../jaxp-1.1/crimson.jar" />
to
  <property name="parser" value="(yourdirectory)/crimson.jar" />
```

```
In "Initialization properties", change "../jakarta-servletapi" of the line
  <property name="servlet.jar"
            value="../jakarta-servletapi/lib/servlet.jar"/>
to the path and directory name of the unpacked servlet directory.
```

```
In "<!-- Copy library JAR files -->", change
  file="{jaxp}"/>
to
  file="{xalan}"/>
```

```
In "Build tomcat", change
  classpath="{servlet.jar};{jaxp};{parser}"
to
  classpath="{servlet.jar};{xalan};{parser}"
```

---

In the above lines, “(yourdirectory)” refers to the directory where you installed xalan.jar and/or crimson.jar. I’m not sure whether all replacements of jaxp are necessary, it worked for me however.

- Execute `./build.sh`
- Change to `../build/tomcat/bin` and execute `./tomcat.sh run`.
- Point your browser at `http://localhost:8080`

### B.3 Tomcat Servlet Setup

Next, we install the TestServlet and configure Tomcat to work with Apache via `mod_jk`.

- In `jakarta/build/tomcat/webapps`, create a directory named `TestServlet`. Inside the new directory, create a directory named `WEB-INF`. Copy the file `web.xml` and the directory `classes` with all contents into `WEB-INF`.
  - Change into the directory `de/tud/cdc/flexiTrust/ad` and compile the `TestServlet` via `javac TestServlet.java`.
  - Look at the file `httpd.conf` in the configuration directory of Apache. You often find a line like this at the end of the file:

```
#Include /etc/apache/mod_ssl.conf
```

Remove the comment mark to activate the statement.
  - Add the following three lines at the end of `httpd.conf` to force client authorization:

```
SSLVerifyClient require
SSLVerifyDepth 1
SSLCACertificateFile /etc/apache/ssl.crt/ca.crt
```

You can also edit `mod_ssl.conf` which contains many examples.
  - Run Tomcat again (next step tells you why).
  - Add the following line to `httpd.conf`:

```
Include /usr/local/jakarta/build/tomcat/conf/mod_jk.conf-myfile
```

Adjust the Directory according to your setup. The `mod_jk.conf-auto` file is created as soon Tomcat is run and contains configuration information to run the Apache-Tomcat-Connector, `mod_jk`. Copy the file with the name “`mod_jk.conf-auto`” to “`mod_jk.conf-myfile`” in the same directory so we have a file we can edit without Tomcat constantly overwriting it at startup.
  - Modify `mod_jk.conf-myfile`:
    - Uncomment `JkExtractSSL` and set from `Off` to `On`
    - Uncomment `JkHTTPSIndicator HTTPS`
    - Uncomment `JkSESSIONIndicator SSL_SESSION_ID`
    - Uncomment `JkCIPHERIndicator SSL_CIPHER`
    - Uncomment `JkCERTSIndicator SSL_CLIENT_CERT`
    - Replace “`JkMount /TestServlet/servlet/* ajp12`” with “`JkMount /TestServlet/servlet/* ajp13`”
    - Replaced “`JkMount /TestServlet/*.jsp ajp13`” with “`JkMount /TestServlet/*.jsp ajp13`”
-

- Edit `conf/server.xml`. Above “Special webapps”, under `</Connector>` of the “Apache AJP12 support”, add:

```
<Connector className="org.apache.tomcat.service.PoolTcpConnector">
  <Parameter name="handler"
    value=
      "org.apache.tomcat.service.connector.Ajp13ConnectionHandler"/>
  <Parameter name="port" value="8009"/>
</Connector>
```
- To use `mod_jk`, we have to build it first. Change into `jakarta-tomcat-3.2.4-src`, the Tomcat Source directory, and go to `/src/native/apache1.3`. Choose a Makefile compatible to your system, e.g.:

```
ln -s Makefile.linux Makefile
```

Then execute “make”.
- After the building is complete, copy the “`mod_jk.so`” module (or similar) into the `libexec`-directory for Apache, for example `/usr/libexec`.

## B.4 Setup Apache server and personal certificates

- You need an installed Apache Server and ModSSL.
- Start Netscape.
- Open Communicator, Tools, Security Info.
- Select Certificates, Yours.
- Click “Import Certificates”.
- For testing purposes, just click on “Ok” without entering a password.
- Select “Christoph Ender\_SIGN.12.p12” and use “12345678” to import the certificate.
- Copy file `caCert.pem.crt` to `ca.crt` in `/etc/apache/ssl.crt/`.
- After that, copy the file `localhost.11.pem.crt` to `server.crt` into the directory named `/etc/apache/ssl.crt/`.
- Next, execute `make -f Makefile.crt` in `/etc/apache/ssl.crt`.
- Copy `localhost.11.pem` to `/etc/apache/ssl.key/server.key`.

## B.5 Start all Components and test the Installation

- Start Tomcat.
  - Start Apache with `httpd -DSSL start`. Enter passphrase “12345678” to enable Apache to read the needed files.
  - Point your Browser to `https://localhost/TestServlet/servlet/TestServlet/`.
-

## Anhang C

# Extracting MIME-Encoded Parameters

The Java Servlet Environment is not able to parse MIME-encoded parameters in the version distributed by Sun. This sheet describes how to use the “Protomatter” package in combination with one more class and a few lines of code to overcome this limitation.

Promomatter is available from <http://sourceforge.net/projects/protomatter>. The only classes used from the package are `com.protomatter.util.MIMEMessage`, which is able to parse a MIME-Message and `com.protomatter.util.MIMEAttachment`, which represents an Attachment. The package has to be compiled and to be available via the classpath environment variable.

The MIME-handling section should only be executed if we actually have a MIME-encoded request. To verify this, we have to examine the `Content-Type` String.

```
if (request.getContentType().substring(0,19).equals("multipart/form-data")) {
```

If there's a MIME-encoded request incoming, the next step is to read the raw request from the request object.

```
reader = request.getReader();
buffer = new StringBuffer();

nextchar = reader.read();
while (nextchar != -1) {
    buffer.append((char)nextchar);
    nextchar = reader.read();
}
```

This raw stringbuffer is now parsed by the protomatter `MIMEMessage` class. To store the parameters we will extract from the `MIMEMessage` object, we also create an instance of the class `AdHttpRequest`.

```
mm = MIMEMessage.parse(buffer.toString().getBytes());
ahsr = new AdHttpRequest(request);
```

---

---

Now, the only thing left to do is to iterate through all available attachments and add these as parameters to the `AdHttpRequest` object. While doing this, it's necessary to take care if any parameter names are quoted. If this turns out to be the case, the quotes have to be removed when creating the parameter name for our storage data.

```

ats = mm.getAttachments();
while (ats.hasMoreElements()) {
    a = (MIMEAttachment)ats.nextElement();
    s = a.getHeader("Content-Disposition");
    if (s.charAt(s.indexOf("=")+1) == '\"')
        name = s.substring(s.indexOf("=")+2, s.indexOf("\"", s.indexOf("=")+2));
    else
        name = s.substring(s.indexOf("=")+1, s.length());
    value = new String(a.getContent());

    ahsr.addNewParameter(name, value);
}

```

The class named `AdHttpRequest` wraps the contents of the servlet-framework-class `javax.servlet.http.HttpServletRequest`. It makes it very easy to adapt existing servlets to the MIME-encoding parameter parsing. As an example, we look at a servlet accepting both `GET` and `POST` requests – `GET` for everything usual, `POST` for MIME-encoded uploads. We implement `doGet` the usual way:

```

public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {
    ...

```

`doPost` however, contains the code to check for MIME-encoded data, that means all the lines of code described in this sheet.

```

public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {
    ( ... variable declarations ... )

    if(request.getContentType().substring(0,19).equals("multipart/form-data")) {
        reader = request.getReader();
        buffer = new StringBuffer();

        ( ... extract MIME-parameters ... )

        ahsr.addNewParameter(name, value);
    }
    doGet(ahsr, response);
}
else
    doGet(request, response);
return;
}

```

---

Since the `AdHttpRequest` implements `HttpServletRequest`, it can be passed to `doGet` like the request object that's received from the servlet framework.

---

# Literaturverzeichnis

- [1] Fachgebiet Kryptographie, Computeralgebra;  
„FlexiPKI – Flexible Public Key Infrastruktur“;  
<http://www.informatik.tu-darmstadt.de/TI/Forschung/FlexiPKI>
  
  - [2] Sun Microsystems;  
„Java(TM) Servlet Technology“;  
<http://java.sun.com/products/servlet>
  
  - [3] Marc A. Saegesser;  
„Jakarta Tomcat“;  
<http://jakarta.apache.org/tomcat>
  
  - [4] Jon S. Stevens;  
„The Jakarta Project“;  
<http://jakarta.apache.org/>
  
  - [5] Sun Microsystems;  
„Java Management(TM) Extensions“;  
<http://java.sun.com/products/JavaManagement>
  
  - [6] Sun Microsystems;  
„Java(TM) Remote Method Invocation(RMI)“;  
<http://java.sun.com/products/jdk/rmi>
  
  - [7] Sun Microsystems; „Creating a Custom RMI Socket Factory“;  
[http://java.sun.com/products/jdk/1.2/docs/guide/  
rmi/rmisocketfactory.doc.html](http://java.sun.com/products/jdk/1.2/docs/guide/rmi/rmisocketfactory.doc.html)
-