INSTITUT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Diplomarbeit

Management eines Internet Telefonie Servers mittels JDMK

Bearbeiter:	Harald Knöchlein
Aufgabensteller:	Prof. Dr. Heinz-Gerd Hegering
Betreuer:	Helmut Reiser Dr. Alexander Keller Thomas Lederer (Siemens AG)

INSTITUT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Diplomarbeit

Management eines Internet Telefonie Servers mittels JDMK

Bearbeiter:	Harald Knöchlein
Aufgabensteller:	Prof. Dr. Heinz-Gerd Hegering
Betreuer:	Helmut Reiser Dr. Alexander Keller Thomas Lederer (Siemens AG)

Abgabetermin: 15. Februar 1999

Hiermit versichere ich, daß ich die vorliegende Diplomarbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15. Februar 1999

....

(Unterschrift des Kandidaten)

Zusammenfassung

An das Management von Netzwerken werden immer höhere Anforderungen gestellt. Ständig wachsende Netze und eine Vielzahl von Komponenten machen eine kontinuierliche Anpassung notwendig. Der daraus resultierende Managementaufwand und die damit verbundenen Kosten sind Gesichtspunkte, welche im Netz- und Systemmanagement eine zentrale Rolle spielen.

Eine der Managementumgebungen, die sehr stark einem evolutionären Wechsel unterliegen, ist die Telekommunikation. Seit Anfang der neunziger Jahre ist der Konkurrenzkampf zwischen den reinen Datenübertragungssystemen und den Sprachsystemen stetig angestiegen. Aus diesem Grund sind die verschiedenen Anbieter gezwungen, mehr und mehr Funktionalität zu ihren Systemen hinzuzufügen. Ein Versuch, eine integrierte Kommunikationsumgebung zu schaffen, ist die Hicom 300 Telefonanlage der Firma Siemens. Unter dem Schlagwort "Integration von Sprache und Daten" wird ein Kommunikationsumfeld geschaffen, welches ein möglichst breites Spektrum an Kundenanforderungen erfüllt. Durch die Vielzahl der zum Einsatz kommenden Geräte ist ein umfangreiches und aufwendiges Management erforderlich.

Das Ziel, den Managementaufwand von Systemen reduzieren zu können hat eine Reihe von Theorien zur Folge. Diese Theorien zielen darauf ab, den Agenten, welche innerhalb eines Netzmanagement-Systems eingesetzt werden, mehr Eigenständigkeit zu geben. Die Theorie der Flexiblen Managementagenten, welche den Agenten Eigenständigkeit, eine Anpaßbarkeit zur Laufzeit und weitere Vorteile bringt ist ein Versuch, den neuen Anforderungen an das Netzmanagement zu begegnen.

Das Java Dynamic Management Kit (JDMK) ist ein von der Firma Sun Microsystems Inc. entworfener Ansatz, weg von einem zentralisierten, statisch limitierten Netzmanagement hin zu einem von Diensten bestimmten Netzwerk "Service Driven Network". JDMK ist auf Java basierend und nutzt die JavaBeans-Technologie um Managementfunktionalität einem Agenten zu dessen Laufzeit hinzuzufügen. Dadurch wird eine schnelle Entwicklung von unabhängigen, intelligenten Java-Agenten ermöglicht, welche System-, Netzwerk- und Servicemanagementaufgaben erfüllen können.

Diese Diplomarbeit untersucht, in wie weit JDMK in der Lage ist, innerhalb der von Siemens geschaffenen Kommunikationsumgebung ein Managementsystem zu integrieren. Es sollen die Möglichkeiten des JDMK und dessen Nähe zu der Theorie der Flexiblen Management Agenten untersucht werden. Schließlich soll anhand einer prototypischen Implementierung ein Beispiel für ein solches Management geschaffen werden.

Das Ergebnis der Untersuchungen hat gezeigt, daß JDMK trotz noch derzeitiger Schwächen in einzelnen Bereichen durchaus für den Aufbau einer modernen Managementumgebung geeignet ist. Dies resultiert im Besonderen daraus, daß JDMK die Nähe zur Theorie der Flexiblen Management Agenten beweist und eine Vielzahl der in dieser Theorie geforderten Punkte erfüllt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einführung

In diesem Kapitel wird das Thema der Arbeit in die aktuelle Fragestellung nach der Leistungsfähigkeit und Realisierbarkeit der Flexiblen Management Agenten in Bezug auf das zu untersuchende Java Dynamic Management Kit eingeordnet. Anschließend wird die Aufgabenstellung dargestellt und ein Überblick über die Vorgehensweise geschaffen.

1.1 Motivation

Die derzeitige Situation des Netzmanagements beinhaltet wesentliche Schwächen. Die heutigen Managementsysteme sind bestimmt durch wenige Managementapplikationen, welche durch die Verwendung von (mehr oder weniger unselbständigen) Agenten in der Durchführung von Managementaktionen unterstützt werden. Diese Agenten werden für die Filterung von Daten oder deren Interpretation benutzt. Sie leiten Managementaktionen zu den Netzelementen, welche sie repräsentieren weiter oder sammeln Informationen über diese. Diese Agenten sind meist auf den Komponenten angesiedelt, welchen sie zugeordnet sind.
Dieses Management-Paradigma besitzt wesentliche Nachteile:

1. <u>zentralisierte Management-Intelligenz</u>
   Die gesamte Intelligenz, welche das Management des Netzes ermöglicht, ist in den Management-Applikationen zentral integriert. Die verwendeten Agenten werden für keine Managemententscheidungen verwendet, sie besitzen lediglich unterstützende Funktion. Der wesentlichste Nachteil hierbei ist die hohe Netzbelastung zwischen Manager und Agent, welche die eigentliche Nutzlast reduziert. Zudem können diese Agenten nicht auf Events, welche auf ihren Management-Knoten auftreten reagieren, sie leiten diese Events nur zu den Managern weiter.

2. <u>statischer Ansatz</u>
   Ein wesentlicher Nachteil vieler Managementsysteme ist deren statischer Ansatz. Dieser Ansatz erfordert, daß die Funktionalität und das Design des Agenten bereits zur Entwicklungszeit bekannt und festgelegt ist. Ein spätere Änderung ist dabei nur unter teilweise erheblichem Aufwand möglich. Oft ist hierbei ein komplettes Neuaufsetzen des Agenten erforderlich. Die Änderung der Managerapplikationen ist aufgrund der niedrigen Anzahl in vertretbarem Maße durchführbar. Bei den Agenten jedoch, die oft in großer Zahl über ein Netzwerk verteilt sind und einzeln geändert werden müssen, kann der Aufwand exorbitanten Umfang annehmen.

3. Hoher Managementaufwand
   Der Managementaufwand vieler heutiger Systeme ist sehr zeitintensiv und damit teuer. Oftmals ist es notwendig, lokal an dem jeweiligen Managementknoten die Administration durchzuführen, was bei räumlich verteilten Systemen zu einem Problem werden kann. Zudem resultiert der hohe Managementaufwand aus dem bisherigen statischen Ansatz der verwendeten Agenten.

Das Ziel ist von den bisherigen statischen Agenten, welche nur rudimentäre Funktionalität besitzen (SNMP-Agenten), zu flexiblen Agenten überzuwechseln, welche selbständiger agieren, mehr Management-Intelligenz besitzen und somit die Netzbelastung durch administrative Aufgaben nicht erhöhen. Diese Management-Intelligenz sollen diese Agenten nur dann erhalten, falls sie diese wirklich benötigen. Diese Anpaßbarkeit (Erweiterbarkeit) wird statisch und in letzter Zeit auch dynamisch gefordert. Die dynamische Anpaßbarkeit und weitere Forderungen, welche das Management von Netzen erleichtern und automatisieren soll, werden unter dem Begriff der Flexiblen Management Agenten subsumiert.
Die Theorie der Flexiblen Management Agenten ermöglicht ein anpaßbares Management von Netzen. Die Fähigkeit, einen Agenten in seiner Grundstruktur möglichst klein zu halten und ihm nur bei Bedarf die notwendige Funktionalität zur Durchführung einer Managementaufgabe zu übertragen ist ein wesentlicher Bestandteil dieser Theorie. Hinzu kommen weitere Eigenschaften wie die Forderung der Einsatzmöglichkeit in heterogenen Systemen, die einen wesentlichen Einfluß auf die Implementierungssprache nimmt.

Das Java Dynamic Management Kit (JDMK) ist der Versuch, unter der Programmiersprache Java eine Managementumgebung zu realisieren, welche der Forderung nach Flexiblen Management Agenten gerecht wird. Sie basiert auf dem JavaBeans-Konzept, wobei die Management-Intelligenz in den Beans integriert ist. Durch dieses Beans-Konzept ist JDMK in der Lage, Funktionalität zur Laufzeit der Agenten in diese zu integrieren, einfach indem die Klasse über das Netz zu dem Agenten transportiert und dort ausgeführt wird. Zudem wird der Agent handlungsfähig und ist nicht mehr nur der verlängerte Arm des Managers. Diese Flexibilität in der Funktionalität der Agenten erfordert nicht mehr die komplette Festlegung zur Designphase. Es ist auch später im laufenden Betrieb möglich, Managementaufgaben, welche zur Designphase übersehen wurden oder erst im späteren Betrieb auftraten, durch Hinzufügen von Funktionalität zu ermöglichen.
Durch den Java-Ansatz ist JDMK geradezu prädestiniert, in heterogenen Management-Umgebungen eingesetzt zu werden. Die Anbindung an bestehende, noch benötigte Implementierungen, welche in C++ geschrieben wurden, ist durch die Verwendung des Java Native Interface möglich.
Die Evaluierung, in wie weit JDMK die Anforderungen, welche durch die Theorie der Flexiblen Management Agenten gefordert werden, erfüllt und wie leistungsfähig es sich bei einer Implementierung eines Prototypen einer Managementumgebung erweist, ist Ziel dieser Arbeit.

Die Systeme, welche Sprach-Kommunikation ermöglichten hatten sich weit vor der Möglichkeit, Daten zu übertragen entwickelt. Als schließlich die Datenübertragung immer umfangreicher und komplexer wurde, wurden diese Systeme erweitert, jedoch unabhängig von den bisherigen Kommunikationssystemen (z.B. Telefonanlagen). Dies hat in der letzten Zeit, in der dieser Markt härter umkämpft wird, zu einem Konkurrenzkampf zwischen diesen Anbietern entwickelt. Technisch gesehen ist es beiden Systemen möglich, beide Arten

der Kommunikation zu unterstützen. Für den Kunden bedeutet dies eine Reduzierung der erforderlichen Kommunikationseinrichtungen, welche bisher getrennt zu beschaffen waren.
Auf der Sprachkommnikationsseite wird nun versucht, Sprache und Daten innerhalb eines Systems zu integrieren. Im Falle der Firma Siemens ist dies die Telefonanlage Hicom 300. Dieses System bietet die Kommunikation (Sprache und Daten) über ATM und Internet. Im Falle des Internet wird diese Anbindung durch Einsatz eines Telephonie Internet Servers (TIS) geleistet. Mittels dieses Servers können verschiedenartige Endgeräte miteinander verbunden werden. So soll es zukünftig möglich sein, innerhalb einer Videokonferenz auch "normale" Telefone oder PC miteinzubinden.

1.2 Aufgabenstellung und Gliederungsbeschreibung

Die Aufgabenstellung ist, das Java Dynamic Management Kit der Theorie der Flexiblen Management Agenten gegenüberzustellen und die Defizite in der Umsetzung zu untersuchen. Zudem muß JDMK seine Brauchbarkeit innerhalb einer Managementumgebung, welche durch den Telephony Internet Server der Firma Siemens repräsentiert wird, beweisen. Es soll überprüft werden, ob das Management durch die Verwendung des JDMK erleichtert wird und Sicherheitsdefizite, welche durch die SNMP-Administration auftreten, behoben werden können.
Die Arbeit gliedert sich in folgende Kapitel:

- **Kapitel 2**
  In diesem Kapitel wird die Theorie der Flexiblen Management Agenten (FMA) dargestellt. Zu Beginn werden wichtige Konzepte, welche damit in Zusammenhang stehen, erläutert. Dies ist einmal das Management by Delegation Konzept. Weiter werden die Forderungen an Intelligente Agenten formuliert, welche eine Vorstufe zu den FMA´s darstellen.
  Zum späteren Vergleich mit der Theorie der FMA´s wird anschließend ein kurzer Einblick in bereits bestehende Paradigmen und Systeme gewährt, wobei die Schwächen dieser Systeme herausgearbeitet werden.
  Schließlich wird die Theorie der FMA´s dargestellt und die Anforderungen, welche an ein System, in dem diese Agenten agieren, gestellt werden, aufgeführt. Ebenso werden die erforderlichen Merkmale der Agenten selbst dargestellt.

- **Kapitel 3**
  Dieses Kapitel beschäftigt sich mit dem Managementtool JDMK und stellt dessen Umfang und Möglichkeiten dar. Es werden die zentralen Konzepte des JDMK dargestellt. Diese Konzepte sind:
  - Core Management Framework (CMF)
    Das Konzept dieses Frameworks ist es, eine Bean-Box zur Verfügung zu stellen, um die M-Beans bei Bedarf in den Agenten "einhängen" zu können und diese danach wieder zu entfernen. Sie bildet die Schnittstelle zwischen den Adaptoren für die verschiedenen Protokolle und den M-Beans.
  - Management Beans (M-Beans)
    Diese Beans, welche nach dem JavaBeans-Konzept realisiert sind, bilden die "Intelligenz" der Agenten. Sie werden in das Framework integriert. Diese Integration

kann durch den Agenten selbst oder durch eine Management-Applikation, aber auch durch einen anderen Agenten geschehen.

  - Client Beans (C-Beans)
    Die C-Beans stellen die Repräsentanten der M-Beans auf der Manager-Seite dar. Durch die Verwendung dieser Beans wird mehr Transparenz beim Zugriff auf die M-Beans eines Agenten erreicht.
  - Adaptoren
    Die Adaptoren bilden die Schnittstelle zur Außenwelt und umfassen derzeit Protokolle wie RMI, HTTP (TCP/UDP/SSL), HTML[1] , IIOP und SNMP. Auch diese Adapter können zur Laufzeit dem Agenten hinzugefügt werden.
  - Adaptor-Clients
    Dieses Beans bieten dem Manager die Möglichkeit, auf einen Agenten zuzugreifen, ohne die Spezifika des jeweiligen Protokolls zu berücksichtigen. Es muß anfangs lediglich der entsprechende Adapter ausgewählt werden (mit Angabe des Host und des Ports) und ab diesem Zeitpunkt wird jeder Zugriff unabhängig vom verwendeten Protokoll gleichartig durchgeführt.
  - Dienste
    Es werden eine Reihe von Diensten zur Verfügung gestellt, welche das Management der Agenten unterstützen. Diese Dienste betreffen den Bereich Ereignisse, Discovery und Sicherheit, sowie Andere, welche unterstützende Funktion besitzen.

  Weiterhin wird die Kombination von JDMK mit bestehenden MIB´s betrachtet. Diese Kombination ist besonders in Hinblick auf die Realisierung des Prototypen wichtig. Nach der Darstellung der Theorie und des Managementtool soll nun auf die Managementumgebung, in welcher der JDMK-Prototyp eingesetzt werden soll, eingegangen werden. Dies geschieht im folgenden Kapitel.

- **Kapitel 4**
  In diesem Kapitel wird auf die Merkmale des Telephony Internet Servers eingegangen. Es werden die wichtigsten Komponenten wie Gateway, Gatekeeper und Administration Maintenance Server dargestellt, sowie die Funktionsweise beschrieben.
  Die Form der Kommunikation zwischen den Komponenten des TIS bzw. die Verwendung eines eigenen Nachrichtenformats verdient hierbei besondere Beachtung, da dies bei der Entwicklung des Prototypen eine wesentliche Anforderung war.
  Um einen Einblick in das bestehende Management-Umfeld zu erhalten, wird anschließend die Realisierung der derzeitigen Management-Lösung vorgestellt. Diese Lösung verwendet Windows NT und den darunter erhältlichen Windows NT SNMP Agenten. Die Funktionsweise dieses Agenten wird dargestellt und die Integration des SXA, eines Windows NT SNMP Extension Agenten, welcher die TIS-MIB verwaltet, dargestellt.

- **Kapitel 5**
  Dieses Kapitel behandelt die Evaluierung von JDMK gegenüber dem Konzept der Flexiblen Management Agenten. Zudem wird die Brauchbarkeit von JDMK in einer

[1] Offensichtlich ist HTML kein Protokoll. Es ist hier jedoch ebenfalls aufgeführt, da unter JDMK ein HTML-Adapter zur Verfügung gestellt wird. Dieser Adapter kommuniziert über HTTP, bietet jedoch kein Sicherheitskonzept (vgl. 3.4.5). Im weiteren wird HTML im Zusammenhang mit den Protokollen immer unter diesem Gesichtspunkt gesehen.

CORBA-Umgebung untersucht und die Anbindungsmöglichkeiten aufgezeigt. Schließlich wird auf die möglichen Konzepte einer JDMK-Managementumgebung eingegangen. Hierbei sind die unter Kapitel 4 dargestellten TIS-Eigenschaften zu berücksichtigen.

- **Kapitel 6**
  In diesem Kapitel wird schließlich das Konzept, welches aus den in Kapitel 5 dargestellten Vorschlägen ausgewählt wurde, vorgestellt. Da diese Lösung eine Generierung von Java-Dateien aus einer gegebenen MIB beinhaltet, wird diese Prozeß betrachtet. Es werden die Klassen, welche aus den Gruppen und Elementen der MIB erzeugt wurden, aufgezeigt und Problematiken aber auch Vorteile, welche sich daraus ergeben, genannt. Die dadurch entstandene Rahmenstruktur ist nur die Basis für die eigentliche Implementierung des Prototypen. Anhand der implementierten Teilbereiche der MIB, welche zentrale Teile davon repräsentieren, wird die Erstellung der eigentlichen Implementierung, sowie dabei verwendete Lösungen vorgestellt. Ein wesentlicher Punkt ist die Entwicklung der Schnittstelle, welche durch JNI realisiert wurde. An dieser Stelle wird auf die Eigenschaften und Besonderheiten des Java Native Interface eingegangen. Es werden Problematiken und Schwierigkeiten, welche sich bei der Implementierung ergaben erläutert, sowie die gewählten Implementierungslösungen dargestellt.
  Die durch JDMK zur Verfügung gestellten Dienste wurden größtenteils, wo es sinnvoll erschien, geprüft. Es werden explizit der Event-Service, welcher die Grundlage für mehrere Dienste des JDMK wie Monitoring bildet, ausführlich auf Brauchbarkeit getestet. Als zweiten wichtigen Dienst sei hier der Discovery-Service zu nennen, welcher es ermöglicht, JDMK-Agenten innerhalb eines Netzwerks zu lokalisieren. Andere Dienste greifen bei diesen Tests nur unterstützend ein und werden daher nicht explizit aufgeführt. Die Möglichkeiten dieser Dienste wird in der Darstellung des JDMK in Kapitel 3.7 ausführlich erläutert. Die während der Testphase gemachten Erfahrungen schließen dieses Kapitel ab.

- **Kapitel 7**
  An dieser Stelle werden die wichtigen Punkte der Diplomarbeit zusammenfassend dargestellt. Dabei werden tabellarisch die Ergebnisse der Gegenüberstellung von JDMK und der Theorie der Flexiblen Managementagenten aufgeführt. Es wird der Vorgang der Prototyperstellung vorgestellt und ein kurzer Einblick in die Testphase gewährt. Als letzter wichtiger Punkt wird kurz auf das Java Native Interface eingegangen, welches bei der Implementierung der Schnittstelle eine zentrale Rolle gespielt hat.
  Zuletzt wird ein Ausblick über die weitere (wahrscheinliche) Entwicklung des JDMK gegeben, sowie die Integration dieser Technologie in andere Managementsysteme vorgestellt.

Kapitel 2

Das Konzept der Flexiblen Management Agenten

Im Kapitel 1 wurde der Bedarf an einer Netzmanagementstruktur angedeutet, welche als Theorie der Flexiblen Management Agenten (FMA) bezeichnet wird. Da diese Theorie die Grundlage für die Philosophie des Java Dynamic Management Kit darstellt, sollen hier wesentliche Elemente dieser Theorie vorgestellt werden. Eine grundlegende Betrachtung ist in [Moun 97] zu finden.
Um auf die Klassifizierung von FMA's durchführen zu können ist es notwendig, zuerst die Anforderungen, welche zur Nutzung der FMA's führen, darzustellen. Diese Anforderungen umfassen im Wesentlichen:

- Management by Delegation
- Intelligente Management Agenten

Diese Anforderungen werden in den nächsten zwei Kapiteln betrachtet. Danach wird auf die Schwächen bestehender Managementsysteme eingegangen um schließlich die Flexiblen Management Agenten darzustellen, welche die Unzulänglichkeiten dieser bestehenden Systeme beseitigen.

2.1 Management by Delegation

Das Konzept des MbD besteht im Kern aus dem Ziel, Managementfunktionalität zu Agenten zu delegieren. Indirekt fordert dieses Konzept dadurch die dynamische Erweiterbarkeit dieser Agenten. Delegierung umfaßt drei zeitlich unterscheidbare Vorgänge. Sie beschreiben den Ablauf einer Delegierung von Managementfunktionalität:

- Before Delegation
  Diese Phase, welche den Zeitraum bis zur Delegierung umfaßt, ist geprägt von der Entscheidung welche Funktionalität delegiert werden soll. Es ist auch festzulegen, welches Ereignis die Delegierung initiiert, sowie die Auswahl des zu delegierenden Agenten. Zudem ist festzulegen, auf welche Art und Weise die Funktionalität in dem Agenten integriert werden soll. Zum einen kann diese statisch hinzugebunden werden, was eine Recompilierung des Agenten bedeuten würde, zum anderen kann es auch dynamisch

geschehen. Die Ereignisse, welche zur Delegierung führen können, können durch Benutzeranforderungen, Manager-Applikationen oder auch dem Agenten selbst herrühren. Schließlich ist auch eine sinnvolle Delegierung aufgrund der Netztopologie anzustreben.

- During Delegation
  Ist die Art der Delegierung in der "Before"-Phase abgeschlossen, so kann unter gegebenen Umständen die Delegierung stattfinden. An dieser Stelle ist von wesentlicher Bedeutung, wie die Delegierung stattfindet. Diese Frage betrifft den Delegations-Mechanismus, sowie das darunterliegende Modell.

  - Delegations-Mechanismus
    Durch diesen Mechanismus wird entschieden, welche Programmiersprache gewählt werden soll, um die geforderte Funktionalität zu implementieren. Es muß an dieser Stelle beachtet werden, daß Sprachen wie C++ sicherlich Laufzeitvorteile besitzen, Java jedoch den unbestreitbaren Vorteil besitzt, in heterogenen Umgebungen ohne Anpassung an die rechnerspezifischen Erfordernisse eingesetzt werden zu können, da dies durch die entsprechende Java Virtual Machine erledigt wird.
    Zudem wird das zu verwendende Protokoll bestimmt, welches bei der Übertragung der Managementfunktionalität benutzt werden soll. Hierbei ist eine Betrachtung der Sicherheitskonzepte, welche die einzelnen Protokolle bieten, unbedingt notwendig.
  - Delegations-Modell
    Das Modell bezieht sich auf die Verwendung des Push-/Pullmechanismus. Der Unterschied besteht in der Art und Weise, wie die Funktionalität zum Agenten gelangt. Einmal kann sie von "außen" durch eine Manager-Applikation initiiert werden (PUSH), aber auch der Agent selbst kann bei Bedarf die geforderte Funktionalität anfordern (PULL).

- After Delegation
  Ist der Agent um die erforderliche Funktionalität erweitert worden, so ist festzulegen, wie lange die Funktionalität dem Agenten erhalten bleibt bzw. wie sie ausgeführt werden soll. Grob kann diese Frage in drei Punkten zusammengefaßt werden:

  - dauerhafte Erweiterung der Funktionalität und kontinuierliche Ausführung,
  - dauerhafte Erweiterung der Funktionalität und ereignisgesteuerte Ausführung,
  - einmalige Ausführung mit Entfernen der Funktionalität bei Beendigung.

Der Zusammenhang zwischen den drei Delegationsphasen ist in Abbildung 2.1 zu sehen. Sie demonstriert die Phasen beispielhaft an der Delegierung eines Listeners[1] für aktive Agenten. In der "Before-Delegation"-Phase wird durch die Management-Applikation ein Listener-Objekt zu dem entsprechenden Agenten transportiert. Danach wird diese Funktionalität als Task auf dem Agenten ausgeführt. Während dieser Phase hat die Management-Applikation die Möglichkeit, den Status des Agenten, welcher delegiert wurde, abzufragen. Nach Beendigung des Tasks wird das Objekt im Zuge der "After-Delegation"-Phase entweder entfernt oder unter

[1] Das Listener-Konzept bietet die Möglichkeit, Veränderungen bei Managed Objects oder anderen Agenten zu registrieren und dadurch nicht auf eine direkte Benachrichtigung angewiesen zu sein. Insbesondere dient dieses Konzept dazu, die Netzlast nur dann zu erhöhen, falls eine Information über die Veränderungen erwünscht ist. Es muß also nicht dafür gesorgt werden, daß der Agent bzw. das Managed Object über jede Veränderung seien Manager benachrichtigt.

bestimmten Voraussetzungen[2] im Agenten beibehalten. Es sei darauf hingewiesen, daß der

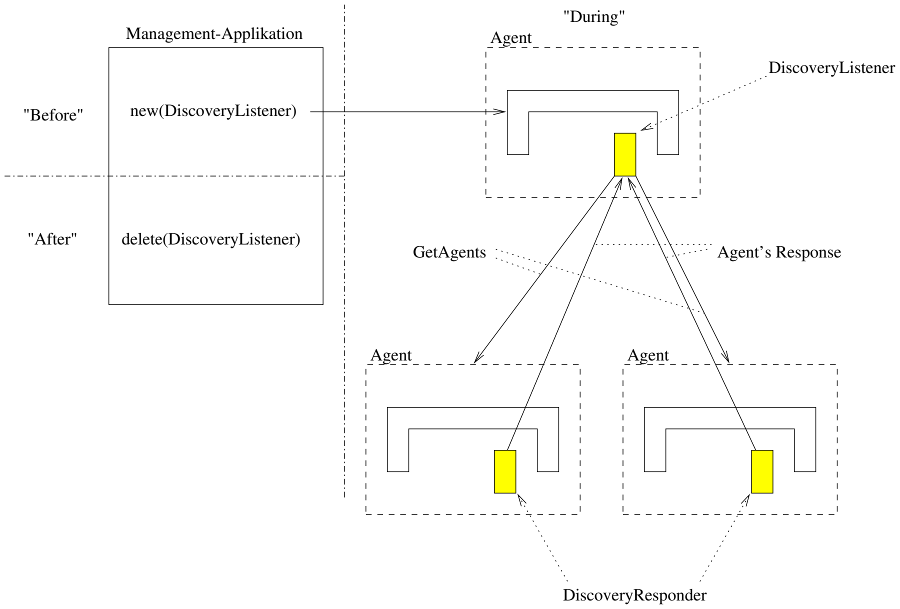

Abbildung 2.1: Phasen des Management by Delegation

Begriff des "Management by Delegation" in der Verwendung außerhalb der Flexiblen Management Agenten zu erweiterbaren Agenten führt (sog. Mid-Level-Manager), welche zwar Funktionalität besitzen, jedoch unter der vollen Kontrolle des Managers laufen. Im Sinne der Theorie der Flexiblen Management Agenten bedeutet Management by Delegation, daß die Agenten, welche Funktionalität durch Manager erhalten, weitgehend selbständig agieren. Es ist kein weiterer Eingriff durch den Manager erforderlich. Zusätzlich besitzen diese Agenten die Fähigkeit, Funktionalität selbst zu anderen Agenten zu delegieren.

2.2 Intelligente Management Agenten

Das Konzept dieser Agenten zielt darauf, den Agenten umfassendere Eigenschaften zuzuordnen. Es gibt mehrere Definitionen welche diese Art von Agenten klassifizieren. In [Moun 97] ist folgende Definition bzw. eine Reihe von Eigenschaften aufgeführt.

- <u>Autonomie</u>
  Diese Eigenschaft ermöglicht es Agenten, ohne direkte Einwirkung von Außen, zum Beispiel durch Benutzereingriffe, zu agieren. Somit können Agenten einen Teil der Managementaufgaben selbst übernehmen.

[2] Diese Voraussetzungen beinhalten die Möglichkeit, Funktonalität dauerhaft im Agenten zu belassen oder zu entfernen. Die Funktionalität kann auch ohne Unterbrechung durchgeführt oder nur bei Erhalt eines Signals vom Manager wieder ausgeführt werden.

- Kooperationsfähigkeit
  Die Kooperationsfähigkeit soll nicht nur auf Interaktion mit einem Administrator beschränkt sein. Agenten mit dieser Eigenschaft besitzen die Möglichkeit, mit anderen Managementkomponenten zu kommunizieren. Im Sinne des Management by Delegation sind sie somit auch in der Lage, Managementaufgaben anderen Agenten zu übertragen bzw. diese zu beauftragen.

- Reaktionsfähigkeit
  Zusätzlich zur Eigenschaft der Autonomie müssen diese Agenten zudem die Fähigkeit besitzen, auf Veränderungen, welche in ihrem Einflußbereich geschehen, reagieren zu können.

- Aktionsfähigkeit
  Außer der Reaktionsfähigkeit fordert das Konzept der Intelligenten Management Agenten, daß der Agent die Fähigkeit besitzen muß, selbst Aktionen ausführen zu können.

In einer Managementumgebung, welche der Theorie der Flexiblen Management Agenten genügen soll, werden an Intelligente Agenten folgende Anforderungen gestellt:

- Skalierbarkeit
  Der Kontext des Agenten (bzw. des Agentenprozesses) muß situationsbedingt anpaßbar sein. Dies wird notwendig, da in heterogenen Netzen nicht an allen Komponenten gleiche Laufzeitumgebungs-Bedingungen anzutreffen sind.

- Plattformunabhängigkeit
  Ebenso ein zentrales Thema in heterogenen Netzen ist die Fähigkeit, ohne große Eingriffe in die Implementierung des Agenten, diesen auf allen zu managenden Netzknoten einsetzen zu können.

- Zeitliche Koordinationsfähigkeit
  Eine der Anforderungen, welche ein modernes Netz bewältigen muß, ist die Koordinierung von zeitkritischen Abläufen.

- Kausalitätsbewußtsein
  Die Auswirkungen der Managemententscheidungen eines Agenten müssen diesem bekannt sein, nur so kann unter Umständen ein Zurücknehmen (oder Zurückweisen) der Management-Entscheidung erreicht werden, falls dadurch festgelegte Rahmenwerte verletzt werden.

- Sicherung des Quality of Service
  Die Sicherung des Quality of Service ist eine zentrale Forderung an Intelligente Agenten. Hier wird der Punkt der Kooperationsfähigkeit besonders wichtig, da zum Beispiel nur in Absprache mit in topologischer Nähe angesiedelten Agenten eine ausreichende Bandbreite für eine Kommunikation erreicht werden kann.

2.3 Herkömmliche Agentensysteme

Die Unterscheidung von Agentensystemen kann in zwei Stufen erfolgen. In der ersten Stufe wird die Unterscheidung zwischen zentralisierten und verteilten Ansätzen getroffen:

- zentralisierter Ansatz
  Der zentralisierte Ansatz stellt die erste Stufe von Agentensystemen dar. Dieser Ansatz verlegt die gesamte Intelligenz des Managementsystems in die zentrale Managerinstanz. Der Manager benutzt die Agenten, um entsprechende Managementinformationen zu sammeln und diese zu verwerten. Dieses Szenario führt zu einer hohen Netzbelastung für reine Administrationszwecke, welche die Bandbreite für die eigentlichen Nutzdaten verringert. Auch wird hier der Manager zu einem Flaschenhals der Managementinformation und es müssen zusätzliche Vorrichtungen geschaffen werden, um die Flut der Informationen zu koordinieren.

- verteilter Ansatz
  Der verteilte Ansatz stellt eine Weiterentwicklung des zentralisierten Ansatzes dar. Er verschiebt Managementfunktionalität in der Hierarchie nach unten zu den eigentlichen Agenten. Wie bei zentralisierten Ansätzen herrscht eine Manager-Agent Hierarchie vor, die es den Agenten selbst nicht ermöglicht, in einem gewissen Rahmen selbständig Managementaktionen durchzuführen. Es wird lediglich Funktionalität, welche bisher im Manager integriert war, nach unten verschoben.

Als weiteres Unterteilungskriterium ist nun zu betrachten, wie bei verteilten Ansätzen die Funktionalität in den Agenten integriert werden soll. Es kommen zwei Möglichkeiten in Betracht:

- statische Integration
  Der Fall der statischen Integration führt zur Verwendung von einfach strukturieren Agenten, da eine Änderung des Funktionsumfanges bei einem gewissen Grad von Funktionalität des Agenten sehr aufwendig werden kann. Die Veränderung der Funktionalität eines statischen Agenten hat in der Regel eine Änderung und Neuübersetzung des Agenten-Codes zur Folge. Daher muß zusätzlich zu diesem Aufwand auch für eine entsprechende Entwicklungsumgebung auf der Agentenmaschine gesorgt werden. Diese Nachteile führen zu einem geringen Grad an Flexibilität[3]. Als Beispiel sind in dieser Kategorie herkömmliche SNMP-Agenten und Modulare SNMP-Agenten (AgentX) zu nennen.

- dynamische Integration
  Um den Grad an Inflexibilität des statischen Ansatzes zu verringern, wurde das Konzept der dynamischen Erweiterbarkeit entworfen. Dieses Konzept fordert den Einsatz von Intelligenten Agenten (siehe Kap. 2.2). Auch die Forderungen, welche im "Management by Delegation"-Konzept erhoben werden, werden bis zu einem gewissen Grad erfüllt. Diese Agentensysteme sind als Vorstufe der Flexiblen Management Agenten zu sehen. Der Hauptunterschied zu den Eigenschaften, welche Flexible Agenten ausmachen ist, daß bei den bisherigen Systemen die Funktionalität nur auf den Management-Knoten lauffähig ist, auf denen die jeweiligen Agenten laufen. Damit ist ein Einsatz in heterogenen Netzen nur bedingt möglich. Auch muß dafür gesorgt werden, die Funktionalität

[3]Es ist zu beachten, daß der Begriff "Flexibilität" in mehrfacher Art und Weise gebraucht werden kann. Einmal kann die Flexibilität die Fähigkeit beschreiben, Funktionalität zu delegieren. Auch kann Flexibilität lediglich die Möglichkeit darstellen, einen Agenten zu erweitern. Dies ist nicht mit der Delegierung gleichzusetzen. Der SNMP-Agent von Windows NT kann nicht zur Laufzeit mit zusätzlicher Funktionalität ausgestattet werden, jedoch kann bei einem Neustart des Dienstes der Funktionsumfang durch Hinzunahme von Bibliotheken erweitert werden.

stets in konsistentem Zustand auf den entsprechenden Management-Knoten verfügbar zu halten.

In Abbildung 2.2 ist der Zusammenhang zwischen den angeführten Klassifizierungen dargestellt.

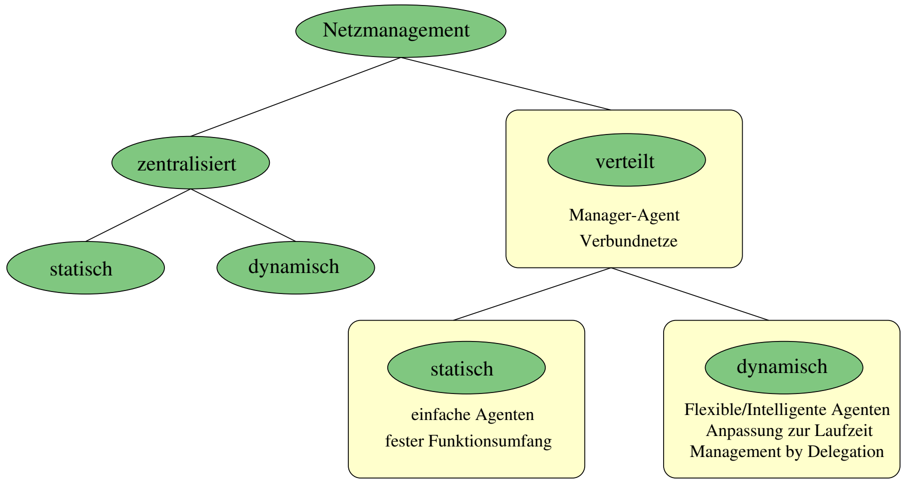

Abbildung 2.2: Formen des Netzmanagement

2.4 Strukturelle Merkmale einer FMA-Netzstruktur

Die Merkmale gliedern sich in drei Teilbereiche. Der erste Teilbereich umfaßt die Komponenten, welche innerhalb eines solchen Netzes miteinander interagieren. Im zweiten Teilbereich wird festgelegt, wie die Komponenten miteinander kommunizieren bzw. welche Anforderungen an ein System gestellt werden, um eine Kommunikation zu ermöglichen. Im letzten Teilbereich werden schließlich die Formen der Kommunikation behandelt.

2.4.1 Organisatorische Merkmale

Es gibt drei Arten von Komponenten, welche miteinander interagieren:

- Agenten
  Dies sind Agenten im herkömmlichen Sinn. Auch diese Agenten (z.B. SNMP-Agenten) werden in einer Netzstruktur für Flexible Management Agenten (FMA) verwendet. Es ist nicht unbedingt zwingend, daß alle Agenten, welche innerhalb dieser Struktur verwendet werden, FMA´s sein müssen. Vielmehr ist eine Gliederung denkbar, in welcher FMA´s als übergeordnete Agenten mit herkömmlichen Agenten interagieren und die Schnittstelle zum Manager bilden. Dabei sind die herkömmlichen Agenten bestimmten Managed Objects zugeordnet und geben Informationen an FMA´s oder direkt an Manager weiter.

- Flexible Management Agenten
  Diese Agenten erhalten die Funktionalität von Agenten mit der Möglichkeit, die Funktionalität zur Laufzeit zu erweitern. Sie werden meist in einer Client-Server Rolle eingesetzt, in dieser können sie auch selbst eine Manager-Rolle gegenüber anderen Agenten einnehmen. Die Hauptmerkmale von solchen Agenten sind deren Erweiterbarkeit und die Fähigkeit, mit anderen FMA´s zu kooperieren. Auf die konkreten Eigenschaften solcher Agenten wird in Kapitel 2.5 eingegangen.

- Manager
  Manager besitzen besondere Privilegien innerhalb der Managementstruktur. Sie haben Zugriff auf alle Elemente und Managed Objects und bieten für den Fall der Administration eine Benutzerschnittstelle.

Um FMA´s innerhalb einer solchen Struktur anzusiedeln, müssen bestimmte Informationen für die jeweiligen Agenten vorhanden sein. Diese Informationen gliedern sich in:

- Domäne
  Hier wird festgelegt, in welcher Domäne der Agent angesiedelt ist. Die Domäne muß einen eindeutigen Bezeichner besitzen. Dieser Bezeichner wird um die Agentenspezifikation erweitert. Diese Spezifikation orientiert sich nach dem Funktionsumfang des Agenten. Es wird ein domänenspezifischer Nameserver (ANS[4]) für die Auflösung von Agentenbezeichnern und Netzwerkadressen zur Verfügung gestellt. Es ist eine Aufteilung von Domänen in Subdomänen möglich.

- Gruppe
  Eine Gruppe ist eine logische Ansammlung von FMA´s, welche eine gemeinsame Managementaufgabe bewältigen müssen. Gruppen können eine oder mehrere Domänen umfassen. Der Unterschied zwischen Gruppen und Domänen besteht darin, daß Domänen nach organisatorischen und administrativen Aspekten gebildet werden, während sich Gruppen eher nach funktionalen Gesichtspunkten orientieren. Es können Peer-to-Peer und hierarchische Gruppierungen sowie eine Kombination daraus gebildet werden.

2.4.2 Kommunikationsmerkmale

Hier werden die Arten von Daten festgelegt, welche zwischen den Kommunikationspartnern, wie in 2.4.1 beschrieben, ausgetauscht werden. Es gibt drei Arten:

- Tasks (Management-Anforderungen)
  Dies sind Anforderungen, welche den entsprechenden Agenten veranlassen, bestimmte Managementaktionen durchzuführen. Der Agent muß über die entsprechende Funktionalität verfügen.

- Information
  Hier handelt es sich um die eigentliche Information, wie Einträge einer Routing-Tabelle, Status eines Netzwerkdruckers oder anderen Informationen.

- Funktionalität
  Diese Art der Daten kommt dann zum Tragen, wenn im Sinne des Management by

[4] ANS = Agent Name Server

Delegation Konzepts Funktionalität zu den Agenten delegiert wird. Es handelt sich um ausführbaren Funktionscode.

Um die Kommunikation zu ermöglichen, muß die Netzstruktur einige Dienste zur Verfügung stellen:

- Delegations-Dienst (Delegation Service)
  Dieser Dienst ermöglicht es FMA´s untereinander bzw. über Manager Funktionalität zu delegieren. Zudem muß dieser Dienst die Anforderung eines Tasks ermöglichen. Dieser Dienst ist als Implementierung des "Management by Delegation"-Konzepts zu sehen.

- Ereignis-Dienst (Event Service)
  Dieser Dienst ermöglicht es Agenten, auf Ereignisse, welche über das Netz bekannt gemacht werden, zu achten. Zu diesem Zweck existieren sogenannte Ereignisserver (AES[5]). Durch einen Eintrag in einem solchen Server kann der Agent die Arten von Ereignissen festlegen, bei deren Eintreffen er benachrichtigt werden will.

- Gruppierungs-Dienst (Group Service)
  Er dient zur Ermöglichung von Gruppenkommunikation mittels Multicast-Nachrichten. Zudem muß er Methoden zum Hinzufügen und Entfernen von Agenten in bzw. aus einer Gruppe bereitstellen.

- Agent-Bestimmungs-Dienst (Location Service)
  Dies ist ein Dienst zur Bestimmung der Position eines Agenten innerhalb einer Managementumgebung. Er umfaßt die Identifizierung, die Position und Verfügbarkeit der jeweiligen Komponenten. Auch ist die Adressenauflösung mit diesem Dienst durchzuführen. Er ist meist an einer zentralen Stelle organisiert, in welcher die Agenten und deren Funktionalität bzw. die Information darüber verwaltet wird.

- Sicherheits-Dienst (Security Service)
  Insbesondere im Hinblick auf die Delegierung von Tasks und Funktionalität ist der Sicherheitsaspekt innerhalb einer Managementarchitektur bedeutend. Dieser Aspekt umfaßt die Authentifizierung und Autorisierung von Managementelementen, welche Dienste von Agenten in Anspruch nehmen.

- Management-Dienst (Management Service)
  Um eine Kontrolle über den Fortgang einer Management-Operation (Task) oder andere Information über Agenten zu erhalten, sind Mechanismen notwendig, welche in diesem Dienst zur Verfügung gestellt werden.

2.5 Anforderung an Flexible Management Agenten

Die Theorie der Flexiblen Management-Agenten versucht, die Schwachstellen in bisherigen Systemen zu beheben. Diese Agenten erfüllen die Forderung an Intelligente Management-Agenten, auch verfolgen sie das Konzept des Management by Delegation. Um innerhalb der unter 2.4 beschriebenen Umgebung integriert werden zu können, müssen diese Agenten folgende Anforderungen erfüllen:

[5] Agent Event Server

- Plattformunabhängigkeit
  Diese Forderung hat zum Ziel, FMA´s in heterogenen Managementumgebungen einsetzen zu können. Diese Unabhängigkeit ermöglicht es, Funktionalität ungeachtet der darunterliegenden Hardware zu delegieren. Dies stellt Anforderungen an die Sprache, mit welcher der Agent bzw. die Funktionalität implementiert wurde. Eine kurze Betrachtung dazu ist in Kapitel 2.6 zu finden.

- Standardisierte Ansammlung von Diensten
  Dies ist notwendig, um den Agenten in das Managementsystem zu integrieren. Dabei handelt es sich um Dienste der Kommunikation, Delegation usw., welche eigentlich unabhängig von der Managementaufgabe sind und nur zur Einbettung des Agenten dienen.

- Flexibilität
  Die Flexibilität, welche diese Art von Agenten auszeichnet, stellt sich in der Möglichkeit dar, den Agenten ohne großen Aufwand mit mehr Funktionalität zu versorgen. Es muß lediglich im Sinne des Management by Delegation die Funktionalität im Agenten integriert werden. Es muß keine Veränderung an der Implementierung des Agenten vorgenommen werden. Zudem kann die Erweiterung zur Laufzeit geschehen.

- Push/Pull Mechanismus
  Dieser Mechanismus beschreibt die Art und Weise, wie Managementfunktonalität zu den Agenten gelangt. Eine Möglichkeit ist, daß die Funktionalität durch den Manager zu dem Agenten transportiert und dessen Funktionsumfang erweitert wird. Da die Funktionalität zu dem Agenten "geschoben" wird, wird dieses Verfahren als "PUSH"-Mechanismus bezeichnet. Falls der Agent jedoch selbst seinen Funktionsumfang durch Anforderung einer Funktionalität erweitert, Funktionalität also zu sich "zieht", wird dieses Verfahren als "PULL"-Mechanismus bezeichnet.

- Management by Delegation
  In Kapitel 2.1 ist dieses Konzept beschrieben worden. An dieser Stelle sei darauf hingewiesen, daß Agenten, welche in einem solchen Managementsystem agieren die Fähigkeit besitzen müssen, delegiert zu werden und auch selbst delegieren zu können. In diesem Kontext muß ein Agent in der Lage sein, lokale wie entfernte Ausführung von Funktionalität durchführen zu können.

- Kaskadierende Aufrufe (FMA-FMA)
  Die Fähigkeit, kaskadierende Aufrufe innerhalb einer Managementumgebung zu ermöglichen ist eines der hervorstechensten Merkmale dieses Agentenkonzepts. Damit kann in beträchtlichem Maße die Netzbelastung reduziert werden, da nicht für sämtliche Managementaktionen ein Manager oder Administrator notwendig wird. Dies ist ganz besonders deutlich wenn man den Fall von Software-Aktualisierungen betrachtet, der zu großer Netzbelastung führen kann.

- Kooperation
  Dies ist die Fähigkeit mit anderen Managementagenten zu kooperieren, um eventuell umfangreichere Managementaufgaben übernehmen zu können.

- Kontrollmechanismen (über delegierte Tasks)
  Diese Mechanismen umfassen eine lokale Kontrolle für Tasks, welche zu dem Agenten de-

legiert wurden, sowie eine entfernte Kontrolle für Tasks, welche der Agent selbst delegiert hat.

- Module zur Nutzung von Diensten, welche von der Managementumgebung zur Verfügung gestellt werden. Diese sind:
  - Kommunikation
  - Sicherheit
  - Information
  - Kontrolle

2.6 Implementierungssprache

Bei der Wahl der Sprache, welche für die Implementierung der Agenten bzw. deren Funktionalität gewählt wird, müssen verschiedene Gesichtspunkte berücksichtigt werden. Wird mehr als eine Sprache erlaubt, dann hat dies zum Vorteil, daß die Benutzer davon profitieren. Dabei ist jedoch zu bedenken, daß auf die Übereinstimmung von Versionen der verwendeten Sprachen geachtet werden muß. Man hat die Wahl zwischen Maschinenspezifischen und Interpreter-Sprachen. In [Moun 97] wird die Unterscheidung zwischen Low-Level und High-Level Sprachen getroffen, wobei die Low-Level Sprachen den maschinennahen und die High-Level Sprachen den Interpreter-Sprachen entsprechen. Nachfolgend sind Vorteile und Nachteile der Sprachen aufgeführt:

- Low-Level
  Diese Sprachen sind meist schwieriger zu implementieren, wodurch die Handhabung erschwert wird. Diese Variante wird meist benutzt, um die Performance bei bestimmten Managed Objects zu erhöhen. Dies ist besonders bei laufzeit-kritischen Anwendungen der Fall, in denen die Hardware-Nähe dieser Sprachen besondere Vorteile bringt. Zudem ist der Speicherplatzverbrauch gegenüber Interpreter-Sprachen meist wesentlich geringer. Sprachen, welche in diese Kategorie gehören sind klassische Assembler-Sprachen, wurden aber im Laufe der Zeit durch leichter programmierbare Sprachen wie C und C++ ersetzt.

- High-Level
  Diese Sprachen sind leichter zu handhaben, was die Implementierung der Agenten erleichtert. Es ist auch (unter Verwendung einer Skript-Sprache wie Tcl) möglich sehr schnell zum Testen kleine Applikationen zu erstellen. Es werden mittels dieser Sprachen maschinenspezifische Problematiken vor dem Entwickler verborgen. Nachteile dieser Sprachen sind deren Verbrauch an Systemresourcen wie Speicherplatz und Laufzeit. Wie bereits bei den Low-Level Sprachen erwähnt, handelt es sich hier häufig um Interpreter- oder Skript-Sprachen. Es können für die Implementierung Sprachen wie Java, Tcl/Tk und Perl benutzt werden.

Kapitel 3

JDMK - Java Dynamic Management Kit

Das Java Dynamic Management Kit[1] ist ein Netzwerkmanagementtool auf der Basis von Java. Es nutzt intensiv die Theorie der Flexiblen Management Agenten, d.h. es ist unter JDMK möglich, einen laufenden Agenten zu erweitern bzw. seine Funktionalität zu verändern. Es ist lediglich zu beachten, daß aufgrund der Java-Basis eine Java Virtual Machine auf dem Rechner, auf welchem der Agent läuft, vorhanden sein muß. Das Kommunikationskonzept beruht auf der Manager-Agent und Agent-Agent Funktionalität und zeigt hier schon die Nähe zur Theorie der Flexiblen Managementagenten.
Die Merkmale und Struktur sowie die Schlüsselkonzepte des JDMK sind in Abbildung 3.1 dargestellt. Nachfolgend wird auf die einzelnen Elemente eingegangen.

3.1 Core Management Framework

Das Grundgerüst für die Manager und Agenten unter JDMK ist das Core Management Framework (CMF). Es ist ein Art Rahmen für den Einbau von erforderlicher Funktionalität und kann im Umfang zur Laufzeit verändert werden[2] . In das CMF werden zu Laufzeit Objekte, welche als Management-Beans (M-Beans) bezeichnet werden eingebunden und auch wieder entfernt, falls sie nicht mehr benötigt werden. Die M-Beans stellen ein weiteres Schlüsselkonzept des JDMK dar und sind in Kapitel 3.2 beschrieben. Das Erweitern des CMF durch M-Beans wird als Registrierung derselben bezeichnet und kann auf zwei Arten geschehen:

- Der Agent, der in seiner Grundstruktur durch das CMF realisiert ist kann zur Startzeit M-Beans registrieren. Dies kann durch entsprechende Methodenaufrufe im Agenten geschehen. Es können Klassen bei Start des Agenten oder aber auch später zur Laufzeit eingelesen und daraus M-Bean-Objekte erzeugt werden.

- Außerhalb des Agenten kann eine Registrierung durch eine Manager- oder Agentenapplikation durchgeführt werden. Dabei kommt ein weiteres Schlüsselkonzept des JDMK ins

[1] JDMK wurde von Sun Microsystems Inc. entwickelt und ist derzeit in der Version 3.0 erhältlich.

[2] Das Core Management Framework als Java-Objekt kann zur Laufzeit nicht erweitert werden. Diese Formulierung zielt darauf, daß die Rahmenstruktur, welche durch das CMF-Objekt organisiert werden kann, durch M-Beans zur Laufzeit erweitert wird. Es wird im weiteren Verlauf vereinfacht als die Erweiterung des CMF bezeichnet.

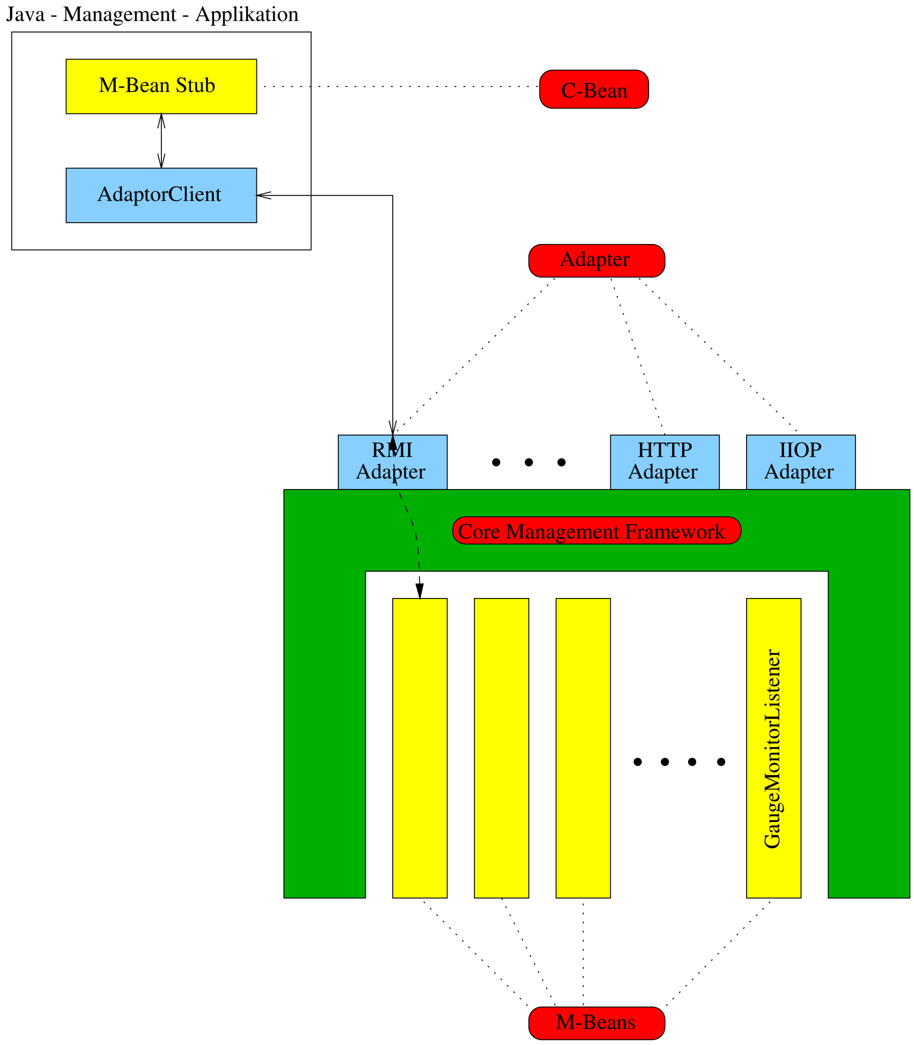

Abbildung 3.1: Übersicht über die Schlüsselelemente des JDMK

Spiel, das Adapterkonzept. Über diese Adapter können Applikationen mit dem JDMK-Agenten kommunizieren. Eine Art der Kommunikation kann die Registrierung von M-Beans ins CMF des Agenten sein.

Zu beachten ist, daß einem Objekt, welches im CMF registriert wird, ein eindeutiger Name zugewiesen werden muß. Wird kein Name angegeben, so bildet JDMK einen Standard-Namen. Dieser umfaßt die Domäne, in welcher der Agent eingebettet ist, gefolgt von dem Klassenbezeichner der M-Bean-Klasse.
Es ist unbedingt zwischen der Instantiierung und Registrierung eines M-Beans zu unterscheiden. Zuallererst ist ein M-Bean als Objekt instantiiert. Erst dann wird es in das CMF des entsprechenden Agenten eingefügt (registriert). Es ist möglich, die Instanz eines M-Beans beizubehalten und dieses M-Bean je nach Bedarf zu registrieren bzw. zu de-registrieren. Damit

kann unter Umständen eine aufwendige mehrmalige Initialisierung dieses M-Beans, die bei gleichzeitiger Garbage Collection und erneutem Anlegen des Objekts notwendig wird, umgangen werden.

3.2 Management Beans (M-Beans)

M-Beans stellen eine zentrale Komponente innerhalb des JDMK dar. Sie entsprechen dem JavaBeans-Konzept[3] und werden für das Management von Managed-Objects verwendet. So kann ein M-Bean stellvertretend für einen Drucker innerhalb eines Netzwerks stehen. M-Beans werden innerhalb des JDMK aber auch für die Verarbeitung von Ereignissen verwendet. In Abbildung 3.1 ist ein M-Bean mit der Bezeichnung GaugeMonitorListener als Bestandteil des CMF dargestellt. Dieses M-Bean stellt einen Monitor für Properties anderer M-Beans dar und kann bei bestimmten Werten Ereignisse an Management-Applikationen senden.

3.2.1 Eigenschaften eines M-Bean

Attribute

Die Attribute eines M-Bean werden in der Designphase festgelegt. Sie umfassen die relevanten Daten, welche das Managed-Object, das durch das M-Bean repräsentiert werden soll, besitzt. In der Implementierung des Telephony Internet Servers wäre zum Beispiel ein M-Bean denkbar, welches die aktuelle Bandbreite des Netzes überprüft, um zu entscheiden, ob eine Videokonferenz durchgeführt werden kann.
In Anlehnung an das JavaBeans-Konzept können bei den M-Beans auch für die Attribute `get-` und `set-` Methoden definiert werden. Dies geschieht in Abhängigkeit der Zugriffsrechte auf die Attribute der Managed-Objects (bzw. der repräsentierenden M-Beans). Werden zum Beispiel innerhalb einer MIB Variablen als read-only gekennzeichnet, so ist selbstverständlich nur eine `get-` Methode zu verwenden.
Die Attribute eines M-Beans werden in 3 Kategorien aufgeteilt:

- <u>Einfache Attribute</u>
  Zu diesen zählen alle Attribute, welche über `get-` und `set-` Methoden zugreifbar sind (abhängig von den Zugriffsrechten).

- <u>Boolesche Attribute</u>
  Hier kann anstatt der gewöhnlichen `get-` Methode auch eine `is-` Methode verwendet werden, um die Lesbarkeit bei der Programmierung zu erhöhen.

- <u>Attribut-Felder</u>
  Es werden zudem unter JDMK indizierte Attribute ermöglicht. Falls solche Attribute verwendet werden, muß bei der `get-` und `set-` Methode der entsprechende Index angegeben werden.

[3] Das JavaBeans-Konzept beruht auf der Voraussetzung, Klassen zu erzeugen, welche einheitliche Zugriffsmuster auf deren Variablen (Attribute) besitzen. So werden für Zugriffsfunktionen auf Attribute Namenskonventionen festgelegt. Aufgrund der Namenskonventionen kann nun eine Applikation durch Verwendung der Reflection-API, welche ebenfalls von Sun für Java zur Verfügung gestellt wird, die von dem Bean unterstützten Attribute, Events und Methoden ermitteln.

Ereignisse und M-Beans

Das JavaBeans Konzept unterstützt die Verwendung von Ereignissen zur Kommunikation zwischen Beans. Auch dieses Konzept wurde in JDMK integriert. So können M-Beans nicht nur zum Management eines Managed-Objects genutzt, sondern auch zur Überwachung von anderen M-Beans verwendet werden. Diese "Listener"-Objekte können von Management-Applikationen instantiiert und im CMF des entsprechenden Agenten registriert werden. Dies geschieht durch `add`⟨EventListenerType⟩und `remove`⟨EventListenerType⟩Methoden, um schließlich den Eventlistener wieder entfernen zu können.
Die Events können als Unicast oder Multicast Ereignisse verschickt werden, um die Benachrichtigung mehrerer Management-Applikationen zu ermöglichen.
Die genaue Verwendung der Ereignisse und die Dienste, welche diesen Mechanismus verwenden, ist in Kapitel 3.7 erläutert.

Ausführbare Aktionen auf M-Beans

Neben den Methoden und Attributen, welche ein M-Bean zu bieten hat, kann es auch allgemeine Aktionen, bzw. Aktionen auf seine Attribute ausführen. Diese Aktionen besitzen das Präfix `perform` gefolgt von der gewünschten Aktion. Es können Parameter übergeben werden. Es ist jedoch zu beachten, daß bei Verwendung der Reflection-API zwar der Typ dieser Parameter ermittelt werden kann, nicht aber deren Bezeichner[4]

Lebenszyklus eines M-Bean

Der Lebenszyklus eines M-Beans beginnt mit dessen Instantiierung. Dabei wird ein Objekt der M-Bean-Klasse angelegt, welches jedoch noch nicht mit dem CMF irgendeines Agenten verbunden ist. Die Verbindung geschieht durch die sogenannte "Registrierung" des M-Beans. Die Registrierung erfolgt durch die `initcmf-` Methode . Sie ist eine Methode der Framework-Klasse. Ist das M-Bean registriert, kann es mit den dem Agenten zur Verfügung stehenden Adaptern von Clients (auch von anderen M-Beans desselben Frameworks) angesprochen werden. Soll das M-Bean nach der Verwendung wieder aus dem CMF eines Agenten entfernt werden, ist die Methode `deletecmf` zu verwenden. Wie bereits erwähnt, geschieht das Löschen der M-Bean Instanz separat, da das M-Bean unter Umständen als Objekt beibehalten werden soll. Während bei der Deregistrierung keine Argumente übergeben werden, sind bei der Registrierung durch `initcmf`[5] bestimmte Argumente notwendig:

- Das CMF, in welchem das M-Bean registriert werden soll,

[4] Dieses Problem stellt sich unter der von JDMK angebotenen Unterstützung zur Generierung von HTML-Zugriffsseiten für M-Beans. Dort werden auch die Aktionen berücksichtigt und es werden Eingabefelder für eventuell zu übergebende Parameter generiert. Jedoch werden diese Parameter nur mit dem Typ beschriftet, welches nicht unbedingt die Lesbarkeit solcher Eingabemasken erhöht.

[5] Die Aufrufsyntax lautet:

```
public void initCmf(Framework cmf,
                    ObjectName name,
                    boolean db,
                    ModificationList list
                    ) throws InstanceAlreadyExistException
```

- der Objektname, welcher das M-Bean bezeichnet,
- eine Angabe, ob das M-Bean persistent registriert werden soll und
- zusätzlich können mit list weitere Spezifikationen angegeben werden. Diese sogenannte "Modificationlist" beinhaltet Einträge (Tupel) in (Schlüssel,Wert)-Struktur und dienen dazu, beispielsweise bei Serialisierung des M-Beans dieses eindeutig zu kennzeichnen.

Die Persistenz von M-Beans wird im Kapitel 3.7.1 beschrieben, hier sei kurz erwähnt, daß es unter JDMK möglich ist, die Konfiguration eines CMF also auch seine M-Beans inklusive deren Status in einem File abzuspeichern, so daß bei einem erneuten Anlauf des Agenten sofort dessen vorhergehende Konfiguration wiederhergestellt werden kann.

Entfernter Zugriff auf M-Beans

Falls auf die Attribute eines M-Beans entfernt zugegriffen werden soll, so ist ein weiteres Schlüsselkonzept des JDMK relevant. Dieses Schlüsselkonzept stellt die sogenannten "ClientBeans" (C-Beans) dar. Eine ausführliche Beschreibung der C-Beans ist in Kapitel 3.3 zu finden. Für den Zugriff auf Attribute des M-Beans werden sie interessant, da sie nach der Instantiierung die Möglichkeit bieten auf die Attribute mittels der M-Bean Methoden zuzugreifen so als würde es sich um ein lokales Objekt der Manager-Applikation handeln.

3.2.2 Identifizierung eines M-Beans

Wie schon in Kapitel 3.1 erwähnt, kann die Registrierung eines M-Beans nur über einen Objektbezeichner geschehen. Dieser wird entweder direkt angegeben oder bei Fehlen eines Bezeichners durch das CMF bestimmt. Die Identifizierung geschieht durch Angabe des eindeutigen Objektbezeichners. Es ist aber auch möglich eine Reihe von M-Beans zu erhalten und dann das oder die Entsprechenden auszufiltern.
Die Objektbezeichner bestehen aus drei Elementen:

- <u>Domänenbezeichner</u>
  Der Kennzeichner der Domäne dient dazu, den Wirkungsbereich eines M-Bean einzugrenzen. Es kann der Host, auf welchem der Bean läuft oder aber eine größere Ansammlung von Rechnern damit bezeichnet werden. Wird mittels JDMK eine bestehende MIB verwaltet, so werden bei der Generierung durch MIBGEN (siehe Kap.3.8) unterschiedliche Domänen für die MIB, Gruppen sowie Tabellen verwendet.
- <u>Klassenbezeichner</u>
  Hier wird die Klasse, welche durch das M-Bean repräsentiert wird, angegeben. Die konkrete Klasse des M-Bean muß nicht notwendigerweise damit übereinstimmen, wie dies bei Dateiname und Klassennamen unter Java der Fall ist. Da damit die Namenswahl frei von der Struktur des M-Beans ist, kann das gleiche M-Bean unter verschiedenen Bezeichnern innerhalb eines CMF verwendet werden.
- <u>Suchschlüssel</u>
  Um aber die M-Beans nun genauer zu spezifizieren, werden diese in einem letzten Schritt durch einen Suchschlüssel eindeutig identifiziert. Der Suchschlüssel kann ein oder mehrere Attribute umfassen. So sind zum Beispiel in der Realisierung des Telephony Internet

Servers die Einträge in der Tabelle "DnsServerTable" durch den Suchschlüssel "DnsServerIndex" (Typ:INTEGER) eindeutig bestimmt. Es ist zu beachten, daß dieser Suchschlüssel den Index der SNMP-Tabelle darstellt. Allerdings muß der Suchschlüssel bzw. dessen Attribute nicht notwendigerweise auf Attribute des M-Bean beschränkt sein, es ist eine beliebige Wahl der Attribute möglich. Es ist zu beachten, daß der Suchschlüssel als optionaler Namensbestandteil angesehen wird. Falls nur eine Instanz der Klasse im zweiten Teil des Namens existiert, so ist der Suchschlüssel nicht unbedingt erforderlich.

3.2.3 Registrierung

Um ein M-Bean innerhalb eines CMF verwenden zu können, muß dieses registriert werden. Es ist hierbei zu beachten, daß es keinerlei Code innerhalb des M-Bean bedarf, um diesen innerhalb eines CMF zu registrieren. Lediglich ein Manager oder Agent hat den Code zur Registrierung bereitzustellen. Dies bietet die Möglichkeit, temporär Agenten zu erzeugen, um andere Agenten um M-Beans zu erweitern. Unter JDMK bieten sich zwei Möglichkeiten, M-Beans zu registrieren:

- Registrierung einer bestehenden Instanz eines M-Beans
  Diese Operation wird mit der CMF Methode `addObject` durchgeführt. Es ist zu beachten, daß es sich um eine nichtpersistente Registrierung eines M-Bean handelt. Das bedeutet, daß der M-Bean nur für die Laufzeit des Agenten existiert. Soll der M-Bean dauerhaft dem Agenten zugeordnet werden, so ist die Methode `addDBObject` zu verwenden. Dauerhaft bedeutet in diesem Zusammenhang, daß das M-Bean bei Verwendung des Repository-Service (siehe Kap. 3.7.1) mit dessen aktuellem Status gespeichert wird. Die Verwendung der beiden Methoden ist nachfolgend zu ersehen.

```
// I. nicht persistente Registrierung bestehender M-Beans
// Irgendeine M-Bean Instanz
mbeanText = new String("M-Bean (nichtpersistent)");

// Aufbau des Objektbezeichners
String domain = cmf.getDomain();
mbeanName = new ObjectName(domain + ":" + mbeanKlasse.ident=1");

//Registrierung des M-Bean im Core Management Framework
cmf.addobject(mbeanText, mbeanName);

// II. persistente Registrierung bestehender M-Beans
// Irgendeine M-Bean Instanz
mbeanText = new String("M-Bean (persistent)");

// Aufbau des Objektbezeichners
String domain = cmf.getDomain();
mbeanName = new ObjectName(domain + ":" + mbeanKlasse.ident=1");

//Registrierung des M-Bean im Core Management Framework
cmf.addDBobject(mbeanText, mbeanName);
```

- Instantiierung und Registrierung eines M-Beans
  Ist keine Instanz eines M-Bean existent, so ist eine andere CMF Methode zu verwenden. Auch hier werden aufgrund der Persistenz des M-Bean zwei Varianten angeboten:
  - `newObject` für nicht persistente und
  - `newDBObject` für persistente Registrierung.

Für die Instantiierung des M-Beans ermittelt das CMF die Java-Klasse des Beans und lädt sie mittels eines "Class Loaders" (vgl. Kapitel 3.7.2). Danach wird die Klasse von dem CMF instantiiert und registriert. Die Registrierung kann auf zwei unterschiedliche Arten geschehen:

– Verwendung der M-Bean eigenen initCmf Methode
  Hierbei muß darauf geachtet werden, daß die Klassendefinition des M-Bean eine initCmf-Methode zur Verfügung stellt. Wenn die Klasse nun instantiiert wurde, so führt das CMF unter Zuhilfenahme des Metadata-Service (siehe Kap. 3.7.1) die initCmf-Methode des M-Beans aus. Dies bietet die Möglichkeit, bei Instantiierung und Registrierung Aktionen auszuführen, welche sonst auf der Manager-Seite realisiert werden müßten.
– Verwendung des CMF
  Dieser Fall tritt ein, falls keine initCmf Methode innerhalb der M-Bean Klassendefinition festgelegt wurde. Die Registrierung kann bei dieser Variante jedoch nicht beeinflußt werden.

Bei Durchführung der Operation `addDBObject` (bzw. `newDBObject`) durch das CMF wird zusätzlich eine Methode `isPersistent` ausgeführt. Diese Methode überprüft, ob eine persistente Speicherung für den Agenten vorgesehen ist. Ist dies nicht der Fall, so wird eine ServiceNotFoundException ausgelöst.

3.2.4 Zugriffsmöglichkeiten auf M-Beans

M-Beans ermitteln

Es bestehen drei Möglichkeiten, auf ein M-Bean zuzugreifen:

- Über einen Objektbezeichner
  Der Objektbezeichner kann hierbei unterschiedlicher Granularität sein. Dies bedeutet, daß ein M-Bean, eine Reihe von M-Beans oder aber alle M-Beans einer Domäne angesprochen werden können, je nachdem wie genau der Bezeichner angegeben wird. Die genaue Aufteilung ist nachfolgend aufgeführt:

  – Angabe des vollständigen Objektnamens zur Identifizierung eines einzelnen spezifischen M-Beans.
  – Unter Umständen der Zugriff auf eine Menge von M-Beans, erreicht durch das Weglassen des M-Bean spezifischen Namensteils (nur Domäne und Klasse).
  – Eine Menge von M-Beans, welche dieselbe Domäne besitzen.
  – Keine Einschränkung, sondern alle M-Beans, welche sich innerhalb des CMF des jeweiligen Agenten befinden.

  Nachfolgend sind beispielhaft die entsprechenden Zugriffsmöglichkeiten aufgeführt.

```
// Beispiele fuer die unterschiedliche Art und Weise wie auf M-Beans
// zugegriffen werden kann.

// Ermittlung des M-Beans fuer
// Tabelleneintrag in Tabelle DnsServerTable
Vector mbeanClassMO = cmf.getObject(
```

```
            new ObjectName ("SBCS-TIS2-MIB
                            :tis20.impl.DnsServerEntry.DnsServerIndex=12
                            , null);

// Ermittlung aller M-Beans, welche die Tabelleneintraege
// in der Tabelle DnsServerTable darstellen
Vector mbeanClassMO = cmf.getObject(
            new ObjectName ("SBCS-TIS2-MIB:tis20.impl.DnsServerEntry")
                            , null);

// Ermittlung aller M-Beans eines TIS-Agenten, der in der Domaene
// SBCS-TIS2-MIB integriert ist.
Vector mbeanClassMO = cmf.getObject(
            new ObjectName ("SBCS-TIS2-MIB:"), null);
```

- Über einen Filtermechanismus, welcher durch die charakteristischen Werte (Attribute) das entsprechende M-Bean identifiziert.

- Über den Repository Dienst, in den die M-Beans eines Agenten registriert werden. Dieser ist in der Lage mittels einer Query bestimme M-Beans herauszufiltern. Ein Query-Ausdruck kann beispielsweise folgendermaßen aussehen:

```
// Ermittlung von M-Beans mittels eines Query-Ausdrucks
QueryExp exp = Query.match(Query.attr("PlatformMiscUserid"), Query.value("A*"));
```

Zugriff auf M-Bean-Eigenschaften

Um auf die Properties eines M-Bean zugreifen zu können, sind zwei Dinge zu beachten.

- Der Agent, welcher das M-Bean registriert hat und die Zugriffe auf Properties des M-Bean ermöglicht, hat einen Dienst, welcher mit "Metadata-Service" bezeichnet wird (siehe Kap. 3.7.1), bereitzustellen. Dieser Dienst ist notwendig, um die `get-` und `set-` Methoden zu interpretieren.

- Das M-Bean, auf welches zugegriffen werden soll, hat `get-` und `set-` Methoden zur Verfügung zu stellen (was durch das Bean-Konzept eigentlich schon gewährleistet sein sollte). Der Aufruf und die Verwendung dieser Methoden ist detailliert in Kapitel 3.2.1 dargestellt.
  Zusätzlich sei erwähnt, daß es möglich ist, bei Zugriff auf ein Attribut, einen Operator mit anzugeben. Dieser Operator kann den Wert zusätzlich verändern (z.B. `toUpperCase` bei setzen eines Stringwertes).

Das Lesen und Schreiben von Attributen kann folgendermaßen aussehen:

```
// Setzen und Lesen von M-Bean Attributen

// Lesen eines Attributs
Object semAnzahl = cmf.getValue("tum.org.student:personal.name=Knoechlein"  // M-Bean
                                ,"Semester");  //Property-Bezeichner

// Setzen eines Attributs
Object semAnzahl = cmf.setValue("tum.org.student:personal.name=Knoechlein"  // M-Bean
                                ,"Semester"  // Property-Bezeichner
                                ,1);  // Neuer Property-Wert
// Verwenden eines Operators
Object wohnort = cmf.getValue("tum.org.student:personal.name=Knoechlein"  // M-Bean
                              ,"Wohnort"  // Property-Bezeichner
                              ,"toUpperCase");  // Ausgabe in Grossbuchstaben
```

Relationships zwischen M-Beans

Es besteht die Möglichkeit, eine Relationship-Beziehung zwischen M-Beans herzustellen. Hierzu muß ein M-Bean instantiiert und registriert werden, welches das `com.sun.jaw.reference.common.RelationIf`- Interface implementiert. Mit Hilfe dieses M-Beans können nun Beziehungen zwischen zwei oder mehreren M-Beans realisiert werden.

3.3 ClientBeans (C-Beans)

Die Client-Beans (C-Beans) stellen die Repräsentanten für die M-Beans auf der Manager-Seite der Kommunikation dar. Änderungen, welche M-Beans betreffen, werden auf den C-Beans ausgeführt und zu den M-Beans weitergeleitet. Events, welche von M-Beans ausgelöst werden können, werden zu den C-Beans gesandt. Um C-Beans zu erzeugen bzw. deren Klassen zu instantiieren, müssen die Klassen durch den "MOGEN"[6] Compiler erzeugt werden. Als Argument wird dem Generator MOGEN die Klasse in Byte-Code übergeben, welche als Managed-Object von Management-Applikationen verwaltet werden soll.
Nach erfolgreicher Beendigung des MOGEN - Compiler-Laufs sind zwei `.java` Dateien erzeugt worden:

- Java Interface
  Diese Datei besitzt den Bezeichner ⟨M-Bean-Klasse⟩`MO.java`. Sie stellt die Interface-Klasse dar, welche die zugreifbaren Methoden des M-Bean durch den Manager oder andere Agenten angibt. Dies sind Methoden, welche innerhalb des M-Beans als `public` gekennzeichnet wurden.

- Java Stub
  Diese Datei wird mit ⟨M-Bean-Klasse⟩`MOStub.java` bezeichnet. Hier werden die Methoden implementiert, die im Java Interface spezifiziert worden sind. Hierbei ist zu beachten, daß alle Eigenschaften des M-Bean in das C-Bean übernommen werden (inklusive der Zugriffsrechte). Im Gegensatz hierzu werden nur die public - Methoden der M-Beans übernommen.

Es ist an dieser Stelle anzumerken, daß unter Umständen weitere Interface und Stub Dateien erzeugt werden können. Dies hängt davon ab, ob ein Listener-Konzept unterstützt wird. Die Eventbehandlung und die einhergehenden Listenerkonzepte werden in Kapitel 3.7.8 genauer erläutert.
Standardmäßig wird bei einem MOGEN Generatorlauf ein C-Bean (Stub und Interface) für die übergebene Klasse erzeugt. Ist es aber notwendig, eine Vererbungshierarchie zu berücksichtigen, so bietet sich die Möglichkeit, statt der einzelnen C-Beans, welche die Klassen der Vererbungshierarchie darstellen, alle Klassen dieser Hierarchie in ein C-Bean zusammenzufassen. Die Funktionsweise ist in Abbildung 3.2 zu ersehen.
Der MO-Generator-Lauf kann durch die Angabe von verschiedenen Parametern benutzergerecht angepaßt werden. Die Switches und deren Möglichkeiten sind in Tabelle 3.1 aufgeführt.
Zusätzlich zu den oben genannten Eigenschaften und Methoden von M-Beans werden auch Methoden in C-Beans verwendet, welche kein Äquivalent im zugehörigen M-Bean besitzen. Diese Methoden dienen zur Steigerung der Performance. Es werden vier hauptsächliche Bereiche angesprochen.

[6] MO = Managed Object, ein Repräsentant eines Objektes des Agenten auf der Client-Seite.

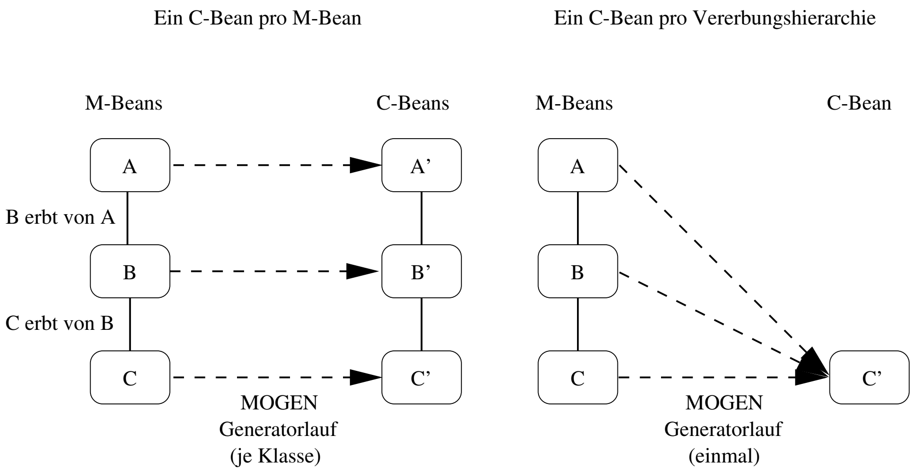

Abbildung 3.2: Vererbung und C-Beans

- Reduzierung des Netzverkehrs
  Eine Möglichkeit ist die Reduzierung des Netzverkehrs zwischen C- und M-Bean. Dies kann dadurch geschehen, daß mehrere Operationen auf ein M-Bean, welche immer in funktionaler Reihenfolge ausgeführt werden, in eine C-Bean Methode zusammengefasst werden können.

- Garbage Collection
  Dies kann durch Hinzunahme einer Methode geschehen, welche das C-Bean dann löscht, wenn das zugehörige M-Bean nicht mehr existiert.

- Sicherstellung der Aktualität
  Da sich der Wert der Properties eines M-Beans ohne die Mitwirkung des C-Bean verändern können, werden in den Stub-Klassen, welche durch MOGEN generiert werden, Methoden vorgesehen, um die Werte der C-Bean-Properties wieder anzupassen. Ein Auszug aus einem Stub ist nachfolgend dargestellt:

```
// Updaten der ScopeId und EnableDnsForWindows-Properties:

  public void handlePropertyList(PropertyList list) {
    if ((list == null) || (list.isEmpty()))
      return;
    super.handlePropertyList(list);
    for(Enumeration e = list.elements(); e.hasMoreElements();) {
      Property  property = (Property)e.nextElement();
      if (property.getProperty().equals("ScopeId"))
        ScopeId= (String) property.getValue();
      if (property.getProperty().equals("EnableDnsForWindows"))
        EnableDnsForWindows= (EnumEnableDnsForWindows) property.getValue();
      ...
    }
  }
```

- Zugriff auf das CMF
  Der Zugriff auf das CMF kann nicht nur auf die M-Beans des entsprechenden Agenten,

Option	Wirkungsweise
-d ⟨Verzeichnispfad⟩	Angabe des Verzeichnisses, in dem der generierte Code abgelegt werden soll. Falls an dieser Stelle nichts angegeben wird, wird das Verzeichnis der Package gewählt.
-ro	Damit ist es möglich, ein C-Bean als read-only zu generieren. Der Zugriff auf M-Beans, welche durch solche C-Beans repräsentiert werden, liefert bei einem `set`-Zugriff eine Exception.
-f	Ist die Klasse, für welche ein C-Bean erzeugt werden soll, von anderen Klassen abgeleitet, so werden bei Angabe dieses Switches alle Attribute und Methoden dieser Klassen in ein C-Bean generiert. Andernfalls werden separate Stubs und Skeletons erzeugt.
-l ⟨Klassenbezeichner⟩	Mit dieser Option, welche nur mit "-f" verwendet wird, kann die Zahl der "zusammengefaßten" Klassen limitiert werden. (vgl. 3.3). Wenn Klassen nicht hinzugenommen werden sollen, so müssen sie hier angegeben werden. Es wird keine Vererbung in das C-Bean propagiert.
-li ⟨Klassenbezeichner⟩	Funktionsweise wie mit "-l", jedoch wird hier die Vererbung berücksichtigt und entsprechende Einträge in das C-Bean durchgeführt.
-nlmo	Der MO-Generator erzeugt Stubs und Interfaces für Listener-Objekte, welche in dem M-Bean verwendet werden. Soll kein Interface erzeugt werden, so ist diese Option anzugeben.
-nlas	Wie "-nlmo", nur wird bei Angabe dieses Schalters kein Stub generiert.
-nl	Kombination der Schalter "-nlmo" und "-nlas".
-tp ⟨Package-Bezeichner⟩	Hier kann ein Package-Bezeichner für die generierten Stubs und Interfaces angegeben werden.
-np	Mit diesem Schalter wird die Generierung von sogenannten "Actions", die "perform"-Methoden des M-Beans, unterdrückt.
-ne	Unterdrückung der Codegenerierung für Events.
-nc	Unterdrückung der Codegenerierung für den "Cascading-Service"
-classpath ⟨Pfadangabe⟩	Angabe eines oder mehrerer durch ";" getrennter Verzeichnisse, in denen die erforderlichen Klassen gefunden werden können. Dies ist zu verwenden, wenn der vorhandene Pfad nicht ausreicht oder nur Teile davon benutzt werden wollen.

Tabelle 3.1: Optionen für MO-Generator "MOGEN"

welche integriert sind, geschehen, sondern es können auch bestimmte Dienste des CMF durch C-Beans genutzt werden (siehe Kapitel 3.7). Dies bedeutet, daß das Konzept der C-Beans nicht nur auf die von dem Entwickler bereitgestellten M-Beans angewendet werden kann, sondern ebenso auf M-Beans, welche vom Core Management Framework zur Verfügung gestellt werden. Dies können beispielsweise die `AdaptorClient`-Objekte sein, welche in 3.6 beschrieben werden. Ebenso fallen unter diese Kategorie sämtliche M-Beans der von JDMK zur Verfügung gestellten Dienste, welche in Kapitel 3.7 dargestellt werden.

3.4 Adapter

Das Adapterkonzept des JDMK ist eine umfangreiche Kommunikationsmöglichkeit zwischen Agenten und Managern. Die Adapter, welche von JDMK zur Verfügung gestellt werden, umfassen die Protokolle RMI, IIOP, HTTP und HTML. Es ist nur notwendig, das entsprechende Adapterobjekt zu instantiieren, im weiteren Verlauf ist der Adapter vollkommen protokollunspezifisch zu verwenden.

3.4.1 Eigenschaften

Ein Adapter stellt selbst ein M-Bean dar. Er muß instantiiert und registriert werden und ebenfalls einen eindeutigen Objektbezeichner besitzen. Es ist notwendig, daß eine Management-Applikation mindestens einen unterstützten Adapter mit einem beliebigen Protokoll besitzt, um auf den Agenten zugreifen zu können. Es ist zu beachten, daß auf der Client- oder Managerseite entsprechend dem C-Bean-Konzept ein sogenannter "AdaptorClient" instantiiert werden muß, um mit einem entsprechenden Adapter auf der Agentenseite zu kommunizieren[7]. Es werden verschiedene Protokolle unterstützt.

3.4.2 Protokolle

- RMI
  Der RMI Adapter kommuniziert über das Java Remote Method Invocation System. Er benutzt als Standard den Port 1099, es kann jedoch optional ein anderer Port ausgewählt werden. Er wird durch die Klasse `com.sun.jaw.impl.adaptor.rmi.AdaptorServer` repräsentiert.

- HTTP / TCP
  Es wird mit diesem Adapter ermöglicht, HTTP über eine TCP/IP Verbindung als Kommunikation zwischen den Agenten (bzw. zwischen Manager und Agent) zu verwenden. Die Kommunikation über Proxy-Server ist damit ebenso möglich, zusätzlich wird eine Login/Passwort Authentifizierung unterstützt. Als Standard wird der Port 8081 verwendet. Ebenso wie bei RMI kann auch hier ein anderer Port verwendet werden.

- HTTP / UDP
  Hier wird UDP als Protokoll der Transportschicht verwendet. Es werden ebenso Login/Passwort Mechanismen unterstützt und der Standard Port ist 8083.

- HTTP / SSL
  Dieser Adapter stellt eine Realisierung der Kommunikation über das HTTPS-Protokoll dar. Die Authentifizierung geschieht nach CRAM-MD5, wie sie in [KCK 97] beschrieben

[7] Eine Ausnahme bilden hierbei der HTML- und der SNMP-Adapter. Diese Adapter werden ohne AdaptorClient angesprochen.

ist. Dieser Adapter ermöglicht es Management-Applikationen über Proxies auf JDMK-Agenten zuzugreifen. Der Standardport, welcher bei Instantiierung des Adapters verwendet wird, ist 8084, es kann jedoch ein anderer Port frei gewählt werden. Bei der Implementierung der Sicherheitsmaßnahmen bei Verwendung des Adapters muß beachtet werden, daß sie zu der von Sun entwickelten SSL Standard Extension API, welche ab JDK 1.1 zur Verfügung steht, konform ist.

- HTML
  Der HTML-Adapter stellt eigentlich einen HTML-Server dar, welcher über HTTP die Kommunikation mit Web-Browsern ermöglicht. Wenn nun ein HTML-Adapter instantiiert wird, wird ein TCP/IP-Socket aufgebaut und auf Anfragen gewartet. Login und Passwort wie bei HTTP vorgesehen, der Standard Port ist hier 8082.
  Bei Zugriff über diesen Adapter werden zur Laufzeit des Agenten dynamisch HTML-Seiten generiert, welche die M-Beans des Agenten darstellen. Durch diese Darstellung kann auf die M-Beans eines Agenten via URL zugegriffen werden. Eine genauere Beschreibung dieser generierten HTML-Seiten ist in Kapitel 6.7.1 zu finden.

- IIOP
  Der IIOP - Adapter ermöglicht es den JDMK-Agenten, Informationen und Managementaktionen über eine CORBA-Schnittstelle[8] auszutauschen. Ein CORBA Client kann so auf die M-Beans eines Agenten zugreifen.
  Um nun den Adapter verwenden zu können, muß ein COS-Naming Service[9] zur Verfügung gestellt werden. Es können beliebige COS-Naming-Service-Implementierungen verwendet werden. Bei der Realisierung des Prototyps wird der JavaIDL[10] Name Server, welcher eine Implementierung eines COS-Naming Service darstellt, benutzt.
  Die Verwendung von IIOP und die Möglichkeiten, JDMK innerhalb einer CORBA- Umgebung zu integrieren, wird in Kapitel 5.2 genauer eingegangen. Dort wird ebenfalls erläutert, wie die Registrierung eines IIOP-Adapters innerhalb eines COS-Naming Service durchgeführt wird.

- SNMP
  Auf die Verwendung von SNMP (SNMPv1/v2) wird in Kapitel 3.8 genauer eingegangen, hier sei nur erwähnt, daß mittels dieses Adapters auch SNMP-Agenten Zugriff auf M-Beans eines Agenten bekommen können. Auch kann mittels dieses Adapters ein MIB-Browser die Properties der M-Beans lesen und je nach Zugriffsrechten auch schreiben. Der Zugriffsschutz geschieht in diesem Fall durch ein ACL-File, welches in einem spezifischen Verzeichnis stehen muß und alle zugriffsberechtigten Rechner sowie deren spezifische Rechte beinhaltet.

---

[8] CORBA wird in der Version 2.0 vorausgesetzt. Kompatibilität dieses Adapters mit der Version 2.2 ist nicht sicher und konnte auch während der Testphase des Adapters nicht ermittelt werden.

[9] Der COS (Common Object Service) Naming Service ist in seiner Funktion mit Verzeichnissen auf Dateisystemen vergleichbar. Die Verzeichnisse entsprechen hier sogenannten "Kontexten", in welche Referenzen auf zur Verfügung stehende Objekte eingetragen werden. Ein Server kann an dieser Stelle seine Methoden und Objekte zur Verfügung stellen, Clients können diese verwenden.

[10] JavaIDL ist eine von Sun zur Verfügung gestellte Package von Java-Klassen, welche einen COS-Naming-Service implementieren. der Aufruf geschieht durch "nameserv" und der optionalen Angabe eines Ports mit "-ORBInitialPort ⟨Port⟩" Wenn keine Angabe gemacht wird, so wird der Standardport 900 verwendet.

3.4.3 Dynamisches Hinzufügen und Entfernen

Aufgrund des M-Bean Konzeptes können Adapter, welche im eigentlichen Sinne auch M-Beans darstellen, dynamisch zu Agenten hinzugefügt und entfernt werden. Es können auch mehrere Adapter desselben Protokolls auf verschiedenen Ports Verwendung finden. Ebenso kann eine Client- oder Managerapplikation ein oder mehrere Adapter zur Kommunikation mit dem JDMK-Agenten verwenden.

3.4.4 Zugriffsmöglichkeiten

Zusammenfassend können die Zugriffsmöglichkeiten auf vier Punkte eingeschränkt werden

- Lesender und ggf. schreibender Zugriff auf Eigenschaften (z.B. Attribute) eines M-Bean.
- Entferntes Ausführen von Methoden, welche von M-Beans auf dem Rechner auf den zugegriffen wird, implementiert werden.
- Empfangen von Events über sogenannte Eventlistener-Objekte, welche von M-Beans bei Eintreten einer bestimmten Bedingung ausgelöst werden. Auf das Eventmodell und die damit verbundenen Möglichkeiten wird in Kapitel 3.7.8 genauer eingegangen.
- Ein entferntes Instantiieren und Registrieren von M-Beans in Agenten (siehe Kapitel 3.2).

3.4.5 Zugriffsschutz

Wie bereits angedeutet, unterstützen die verschiedenen Protokolle unterschiedliche Sicherheitskonzepte, welche den Zugriffsschutz auf M-Beans gewährleisten. An dieser Stelle werden diese Konzepte zusammengefaßt:

- HTTP (UDP / TCP / SSL)
  Diese Adapter bedienen sich der Passwort/Login-Authentifizierung, wobei der Agent dabei eine Liste von Passwort/Login-Kombinationen verwaltet. Sind keine Einträge zu finden (Die Liste ist null), dann kann jede Applikation auf den Agenten über diese Adapter zugreifen. Die Authentifizierung richtet sich nach dem CRAM-MD5[11]- Schema, welches in [RDK+ 97] und [KCK 97] beschrieben ist.

- HTML
  Dieser Adapter verwendet eine Passwort/Login-Authentifizierung, welche im "Basic Authentification Scheme" in [BLFF 96] beschrieben ist. Der Sicherheitswert, den eine solche Authentifizierung bietet ist als sehr gering einzustufen, da die Logins und Passwörter im Klartext übertragen werden.

- SNMP
  Um den Zugriffsschutz über SNMP zu realisieren, wird in JDMK das Konzept einer ACL (Access Control List) verwendet. Diese Liste, welche in ihrer Struktur in 3.5 beschrieben ist, kann als Datei und als Objekt einer Klasse realisiert sein. Falls die Realisierung mittels einer Klasse gewählt wird, muß diese Klasse von `com.sun.jaw.reference.agent.services.IPAclSrvIf` abgeleitet sein.

[11] CRAM-MD5 : Challenge-Response Authentification Mechanism, using MD-5

- RMI/IIOP
  Für diese Adapter wird unter JDMK kein explizites Sicherheitskonzept angeboten. Es muß in diesem Fall vom Entwickler sichergestellt werden, daß die notwendigen Sicherheitskriterien erfüllt werden.

3.5 Zugriffsschutz über ACL

Diese Sicherheitsschranke, welche als Datei und als Klassenobjekt dem Agenten zur Verfügung stehen kann[12], stellt eine "Access Control List", in welcher die Rechner, welche auf den Agenten Zugriff erhalten sollen, eingetragen sind. Es wird zwischen "read-only" und "read-write" Zugriff unterschieden. Ebenso können Rechner eingetragen werden, an welche auftretende Traps gesandt werden. Die ACL-Datei wird im Zusammenhang mit dem SNMP-Adapter verwendet. Die Datei muß auf dem Rechner vorhanden sein, auf dem der JDMK-Agent gestartet wird. Die Datei muß mit "jaw.acl" bezeichnet werden und im Konfigurationsverzeichnis[13] abgelegt sein. Gegebenenfalls ist die CLASSPATH Environmentvariable des NT-Systems entsprechend zu setzen, um auf die Klassen des JDMK zugreifen zu können. Es ist zu beachten, daß eine Datei für alle Agenten gilt, welche einen SNMP Adapter verwenden und auf diesem Rechner laufen. Mit dieser Datei kann zusätzlich mit den entsprechenden Zugriffsrechten die Zahl der SNMP-Manager, welche auf die Agenten zugreifen können, beschränkt werden.
Eine ACL-Datei kann in zwei Teilbereiche gegliedert werden, gekennzeichnet durch Gruppen von Variablen-Bezeichnern:

- acl
  Die Gruppen von Variablen-Bezeichnern werden in verschiedenen Listen gegliedert, welche durch folgende Bezeichner beschrieben werden:
  - communities
    Hier werden verschiedene Zugriffsrechte festgelegt (private, public)
  - access
    Hier werden die Zugriffsrechte (read-only, read-write) der Manager (bzw. Rechner) bestimmt, welche unter "managers" aufgeführt sind.
  - managers
    Hier werden die Manager festgelegt, welche mit den oben genannten Rechten auf die unter "communities" bezeichneten Variablen zugreifen können. Die Manager können als Alias, IP-Adresse oder als Subnet(Intranet)-Adresse angegeben werden.
- trap
  An dieser Stelle werden die Gruppen festgelegt, an welche die Agenten gegebenenfalls SNMP-Traps senden. Auch hier werden Listen verwendet, welche bestimme Bezeichner vorangestellt bekommen:

[12] Im weiteren wird auf die Datei, welche die ACL darstellt, vorgestellt. Der Aufbau der Klasse und die Art der Einträge sind zwischen diesen Varianten identisch.

[13] Die Verzeichnisse für die Konfigurationsdateien sind

- Solaris : /etc/opt/SUNWconn/jaw/conf/
- Windows NT : C:\Programme\SUNWconn\jaw\etc\conf\

- trap-community
  Spezifikation des SNMP Community Strings.
- hosts
  Angabe der Hosts, welche mit dem Community String versehene Traps erhalten. Hier kann ein Alias oder eine IP-Adresse stehen.

Nachstehend ist ein Beispiel einer ACL Datei zu sehen. Zusätzlich ist in Anhang A die Syntax zur Erstellung einer ACL-Datei zu finden.

```
acl = {
  {
    communities = public, private
    access = read-only
    managers = Majestix, Methusalix
  }
  {
    communities = students
    access = read-write
    managers = 123.123.123.123
  }
}
```

3.6 Adaptor-Clients

Wie bereits erwähnt, entsprechen die Adaptor-Clients dem C-Beans Konzept des JDMK. Wenn eine Management-Applikation auf einen Agenten über einen Adapter zugreifen möchte, muß er sich zunächst mit diesem verbinden. Die Verbindung geschieht dadurch, daß auf Agentenseite ein AdaptorClient-Objekt instantiiert wird, welches die spezifischen Eigenschaften (Port, Rechner, Protokoll) des Adapters enthält, zu welchem verbunden werden soll. So kann beispielsweise mittels dieses AdaptorClient-Objekts ein M-Bean im Agenten instantiiert bzw. registriert oder de-registriert bzw. gelöscht werden. Falls ein AdaptorClient-Objekt verwendet wird, bedeutet dies implizit, daß hierbei eine Java-Implementierung vorausgesetzt wird. Falls mit einem Manager, der in einer anderen Programmiersprache implementiert wurde, auf den Agenten zugegriffen wird, muß das AdaptorClient-Objekt unter Zuhilfenahme des Java Native Interface (JNI), welches in Kapitel 6.12 beschrieben wird, mit dem C++-Programmcode verbunden werden. Im Falle des SNMP-Adapters kann der Zugriff durch einen MIB-Browser geschehen, welcher keinen AdaptorClient verwendet. Es sei darauf hingewiesen, daß bei Zugriff über den SNMP- und HTML-Adapter kein AdaptorClient verwendet wird. In Tabelle 3.2 ist aufgeführt, welche Adapter verwendet werden und über welche Zugriffsmöglichkeiten diese verfügen. Bei Verwendung des AdaptorClient ist in erster Linie keine besondere Angabe des Protokolls bei der Instantiierung notwendig, es genügt, das Interface `adaptorMO` zu verwenden. Das bedeutet, daß Klassen, welche dieses Interface implementieren mit jedem AdaptorClient kommunizieren können. Die Auswahl des Protokolls erfolgt bei Aufruf der `connect`-Methode, welche die Verbindung zu dem Adapter auf der Agentenseite realisiert. Auf das `adaptorMO` Interface gibt es zwei Zugriffsebenen:

- Low Level
  Über diese Ebene kann mit M-Beans über deren Objektbezeichner kommuniziert werden. Auf dieser Zugriffsebene muß für jedes M-Bean Property ein Aufruf mit `getValue` bzw. `setValue`-Methoden durchgeführt werden. Diese Methode ist nicht sehr transparent, und

Adaptor	AdaptorClient	Andere
RMI	+	RMI-(Client/Manager)
IIOP	+	IIOP-(Client/Manager)
HTTP	+	HTTP-(Client/Manager)
HTTPS	+	HTTPS-(Client/Manager)
SNMP	-	SNMP-MIB-Browser, SNMP-Manager
HTML	-	Web-Browser

Tabelle 3.2: Verwendung des AdaptorClient

eignet sich für kleinere Änderungen an M-Beans, wenn kein C-Bean instantiiert werden muß.

- High Level
  Ist der Zugriff auf ein M-Bean umfangreicher, bietet sich die High Level Zugriffsebene an. Diese Ebene basiert auf der Instantiierung eines C-Beans. Dies ermöglicht den Zugriff auf die M-Bean-Properties wie durch das M-Bean Objekt selbst. Durch die gewonnene Transparenz werden die Zugriffe auf M-Bean-Properties erleichtert.

Nachfolgend sind zwei Beispiele für die Verwendung der verschiedenen Zugriffsebenen aufgeführt:

```
// Low Level Interface
...
ObjectName objName
    = new ObjectName("SBCS-TIS2-MIB:tis20.impl.NetworkParmsImplMO");
String ComputerName
    = (String) adaptor.getValue(objName, "ComputerName");
...

// High Level Interface
...
String GkStatisticsImplClass = "GkStatisticsImpl";
ObjectName GkStatisticsImplName
    = new ObjectName(domain + ":tis20.impl.GkStatisticsImpl");
tis20.impl.NetworkParmsImplMO NetworkParmsImpl
    = (tis20.impl.NetworkParmsImplMO)
      ((Vector)adaptor.getObject(NetworkParmsImplName, null)).firstElement();
String ComputerName = NetworkParmsImpl.getComputerName();
...
```

Die unter Kapitel 3.4 beschriebenen Protokolle können durch AdaptorClients realisiert werden, indem eine Instanz kreiert wird. Das Zusammenspiel zwischen AdaptorClient, Adapter und M-Beans ist in Abbildung 3.3 dargestellt.

3.6.1 Initialisierung

Die Initialisierung eines AdaptorClient geschieht in zwei Schritten:

- Anlegen eines AdaptorMO Objekts auf der Managerseite
  Der Konstruktor wird ohne Parameter aufgerufen und ist als "public" gekennzeichnet. Diese Variante der Realisierung eines AdaptorClients bietet den Vorteil, daß zur Entwicklungszeit des Java-Clients keine Festlegung auf ein bestimmtes Protokoll zu geschehen

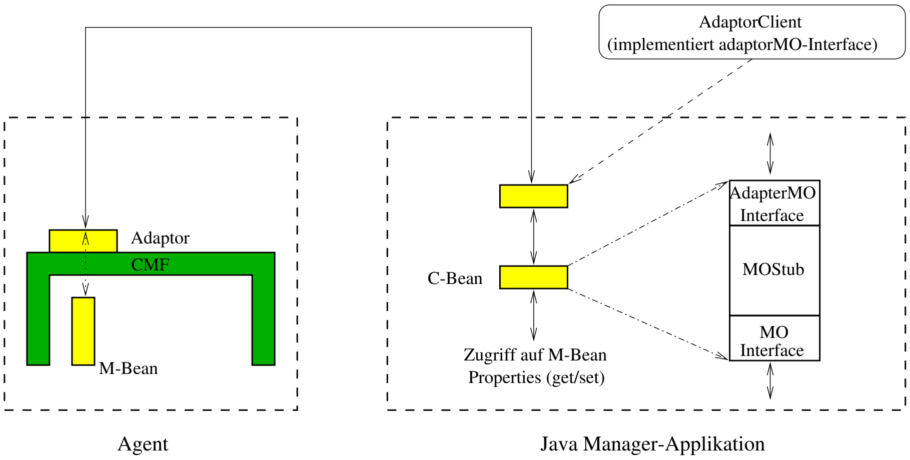

Abbildung 3.3: Zugriff über AdaptorClient

braucht. Die Angabe des jeweiligen Protokolls und damit die Auswahl des entsprechenden AdaptorClients geschieht zur Laufzeit über einen String, welcher beispielsweise als Kommandozeilenargument übergeben werden kann. Nachfolgend ist ein Ausschnitt aus einer Client-Applikation aufgezeigt, in welcher diese Art der Verwendung demonstriert wird.

```
...
// String runtimeSelectedString kann beispielsweise
// com.sun.jaw.impl.adaptor.rmi.AdaptorClient
// sein.
Class adaptorClient = Class.forName(runtimeSelectedString);
AdaptorMO MOFactory
    = (AdaptorMO) adaptorClient.newInstance();
```

- Das angelegte Object mit dem gewünschten Agenten verbinden
  Die Verbindung zu dem Agenten wird durch die Methode `connect` erreicht. In dem Methodenaufruf müssen folgende Parameter spezifiziert werden:
  - Die Authentifizierung, welche eine Login-Kennung und ein Passwort erfordert, ist bei dem HTTP-Adapter (UDP/SSL) anzugeben. Bei den sonstigen Adaptern (RMI, IIOP) ist "NULL" anzugeben..
  - Hostname der Maschine, auf welcher der Agent läuft.
  - Portnummer, über welche die Ereignisse gesendet werden (bzw. auf welcher die entsprechende Maschine wartet).
  - Logischer Bezeichner, z.B. der Service-Name.

Nachfolgend ist ein Beispiel für die Initialisierung eines Adapter Clients mit nachfolgendem Verbindungsaufbau zu sehen.

```
// Aufbau eines AdaptorClient mit HTTP-Protokoll und
// Authentifizierung
```

```
...
import com.sun.jaw.impl.adaptor.http.AdaptorClient;
import com.sun.jaw.impl.adaptor.comm.*;
...
AdaptorClient adaptor
       = new AdaptorClient();

// Authentifizierzung festlegen
AuthInfo security = new AuthInfo("root", "RootPassword");

// Verbindung mit HTTP-Adaptor auf JDMK-Agenten-Seite
try {
   adaptor.connect(security, JDMKAgentHost, JDMKAgentPort, ServiceName.ART_HTTP);
} catch (UnauthorizedSecurityException e) {
   // Process
}
```

3.6.2 Voraussetzungen

Um einen AdaptorClient zu instantiieren und Operationen darauf ausführen zu können, müssen zwei Dienste auf der jeweiligen Maschine zur Verfügung stehen.

- Abbildungsdienst (Mapping Service)
  Dieser Dienst wird benötigt, um die Zusammenhänge zwischen C- und M-Beans abzubilden und die jeweiligen Klassen zu liefern. (Mapping der M-Bean Methoden auf Stubs und Skeletons, welche durch MOGEN generiert wurden)

- Class Loader
  Sämtliche Klassen, welche für einen Zugriff auf M-Bean-Properties über C-Beans benötigt werden, müssen mittels eines ClassLoaders geladen werden. Sind sämtlche Klassen physikalisch auf dem lokalen Rechner vorhanden, auf dem der AdaptorClient (und damit auch die Management-Applikation) läuft, ist eine spezielle Angabe des ClassLoaders nicht notwendig. (Es wird in diesem Fall der Standard-ClassLoader verwendet). Werden jedoch Klassen auf anderen Rechnern benötigt, so muß ein ClassLoader instantiiert werden, welcher die Verbindung zu dem entsprechenden ClassServer realisiert. In diesem Fall ist in die Management-Applikation die `com.sun.jaw.impl.agent.services.loader.rmi.NetClassLoader`-Klasse zu importieren. Die ClassServer (bzw. LibraryServer) stellen eigenständige Java-Applikationen dar und werden in Kapitel 3.7.4 genauer erläutert. In Abbildung 3.4 wird der Zusammenhang zwischen diesen Komponenten dargestellt.

3.6.3 Zugriffsmöglichkeiten

Nachdem der AdaptorClient instantiiert und registriert wurde, können verschiedene Operationen darauf ausgeführt werden.

- Zugriff auf M-Beans
  Es ist möglich, einen oder mehrere M-Beans eines Agenten in einen Vektor von Managed Objects zu laden. Durch einen Filtermechanismus über die Eigenschaften der M-Beans kann die Zahl und Art beschränkt werden.

- Zugriff auf M-Bean Eigenschaften
  Wie in Kapitel 3.2.1 beschrieben, kann der Zugriff auf M-Bean-Eigenschaften lesend und

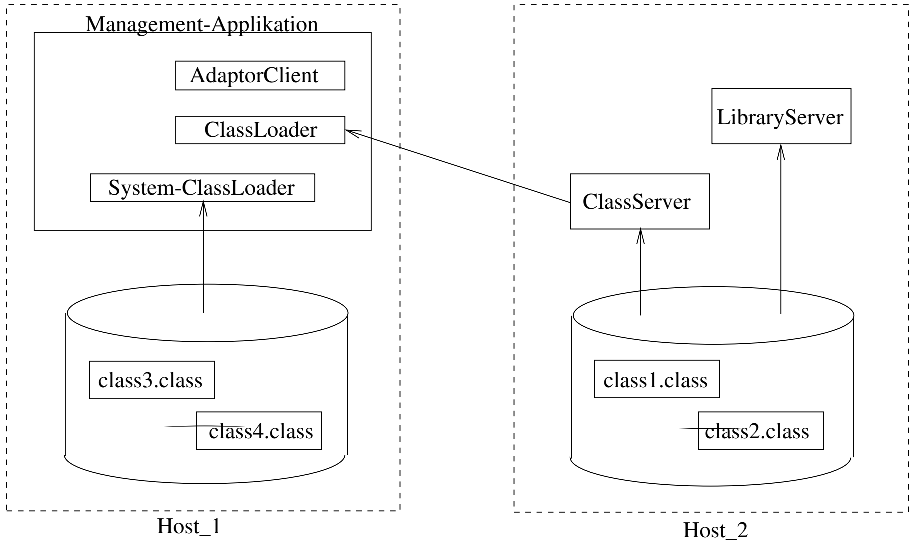

Abbildung 3.4: Zusammenhang Class/Library-Loader und Class/Library-Server

schreibend erfolgen. In beiden Fällen kann das über die in Kapitel 3.6 beschriebenen Low- und High-Level-Interfaces geschehen.
Zusätzlich ist es unter JDMK möglich, Properties zu cachen. Dadurch kann zusätzlich Netzbelastung reduziert werden. Es werden dabei zuerst alle Werte des M-Bean gelesen und im Cache zwischengespeichert. Bei Änderung (bzw. Setzen eines Wertes) können mehrere Operationen mit dem Befehl `setGroupOper(BOOLEAN)` zusammengefaßt werden. Nachfolgend ist eine solche Bündelung zu sehen.

```
// Lesen aller Attribute des M-bean
cbean.readAll();

// Zuerst Aenderung der Werte, dann
// einen Request an den Agenten schicken
cbean.setGroupOper(true);
String state = cbean.getState();
Integer counter = cbean.getNbChanges();

// Request absetzen
cbean.setGroupOper(false);
```

- <u>Instantiierung von M-Beans</u>
  Eine Instantiierung eines M-Bean kann auch entfernt über einen AdaptorClient geschehen. Falls ein anderer als der Standard Class Loader benötigt wird, so kann dieser spezifiziert werden.

- <u>Registrierung eines M-Beans</u>
  Ist ein M-Bean Objekt instantiiert, kann es dem Agenten hinzugefügt werden. Dieses Hinzufügen wird als Registrierung bezeichnet. Ist ein M-Bean registriert, kann darauf über die Adapter des Agenten zugegriffen werden. Auch kann das M-Bean über die

Dienste, welche von JDMK zur Verfügung gestellt werden, angesprochen werden. Es kann diese auch selbst nutzen.

- De-Registrieren eines M-Beans
  Wird ein M-Bean nicht länger benötigt und soll es aus dem Agenten wieder entfernt werden, so ist es zu de-registrieren. (falls es registriert wurde). Nach dieser De-Registrierung ist das M-Bean weiterhin als Java-Objekt vorhanden.

- Entfernen eines M-Beans
  Ist das M-Bean ggf. de-registriert worden und soll das Objekt nun gelöscht werden, so geschieht dies explizit durch Verwendung der Methode `deleteMO`. Die Unterscheidung zwischen De-Registrierung und dem Löschen eines M-Beans wird getroffen, da es vielleicht zu einem späteren Zeitpunkt sinnvoll erscheint, das Objekt wieder zu registrieren. Ist dies der Fall, so muß dieses Objekt nicht noch einmal instantiiert werden.

3.7 Dienste

3.7.1 Basisdienste

Die Basisdienste stellen die Grundvoraussetzung für die Implementierung eines lauffähigen Agenten oder Managers dar. Es wird keine eigenständige Basisklasse zur Verfügung gestellt, sondern eine Reihe von eigenständigen Klassen, welche als Summe die Basisdienste darstellen. Dies hat den Vorteil, daß diese Einzelkomponenten später leichter ersetzt, angepaßt oder erweitert werden können. Die Basisdienste werden nachfolgend aufgeführt:

- Initialisierung des Core Management Framework
  Jeder Agent, welcher unter JDMK entwickelt wird, muß als Grundlage eine Instanz des Core Management Framework besitzen. In diesem Rahmen werden alle M-Beans des Agenten integriert. Die Instantiierung kann auf drei verschiedene Arten erfolgen:
  - Standard-Konstruktor ohne weitere Argumente.
  - Konstruktor unter Angabe der Domäne, in welcher der Agent aktiv sein wird.
  - Konstruktor mit Angabe der Domäne und einem spezifischen Repository Service. Diese Variante wird bei persistenter Speicherung der M-Beans verwendet.

- Der Repository Service
  Dieser Dienst dient dazu, die M-Beans im CMF zu registrieren. Diese Beans werden mit dem Objektnamen als Java-Objekte gespeichert. Sind die Beans erst registriert, so kann auf sie über den Objektnamen entsprechend ihrer Properties zugegriffen werden. Es ist nur eine Instanz eines Repository Service pro JDMK-Agent möglich, d.h. auch die Art und Weise wie der Agent seine M-Beans behandelt ist auf eine Richtung festgelegt. Es werden drei Arten des Repository Service unterschieden:
  - Speicherung der M-Beans im Arbeitsspeicher (Volatile Repository)
    Dies ist die einfachste Form des Repository Services. Die M-Beans, welche in diesem Repository registriert werden, existieren nur zur Laufzeit des Agenten. Falls der Prozeß des Agenten beendet wird, sind auch die M-Beans und deren aktueller Zustand verloren.

- Dauerhafte Speicherung der M-Beans und deren derzeitigen Zustand. (Persistent Repository)
  Abhilfe der temporären Lösung des Volatile Repository bietet das Persistent Repository. Wie der Begriff der Persistenz bereits aussagt, können hier die M-Beans und deren Zustand bei Beendigung des Agenten gesichert werden, um bei einem erneuten Anlauf wieder zur Verfügung zu stehen. Die Speicherung der M-Beans und deren aktueller Zustand geschieht durch Serialisierung als Java Objekt in eine Datei. Für jedes M-Bean werden drei Elemente gespeichert:
  * Der eindeutige Objektbezeichner des M-Bean.
  * Das serialisierte M-Bean (Java Objekt).
  * Der Bezeichner des ClassLoaders, welcher das jeweilige M-Bean geladen hatte. Da es möglich ist, mehrere ClassLoader zu verwenden um auf unterschiedlich lokalisierte Klassen zuzugreifen, ist es bei der Deserialisierung erforderlich, den richtigen ClassLoader, welcher sich auf den entsprechenden ClassServer bezieht, zu ermitteln.

  Um den persistenten Repository Service zu verwenden muß zum einen eine Instanz der `FullPersistentRepSrv` Klasse geschaffen werden. Danach ist eine Konfiguration notwendig, welche beinhaltet, daß das Verzeichnis und der Bezeichner der Speicherungsdatei angegeben wird.
  Die Serialisierung eines M-Bean setzt voraus, daß das M-Bean
  * das Interface `java.io.Serializable` oder `java.io.Externalizable` implementiert,
  * Zugriff auf den Standard-Konstruktor der ersten nicht serialisierbaren Superklasse besitzt und
  * alle Elemente des M-Bean, welche nicht serialisiert werden sollen, den `transient` Kennzeichner besitzen.

  Die Deserialisierung eines M-Bean setzt, wie vorher erwähnt, voraus, daß der Class Loader, welcher für das M-Bean zuständig ist, auf dem Rechner existiert. Zudem muß er eine Methode `getObjectName` unterstützen, um den Objektbezeichner des M-Bean interpretieren zu können.
- Mischung zwischen dauerhafter und "flüchtiger" Speicherung der M-Beans (Mixed Repository)
  Es ist ebenso möglich, eine M-Bean bzw. seine Komponenten unterschiedlich zu behandeln. An dieser Stelle kommt der oben beschriebene `transient` Kennzeichner zu tragen.

• Der Metadata Service
  Der Metadata-Service wird benötigt, um die Eigenschaften und Methoden eines M-Bean im CMF verwenden zu können. Der Service wird unter JDMK mit der Klasse `com.sun.jaw.reference.agent.services.MetaDataSrv` zur Verfügung gestellt. Diese Klasse basiert auf der Reflection API des JDK und implementiert das `com.sun.jaw.reference.agent.services.MetaDataSrvIf` Interface.
  Es gibt zwei Möglichkeiten, einen eigenen Metadata Service zu implementieren:
  - Verwendung einer `set`-Methode innerhalb des CMF.
    Dies hat den Nachteil, daß externe Applikationen wie Manager und andere Agenten

keinen Zugriff auf diesen Service bekommen können. Zudem kann er nur nicht persistent gespeichert werden.

  – Instantiierung und Registrierung eines neuen Metadata-Service Objekts.
    Damit wird der vorhergehende Metadata-Service ersetzt. Es muß ein eindeutiger Bezeichner bestimmt werden. Der Vorteil eines eigenen Metadata-Service besteht darin, daß für Applikationen, welche über die Reflection API auf den Agenten zugreifen und damit seine Struktur analysieren können, nur ein vom Entwickler gewünschter Teilbereich der Gesamtfunktionalität des Agenten bekannt gemacht werden kann.

- Der Filtering-Service
  Dieser Dienst ermöglicht es Applikationen auf spezifische M-Beans zuzugreifen. Es kann eine Auswahl über die Wertebereiche von Properties der M-Beans erfolgen.

3.7.2 Class-Loader

Der Class Loader wird unter JDMK dazu verwendet, benötigte Klassen in das CMF zu laden. Zum Einen kann das durch den lokalen Agenten selbst geschehen oder entfernt über einen Manager oder anderen Agenten durch den AdaptorClient (siehe Kapitel 3.6). Die Klassen, welche durch ClassLoader geladen werden, müssen sich entweder auf dem lokalen Rechner des JDMK-Agenten befinden oder durch einen ClassServer, welcher sich auf dem Rechner befindet, auf dem sich die Klassen befinden, zugreifbar sein.
Um den ClassLoader zu verwenden, muß dieser zu dem CMF des Agenten hinzugefügt werden. Es ist notwendig, einen eindeutigen Objektbezeichner zu wählen. Der Bezeichner beinhaltet genauere Informationen über den ClassLoader wie den Hostnamen auf welchem der ClassLoader läuft, über welchen Port er ansprechbar ist und die Angabe des ClassServers, welcher die Klasse zur Verfügung stellt.
Das Hinzufügen kann auf zwei verschiedene Arten erfolgen. Einmal kann der Agenten-Quellcode entsprechende Kommandos enthalten oder ein Manager kann entfernt über das `adaptorMO` Interface (AdaptorClient) das Objekt instantiieren.
Es ist möglich, mehr als einen Class Loader pro Agent zu verwenden. Dies hat den Vorteil, daß Klassen von verschiedenen Servern geladen werden können.
Alle verwendeten Class Loader müssen das `java.lang.ClassLoader` Interface implementieren. Dieses Interface wird durch JDK zur Verfügung gestellt. Es werden keine weiteren Interfaces durch JDMK unterstützt.

3.7.3 LibraryLoader

Der Library Loader wird dazu benutzt, "Native Libraries" zu laden[14]. Dies ist der Fall, wenn eine M-Bean Klasse, welche rechnerspezifischen Code verwendet (und somit eine Library zu laden hat, um korrekt arbeiten zu können), in das CMF geladen wird. Es ist zu beachten, daß der Library Loader nur dynamische Bibliotheken lädt, also .dll (Windows NT) und .so (Solaris) Bibliotheken. Um Bibliotheken zu laden, wird die CMF-Methode `loadLibrary` verwendet.
Da die Java Virtual Machine das Laden von Klassen oder Bibliotheken (native Libraries) über

[14] Die Anbindung dieser Bibliotheken kann beispielsweise durch das Java Native Interface realisiert werden

das Netz durch Verwendung der `SecurityManager`-Klasse[15] einschränkt, muß für das Laden der erforderlichen Klassen unter JDMK eine eigene Implementierung eines SecurityManagers zur Verfügung gestellt werden, um nicht auf die Standard-Zugriffsbeschränkungen angewiesen zu sein. Dies geschieht durch die Verwendung des AgentSecurityManagers, welcher in der Package `com.sun.jaw.impl.agent.services.security` von JDMK zur Verfügung gestellt wird. Die Verwendung wird innerhalb des Agenten durch das Importieren der Package und den Methodenaufruf `System.setSecurityManager(new AgentSecurityManager())` erreicht.

3.7.4 Class- und Library Server

Die Server dienen den Class- und Library-Loadern zum Zugriff auf Klassen und Bibliotheken. Sie stellen eigenständige Java Applikationen dar. Allerdings sind sie als M-Beans implementiert, so daß sie bei Bedarf in den CMF eines Agenten integriert werden können.

3.7.5 M-Let Service

Der M-Let Service (bzw. Management Applet Service) ermöglicht es dem Agenten, bei der Instantiierung und Registrierung M-Bean-Klassen aus einer angegebenen ".jar"-Datei[16] zu laden. Die Ermittlung des ".jar" Files geschieht durch das Laden und Parsen einer ".html"-Datei, welche sogenannte M-Let Tags besitzt. Die Stelle an welcher die Datei zu finden ist, ist als URL zu spezifizieren. Diese wird danach als String einer `performLocalLoadURL`-Methode (bzw. `performRemoteLoadURL`) übergeben. Somit kann entweder fest im Agenten die Adresse codiert oder von außen durch Angabe der Methode und dem entsprechenden Argument das Laden der Klasse initiiert werden.
Diese Art des Zugriffs auf M-Bean Klassen bietet verschiedene Vorteile:

- Selektion bestimmter M-Bean Klassen.
- Verteilte Speicherung von M-Bean Klassen. Hinzu kommt die Möglichkeit, in ihrer Struktur unterschiedliche M-Bean Klassen, welche jedoch den gleichen Klassenbezeichner besitzen, für einen Agenten zu verwenden. Es muß hierbei lediglich das entsprechende Archiv ausgewählt werden.
- Gruppierungsmöglichkeit für M-Bean Klassen zu logischen Einheiten.

Um nun den M-Let Service zu verwenden, muß ein ".html"-File erstellt werden, in welchem die M-Beans innerhalb von M-Let Tags beschrieben werden. Danach wird ein ".jar"-File erzeugt, das die Klassen der M-Beans beinhaltet, welche in der ".html"-Datei beschrieben werden. Es können auch mehrere und verschiedene ".jar"-Dateien existieren. Zusätzlich muß eine Instanz des M-Let Service im Agenten existieren.
Innerhalb des M-Let Tags sind verschiedene notwendige und auch optionale Parameter anzugeben.

[15] Diese Klasse ist in der Package java.security integriert. Der Grund für die Verwendung einer solchen Klasse ist die Annahme, daß über das Netz geladene Klassen als nicht vertrauenswürdig angesehen werden. Aus diesem Grund werden Zugriffsbeschränkungen für solche Klassen (z.B. Applets) geschaffen.

[16] JAR (Java Archive) Dateien sind mit anderen Archivdateien wie beispielsweise zip und tar(gz) vergleichbar. In diesen Dateien werden "class"-Dateien zusammengefaßt und komprimiert. Der Vorteil in der Verwendung dieser Dateien ist die einfachere Handhabung in Bezug auf die Übertragung von Klassen über ein Netzwerk. (Es muß lediglich nur eine Datei, welche zudem komprimiert ist, übertragen werden.

- CODE=⟨Klassenbezeichner des M-Bean⟩ Es ist der volle Klassenbezeichner des M-Bean anzugeben, einschließlich des Packagenamens.

- OBJECT=⟨Dateiname bei serialisierten M-Beans⟩ An dieser Stelle muß das ".ser"-File angegeben werden, falls es sich um ein serialisiertes M-Bean handelt. Auch hier muß wiederum die Datei in einer der ".jar"-Dateien vorhanden sein.

- ARCHIVE=⟨Liste der .jar Dateien, welche verwendet werden⟩ In diesen Dateien werden die M-Bean Klassen oder andere Dateien (Libraries, ".ser"-Dateien) gepackt und als Archiv zur Verfügung gestellt. Es sind mehr als eine ".jar"-Datei möglich, wobei diese dann durch Komma getrennt sein müssen und in "" gesetzt werden.

- CODEBASE=⟨URL des ".jar"-Files⟩ (optional)

- NAME=⟨Objektbezeichner des M-Bean, falls vorhanden⟩ (optional)

- VERSION=⟨Angabe der Versionsnummer des M-Bean⟩ (optional)

- PARAM=⟨Attributbezeichner⟩VALUE=⟨Attributwert⟩ (optional) Mit dieser Option ist es möglich, M-Beans bei Aufruf ihrer Konstruktoren im Zuge der Instantiierung erforderliche Parameter übergeben zu können. Einschränkend hierbei ist, daß die Parameter als String übergeben werden müssen. Dies bedeutet, daß das M-Bean eine `initCmf`-Methode (vgl. 3.2.1) zur Verfügung stellen muß, in welcher entsprechende Casts auf den gewünschten Typ von Parameter durchgeführt werden.

Es ist zu beachten, daß es entweder einen CODE- oder einen OBJEKT-Parameter geben muß. Nachfolgend ist ein Beispiel für eine ".html"-Datei aufgeführt.

```
<MLET
        CODE=Example.class
        ARCHIVE=exampleFile.jar
        CODEBASE=http://www.foo.com/jars
        NAME=:example.ident=1
        VERSION=1.0
>
</MLET>
```

Zusätzlich bietet der M-Let Service die Möglichkeit, Klassen, welche durch diesen Dienst geladen werden, zwischenzuspeichern. Dabei wird ein erneutes Laden der gewünschen Klasse (falls sich die Klasse noch im Cache befindet) verhindert und somit zusätzliche Netzbelastung vermieden. Durch diesen Cache-Mechanismus kann eine Versionskontrolle der Klassen und ".jar"-Dateien stattfinden. Die Version der Klasse, welche geladen werden soll, wird bei Parameter "PARAM" innerhalb des M-Let-Tag Bereichs angegeben. Ist die angegebene Versionsnummer der M-Bean Klasse höher als die Klasse, welche zwischengespeichert wurde, so wird die Klasse erneut geladen. Danach wird die Versionsnummer aktualisiert. Ist die Versionsnummer der zu ladenden Klasse gleich oder kleiner der zwischengespeicherten, dann ist kein erneutes Laden erforderlich[17].

[17] Wie bereits erwähnt, können gleichnamige, in ihrer Struktur jedoch unterschiedliche Klassen mit Hilfe der Archive für einen Agenten verwendet werden. Dieses Verfahren entspricht in etwa dem Überladen von Methoden bei objektorientierten Programmiersprachen. Ein Irrtum hieraus bei der Überprüfung von gültigen Versionen im Cache wird durch die Angabe des zugehörigen Archivs (Name des ".jar"-Files) ausgeschlossen.

3.7.6 Bootstrap Service

Der Bootstrap Service ist ein lauffähiges Java-Programm, welches zum Updaten von JDMK-Agenten benutzt werden kann[18]. Dieser Dienst nutzt das Konzept des M-Let-Service, um die Klassen, welche aktualisiert werden sollen, zu laden. Es wird eine M-Let-HTML-Datei benutzt und innerhalb des M-Let Tag werden die Applikation (bzw. der entsprechende Agent), das zugehörige ".jar"-File für die neueren Klassen und gegebenenfalls die URL des Archivs innerhalb des M-Let-Tags spezifiziert.
Bei der Aktualisierung muß darauf geachtet werden, daß die Applikation das `java.lang.Runnable`-Interface implementiert. Nur dann kann der Bootstrap Service nach erfolgtem Laden der Klasse, diese auch durch den Aufruf der `run()`- Methode starten. (Bei JDMK-Agenten wird dann das CMF initiiert und die erforderlichen Adapter und Dienste sowie M-Bean-Klassen geladen).
Nachfolgend ist die Verwendung dieses Dienstes dargestellt:

```
// Die lauffaehige BootstrapSrv-Klasse
// ist in der Package:
// com.sun.jaw.impl.agent.services.bootstrap
// zu finden.
>java BootstrapSrv /versions/update/jdmkAgent/MLet_for_Update.html
```

3.7.7 Launcher-Service

Der Launcher Service bietet die Möglichkeit, sämtliche JDMK-Agenten, welche auf einem Rechner benutzt werden sollen, innerhalb eines von JDMK zur Verfügung gestellten "Base Agent" zu integrieren (bzw. deren M-Beans). Der Hauptvorteil besteht darin, daß die Vergabe der Ports, welche durch die Adaptoren der unterschiedlichen JDMK-Agenten belegt werden, zu keinerlei Konflikten führen kann. Die JDMK-Agenten (bzw. die M-Bean Klassen), welche in den "Base Agent" integriert werden sollen, müssen über den M-Let Service spezifiziert werden. Aufgrund seiner Struktur ist nur ein Launcher Service pro Agent erlaubt.

3.7.8 Event-Handling-Service

Der Service wird benötigt, wenn Events (welche in ihrer Struktur auch vom Benutzer selbst implementiert werden können) innerhalb einer Managementumgebung zu verarbeiten sind. Unter JDMK sind hierzu drei Elemente zu implementieren.

- <u>Event-Object</u>
  Soll eine Klasse auf Events, die sie aussendet, überwacht werden, muß die Klasse `java.util.EventObject` erweitern.

- <u>Listener-Objects</u>
  Listener Objects stellen Interfaces dar. Sie stellen Methoden zur Verfügung, die es ermöglichen, auf Events zu reagieren. Jede Methode besitzt nur einen Parameter für das Event Object, welches überwacht wird. Diese Methoden ermöglichen das Hinzufügen und Entfernen von ListenerObjekten innerhalb des CMF.

[18] Das Updaten kann so realisiert werden, daß ein ".jar"-File erzeugt wird, welches die aktuellen Klassen enthält. Danach wird in die Agenten, welche aktualisiert werden sollen, M-Beans eingehängt, welche mittels des M-Let-Service das Laden der Klassen aus dem "-jar"-File übernehmen. Nun müssen nur noch die alten M-Beans durch die neuen Klassen ersetzt werden

- Methoden zum Hinzufügen und Entfernen von Listener Objects
  Falls nun ein M-Bean mit dem Event Handling Service versehen werden soll, so müssen Methoden zum Hinzufügen und Entfernen von Listener Objects implementiert werden.

Um den Event-Handling-Service zu verwenden, muß zunächst ein Event-Objekt erzeugt werden. Dieses Objekt ist als Klasse zu definieren und mit eventuellen Properties zu versehen, welche zur späteren Parameterübergabe genutzt werden können. Zudem ist eine Klasse zu erstellen mit der das `java.util.EventListener`-Interface implementiert wird und die Methoden (bzw. deren Prototypen) zur Verarbeitung von Event zur Verfügung stellt.
Innerhalb des M-Beans, das bei Veränderung bestimmter Properties einen Event senden soll, müssen Methoden zum Hinzufügen und Entfernen von ListenerObjekten zur Verfügung gestellt werden. Um das Instantiieren von Listenern von einem Manager zu veranlassen, muß nun ein MOGEN-Generatorlauf durchgeführt werden, welcher die entsprechenden Stubs und Interfaces erzeugt.
Die erforderlichen Klassen und die erzeugten Klassen nach dem MOGEN-Generatorlauf sind in Abbildung 3.5 zu ersehen.
Die Verarbeitung der Events geschieht nach folgendem Muster:

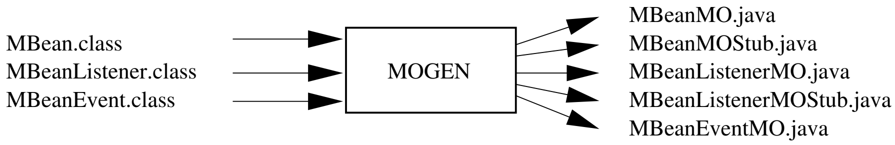

Abbildung 3.5: Output des MOGEN Compilers

1. Ein M-Bean erzeugt einen Event.
2. Dieses Event-Objekt wird vom Listener registriert und über den ListenerStub an den entsprechenden C-Bean auf der Managerseite weitergeleitet.
3. Der C-Bean benachrichtigt nun alle `EventListenerMO`- Objekte mit dem Event-Objekt. Diese C-Beans führen dann entsprechende Methoden aus.

Der Ablauf eines Events und der Weg der Information ist in Abbildung 3.6 graphisch dargestellt.

3.7.9 Alarm-Clock-Service

Dieser Dienst ermöglicht es, nach festzulegenden Zeitintervallen Events zu senden. Die Intervalle werden in Millisekunden angegeben. Es werden alle Listener, welche auf `AlarmClockEvent`-Objekte achten, benachrichtigt.

3.7.10 Scheduler-Service

Dieser Service dient dazu, Events zu einem bestimmten Zeitpunkt auszulösen. Wenn ein Zeitpunkt eintritt, werden alle Listener, die auf `SchedulerEvent`-Objekte warten, benach-

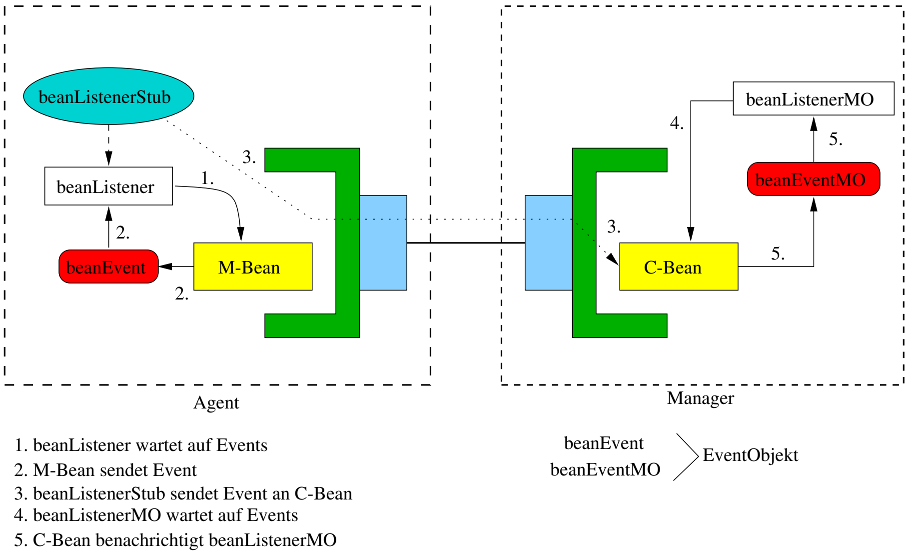

Abbildung 3.6: Verarbeitung von Events

richtigt. Alle Zeitpunkte werden als SchedulerEvent-Objects in einem Objekt-Vektor gespeichert. Sie repräsentieren eigene lauffähige Threads. Sie dienen dazu, das Eintreffen eines Zeitpunkts überwachen. Falls ein Zeitpunkt beim Start des Thread bereits eingetreten war, dann kann entschieden werden, ob ein Event ausgelöst werden soll, oder nicht. (Methode `notifyPastEvents(BOOLEAN)`)
Um einen Scheduler Listener einzurichten, muß das `SchedulerListener` Interface implementiert werden. Nach der Instantiierung ist das Objekt als Scheduler Listener registriert. Nachfolgend werden noch bestimmte Eigenschaften von Scheduler Events aufgeführt.

- Eigenschaften
  Scheduler Events stellen Instanzen der Klasse `SchedulerEvent` dar. Es werden zwei Arten von Events unterschieden. Ein Event kann einmalig auftreten oder innerhalb einer Periode. Die Periode kann entweder ein Zeitraum oder die Anzahl von bestimmten Ereignissen darstellen.

- Events zu einem Scheduler hinzufügen
  Um Events zu einem Scheduler Objekt hinzuzufügen wird die Methode `performAddEvent` verwendet. Hier kann entweder der Zeitraum in Millisekunden oder aber die Anzahl der Ereignisse, welche bis zum Auslösen des Events eintreten müssen, angegeben werden.

- Events aus einem Scheduler entfernen
  Das Entfernen geschieht durch die Methode `performRemoveEvent`. Es ist der Index des Events innerhalb des Vektors als Parameter anzugeben.

- Aktivieren/Deaktivieren

Ein Scheduler kann aktiviert und deaktiviert werden. Es kann durch eine Methode geprüft werden, ob der Scheduler zur Zeit aktiv ist oder nicht.

Der Unterschied zwischen Alarm-Clock-Service und Scheduler-Service besteht darin, daß bei ersterem Events nach festgelegten Intervallen ausgesandt werden, während bei Letzterem nur ein Event zu einem definierten Zeitpunkt geschickt wird.

3.7.11 Monitoring-Service

Der Monitoring Service bietet die Möglichkeit, eine Veränderung im Wertebereich eines M-Beans festzustellen. Es können bestimmte Properties überwacht werden. Die Intervallgröße zwischen den Prüfungszeitpunkten kann vom Benutzer festgelegt werden. Es werden zwei Arten von Monitoren zur Verfügung gestellt.

- Counter Monitor
  Mit diesem Monitor kann ein Zähler überwacht werden. Es wird eine obere Schranke festgelegt und dann ein Event ausgelöst, wenn diese Schranke erreicht oder überschritten wurde. Falls dieser Fall eintritt, kann ein Offset festgelegt werden, um diesen die Schranke erhöht wird. Das Überlaufen eines Typbereichs wird dadurch vermieden, daß ein Modulus gesetzt wird. In Abbildung 3.7 ist die Funktionsweise des Countermonitors aufgezeichnet.

- Gauge Monitor
  Um Schwankungen innerhalb eines Wertebereichs feststellen zu können, wird dieser Monitortyp verwendet. Dazu wird eine obere und untere Schranke festgelegt. Es werden Events ausgelöst, wenn eine der Schranken über- bzw. unterschritten wird. Um eine Anhäufung von Events bei einer Oszillation der Werte zu vermeiden, wird ein Hystereseschleifenmechanismus verwendet. Nach Überschreitung einer Schranke und Senden eines Events, wird erst dann wieder ein Event ausgelöst, wenn die entgegengesetzte Schranke erreicht und über- bzw. unterschritten wird. Als Wertetyp wird Integer und Float unterstützt. In Abbildung 3.8 ist der Mechanismus dargestellt.

Zusätzlich bietet der Monitoring Service, den Grad der Veränderung eines Wertes zu ermitteln. Die Differenz wird zwischen zwei Meßzeitpunkten berechnet und kann mit einem Vergleichswert geprüft werden. Events werden bei Gleichheit oder Überschreitung ausgelöst. Es ist zu beachten, daß die Differenz bei Verwendung des Counter Monitors durchaus auch negativ sein kann. In diesem Fall ist der Modulus der Differenz aufzuaddieren.

3.7.12 Discovery-Service

Mit diesem Dienst können JDMK Agenten innerhalb eines Netzwerkes ermittelt werden. Der Discovery Service wird in zwei Teilbereiche gegliedert:

- Discovery-Search-Service
  Hier werden die Elemente festgelegt, die notwendig sind, um Agenten innerhalb eines Netzwerkes feststellen zu können. Dies ist zum einen der Discovery Client, welcher vom Manager unterstützt werden muß und zum anderen der Discovery Responder, der für die Beantwortung der Anfrage des Clients zuständig ist. Beide Elemente besitzen spezifische Eigenschaften.

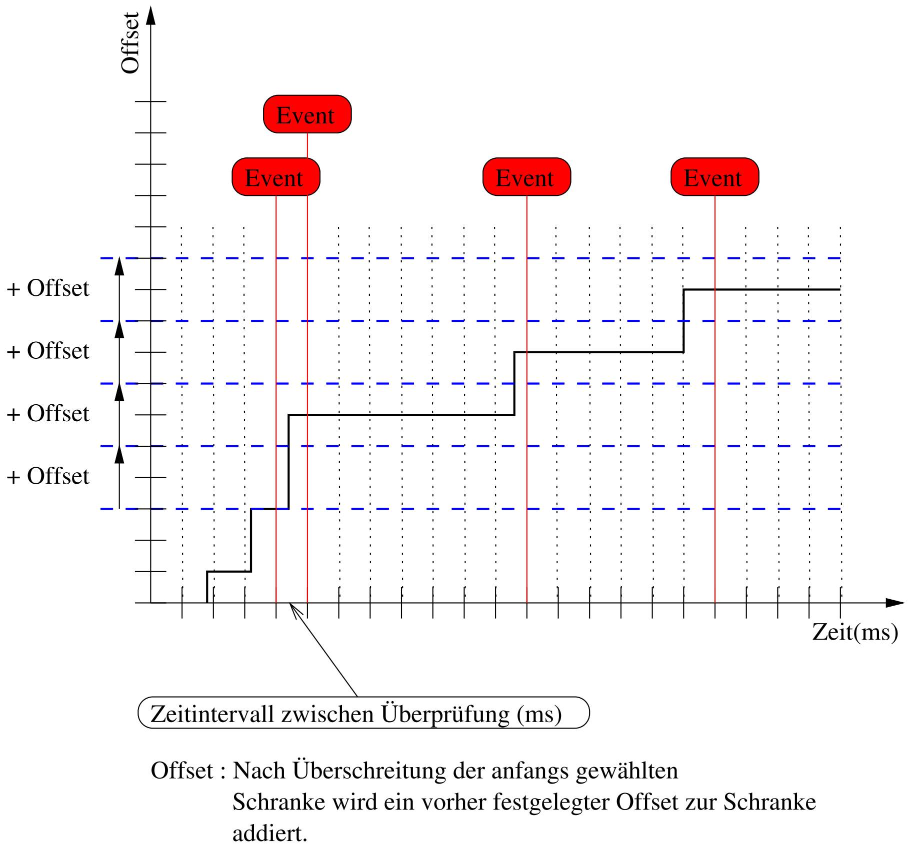

Abbildung 3.7: Funktionsweise des Counter Monitors

– Discovery-Client
  Hier werden Methoden zur Verfügung gestellt, welche das Auffinden von Agenten ermöglichen. Dazu wird ein Multicast Discovery Request ausgesandt und auf eventuelle Antworten gewartet.[19] Die Multicast Group ist standardmäßig als 224.224.224.224 und der Port als 9000 festgelegt. Es besteht die Möglichkeit, diese beiden Werte benutzerspezifisch anzupassen. Es ist zudem erlaubt, mehrere unterschiedliche Discovery Clients pro Manager zu verwenden. So können mehrere Gruppen von Agenten zusammengefaßt werden. Der Wartezeitraum nach einem Discovery Request wird durch ein Timeout Attribut spezifiziert. Als Standard ist 1 Sekunde festgesetzt. Der Lebenszyklus eines Requests kann ebenfalls geändert werden ist standardmäßig 1.

– Discovery Responder
  Jeder Agent, welcher durch den Discovery Service angesprochen wird, muß eine Instanz des Discovery Responders im CMF besitzen, um eine Antwort senden zu können. Sobald diese Instanz registriert wurde, schickt sie eine Registrierungsnachricht

[19] Dieser Request kann von Management-Applikationen oder anderen Agenten ausgesandt werden, welche ein Discovery-Client-Objekt erzeugt haben

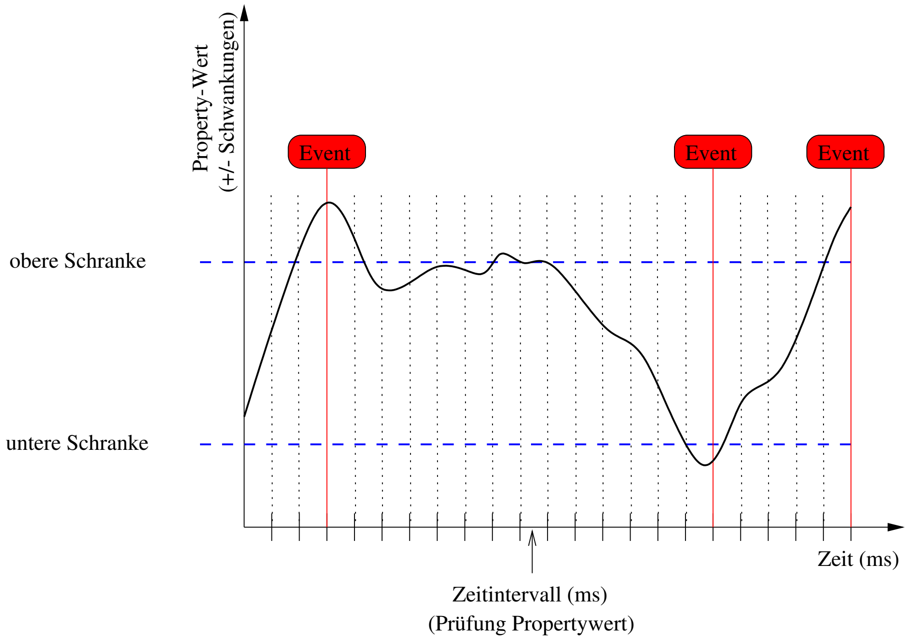

Abbildung 3.8: Funktionsweise des Gauge Monitors

an die Mulicast Gruppe, zu der der Agent gehört. Eine De-Registrierungsnachricht existiert analog. Diese Nachricht ebenso wie die Antwort auf einen Manager-Request enthält den Hostnamen sowie eventuelle Adapter des Agenten.
Die Antwort auf einen Request des Managers kann auf zwei Arten erfolgen. Einmal kann eine Antwort als Unicast realisiert werden, was standardmäßig der Fall ist, zum Anderen ist eine Multicast Variante möglich. Die Varianten für Uni- und Multicast sind in Abbildung 3.9 zu sehen.
Der Vorteil dieses Multicast-Konzeptes liegt darin, daß alle JDMK-Agenten, welche über einen Discovery-Responder verfügen, durch Initiierung eines einzigen Discovery-Search Aufrufs ermittelt werden können. Dies geschieht nach dem Schneeballprinzip. Bei Unicast wird nur der gesuchte Agent, falls verfügbar, antworten.

- Discovery Support Service
  Hier wird speziell der Discovery Monitor angesprochen, der die Registrierung und Deregistrierung von Discovery Responder Objekten überwacht. Falls dies der Fall ist, wird ein Event ausgelöst und damit das zugehörige Listener Objekt benachrichtigt. In Abbildung 3.10 ist die Funktionsweise des Discovery Support Service zu sehen.
  Der Vorteil eines solchen Monitors ist, daß eine Management-Applikation aber auch andere Agenten an einer zentralen Monitoring-Stelle Informationen über andere Agenten im Netz sammeln können. Es muß lediglich an dieser Stelle ein Monitor-Objekt vorhanden sein.

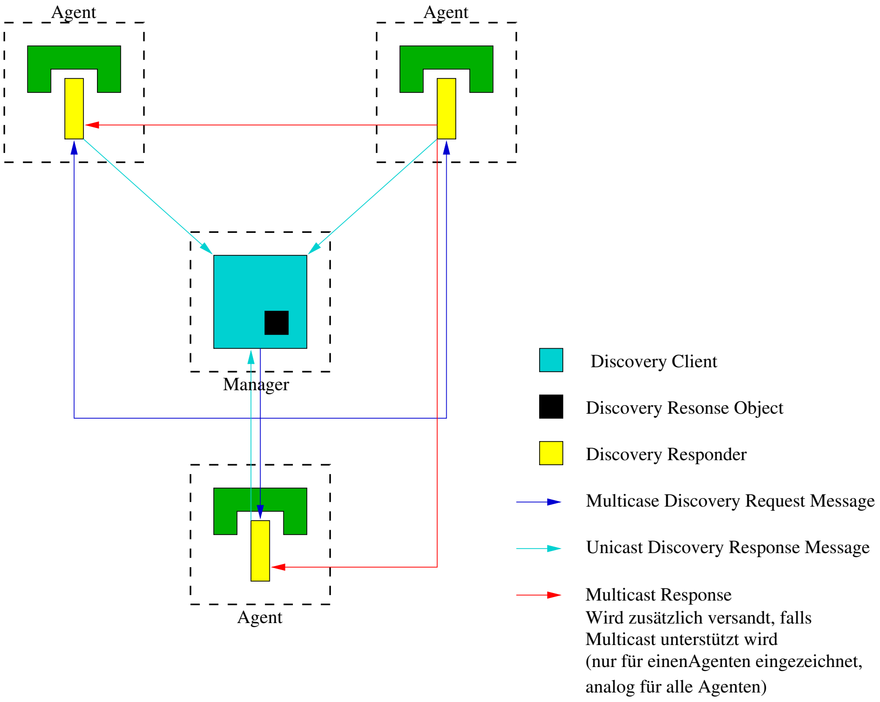

Abbildung 3.9: Funktionsweise des Discovery Search Service

3.7.13 Cascading Agent Service

Dieser Dienst ermöglicht den Aufbau einer Master- und Subagenten Hierarchie. Wird dieser Dienst unterstützt, so werden bei einem eventuellen Aufruf alle M-Beans des Subagenten in das CMF des Masteragenten gespiegelt. Es ist zu beachten, daß aktive Subagenten eines Masteragenten Zugriff auf die Eigenschaften (z.B. Attribute, Methoden) des jeweils anderen Subagenten besitzen. Abbildung 3.11 beschreibt den Zusammenhang zwischen Master- und Subagenten.

3.8 JDMK und SNMP

Das JDMK bietet eine Möglichkeit, SNMP-Agenten zu entwickeln. Die SNMP MIB wird daraufhin als eine Menge von M-Beans realisiert. Dies ermöglicht es JDMK auf SNMP Komponenten wie auf herkömmliche M-Beans zuzugreifen.

3.8.1 Der SNMP Agent Toolkit

Mittels JDMK können SNMP Agenten unter Java entwickelt werden. Es sind zwei Arten von Agenten möglich:

- Integrated Agents
  Diese Agenten laufen unter dem CMF des jeweiligen JDMK Agenten und können dessen

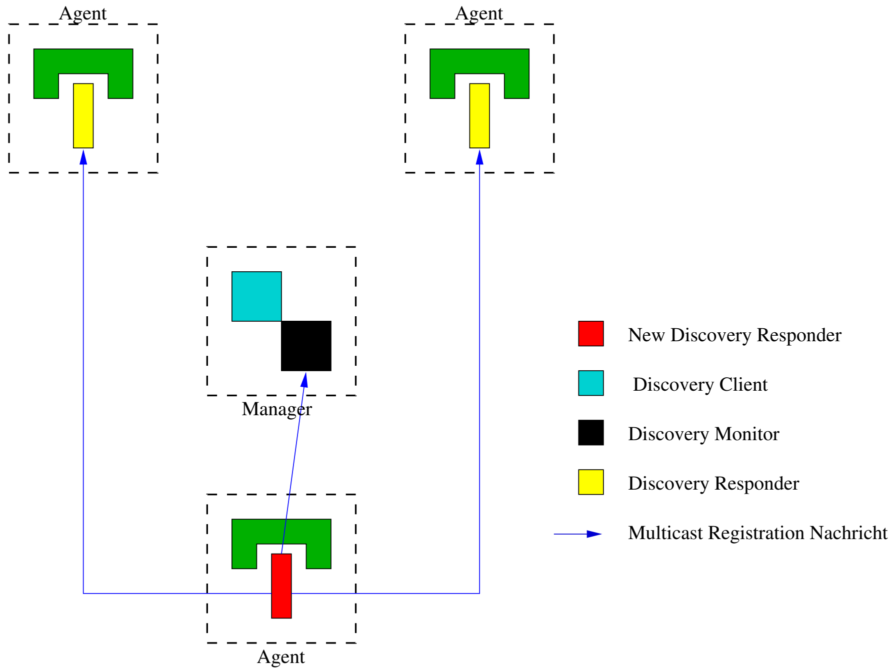

Abbildung 3.10: Funktionsweise des Discovery Support Service

Dienste nutzen. Sie können über die Adapter, welche durch den JDMK Agenten zur Verfügung gestellt werden, kommunizieren. Der große Vorteil dieser Art des SNMP Agenten ist es, daß die MIB direkt über sämtliche Adapter (RMI, IIOP, ...) des CMF zugreifbar ist. Das bedeutet, daß es möglich ist, die Elemente der MIB beispielsweise über einen CORBA-Client zu bearbeiten.

- Standalone Agents
  Dieses sind selbständig laufende Applikationen, welche nur über SNMP ansprechbar sind. Es können keine Elemente des JDMK genutzt werden, wie das bei den Integrated Agents der Fall ist. Der einzige Vorteil dieser Implementierungsmethode ist, daß die Größe der verbrauchten Resourcen minimal gehalten werden kann.

Der SNMP Adapter, welcher von JDMK bereitgestellt wird, unterstützt den SNMPv1 und SNMPv2 Protokollstack. Es ist möglich, MIB's zu Laufzeit des Agenten in das CMF zu laden und entsprechende MIB-Variablen zu verändern. Es können beliebig viele MIB's gemanagt werden.

3.8.2 Der SNMP MIB Compiler "MIBGEN"

Dieser Compiler nimmt als Eingabe eine SNMPv1 oder SNMPv2 MIB und erzeugt daraus Java Dateien, welche folgende Funktionen besitzen:

- Eine Klasse, die als Bezeichner den MIB-Bezeichner enthält
  Diese Klasse stellt den "Einstiegspunkt" der MIB innerhalb des JDMK-Agenten dar. Sie

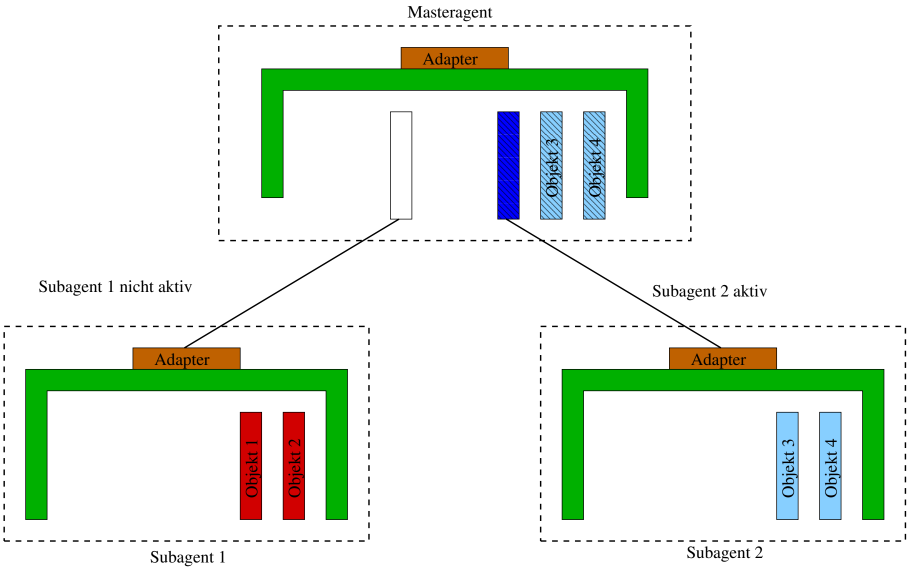

Abbildung 3.11: Funktionsweise des Cascading Service

instantiiert und ggf. registriert alle weiteren MIB-Elemente als M-Beans. Sie erweitert die abstrakte Java-Klasse `SnmpMib`. Diese Datei kann nachträglich bearbeitet werden um z.B. verschiedene SNMP Gruppen auszuschließen und die generierten Klassen durch die angepaßten Klassen zu ersetzen.

- Klassen, welche verschiedene SNMP Gruppen darstellen.
  Jede SNMP Gruppe wird durch zwei Java Dateien dargestellt. Einmal wird ein M-Bean Rahmen, welcher um Zugriffsmethoden erweitert werden muß, erzeugt. Er stellt die Initialisierung der MIB Komponenten sicher. Zum anderen wird eine Metadata-Datei erzeugt. Diese Datei stellt den Zugriff von SNMP auf das M-Bean sicher[20].

- Je eine Klasse für SNMP-Tabellen
  Diese Dateien besitzen das Präfix `Table` und beinhalten Methoden zur Indexermittlung und -generierung, das Einfügen bzw. Entfernen in der MIB, sowie alle Zugriffsmethoden.

- Je eine Klasse für Enumerationstypen
  Hier werden Klassen für alle Enumerationstypen von MIB Variablen erzeugt. In diesen Klassen werden alle möglichen Wertebereiche berücksichtigt. Somit können Web-Browser auf solche Typen über entsprechende Adapter zugreifen.

In Abbildung 3.12 ist graphisch der Zusammenhang zwischen Input und Output des MIBGEN Compiler dargestellt.
Beim Zugriff von SNMP Managern sei erwähnt, daß solche Manager nur Zugriff auf M-Beans

[20] Diese Metadata-Datei ist nicht mit dem Metadata-Service gleichzusetzen. Während der Metadata-Service für die Reflection-API unter Java benutzt wird, beinhaltet die Metadata-Datei die möglichen Werte der durch die MIB repräsentierten Variablen und Tabellen.

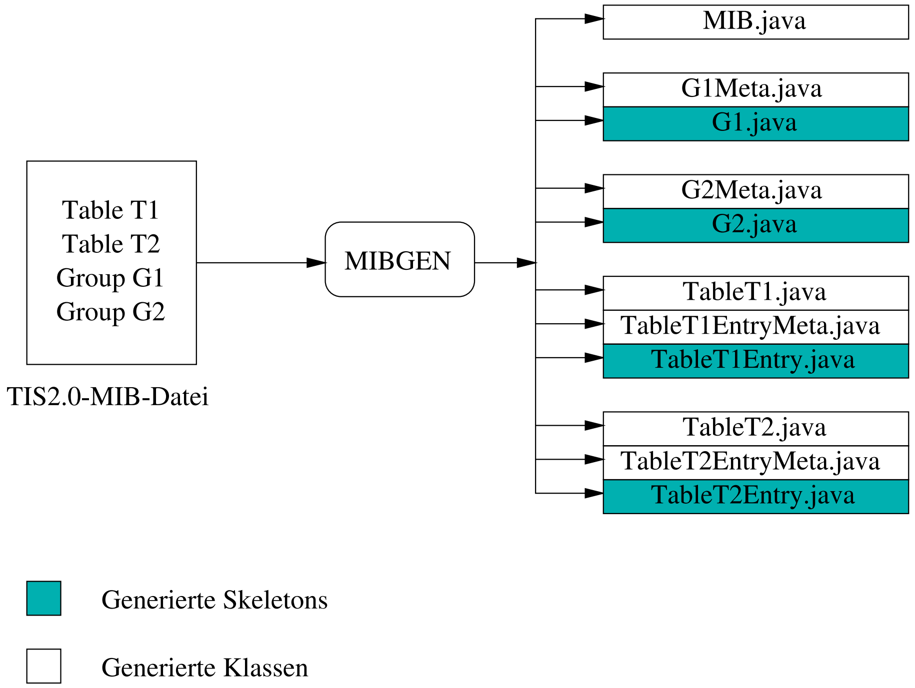

Abbildung 3.12: MIB - Generierung durch MIBGEN

besitzen, die durch MIBGEN generiert wurden. Der Zugriff geschieht über den SNMP Adapter. Manager, die über andere Adapter mit dem Agenten verbunden sind, besitzen Zugriff auf alle M-Beans.

3.8.3 Information Mapping

Die Variablen der MIB müssen in den Klassen, die durch MIBGEN erzeugt werden, umgesetzt werden. Für jede Variable und deren Zugiffsart werden `get-` und `set`-Methoden innerhalb des entsprechenden M-Bean erzeugt. Die MIB Variablen werden in eine entsprechende Java Syntax umgesetzt, wobei bestimme Regeln gelten. Diese Regeln sind in Tabelle 3.3 aufgeführt.

3.8.4 Laden der MIB über einen SNMP Adapter

Um den Zugriff auf ein MIB Objekt zu erhalten, muß es an einen SNMP Adapter gebunden sein. Die Bindung der MIB Objekte kann zur Laufzeit des Agenten geschehen (auch das Lösen der Verbindung). Ebenso kann der UDP Port für ein MIB Objekt zur Laufzeit geändert werden. Für den Ladevorgang werden zwei Varianten angeboten. Das statische und dynamische Laden.

- statisches Laden
  Dies hat den Vorteil, daß ein selbständig laufender Agent konstruiert werden kann.

- dynamisches Laden

SNMP Syntax	M-Bean Syntax
Objektbezeichner	Java String Objekt
IP Adresse	Java String Objekt
Display String	Java String Objekt
Opaque String	Array von Byte Objekten
Integer	Java Integer Objekt
Counter	Java Integer Objekt
Integer64	Java Long Objekt
Counter64	Java Long Objekt
TimeTicks	Java Integer Objekt
Truth value	Java Boolean Objekt
Enumerated List of Integers	Spezifische Java Klasse

Tabelle 3.3: Mapping der SNMP Variablen in Java Objekte

Wie bereits in Kapitel 3.8.4 erwähnt, kann eine MIB zur Laufzeit des Agenten hinzugefügt und wieder entfernt werden. Hierfür wird der dynamische Ladevorgang gewählt.

Kapitel 4

Der Telephony Internet Server

4.1 Einführung

Die bisherige Entwicklung der Kommunikationslandschaft war geprägt von der Trennung von Sprache und Daten. Während die analoge Kommunikation wie Audio, Video und Sprache weiterhin auf den bestehenden Systemen aufgebaut wurde, erstellte man für die Übertragung von Daten eigenständige Netze. Dies führte in vielen Bereichen zu Redundanz. Um diese Redundanz und somit auch die Kosten für Anschaffung und Wartung der Netze zu verringern, wurden Wege gesucht, die Übertragung von Sprache und Daten in ein System zu integrieren. Im Falle des Produkts Siemens Hicom 300 geschieht dies durch Einführung verschiedener Gateways und weiterer Komponenten, welche die Verbindung über ATM oder Internetprotokolle ermöglichen. Im Falle der Internetprotokolle ist dies der Telephony-Internet-Server. Dieser übernimmt die Funktion, eine Schnittstelle zwischen analoger Übertragung (Sprache, Video, Audio) und dem Internet zu definieren. Zudem soll es möglich sein, verschiedene Endgeräte miteinander zu verbinden. In Abbildung 4.1 ist eine solche Kommunikationslandschaft dargestellt.
Der TIS (Telephony Internet Server) bietet die Möglichkeit, moderne Kommunikation wie beispielsweise Videokonferenz durchzuführen. Zudem soll er in der Lage sein, Endgeräte wie ein normales Telefon (Hicom-Apparat) in eine solche Konferenz mit einzubeziehen. Er stellt eine Schnittstelle zwischen der IP-Welt und der ISDN-Welt dar und ermöglicht damit die Verbindung zweier Hicom 300 Telefonanlagen über Internet. So besitzt der TIS in diesem Zusammenhang Backboning-Funktionalität. Der TIS ist in 4 Teilbereiche gegliedert:

- Application Monitor
- Administration Server (AMS)
- Gateway
- Gatekeeper

4.2 Application Monitor

Dieser Bereich ist als Windows NT - Dienst implementiert worden. In einer Konfigurationsdatei, die beim Start des Dienstes eingelesen wird, kann festgelegt werden, welche Applikationen durch diesen Dienst gestartet und während ihrer Laufzeit beobachtet werden sollen.

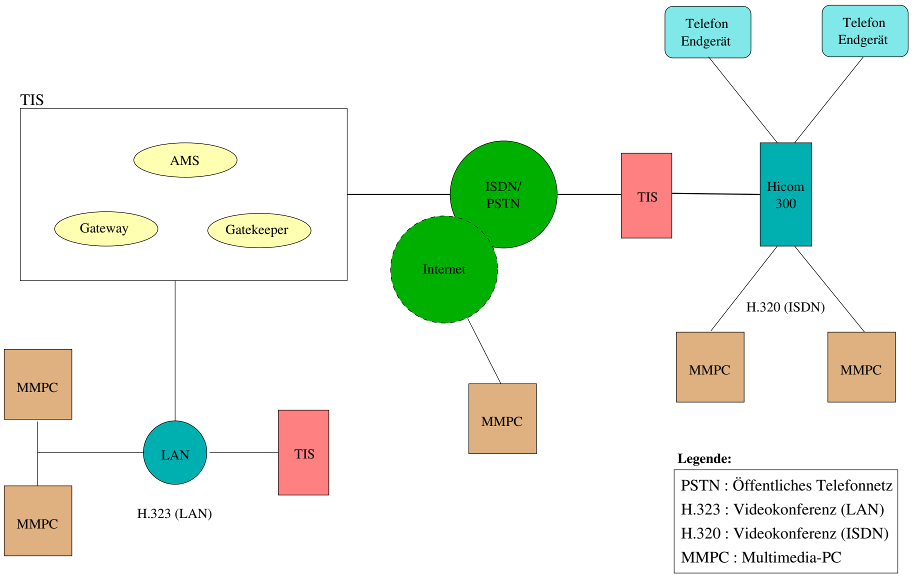

Abbildung 4.1: Kommunikation mit Telephony Internet Server (TIS)

Dieser Dienst startet die anderen Applikationen als Kindprozesse ohne direkten Eingriff durch einen Benutzer. Daraufhin überwacht er die Applikationen, bis sie beendet werden. Ist dies der Fall, so wird geprüft, ob dies eine gewollte Beendigung war oder ob sie Aufgrund eines Fehlers in der Applikation passiert ist. Ist ein Fehler aufgetreten, wird versucht, die Applikation erneut zu starten.
Der Application Monitor bietet eine API, die es Management-Applikationen ermöglicht, eine definierte Beendigung der Kindprozesse (Gateway etc.) einzuleiten bzw, diese wieder zu starten.

4.3 Administration Maintenance Server (AMS)

Dieser Server wird (falls angegeben) durch den Application Monitor gestartet. Die Hauptaufgabe des AMS (Administration Maintenance Server) ist es, Informationen bereitzustellen, die nicht durch die anderen Bereiche abgedeckt werden. Zu diesen Informationen gehören:

- TCP/IP Konfiguration
- SNMP Konfiguration
- Administrative Aufgaben wie Benutzerverwaltung
- Traps und deren Behandlung

4.4 Gateway

Das Gateway hat innerhalb des TIS verschiedene Aufgaben zu erfüllen:

- Verbindung von Telefonendgeräten mit anderen (Multimedia-)Endgeräten über LAN.
- Unterstützung des H.323 Standards (Videokonferenz über LAN)
- Aufbau einer bestmöglichen Videoverbindung nach H.323 unter Berücksichtigung der Netzbelastung.
- Unterstützung von PPP und 10BaseT/100BaseT Ethernetanschluß.
- CAPI - Unterstützung
- Routing-Funktionalität (Firewall, IP, Callback-Funktion)

4.5 Gatekeeper

Als weitere wichtige Komponente des TIS hat der Gatekeeper folgende Eigenschaften zu erfüllen:

- Die Kontrolle der zur Verfügung stehenden Bandbreite ist besonders wichtig, da dem Gatekeeper hier die Aufgabe zufällt, ausreichend Bandbreite für Videokonferenzen bereitzustellen. Als Kontrollelemente wären hier die Gesamtzahl der Anrufe, die gesamte maximale Bandbreite und die Bandbreite pro Endgerät zu nennen.
- Updaten der im Netz vorhandenen IP-Adressen der Endgeräte und Verbindungsstellen bei Registrierung und Deregistrierung. Da der Gatekeeper eine Tabelle mit Internetadressen der Komponenten besitzt, welche er überwacht, muß diese bei einer Veränderung der Managementumgebung wie das Hinzufügen oder Entfernen von Komponenten und damit das Neuauftreten und Verschwinden von Adressen angepaßt werden. Dies erfordert eine stetige Aktualisierung dieser Tabelle.
- Eine weitere Aufgabe ist die Adreßauflösung. Es werden Präfixe für andere Gateways, welche mit dem Gatekeeper verbunden sind, definiert. (Es ist zu beachten, daß ein Gatekeeper eine sogenannte "Zone" verwaltet. Es kann mehrere Gateways pro Zone geben, die mit verschiedenen Netzen und Endgeräten verbunden sind)
- Anforderung von regelmäßigen Statusrequests der im Netz vorhandenen Endgeräte.
- Der Gatekeeper übernimmt die Aufgabe, Die Komponenten innerhalb der TIS-Umgebung, die als Empfänger von IPC-Nachrichten vorgesehen sind, bei deren Hinzunahme zu registrieren bzw. bei Entfernen zu de-registrieren. Durch diese Registrierung ist der Empfänger einer IPC-Nachricht identifizierbar.

4.6 Struktur Gateway und Gatekeeper

Gateway und Gatekeeper werden beide in sieben weitere Unterbereiche gegliedert, welche als "Sektionen" bezeichnet werden. Diese Sektionen sind im folgenden:

4.6.1 Dispatcher

Der Dispatcher ist verantwortlich für die Zustellung von Nachrichten innerhalb des TIS. Diese Nachrichten stellen MsgBase-Objekte dar und werden als "IPC-Messages"[1] bezeichnet. Der Dispatcher hat folgende Aufgaben:

- Senden und Empfangen von Nachrichten zwischen sogenannten "MessageReceiver"-Objekten, welche sich in verschiedenen Prozessen, aber auch auf verschiedenen Rechnern befinden.
- Management von einer Reihe von Tasks, welche den Durchsatz von Nachrichten regeln.
- Bereitstellung einer API zur Registrierung und Deregistrierung von "MessageReceiver"-Objekten.
- Nachrichten können nach Priorität synchron aber auch asynchron versandt werden.

Dem Konzept des Dispatchers liegen mehrere Schlüsselkonzepte zugrunde:

- Die gesamte Information, wohin bestimmte IPC-Messages gesendet werden, ist im Dispatcher integriert. Ein Subsystem, also Gateways, Gatekeeper und die restlichen möglichen Komponenten geben nur bekannt, welche Nachrichten sie empfangen wollen und welche Art von Nachrichten sie aussenden. Der Dispatcher verwaltet eine Liste mit Message-Arten. Für jede Message-Art ist eine Liste von Empfängern vorhanden, welche die Nachrichten empfangen kann. Die Liste der Empfänger ist nach Prioritäten geordnet. Falls zwei Empfänger dieselbe Priorität besitzen, dann wird derjenige zuerst benachrichtigt, welcher sich zuerst eingetragen hat. Wenn nun der Dispatcher eine solche Nachricht erhält, wird er für jeden Empfänger eine Methode für die Weiterleitung dieser Nachricht ausführen. Die Empfänger werden prüfen, ob die Nachricht für sie bestimmt ist und entsprechende Aktionen ausführen, falls dies der Fall sein sollte. Falls während der Sendung der Nachrichten ein Fehler auftritt ist das Logging und die Benachrichtigung des Fault Managers Aufgabe des Empfängers. Eine Liste mit verschiedenen Message-Arten und ihren Empfängern ist in Abbildung 4.2 dargestellt.

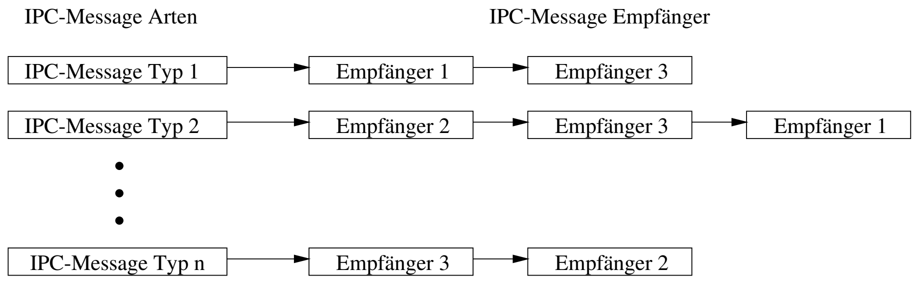

Abbildung 4.2: IPC-Message-Typen und Empfänger

[1] IPC steht für Inter-Prozeß-Kommunikation, wobei die Prozesse die Applikationen darstellen, welche über den Dispatcher verbunden sind

- Es soll ermöglicht werden, synchrone (sendMessage-Aktion) und asynchrone (postMessage-Aktion) Nachrichten senden zu können. Eine synchrone Nachricht wird direkt an die Empfänger weitergeleitet und der Sender fährt nicht eher fort, ehe alle Empfänger die Nachricht verarbeitet haben.
  Wenn der Sender keine Veranlassung besitzt, auf die Empfänger zu warten, kann eine asynchrone Nachricht geschickt werden. Aus diesem Grund werden Threads zur Verfügung gestellt, welche das Abwarten der Empfänger übernehmen.

- Da sich die Kommunikationsumgebung, welche durch den Dispatcher verwaltet wird, laufend ändern kann (unterschiedliche Komponenten werden hinzugefügt und entfernt), ist eine dynamische Messagestruktur erforderlich. Dies wird durch die MsgBase-Struktur realisiert.
  Ein MsgBase-Objekt besteht aus einer Liste von Tripeln, bestehend aus:

  - Name (ASCII-Feld)
  - Typ (Enumeration)
  - Wert

  des Parameters, der in der Liste eingefügt ist. Eine MsgBase-Struktur ist in Abbildung 4.3 exemplarisch dargestellt. Da der Dispatcher über alle Datentypen informiert ist,

Name	Typ	Wert
B-Channel	ULONG	17
Payload	MsgBase	
State	ULONG	4

Name	Typ	Wert
Device	ULONG	2
Format	STRING	"G.711"

Parameter

Abbildung 4.3: MsgBase - Struktur

können diese IPC-Messages über das Netz ohne Rücksicht auf deren Struktur gesendet werden.

- Die Vielzahl der von dem Dispatcher verwalteten Objekte werden in einer Baumstruktur verwaltet. Es werden zwei Arten von Objekten in diesen Baum eingefügt. Dies sind Containerobjekte und Basisobjekte. Der Unterschied zwischen diesen beiden Objekten ist, daß Basisobjekte nicht weiter aufgeteilt werden können, Containerobjekte jedoch weitere Containerobjekte aber auch Basisobjekte unter sich verwalten können. Es werden folgende Container-Objekte unterstützt:

  - Hash-Tabelle
    Objekte werden mit String-Kennzeichner gespeichert. Die Speicherung ähnelt einem Verzeichnisbaum, wobei die Objekte den Dateien entsprechen.
  - Array

  Die Objekte der Baumstruktur können selbst Funktionen ausführen, um Informationen über andere Objekte zu erlangen. Dies bildet mit dem Dispatcher das Rückgrad für eine Managementumgebung. Zudem wird diese Baumstruktur auch benutzt, um bei Bedarf die Objekte mit deren derzeitigen Status in einer Datenbank abzulegen.

4.6.2 Database

Da die Objekte, welche gespeichert werden müssen, dynamisch in ihrer Struktur sein können (unterschiedliche MsgBase-Objekte), mußte eine eigene Datenbasis implementiert werden. Die Datenbank ist in drei Schichten unterteilt:

- Seiten
  Es wird eine Liste von Seiten (Pages) gleicher Größe zur Verfügung gestellt. Diese werden zur Speicherung der Objekte benutzt. Sie werden bei Entfernen des Objekts wieder freigegeben. Jede Seite besitzt eine eindeutige ID.

- Records
  Records stellen ein in der Größe veränderbares Array von Bytes dar. Sie besitzen eine eindeutige "Object-ID". Records können mehrere Seiten umfassen.

- Dispatcher Objekte
  Diese Objekte werden mit Hilfe der Records gespeichert. Es wird ein Record pro Dispatcher-Objekt (Objekte innerhalb der Baumstruktur des Dispatchers) zugeordnet.

Für das Rücksetzen eines Datenzustandes ist eine Rollback-Funktion implementiert.

4.6.3 Dependability

Hier werden von den Komponenten abhängige und unabhängige Elemente verwaltet. Sie beschreiben, welches Verhalten Gateway und Gatekeeper im Fehlerfall zeigen. Dieser Komplex ist für die Fehleranalyse, Fehlerbeschreibung und eventuelles Reporting ausgeführt. Die Aufteilung geschieht durch sogenannte Manager:

- Fault Manager
  Der Fault Manager ist eine einzelstehende Komponente und kommuniziert mittels des Dispatchers mit den restlichen Komponenten des Gateways (Gatekeepers). Er verwaltet intern eine Tabelle, welche Nachrichten von anderen Komponenten, die ebenfalls Fault Manager sein können, auf Aktionen, die der Manager ausführen kann, abbildet. Der

Fault Manager kann selbst neue IPC-Messages erzeugen bzw. Logging betreiben. Dieses Logging geschieht nur an dieser Stelle, somit ist hier ein zentraler Einstiegspunkt zur Fehlerbehandlung zu finden.

- Device Manager
  Dieser Manager ist für die Boards, die innerhalb des TIS verwendet werden, zuständig. Zur Initialisierungsphase registriert[2] dieser Manager die von TIS verwendeten Boards und Hardwarekomponenten. Er verarbeitet nun IPC-Messages, die aktive und inaktive Boards kennzeichnen. Den Vergleich, welche Boards aktiv sein sollten und welche deaktiviert, findet dieser Manager innerhalb gewisser Einträge der Datenbasis (siehe 4.6.2). Falls dort ein Board als aktiv gekennzeichnet ist und während der Initialisierungsphase erkannt wird, es sei deaktiviert, so wird eine IPC-Message an den Fault Manager gesandt.

- Protocol Managers
  Diese Manager kontrollieren die Übertragung korrekter Protokolle. Werden fehlerhafte oder dem System unbekannte Protokolle empfangen, so wird der Fault Manager benachrichtigt. Dieser kann dann unter Umständen bestimmte Aktionen (die beispielsweise das Anpassen von Boards betreffen) durch den Device Manager veranlassen.

- Administrations-Prozeduren
  Es werden bestimmte Prozeduren, welche die Administration ermöglichen, durch HTML-Dokumente als Applets zur Verfügung gestellt.

- Error-Logging
  Das Error-Logging, welches durch den Fault-Manager betrieben wird, erfolgt durch den Windows NT Logging Service. Dies hat mehrere Vorteile:

  - Automatisches Management des Log-Files
  - Zeit/Datumsstempel für jeden Eintrag, sowie die Angabe der Fehlerquelle und Art (Klasse) der Fehlermeldung.
  - Erweiterung der Fehlermeldung durch Hinzunahme von Daten und Text durch Benutzer
  - Ein Event-Viewer, welcher innerhalb eines Netzwerks arbeitet und das Filtern und Sortieren von Events ermöglicht.
  - Eine API, welche Methoden zum Senden und Empfangen von Fehlermeldungen bereitstellt.

4.6.4 Administration

Diese Sektion verarbeitet Managementinformationen.

[2] Hier hat die Registrierung mit JDMK nichts zu tun, es ist nur notwendig, alle Elemente, welche Nachrichten vom Dispatcher empfangen oder versenden können, dem Dispatcher bekannt zu machen. Dies wird innerhalb des TIS als Registrierung bezeichnet. Ist ein Element registriert, kann der Dispatcher IPC-Messages eindeutig zuordnen.

4.6.5 Call Processing

Hiermit werden ankommende IPC-Messages und Events verarbeitet. Es werden drei Arten von "Calls" unterschieden (Low-Level, Call-Level und Session-Level), welche den Weg einer IPC-Message bzw. Events von der oberen Kommunikationsebene zu den (OEM-)Boards[3] handhaben.

4.6.6 Feature Processing

Hier werden die Dienste, welche durch das Gateway bzw. den Gatekeeper bereitgestellt werden, ausgeführt. Im Fehlerfall wird eine IPC-Message an den Fault Manager geschickt. Dies ist die eigentliche Funktionalität von Gateway und Gatekeeper.

4.6.7 Services

- Network Routing Service

  Der Network Routing Service hat die Aufgabe, ankommende Nachrichten, zum Beispiel Kommunikationsanfragen zu dem entsprechenden Endteilnehmer (oder auch dem Gateway) weiterzuleiten. Diese Anfragen können aufgrund der vorhergehenden Registrierung der TIS-Komponenten im Dispatcher eindeutig aufgelöst werden. Die Auflösung geschieht über eine Routing-Tabelle, wobei die Routing-Einträge nach Priorität sortiert sind.

- Protocol Interworking Service
  Dieser Dienst ist durch eine Klasse implementiert. Diese Klasse stellt Methoden zur Konvertierung von Protokollen in andere Protokollformate zur Verfügung. Es ist zu unterscheiden, daß die IPC-Messages nach außen hin immer die gleiche Klasse darstellen. Es werden nur je nach verwendetem Protokoll unterschiedliche Parameter hinzugefügt.

4.7 Bestehendes TIS-Management

Das bisherige Management umfaßt die in Kapitel 4 beschriebenen Applikationen, das Konzept des Windows NT SNMP Extension Agenten und verschiedene Applets, welche das derzeitige Front-End der Managementumgebung darstellen. Im folgenden Kapitel soll auf die für die Entwicklung des Prototypen wichtigen Komponenten eingegangen werden. Zunächst wird die Funktionsweise des Windows NT SNMP Extension Agent dargestellt und auf die Verwendung und Erweiterung des Agenten eingegangen. Danach wird der TIS-Extension Agent und die für die Einbindung in den SNMP-Agenten erforderlichen Methoden dargestellt. Zuletzt werden die Applets, welche für das derzeitige Management verwendet werden beschrieben. Das Zusammenspiel der zentralen Applikationen ist in Abbildung 4.4 schematisch dargestellt.

[3] OEM steht für Original Equipment Manufacturer und stellt eine Karte zur Kommunikation über LAN dar. Mit dieser Karte wird eine API mitgeliefert, die das Gateway benutzt.

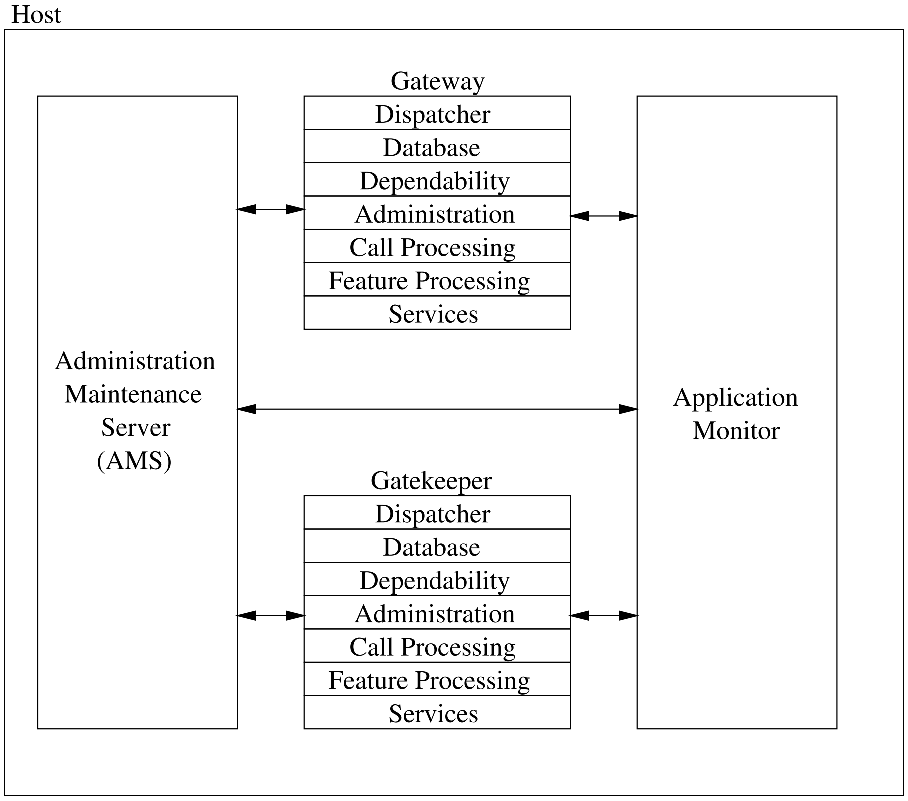

Abbildung 4.4: Zusammenspiel der TIS Applikationen

4.7.1 Windows NT SNMP Extension Agent

Unter Windows NT ist SNMP als eine Zusammenfassung zweier Dienste implementiert. Zum einen ist hier der SNMP-Agent zu nennen, welcher mit "SNMP.EXE" zur Verfügung steht. Dieser Dienst verarbeitet Get- und SetRequests und liefert GetResponse-Nachrichten und Traps zurück. Der Agent unterstützt das Windows Socket API, SNMP Message Parsing, ASN.1 bzw. BER Kodierung. Der SNMP Trap Service, ihm entspricht die "SNMPTRAP.EXE" Applikation, ist dafür geeignet, SNMP-Traps versenden zu können.
Die Erweiterbarkeit dieser SNMP-Agenten beruht auf der Möglichkeit, Bibliotheken (DLL's), welche MIB-Informationen tragen, zu dem Agenten hinzu zubinden. Dabei ist eine erneute Übersetzung der Agenten-Applikation nicht notwendig. Lediglich muß der Dienst nach der Anpassung wieder neu gestartet werden. Diese Bibliotheken, welche den SNMP-Agenten erweitern, werden "Extension Agents" genannt. Diese Subagenten werden in den Kontext des SNMP-Agenten eingefügt und dieser leitet ankommende Requests an den entsprechenden Subagenten weiter. In Abbildung 4.5 ist das Zusammenspiel der Komponenten dargestellt. Der SNMP-Agent ist dadurch zu erweitern, daß in der (Windows NT-)Registry[4] in dem Registry-"Verzeichnis" "LOCAL_MACHINE\SYSTEM\Services\SNMP\ ExtensionAgents" ein Ein-

[4] Die Registry unter Windows NT beinhaltet Informationen, welche für Applikationen, Dienste und Systemprozesse verwendet werden. Sie bezeichnen Einträge, an welcher Stelle im System wichtige Informationen wie

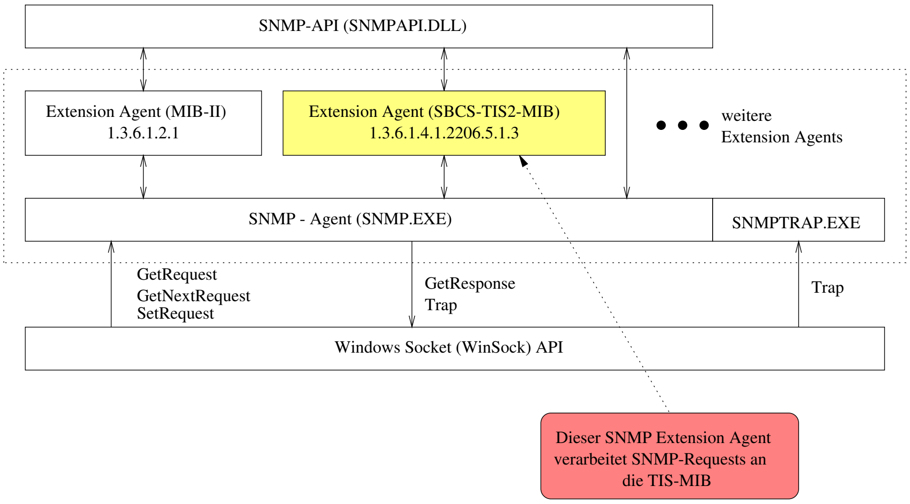

Abbildung 4.5: Windows NT Extension Agents

trag für den Extension Agent hinzugefügt werden muß. Dieser Eintrag verweist auf das Verzeichnis “LOCAL_MACHINE\SOFTWARE\ TIS-SNMP-Agent” (In diesem Fall ist dies der Extension Agent für die TIS-Box). In diesem hinzugefügten Verzeichnis (Schlüssel) ist nun die Position der DLL auf dem System (z.B. c:\TIS\Release\sxa.dll) anzugeben. Eine Einbettung des TIS-SNMP-Extension Agent ist in Abbildung 4.6 zu sehen.

4.7.2 SNMP Extension Agent (SXA)

Der Extension Agent für TIS (SXA) hat für die Einbindung in den SNMP-Agenten unter Windows NT folgende Methoden unbedingt zur Verfügung zu stellen:

- `SNMPExtensionInit()`
  Diese Methode wird bei der Initialisierungsphase während des Hochfahrens des SNMP-Agenten verwendet, um die Elemente der zu verwaltenden MIB in eine Baumstruktur einzulesen. Dafür wird eine Klasse `CAsnNode` zur Verfügung gestellt. Diese Klasse kann sämtliche in der MIB verwendeten Strukturen verarbeiten und somit eine einheitliche Knotenstruktur ermöglichen. Der Aufbau wie auch das spätere Auffinden der MIB-Knoten erfolgt durch Bisektion über die OID der entsprechenden Knoten. Die Information über die zu managenden Knoten ist in einer Datei, welche mit `tis_mib.mdf` bezeichnet wird, enthalten. Ein Auszug aus solch einer MIB ist nachfolgend dargestellt:

```
// Auszug aus einer tis_mib.mdf - Datei
// Einstiegspunkt
SEGMENT .1.3.6.1.4.1.2206.5.1.
// Dispatcher-Festlegung
DISPATCHER AdminTestDispatcher [127.0.0.1] 0x5005
DISPATCHER GatewayDispatcher [127.0.0.1] 0x1003
```

Bibliotheken zu finden sind, sowie andere wichtige Informationen. Im Fall des SNMP-Agenten von Windows NT können hier Extension Agents hinzugefügt werden, die bei Neustart des Agenten berücksichtigt werden.

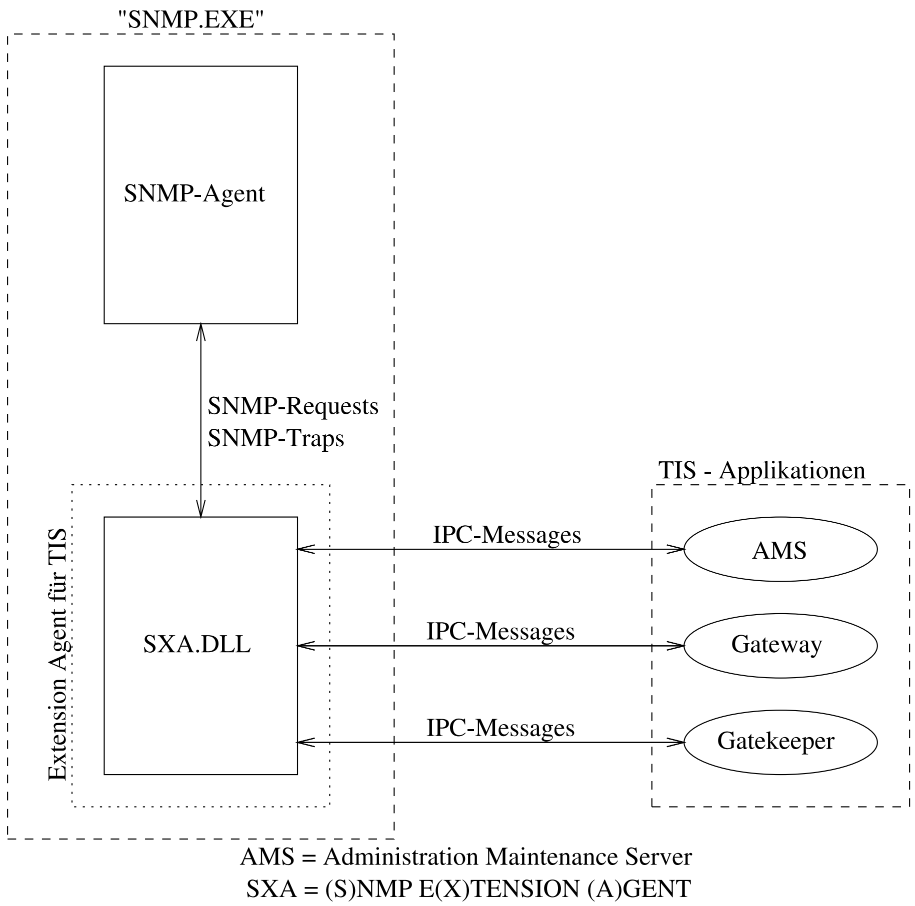

Abbildung 4.6: Einbettung TIS-SNMP-Extension Agent

```
DISPATCHER GatekeeperDispatcher [127.0.0.1] 0x5006
// Beginn Angabe der Knotenstruktur
// Einfacher MIB-Knoten
NODE RW .1.3.6.1.4.1.2206.5.1.1.1.1 AdminTestDispatcher computerName 2166784 SO
...
// MIB-Tabelle
NODE RW .1.3.6.1.4.1.2206.5.1.1.1.7.1.1
    AdminTestDispatcher
    dnsServerIndex 2166790 TI dnsServerTable 2166791 dnsServerIndex 2166790 II
...
// Trap-Festlegung
TRAP .1.3.6.1.4.1.2206.5.1.1.4.4 .1.3.6.1.4.1.2206.5.1.1.4.5
```

Diese Datei wird nun während der Initialisierung zeilenweise eingelesen und entsprechend Zeilen mit der Kennzeichnung "NODE" in die lineare Knoten-Liste als `CAsnNode`-Objekt eingehängt. Zeileneinträge mit "DISPATCHER"-Kennzeichnung werden an eine lineare Liste von Dispatchern angefügt. Der erste Eintrag mit "SEGMENT"- Kennzeichnung kennzeichnet den offiziellen Einstiegspunkt der TIS-MIB.

- `SNMPExtensionQuery()`

Wird über den SNMP-Agenten ein Request empfangen und dieser ist für den TIS-Extension Agent bestimmt, so wird SNMP diese Methode mit der Übergabe einer Varbind-Liste aufrufen. Innerhalb dieser Methode werden die Varbinds aus der Liste nacheinander mit einer Methode `ResolveVarbind()` verarbeitet[5].
Diese Methode gliedert nach Request-Typen und bildet je nach Anfrage[6] ein IPC-Message-Objekt. Dies wird durch Verwendung des `BuildMsgBase()`-Konstruktors erreicht, die wie in Kapitel 4 beschrieben, mit `addParameter` die entsprechenden Datenstrukturen der Nachricht hinzufügt. Mit dem Konstruktor `BuildVarBind` wird schließlich, nachdem ein intern verwalteter Cache geprüft wurde, der SNMP-Request als IPC-Message dem Empfänger zugesandt. Der Konstruktor wird erst beendet, wenn eine Response-IPC-Message vom Empfänger eintrifft.

- `SNMPExtensionTrap()`
  Als dritte Methode, welche von einem Extension Agent exportiert werden muß, um von dem SNMP-Agenten verwendet zu werden, ist für die Behandlung von Traps zuständig. Zum Zeitpunkt der Erstellung des Prototypen wurde noch keine offizielle Verwendung eines Trap-Mechanismus unterstützt. Gleichwohl sind bereits Programmteile dafür vorgesehen.

4.7.3 Management-Applets

Für das Management des TIS werden eine Reihe von Java-Applets zur Verfügung gestellt. Diese umfassen das Hinzufügen und Entfernen von Benutzern, Hardware und anderer Elemente, welcher in der TIS-MIB spezifiziert sind. Nachfolgend sind zwei Screenshots solcher Applets aufgeführt. In Abbildung 4.7 ist die Einstiegsseite für das TIS-Management dargestellt. Dort kann auf die einzelnen Komponenten gewechselt werden. Die zweite Abbildung (Abb. 4.8) zeigt das Applet, welches für das Management des Administration Maintenance Servers (AMS) zur Verfügung gestellt wird.

[5] Diese Methode wird ebenso innerhalb der JNI-Schnittstelle des Prototypen verwendet
[6] Die Anfrage kann von dem Anfragentyp und der betroffenen MIB-Variable abhängen

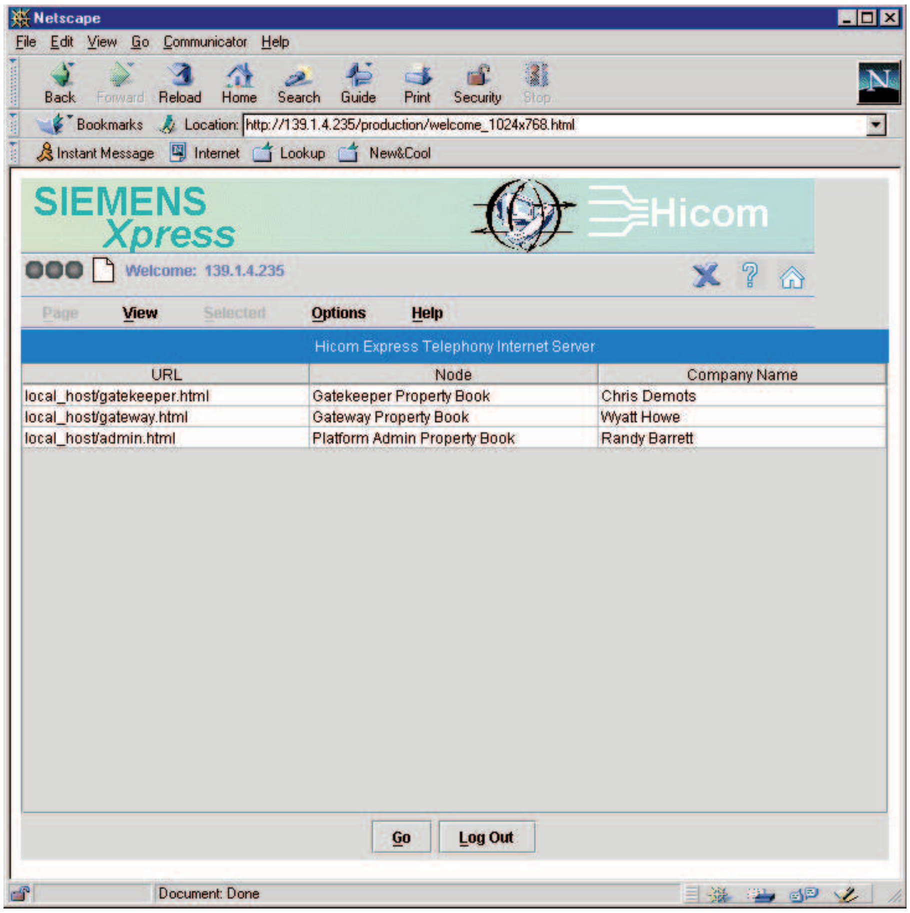

Abbildung 4.7: TIS-Einstiegsseite

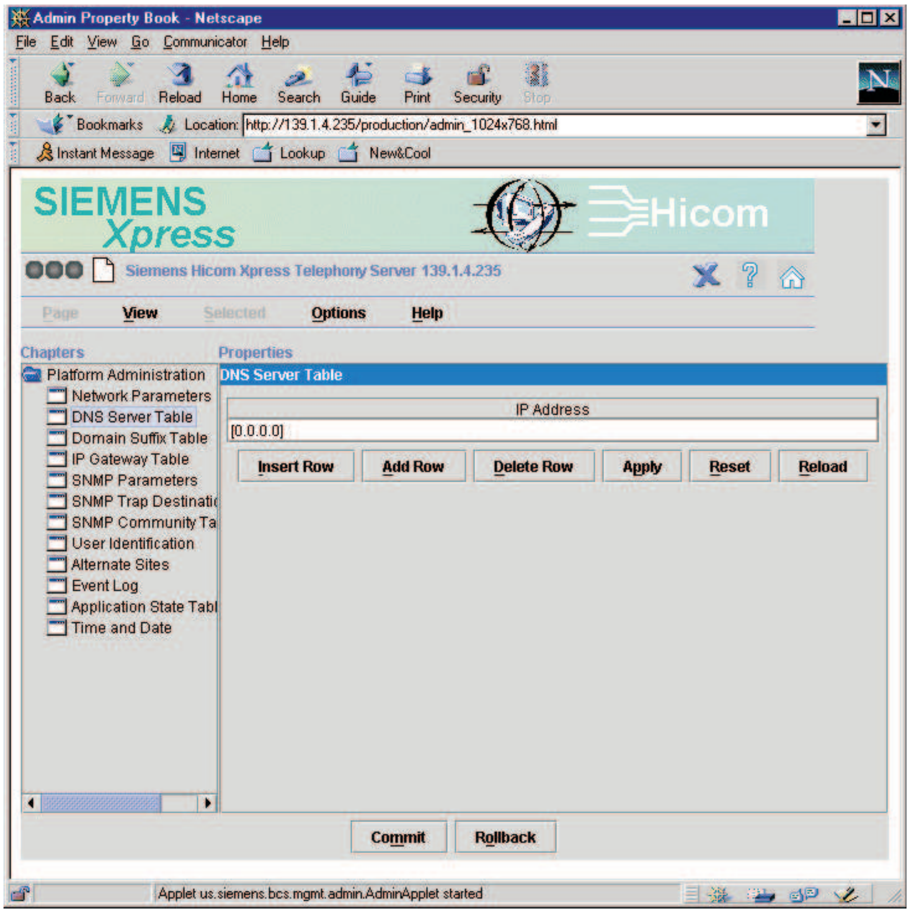

Abbildung 4.8: TIS-ManagementApplet für Gateway

Kapitel 5

Evaluierung von JDMK-Managementkonzepten

5.1 Bewertung des JDMK gegenüber FMA-Theorie

Das Java Dynamic Management Kit erfüllt auf den ersten Blick viele Forderungen, welche an Flexible Management Agenten und deren Umgebung gestellt werden. Im Folgenden soll ein Vergleich angestellt werden, in wieweit diese Forderungen, die in [Moun 97] formuliert wurden, erfüllt werden.

5.1.1 Systemeigenschaften

Bevor auf die Übereinstimmungen der Agentenarchitektur eingegangen wird, muß betrachtet werden, ob Merkmale und Anforderungen einer Managementumgebung, wie sie unter 2.4 beschrieben wurde, von JDMK erfüllt werden.

Organisatorische Merkmale

Nun soll geprüft werden, inwieweit JDMK diese Merkmale erfüllt, beginnend mit den Komponenten, die in einem solchen System agieren.

- Agenten
  Die Realisierung von herkömmlichen Agenten, wie sie SNMP-Agenten darstellen, ist unter JDMK durch die Verwendung des SNMP-Adapters gegeben. Dieser ermöglicht es unter JDMK eigenständige SNMP-Agenten zu entwickeln. Allerdings besitzen Management-Applikationen keine Zugriffsmöglichkeit über die restlichen Adaptoren und Protokolle. Der Zugriff wird erst ermöglicht, wenn diese M-Beans, welche durch das Generierungstool "MIBGEN" erzeugt wurden, innerhalb des Core Management Framework (CMF) registriert sind. Sind die generierten M-Beans nicht registriert, können sie nur von SNMP-Managern über den entsprechenden SNMP-Adapter angesprochen werden.

- Flexible Management Agenten
  Diese Agenten werden innerhalb von JDMK durch die Rahmenstruktur des Core Management Frameworks (CMF) dargestellt. Durch die Möglichkeit, M-Beans der Rahmenstruktur zur Laufzeit des Agenten hinzuzufügen bzw. diese wieder zu entfernen, wird

bereits eine wichtige Forderung erfüllt. Die Fähigkeit, mit anderen Agenten zu kooperieren ist durch ein Eventmodell und Dienste, die durch JDMK zur Verfügung gestellt werden, gegeben. Die genaue Darstellung der Übereinstimmungen von JDMK-Agenten mit Flexiblen Management Agenten wird in Kapitel 5.1.2 betrachtet.

- Manager
  Durch das Adaptor-Konzept, welches die Anbindung der unter JDMK entwickelbaren Agenten unter Verwendung einer Reihe von Protokollen ermöglicht, kann eine Management-Applikation das Management von JDMK-Agenten durchführen. Das modulare Konzept des JDMK bietet keine eigenständige "Management"-Applikation. Jedoch kann mit den gegebenen Modulen eine solche Applikation erstellt werden. Die Bereitstellung einer Benutzerschnittstelle liefert JDMK durch die Möglichkeit über HTML den Agenten[1] (bzw. den Manager) anzusprechen.
  Wenn der JDMK-Agent über den HTML-Adapter mittels eines Web-Browsers angesprochen wird, so generiert er HTML-Seiten, die den Zugriff auf die M-Beans des Agenten (falls es die Zugriffsrechte erlauben) ermöglichen. Zudem wird ein Management-Tool "JOB" angeboten. Es bietet ebenso den Zugriff auf Agenten und deren M-Beans, allerdings über die Adaptoren, welche die Protokolle RMI und HTTP zur Verfügung stellen. Dieses Tool ebenso wie die Möglichkeit der HTML-Darstellung wird in Kapitel 6.7 behandelt.

Weitere organisatorische Merkmale, die an eine solche Umgebung gestellt werden ist die Strukturierung ihrer Agenten nach Domänen und Gruppen:

- Domänen
  Die Anordnung von Agenten innerhalb von Domänen ist ein Bestandteil der Namensgebung für M-Beans, welche in JDMK-Agenten registriert werden. Der Aufbau eines Bezeichners ist in 3.2.2 dargestellt. Es muß beachtet werden, daß die Domänenstruktur, wie sie durch JDMK unterstützt wird, nur einstufig verwendet werden kann. Es ist nicht möglich, Subdomänen zu bilden. Zumindest ist es nicht möglich, eine Subdomäne innerhalb der Bezeichnersyntax anzugeben. Hier kann nur eine geschickte Wahl des Domänenbezeichners das Problem lösen.

- Gruppen
  Der Bezeichner, der die Identifizierung des M-Beans ermöglicht, stellt keine explizite Angabe einer Gruppe, zu der der Agent gehören könnte, zur Verfügung. Eine Gruppierung kann unter Umständen durch die Anordnung von M-Beans einer "Gruppe" innerhalb eines Agenten oder durch einen Zusatz innerhalb des Bezeichners geschehen. Bei der Anpassung des Bezeichners, der frei wählbar[2] ist, kann eine Gruppenbezeichnung realisiert werden.
  Der Zugriff auf eine Gruppe von M-Beans kann mit Hilfe des Relationship-Service, der durch JDMK unterstützt wird, erfolgen (vgl. 3.2.4). Dieser Dienst stellt eine Abfragemöglichkeit, wie sie beispielsweise unter SQL zu finden ist, zur Verfügung. Durch die eingeschränkte Suche kann eine "Gruppe" von M-Beans herausgefiltert werden.

[1] Im weiteren werden die Begriffe JDMK-Agent und Manager synonym verwendet, falls es eine Management-Applikation ist, die mit Hilfsmitteln des JDMK entworfen wurde. Da sich die Rollenverteilung unter JDMK dynamisch anpaßt.

[2] Es ist selbstverständlich, daß ein Index, der für die Fortzählung bei gleichnamigen M-Beans benötigt wird, berücksichtigt werden muß

Kommunikationsmerkmale

Zuerst muß betrachtet werden, ob JDMK die drei Datenarten, welche innerhalb einer Managementumgebung von FMA´s transportiert werden, unterstützt.

- Tasks
  Tasks entsprechen auf Agenten ausgeführte Aktionen, welche von Managern oder anderen Agenten angestoßen werden. Dies kann die Abfrage oder das Setzen einer Variablen sein, aber auch Methoden, welche ausgeführt werden. Die Verwendung von Tasks wird unter JDMK unter zwei Arten ermöglicht. Einmal kann die Management-Applikation durch die direkte Angabe des M-Bean-Bezeichners und der Property, welche die Information enthält mittels einer `get-` und `set-` Methode angesprochen werden. Dies läuft unter dem Begriff "Low-Level-Interface". Die andere Variante wird unter der Bezeichnung "High-Level-Interface" von JDMK zur Verfügung gestellt. Auf diese Weise wird ein Task durchgeführt, indem auf der Client-Seite ein C-Bean instantiiert wird und dann das entsprechende Property "direkt" mit diesem C-Bean angesprochen wird. Die Verwendung dieser beiden Interfaces wird in Kapitel 3.6 beschrieben.
  Methoden können durch den Aufruf einer `perform<Aktion>`-Methode angestoßen werden. Diese Methoden müssen jedoch durch den Agenten, auf welchem der Task läuft, bereitgestellt werden.

- Information
  Informationen aller Art werden innerhalb einer JDMK-Managementstruktur zwischen den Komponenten ausgetauscht. Dies können einfache Property-Werte, Events oder andere Objekte sein.

- Funktionalität
  Die Übertragung von Funktionalität findet unter JDMK statt, wenn eine Manager-Applikation (oder der Agent selbst) ein M-Bean, welches die geforderte Funktionalität besitzt, in das CMF des entsprechenden Agenten integriert (Instantiierung und Registrierung). Der Fall, in dem "physikalisch" ein Klasse eines solchen Beans transportiert wird, tritt auf, wenn die Klasse mittels des M-Let-Service (siehe Kap. 3.7.5) oder unter Zuhilfenahme eines Remote Class Loaders aus einem Repository geladen wird.

In den Bereich der Kommunikationsmerkmale fällt auch die Bereitstellung von Diensten durch die Managementumgebung. Die Gegenüberstellung der geforderten mit den von JDMK zur Verfügung gestellten Diensten ist im folgenden aufgeführt:

- Delegations-Dienst (Delegation Service)
  Unter JDMK wird kein eigener Delegationsdienst mit spezifischen Methoden zur Verfügung gestellt. Die Delegierung von Management-Funktionalität geschieht durch die direkte Verbindung über Adaptoren und die entsprechenden Aktionen zur Integration eines M-Beans innerhalb des zu delegierenden Agenten. Trotzdem ist die Forderung der Delegierung erfüllt. Der Agent kann selbst Management-Funktionalität erzeugen (bzw. auch die Klassendateien mittels M-Let-Service laden), Management-Funktionalität in andere Agenten delegieren. Dies bedeutet er kann als Delegierer und Delegierter auftreten.

- Ereignis-Dienst (Event Service)
  JDMK stellt einen umfangreichen Ereignis-Dienst zur Verfügung. Es können Event-Objekte zu den Agenten gesandt werden, welche vorher ein EventListener-Objekt an

der Stelle, an der der Event auftritt, instantiiert haben. Es wird also kein zentraler Event-Server, wie er in der Forderung einer Management-Umgebung für FMA´s definiert wurde, zur Verfügung gestellt. Der Agent muß sich selbst ermitteln, an welcher Stelle der Event auftreten kann und angeben, welche Art von Events er erwartet. Diese Events werden als abgeleitete Klassen aus der Event-Klasse von JDK bestimmt. Der Ereignisdienst ist bereits als Alarm-Clock-Service (siehe Kap. 3.7.9), Monitoring-Service (siehe Kap. 3.7.11) und Scheduler-Service (siehe Kap. 3.7.10) vordefiniert. Diese Dienste stellen Basisklassen für Event-Objekte und Listener zur Verfügung.

- Gruppierungsdienst (Group Service)
  Die Gruppenkommunikation wurde unter JDMK mittels des Discovery-Service (siehe Kap. 3.7.12) realisiert. Die Voraussetzung, um diesen Dienst zu nutzen bzw. um unter Benutzung dieses Dienstes erkannt zu werden ist, ein Discovery-Responder-Objekt im CMF des Agenten zu registrieren. Ist dies geschehen, so kann der Agent mittels des "Discovery-Search-Service"-Dienstes ermittelt werden. Bei der Registrierung eines Discovery-Responder-Objekts ist die Angabe einer Multicast-Gruppe (z.B. 224.224.224.224) und eines Ports möglich. Wenn keine Angabe gemacht wird, werden Default-Werte verwendet. Bei der Ermittlung der Agenten muß nun die Multicast-Gruppe und der Port angegeben werden. Sind in dieser Konstellation Agenten "eingetragen", so werden diese antworten.

- Agent-Bestimmungs-Dienst (Location Service)
  Ein solcher Dienst ist in JDMK nicht explizit implementiert. Ein Agent wird direkt über seine Rechner-Port-Adresse (z.B. localhost:8082) angesprochen. Genauer gesagt wird eigentlich einer seiner Adapter, welcher auf den Port gesetzt ist, angesprochen. Der Agent hat keine Möglichkeit, seine Adaptoren bzw. seine Position zentral innerhalb der Management-Umgebung bekanntzugeben. Sehr wohl kann eine Management-Applikation realisiert werden, welche unter Zuhilfenahme des Discovery-Service die Agenten, welche sich im der Managementumgebung befinden, in eine Liste einträgt. Es ist allerdings nicht möglich, die Verfügbarkeit eines Agenten zentral zu bestimmen. Die Verfügbarkeit wird erst dann deutlich, wenn der Versuch einer Verbindung mit einem seiner Adapter zum Erfolg führt oder fehlschlägt. Unter der Verwendung des IIOP-Adapters kann in einem Naming-Service der Agent mit seinen zur Verfügung stehenden Methoden eingetragen werden. Genaueres ist unter 5.2 zu finden.

- Sicherheits-Dienst (Security Service)
  Die Sicherheit innerhalb der JDMK-Management-Umgebung wird über die verwendeten Protokolle realisiert. JDMK bietet die Möglichkeit eines ACL-Files (siehe Kap. 3.5), welches den Zugriff für bestimmte Adressen mit Zugriffsrechten verknüpft. Diese Datei wird beispielsweise verwendet, wenn z.B. über den SNMP-Adapter kommuniziert wird. Ansonsten bietet JDMK ein Login/Passwort-Sicherheitskonzept (z.B. HTTP/SSL, HTML). Die Login´s und Passwörter sind jedoch nicht zentral, sondern müssen dem jeweiligen Adapter, welcher im CMF des entsprechenden Agenten registriert ist, hinzugefügt oder entfernt werden. Durch die Verwendung des HTTPS (SSL) Protokolls ist eine sichere Übertragung der Authentifizierungsinformation gewährleistet.

- Management-Dienst (Management Service)
  Dieser Dienst findet seine Entsprechung in dem Monitoring-Service des JDMK. Damit

hat eine Management-Applikation die Möglichkeit, innerhalb eines Agenten auf das Eintreten vorher festgelegter Ereignisse zu warten. Somit kann bei der Delegierung von Funktionalität gleichzeitig ein Monitor in dem Agenten integriert werden, der den Fortgang der Ausführung der Funktionalität überwacht.

5.1.2 Agenteneigenschaften

Plattformunabhängigkeit

Die Forderung der Plattformunabhängigkeit ist durch die Verwendung von Java als Implementierungssprache erfüllt. Ein wesentlicher Nachteil bei dieser Art von Sprachen ist, daß eine Java Virtual Machine auf dem Rechner, auf dem der Agent zu laufen hat, vorhanden sein muß. Dies kann besonders bei Managed-Devices (Knoten), welche nicht über ausreichend Speicherplatz verfügen, zu einem Problem werden. An diesem Punkt ist es zu überdenken, ob nicht der Einsatz eines einfachen SNMP-Agenten angebrachter ist. Diese Problematik kann unter Umständen dadurch umgangen werden, einen Proxy zu implementieren, der in topologischer Nähe zu dem Managed Object auf einem Server realisiert ist.

Flexibilität

Die Verwendung des Begriffs der Flexibilität ist wie bereits erwähnt, unter mehreren Gesichtspunkten zu verstehen. Einmal kann es die Fähigkeit sein, Funktionalität zu delegieren (wie dies bei den Flexiblen Management Agenten der Fall ist), aber auch die schlichte Erweiterbarkeit von Agenten kann als Flexibilität betrachtet werden. An dieser Stelle wird die erstere Vorstellung betrachtet.
Die Forderung der Hinzunahme und Entfernen von Funktionalität bei Agenten, ohne deren derzeitigen Status zu verändern (z.B. Neustart) kann durch das Konzept der variablen Anzahl von M-Beans innerhalb von JDMK-Agenten als erfüllt angesehen werden. Die Struktur dieser Agenten, einen Rahmen zur Verfügung zu stellen, in welchen beliebige Funktionalität eingebettet werden kann, ermöglicht eine Veränderung von Funktionalität ohne den Agenten bzw. dessen aktuellen Status zu beeinträchtigen.

Push/Pull-Mechanismus

Das JDMK ermöglicht es, Java-Objekte, welche Managementfunktionalität beinhalten, als M-Beans in das CMF des Agenten zu integrieren. Die Integration, die als Registrierung bezeichnet wird, kann entweder durch den Agenten selbst geschehen (PULL) oder durch eine Management-Applikation, welche über einen Adapter mit dem Agenten verbunden ist (PUSH).

Management by Delegation

Wie in der Beschreibung der Management-Umgebung erwähnt, existiert kein eigener Dienst mit spezifischen Methoden. Trotzdem können alle Forderungen dieses Konzepts mit JDMK-eigenen Mitteln erreicht werden. Durch die Möglichkeit, M-Beans beliebiger Funktionalität zu einem Agenten hinzu zunehmen bzw. zu entfernen kann dieser Agent als Delegierter und Delegierer auftreten. Letztere Funktion wird durch die Verwendung des Cascading-Service erreicht.

Kaskadierende Aufrufe

Wie bereits erwähnt, stellt JDMK einen Cascading-Service zur Verfügung. Mit diesem Dienst ist es den Agenten möglich, weitere Agenten zu "beauftragen". Diese Beauftragung erfolgt jedoch dadurch, daß bei Aufbau der Aufrufstruktur im sogenannten "Master-Agenten" ein M-Bean für jeden "Sub-Agenten" integriert wird. Findet nun ein Aufruf eines Sub-Agenten statt, so werden dessen gesamte M-Beans in das CMF des Master-Agenten gespiegelt. Ist dies geschehen, so kann der Master-Agent direkt die M-Beans des Sub-Agenten ansprechen. Diese Vorgehensweise hat in erster Linie das Ziel, die Hierarchie der Agenten zu "verbergen", da der Master-Agent alle M-Beans wie seine eigenen behandeln kann. Ist der Aufruf des Sub-Agenten beendet, werden alle zuvor gespiegelten M-Beans (mit Berücksichtigung der geänderten Properties) wieder in das CMF des Sub-Agenten "zurückgeschrieben" (Die M-Beans sind als Objekte während des gesamten Vorgangs weiterhin im CMF des Sub-Agenten vorhanden).

Kooperationsfähigkeit

Kooperation bedeutet die Fähigkeit eines Agenten, mit weiteren Agenten zur Erfüllung einer Managementoperation zusammenzuarbeiten. Das JDMK bietet keinen expliziten Kooperationsmechanismus, welcher spezielle Methoden zur Verfügung stellt. Es ist jedoch eine Koordination zwischen den Agenten unter Zuhilfenahme des Event-Services und der Möglichkeit, Listener-Objekte zu verwenden, möglich. Allerdings ist dies durch eine benutzereigene Implementierung zu realisieren.
Ist es beispielsweise aufgrund einer Managementaufgabe notwendig, Methoden eines anderen Agenten zu nutzen, so muß dieser über seine Adaptoren (bzw. einen davon) kontaktiert werden und das entsprechende M-Bean, welches die Methode zu Verfügung stellt, ermittelt werden. Danach kann auf die entsprechende Methode zugegriffen werden.

Kontrollmechanismen

Die Kontrollmechanismen werden durch die Bereitstellung des Monitoring-Service ermöglicht. Die Verwendung des Monitoring Service ist in Kapitel 3.7.11 und seine Nähe zum Management-Dienst in 5.1.1 dargestellt.

Module

Die Module, welche zu den Anforderungen unter Kapitel 2.5 gehören, werden durch zur Verfügung gestellte Packages für Adapterklassen etc. durch JDMK vorbereitet und müssen durch den Benutzer erweitert werden. So muß der Benutzer, um das "Kommunikationsmodul" verwenden zu können, ein AdaptorClient-Objekt anlegen und mit diesem unter der Angabe der Portnummer den gewünschen Adapter kontaktieren. Für den Grundstock der Module werden folgende Packages zur Verfügung gestellt:

- Kommunikation : `com.sun.jaw.impl.adaptor`
- Sicherheit : `com.sun.jaw.impl.agent.services.security`
- Information : `com.sun.jaw.iml.agent.services.discovery`
- Kontrolle : `com.sun.jaw.impl.agent.services.monitor`

Zusammenfassend kann festgestellt werden, daß das Java Dynamic Management Kit die Theorie der Flexiblen Management Agenten in wesentlichen Punkten wie Flexibilität und Delegation von Managementfunktionalität erfüllt, jedoch in Forderungen wie Gruppierungsmöglichkeit und Subdomänenbildung keine direkten Mechanismen zur Verfügung stellt. Ein großer Vorteil von JDMK resultiert aus der Java-Implementierung, welche es besonders für Heterogene Netze prädestiniert. Auch die Möglichkeit des entfernten Aufrufs von Managementfunktionen mittels C-Beans verdient besondere Beachtung, da durch diesen Mechanismus die Transparenz eines solchen Systems erhöht wird. Eine abschließende Betrachtung ist in Kapitel 7 zu finden.

5.2 Analyse einer Kombination von JDMK und CORBA

CORBA (Common Object Request Broker) ist derzeit in der Spezifikation 2.0 zur Verwendung freigegeben. Die Freigabe von Spezifikationen wird von der OMG (Object Management Group), einem Konsortium von Software-Herstellern und weiteren Organisationen, geregelt. CORBA dient dazu, Objekte verteilt zu organisieren, sprich den Zugriff auf Objekte über ein ganzes Netz zu ermöglichen. Objekte, die dieser Spezifikation genügen (sogenannte CORBA-Objekte), unterscheiden sich von herkömmlichen (in herkömmlichen Sprachen erzeugte) Objekte in drei wesentlichen Punkten:

- unbeschränkte Verteilung der Objekte innerhalb eines Netzes
- Plattformunabhängigkeit, besonders geeignet in heterogenen Netzen
- Implementierungssprache kann beliebig sein

Greift ein Client auf ein CORBA-Objekt zu, hat er keine Information darüber, wo sich das Objekt befindet, auf welcher Plattform es unter welcher Sprache realisiert ist. Er kann nur eine Referenz auf dieses Objekt erlangen bzw. dessen Methoden erfahren, die dieses Objekt zur Verfügung stellt. Auf der anderen Seite muß sich der Entwickler der CORBA-Objekte keine Gedanken darüber machen, ob er spezielle Schnittstellen er zur Verfügung zu stellen hat, um seine Objekte verschiedenen Clients im Netz verfügbar zu machen.
Die Spezifikationen und Elemente von CORBA sind in [omg91] beschrieben. An dieser Stelle soll auf die speziellen Implementierungen dieser Elemente unter Java und JDMK eingegangen werden.
Die IDL (Interface Description Language) dient dazu, die Schnittstelle eines Servers innerhalb einer CORBA-Umgebung zu definieren. Um die Dienste nutzen zu können, müssen aus einer in dieser Sprache geschriebenen Schnittstelle Stubs und Skeletons in der für die Implementierung von Client oder Server geeigneten Sprache erzeugt werden. Im Falle von Java geschieht dies durch die Package JavaIDL und den mit dieser Package gelieferten Compiler "idltojava". Eine IDL-Datei, wie sie von JDMK für den IIOP-Adaptor zur Verfügung gestellt wird, ist in Anhang D zu finden. Dort werden die generischen Zugriffsmethoden auf M-Beans des Agenten definiert. In Abbildung 5.1 ist eine Verwendung von CORBA unter JDMK dargestellt.
Es gibt vier Möglichkeiten, auf M-Beans, welche innerhalb des CMF eines Agenten integriert sind, zuzugreifen. Diese Möglichkeiten sollen hier untersucht und vorgestellt werden.

5.2.1 Zugriff über AdaptorClient

Der zentrale Punkt dieser Art der IIOP-Kommunikation ist der IIOP-Adapter des JDMK. Dieser Adapter wird mit der Package `com.sun.jaw.impl.adaptor.iiop` zur Verfügung gestellt.

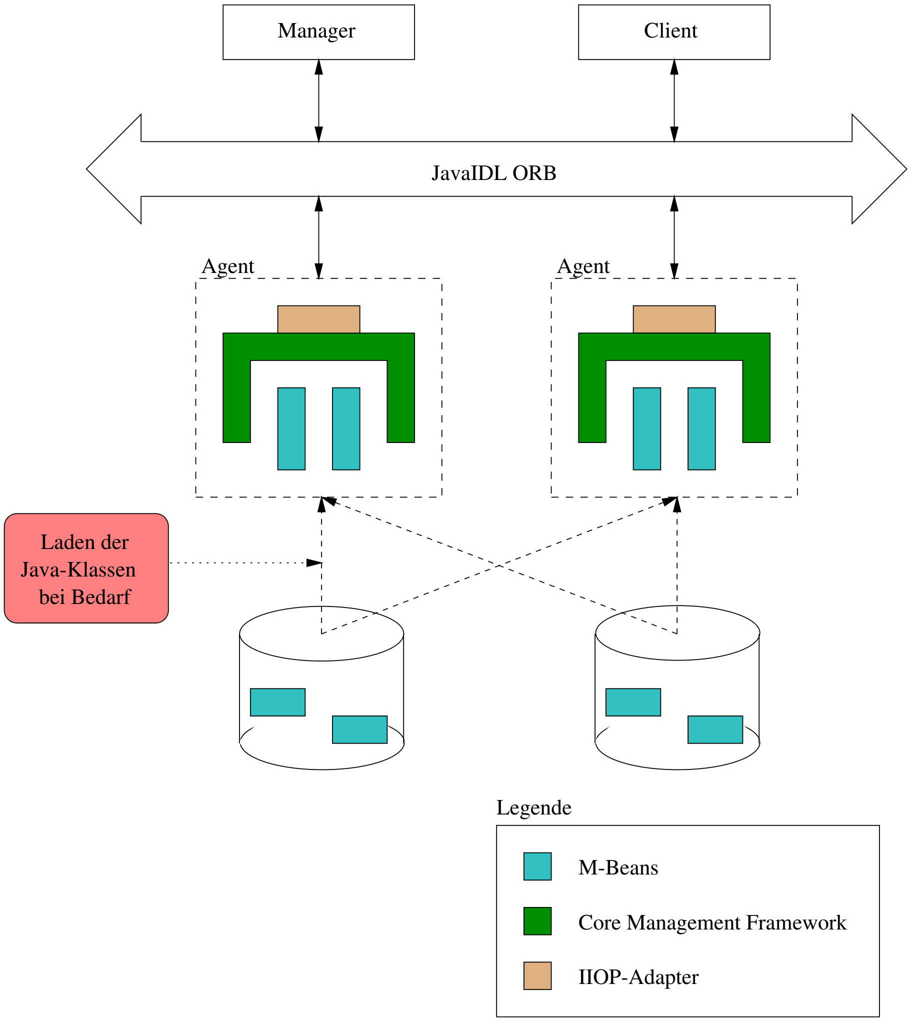

Abbildung 5.1: CORBA-Kommunikation unter JDMK

Diese Package umfaßt mehrere Klassen:

- Interface AdaptorServerImplMO
  Über dieses Interface werden die Methoden, die durch die Klasse AdaptorServerImpl zu Verfügung gestellt werden, bekanntgemacht.

- Klasse AdaptorServerImpl
  Dies ist die eigentliche Adapterklasse. Sie stellt den Server für die generischen Zugriffsmethoden dar. Diese Klasse ist als M-Bean realisiert, sie kann somit dem JDMK-Agenten zur Laufzeit hinzugefügt und wieder entfernt werden.

- Klasse AdaptorServerImplMOStub
  Mittels dieses Stub wird es AdaptorClients ermöglicht, den Adapter anzusprechen. Er

unterstützt Methoden zum Auf- und Abbau einer Adapterverbindung (`connect` bzw. `disconnect`), sowie Kontrollmethoden, die dazu dienen, den Status und die Konfiguration des Adapters abzufragen. Es werden auch Methoden zum Aktivieren und Deaktivieren des Adapters (`performStart` bzw. `performStop`) zur Verfügung gestellt.

- Klasse AdaptorClient
  Diese Klasse wird auf der Managerseite verwendet, um den Agenten über den Adapter zu kontaktieren. Er kann auf dieses Objekt die Methoden des `AdaptorServerImplMOStub`-Objekts anwenden.

Bei der Instantiierung und Registrierung des Adapters kann der Service-Bezeichner, der in den verwendeten Naming-Service eingetragen wird, angegeben werden:

- Port
  Angabe des Ports, unter dem der zu verwendende Naming-Service ansprechbar ist. Wird kein Port angegeben, so wird als Grundeinstellung der Port 8085 gewählt.

- Host
  Angabe des Hosts, auf welchem der verwendete Naming-Service läuft. Wird kein Host angegeben, sie wird als Grundeinstellung der lokale Host verwendet.

Ein kompletter Bezeichner des Adapters, wie er sich in einem Naming-Service darstellt, hat somit folgende Struktur:

```
iiop://<Host>:<Port>/<Host>/com.sun.jaw.impl.adaptor.iiop.AdaptorServerImpl
```

Diese Methode des Zugriffs verdeckt sämtliche CORBA-spezifischen Eigenheiten. Es stellt sich für den Benutzer letztlich gleich dar, ob er über RMI, HTTP oder IIOP mit dem Agenten kommuniziert. Die gesamte Namensauflösung und Verwendung der Schnittstellen-Methoden wird vor dem Benutzer verdeckt und mit dem AdaptorClient erledigt. In Abbildung 5.2 ist ein Zugriff über den AdaptorClient dargestellt.

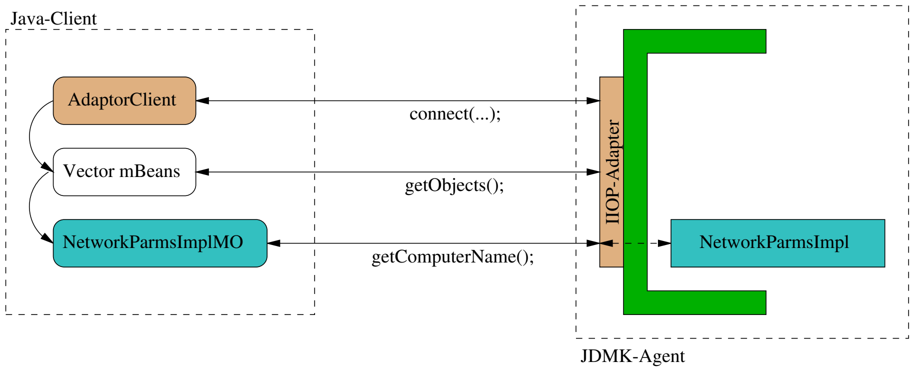

Abbildung 5.2: CORBA-Zugriff über AdaptorClient

5.2.2 Zugriff über CORBA-Client

Die Methoden, die durch den Adapter normalen CORBA-Clients zur Verfügung gestellt werden, sind in der von JDMK bereitgestellten IDL-Datei[3] `AdaptorServer.idl` aufgeführt. Der Inhalt dieser Datei ist unter Anhang D zu finden. Es werden nur einfache `get-` und `set-`Methoden sowie andere Zugriffsmethoden für M-Beans zur Verfügung gestellt. Diese Methoden dienen als Grundmuster, um auf die M-Beans eines Agenten zuzugreifen. Bei Verwendung dieser Methoden ist das anzusprechende M-Bean und die entsprechende Property anzugeben. Dies erfordert jedoch das Wissen über die M-Bean-Klasse. Dies entspricht nicht gänzlich der CORBA-Philosophie, wie sie in der Einleitung dieses Kapitels vorgestellt wurde. Dies bedeutet, daß CORBA-Clients ohne die Verwendung eines AdaptorClient-Objekts nicht über die AdaptorServerImpl-Schnittstelle mit einem JDMK-Agenten kommunizieren können.

5.2.3 Zugriff über eigenständigen CORBA-Server

Eine Möglichkeit den direkten Zugriff durch einen CORBA-Client zu erreichen ist es, einen eigenen CORBA-Server zu entwickeln. Dies bedeutet, daß für ein M-Bean eine eigene IDL-Schnittstelle bereitzustellen ist. Nachfolgend ist ein Ausschnitt aus einer solchen Schnittstelle aufgeführt:

```
module NetworkParmsApp
{
   // ---------------------------------------------------------------
   // typedef declaration
   // ---------------------------------------------------------------
   //
   // String list
   //
   typedef sequence <string>  StringListType;

   //
   // object serialization
   //
   typedef sequence <octet>   SerializedObjectType ;

   //
   // serialized object list
   //
   typedef sequence <SerializedObjectType>   SerializedObjectListType ;

   interface NetworkParms
   {
      // ---------------------------------------------------------------
      // DESCRIPTION
      //     Get value of ComputerName-Property, found in NetworkParms-M-Bean
      //     registered in JDMK-Agent.
      //
      // PARAMETERS
      //     - none
      //
      // RETURN
      //     string (Attribute "ComputerName")
```

[3] Die IDL oder Interface Definition Language ist eine Sprache, mit deren Hilfe Schnittstellen für Server-Objekte und deren angebotene Dienste (Methoden) im Sinne einer CORBA Umgebung geschaffen werden können. In einer ".idl"-Datei, die serverspezifisch erstellt wird, kann mittels Compiler eine Umsetzung der IDL-Schnittstellen-Spezifikation in die verwendete Programmiersprache (z.B. Java) durchgeführt werden. Dabei können Dateien, sogenannte Stubs und Skeletons erstellt werden. Diese Dateien werden für den Zugriff von Clients auf Server benötigt. Genauere Informationen sind unter [omg91] zu finden.

```
      //
      // EXCEPTIONS
      //     - none
      //
      // -------------------------------------------------------------
      string getComputerName();
      // -------------------------------------------------------------
      // DESCRIPTION
      //     Set value of ComputerName-Property, found in NetworkParms-M-Bean
      //     registered in JDMK-Agent.
      //
      // PARAMETERS
      //     - NewComputerName : New Value for ComputerName Attribute
      //
      // RETURN
      //     No  return parameter
      //
      // EXCEPTIONS
      //
      // -------------------------------------------------------------
      void setComputerName(in string NewComputerName);
   };
};
```

Anschließend müssen mittels des "idltojava"-Compilers die erforderlichen Stubs und Skeletons erzeugt werden. Ist dies geschehen, kann nach dem Start des Servers ein CORBA-Client die entsprechenden Einträge im Naming-Service zum Zugriff auf das M-Bean erhalten. Damit kann der Client ein M-Bean, welches im CMF eines Agenten integriert ist, auch direkt ansprechen. Es muß allerdings darauf hingewiesen werden, daß dies unter Umständen Sicherheitslücken öffnet, da der Zugriff unkontrolliert direkt über die Adapter des JDMK-Agenten hinwegläuft. Zudem muß eine Änderung, die ein M-Bean innerhalb des CMF über Adapter erfährt zu dem CORBA-Client propagiert werden, da der automatische Event-Mechanismus, welcher bei JDMK unterstützt wird, hier nicht greift. Die Verhinderung dieses Problem kann sich dadurch lösen lassen, daß die Kommunikation des Servers nicht direkt mit dem M-Bean, sondern über Umwege durch die Adapter des JDMK-Agenten geschieht. Es werden somit zwei Möglichkeiten des Zugriffs ermöglicht. In Abbildung 5.3 ist eine solche Verwendung dargestellt. Ein Vorteil, welcher sich aus der Verwendung eines eigenen Servers ergibt ist die Möglichkeit, mehrere M-Beans in eine Schnittstelle zusammenzufassen und somit logische Management-Einheiten zu bilden.

5.2.4 Zugriff über CORBA-Server als M-Bean

Der Schwachpunkt eines CORBA-Servers, der nicht als M-Bean realisiert ist, ist die Verfügbarkeit. Durch die Realisierung eines Servers als M-Bean hat man den Vorteil, diesen Server bei Bedarf zu starten und wieder zu beenden, indem er dynamisch dem CMF des Agenten hinzugefügt werden kann. Zudem bietet sich die Möglichkeit, die Administration des Servers über die Adapter des JDMK-Agenten durchzuführen. Es ist letztlich zusätzlich mehr Lokalität der Software erreicht. Abbildung 5.4 zeigt die Verwendung eines solchen Servers.

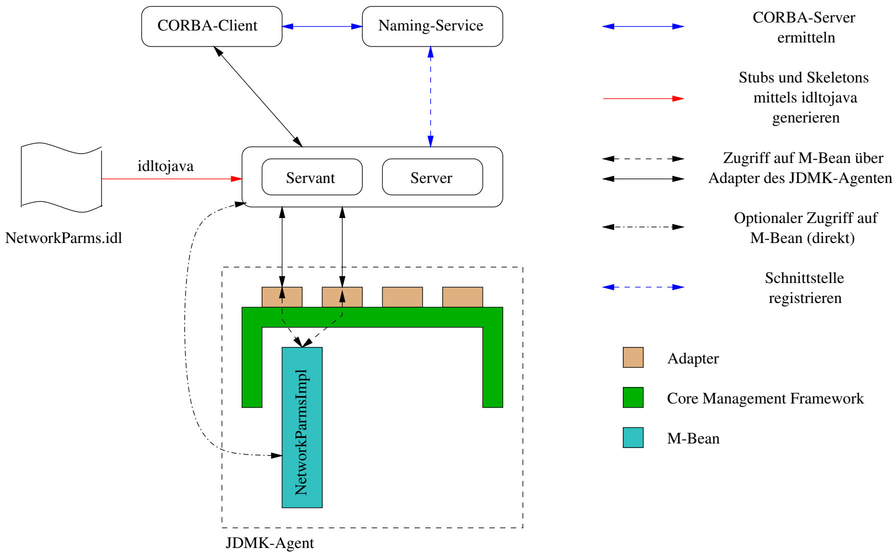

Abbildung 5.3: Ein eigenständiger CORBA-Server

5.3 Konzepte einer TIS-Managementumgebung unter JDMK

5.3.1 Motivation

Die Hauptgründe für die Evaluierung von JDMK für das Management des Telephony Internet Servers (TIS) sind im wesentlichen folgende:

- Flexibilität
  Die Flexibilität der JDMK-Agenten, welche durch die thematische Nähe zu der Theorie der Flexiblen Management Agenten gegeben ist, ist ein wichtiges Kriterium für die Evaluierung. Diese Flexibilität sichert einen universellen Einsatz auf unterschiedlichen Managed Objects, welche verschiedene Ressourcen zur Verfügung haben. So kann ein Agent, an die Erfordernisse und Bedingungen, welche durch begrenzte Ressourcen gegeben sein können, angepaßt werden.

- Sicherheit
  Obwohl JDMK in Sachen Sicherheit gewisse Mängel aufweist (vgl. 3.4.5), bietet es mehr Möglichkeiten, eine sichere Übertragung zu gewährleisten, als SNMP.

- Protokolle
  Durch die Möglichkeit, verschiedene Adapter für einen Agenten gleichzeitig zu verwenden, kann eine breite Palette von Protokollen und deren Vorzüge (Sicherheit, Universalität) genutzt werden. Somit muß die Kommunikation nicht nur auf SNMP beschränkt bleiben, kann aber durch Einsatz des SNMP-Adapters weiterhin beibehalten werden.

- Plattformunabhängigkeit
  Durch die Implementierungssprache Java ist die Möglichkeit zur Verwendung von JDMK

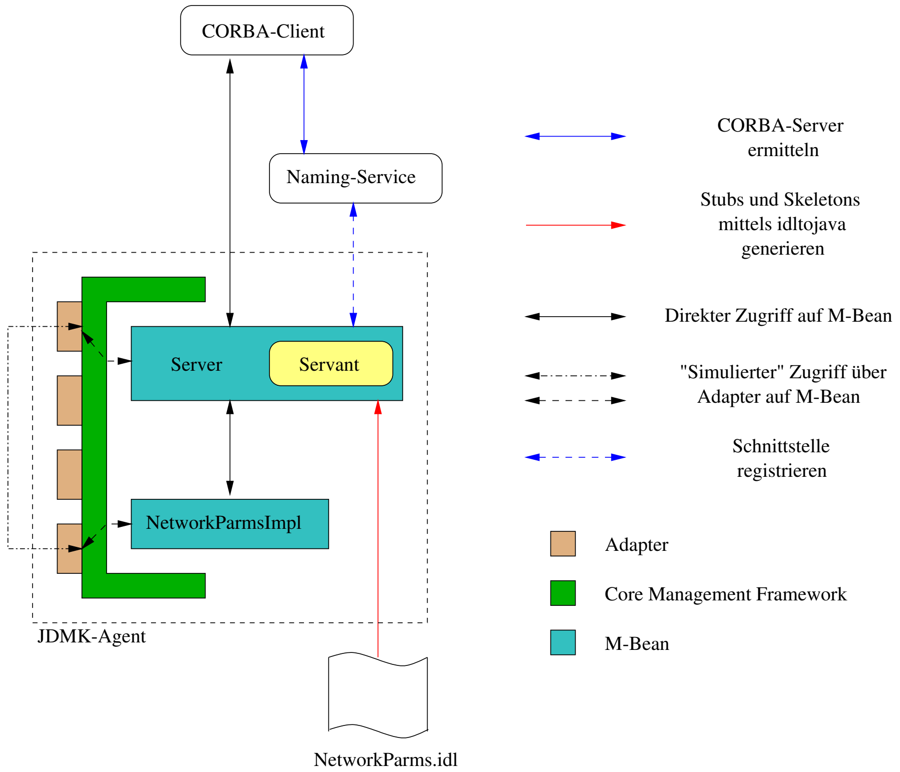

Abbildung 5.4: Ein CORBA-Server als M-Bean

in heterogenen Netzumgebungen geschaffen worden. Dies ist ein wichtiger Punkt beim Einsatz in der von TIS bestimmten Umgebung, da dort unterschiedlichste Komponenten zum Einsatz kommen. Derzeit ist zwar nur ein Agent pro TIS-Box vorgesehen (Prototyp), jedoch ist eine andere Aufteilung der Agentenstruktur durchaus denkbar. Hierbei können die Vorzüge von Java genutzt werden. Dabei ist jedoch stets zu beachten, daß auf den Komponenten, auf welchen ein JDMK-Agent laufen soll, eine Java Virtual Machine zur Verfügung stehen muß (sei es nun ein Proxy oder das Managed Object selbst).

5.3.2 Die Konzepte

Um die verschiedenen Konzepte des Management eines Telephony Internet Servers zu beleuchten ist eine hierarchische Gliederung der Möglichkeiten ratsam. In Abbildung 5.5 ist der Entscheidungsbaum dargestellt.
Zuerst ist eine Entscheidung zu treffen, mit welchem Protokoll oder Nachrichtenformat mit den Komponenten der Managementumgebung kommuniziert werden soll. Innerhalb der TIS-Managementumgebung bieten sich hier zwei Möglichkeiten:

- SNMP
  Diese Variante wäre ein Aufsetzen auf das bisherige Management des TIS. Der Weg der Managementinformation würde über den SNMP-Agenten geschehen, welcher mit dem

TIS-Extension Agent (SXA) erweitert wird. Der JDMK-Agent müsste somit SNMP-Requests erzeugen und diese mit Hilfe des SNMP-Agenten von Windows NT an die Managementumgebung weiterleiten. Der einzige Vorteil dieser etwas umständlichen und ineffizienten Lösung wäre die Realisierung des Agenten allein durch Java. Die Umsetzung in IPC-Nachrichtenformat würde durch den SXA durchgeführt.

- IPC-Nachrichten
  Die zweite Variante ist die Erzeugung von IPC-Nachrichten durch den JDMK-Agenten. Hierbei ergeben sich weitere Unterscheidungskriterien:

  - Realisierung in Java
    Da die Kommunikation innerhalb des TIS über Sockets geschieht, kann durchaus eine pure Java-Implementierung des JDMK-Agenten in Betracht gezogen werden. Mit dieser Variante ist es notwendig, die IPC-Nachrichten in Java zu realisieren und über Sockets mit den Komponenten des TIS zu kommunizieren.
  - Realisierung in C++
    Die Realisierung in C++ bietet den Vorteil, "näher" an der bestehenden Implementierung zu sein und somit Vorteile schon bestehender Implementierungslösungen nutzen zu können. Trotzdem kann hier zwischen der Möglichkeit einer eigenen Implementierung einer Schnittstelle und der Wiederverwendung von bestehendem Code unterschieden werden. Während bei einer eigenständigen Implementierung der gesamte Aufbau einer Schnittstelle selbst realisiert werden müßte, bietet die Wiederverwendung bestehenden Codes große Vorteile, besonders in Hinblick auf die noch kontinuierliche Veränderung der derzeitigen Managementumgebung. Bei der Variante mit Wiederverwendung des bestehenden Codes bietet sich die Möglichkeit, den Wartungs- und Anpassungsaufwand zur Entwicklungszeit zu reduzieren, da die Erzeugung und das Handling der IPC-Nachrichten nicht explizit den Veränderungen angepaßt werden muß.

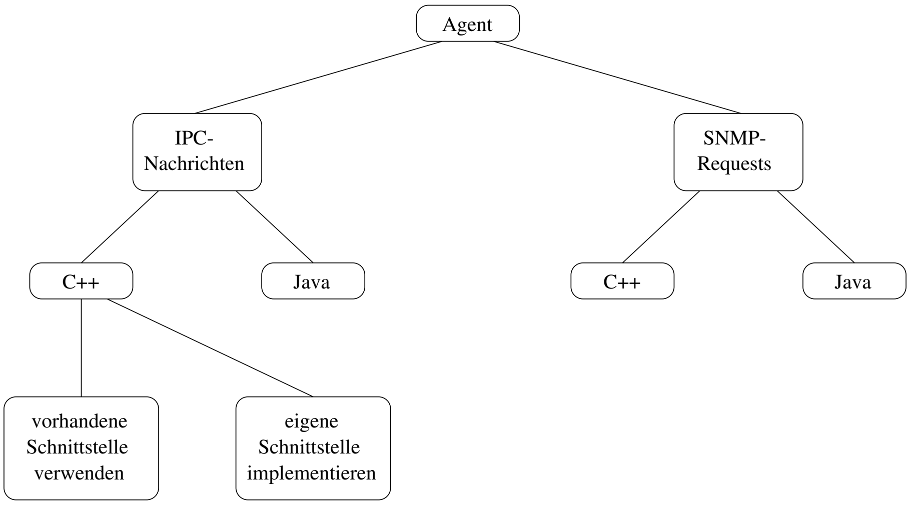

Abbildung 5.5: Konzeptmöglichkeiten des TIS-Managements

Kapitel 6

Konzept eines TIS-Managements unter JDMK

6.1 Motivation

Bei der Auswahl der Managementlösung wurde entschieden, daß der Prototyp IPC-Nachrichten erzeugen sollte, da die Lösung, per SNMP mit den TIS-Komponenten zu kommunizieren eine sehr unpraktikable und ineffiziente Lösung darstellen würde. Einmal wäre die Kommunikation nach wie vor auf den Windows NT SNMP-Agenten angewiesen, zum anderen wäre es reichlich sinnlos, über beispielsweise RMI mit dem JDMK-Agenten zu kommunizieren, um dann die Managementaktion in SNMP umzuwandeln, an den SNMP-Agenten zu senden und diese dann in IPC-Nachrichten umzuwandeln. Des weiteren fiel die Entscheidung über die Verwendung bestehenden Codes, da hiermit der Entwicklungs- und der weitere Anpassungsaufwand reduziert werden konnte. Ein weiteres wichtiges Kriterium war die bereits definierte Managementumgebung, welche durch die TIS-MIB repräsentiert wurde. Durch die vorhandene MIB konnten durch das von JDMK gelieferte Generierungs-Tool "MIBGEN" M-Bean-Klassen aus den Elementen der MIB generiert werden. Die Regeln, nach welchen die Generierung geschieht, sind in Kapitel 3.8.2 beschrieben.
Es wäre ebenfalls denkbar gewesen, eine eigene Implementierung aus der MIB heraus zu entwickeln. Da aber die MIB-Struktur einer laufenden Veränderung unterliegt, war eine generierbare Lösung unbedingt vorzuziehen. Ein Nachteil, welcher aus dieser Wahl erwächst ist die Festlegung des Klassenmodells durch den Generator. Die Art der Klassen und deren Generierung stellte jedoch keine großen Probleme in der weiteren Entwicklung des Prototypen dar. Ein Beispiel für die Generierung der MIB-Gruppe NetworkParms[1] ist in Abbildung 6.1 dargestellt.

6.2 Struktur der konkreten Managementumgebung

Die für das TIS-Management entwickelte Architektur ist in Abbildung 6.2 schematisch dargestellt. Die zentralen Elemente werden nachfolgend beschrieben. Auf die spezifischen Implementierungen und Strukturen wird in den folgenden Kapiteln eingegangen.

[1]Diese MIB-Gruppe ist in der TIS-MIB in Anhang C beschrieben.

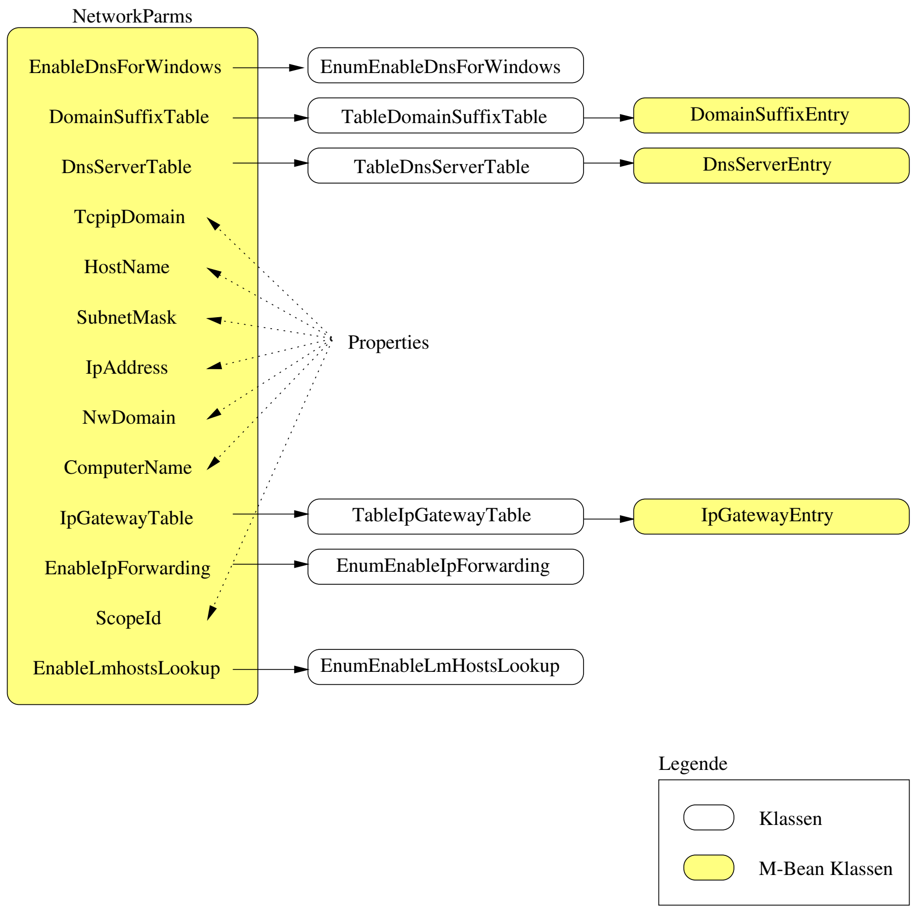

Abbildung 6.1: Generierte Klassen der MIB-Group "NetworkParms"

6.2.1 TIS-Applikationen

Wie bei der bisherigen Management-Umgebung werden die Elemente des TIS durch die zentralen Applikationen AMS (Administration Maintenance Server), Gateway und Gatekeeper bestimmt. Sollen Operationen auf diesen Applikationen ausgeführt werden, so muß eine entsprechende IPC-Message bzw. deren Objekt erzeugt und über die durch SXA[2] bereitgestellten Dispatcher versandt werden. Derzeit wird pro Applikation ein Dispatcher unterstützt. Die Dispatcher bzw. deren Objekt werden innerhalb der SXA-Komponente integriert und bei der Initialisierung von SXA instantiiert. Die Dispatcher sowie weitere MIB-Komponenten werden in dem bereits erwähnten ".mdf"-File spezifiziert.

[2] Im weiteren sei SXA stellvertretend für die SXA-Funktionalität genannt, welche in der nativeBase-Bibliothek realisiert wurde. Diese Bibliothek stellt die unterste Schnittstelle zwischen TIS-Applikationen und JDMK-Managementumgebung dar.

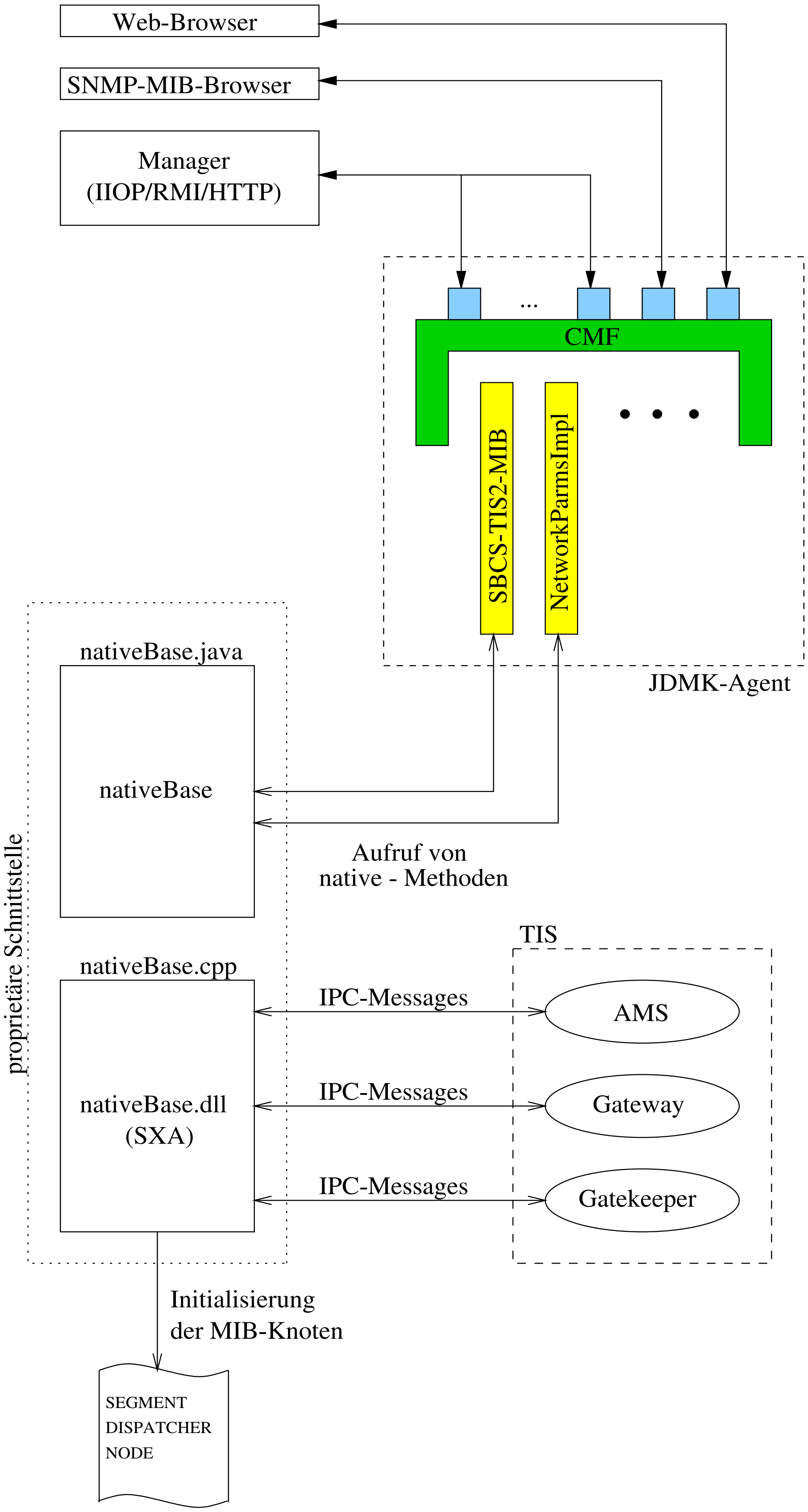

Abbildung 6.2: Übersicht JDMK-Management-Umgebung

6.2.2 Proprietäre Schnittstelle

Da die gewählte Managementlösung beinhaltet, daß der JDMK-Agent die IPC-Nachrichten selbst konstruiert und den Dispatcher für die Versendung beauftragt, ist es erforderlich, eine Schnittstelle zu konstruieren, die den bereits vorhandenen Quellcode für die Konstruktion von IPC-Nachrichten integriert.
Für die Implementierung dieser Schnittstelle wird das Java Native Interface verwendet. Die Schnittstelle hat die gleiche Funktionalität wie die SXA-Bibliothek. Die Implementierung der Native-Klasse ist in Kapitel 6.5 beschrieben.

6.2.3 JDMK-Agent

Der JDMK-Agent, welcher das Management des TIS übernimmt, erhält seine Funktionalität im wesentlichen durch die Erzeugung verschiedener Klassen entsprechend der zu verwaltenden MIB. Die Generierung der M-Beans, sowie deren wesentliche Merkmale werden in Kapitel 6.3 beschrieben. Generell ist zu bemerken, daß alle M-Beans, welche das spätere Management des TIS ermöglichen sollen, mit den durch die Native-Klasse bereitgestellten Methoden auf die Elemente der MIB zugreifen. Zugleich ist der Zugriff über Adapter-Objekte auf die M-Beans möglich.

6.2.4 Manager

Um den Zugriff über die gesamte Breite des durch JDMK möglichen Spektrums zu ermöglichen und zu testen, werden durch den Agenten sämtliche Adapter mit den entsprechenden Protokollen zur Verfügung gestellt. Die Adapter und deren Protokolle werden in Kapitel 3.4 beschrieben.
Das Management bzw. die erforderlichen Aktionen erfolgen durch Java-Manager-Applikationen, HTML-Web-Browsern, SNMP-MIB-Browsern und auch durch Hilfsmittel des JDMK. Eine genaue Beschreibung ist in Kapitel 6.7 aufgeführt. Zu bemerken ist, daß es möglich ist, die HTML-Seiten, welche durch JDMK bereitgestellt werden, zu erweitern.

6.3 M-Bean-Generierung aus TIS-MIB

Die Generierung der M-Beans ist mit Hilfe des MIBGEN-Generators (siehe Kap. 3.12) durchgeführt worden. Es besteht die Möglichkeit, bei der Generierung einen Package-Bezeichner für alle generierten Klassen anzugeben. Dieser wurde auf `tis20.mibgen` gesetzt, um damit eine einfache Abgrenzung zu den später zu implementierenden Klassen zu schaffen, da diese in der Package `tis20.impl` integriert sind.
Die generierten Klassen importieren verschiedene SNMP-Packages. Diese enthalten die benötigten Datenstrukturen für die SNMP-Elemente (SnmpMIB, SnmpOID, ...) sowie Exception-Klassen. Diese Packages sind in [jdmkOnline] beschrieben.

6.3.1 Rahmenstruktur

Die Reihe der Klassen, welche die M-Beans der MIB repräsentieren, beginnt mit einem speziellen "Haupt-"M-Bean. Dieses Bean repräsentiert die oberste Stufe der MIB-Hierarchie. Die Klasse wird mit dem MIB-Bezeichner benannt. Sie stellt eine Unterklasse der SnmpMIB-Klasse

dar, welche durch die oben genannten Pakete bereitgestellt wird. Diese Klasse besitzt zwei Initialisierungsvarianten. Diese Varianten ermöglichen die Instantiierung der MIB innerhalb des JDMK-Agenten mit und ohne Registrierung der M-Beans innerhalb des CMF. Es ist durchaus denkbar, um eventuell die Anzahl der M-Beans zu verringern, auf die Registrierung zu verzichten. Nach wie vor ist ein Zugriff auf die Elemente durch den SNMP-Adapter möglich. Allerdings setzt der Zugriff über die anderen Adaptoren eine Registrierung voraus.
Die Initialisierung verläuft kaskadenartig in der MIB-Hierarchie nach unten. Eine Gruppe nach der anderen wird instantiiert und ggf. im CMF registriert.[3] Innerhalb jeder Gruppe (bzw. Klasse, die die Gruppe repräsentiert) werden zunächst die Properties, welche einfache SNMP-Datenstrukturen darstellen, initialisiert. Danach werden, falls vorhanden, die Tabelleneinträge instantiiert und ggf. im CMF registriert. Es ist zu beachten, daß für die Tabelle selbst kein M-Bean erzeugt wird, sondern die Bezeichner der Tabelleneinträge als Domäne den Tabellenbezeichner erhalten. Nachfolgend ist ein Beispiel für die Instantiierung und Registrierung einer Klasse zu sehen. Es ist bereits die generierte Klasse auskommentiert und durch eine benutzerspezifische Klasse ersetzt worden. Diese Klasse wurde von der generierten Klasse abgeleitet. Das Erstellen von benutzerspezifischen Klassen ist in Kapitel 6.4 beschrieben. Die Methode `registerNode` wird verwendet, um die Klasse bzw. die MIB-Gruppe die sie repräsentiert in die JDMK-MIB-Struktur zu integrieren.

```
// Initialization of the "NetworkParms" group.
// For disabling support of this group, just comment out the section.
//
{
    NetworkParmsMeta meta= new NetworkParmsMeta((SnmpMib)this);
    // generated M-Bean, now obsolete
    //NetworkParms instance = new NetworkParms((SnmpMib)this);
    // The following line obsoletes genarated M-Bean ...
    NetworkParmsImpl instance = new NetworkParmsImpl((SnmpMib)this, cmf);
    // Set Metadata for SNMP-Access ...
    meta.setInstance(instance);
    root.registerNode("1.3.6.1.4.1.2206.5.1.1.1", (SnmpMibNode)meta);
    // The following line obsoletes generated registration line ...
    cmf.addObject(instance
                 , new ObjectName(mibName + ":" + "tis20.impl.NetworkParmsImpl"));
    // generated registration, now obsolete
    //cmf.addObject(instance
                 , new ObjectName(mibName + ":" + "tis20.mibgen.NetworkParms"));
}
```

Wie in dem Beispiel ersichtlich, wird eine Metadaten-Klasse generiert, welche bei Zugriffen über den SNMP-Adapter verwendet wird. Diese Metadaten bzw. die dafür generierten Klassen bieten die notwendigen Informationen, falls über einen MIB-Browser oder SNMP-Agenten auf den JDMK-Agenten zugegriffen wird. In Kapitel 6.3.3 wird auf die Verwendung der Metadaten eingegangen.
Im Falle der TIS-MIB ist es bei der Initialisierung notwendig, statt der von JDMK generierten Defaultwerte mittels Native-Methoden die relevanten Werte aus den entsprechenden Applikationen zu laden.
Der Vorgang der Initialisierung und dessen Verlauf ist in Abbildung 6.3 dargestellt. Der Ablauf der Generierung richtet sich nach folgendem Schema:

[3] Achtung : Es ist unbedingt zwischen der Registrierung innerhalb des CMF (addObject) und der "registerNode"-Methode zu unterscheiden. Letztere erwirkt die Registrierung des MIB-Knotens im JDMK-eigenen MIB-Baum (dieser Baum wird benötigt, um über den SNMP-Adapter auf MIB-Variablen zuzugreifen)

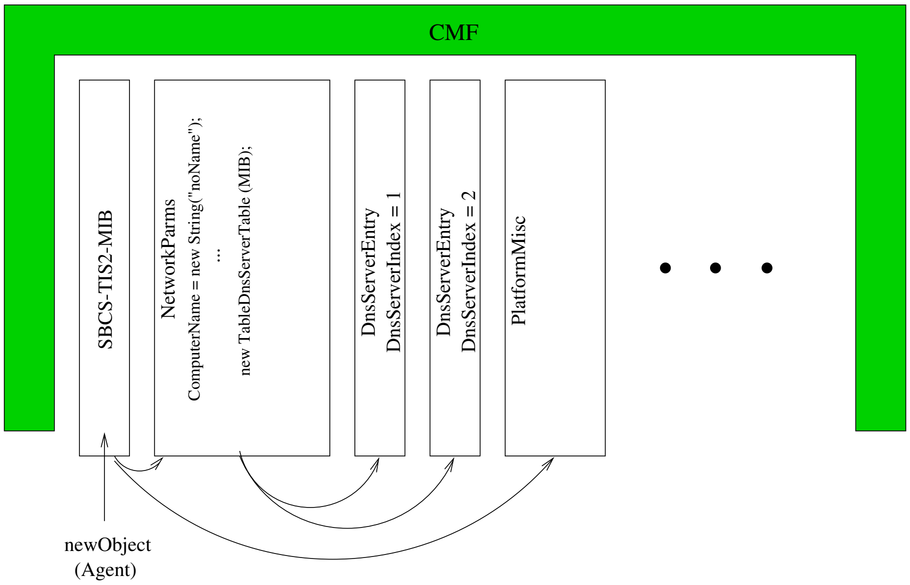

Abbildung 6.3: Erzeugung der MIB-Rahmenstruktur

1. Generierung des "SBCS-TIS2-MIB" M-Bean. Dieses M-Bean stellt den Einstiegspunkt für die MIB-Repräsentation innerhalb des JDMK-Agenten dar.

2. Sukzessive Generierung der M-Beans für jede Gruppe, welche innerhalb der TIS-MIB vorhanden ist. In Abbildung 6.3 ist dies für die Gruppe NetworkParms gefolgt von der Gruppe PlatformMisc dargestellt.

3. Falls in einer Gruppe Tabellen vorhanden sind, werden für die Tabelleneinträge ebenfalls M-Beans erzeugt. (Die Generierung der erforderlichen Klassen ist am Beispiel von NetworkParms in Abbildung 6.1 dargestellt). In Abbildung 6.3 ist eine solche Erzeugung von M-Beans am Beispiel der Tabelle DnsServerTable und deren Einträge dargestellt. Damit besteht die Möglichkeit, auf jeden Eintrag innerhalb einer Tabelle einzeln direkt zuzugreifen.
   Das dynamische Verhalten des Prototypen im Falle von Zugriffen über die verwendeten Adapter ist in Kapitel 6.10 beschrieben.

6.3.2 Umsetzung MIB-Variablen

Der MIBGEN-Generator führt für die verschiedenen SNMP-Datentypen eine Umsetzung durch. Dabei werden drei Arten von Datentypen betrachtet:

- Enumerationen
  Für Enumerationstypen erstellt der Generierungsvorgang jeweils eigene Klassen. Diese werden mit dem Variablenbezeichner, welcher um das Präfix "Enum" erweitert wird, benannt. In diesen Klassen werden mit den möglichen Werten dieser Enumerations-Variablen zwei Hashtabellen erzeugt. Eine String-Tabelle beinhaltet die formulierten Va-

riablenwerte aus dem SYNTAX-Feld, während eine Integer-Tabelle den jeweiligen Index speichert. Es werden Konstruktoren zur Verwendung mit Strings und Integer-Werten, sowie für die Ausgabe der Hashtabellen erzeugt. Nachfolgend ist ein MIB-Enumerationstyp und dessen Umsetzung dargestellt.

```
// Auszug aus der TIS-MIB
enableDnsForWindows OBJECT-TYPE
    SYNTAX  INTEGER { checked(1), notChecked(2) }
    ACCESS  read-write
    STATUS  mandatory
    DESCRIPTION
    "This is the 'Enable DNS' flag as found in
    WinNT Control Panel-> Network->Protocols."
::= { networkParms 9 }

// Umsetzung der Enumerations-Variablen

/**
 * The class is used for representing "EnableDnsForWindows".
 */
public class EnumEnableDnsForWindows extends Enumerated
          implements  Serializable {
  protected static Hashtable intTable= new Hashtable();
  protected static Hashtable stringTable= new Hashtable() ;
  static  {
    intTable.put(new Integer(1), "checked");
    intTable.put(new Integer(2), "notChecked");
    stringTable.put("checked", new Integer(1));
    stringTable.put("notChecked", new Integer(2));
  }
...
  public EnumEnableDnsForWindows(int valueIndex)
       throws IllegalArgumentException {
    super(valueIndex);
  }
...
  public EnumEnableDnsForWindows(String x)
       throws IllegalArgumentException {
    super(x);
  }
```

- Tabellen

  Tabellen werden, wie in Kapitel 6.3.1 erwähnt ebenfalls auf besondere Weise umgesetzt. Es wird pro Tabelleneintrag ein M-Bean erzeugt. Dies hat den Vorteil daß auf einfache Art und Weise auf die einzelnen Tabelleneinträge zugegriffen werden kann. Die Tabelleneinträge werden auf der zur Laufzeit des Agenten generierten HTML-Seite aufgeführt, falls über den HTML-Adapter und einen Web-Browser auf den Agenten zugegriffen wird. So können die Tabelleneinträge als URL direkt angewählt werden. Der Bezeichner des M-Beans setzt sich aus dem Tabellenbezeichner als Domäne, gefolgt vom Variablenbezeichner und einem (bzw. mehreren) Indizes zusammen. Im nachstehenden Source-Code ist ein Beispiel für eine Tabellenumsetzung zu finden.

```
// Auszug aus TIS-MIB
...
dnsServerTable OBJECT-TYPE
    SYNTAX  SEQUENCE OF DnsServerEntry
    ACCESS  not-accessible
    STATUS  mandatory
    DESCRIPTION
    "This is the list of DNS servers as found in
```

```
    WinNT Control Panel-> Network->Protocols."
::= { networkParms 7 }

// Auszug aus Datei "TableDnsServerTable.java"
...
/**
 * The class is used for implementing the "DeviceTable" group.
 * The group is defined with the following oid: 1.3.6.1.4.1.2206.5.1.2.1.1.
 */
public class TableDeviceTable extends SnmpMibTable  implements  Serializable {
...
/**
 * Constructor for the table. Initialize metadata for "TableDeviceTable"
 */
public TableDeviceTable(SnmpMib myMib) {
  super(myMib);
  node= new DeviceEntryMeta(myMib);
}
...
public synchronized  void addEntry(DeviceEntry entry)
  throws SnmpStatusException {
  SnmpIndex index= buildSnmpIndex(entry);
  addEntry(index, entry);
}
...
```

Wie im dem Beispiel angedeutet, werden eine Reihe von Methoden erzeugt, welche den Zugriff auf die Elemente der Tabelle ermöglichen. Die wesentlichen Methoden wären:

- `buildSnmpIndex`
  Diese Methode erstellt ein Objekt der Klasse SnmpIndex. Diese Klasse wird von JDMK in der Package `com.sun.jaw.snmp.agent` bereitgestellt und ermöglicht eine bequeme Handhabung von SNMP-Indizes. Das Index-Objekt wird als Parameter von buildOidFromIndex verwendet.
- `buildOidFromIndex`
  Mit Hilfe dieser Methode wird aus dem mit `buildSnmpIndex` erstellten Tabellenindex ein SnmpOID-Objekt erzeugt. Dieses Objekt beinhaltet die erzeugte OID sowie mehrere Methoden, welche es ermöglichen, die OID unterschiedlich zu repräsentieren. In der Implementierung werden diese Methoden verwendet, um eine Stringdarstellung der OID für den Zugriff über Native-Methoden zu erstellen.
- `addEntry`
  Ermöglicht das Hinzufügen von Tabelleneinträgen innerhalb der JDMK-MIB. Es ist allerdings bei der Implementierung darauf zu achten, daß die Tabelleneinträge zu den Applikationen (AMS etc.) durchgeschrieben werden müssen. Dies geschieht durch Anwendung von Native- Methoden und wird in Kapitel 6.4 beschrieben..
- `removeEntry`
  Analog zum Einfügen von Tabelleneinträgen können diese wieder entfernt werden.

• <u>einfache MIB-Datentypen</u>
Einfache Datentypen wie INTEGER und OCTET STRING werden direkt als Java-Integer bzw. Byte-Array umgesetzt. Je nach Zugriffsmöglichkeit werden get- und set-Methoden generiert. Diese Methoden werden bei der Implementierung um die eigentlichen Zugriffe auf die Applikationen (AMS etc.) erweitert. Es sei erwähnt, daß diese Umsetzung unter Umständen zu Schwierigkeiten führen kann, da z.B. die IP-Adresse

als OCTET STRING gekennzeichnet wurde. Dies schränkt den Wertebereich jedoch auf -127 bis 127 ein und hat zudem zur Folge, daß auf der zur Laufzeit generierten HTML-Seite die Repräsentation der Adresse als Byte-Array dargestellt ist. Hier ist es notwendig eine benutzerspezifische Anpassung der HTML-Seite durchzuführen (vgl. 6.7.1).

6.3.3 Aufbau der SNMP-Struktur unter JDMK

Bei der Initialisierung des JDMK-Agenten werden auch die Elemente der MIB bzw. die daraus generierten Klassen initialisiert und ggf. M-Beans der CMF hinzugefügt. Gleichzeitig wird eine von dem JDMK-Agenten verwaltete MIB-Struktur aufgebaut. Das Hinzufügen erfolgt durch die Methode `registerNode` unter Angabe der OID in String-Repräsentation sowie der Meta-Information für diesen Knoten. Eine Registrierung ist nachfolgend dargestellt:

```
// Initialization of the "PlatformMisc" group.
// For disabling support of this group, just comment out the section.
//
{
      PlatformMiscMeta meta= new PlatformMiscMeta((SnmpMib)this);
      meta.setInstance(new PlatformMisc((SnmpMib)this));
      root.registerNode("1.3.6.1.4.1.2206.5.1.1.4", (SnmpMibNode)meta);
}
```

Die MIB-Struktur, welche vom JDMK-Agenten verwaltet wird, ist schematisch in Abbildung 6.4 dargestellt. Die Abbildung zeigt den Zusammenhang zwischen CMF, den Meta-Objekten für SNMP und den M-Beans.
Ein Ausschnitt der TIS-MIB (Stand 2.11.1998) ist in Anhang C zu finden.

6.4 Implementierung der Rahmenstruktur

Die Ersetzung der generierten Dateien ermöglicht es, benutzerspezifischen Code in die generierte Umgebung einzubringen. Da die noch nicht erweiterten Klassen, welche aus der MIB generiert wurden, lauffähig (wenn auch mit Dummy-Werten) sind, kann der Agent bereits nach der Generierung gestartet werden. Die Elemente, welche (noch) nicht gemanagt werden sollen, können durch einfaches Auskommentieren vorerst aus dem Verantwortungsbereich des Agenten herausgenommen werden. Auch ist es denkbar, daß verschiedene Beans auf Agenten verteilt werden.

6.4.1 Standardmethoden

Bei der Generierung der Klassen werden zwei Varianten für die Initialisierung der MIB-M-Beans verwendet. Eine Initialisierung baut lediglich die SNMP-MIB-Struktur innerhalb des JDMK-Agenten auf, ohne die dabei generierten M-Beans innerhalb des Core Management Frameworks zu registrieren. Diese Art der Initialisierung wird dann verwendet, falls der JDMK-Agent als Standalone SNMP-Agent agieren soll. Es ist aber dann nicht möglich, über andere Adapter auf diese generierten M-Beans zuzugreifen. Die andere Variante der Initialisierung registriert die generierten M-Beans zusätzlich und ermöglicht so die Nutzung der von JDMK bereitgestellten Dienste und Protokolle (es ist zu beachten, daß bei der ersten Variante ohne Registrierung die Dienste des JDMK nicht genutzt werden können).

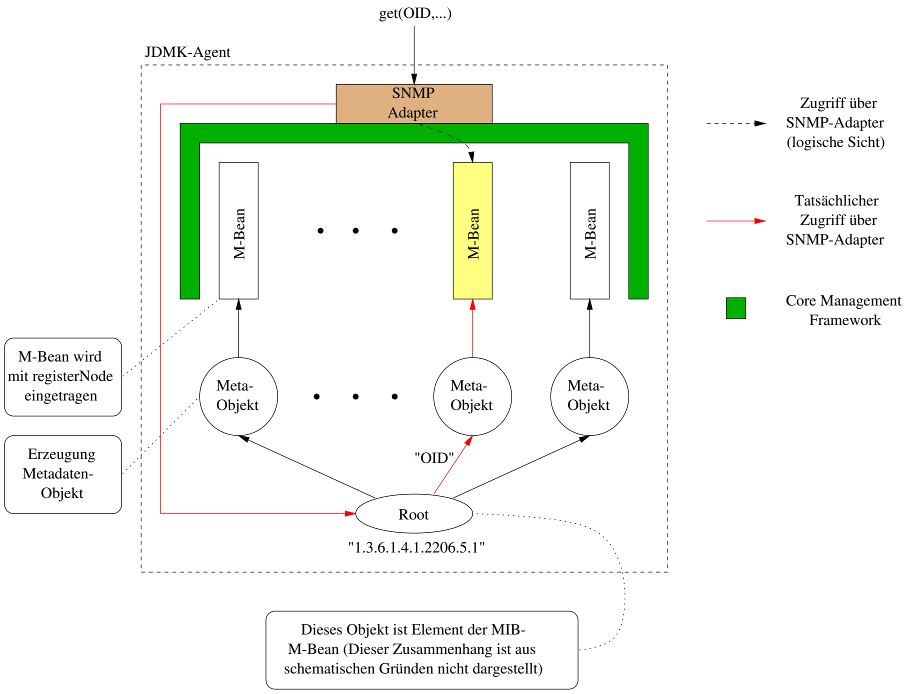

Abbildung 6.4: Zugriff über SNMP-Adapter

```
// Konstruktor einer Implementierung des generierten M-Beans
// NetworkParms. Es stellt eine Gruppe innerhalb der MIB dar.

public NetworkParmsImpl(SnmpMib myMib, Framework cmf) {
   super(myMib);
   init(myMib, cmf);

    // Get these Objects to insert and remove Tableentries
   NetworkParmsMIB = myMib;
   NetworkParmsCMF = cmf;
}
```

Bei der Initialisierung werden die einzelnen Properties der Klasse (bzw. des M-Beans) mit Werten aus den TIS-Applikationen, welche mit dem entsprechenden M-Bean gemanagt werden sollen, belegt. Die Properties werden innerhalb der Superklasse (also der generierten) festgelegt. Dies sind einfache MIB-Variablen wie OCTET STRING und INTEGER:

```
...
// Festlegung einer OCTET STRING - Variablen
// SYNTAX : DisplayString
/**
 * Variable for storing the value of "ComputerName".
 * The variable is identified by: "1.3.6.1.4.1.2206.5.1.1.1.1".
 */
protected String ComputerName= null;
...
// Festlegung einer OCTET STRING - Variablen
// SYNTAX : DisplayString
```

```
/**
 * Variable for storing the value of "IpAddress".
 * The variable is identified by: "1.3.6.1.4.1.2206.5.1.1.1.3".
 */
protected Byte[] IpAddress = null;
...
// Festlegung der Tabellen
/**
 * Variable for storing the value of "DnsServerTable".
 * The variable is identified by: "1.3.6.1.4.1.2206.5.1.1.1.7".
 */
protected TableDnsServerTable DnsServerTable;
```

Wie in diesem Beispiel-Code dargestellt, wird eine Tabelle nur als grobe Rahmenstruktur zur Verfügung gestellt, welche Methoden zum Zugriff auf seine Tabelleneinträge zur Verfügung stellt, da die eigentliche Information in den Einträgen vorhanden ist. So werden Klassen, welche den Tabellenbezeichner, gefolgt von dem Suffix "Entry" tragen, erzeugt. Diese Klassen müssen mit den implementierten Klassen erweitert werden, um zum Beispiel eine Möglichkeit zum Entfernen von Tabelleneinträgen zu schaffen. Es muß im Sinne der kaskadierenden Erstellung der M-Beans (siehe Kap. 6.3.1) eine Speicherungsstruktur geschaffen werden. Zudem ist für das Einfügen und Entfernen von Tabelleneinträgen eine Referenz auf die MIB (Eintrag in MIB-Struktur des JDMK-Agenten) und das Core Management Framework (Registrierung des M-Beans) zur Verfügung zu stellen. Aus diesem Grund werden weitere Membervariablen zu den Implementierungsklassen hinzugenommen. Nachfolgend ist ein Beispiel für solche Variablen zu sehen:

```
// Zusaetzliche Membervariablen bei Implementierungen
...
//
// Private variables.
//
private SnmpMib NetworkParmsMIB;
private Framework NetworkParmsCMF;
private Vector NetworkParmsImplListeners = new Vector();

private static DnsServerEntryImpl DnsServerEntry[]
        = new DnsServerEntryImpl[10];
private static DomainSuffixEntryImpl DomainSuffixEntry[]
        = new DomainSuffixEntryImpl[10];
private static IpGatewayEntryImpl IpGatewayEntry[]
        = new IpGatewayEntryImpl[10];
...
```

Da in der generierten Klasse die Elemente der MIB-Gruppe mit Dummy-Werten vorbelegt werden, müssen in der Initialisierungsphase die realen Werte ermittelt werden. Dabei wird zwischen tabellen- und nicht-tabelleneigenen Einträgen sowie Enumerationen unterschieden:

- Tabelleneinträge
  Hier wird eine Methode `init<Klassenbezeichner Tabelle>` mit Übergabe eines MIB- und CMF-Objektes aufgerufen. Diese Methode beinhaltet die notwendigen OID's für die Spalten der Tabelle als Integer- und String-Array. Diese Zweiteilung, welche nur bei der Initialisierung von Tabellen benutzt wird, hat hier den Vorteil daß bei den Native-Methoden wie `getNextStringNative` bzw. `getNextIntNative`[4] die nächste OID wieder

[4] Diese Methoden sind Bestandteil der Schnittstelle des JDMK-Agenten, welche mit Hilfe des Java Native Interface (vgl. Kapitel 6.5) implementiert wurde. Sie dienen zum Durchlaufen von MIB-Tabellen.

zurückgegeben wird. An dieser Stelle würde die Konvertierung der Daten unnötige Probleme aufwerfen. Die String-Variante wird üblicherweise verwendet, insbesondere da JDMK entsprechende Packages zur Verfügung stellt (siehe Kap. 6.3), die diese Verwendung nahelegen. Innerhalb dieser Initialisierungsmethode werden nun die Einträge der Tabelle spaltenweise abgearbeitet. Die String-Variante der OID-Darstellung dient hier bei einem ersten Durchlauf zur Ermittlung der Anzahl der Einträge.[5] Nachfolgend ist ein Beispiel einer solchen Initialisierungs-Methode zu sehen:

```
private void initDnsServerTable(SnmpMib myMib, Framework cmf) {
   //Loading the DnsServerTable
   //to initialize, it's better to use the oid as an integer-array
   int[][] oid = {{1,3,6,1,4,1,2206,5,1,1,1,7,1,1}
                 , {1,3,6,1,4,1,2206,5,1,1,1,7,1,2}};
   String[] StartOid = {"1.3.6.1.4.1.2206.5.1.1.1.7.1.1"
                       , "1.3.6.1.4.1.2206.5.1.1.1.7.1.2"};

   int entries = 0;
   int NrOfRows;
   int tempInt = 0;
   DnsServerEntry[entries]
        = new DnsServerEntryImpl(myMib, cmf, DnsServerTable);
   boolean NoMoreRows = false;

   // Get the DnsServerIndex
   try {
       tempInt = nativeBase.getNextIntNative(oid,0);
      DnsServerEntry[entries].setDnsServerIndex(new Integer(tempInt));
      NoMoreRows = (StartOid[0].equals(tools.ConvertTools.Int2String(oid)[0]));
   } catch (Exception e) {
      e.printStackTrace();
   }

   while (!NoMoreRows) {
      entries++;
      DnsServerEntry[entries] = new DnsServerEntryImpl(myMib);
      try {
         tempInt = nativeBase.getNextIntNative(oid,0);
         DnsServerEntry[entries].setDnsServerIndex(new Integer(tempInt));
         NoMoreRows = (StartOid[0].equals(tools.ConvertTools.Int2String(oid)[0]));
      } catch (IllegalArgumentException iae) {
      // no more rows !??
         NoMoreRows = true;
      } catch (Exception e) {
         e.printStackTrace();
      }
   }

   NrOfRows = entries;

   // Get the DnsServerAddr
   // inputOID now at first Element of second column
   entries = 0;
   while (entries != NrOfRows) {
      String tempDnsServerAddr = null;
      try {
         tempDnsServerAddr = new String(nativeBase.getNextStringNative(oid,1));
         DnsServerEntry[entries]
        .setDnsServerAddr(tools.ConvertTools.String2Byte(tempDnsServerAddr));
      } catch (Exception e) {
```

[5] Hier sei darauf hingewiesen, daß die SNMP-Tabellen unter TIS nicht im herkömmlichen Sinne durchlaufen werden können. Ist eine Spalte abgearbeitet, so ergibt der nächste "getNext"-Aufruf wieder die Start-OID der Tabelle. Aus diesem Grund ist auch ein Vergleich mit der String-Variante der OID notwendig

```
            e.printStackTrace();
         }
         entries++;
      }

      entries = 0;
      while (entries != NrOfRows) {
         try {
            //Register in DnsServerTable
            DnsServerTable.addEntry(DnsServerEntry[entries]);
         } catch (Exception e) {
            e.printStackTrace();
         }
         try {
            //Register in Core management Framework
            DnsServerEntry[entries].addInCmf(cmf);
         } catch (Exception e) {
               e.printStackTrace();
         }
         entries++;
      }
   }
```

- Nicht-Tabelleneinträge
  Bei diesen Einträgen werden durch einfache `getStringNative-` und `getIntNative-`Methoden die entsprechenden Werte ermittelt. Die OID wird hier als String-Variante verwendet. Ein Konvertierung in ein Integer-Array ist dadurch notwendig geworden, da innerhalb des Java Native Interface die Datenstruktur `RFC1157Varbind` (vgl. 6.5) die OID als Integer-Array voraussetzt. Um aber das Interface möglichst schmal zu halten, wurde die Konvertierung auf der Java-Seite durchgeführt. Das folgende Codestück stellt eine solche Methode dar:

```
/**
* Getter for the "ComputerName" variable.
*/
public String getComputerName() throws SnmpStatusException {
   String oid = "1.3.6.1.4.1.2206.5.1.1.1.1.0";
   ComputerName = nativeBase.getStringNative(tools.ConvertTools.String2Int(oid));
   return ComputerName;
}
```

- Enumerationen
  Da Enumerationen als eigene Klassen generiert werden (siehe Kap. 6.3.1), müssen auch diese Einträge gesondert behandelt werden. Es werden zwar ebenso `get-` und `set-` Methoden verwendet, jedoch ist der Inhalt davon geprägt, den Wert, welcher von der Applikation erhalten wird, auf das Enumerations-Objekt abzubilden. Dies geschieht mittels einer `switch`-Anweisung. Eine `get`-Methode kann somit folgendermaßen aussehen:

```
/**
 * Getter for the "EnableDnsForWindows" variable.
 */
public EnumEnableDnsForWindows getEnableDnsForWindows()
        throws SnmpStatusException {
   String oid = "1.3.6.1.4.1.2206.5.1.1.1.9";
   switch (nativeBase.getIntNative(tools.ConvertTools.String2Int(oid))) {
   case 1:
      EnableDnsForWindows = new EnumEnableDnsForWindows(1);
      break;
   case 2:
```

```
            EnableDnsForWindows = new EnumEnableDnsForWindows(2);
            break;
         default:
            EnableDnsForWindows = new EnumEnableDnsForWindows();
         }
         return EnableDnsForWindows;
      }
```

Ebenso wie die `get`-Methoden, werden auch `set`-Methoden zur Verfügung gestellt. Es wird ebenso die OID als String verwendet, die Änderung erfolgt durch Anwendung der Native-Methode `setNative`. Fehlerfälle (Exceptions) können durch das Einfügen der Werte in die JDMK-MIB-Struktur und bei Änderungen der Werte in den Applikationen entstehen. Diese Fehlerfälle werden bis jetzt mit Ausgabe des Trace-Stacks abgefangen.

6.4.2 Benutzerdefinierte Methoden

Diese Methoden hängen von der Verwendung der unter JDMK verfügbaren Dienste ab und werden vom Entwickler bei der Nutzung von JDMK-Diensten implementiert. Die Verwendung der Dienste mit Darstellung von Beispielen wird in Kapitel 6.6 dargestellt. Diese Methoden umfassen die Möglichkeit, Events bei Änderung der Properties zu versenden, innerhalb der von JDMK-basierten HTML-Darstellung (siehe Kap. 6.7.1) Erweiterungen vorzunehmen und auch Listener-Objekte (siehe Kap. 6.6.1) hinzu zunehmen. Zusätzlich können benutzerdefinierte Methoden den M-Beans hinzugefügt werden. Diese sogenannten "Aktionen", welche durch das Präfix "perform" gekennzeichnet sind, können für Managementaktionen wie Reboot, Restart und Shutdown und Andere implementiert werden. Nachfolgend ist ein Beispiel für die Erweiterung einer Funktionalität innerhalb der HTML-Darstellung zu sehen (Es wird ein Button erzeugt, welche bei Angabe der im Parameterteil übergebenen Werte einen Tabelleneintrag in der DnsServer-Tabelle durchführt) :

```
public void performCreateDnsServerEntry(int NewDnsServerIndex, String NewDnsServerAddr)
{
   String[] oid = {"1.3.6.1.4.1.2206.5.1.1.1.7.1.1", "1.3.6.1.4.1.2206.5.1.1.1.7.1.2"};
    DnsServerEntryImpl DnsServerEntry =
   new DnsServerEntryImpl(NetworkParmsMIB, NetworkParmsCMF, DnsServerTable);
    try {
      int[] LegalDnsServerIndex = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};

      // Check index range
      if (!tools.ArrayTools.isElement(NewDnsServerIndex, LegalDnsServerIndex)) {
         throw new ArrayIndexOutOfBoundsException();
     }
      // Set the M-bean properties
      DnsServerEntry.setDnsServerIndex(new Integer(NewDnsServerIndex));
      DnsServerEntry.setDnsServerAddr(tools.ConvertTools.String2Byte(NewDnsServerAddr));

      // get the Index and oid
      SnmpIndex testIndex = DnsServerTable.buildSnmpIndex(DnsServerEntry);
      SnmpOid testOid = DnsServerTable.buildOidFromIndex(testIndex);

      // concatenate base-oid with index
      oid[0] = oid[0] + "." + testOid.toString();
      oid[1] = oid[1] + "." + testOid.toString();

      // convert in useable format for the native interface
      int[][] tempOid = tools.ConvertTools.String2Int(oid);

      // set values in ams
      nativeBase.setNative(tempOid[0], NewDnsServerIndex);
```

```
      nativeBase.setNative(tempOid[1], NewDnsServerAddr);

      //Register in PlatformMiscUserTable
      System.out.print("Register entry in DnsServerTable ...");
      DnsServerTable.addEntry(DnsServerEntry);
      System.out.println(" done.");

      //Register in Core Management Framework
      System.out.print("Register entry in CMF ...");
      DnsServerEntry.addInCmf(NetworkParmsCMF);
      System.out.println(" done.");
   } catch (SnmpStatusException snmpe) {
      System.out.println("Index already used, please check entries for free index!");
   } catch (ArrayIndexOutOfBoundsException aioobe) {
      System.out.println("Wrong index (must be between 1..15), please retry");
   } catch (NullPointerException npe) {
      System.out.println("Entries not complete, please retry");
   } catch (Exception e) {
      e.printStackTrace();
   }
}
```

In Anhang E ist ein Beispiel einer Implementierung der GKStatistics-Gruppe[6] dargestellt.

6.5 Aufbau des Java Native Interface für SXA

6.5.1 Möglichkeiten der Schnittstellendefinition

Der Grund für die Verwendung des Java Native Interface war, den Nachrichtenaustausch zwischen den TIS-Komponenten (Admin-Server, Gateway, Gatekeeper) mittels der für diese Umgebung entwickelten IPC-Messages zu realisieren. So mußte eine Schnittstelle geschaffen werden, welche zum einen die Funktionalität des SNMP-Extension Agent (SXA), welcher in der bisherigen Managementumgebung verwendet wird, bietet, andererseits eine leichte Handhabung auf der Seite des JDMK-Agenten ermöglicht. Die Schnittstelle zu den TIS-Komponenten mußte zudem möglichst schmal ausgelegt werden, da aufgrund der häufigen Änderungen, die während der Entwicklungszeit auftraten, eine häufige Anpassung des C++-Source-Codes erforderlich machte. Generell bieten sich für die Kommunikation zweierlei Varianten an:

- <u>Kommunikation über SNMP</u>
  Bei dieser Variante wird die Kommunikation über den SNMP-Agenten geführt, welcher die SXA.DLL, welche bis jetzt verwendet wird, weiterhin beinhaltet. Die Kommunikation würde dann auf Seite des JDMK-Agenten über SNMP-Aufrufe laufen, die Umsetzung der SNMP-Aufrufe würde dann die Java Native Interface Schnittstelle übernehmen. Eine solche Art der Kommunikation ist in Abbildung 6.5 zu sehen. Wie aus der Abbildung ersichtlich wird, ist eine derartige Realisierung nicht nur sehr ineffektiv und umständlich, sondern erfordert zudem die Verfügbarkeit des Windows NT SNMP-Agenten. Den einzigen Vorteil, den diese Variante bietet ist, daß die C++-Seite des TIS vollständig gekapselt sein kann und der JDMK-Agent keine Eingriffe in den C++-Source-Code durchführen muß.

[6] In dieser MIB-Gruppe werden Werte für die Überwachung der aktuellen Verbindungen, sowie der im Moment verfügbaren Bandbreite für den Aufbau neuer Verbindungen gespeichert. Zusätzlich wird eine Tabelle verwaltet, in welcher in Abständen Vergleichswerte für die Anzahl der aktuellen Verbindungen, die gesamte Verbindungsanzahl, die Durchschnittslänge der Verbindungen und die maximale Bandbreite, welche bisher belegt wurde, gespeichert.

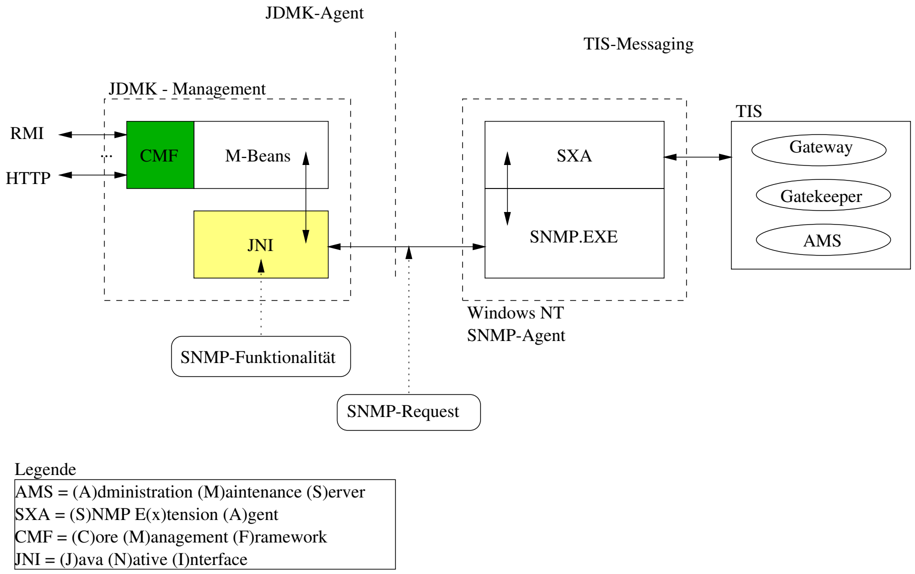

Abbildung 6.5: Kommunikation JDMK über SNMP-Requests

- Kommunikation über IPC-Nachrichten
  Diese Variante ermöglicht es dem JDMK-Agenten, selbst IPC-Nachrichten zu generieren und diese an die entsprechenden Applikationen zu versenden. Dabei kann C++-Source-Code des SXA verwendet werden[7] . Dabei werden die Managementaktionen, welche auf den M-Beans des JDMK-Agenten ausgeführt werden, in IPC-Nachrichten verpackt. Es ist notwendig, daß bei Start des Agenten oder zumindest vor der erstmaligen Benutzung des Interfaces (erste Managementaktion) eine Initialisierung (des Interfaces) durchgeführt wird. Diese Initialisierung besteht in dem Durchlaufen der in Kapitel 4.7 dargestellten ".mdf"-Datei, die die Management-Knoten enthält und dem simultanen Aufbau einer Baumstruktur, welche diese Knoten darstellt[8]. Wenn nun ein Request auf eine MIB-Variable ausgeführt wird, so bildet die Schnittstelle die übergebene OID auf den entsprechenden Knoten (Dateityp CAsnNode, (vgl. Kapitel 4.7) ab und kann so die Struktur der erforderlichen IPC-Nachricht ermitteln. Mit der Ausführung der Methode `BuildVarBind` wird schließlich der Request mit Übergabe der jeweiligen IPC-Nachricht ausgeführt und auf Antwort gewartet.
  Die Integration der SXA-Funktionalität in das JNI-Interface ist in Abbildung 6.6 dargestellt. Obwohl diese Variante einen Mehraufwand in der Implementierung bedeutet, ist sie der ersten unbedingt vorzuziehen. Zum einen kann ein Großteil des Programm-Codes wiederverwendet werden und zum anderen ist nicht die Präsenz des Windows NT

[7] Wie bereits in Kapitel 5.3.2 erwähnt, geschieht die Kommunikation innerhalb des TIS über Sockets. Es kann somit auch eine Realisierung in Java geben. Aus bekannten Gründen wurde jedoch die C++-Variante gewählt.

[8] Diese Baumstruktur ist nicht mit der durch den JDMK-Agenten verwalteten Baumstruktur zu verwechseln. Die Baumstruktur des JDMK-Agenten dient zum Zugriff über den SNMP-Adapter während die Baumstruktur, die durch die ".mdf"-Datei erzeugt wird, für das Zusammensetzen der IPC-Nachrichten erforderlich ist

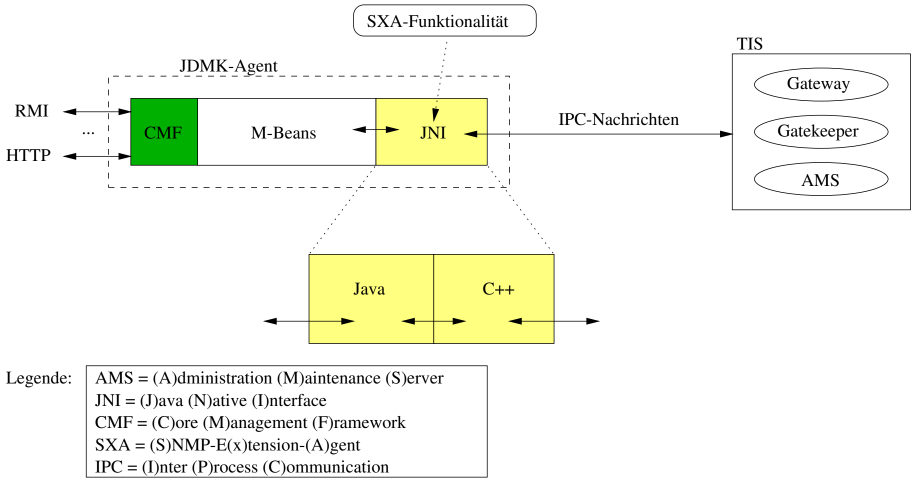

Abbildung 6.6: Kommunikation JDMK über IPC-Nachrichten

SNMP-Agenten notwendig. Auch ist die Kommunikation nicht auf SNMP beschränkt.

6.5.2 Möglichkeiten der Schnittstellenimplementierung

Ist die Definition der Schnittstelle durchgeführt[9], ist die Art der Implementierung festzulegen. Es müssen hierbei zwei Möglichkeiten berücksichtigt werden:

- Eine Schnittstelle pro M-Bean (eines Agenten)
  Eine Möglichkeit, die Schnittstelle zu implementieren ist die Realisierung einer M-Bean-eigenen Schnittstelle. Diese könnte spezifisch auf das M-Bean zugeschnitten sein. Dies hätte den Vorteil, daß Konvertierungen, die bei einem generischen, zentralen Ansatz berücksichtigt werden müssten, weitestgehend reduziert werden können. Ebenso wird die Größe der Schnittstelle und somit unter Umständen der Speicherplatzbedarf einer Bibliothek, die diese Schnittstelle darstellt, auf ein Minimum reduziert. Auch kann damit erreicht werden, daß nur Objekte, also Klassen, welche eine Schnittstelle benötigen, diese auch erhalten. Andere Objekte bleiben davon unberührt. Allerdings ist dieser Ansatz nur in Erwägung zu ziehen, falls die Zahl der Klassen, welche für eine Schnittstelle vorgesehen sind, auf ein Mindestmaß reduziert werden kann. Es wird sonst eine Anpassung, die sich bei Veränderung der Entwicklungsumgebung ergibt, zu umfangreich ausfallen. Die Implementierung einer solchen Lösung ist in Abbildung 6.7 dargestellt.

- Eine gemeinsame Schnittstelle aller M-Beans (eines Agenten)
  Der Aufwand für Änderungen ist bei zentralen, globalen Schnittstellen gegebüber dem lokalen Ansatz wesentlich geringer. Dies ist einer der größten Vorteile dieses Ansatzes. Der Nachteil dieser generischen Ansätze ist die Schwierigkeit, auf Sonderfälle, welche immer bei Implementierungen berücksichtigt werden müssen, einzugehen. Hier ist es

[9] Die Wahl der Schnittstellendefinition fiel bei der Implementierung des Prototypen auf die eigenständige Erzeugung von IPC-Nachrichten und somit der Umgehung des SNMP-Agenten.

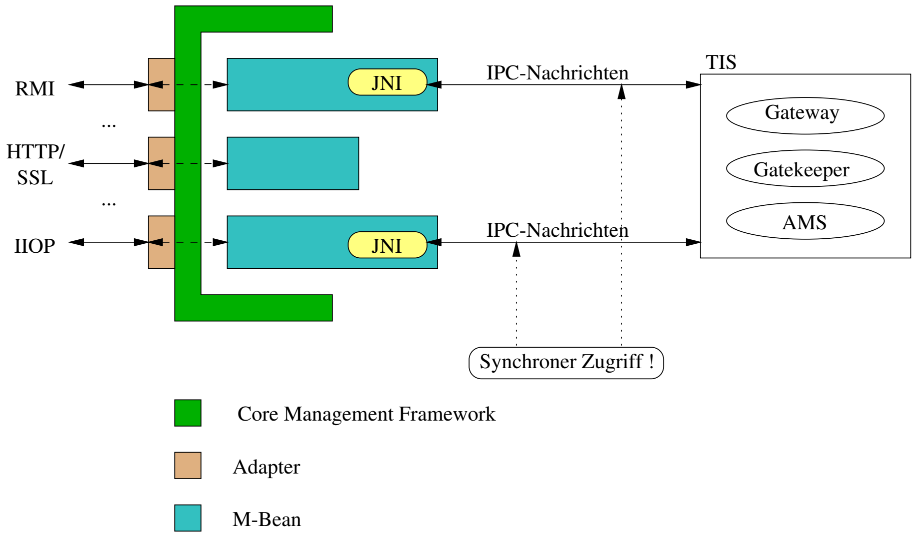

Abbildung 6.7: Eine lokale Schnittstelle pro M-Bean

unter Umständen schwierig, den generischen Ansatz weiterzuverfolgen. Da in der Implementierung des Prototypen und der damit einhergehenden Wahl der M-Beans für die darzustellende MIB eine Vielzahl von Klassen (M-Beans) erzeugt werden, war ein generischer Ansatz vorzuziehen. Ein Beispiel für das Zusammenwirken der Komponenten bei einem Zugriff über die JNI-Schnittstelle ist in Abbildung 6.8 zu sehen. Wie in der Abbildung ersichtlich, ist es absolut notwendig, ein zentrales Objekt für die Schnittstelle zu verwalten. Dies resultiert aus der Tatsache, daß die Baumstruktur die MIB-Knoten (CAsnNode-Klasse) bzw. deren aktuellen Zustand für alle M-Beans gleich zur Verfügung stellt. Ein Zugriff mehrerer M-Beans gleichzeitig ist durch die Struktur von M-Beans, wie sie von JDMK verwendet werden, auszuschließen. Der Grund für diese Annahme besteht darin, daß die Generierung eines Threads, welcher die Asynchronität ermöglicht, eine Klasse, die von Runnable abgeleitet wurde, voraussetzt. Dies ist bei M-Beans nicht der Fall. Falls dies zusätzlich gewünscht ist, sind zusätzliche Maßnahmen wie die Verwendung von `synchronized` bei den Native-Methoden erforderlich[10].

6.5.3 Abbildung OID zu MIB-Knoten

Die Abbildung der OID, welche den Native-Methoden übergeben wird, bedarf noch einer kurzen Betrachtung. Insbesondere, wenn Indizes, wie in der Tabelle `PortTable` verwendet werden (PortBoard und PortNumber), ist die Verwendung und Auflösung von besonderem Interesse. So kann bei den OID´s folgende Unterscheidung getroffen werden:

- <u>kein Index</u> (normale MIB-Variable)

[10] Die Möglichkeit, daß mehrere JDMK-Agenten gleichzeitig laufen können und somit wieder ein asynchroner Zugriff auf die MIB-Variablen geschehen kann ist zu berücksichtigen. Da in der Beispiel-Implementierung von einem JDMK-Agenten ausgegangen werden kann ist dieser Fall nur theoretisch untersucht worden.

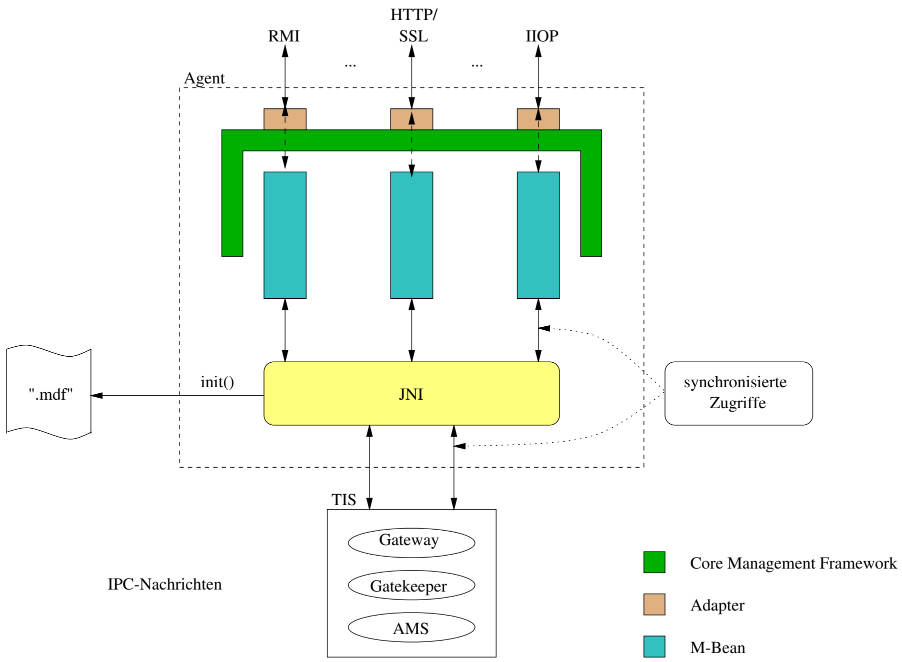

Abbildung 6.8: Eine zentrale, globale Schnittstelle für alle M-Beans

Dies stellt die einfachste Handhabung dar. Es werden keine Erweiterungen für die Basis-OID, abgesehen von der 0 bei der Verwendung der `set`-Methode benöigt. Das Durchwandern der Baumstruktur und somit das Ermitteln der entsprechenden MIB-Variable erfolgt nach dem Prinzip der Bisektion. Dadurch sind die CAsnNode-Objekte durch eine Baumstruktur zugreifbar, obwohl sie physikalisch eine gelinkte Liste im Speicher darstellen.

- einstufiger Index
  Bei einstufigem Index wird der Basis-OID der Tabelle der entsprechende Index angehängt, Das Durchwandern der Baumstruktur erfolgt wieder nach dem Bisektionsprinzip. Ist die Tabelle gefunden, wird linear der entsprechende Index für den Eintrag gesucht.

- zweistufiger Index
  Ist, wie in der Tabelle `PortTable`, ein zweistufiger Index vorhanden,[11] so wird der Index wie in Tabellen-OID´s üblich durch Punkt getrennt an die Basis-OID der Tablle angehängt. Eine Kodierung für ein bestimmtes Board würde folgendermaßen aussehen:

[11] Stellt der Index zudem noch einen String dar, so muß zum Einen der String in eine brauchbare Darstellung gebracht werden, die es ermöglicht, diesen in die Baumstruktur zu integrieren. Zudem muß bei mehreren Einträgen aufgrund der unter Umständen hohen Zahl von Einträgen auf eine lineare Suche innerhalb der Tabelle verzichtet werden. Dieses Problem wird gelöst, indem die ASCII-Codierung der Zeichen, welche die Einträge ausmachen, verwendet wird. Es wird, um die Trennung zwischen den Einträgen zu erreichen, eine Längenangabe der Zeichen vorangestellt. Auf diese Art und Weise kann ein Eintrag wieder durch Durchlaufen einer Baumstruktur gefunden werden.

⟨OID der Tabelle⟩.10.5

Für die Implementierung der Schnittstelle wurde die Variante mit einer zentralen und gemeinsamen Schnittstelle für alle M-Beans eines Agenten gewählt.

6.5.4 Schnittstellenrealisierung

Die Schnittstelle gliedert sich in zwei Teile. Ein Teil ist in Java implementiert und stellt die Java-Repräsentation der Schnittstelle mit deren Methoden dar. Der andere Teil ist in C++ realisiert und beinhaltet die eigentlichen Methoden, welche die Zugriffe auf die TIS-Applikationen durchführen:

- nativeBase.java
  Diese Klasse lädt bei der Instantiierung die Bibliothek, welche mit "nativeBase.cpp" zur Verfügung gestellt wird. Die Instantiierung wird bei dem ersten Aufruf einer Native-Methode durchgeführt. Die zur Verfügung gestellten Native Methoden lauten:

  - getStringNative
    Methode zum Einlesen von String-Variablen.
  - getIntNative
    Methode zum Einlesen von Integer-Variablen.
  - getNextStringNative
    Methode zum Durchlaufen von OCTET-STRING-Spalten in SNMP-Tabellen. Es wird die Folge-OID zurückgegeben.
  - getNextIntNative
    Methode zum Durchlaufen von INTEGER-Spalten in SNMP-Tabellen. Es wird die Folge-OID zurückgegeben.
  - setNative
    Methode zum Setzen von MIB-Variablen. Diese Methode wird mit den entsprechenden Parametern implementiert und somit kann der Methodenbezeichner überladen werden.
  - removeNative
    Methode zum Löschen von MIB-Variablen bzw. zum Entfernen von Tabelleneinträgen.

  Zusätzlich werden Methoden zur Konvertierung von Datenformaten wie sie in TIS verwendet werden, und zum Aufbau der Knotenstruktur der MIB-Variablen, wie sie in 4.7 beschrieben wird, zur Verfügung gestellt. Die Konvertierung resultiert daraus, daß MIBGEN aus den Informationen der MIB festgelegte Darstellungen der Variablen generiert und diese Darstellung nicht unbedingt direkt lesbar ist[12]. In Kapitel 6.7.1 ist eine weitere Möglichkeit aufgezeigt, wie durch eine Erweiterung die Lesbarkeit der automatisch generierten HTML-Seiten bei Zugriff über einen Web-Browser erhöht werden kann. Ein Beispiel, wie `get-` und `getNext`-Methoden auf der Java-Seite definiert werden ist nachfolgend zu sehen:

[12] Beispielsweise ist die Variable IPAdress in der MIB-Gruppe NetworkParms als OCTET-STRING gekennzeichnet. Die Darstellung als einzelne Bytes innerhalb der generierten Klasse erfordert eine Umsetzung in eine Stringrepräsentation.

```
// get
public static native String getStringNative(int[] mibVarOID);
public static native int getIntNative(int[] mibVarOID);

// getNext
public static native int
    getNextIntNative(int[][] mibVarOID, int oidIndex);
public static native String
    getNextStringNative(int[][] mibVarOID, int oidIndex);
```

Es ist zu beachten, daß die Übergabeparameter bei `get` ein- und bei `getNext` zweidimensional ausgelegt sind. Dies resultiert daraus, daß bei `getNext`-Aufrufen eine OID zurückgegeben werden muß und da die Länge einer OID (welche zurückgegeben wird) im Voraus nicht bekannt ist, kann diese nicht als normaler Rückgabewert behandelt werden. So wird ein zweidimensionales Feld benutzt, welches in der einen Dimension den "Aufhänger" für die beliebig lange Rückgabe-OID besitzt und in der anderen Dimension die eigentlichen OID-Werte beinhaltet.

- nativeBase.h
  Diese Header-Datei wird durch Anwendung der `javah`-Applikation, welche von JDK zur Verfügung gestellt wird und als Parameter ein ".class"-File benötigt, erzeugt, Als Parameter wird die Klasse "jni.nativeBase.class" übergeben, die aus der nativeBase.java Datei entstanden ist. Es muß hierbei die Option "-jni" angegeben werden, damit die Header-Datei generiert werden kann. Innerhalb dieser Datei werden die Prototypen der Native-Methoden aufgeführt. Der Prototyp für die Native-Methode `getIntNative` sieht folgendermaßen aus:

```
/*
 * Class:     jni_nativeBase
 * Method:    getIntNative
 * Signature: ([I)I
 */
JNIEXPORT jint JNICALL Java_jni_nativeBase_getIntNative
  (JNIEnv *, jclass, jintArray);
```

  Die Angabe "Signature: ([I)I" stellt dar, daß diese Methode einen Integer-Wert als Eingabe und als Ausgabeparameter enthält. Die Kodierung der Ein- und Ausgabeparameter ist in Tabelle 6.1 aufgeführt.

- nativeBase.cpp
  Diese Klasse führt die eigentlichen Methodenaufrufe aus. Da die IPC-Nachrichten, welche innerhalb der TIS Managementumgebung Verwendung finden als Grundlage die Datenstruktur `RFC1157VarBind`, welche von der Windows NT API zur Verfügung gestellt wird, benutzen, muß ein solches Objekt zuerst aufgebaut werden. Danach wird der erforderliche Request gebildet und mit Aufruf der `ResolveVarbind`-Methode der Request abgesetzt. Nach erfolgreicher Beendigung des Requests wird ggf. ein Ergebnis an die Java-Umgebung übergeben. In der Testphase des Prototypen wird kein Exception Handling auf der Native-Seite durchgeführt, jedoch ist es durch das Java Native Interface möglich, dies zu realisieren. Eine Native-Methode ist anhand von `getIntNative` nachfolgend dargestellt.

```
/*
 * Class:     jni_nativeBase
 * Method:    getIntNative
 * Signature: ([I)I
 */
JNIEXPORT jint JNICALL Java_jni_nativeBase_getIntNative
  (JNIEnv *env, jclass, jintArray intArray) {
  RFC1157VarBind* varBindPtr
      = (RFC1157VarBind*) calloc(1,sizeof(RFC1157VarBind));
  varBindPtr->name.idLength = env->GetArrayLength(intArray);
  varBindPtr->name.ids
      = (unsigned int*) env->GetIntArrayElements(intArray,0);
  ResolveVarBind(varBindPtr,ASN_RFC1157_GETREQUEST);
  return varBindPtr->value.asnValue.number;
}
```

Die Implementierungen der Schnittstelle für die Java und C++-Seite sind in Anhang B zu finden.

6.5.5 Caching

Da das Caching der Werte auf bereits zugegriffene MIB-Variablen innerhalb der SXA-Funktionalität geschieht, wurde auf eine Realisierung auf M-Bean-Seite verzichtet. Zur Demonstration und da die MIB-Variablen durch den MIBGEN-Generator erzeugt werden, wurde eine Zwischenspeicherung durch M-Bean Properties beibehalten. Allerdings wird bei einem Zugriff auf das entsprechende M-Bean durch Aufruf der Native-Methoden der aktuelle Stand der Applikation bzw. des Teilbereichs in das M-Bean geladen. Hierbei sei zu beachten, daß es sich um einen "write through"-Cache handelt. Es werden nur gelesene Werte gespeichert.
Das Caching innerhalb des JNI-Interfaces geschieht durch Vergleich der bisher gesendeten Requests mit dem aktuellen Request. Die Werte, welche in dem Cache verwaltet werden, stellen die bisher gesendeten IPC-Nachrichten dar. Sie besitzen innerhalb des Caches eine bestimme Lebensdauer. Ist diese überschritten, wird der Cache-Eintrag verworfen und eine Nachricht gesendet. Ist eine Ermittlung des Wertes durch Cache-Zugriff möglich, so ist die Versendung einer IPC-Nachricht nicht notwendig und es wird sofort der ermittelte Wert zurückgegeben. Andernfalls wird die Nachricht mit `BuildVarbind` gesendet und auf die Antwort gewartet. Danach wird der Cache aktualisiert und der Wert zurückgegeben.
Die Caching Möglichkeiten und der Ablauf eines `get`-Zugriffs sind in Abbildung 6.9 dargestellt.

6.6 Nutzung der JDMK-Services

Im Zuge der Implementierung des Prototypen werden zu Testzwecken einige Dienste des JDMK verwendet und deren Brauchbarkeit überprüft. Besonders hervorzuheben sind in diesem Zusammenhang der Event-Service und Discovery-Service, welche nachfolgend genauer beschrieben werden. Ebenso wurde der M-Let-Service zum Laden von Event-Listenern, der Alarm-Clock- und Security-Service verwendet.

6.6.1 Event-Service

Die Nutzung des Event-Modells ist von zentraler Bedeutung bei dem Management unter JDMK. Mit diesem Konzept ist es möglich, selbstagierende Agenten zu entwickeln, die auf

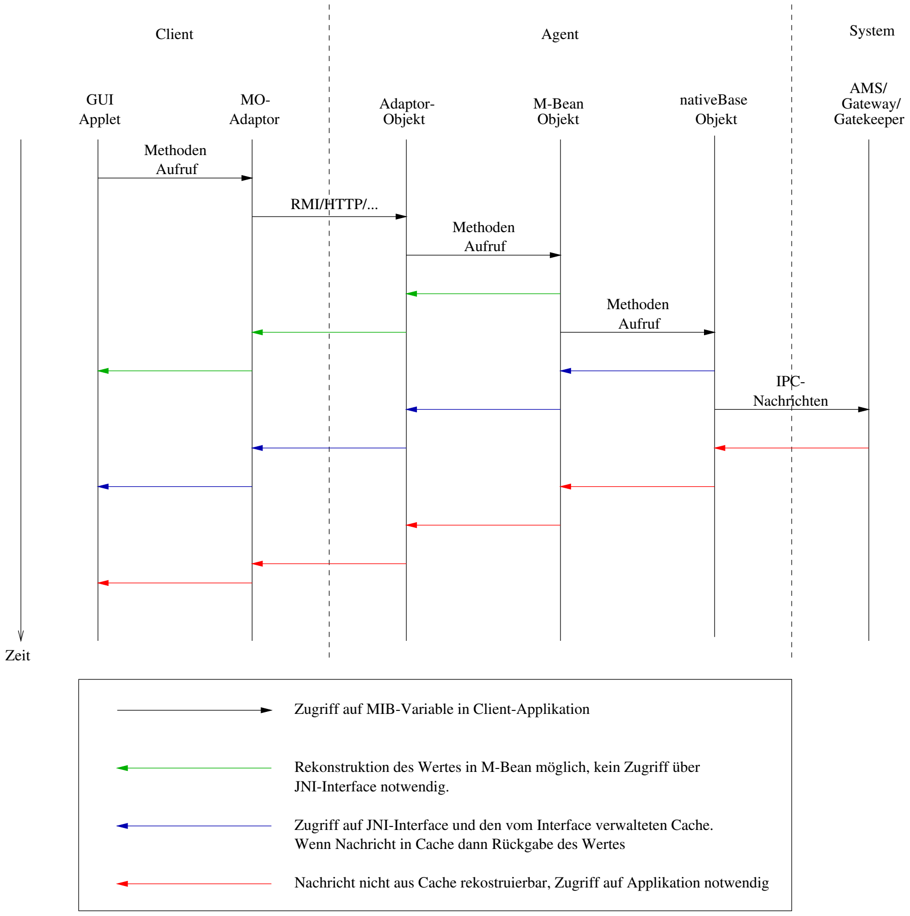

Abbildung 6.9: Caching-Mechanismus (GET)

Ereignisse, welche von Agenten spezifiziert werden können, reagieren.
Unter JDMK muß bei Verwendung dieses Dienstes ein Listener-Objekt innerhalb des zu überwachenden Agenten registriert werden. Die Theorie des Event-Service unter JDMK ist in Kapitel 3.7.8 beschrieben. Eine Registrierung ist deshalb notwendig, da nur ein registriertes M-Bean durch die Adapter eindeutig (über den Bezeichner) angesprochen werden kann.
Dieses Kapitel und die folgenden Unterkapitel beschreiben eine exemplarische Verwendung des Eventmodells innerhalb der TIS-Management-Umgebung[13].

EventListener

Dieser Ansatz definiert ein generisches Eventmodell. Die Events können durch den Benutzer durch Erzeugung einer Klasse selbst spezifiziert werden. Der Vorgang, ein M-Bean mit einem EventListener zu versehen und diesen durch einen Client zu überwachen ist durch mehrere

[13] Die Verwendung des Eventmodells wird insbesondere bei der Überwachung von MIB-Variablen wie aktuelle Bandbreite und bestehende Verbindungen verwendet.

Schritte festgelegt:

1. Spezifikation Event-Klasse
   Wenn ein benutzerdefinierter Typ von Events gewünscht wird, dann ist eine eigene Klasse zu entwickeln, welche die Art des Events darstellt. Die Klasse ist als Unterklasse von `java.util.EventObject` zu implementieren. Innerhalb dieser Klasse können Properties verwendet werden, die bei Empfang und Auswertung des Events zusätzliche Informationen enthalten können (z.B. Laufzeit des Objekts innerhalb eines Netzes). Die Erstellung einer Klasse ermöglicht die Verwendung mehrerer Konstruktoren. Dadurch kann die Art und Struktur der Event-Klasse spezifischen Erfordernissen angepaßt werden. Ein Beispiel für eine Event-Klasse ist nachfolgend dargestellt:

```
package tis20.impl;

import java.util.EventObject;

public class NetworkParmsImplEvent extends EventObject {

  /**
   * Create a NetworkParmsImplEvent object. The source of the event
   * as well as the value of the "NbChanges" property needs
   * to be specified.
   */
  public NetworkParmsImplEvent(NetworkParmsImpl source) {
    super(source);
  }

  public NetworkParmsImplEvent(NetworkParmsImpl source
                             , String ComputerName) {
    super(source);
  }

  /**
   * Returns the version of this class.
   */
  public String getClassVersion() {
    return (sccs_id);
  }

  /**
   * Version of the implementation
   */
  private static final
    String sccs_id = "@(#)NetworkParmsImplEvent.java 1.0 01/01/99";

}
```

2. Spezifikation Listener-Interface
   Nun muß ein Interface entwickelt werden, das als M-Bean in das CMF des Agenten eingefügt werden muß, um eventuelle Events abzufangen. Die Realisierung als M-Bean macht diesen Listener durch die Adapter und damit den verschiedenen Möglichkeiten des Managements zugreifbar. Da das Listener-Objekt durch eine Client-Applikation in das CMF integriert wird, ist mit MOGEN (vgl. 3.3) ein Interface und Stub zu erzeugen. Dieses Interface ist von `java.util.EventListener` abzuleiten und es sind die Methoden bekanntzugeben, welche bei Erhalt eines Events angewendet werden sollen. Eine Listener-Klasse kann folgendermaßen aussehen:

```
package tis20.impl;
```

```
import java.util.*;

/**
 * A NetworkParmsImpl event listener interface.
 * This interface needs to be implemented
 * in order to receive
 *
 * NetworkParmsImpl events emitted by the
 * NetworkParmsImplEvent M-bean.
 *
 * @version     1.0     01/01/99
 * @author      Harald Knoechlein
 */

public interface NetworkParmsImplListener extends java.util.EventListener {

  public void handleEvent1(NetworkParmsImplEvent evt);

  public void handleEvent2(NetworkParmsImplEvent evt);
}
```

3. Entwicklung Client-Applikation
   Um das Listener-Objekt innerhalb des CMF des Agenten integrieren zu können, ist ein Client zu entwickeln. Die Events, welche später durch dieses Objekt gesendet werden, können allerdings von allen Clients oder Agenten, welche dieses Listener-Objekt spezifiziert haben, empfangen werden.
   Es ist zu beachten, daß Agenten selbst ebenso die Möglichkeit besitzen, Listener-Objekte für ihre eigenen aber auch M-Beans anderer Agenten anzulegen.

4. Ausführen der Client Applikation und Anlegen des Listener-Objekts
   Um dieses Objekt zu integrieren, muß zunächst der Agent über einen Adapter kontaktiert werden. Gegebenenfalls ist ein Paßwort und Login zu verwenden, falls dies der Adapter erfordert.
   Danach muß das M-Bean, welches überprüft werden soll, mit dessen Objektnamen ausgewählt und als Managed-Objekt gehandhabt werden. (Dies geschieht durch einen Cast auf die MO-Klasse).
   Ist nun das M-Bean ermittelt, so kann durch Verwendung der add<M-Bean-Klasse>Listener Methode der Listener dem CMF hinzugefügt werden. Es ist zu beachten, daß diese Methode, ebenso wie die Methode für das Entfernen des Listeners von dem M-Bean zu Verfügung gestellt werden muß. Falls solche Methoden zum M-Bean hinzugefügt werden sollen, so muß das M-Bean gestoppt, um den erforderlichen Quellcode erweitert und wieder dem Agenten hinzugefügt werden (Restart). Es kann allerdings in Erwägung gezogen werden, generell diese Methoden zu den M-Beans hinzuzunehmen und sich damit eine Hintertür offen zu halten.
   Ausschnittsweise kann ein solcher Vorgang innerhalb der Client-Applikation folgendermaßen implementiert werden:

```
...
// connect to Adaptor ...
  AdaptorClient adaptor = new AdaptorClient();
  adaptor.connect(null, agentHost, agentPort, ServiceName.APT_RMI);
...
// Get the instance of the NetworkParmsImpl m-bean in the
// remote object server.
  String NetworkParmsImplClass = "NetworkParmsImpl";
```

```
  ObjectName NetworkParmsImplName
    = new ObjectName(domain + ":tis20.impl.NetworkParmsImpl");
  tis20.impl.NetworkParmsImplMO NetworkParmsImpl
    = (tis20.impl.NetworkParmsImplMO)
      ((Vector)adaptor.getObject(NetworkParmsImplName, null)).firstElement();
...
// Create an event listener.
  ClientListener listener= new ClientListener();
...
// Register a listener with the remote event source.
  NetworkParmsImpl.addNetworkParmsImplListenerMO(listener);
...
```

5. <u>Senden von Events durch Agent</u>
   Treten bei den "überwachten" Properties Veränderungen ein (beispielsweise das Hinzufügen eines neuen Tabelleneintrags oder das Setzen einer MIB-Variablen), werden Events an alle Listener gesandt. Das Senden muß ebenso wie die Methoden zum Hinzufügen und Entfernen von Listener-Objekten innerhalb der überwachten M-Bean-Klasse durch Methodenaufrufe realisiert werden. Ein Aufruf bei Veränderung einer MIB-Variablen kann beispielsweise so aussehen:

```
...
/**
 * Setter for the "ComputerName" variable.
 */
public  void setComputerName(String x) throws SnmpStatusException {
   String oid =  "1.3.6.1.4.1.2206.5.1.1.1.1.0";
   ComputerName = x;
   nativeBase.setNative(tools.ConvertTools.String2Int(oid), x);
   // Send an event !
   //
   NetworkParmsImplEvent event
     = new NetworkParmsImplEvent(this, new String(ComputerName));
   deliverChange(event);
}
...
/**
 * Send an event to all the registered listeners.
 *
 * @param event the event to emmit.
 *
 */
private void deliverChange(NetworkParmsImplEvent event) {
  Vector listeners;
  int n;
  synchronized (this) {
    listeners = (Vector)NetworkParmsImplListeners.clone();
    n = listeners.size();
    if (n == 0) return;
  }
  Vector symbols = new Vector();
  for (int i = 0; i < listeners.size(); i++) {
    NetworkParmsImplListener NetworkParmsImplListener
      = (NetworkParmsImplListener)listener.elementAt(i);
    NetworkParmsImplListener.handleEvent1(event);
  }
}
...
```

   Es wird ein Event-Objekt mit entsprechendem Konstruktor instantiiert und mittels der `deliverChange`-Methode an alle Listener geschickt. Eine Empfangsüberprüfung findet nicht statt. Eine Möglichkeit wäre, in den Objekten, die Events erhalten, Methoden zur

Versendung von "EmpfangsEvents" vorzusehen. Diese Event-Klasse ist jedoch selbst zu implementieren, es wird kein expliziter Mechanismus von JDMK unterstützt. Wie aus dem obigen Beispiel zu ersehen ist, wird das EventHandler-Interface von JDMK verwendet. Dieses Interface ist für die Weiterleitung und Verarbeitung der Events zuständig.

6. Entfernen von Event-Listener-Objekten
   Ist ein Überwachen des M-Beans nicht mehr erforderlich, so werden die Listener-Objekte wieder entfernt. Dies geschieht durch die Methode `remove<M-Bean-Klasse>Listener`, welche von der M-Bean-Klasse zur Verfügung gestellt werden muß.

Die Verwendung von Listenern ist in Abbildung 3.6 dargestellt.

Gauge/Counter-Monitoring

Ein Dienst des JDMK, welcher auf das Event-Modell aufsetzt, ist das Monitoring. Es können wie in Kapitel 3.7.11 beschrieben, Gauge und Counter-Monitore verwendet werden. Innerhalb des TIS-Management ist dies ein wesentlicher Punkt, da mit diesem Dienst die zur Verfügung stehende Bandbreite für eine Übertragung und auch die Gesamtzahl der Kommunikationsverbindungen überwacht werden kann.
Die Handhabung erfolgt wie bei den benutzerdefinierten Events. Es ist ebenso eine Event-Klasse, welche die Klasse `MonitorEvent` (von JDMK angeboten) erweitert, zu implementieren. Damit kann zusätzliche Information in ein Event-Objekt integriert werden. Allerdings ist die Handhabung der auftretenden Events umfangreicher wie bei den benutzerdefinierten Events. Dies betrifft insbesondere die jeweilige Client-Applikation und die Monitor-Listener-Klasse. Die wesentlichen Punkte sind nachfolgend aufgeführt:

- Client-Applikation
  Hier muß unterschieden werden, ob es sich um einen Counter- oder Gauge-Monitor handelt.

  - Counter-Monitor
    Bei einem Counter-Monitor sind die obere Schranke bei Beginn der Zählung, ein Offset, welcher gegebenenfalls bei Überschreitung der Schranke aufaddiert wird und ein Modulus mitanzugeben, wobei Letzterer das Verhalten bei Überschreitung eines Wertebereichs bestimmt. Ein Beispiel für die Verwendung eines Counter-Monitors innerhalb des TIS kann folgendermaßen aussehen:

```
package ...
imports ...
...
public class Client implements Runnable {
...
  // Create a counter monitor m-bean.
  //
  String GkStatsCurrentNoOfCallsMonitorClass
    = "com.sun.jaw.impl.agent.services.monitor.CounterMonitor";
  ObjectName GkStatsCurrentNoOfCallsMonitorName
    = new ObjectName(domain
                    + ":com.sun.jaw.impl.agent.services.monitor.CounterMonitorMO.id
                       =GkStatsCurrentNoOfCalls");
  CounterMonitorMO GkStatsCurrentNoOfCallsMonitor
    = (CounterMonitorMO) adaptor.cb_newMO(GkStatsCurrentNoOfCallsMonitorClass,
                                          GkStatsCurrentNoOfCallsMonitorName,
                                          null);
```

```
...
  // Set the values of the counter monitor object.
  // The counter monitor will check every second the value of the
  // "GkStatsCurrentNoOfCalls" property for the GkStatistics m-bean.
  // A threshold reached monitor event will be emited if the
  // "GkStatsCurrentNoOfCalls" property value reached or exceeded the
  // comparison level.
  // If the comparison level is reached or exceeded the comparison level
  // will be incremented by the offset value.
  //
  GkStatsCurrentNoOfCallsMonitor
      .setObservedObject(GkStatisticsImplName);
  GkStatsCurrentNoOfCallsMonitor
      .setObservedProperty("GkStatsCurrentNoOfCalls");
  GkStatsCurrentNoOfCallsMonitor
      .setNotifyOnOff(Boolean.TRUE);
  GkStatsCurrentNoOfCallsMonitor
      .setComparisonLevel(GkStatsCurrentNoOfCallsHigh);
  GkStatsCurrentNoOfCallsMonitor
      .setOffsetValue(offSet);
  GkStatsCurrentNoOfCallsMonitor
      .setGranularityPeriod(new Integer(1000));
  GkStatsCurrentNoOfCallsMonitor
      .setModulusValue(new Integer(32));
  GkStatsCurrentNoOfCallsMonitor
      .performStart();
...
  // Register a monitor listener with the counter monitor,
  // enabling the client to receive monitor events emitted by the
  // counter monitor.
  //
  GkStatsCurrentNoOfCallsMonitor.addMonitorListenerMO(listen);
...
```

Wie aus dem Beispiel ersichtlich, kann ebenso die Überwachung ausgesetzt (`setNotifyOn/Off`), das Listener-Objekt aktiviert/deaktiviert (`performStart/Stop`) und die Zeitspanne der Überwachungszeitpunkte (`setGranularityPeriod`) angegeben werden.

– Gauge-Monitor
  Ist ein Gauge-Monitor anzulegen, dann müssen weitere Einstellungen vor der Verwendung durchgeführt werden. Es wird kein Modulus verwendet, es ist aber notwendig, die Besonderheit, welche durch die Überwachung von Bandbreite zu Tage tritt, zu berücksichtigen. Diese besteht in der Festlegung der oberen und unteren Schranke, in welcher sich der Wert bewegen darf. Die Funktionsweise ist in Kapitel 3.7.11 beschrieben. Nachfolgend ist ein Beispiel für die Verwendung dieses Monitor innerhalb des TIS zu sehen.

```
packages ...
imports ...
...
public class Client implements Runnable {
...
  // Create a gauge monitor m-bean.
  //
  String GkStatsCurrentBandwidthMonitorClass
    = "com.sun.jaw.impl.agent.services.monitor.GaugeMonitor";
  ObjectName GkStatsCurrentBandwidthMonitorName
    = new ObjectName(domain
                   + ":com.sun.jaw.impl.agent.services.monitor.GaugeMonitorMO.id
                   =GkStatsCurrentBandwidth");
  GaugeMonitorMO GkStatsCurrentBandwidthMonitor
```

```
      = (GaugeMonitorMO) adaptor.cb_newMO(GkStatsCurrentBandwidthMonitorClass,
                                           GkStatsCurrentBandwidthMonitorName,
                                           null);
...
 // Initialize the threshold values for the gauge monitor.
 Integer GkStatsCurrentBandwidthHigh
     = new Integer(GkStatisticsImpl.getGkStatsCurrentBandwidth()
                                     .intValue() + 15);
 Integer GkStatsCurrentBandwidthLow
     = new Integer(GkStatisticsImpl.getGkStatsCurrentBandwidth()
                                     .intValue() - 15);
 // Set the values of the gauge monitor object.
 // The gauge monitor will check every second the value of the
 // "GkStatsCurrentBandwidth" property for the AskMe m-bean.
 // A threshold high monitor event will be emited if the
 // "GkStatsCurrentBandwidth"
 // property value reached or exceeded the high threshold value.
 // A threshold low monitor event will be emited if the
 // "GkStatsCurrentBandwidth"
 // property value reached or fell below the low threshold value.
 GkStatsCurrentBandwidthMonitor
     .setObservedObject(GkStatisticsImplName);
 GkStatsCurrentBandwidthMonitor
     .setObservedProperty("GkStatsCurrentBandwidth");
 GkStatsCurrentBandwidthMonitor
     .setNotifyLowOnOff(Boolean.TRUE);
 GkStatsCurrentBandwidthMonitor
     .setNotifyHighOnOff(Boolean.TRUE);
 GkStatsCurrentBandwidthMonitor
     .setThresholdLowValue(GkStatsCurrentBandwidthLow);
 GkStatsCurrentBandwidthMonitor
     .setThresholdHighValue(GkStatsCurrentBandwidthHigh);
 GkStatsCurrentBandwidthMonitor
     .setGranularityPeriod(new Integer(1000));
 GkStatsCurrentBandwidthMonitor.performStart();
 // Register a monitor listener with the gauge monitor,
 // enabling the client to receive monitor events emitted by the
 // gauge monitor.
 GkStatsCurrentBandwidthMonitor.addMonitorListenerMO(listen);
...
```

- Monitor-Listener-Klasse
  Ebenso wie in der Client-Applikation ist der Verarbeitung von Events in der Listener-Klasse besondere Beachtung zu schenken. In dieser Klasse werden die Events nach ihrer Art unterschieden. Es werden folgende Event-Typen unterschieden[14]:

  - GRANULARITY_PERIOD_EVT
    Dieser Event tritt auf, wenn die Zeitspanne, in welcher die Überwachung einer bestimmten M-Bean-Property, stattfindet, einen negativen Wert oder null ist.
  - OBSERVED_OBJECT_EVT
    Falls eine M-Bean (bzw. deren Property) angesprochen wird, die nicht innerhalb des CMF des jeweiligen Agenten registriert ist, tritt dieser Event auf.
  - OBSERVED_PROPERTY_EVT
    Tritt auf, falls das Property, welches überwacht werden soll, von dem angesprochenen M-Bean nicht unterstützt wird.

[14] Es sei darauf hingewiesen, daß es sich um die Klasse der Monitor-Events handelt. Damit werden viele der Arten, die aufgeführt werden, von Counter- und Gauge-Monitoren gesendet. Falls sich der Event auf eine bestimme Klasse von Monitoren beschränkt, wird gesondert darauf hingewiesen

- OBSERVED_PROPERTY_TYPE_EVT
  Tritt auf, falls der falsche Property-Typ verwendet wird.
- COMPARISON_LEVEL_EVT
  Tritt auf, falls ein negativer Schrankenwert angegeben wird. Dieser Typ von Event wird nur von Counter-Monitoren gesendet.
- OFFSET_VALUE_EVT
  Tritt auf, falls ein negativer Offsetwert angegeben wird. Dieser Typ von Event wird nur von Counter-Monitoren gesendet.
- MODULUS_VALUE_EVT
  Tritt auf, falls ein negativer Modulus angegeben wird. Dieser Typ von Event wird nur von Counter-Monitoren gesendet.
- THRESHOLD_VALUE_REACHED_EVT
  Tritt auf, falls die Property, welche überwacht wird, die vorher festgelegte Schranke überschreitet. Danach kann entschieden werden, ob ein Offset addiert werden soll, oder eine andere Aktion ausgeführt wird. Dieser Typ von Event wird nur von Counter-Monitoren gesendet.
- THRESHOLD_TYPE_EVT
  Tritt auf, falls der Typ von der unteren Schranke mit dem von der oberen Schranke nicht übereinstimmt. Es ist notwendig für ein sinnvolles Überwachen, gleiche Typen zu verwenden. Dieser Typ von Event wird nur von Gauge-Monitoren gesendet.
- THRESHIGH_LOWER_THAN_THRESLOW_EVT
  Tritt auf, wenn die obere Schranke kleiner als die untere Schranke ist. Falls negative Werte verarbeitet werden sollen, so ist entweder mit Absolutwerten zu arbeiten, oder die Schranken sinnvoll umzubenennen. Dieser Typ von Event wird nur von Gauge-Monitoren gesendet.
- THRESHOLD_HIGH_VALUE_EXCEEDED_EVT
  Tritt auf, wenn die überwachte Property die obere Schranke überschreitet. Dieser Typ von Event wird nur von Gauge-Monitoren gesendet.
- THRESHOLD_LOW_VALUE_EXCEEDED_EVT
  Tritt auf, wenn die überwachte Property die untere Schranke unterschreitet. Dieser Typ von Event wird nur von Gauge-Monitoren gesendet.

Die Unterscheidung zwischen diesen Arten ist innerhalb der Monitor-Listener-Klasse zu treffen. Ein Beispiel bei der Verwendung innerhalb des TIS-Prototypen hat folgende Gestalt:

```
packages ...
imports ...
...
public class ClientListener implements MonitorListenerMO {
...
  public void handleMonitor(MonitorEventMO event) {
    // Get a handle on the monitor responsable for the event
    // emmited.
       MonitorMO Monitor = (MonitorMO) event.getSource();
    // Process the different types of events fired by the monitors.
    //
    try {
      switch (event.getMonitorEventType().intValue()) {
```

```
              case MonitorEvent.COMPARISON_LEVEL_EVT:
              case MonitorEvent.GRANULARITY_PERIOD_EVT:
              case MonitorEvent.MODULUS_VALUE_EVT:
              case MonitorEvent.OBSERVED_OBJECT_EVT:
              case MonitorEvent.OBSERVED_PROPERTY_EVT:
              case MonitorEvent.OBSERVED_PROPERTY_TYPE_EVT:
              case MonitorEvent.OFFSET_VALUE_EVT:
              case MonitorEvent.THRESHIGH_LOWER_THAN_THRESLOW_EVT:
              case MonitorEvent.THRESHOLD_TYPE_EVT:
              case MonitorEvent.THRESHOLD_HIGH_VALUE_EXCEEDED_EVT:
              System.out.println("Clientlistener: Monitor event -- "
                                   + event.getMonitorEventObservedProperty()
                                   + " Has Exceeded The Threshold High -- Value = "
                                   + event.getMonitorEventDerivedGauge());
              System.out.println("Clientlistener: GkStatsCurrentBandwidth HIGH");
            break;
            case MonitorEvent.THRESHOLD_LOW_VALUE_EXCEEDED_EVT:
              System.out.println("Clientlistener: Monitor event - "
                                 + event.getMonitorEventObservedProperty()
                                 + " Has Exceeded The Threshold Low >> Value = "
                                 + event.getMonitorEventDerivedGauge());
              System.out.println("Clientlistener: GkStatsCurrentBandwidth LOW");
            break;
            case MonitorEvent.THRESHOLD_VALUE_REACHED_EVT:
              System.out.println("Clientlistener: Monitor event -- "
                                 + event.getMonitorEventObservedProperty()
                                 + " Has Reached The Threshold -- Value "
                                 + event.getMonitorEventDerivedGauge());
              System.out.println("Clientlistener:
                                 GkStatsCurrentNoOfCalls increased by 5");
            break;
          default:
            System.out.println("Clientlistener: Unknown event Type (?)");
          }
        } catch (Exception e) {
          System.out.println("Got An Exception !");
          System.out.println(e);
        }
      ...
      }
```

6.6.2 Discovery-Service

Dieser Dienst ermöglicht die Feststellung der JDMK-Agenten, welche über einen Discovery Responder verfügen (vgl. Kapitel 3.7.12). Dies kann unter dem TIS-Management benutzt werden, um mehrere TIS-Boxen (Zonen (vgl. Kapitel 4), zu ermitteln. Der Discovery Responder kann als M-Bean in das CMF der entsprechenden Agenten zur Laufzeit integriert werden. Im Beispiel des TIS-Prototypen wurde eine Discovery-Monitor-Klasse erzeugt, welche entweder durch den M-LET-Service durch den Agenten selbst oder durch eine Client-Applikation instantiiert und registriert wird. Die Klasse stellt zwei Methoden zum Starten und Stoppen des Monitors zu Verfügung. Die Methoden zum Hinzufügen und Entfernen von Respondern beschränken sich auf das Anlegen und Entfernen von M-Beans in den jeweiligen Agenten. Interessant ist die Methode, den Discovery-Search-Service (vgl. Kapitel 3.7.12) zu verwenden. Nachfolgend ist die entsprechende Methode aufgezeigt:

```
public class DiscMonitor {
...
   public void performLookingForAdaptors() {
      try {
         int multicastPort = 9000;
         String multicastGroup = "224.224.224.224";
```

```
         int ttl = 1;

         DiscoveryClient discoveryClient = new DiscoveryClient();
         discoveryClient.setMulticastGroup(multicastGroup);
         discoveryClient.setMulticastPort(multicastPort);
         discoveryClient.setTimeToLiveInt(ttl);
         discoveryClient.performStart();
         Vector result = discoveryClient.performFindAdaptors();
         Enumeration adaptorList = result.elements();
         htmlOutput = new StringBuffer("Adaptors:<BR>");
         while (adaptorList.hasMoreElements()) {
            DiscoveryResponse response = (DiscoveryResponse) adaptorList.nextElement();
            Vector adaptors = response.objectList;
            Enumeration enum = adaptors.elements();
            while (enum.hasMoreElements()) {
               //System.out.println("Adaptor="+enum.nextElement().toString());
               htmlOutput.append(enum.nextElement().toString());
               htmlOutput.append("<BR>");
            }
         }
         discoveryClient.performStop();
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
...
   private StringBuffer htmlOutput = new StringBuffer("Adaptors: no adaptors searched");
   private ObjectName DiscoveryMonitorName;
   private AdaptorClient adaptor;
   private DiscoveryMonitorMO AgentMonitor;
}
```

Es wird ein neuer Discovery-Client erzeugt, der später die Antworten der Responder auswertet. Dann kann eine Multicast-Gruppe und ein Port spezifiziert werden. Ist dies nicht der Fall, wird ein Default-Wert verwendet. Es ist möglich, die "Time-to-Live"-Zeit (Zeitraum, in dem der Request gültig ist) eines Discovery-Requests festzulegen (`setTimeToLive`). Der Search-Service kann gestartet und auch wieder gestoppt werden, ohne das Objekt zu zerstören.

6.7 Management-Möglichkeiten

6.7.1 HTML-Browser

JDMK ermöglicht das Management eines Agenten über einen HTML-Adapter. Dieser Adapter ist in Kapitel 3.4 beschrieben. Falls nun ein Web-Browser über diesen Adapter auf einen Agenten zugreift, dann generiert dieser Adapter zur Laufzeit HTML-Seiten, welche den aktuellen Zustand des Agenten und dessen M-Beans darstellen. Innerhalb dieser Seiten kann mittels URL auf HTML-Seiten, welche weitere Eigenschaften der M-Beans beinhalten, gewechselt werden. Eine Einstiegsseite für den Prototypen mit seinen M-Beans ist in Abbildung 6.10 zu sehen. Auf dieser Seite werden M-Beans durch deren Domänenbezeichner grob strukturiert. Danach werden sie mit deren Bezeichner aufgelistet und als URL in die Seite integriert. Wie zu ersehen ist, stehen neben den Adaptoren auch die einzelnen Tabelleneinträge der MIB-Tabellen auf dieser Einstiegsseite. Dies resultiert aus der Tatsache, daß die Elemente einer SNMP-Tabelle als M-Beans realisiert sind.
Nun kann eines der M-Beans ausgewählt werden. Ist beispielsweise das M-Bean NetworkParmsImpl ausgewählt worden, so erscheint danach eine neue Seite, die den Inhalt wie in Abbildung 6.11 dargestellt besitzt. Auf dieser Seite werden die Properties des M-Bean aufgelistet und der

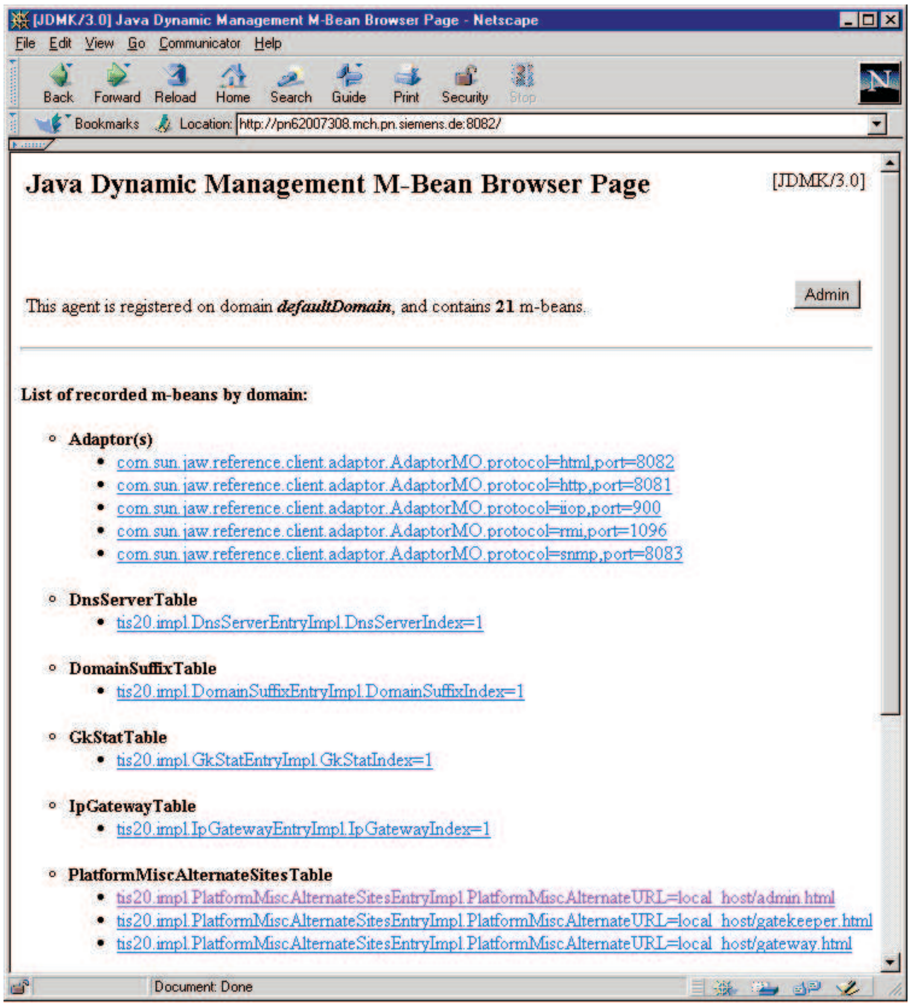

Abbildung 6.10: HTML Einstiegsseite

Zugriff je nach Zugriffsrechte, die in der MIB und in den M-Beans definiert wurden, ermöglicht. Beispielsweise kann auf die Property "ComputerName" lesend und schreibend zugegriffen werden, während "SysUpTime" lediglich eine lesende Zugriffsberechtigung bietet und somit nur dargestellt wird. Auch sind auf dieser Seite die Tabellen der MIB-Gruppe dargestellt. Sie können ebenfalls als URL ausgewählt werden und es wird nach der Wahl eine entsprechende Seite für die SNMP-Tabelle generiert. SNMP-Tabellen werden in der Evaluationsversion von JDMK nur eindimensional unterstützt, daher konnte diese Möglichkeit der Darstellung mehrdimensionaler SNMP-Tabellen nicht getestet werden.
HTML-Seiten können durch den Benutzer erweitert und sogar ersetzt werden. Diese Möglichkeit bietet JDMK durch Bereitstellung des `HtmlStreamableIf`-Interface, welches zwei Methoden bietet:

- `public String isCustomizedViewOnly()`

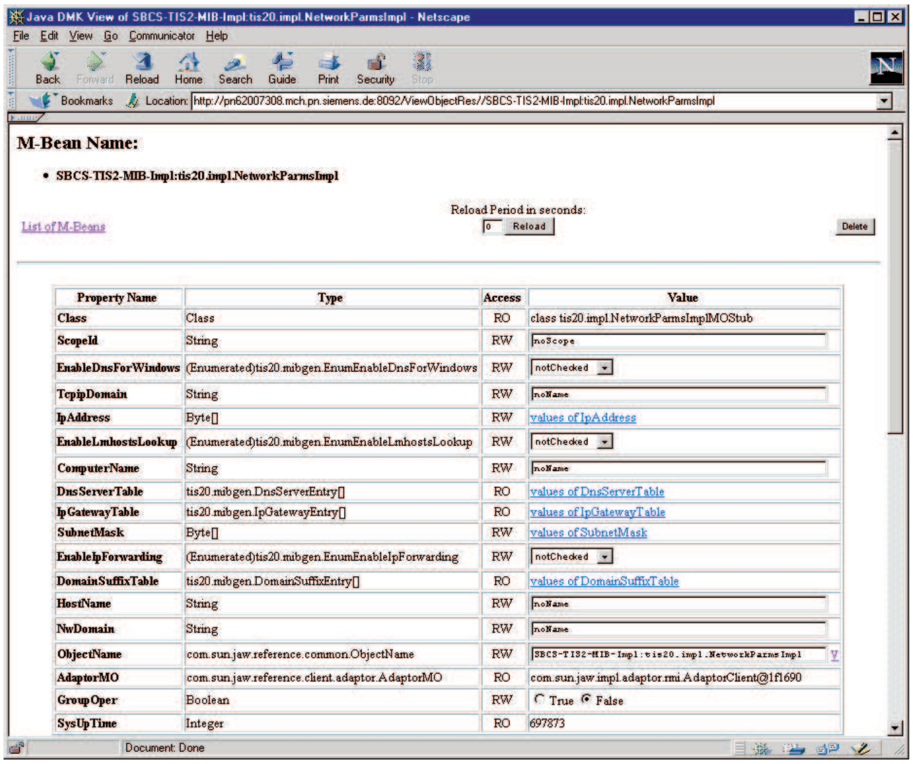

Abbildung 6.11: HTML-Seite des NetworkParmsImpl - M-Bean

Diese Methode wird "true" zurückgeben, falls die generierte HTML-Seite durch die benutzerdefinierte Seite ersetzt werden soll. Wird der "false"-Wert zurückgegeben, wird die generierte Seite lediglich ergänzt.

- `public String WriteToHtml(Object o)`
  Diese Methode schreibt den durch das Objekt übergebenen Inhalt auf die HTML-Seite vor der generierten Information. Es kann beispielsweise ein Streambuffer-Objekt übergeben werden, um den Seiteninhalt zu erweitern. Eine andere Alternative wäre, innerhalb des M-Bean, für welches die Seite erweitert werden soll, eine Methode zu implementieren, die durch diese Methode bei der Generierung der HTML-Seite aufgerufen wird. Eine solche Methode kann folgende Form besitzen:

```
package tis20.impl;

public class NetworkParmsImplHtmlBeanInfo
    implements com.sun.jaw.impl.adaptor.html.HtmlStreamableIf
   public String WriteToHtml(Object o)
      String returnString = ((NetworkParmsImpl)o).printIpAddress()+"<BR>"
         + ((NetworkParmsImpl)o).printSubnetMask()+"<BR>";
      return ( returnString);

   public boolean isCustomizedViewOnly ()
      return false;
```

Die Auswirkung auf die HTML-Seite nach Hinzunahme einer benutzerspezifischen Ergänzung stellt sich wie in Abbildung 6.12 dar.

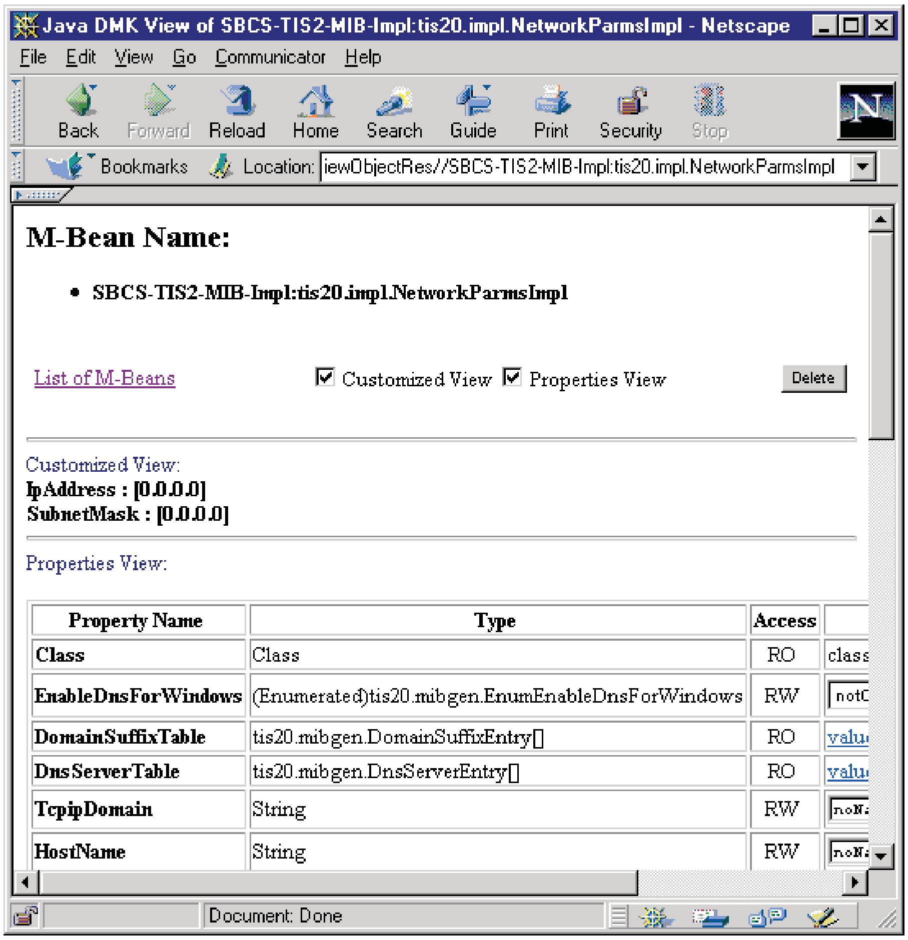

Abbildung 6.12: NetworkParmsImpl mit Ergänzung

6.7.2 Manager-Applikationen

Eine weitere Methode, auf JDMK-Agenten zuzugreifen ist, eine Client- oder Managerapplikation zu entwickeln. Diese Applikation kann entweder in Java oder C++ geschrieben sein. Im zweiten Fall ist jedoch der Zugriff auf JDMK-Agenten über das Java Native Interface durchzuführen. Eine einfache Client-Java-Applikation kann folgende Gestalt besitzen:

```
package trial;

import java.net.*;
import java.util.*;
```

```
// jaw import
import com.sun.jaw.reference.common.*;
import com.sun.jaw.impl.adaptor.rmi.*;
import com.sun.jaw.impl.agent.services.monitor.*;
import trial.monitor.ClientListener;

public class Client implements Runnable {

  // CONSTRUCTOR
  //------------
  public Client() {
  }

  /**
   * It is possible to specify on the command line the implementation to use
   * for the adaptor function.
   */
  public static void main(String argv[]) {
    System.exit( new Client().execute(argv) ) ;
  }

  /**
   * When an object implementing interface Runnable is used to create a thread,
   * starting the thread causes the object's run method to be called in that
   * separately executing thread.
   */
  public void run() {
    String[] s = new String[0];
    execute(s);
  }

  // PRIVATE METHODS
  //----------------
  private int execute(String argv[]) {
    try {
      // Set the host name of the remote agent.
      //
      String agentHost = InetAddress.getLocalHost().getHostName();
      if (argv.length >= 1) {
         agentHost = argv[0];
      }
      // Set the port number of the remote host.
      //
      int agentPort = 1096;
      if (argv.length >= 2) {
         agentPort = Integer.decode(argv[1]).intValue();
      }
      // Create an adaptor client to enable the client to manage the remote agent
      // and initializes the communications with the remote agent.
      //
      trace(">>> CREATE and INITIALIZE communication with the remote agent,");
      trace("HOST     = " + agentHost);
      trace("PORT     = " + agentPort);
      trace("PROTOCOL = RMI");
      trace("SERVER   = " + ServiceName.APT_RMI);
      AdaptorClient adaptor = new AdaptorClient();
      adaptor.connect(null, agentHost, agentPort, ServiceName.APT_RMI);
      trace("Communication ok");
      // Get the domain name from the remote agent.
      //
      String domain = "SBCS-TIS2-MIB-Impl";
      // Get an instance of the GkStatistics m-bean in the
      // remote agent.
      //
      String NetworkParmsImplClass = "NetworkParmsImpl";
```

```
      ObjectName NetworkParmsImplName
        = new ObjectName(domain + ":tis20.impl.NetworkParmsImpl");
      System.out.println("Get an instance of NetworkParmsImpl in the remote object server");
      System.out.println("with the object name -> " + NetworkParmsImplName.toString());
      tis20.impl.NetworkParmsImplMO NetworkParmsImpl
        = (tis20.impl.NetworkParmsImplMO)
          ((Vector)adaptor.getObject(NetworkParmsImplName, null)).firstElement();
      System.out.println("ok");

      String[] test = NetworkParmsImpl.performStringTableValues("DomainSuffixTable",1,1);

      // Terminate communication with the remote agent.
      //
      adaptor.disconnect();
      return 0;
    } catch(Exception e) {
      trace("Got an exception !");
      e.printStackTrace();
      return 1;
    }
  }

  /**
   * Display trace messages.
   */
  private void trace(String msg) {
     System.out.println(msg);
  }
}
```

6.7.3 SNMP-MIB-Browser/SNMP-Manager

Über den SNMP-Adapter des JDMK kann ein SNMP-MIB-Browser oder ein SNMP-Manager auf den Agenten zugreifen. Die Besonderheit hierbei ist, daß kein AdaptorClient (vgl. 3.6 notwendig ist (Außer bei HTML ist dies bei jedem anderen Adapter der Fall). Es wird intern im Agenten eine MIB-Struktur aufgebaut. Die geschieht sukzessive durch Einlesen und Registrieren der M-Beans, welche die MIB repräsentieren und über diese werden die Request bzw. die dabei auftretenden OID´s verarbeitet.

6.7.4 JDMK-Tool "job"

Von JDMK wird eine graphische Managementumgebung mitgeliefert, welche den Zugriff auf JDMK-Agenten ermöglicht. Es können Adapter über den Discovery Service (vgl. 3.7.12) ermittelt und anschließend ausgewählt werden. Nach der Auswahl wird der Verbindungsaufbau zu dem entsprechenden Adapter versucht und bei Erfolg werden die M-Beans des Agenten angezeigt. Genauere Informationen sind unter [jdmk3b] zu finden.

6.8 Die Agenten

Für Testzwecke wurde die Agentenstruktur nicht auf einen Prototypen beschränkt, sondern es werden drei Agenten verwendet. Die Verwendung geschieht folgendermaßen:

- Master Agent
  Dieser Agent wird benutzt, um den Cascading Service (vgl 3.7.13) des JDMK zu testen. er besitzt die beiden anderen Agenten als "Remote Agents". Er besitzt wie alle anderen Agenten ein Discovery-Responder-Objekt, um bei späteren Tests mit dem Discovery

Service (vgl 3.7.12) ansprechbar zu sein. Zusätzlich wird zur Startzeit mittels M-Let-Service (vgl 3.7.5) ein Discovery-Monitor-Objekt eingebunden, um Events, welche von Agenten ausgesendet werden, zu überwachen. Schließlich wird mit diesem Agenten der Discovery-Search-Service geprüft, welcher die JDMK-Agenten ermitteln kann, die über einen Discovery Responder verfügen. (vgl 3.7.12).

- FirstSubAgent
  Dieser Agent wird für die Integration der implementierten MIB-M-Beans, welche die generierten Klassen ersetzen, benutzt. Die derzeitig verwendeten Klassen umfassen die Gruppen:
  - NetworkParms
  - PlatformMisc
  - GkStatistics
  - GwH323StackParms
- SecondSubAgent
  Dieser Agent beinhaltet die generierten und noch nicht ersetzten M-Bean-Klassen. Diese Klassen besitzen Dummy-Werte, werden aber dennoch verwendet, um bestimmte Dienste wie den Cascading Service testen zu können.

Der Programmcode der Agenten ist in Anhang F zu finden. Die so entstandene Struktur ist in Abbildung 6.13 dargestellt.

6.9 Die Clients

Es wurden verschiedene Client-Applikationen zu Testzwecken entwickelt. Diese Applikationen sind derzeit als Shell-Skripten ausgeführt, jedoch wurde damit begonnen, eine graphische Oberfläche für diese Applikationen zu schaffen. Insgesamt wurden für folgende Teilbereiche Client-Applikationen geschaffen:

- Event-Modell
- Discovery Service
- AlarmClock Service
- Monitoring Service (Gauge/Counter)
- Security Service
- CORBA-Zugriff

6.10 Dynamisches Verhalten des Prototypen

Nun soll das Verhalten des Prototypen bei Zugriffen über die Adapter durch Manager und andere Agenten (bzw. M-Beans) dargestellt werden. Es werden Managementaktionen wie das Setzen und Lesen von MIB-Variablen, das Anlegen und Löschen von Tabelleneinträgen sowie das Anstoßen von Aktionen, welche M-Beans ausführen, dargestellt. Zur Darstellung werden

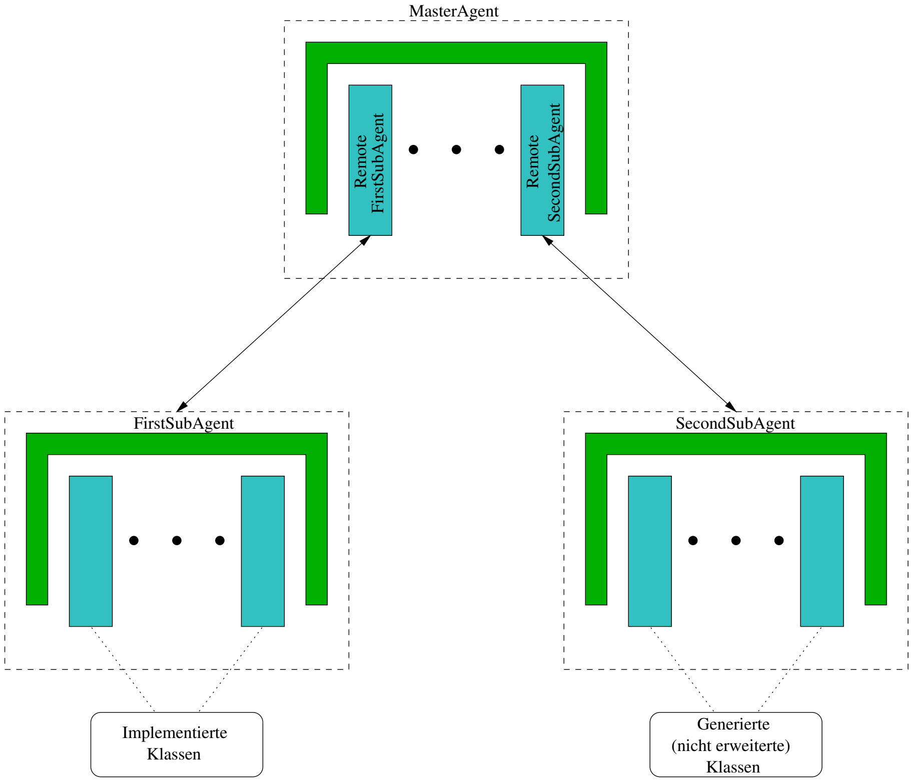

Abbildung 6.13: Aufbau der Agenten-Hierarchie

Sequenz-Diagramme verwendet. Nachfolgend sind die Initialisierungsphase des Agenten sowie Aktionen, die während der Laufzeit des Agenten ausgeführt werden aufgeführt:

- GET-Request (vgl. Abbildung 6.14)
  Dargestellt wird der Zugriff auf eine beliebige M-Bean des Agenten. Es wird der Weg des Aufrufs bis zu den Applikationen und zurück betrachtet.

- SET-Request (vgl. Abbildung 6.15)
  In dieser Abbildung wird ein beliebiger Zugriff auf eine MIB-Variable der TIS-MIB dargestellt.

- CREATE-Request für Tabelleneintrag (vgl. Abbildung 6.16)

- REMOVE-Request für Tabelleneintrag (vgl. Abbildung 6.17)

- PERFORM (M-Bean Methode)
  Diese Art des Zugriffs ist bei Abbildung 6.16 verwendet worden. Es wird ein Zugriff auf die M-Bean-Methode `performCreateDnsServerEntry` dargestellt.

- Initialisierung Prototyp
  In Abbildung 6.18 und 6.19 wird die Initialisierungsphase und die dabei ausgeführten Methodenaufrufe dargestellt. Im ersten Teil ist die Initialisierung bis zu den TIS-M-Beans beschrieben. Der zweite Teil stellt die Initialisierung der TIS-M-Beans dar. Diese Phase wird beispielhaft anhand der M-Bean `NetworkParms` und einer ihrer Tabellen (`DnsServerTable`) vorgestellt. Es wird deutlich, wie die einzelnen Tabelleneinträge die Native-Schnittstelle nutzen, um die Werte der MIB-Variablen aus der TIS-Managementumgebung zu erhalten.

Es sei darauf hingewiesen, daß die Darstellung der Diagramme bewußt einfach und schematisch gewählt wurde, um einen Einblick in die Funktionsweise des Prototypen bei Zugriffen durch Managementapplikationen zu gewähren. Diese Diagramme erheben keinen Anspruch auf Vollständigkeit.

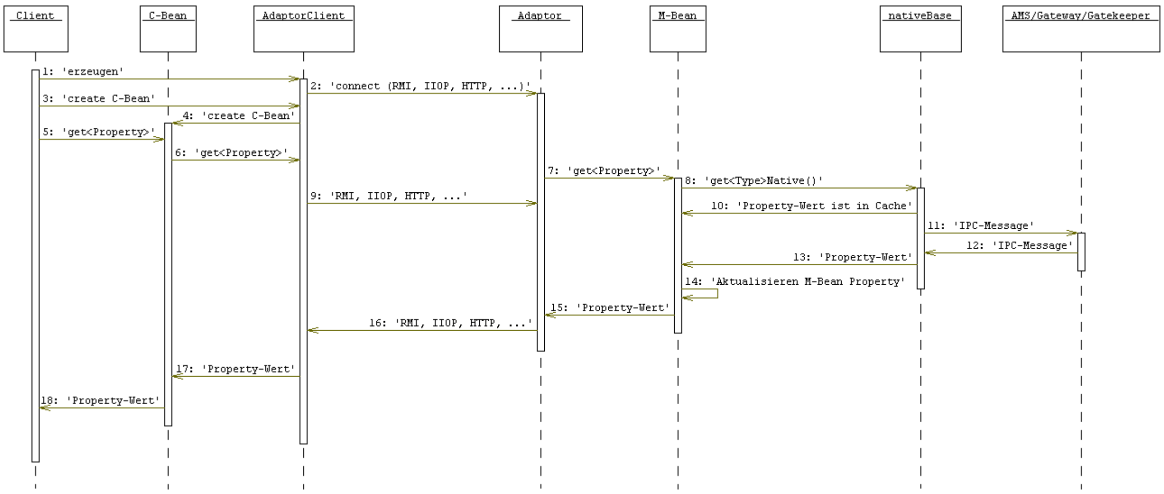

Abbildung 6.14: Sequenzdiagramm eines GET-Zugriffs

6.11 CORBA-Variante des Prototypen

Für Testzwecke wurde damit begonnen, eine CORBA-Variante des Prototypen zu erstellen. Diese Variante beinhaltet die Implementierung einer IDL-Schnittstelle für die einzelnen M-Beans des Agenten, um diese über einen Standard-CORBA-Client ansprechen zu können in Anhang D ist eine solche Schnittstelle, wie sie für die MIB-Gruppe "GkStatistics" bereitgestellt wird, aufgeführt. Die Zugriffe beschränken sich derzeit lediglich auf einfache MIB-Variablen (keine Tabellen), da mit dieser Variante lediglich qualitativ die in Kapitel 5.2 eruierten Möglichkeiten getestet werden sollten.

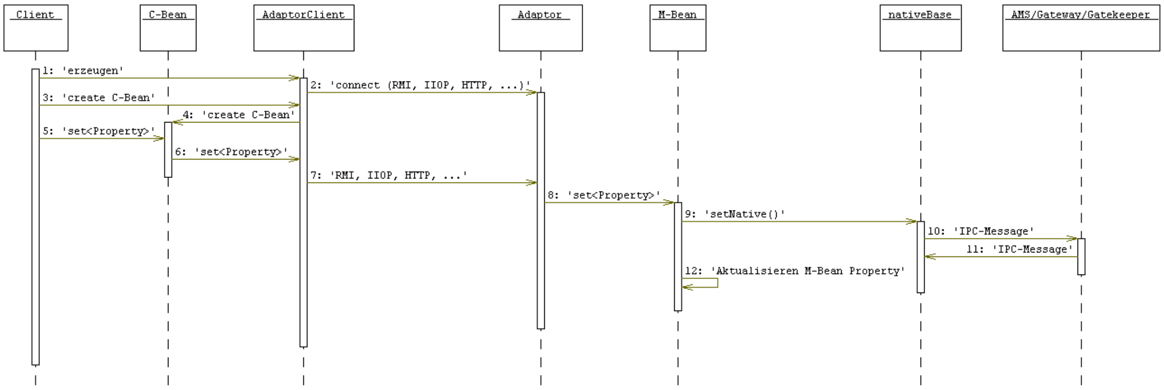

Abbildung 6.15: Sequenzdiagramm eines SET-Zugriffs

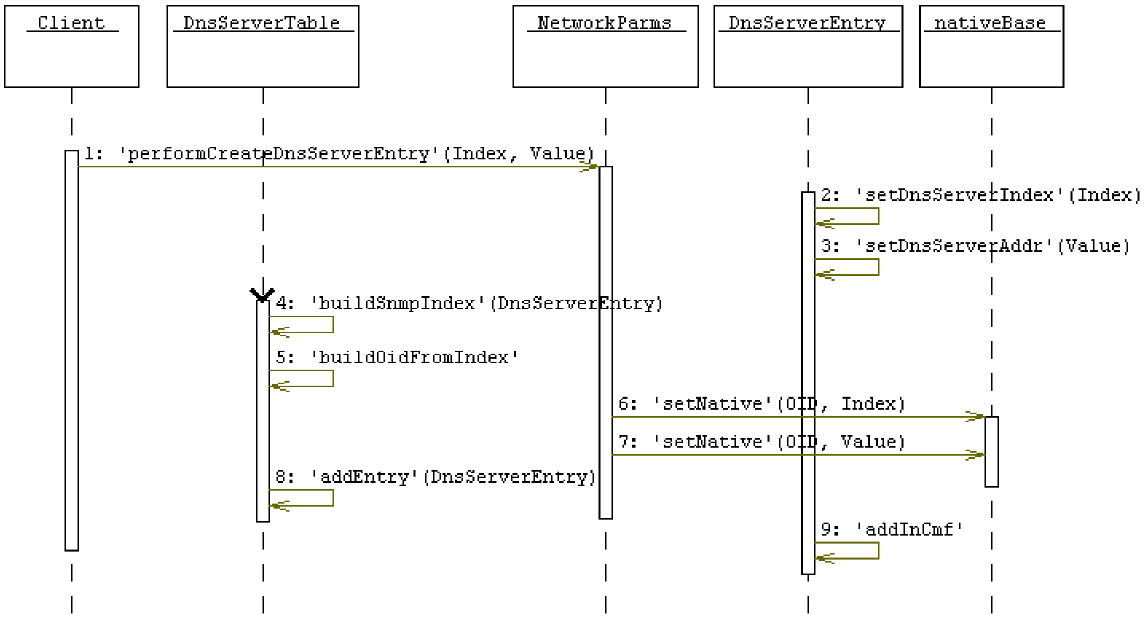

Abbildung 6.16: Sequenzdiagramm bei Erstellung eines Tabelleneintrags

6.12 JNI - Java Native Interface

Das Java Native Interface stellt die Schnittstelle zwischen Java-Programmen und proprietären Libraries her. Damit ist es möglich aus einem Java Programm, welches innerhalb der Java Virtual Machine läuft, plattformspezifische Funktionen aufzurufen. Umgekehrt kann ein pro-

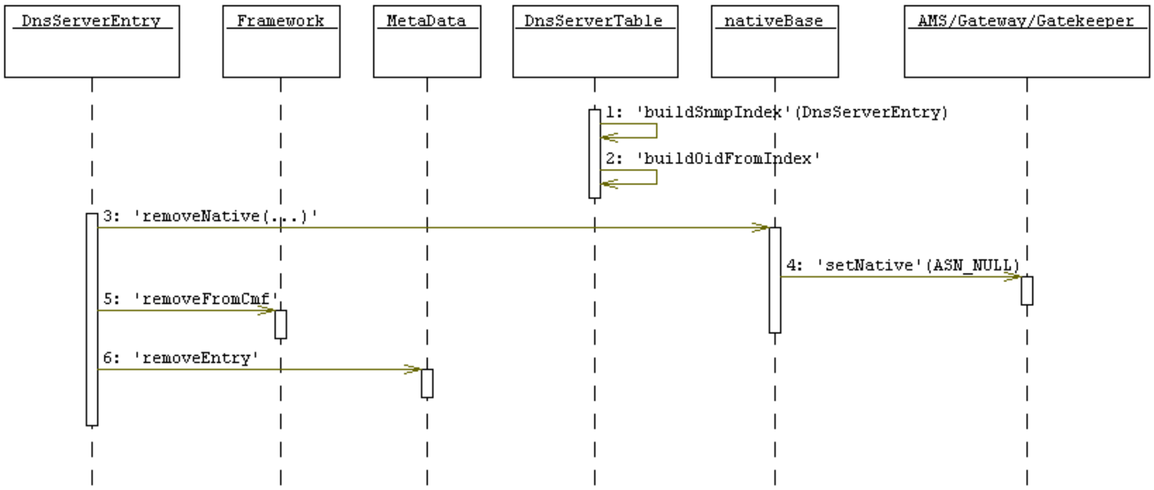

Abbildung 6.17: Sequenzdiagramm bei Entfernen eines Tabelleneintrags

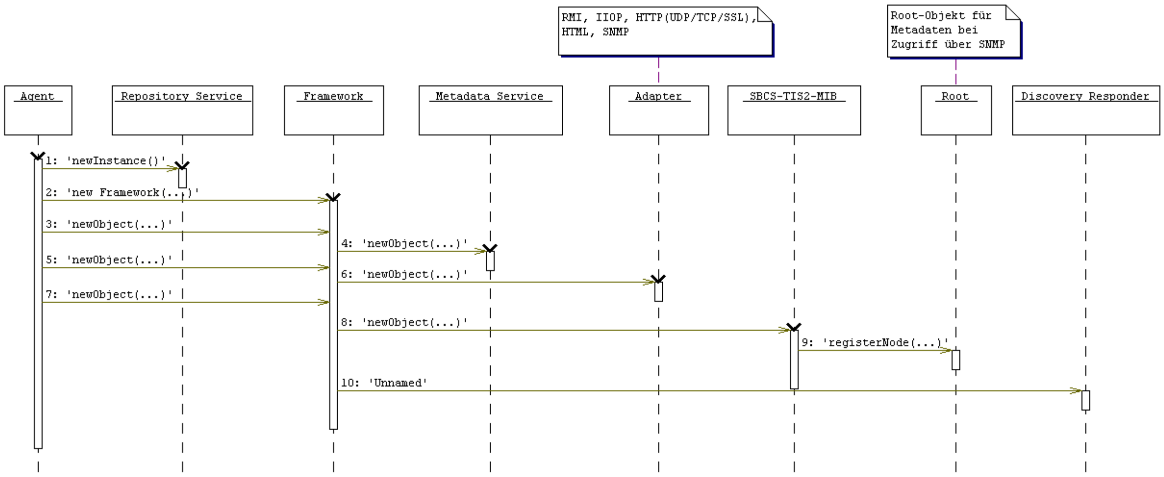

Abbildung 6.18: Sequenzdiagramm bei Initialisierung des Agenten I

prietäres Programm über die Invocation API auf Java Programme zugreifen. Dieses Interface kommt zum Tragen, falls:

- eine Bibliothek, welche z.B. in C++ geschrieben wurde, weiterhin verwendet werden soll.
- Befehle und Funktionen unter Java nicht implementiert werden können oder sollen.

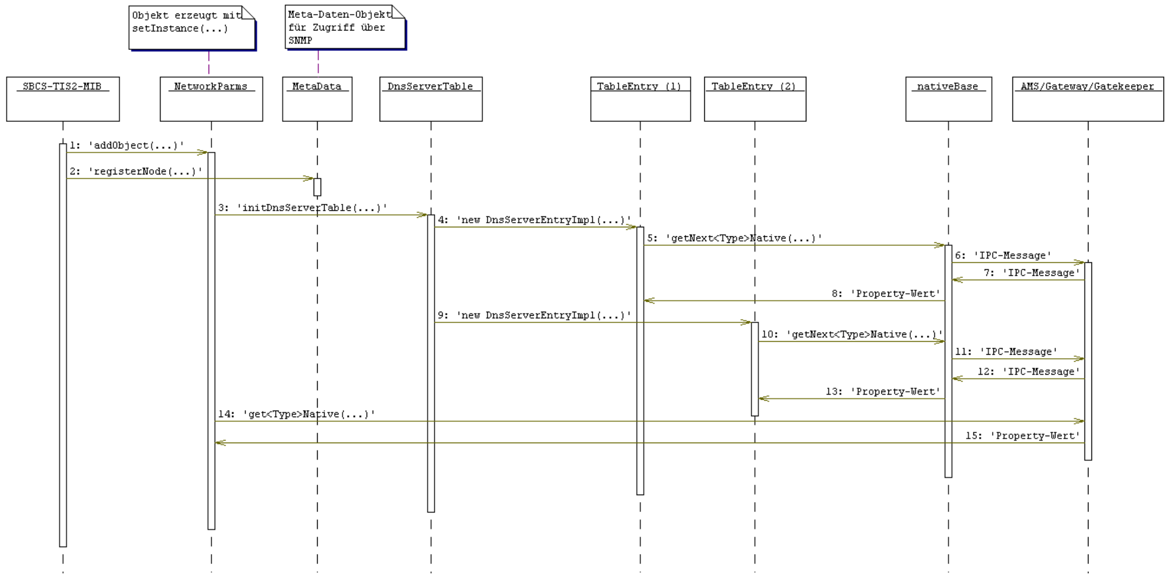

Abbildung 6.19: Sequenzdiagramm bei Initialisierung des Agenten II

- Programmteile (z.B. zeitkritische Elemente) besser in einer anderen Sprache wie Assembler oder C zu realisieren sind.

Der besondere Vorteil dieses Interfaces liegt darin, daß rechnerspezifische Methoden Java Objekte erzeugen und zerstören, auf Java-Methoden zugreifen und Parameter austauschen können. Selbst das Exception-Handling kann zwischen proprietären und Java-Implementierungen durchgereicht werden, falls dies erforderlich sein sollte.
In Abbildung 6.20 wird schematisch der Aufbau einer Library, welche das Interface beinhalten soll, dargestellt. Die einzelnen Schritte werden in den nachfolgenden Unterkapiteln genauer ausgeführt. Zudem werden Regeln und Vorschriften bei der Erstellung eines Interfaces dargestellt.

6.12.1 Deklaration rechnerspezifischer Methoden

Die Deklaration dieser Methoden muß von zwei Seiten betrachtet werden.

Deklaration auf der Java-Seite

Bis auf zwei Unterschiede wird die Deklaration von rechnerspezifischen Methoden auf der Java Seite analog zu den üblichen Java-Methoden durchgeführt. Die Unterschiede sind folgende:

- Kennzeichnung durch das Schlüsselwort `native`.
- Die Deklaration wird mit einem Semikolon abgeschlossen, da auf der Java-Seite keine Implementierung der Methode vorliegt.

Die Deklaration ist in Abbildung 6.20 unter Schritt 1 zu ersehen.

Deklaration auf der proprietären Seite

Die Realisierung der Methode auf proprietärer Seite muß einmal im Header-File und in einer Implementierung geschehen. Das Header-File wird unter Zuhilfenahme des Java Development Kit erzeugt. Einmal wird das .java File mit dem `javac` Compiler übersetzt und schließlich das .h File mit javah -jni <Dateiname> erzeugt. Die Struktur und die Einträge, welche in dem .h File gemacht werden sind exemplarisch in Abbildung 6.20 abgebildet. Im wesentlichen ist darauf zu achten, wie der Prototyp der Methoden deklariert wird. Es ist darauf zu achten, daß JNICALL und JNIEXPORT immer dann in den Methodenprototyp eingebunden werden sollen, wenn der Code auf Plattformen wie Win32 übersetzt wird. Diese Bezeichner stellen den Export und Import spezieller Schlüsselwörter oder Funktionen zwischen DLL's zur Verfügung. Der Bezeichner setzt sich im wesentlichen aus 4 Elementen zusammen:

- Das Präfix `Java_`
- Der vollständige Klassenbezeichner
- Trennung der einzelnen Elemente durch "_"
- Der Methodenbezeichner[15]

Zusätzlich zu den benutzerspezifischen Parametern werden zwei zusätzliche Parameter bei der Übergabe benötigt. Der erste Parameter `JNIEnv*` stellt den JNI Interface Pointer dar. Der Pointer verwaltet eine Funktionstabelle, in welcher die rechnerspezifischen Methoden als ein jeweiliger Eintrag aufgeführt sind. Der andere Parameter `jobject` stellt die Referenz auf das Objekt selbst dar (ähnlich dem `this` Pointer unter C++).
In Abbildung 6.20 ist die Implementierung einer rechnerspezifischen Methode zu finden. Dort wird bei den `(*env)->` Aufrufen deutlich, wie das Interface genutzt wird, um den Datenaustausch zwischen der Methode und der Java Virtual Machine (JVM) zu realisieren.
Innerhalb der Header-Datei, welche für eine proprietäre Schnittstelle benötigt wird, werden für den späteren Gebrauch die Typ-Signaturen der übergebenen Parameter angegeben. Die Elemente, aus denen die Signaturen aufgebaut werden sind in Tabelle 6.1 aufgeführt.

6.12.2 Mapping zwischen Java und rechnerspezifischer Methoden

Der Datenaustausch zwischen der JVM und den rechnerspezifischen Methoden kann unter drei Gesichtspunkten geschehen:

- Zugriff auf Java-Parameter durch rechnerspezifische Methoden.
- Instantiierung eines Java-Objekts innerhalb einer rechnerspezifischen Methode.
- Übergabe von Ergebnisparametern aus rechnerspezifischen Methoden.

Hierbei ist zu beachten, daß nur auf primitive Datentypen zugegriffen werden kann. Es wird der entsprechende Parameter (z.B. `boolean`) in den rechnerspezifischen Datentyp (z.B. jboolean) gemappt. Die gesamten Mappingregeln sind in Tabelle 6.2 dargestellt. Bei der Übergabe von Parametern ist zu beachten, daß lediglich der "passed by reference" Mechanismus unterstützt wird. Alle Parameter sind in erster Linie vom Typ `jobject`. Es werden jedoch noch Verfeinerungen (z.B. `jclass`, `jstring` etc.) zur Verfügung gestellt.

[15] Bei überladenen Methoden wird hinter dem Methodenbezeichner "__" angehängt.

Signatur	Java Datentyp
Z	boolean
B	byte
C	char
S	short
I	int
J	long
F	float
D	double
L ⟨Klassenbezeichner⟩;	⟨Klassenbezeichner⟩
[⟨Datentyp⟩	⟨Datentyp ⟩[]
(Argumenttypen) Rückgabetypen	Methodenstruktur

Tabelle 6.1: Die Typ-Signaturen des Java Native Interface

Java Datentyp	Rechnerspezifischer Datentyp	Größe (Bit)
boolean	jboolean	8, vorzeichenlos
byte	jbyte	8
char	jchar	16, vorzeichenlos
short	jshort	16
int	jint	32
long	jlong	64
float	jfloat	32
double	jdouble	64
void	void	

Tabelle 6.2: Mapping zwischen Java- und rechnerspezifischen Parametern

6.12.3 Zugriff auf Java Objekte

Die einzelnen Unterklassen von `jobject` besitzen alle ihre eigenen Zugriffsmethoden. Die Klassen und deren Methoden werden nachfolgend genauer erläutert. Eine Klassenstruktur ist in Abbildung 6.21 zu finden.

Strings

Strings werden mit der Unterklasse jstring an Methoden übergeben. Es ist zu beachten, daß sich dieser Typ vom klassischen C-String (char*) unterscheidet. Er kann z.B. nicht mit einem einfachen printf ausgegeben werden. Um also Strings verwenden zu können, müssen JNI-Funktionen eine Konvertierung durchführen. Es ist eine Konvertierung zwischen Java und UTF-8 Strings möglich. UTF-8 Strings sind 0-terminiert und beinhalten ebenso den 7-Bit ASCII Zeichensatz. In Tabelle 6.3 sind die Funktionen und deren Wirkungsweise aufgeführt. Es ist zu beachten, daß bei den Funktionen wie `GetStringChars` als Parameter der JNIEnv Interfacepointer als erster Parameter mit übergeben werden muß. Die Verwendung dieser Funktionen ist in Abbildung 6.20 dargestellt.

Funktionsbezeichner	Wirkungsweise
`GetStringChars`	Umwandlung eines Java Strings (Unicode) in eine UTF-8 Repräsentation
`ReleaseStringChars`	Freigeben des Pointers auf das String-Objekt. Dies ist notwendig, um die JVM darüber zu informieren, daß der Speicherbereich wieder freigegeben werden kann. (Keine automatische garbage collection)
`NewString`	Instantiierung eines String-Objekts
`GetStringLength`	Länge eines Strings in Unicode-Format.
`GetStringUTFLength`	Länge eines Strings in UTF-8 Format.

Tabelle 6.3: String-Funktionen des Java Native Interface

Arrays

Das Äquivalent zu Java Arrays stellt in JNI die Unterklasse jarray dar. Ebenso wie bei jstring kann auf dieses Objekt nicht direkt zugegriffen werden. Es werden durch das JNI Funktionen zur Verfügung gestellt, welche den Zugriff auf die Elemente des Arrays ermöglichen. Die Funktionen sind in Tabelle 6.4 aufgeführt. Es sei noch erwähnt, daß Felder aus primitiven Datentypen wie in Tabelle 6.2 dargestellt, aber auch aus Objekten aufgebaut sein können.

Methoden

Aus der rechnerspezifischen Methode können Aufrufe auf statische und nicht-statische Java-Methoden durchgeführt werden. Der Aufruf von nicht-statischen Methoden geschieht in 3 Schritten:

- Aufruf der Funktion `GetObjectClass`. Diese Funktion liefert das entsprechende Java-Klassenobjekt.
- Aufruf von `GetMethodID`. Ermittlung der MethodID, liefert als Rückgabewert Null. Falls ein Wert ungleich Null zurückgegeben wird, wird eine `NoSuchMethodError` Exception geworfen.
- Aufruf der Java-Methode durch `Call<Type>Method`. Es werden das Objekt, die Method-ID und die Parameter übergeben.

Felder

Für den lesenden und schreibenden Zugriff auf Java Felder müssen zur Bestimmung der Elemente des Feldes zwei Schritte durchgeführt werden:

- Ermittlung der ID des Feldes

Funktionsbezeichner	Wirkungsweise
Get/Set<Datentyp>ArrayElements	Zugriff auf das Java Array. Es ist zu beachten, daß dieses Array nicht verschiebbar realisiert ist ("Normale" Array werden bei garbage collection durch die JVM unter Umständen verschoben), so daß der Zugriff während des gesamten Lebenszyklusses gewährleistet ist.
Get/Set<Datentyp>ArrayRegion	Da die "normalen" Get/Set Funktionen generell das gesamte Array kopieren, kann es notwendig sein, die Größe des Arrays zu begrenzen. Die kann durch Reduzierung der Elementanzahl erreicht werden.
Release<Datentyp>ArrayElements	Wie bei den Get/Set Funktionen angedeutet, werden die Arrays nicht verschiebbar realisiert. Daher müssen sie nach Verwendung wieder explizit freigegeben werden.

Tabelle 6.4: Array-Funktionen es Java Native Interface

- Ermittlung des Objekts mit oben ermittelter ID

```
// Bestimmung der ID eines static definierten Feldes
jfieldID fieldID = env->GetStaticFieldID(class, "intField", "I");

// Bestimmung der ID eines nicht static definierten Feldes
jfieldID fieldID = env->GetFieldID(class, "intField", "I");

// Bestimmung des Field-Objektes mit oben ermittelter ID
jint field = env->GetStaticIntField(class, fieldID);
```

Exception Handling

Das Java Native Interface ermöglicht es, Exceptions innerhalb der native Methoden zu werfen und auch zu verarbeiten. Zusätzlich ist es jedoch möglich, Exceptions zur Java Virtual Machine weiterzureichen. Hierbei wird der Typ `jthrowable` verwendet und im Falle einer Exception wird diese durch die Methode `ExceptionOccured` zur JVM weitergereicht. Nachfolgend ist ein Beispiel für die Verwendung des `jthrowable`-Objekts zu sehen:

```
// Exception-Handling
jthrowable exception;
jclass NewException;
...
exception = env->ExceptionOccured();
if (exception) {
  env->ExceptionDescribe();
  env->ExceptionClear();
  newException = env->FindClass("java/lang/NullPointerException");
```

```
  if (NewException == 0) return; // Exception wird von Java nicht unterstützt
  env->ThrowNew(newException, "JNI-Exception!");
 ...
```

Lokale und Globale Referenzen

Die Methoden, die innerhalb des JNI verwendet werden, erstellen für alle verwendeten Objekte lokale Referenzen. Dies gilt für die Parameter, welche übergeben werden, die Rückgabewerte wie auch für die Objekte, welche innerhalb der Routine instantiiert werden. Damit werden diese Speicherbereiche nach Verlassen der Methode wieder freigegeben und die übergebenen Objekte werden der JVM-Garbage Collection übergeben. Ist es aber notwendig, eine Datenstruktur über die Lebensdauer einer Methode hinaus zu verwenden, so muß eine globale Referenz eingeführt werden. Ist eine Datenstruktur erst außerhalb der Methoden statisch definiert, so kann die globale Referenz innerhalb der Methoden durch die Methoden `NewGlobalRef` angelegt und durch `NewGlobalRef` wieder freigegeben werden.

Threads und rechnerspezifische Methoden

Da Java Multithreading unterstützt, muß bei der Verwendung von JNI-Methoden damit gerechnet werden, daß es zu Race-Conditions und Deadlocks zwischen verschiedenen Threads, welche dieselbe JNI-Methode benutzen, führen kann. Im Wesentlichen müssen drei Dinge beachtet werden:

- Der JNI-Interface Pointer ist nur für einen Thread gültig. Dieser wird bei einem wiederholten Aufruf des Threads wieder verwendet. Es werden ein Interface-Pointer pro Thread genutzt.
- Lokale Referenzen sollen nicht zwischen Threads, welche auf dieselbe Methode zugreifen, verwendet werden. Da es nicht bestimmt werden kann, wann diese Objekte wieder freigegeben werden, sollte in diesem Fall eine globale Referenz erzeugt werden.
- Der gemeinsame Zugriff auf globale Variablen sollte synchronisiert werden.

In Java ist zur Synchronisation von Threads das Schlüsselwort `synchronized` zu verwenden. Das JNI unterstützt diese Funktionalität durch die Methoden `MonitorEnter` und `MonitorExit`, welche den kritischen Bereich eingrenzen. Es werden die Anzahl der Zugriffe eines Threads festgehalten und durch `MonitorEnter` inkrementiert bzw. durch `MonitorExit` decrementiert. So können Threads in den kritischen Bereich eintreten, wenn der entsprechende Zähler wieder auf 0 gesetzt ist.

JNI und Java Virtual Machine

Das JNI bietet die Möglichkeit, die JVM zu laden, instantiieren und zu starten. (`java` ist im Wesentlichen auch ein C-Programm, welches die übergebenen Argumente parst und die JVM mittels der Invocation API startet).

6.13 Erfahrungen

Das Java Dynamic Management Kit bietet in seiner derzeitigen Ausprägung neben seinen Vorzügen auch teilweise erhebliche Mängel, welche sich über die gesamte Architektur erstrecken.

An dieser Stelle soll auf die Schwachstellen des JDMK und auf eventuelle Konsequenzen, welche sich daraus ergeben, hingewiesen werden.

6.13.1 Dienste

Security Service

Als erster Schachpunkt sei hier zu nennen, daß es keine Möglichkeit gibt, den Zugriff auf einzelne M-Beans eines Agenten zu sichern. Wenn eine Management-Applikation die Zugriffsberechtigung über einen Adapter erhalten hat (oder den Zugriff über einen ungeschützten Adapter erhält), dann kann diese Applikation auf jedes M-Bean des Agenten zugreifen, ohne sich nochmals authentifizieren zu müssen. Es ist also nicht möglich, einzelne M-Beans mit Zugriffsschutz zu versehen. Eine Möglichkeit wäre, wenn auch etwas umständlich, einen eigenen Agenten für dieses M-Bean bzw. eine Gruppe von M-Beans für die entsprechender Zugriffsschutz gewünscht ist, zu verwenden.

Cascading Service

Außer der in Kapitel 6.13.1 beschriebenen Sicherheitsmängel ist in der Testphase ein Problem mit dem Cascading Service aufgetreten, welches nicht bestimmt werden konnte. Das Problem beruhte darauf, daß auf der Master-Agenten-Seite bei "Aktivierung" des Sub-Agenten nur eine Teilmenge der M-Beans des Agenten in das CMF des Master-Agenten geladen wurden.
Bei Zugriff über den HTML-Adapter und die damit verbundenen generierten Seiten, über welche auf die M-Beans zugegriffen wird, stellte sich folgendes Problem dar. Falls für HTML-Seiten benutzerspezifische Erweiterungen[16] implementiert wurden, werden diese bei Verwendung des Cascading Service nicht berücksichtigt.

Discovery Service

Der Discovery Service besitzt in der Version 3.0 den Fehler, daß nur Adaptoren, welche als Domäne die Bezeichnung "defaultDomain" besitzen von dem Discovery Search Service erkannt werden können. Es war aus den zur Verfügung stehenden Unterlagen nicht zu ersehen, wie der Discovery Search Service angepaßt werden kann, um andere Domänen zu berücksichtigen.

Scheduler Service

Der Scheduler Service unterstützt keine Kalenderfunktionalität. Es ist somit nicht möglich, immer wiederkehrende Events (z.B. jeden Montag um 11:00 Uhr) zur erzeugen.

Dienstverträglichkeiten

Es ist nicht möglich, den M-Let Service mit einem persistent Repository zu verbinden. Es ist entweder möglich, die Klassen zu Startzeit des Agenten über den M-Let Service zu laden oder den Stand vor dem letzten Beenden des Agenten wiederherzustellen.

[16] Die Möglichkeit, benutzerspezifische Anpassungen an die generierten HTML-Seiten zu implementieren oder die generierten Seiten durch benutzerspezifische Seiten gänzlich zu ersetzen, wird in Kapitel 6.7.1 genauer eingegangen.

6.13.2 Protokolle

Auch wenn JDMK eine Reihe von Protokollen zur Verfügung stellt und damit eine große Bandbreite der Kommunikation eröffnet, sind einschränkende Umstände festzustellen. Der IIOP-Adapter bietet lediglich die Möglichkeit, generische Methoden auf einen Agenten bzw. dessen M-Beans anzuwenden. Es ist einem CORBA-Client nicht möglich auf wie unter CORBA üblich auf M-Beans zuzugreifen. Dies würde bedeuten, daß der Agent die Methoden der M-Beans über einen Naming Service zur Verfügung stellt. Der Zugriff auf die Methoden eines M-Bean erfordert die Kenntnis der Klassenstruktur des jeweiligen M-Beans. Dies widerspricht der Voraussetzung, daß Systemwissen unter CORBA nicht notwendig ist. Es ist zwar möglich, einen Weg für einen direkten Zugriff zu finden (vgl. 5.2), jedoch eröffnet dieser Weg neue Problematiken und Komplikationen.
Die Zugriffsmöglichkeiten über HTML und die automatische Generierung der HTML-Seiten für das Management der M-Beans ist hilfreich, jedoch ist die Authentifizierung über den HTML-Adapter im Klartext durchzuführen und somit außer zu Testzwecken unbrauchbar. Eine Möglichkeit ist die Verwendung von HTTP/SSL. Dieser Adapter unterstützt die CRAM-MD5 Authentifizierung und bietet eine sichere Übertragung der Paßwörter.

6.13.3 JDK 1.2

Die Verträglichkeit von JDMK 3.0 mit JDK 1.2 ist nicht sichergestellt. Es wird von Sun die Mitteilung ausgegeben, daß keine Unterstützung bei Problemen, welche durch die Verwendung von JDK 1.2 auftreten, gewährt wird. Es wird jedoch der Hinweis ausgegeben, daß sich die Problematik auf den IIOP-Adapter und den Sicherheitsdienst (HTTP/SSL) und M-Let(secure) beschränkt und bei Vermeidung dieser Punkte JDK 1.2 verwendet werden kann. Eine Anpassung an JDK 1.2 ist mit der JDMK 3.1 Version geplant, ein Release-Datum ist jedoch nicht bekannt.

1. Schritt : Erstellung des Java-Codes

Proprietäre Methoden werden mit "native" gekennzeichnet

nativeBase.java

```
public static class nativeBase {
    public native void getStringNative();
    static {
        System.loadLibrary("nativeBase.dll");
    }
}
```

Prop. Methoden werden in Libraries (z. B.".h" in C) zusammengefasst und an dieser Stelle geladen

client.java

```
class client {
    public static void main(String[] args){
    nativeBase native = new nativeBase();
    String intArray = "1.3.6.1.4.1.2206.5.1.1.1.1";
    String value = native.getStringNative(intArray);
    }
}
```

Im Java-Programm wird die Methode wie eine reguläre Java-Methode aufgerufen

2.Schritt : Kompilierung des Java-Codes

3.Schritt : Erstellung der Library (".h" ind C)

```
javah nativeBase.class
```

javah auf die erstellte Klasse anwenden um eine ".h" Datei zu erzeugen

```
#include <path>/nativeBase.h
```

In myclass.h mit #include die Library einbinden

4.Schritt : Erstellung der rechnerspezifischen Methode

".h"

```
/*
 * Class:     jni_nativeBase
 * Method:    getStringNative
 * Signature: ([I)Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_jni_nativeBase_getStringNative
  (JNIEnv *, jclass, jintArray);
```

```
#include <jni.h>
#include "nativeBase.h"
```

".cpp"

```
/*
 * Class:     jni_nativeBase
 * Method:    getStringNative
 * Signature: ([I)Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_jni_nativeBase_getStringNative
(JNIEnv *env, jclass, jintArray intArray) {
    RFC1157VarBind* varBindPtr = (RFC1157VarBind*) calloc(1,sizeof(RFC1157VarBind));
    char returnString[128];
    varBindPtr->name.idLength = env->GetArrayLength(intArray);
    varBindPtr->name.ids = (unsigned int*) env->GetIntArrayElements(intArray,0);
    ResolveVarBind(varBindPtr,ASN_RFC1157_GETREQUEST);
    sprintf(returnString, "%s", varBindPtr->value.asnValue.string.stream);
    return env->NewStringUTF(returnString);
}
```

5.Schritt : Erstellung einer shared library (Solaris) oder einer dll (Windows NT)

Abbildung 6.20: Verwendung des Java Native Interface

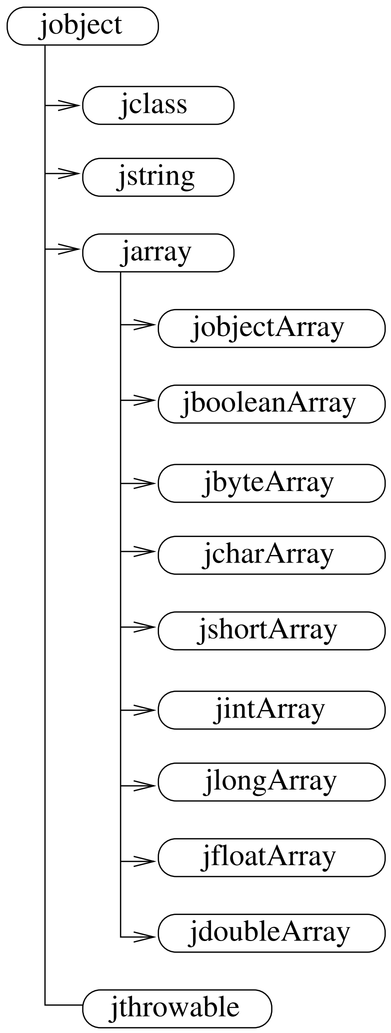

Abbildung 6.21: Die Klassenstruktur des Java Native Interface

Kapitel 7

Zusammenfassung und Ausblick

7.1 Zusammenfassung

Die Diplomarbeit umfaßt verschiedene Schwerpunkte. Neben den im Titel genannten Schwerpunkten waren begleitend weitere Themen von Interesse und notwendig. Die gesamte Themenpalette ist in nachfolgenden Unterkapiteln dargestellt.

7.1.1 Telephony Internet Server (TIS)

Der Telephony Internet Server der Firma Siemens ist eine Möglichkeit über das Internet Sprache und Daten zu übertragen. Hierbei werden Kommuniktionsmöglichkeiten wie Videokonferenz ermöglicht. Dies ist jedoch nicht nur zwischen speziellen Endgeräten der Fall, auch herkömmliche Telefonapparate sollen in der Lage sein, in eine Videokonferenz mit eingebunden zu werden. Dieser Server ist ein Zusatzelement zu der Telefonanlage Hicom 300 und steht parallel zu einem ATM-Switch als weitere Kommunikationskomponente zur Verfügung. Der TIS besteht aus einer Anzahl von Komponenten, welche in Kapitel 4 beschrieben werden. Diese Komponenten stellen Applikationen und Prozesse dar. Als wesentliche Applikationen wären hier das Gateway, der Gatekeeper und der Administration Maintenance Server (AMS) zu nennen. Die Aufgaben und der Aufbau sind in Kapitel 4 dargestellt. Hier sei bemerkt, daß das Gateway für die Umsetzung der eintreffenden Kommunikationswünsche, der Gatekeeper für die Bereitstellung der erforderlichen Bandbreite und der AMS für administrative Aufgaben vorgesehen ist. Für diese Komponenten war qualitativ zu eruieren, inwieweit JDMK für ein Management dieser Komponenten geeignet ist. Für einen Überblick war es notwendig, sich über die bestehende Managementstruktur zu informieren.
Die Struktur der bestehenden Managementumgebung, die verwendete Kommunikationsart und die Komponenten, welche innerhalb der Managementumgebung interagieren war hierbei zu analysieren. Hierbei stellte sich heraus, daß für das TIS-Management eine eigene Nachrichtenstruktur gewählt wurde. Diese Nachrichtenstruktur war für die Entwicklung des Prototypen von großer Bedeutung, da der Prototyp mit den Komponenten des TIS (Gateway,Gatekeeper etc.) notwendigerweise kommunizieren mußte. Damit war es erforderlich, den bestehenden SourceCode zu analysieren, um den Aufbau und die Handhabung der Nachrichten zu ermitteln. Der Aufbau der Nachrichtenstruktur und deren Besonderheit ist in Kapitel 4.6.1 beschrieben. Die bisherige Managementumgebung basiert auf der Erweiterung des Windows NT SNMP Agenten durch einen sogenannten "SNMP Extension Agent". Dieser ist in Kapitel 4.7.1 beschrieben und setzt SNMP-Requests in das spezielle Nachrichtenformat des TIS um. Dies

ermöglicht das Management über Applets bzw. MIB-Browser mittels SNMP-Requests. Da an dieser Stelle die Umsetzung der Requests geschieht, war hier der Ansatzpunkt für die Implementierung der Schnittstelle, wie sie in Kapitel 6.2.2 beschrieben wird.
Abschließend sei zu bemerken, daß für die Erprobungsphase des Prototypen bedauerlicherweise keine realistische Umgebung geschaffen werden konnte, da die Testumgebung nicht für Testzwecke verfügbar war. Zudem war es schwierig, den SourceCode für die Applikationen, speziell für den Gatekeeper einsehen zu können, da dieser Programmcode in den USA entwickelt wird und einer gewissen Geheimhaltung unterliegt. So mußte für die Testzwecke auf Zufallswerte zurückgegriffen werden.

7.1.2 Java Dynamic Management Kit (JDMK)

Das JDMK war die zweite zentrale Thematik der Diplomarbeit. Nach dem Testen der einzelnen Komponenten und Dienste, wie sie in Kapitel 3 beschrieben werden, war die Frage zu klären, inwieweit dieses Managementtool die Anforderungen, welche durch die Theorie der Flexiblen Management Agenten, wie sie in Kapitel 2 beschrieben werden, erfüllt. Die Anforderungen, welche in dieser Theorie formuliert werden und die Erfüllung durch JDMK sind in Tabelle 7.1 für die Systemeigenschaften und in Tabelle 7.2 für die Agenteneigenschaften gegenübergestellt. Falls die Eigenschaften von JDMK mit den Forderungen der Theorie der Flexiblen Management Agenten übereinstimmen, dann ist dies mit einem "+" gekennzeichnet, andernfalls zeigt ein "-" das Fehlen eines Mechanismusses auf Seiten von JDMK an. Ist die Eigenschaft "teilweise" erfüllt, so hat dies die Bedeutung, daß es zwar keinen expliziten Mechanismus in JDMK gibt, es jedoch möglich ist, mit der durch JDMK gelieferten API einen solchen Mechanismus zu realisieren.
Der zweite Punkt war die Realisierung eines Prototypen, welcher auf qualitative Art die Verwendbarkeit von JDMK für das Management des TIS prüfen sollte. An dieser Stelle war die Wahl des Design von zentraler Bedeutung. Es mußte beachtet werden, daß zum einen die TIS-MIB schon vorhanden war und sich noch in Veränderung befand (bzw. noch befindet) und somit ein Ansatz, welcher die Generierung von Dateien aus der MIB ermöglicht große Vorteile bringt. Die Wahl fiel auf das Generierungstool MIBGEN (vgl. Kapitel 3.8.2), welches von JDMK zu Verfügung gestellt wird. Dieses Tool generiert aus einer gegebenen MIB die erforderlichen Java-Dateien als M-Beans nach einem bestimmten Schema, welches in Kapitel 5.3.2 anhand des Prototypen erläutert und vorgestellt wird. In den dadurch erstellten Rahmen wurde die eigentliche Implementierung des Prototypen erstellt. Die Erstellung des Prototypen ist in Abbildung 7.1 zu sehen. Wie in der Abbildung zu ersehen, spielte die proprietäre Schnittstelle eine entscheidende Rolle bei der Integration des Prototypen in die bestehende TIS-Umgebung. Um diese Schnittstelle zu erstellen, war es notwendig, das Java Native Interface (JNI) zu verwenden. Diese API für die Erstellung von proprietären Schnittstellen, welche in C oder C++ erstellt sind, wird in Kapitel 7.1.3 einleitend vorgestellt. Um die Dienste anhand der TIS-Management-Umgebung zu testen war es notwending, einerseits Elemente der TIS-MIB zu überprüfen, anderseits künstlich eine Notwendigkeit einer JDMK-Dienstnutzung zu erzeugen. So wurde beispielsweise der Cascading Service (vgl. Kapitel 3.7.13) verwendet, um eine Master/Slave-Agentenhierarchie aufzubauen, auch wenn nicht explizit die Verwendung notwendig gewesen wäre. Der Aufbau der Agenten und deren Struktur ist in Kapitel 6.8 beschrieben.

Anforderungen des Systems			Erfüllung durch JDMK
System-eigenschaften	Organisatorische Merkmale	Agenten	+
		Flexible Management Agenten	+
		Manager	+
		Domänen	teilweise
		Gruppen	-
	Kommunikations-merkmale	Tasks	+
		Information	+
		Funktionalität	+
	Dienste	Delegation Service	+
		Event Service	teilweise
		Group Service	-
		Location Service	teilweise
		Security Service	teilweise
		Management Service	+

Tabelle 7.1: FMA Systemeigenschaften und JDMK

Anforderungen des Agenten	Erfüllung durch JDMK
Plattformunabhängigkeit	+
Flexibilität	+
Push/Pull Mechanismus	+
Management by Delegation	+
Kaskadierende Aufrufe	teilweise
Kooperationsfähigkeit	teilweise
Kontrollmechanismen	+
Module	+

Tabelle 7.2: FMA Agenteneigenschaften und JDMK

7.1.3 Java Native Interface (JNI)

Ein weiterer wichtiger Punkt bei Erstellung des Prototypen war die Verwendung des Java Native Interface. Mit dieser API war es möglich, die Schnittstelle zu den TIS-Applikationen zu implementieren. Eine genaue Beschreibung dieser Schnittstelle ist in Kapitel 6.12 zu finden und die Verwendung innerhalb des Prototypen ist in Kapitel 6.2.2 erläutert. Diese Schnittstelle ermöglicht eine umfangreiche Einbindung einer proprietären Schnittstelle. Es können nicht nur C++-Methoden aus Java-Applikationen heraus aufgerufen werden. Es ist ebenso möglich, Exceptions, welche in C++ auftreten zu den Java-Applikationen weiterzuleiten, Java-Methoden aus C++-Methoden heraus aufzurufen und umfangreiche Objektstrukturen als Parameter zu übergeben. In der Implementierung der Schnittstelle für den Prototypen

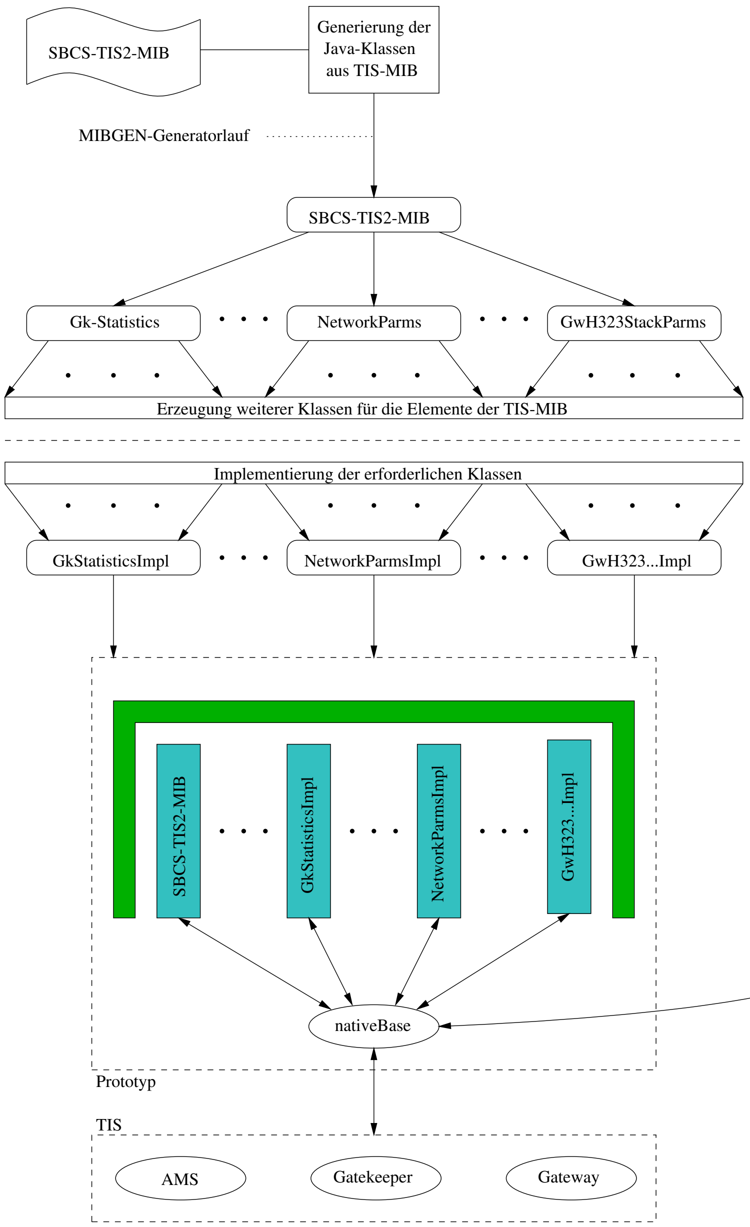

Abbildung 7.1: Phasen der Prototyperstellung

war hierbei von besonderer Wichtigkeit, daß bei Durchlaufen einer SNMP Tabelle eine OID des nächsten Eintrages zurückgegeben werden muß. Dies ist deswegen von so zentraler Bedeutung, da ein Übergabeparameter in seiner Größe feststehen muß. Als Lösung wurde ein zweidimensionales OID-Feld gewählt, welches in der einen Dimension lediglich einen Eintrag enthält während die zweite Dimension (die eigentliche OID) innerhalb der proprietären Methode verändert werden kann. So wurde die Möglichkeit geschaffen, OID´s auch wieder zu den

Java-Applikationen zurückzugeben.

7.2 Ausblick

Die Verwendung des JDMK für das Management bietet große Vorteile, besonders in Hinblick auf die in der Theorie der Flexiblen Management Agenten formulierten Forderungen. Die Mailing-Liste, welche während der Diplomarbeit regelmäßig genutzt wurde, bestätigte, daß ein großes Interesse an dieser Art des Management besteht und die Verwendung von einigen großen Firmen (AT & T) in Betracht gezogen wird.
Aus der Verfolgung der Korrespondenz innerhalb der JDMK- und JMAPI-Mailingliste war festzustellen, daß die Firma Sun Microsystems Inc. JDMK als Basis für weitere Management-Applikationen verwenden wird. Es sollen folgende Produkte mit JDMK verbunden werden:

- JDMAPI 2.0[1]
  Es ist geplant, JDMK als Basis für eine Managementplattform mit JMAPI2.0 als Schnittstelle anzubieten. Diese Kombination soll ab JDMK Version 4.0 zur Verfügung stehen.
- JINI
  Des weiteren ist eine Kombination von JINI und JDMK geplant. Während in dieser Kombination JDMK für die Basisdienste, Protokolle und Interfaces zuständig ist, wird JINI das Management der entsprechenden Komponenten, welche innerhalb einer JINI-Umgebung auftreten, übernehmen.

Eine Anpassung von JDMK an Java2 ist mit der Version 3.1 geplant, wobei die Freigabe dieser Version bis dato nicht bekannt ist.

[1] JMAPI = Java Managent API

Index

Anhang A

Die Syntax der ACL (Access Control List)

```
<acl_file:> :[ < acls> ] [ <trap_block> ]
<acls> : "acl" "=" <acls_list>
<acls_list> : /*empty*/ | <acls_list> <acl_item>
<acl_item> : <communities_stmt> <acl_access> <hosts>
<communities_stmt> : "communities" "=" <communities_set>
<communities_set> : <communities_set> , <community_elem> | <community_elem>
<community_elem>: alphanumeric_string
<acl_access> : "access" "=" <acl_access_type>
<acl_access_type> : read-only | read-write
<hosts> : "managers" "=" <hosts_list>
<hosts_list> : <hosts_list> , <host_item> | <host_item>
<host_item> : <host_alphanumeric_string>
<host_alphanumeric_string> : <hostname> | <ipaddress> | <subnet_mask>
<trap_block> : "trap" "="  <traps_list>
<traps_list> : /*empty*/ | <traps_list> <trap_item> | <trap_item>
<trap_item> :  <trap_community_string> <trap_interest_hosts>
<trap_community_string> : "trap-community" "=" alphanumeric_string
<trap_interest_hosts> : "hosts" "=" <trap_interest_hosts_list>
<trap_interest_hosts_list> : <trap_interest_hosts_list> ","
<trap_interest_host_item> | <trap_interest_host_item>
<trap_interest_host_item> : <hostname> | <ipaddress>
<hostname> : alphanumeric_string
<ipaddress> : ###.###.###.###
<subnet_mask> : ###!###!###!###
```

Anhang B

Realisierung der proprietären Schnittstelle

B.1 Java Teilbereich

```
package jni;
import java.util.* ;
import tools.*;

import java.lang.Integer;
import java.lang.Byte;
import java.util.StringTokenizer;
import com.sun.jaw.snmp.common.*;

/**
 * Klasse zur Verwendung von Java-Native Methoden<BR>
 * Diese Methoden werden an dieser Stelle angegeben, aber noch
 * nicht implementiert.<BR> Dies geschieht in dem entsprechenden
 * C++ File (nativeBase.cpp).
 * @version created 14.12.1998
 * @since 0.99
*/
public class nativeBase \{
   // get
   public static native String getStringNative(int[] mibVarOID);
   public static native int getIntNative(int[] mibVarOID);

   // get Methods to convert Data in accessible format
   /**
   * Get the IpAddress Property of NetworkParms and turn it
   * into a Byte-array. like this :<BR>
   *  [127.0.0.1] -> 91,49,50,55,46,48,46,48,46,49,93<BR>
   * @param mibVarOID OID as int-Array
   * @return Byte-Array
   */
   public static Byte[] getIpAddress(int[] mibVarOID) \{

      String tempString = getStringNative(mibVarOID);
      SnmpString snmpString = new SnmpString(tempString);
      return snmpString.toByte();
   \}

   /**
   * Get the SubnetMask Property of NetworkParms and turn it
   * into a Byte-array. like this :<BR>
   *  [127.0.0.1] -> 91,49,50,55,46,48,46,48,46,49,93<BR>
   * @param mibVarOID OID as int-Array
   * @return Byte-Array
   */
   public static Byte[] getSubnetMask(int[] mibVarOID) \{
      String tempString = getStringNative(mibVarOID);
      SnmpString snmpString = new SnmpString(tempString);
      return snmpString.toByte();
   \}

   /**
   * Get the Adress Format "[0.0.0.0]" and convert it
   * into a Byte-array. like this :<BR>
   *  [127.0.0.1] -> 91,49,50,55,46,48,46,48,46,49,93<BR>
   * @param mibVarOID OID as int-Array
   * @return Byte-Array
   */
   public static Byte[] getByteArray(int[] mibVarOID) \{
      String tempString = getStringNative(mibVarOID);
      SnmpString snmpString = new SnmpString(tempString);
      return snmpString.toByte();
   \}

   // getNext
   public static native int getNextIntNative(int[][] mibVarOID, int oidIndex);
   public static native String getNextStringNative(int[][] mibVarOID, int oidIndex);
   public static native String[] getIndexedStringNative(int[][] oids, int IndexRange);
   public static native int[] getIndexedIntNative(int[][] oids, int IndexRange);

   // set
   public static native void setNative(int[] mibVarOID, int value);
   public static native void setNative(int[] mibVarOID, String value);

   // set Methods to convert Data in accessible format
   /**
   * Set the IpAddress.<BR>
   * The new IP-Address shows : [127.0.0.1]
   * @param mibVarOID OID for MibVariable IpAddress
   * @param ipAddress new IpAddress as Byte-Array
   * @return none
   */
   public static void setIpAddress(int[] mibVarOID, Byte[] ByteArray) \{
```

```
      try \{
         SnmpString snmpString = new SnmpString(ByteArray);
         setNative(mibVarOID, snmpString.toString());
      \} catch (Exception e) \{
         e.printStackTrace();
         System.exit(0);
      \}
   \}

   /**
   * Set the SubnetHask.<BR>
   * The new IP-Address shows : [127.0.0.1]
   * @param mibVarOID OID for HibVariable SubnetHask
   * @param SubnetHask new SubnetHask as Byte-Array
   * @return none
   */
   public static void setSubnetHask(int[] mibVarOID, Byte[] ByteArray) \{
      try \{
         SnmpString snmpString = new SnmpString(ByteArray);
         setNative(mibVarOID, snmpString.toString());
      \} catch (Exception e) \{
         e.printStackTrace();
         System.exit(0);
      \}
   \}

   public static void setByteArray(int[] mibVarOID, Byte[] ByteArray) \{
      SnmpString snmpString = new SnmpString(ByteArray);
      setNative(mibVarOID, snmpString.toString());
   \}

   // andere Native Hethoden
   public static native void buildCAsnNodeStructure();
   public static native void removeNative(int[] mibVarOID);

   public static void encodeStringToAscii(String str, int[][] intArray, int oidIndex) \{
      int strIndex = 0;
      char charArray[] = str.toCharArray();

      intArray[oidIndex] = new int[str.length()];

      for (int i = 0; i < charArray.length; i++) \{
         intArray[oidIndex][i] = (int) charArray[i];
      \}
   \}

   static \{
      System.out.print("JNI : Loading nativeBase.dll ... ");
      System.loadLibrary("c:/jdmk/jni/Debug/nativeBase");
      System.out.println("loaded.");
      System.out.println("JNI : buildCAsnNode ... ");
      buildCAsnNodeStructure();
      System.out.println("executed.");
   \}
\}
```

B.2 C++ Teilbereich

B.2.1 Implementierung

```
#include <jni.h>
#include "jni_nativeBase.h"
#include <stdio.h>
#include <common/tools/tls_files.h>
#include "d:\\tis\\source\\admin\\sxa\\sxa.h"
#include "d:\\tis\\source\\admin\\sxa\\sxa_routines.h"

extern void *thePointerToIt ;
extern void **theNextElement ;
extern VArray theNodes;
extern VArray theDispatchers;
extern HsgBase* theHsgBase;
extern bool theResult ;
extern BArray gpRegisteredTypes;

/*
 * Class:     jni_nativeBase
 * Hethod:    getIntNative
 * Signature: ([I)I
 */
JNIEXPORT jint JNICALL Java_jni_nativeBase_getIntNative
  (JNIEnv *env, jclass, jintArray intArray) \{
  RFC1157VarBind* varBindPtr = (RFC1157VarBind*) calloc(1,sizeof(RFC1157VarBind));
  varBindPtr->name.idLength = env->GetArrayLength(intArray);
  varBindPtr->name.ids = (unsigned int*) env->GetIntArrayElements(intArray,0);
  ResolveVarBind(varBindPtr,ASN_RFC1157_GETREQUEST);
  return varBindPtr->value.asnValue.number;
\}

/*
 * Class:     jni_nativeBase
 * Hethod:    getStringNative
 * Signature: ([I)Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_jni_nativeBase_getStringNative
(JNIEnv *env, jclass, jintArray intArray) \{
  RFC1157VarBind* varBindPtr = (RFC1157VarBind*) calloc(1,sizeof(RFC1157VarBind));
  char returnString[128];
  varBindPtr->name.idLength = env->GetArrayLength(intArray);
  varBindPtr->name.ids = (unsigned int*) env->GetIntArrayElements(intArray,0);
  ResolveVarBind(varBindPtr,ASN_RFC1157_GETREQUEST);
  sprintf(returnString, "%s", varBindPtr->value.asnValue.string.stream);
  return env->NewStringUTF(returnString);
\}

/*
 * Class:     jni_nativeBase
 * Hethod:    getNextIntNative
 * Signature: ([[I)I
 */
JNIEXPORT jint JNICALL Java_jni_nativeBase_getNextIntNative
(JNIEnv *env, jclass, jobjectArray objArray, jint oidIndex) \{
  RFC1157VarBind* varBindPtr = (RFC1157VarBind*) calloc(1,sizeof(RFC1157VarBind));
  // Read the incoming OID
  jintArray xxx = (jintArray) env->GetObjectArrayElement(objArray, oidIndex);
  varBindPtr->name.idLength = env->GetArrayLength(xxx);
  varBindPtr->name.ids = (unsigned int*) env->GetIntArrayElements(xxx,0);
  // Send the GETNEXT Request
  ResolveVarBind(varBindPtr,ASN_RFC1157_GETNEXTREQUEST);
  // Update OID for next GETNEXT-Request
  jobjectArray yyy = (jobjectArray)env->NewIntArray(varBindPtr->name.idLength);
  env->SetIntArrayRegion((jintArray)yyy,0,varBindPtr->name.idLength,(long*)varBindPtr->name.ids);
  env->SetObjectArrayElement(objArray,oidIndex,yyy);
  // attention: watch out for memory leaks!
  //Return Integer Value of assigned HIB-Variable
  return (int) varBindPtr->value.asnValue.number;
\}

/*
```

```
 * Class:     jni_nativeBase
 * Method:    getNextStringNative
 * Signature: ([[I)Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_jni_nativeBase_getNextStringNative
(JNIEnv *env, jclass, jobjectArray objArray, jint oidIndex) \{
  RFC1157VarBind* varBindPtr = (RFC1157VarBind*) calloc(1,sizeof(RFC1157VarBind));
  char returnString[128];
  // Read the incoming OID
  jintArray xxx = (jintArray) env->GetObjectArrayElement(objArray, oidIndex);
  varBindPtr->name.idLength = env->GetArrayLength(xxx);
  varBindPtr->name.ids = (unsigned int*) env->GetIntArrayElements(xxx,0);
  // Send the GETNEXT Request
  ResolveVarBind(varBindPtr,ASN_RFC1157_GETNEXTREQUEST);
  // Update OID for next GETNEXT-Request
  jobjectArray yyy = (jobjectArray)env->NewIntArray(varBindPtr->name.idLength);
  env->SetIntArrayRegion((jintArray)yyy,0,varBindPtr->name.idLength,(long*)varBindPtr->name.ids);
  env->SetObjectArrayElement(objArray,oidIndex,yyy);
  // attention: watch out for memory leaks!
  //Return Integer Value of assigned MIB-Variable
  sprintf(returnString, "%s", varBindPtr->value.asnValue.string.stream);
  return env->NewStringUTF(returnString);
\}

/*
 * Class:     jni_nativeBase
 * Method:    setNative
 * Signature: ([II)V
 */
JNIEXPORT void JNICALL Java_jni_nativeBase_setNative___3II
(JNIEnv *env, jclass, jintArray intArray, jint newInt) \{
  RFC1157VarBind* varBindPtr = (RFC1157VarBind*) calloc(1,sizeof(RFC1157VarBind));
  varBindPtr->name.idLength = env->GetArrayLength(intArray);
  varBindPtr->name.ids = (unsigned int*) env->GetIntArrayElements(intArray,0);
  varBindPtr->value.asnType = ASN_INTEGER;
  varBindPtr->value.asnValue.number = newInt;
  ResolveVarBind(varBindPtr,ASN_RFC1157_SETREQUEST);
\}

/*
 * Class:     jni_nativeBase
 * Method:    setNative
 * Signature: ([ILjava/lang/String;)V
 */
JNIEXPORT void JNICALL Java_jni_nativeBase_setNative___3ILjava_lang_String_2
(JNIEnv *env, jclass, jintArray intArray, jstring newString) \{
  RFC1157VarBind* varBindPtr = (RFC1157VarBind*) calloc(1,sizeof(RFC1157VarBind));
  const char* pStr = env->GetStringUTFChars(newString,0);
  varBindPtr->name.idLength = env->GetArrayLength(intArray);
  varBindPtr->name.ids = (unsigned int*) env->GetIntArrayElements(intArray,0);
  varBindPtr->value.asnType = ASN_OCTETSTRING;
  varBindPtr->value.asnValue.string.stream = (unsigned char*) pStr;
  varBindPtr->value.asnValue.string.length = env->GetStringLength(newString);
  ResolveVarBind(varBindPtr,ASN_RFC1157_SETREQUEST);
  env->ReleaseStringUTFChars(newString, pStr);
\}

/*
 * Class:     jni_nativeBase
 * Method:    removeNative
 * Signature: ([I)V
 */
JNIEXPORT void JNICALL Java_jni_nativeBase_removeNative
(JNIEnv *env, jclass, jintArray intArray) \{
  RFC1157VarBind* varBindPtr = (RFC1157VarBind*) calloc(1,sizeof(RFC1157VarBind));
  varBindPtr->name.idLength = env->GetArrayLength(intArray);
  varBindPtr->name.ids = (unsigned int*) env->GetIntArrayElements(intArray,0);
  varBindPtr->value.asnType = ASN_INTEGER;
  varBindPtr->value.asnValue.number = SNMP_NULL;
  ResolveVarBind(varBindPtr,ASN_RFC1157_SETREQUEST);
\}

/*
 * Class:     jni_nativeBase
 * Method:    buildCAsnNodeStructure
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_jni_nativeBase_buildCAsnNodeStructure
(JNIEnv *, jclass) \{
  char path[MAX_FILENAME_LENGTH];
  FILE* mibFile ;
  char nextLine[256] ;
  VString* words ;
  UINT min_oid[] = \{ 1, 1\};
  AsnObjectIdentifier a_small_oid = \{ OID_SIZEOF(min_oid), min_oid \};
  UINT max_oid[] = \{ 1, 9\}; //
  AsnObjectIdentifier a_big_oid = \{ OID_SIZEOF(max_oid), max_oid \};
  InitializeCriticalSection(&gAlarmMailerCritSect);  // alarm mailer
  SetTypeRegistry(&gpRegisteredTypes);
  if (!CfgGetFileRoot("\\config\\tis_mib.mdf", path, sizeof(path)))
    exit(1)
    mibFile = fopen(path, "rt") ;
    if (mibFile != NULL) \{
      // Start node list with lowest possible OID
      thePointerToIt = new CAsnNode(a_small_oid) ;
      if (0 == thePointerToIt) \{
        theResult = false ;
      \} else \{
        theNextElement = (void**)theNodes.Append() ;
        *theNextElement = thePointerToIt ;
      \}
      while (0 != fgets(nextLine,256,mibFile)) \{
        words = LineToWords(nextLine) ;
        if (*(words[0]) != 0) \{\}
        if (words[0] == "NODE") \{
          thePointerToIt = new CAsnNode(words) ;
          if (0 == thePointerToIt) \{
            theResult = false ;
          \} else \{
            static_cast<CAsnNode*>(thePointerToIt)->Log() ;
            theNextElement = (void**)theNodes.Append() ;
            *theNextElement = thePointerToIt ;
          \}
        \} else if (words[0] == "DISPATCHER") \{
          thePointerToIt = new CDispatcherProperties(words) ;
          if (0 == thePointerToIt) \{
            theResult = false ;
          \} else \{
            static_cast<CDispatcherProperties*>(thePointerToIt)->Log() ;
            theNextElement = (void**)theDispatchers.Append() ;
            *theNextElement = thePointerToIt ;
          \}
        \}
        fclose(mibFile) ;
      \} else \{
        theResult = false ;
      \}
    InitDispatch();
    InitDispIPC();
\}
```

B.2.2 Header

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class jni_nativeBase */

#ifndef _Included_jni_nativeBase
#define _Included_jni_nativeBase
#ifdef __cplusplus
extern "C" \{
#endif
/*
 * Class:     jni_nativeBase
 * Method:    getStringNative
 * Signature: ([I)Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_jni_nativeBase_getStringNative
  (JNIEnv *, jclass, jintArray);

/*
 * Class:     jni_nativeBase
 * Method:    getIntNative
 * Signature: ([I)I
 */
JNIEXPORT jint JNICALL Java_jni_nativeBase_getIntNative
  (JNIEnv *, jclass, jintArray);

/*
 * Class:     jni_nativeBase
 * Method:    getNextIntNative
 * Signature: ([[II)I
 */
JNIEXPORT jint JNICALL Java_jni_nativeBase_getNextIntNative
  (JNIEnv *, jclass, jobjectArray, jint);

/*
 * Class:     jni_nativeBase
 * Method:    getNextStringNative
 * Signature: ([[II)Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_jni_nativeBase_getNextStringNative
  (JNIEnv *, jclass, jobjectArray, jint);

/*
 * Class:     jni_nativeBase
 * Method:    setNative
 * Signature: ([II)V
 */
JNIEXPORT void JNICALL Java_jni_nativeBase_setNative___3II
  (JNIEnv *, jclass, jintArray, jint);

/*
 * Class:     jni_nativeBase
 * Method:    setNative
 * Signature: ([ILjava/lang/String;)V
 */
JNIEXPORT void JNICALL Java_jni_nativeBase_setNative___3ILjava_lang_String_2
  (JNIEnv *, jclass, jintArray, jstring);

/*
 * Class:     jni_nativeBase
 * Method:    buildCAsnNodeStructure
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_jni_nativeBase_buildCAsnNodeStructure
  (JNIEnv *, jclass);

/*
 * Class:     jni_nativeBase
 * Method:    removeNative
 * Signature: ([I)V
 */
JNIEXPORT void JNICALL Java_jni_nativeBase_removeNative
  (JNIEnv *, jclass, jintArray);

#ifdef __cplusplus
\}
#endif
#endif
```

Anhang C

TIS-MIB : Gruppe NetworkParms

```
        SBCS-TIS2-MIB DEFINITIONS ::= BEGIN

        --
        -- Derived from the infamous toaster mib.
        -- I am not sure what the next line does
        -- $Logfile:   C:/snmp/testmibs/tis.miv  $ $Revision:   1.0  $
        --
...
--
-- textual conventions used in this module
--

DateAndTime ::= OCTET STRING (SIZE(11))

--           "A date-time specification.
--
--            field  octets  contents                  range
--            -----  ------  --------                  -----
--              1      1-2   year                      0..65536
--              2       3    month                     1..12
--              3       4    day                       1..31
--              4       5    hour                      0..23
--              5       6    minutes                   0..59
--              6       7    seconds                   0..60
--                           (use 60 for leap-second)
--              7       8    deci-seconds              0..9
--              8       9    direction from UTC        '+' / '-'
--              9      10    hours from UTC            0..11
--             10      11    minutes from UTC          0..59
--
--           For example, Tuesday May 26, 1992 at 1:30:15 PM EDT would be
--           displayed as:
--
--                            1992-5-26,13:30:15.0,-4:0
--
--           Note that if only local time is known, then timezone
--           information (fields 8-10) is not present."
--   SYNTAX       OCTET STRING (SIZE (8 | 11))
```

```
 -- We probably need a Siemens BCS root mib - maybe someone in SC
 -- has created this.

         siemensBCS     OBJECT IDENTIFIER ::= { enterprises 2206 }
         boca           OBJECT IDENTIFIER ::= { siemensBCS 5 }
  tis2Mib        OBJECT IDENTIFIER ::= { boca 1 }

         --  tis2 MIB
         --
         --  Section 1 - Platform Parameters

         tisPlatform       OBJECT IDENTIFIER ::= { tis2Mib 1 }

  --  This is the TIS-2.0 WinNT/Caribou Platform data.
  --  This should probably be in it's own MIB segment.

-- Section 1.1 - Network Parameters

networkParms       OBJECT IDENTIFIER ::= { tisPlatform 1 }

-- These are various network parameters for the WinNT/Caribou
-- platform

--
-- ** 1.1.1 Computer name
--

computerName OBJECT-TYPE
SYNTAX  DisplayString
ACCESS  read-write
STATUS  mandatory
DESCRIPTION
"This is the Computer name as found in
 WinNT Control Panel-> Network->Identification."
::= { networkParms 1 }

--
-- ** 1.1.2 Network Domain
--
nwDomain OBJECT-TYPE
SYNTAX  DisplayString
ACCESS  read-write
STATUS  mandatory
DESCRIPTION
"This is the Domain as found in
 WinNT Control Panel-> Network->Identification."
::= { networkParms 2 }

--
-- ** 1.1.3 IP Address
--
ipAddress OBJECT-TYPE
SYNTAX  OCTET STRING
ACCESS  read-write
STATUS  mandatory
DESCRIPTION
"This is the Computer Name as found in
 WinNT Control Panel-> Network->Protocols."
::= { networkParms 3 }

--
-- ** 1.1.4 Subnet Mask
--

subnetMask OBJECT-TYPE
SYNTAX  OCTET STRING
ACCESS  read-write
STATUS  mandatory
DESCRIPTION
"This is the Subnet Mask as found in
```

```
 WinNT Control Panel-> Network->Protocols."
::= { networkParms 4 }

--
-- ** 1.1.5 Host name
--
hostName OBJECT-TYPE
SYNTAX  DisplayString
ACCESS  read-write
STATUS  mandatory
DESCRIPTION
"This is the Host Name as found in
 WinNT Control Panel-> Network->Protocols."
::= { networkParms 5 }

--
-- ** 1.1.6 TCP/IP Domain
--
tcpipDomain OBJECT-TYPE
SYNTAX  DisplayString
ACCESS  read-write
STATUS  mandatory
DESCRIPTION
"This is the Domain Name as found in
 WinNT Control Panel-> Network->Protocols."
::= { networkParms 6 }


--
-- ** 1.1.7 Table Definitions for the DNS Server table
--

dnsServerTable OBJECT-TYPE
SYNTAX  SEQUENCE OF DnsServerEntry
ACCESS  not-accessible
STATUS  mandatory
DESCRIPTION
"This is the list of DNS servers as found in
 WinNT Control Panel-> Network->Protocols."
::= { networkParms 7 }

dnsServerEntry OBJECT-TYPE
SYNTAX DnsServerEntry
ACCESS not-accessible
STATUS mandatory
DESCRIPTION
"This is one entry for a DNS server in the
table of DNS servers as found in WinNT
Control Panel-> Network->Protocols.
Unused entries should contain the null IP address
(0.0.0.0?)."
INDEX {dnsServerIndex}
::= {dnsServerTable 1}

DnsServerEntry ::= SEQUENCE
{
dnsServerIndex INTEGER,
dnsServerAddr OCTET STRING
}

           dnsServerIndex OBJECT-TYPE
               SYNTAX  INTEGER (1..15)
          ACCESS  read-write
         STATUS  mandatory
            DESCRIPTION
"A unique value for each DNS server.  Its value
ranges between 1 and 15. This is used to index the
rows of this table and also indicates the search
order."
               ::= { dnsServerEntry 1 }

 dnsServerAddr OBJECT-TYPE
               SYNTAX  OCTET STRING
          ACCESS  read-write
         STATUS  mandatory
DESCRIPTION
"A unique value for each DNS server.  This is an
IP address for a DNS server.
Please see dnsServerIndex for the rules to set
this object."
 ::= { dnsServerEntry 2 }

--
-- ** 1.1.8 Table Definitions for the Domain Suffix table
--

domainSuffixTable OBJECT-TYPE
SYNTAX  SEQUENCE OF DomainSuffixEntry
ACCESS  not-accessible
STATUS  mandatory
DESCRIPTION
"This is the list of domain suffixes as found in
 WinNT Control Panel-> Network->Protocols."
::= { networkParms 8 }

domainSuffixEntry OBJECT-TYPE
SYNTAX DomainSuffixEntry
ACCESS not-accessible
STATUS mandatory
DESCRIPTION
"This is one entry for a domain suffix in the
 table of domain suffixes as found in WinNT
 Control Panel-> Network->Protocols.
 Unused entries should be indicated by setting
 domainSuffix to a null string."
INDEX {domainSuffixIndex}
::= {domainSuffixTable 1}

DomainSuffixEntry ::= SEQUENCE
{
domainSuffixIndex INTEGER,
domainSuffix DisplayString
}

           domainSuffixIndex OBJECT-TYPE
               SYNTAX  INTEGER  (1..15)
          ACCESS  read-write
         STATUS  mandatory
            DESCRIPTION
"A unique value for each domain suffix.  Its value
ranges between 1 and 15. This is used to index the
rows of this table and also indicates the search
order.   To add a suffix, select an unused donmainSuffixIndex
and fill in the domainSuffix."
        ::= { domainSuffixEntry 1 }

 domainSuffix OBJECT-TYPE
               SYNTAX  DisplayString
          ACCESS  read-write
         STATUS  mandatory
            DESCRIPTION
"A unique value for each domain suffix.
Please see domainSuffixIndex for the rules to set
this object."
 ::= { domainSuffixEntry 2 }

--
-- ** 1.1.9 Enable DNS Flag
--
enableDnsForWindows OBJECT-TYPE
```

```
SYNTAX  INTEGER { checked(1), notChecked(2) }
ACCESS  read-write
STATUS  mandatory
DESCRIPTION
"This is the 'Enable DNS' flag as found in
WinNT Control Panel-> Network->Protocols."
::= { networkParms 9 }

--
-- ** 1.1.10 Enable LMHOSTS Lookup
--
enableLmhostsLookup OBJECT-TYPE
SYNTAX  INTEGER { checked(1), notChecked(2) }
ACCESS  read-write
STATUS  mandatory
DESCRIPTION
"This is the 'Enable LMHOSTS Lookup' flag as found in
WinNT Control Panel-> Network->Protocols."
::= { networkParms 10 }

--
-- ** 1.1.11 Scope ID
--
scopeId OBJECT-TYPE
SYNTAX  DisplayString
ACCESS  read-write
STATUS  mandatory
DESCRIPTION
"This is the Scope ID string, whatever that is, as found
in WinNT Control Panel-> Network->Protocols."
::= { networkParms 11 }

--
-- ** 1.1.12 Enable IP Forwarding
--
enableIpForwarding OBJECT-TYPE
SYNTAX  INTEGER { checked(1), notChecked(2) }
ACCESS  read-write
STATUS  mandatory
DESCRIPTION
"This is the 'Enable IP Forwarding' flag as found in
WinNT Control Panel-> Network->Protocols."
::= { networkParms 12 }

--
-- ** 1.1.13 Table Definitions for the WinNT IP Gateway Table
--

ipGatewayTable OBJECT-TYPE
SYNTAX  SEQUENCE OF IpGatewayEntry
ACCESS  not-accessible
STATUS  mandatory
DESCRIPTION
"This is the list of IP Gateways as found in
 WinNT Control Panel-> Network->Protocols->TCPIP->Advanced."
::= { networkParms 13 }
```

```
ipGatewayEntry OBJECT-TYPE
SYNTAX IpGatewayEntry
ACCESS not-accessible
STATUS mandatory
DESCRIPTION
"This is one entry for each gateway in the
 table of IP gateways as found in WinNT
 Control Panel-> Network->Protocols-TCPIP->Advanced."
INDEX {ipGatewayIndex}
::= {ipGatewayTable 1}

IpGatewayEntry ::= SEQUENCE
{
ipGatewayIndex INTEGER,
ipGateway OCTET STRING
}

           ipGatewayIndex OBJECT-TYPE
               SYNTAX  INTEGER  (1..15)
          ACCESS  read-write
         STATUS  mandatory
            DESCRIPTION
"A unique value for each gateway.  Its value
ranges between 1 and 15. This is used to index the
rows of this table and also indicates the search
order."
        ::= { ipGatewayEntry 1 }

 ipGateway OBJECT-TYPE
               SYNTAX  OCTET STRING
          ACCESS  read-write
         STATUS  mandatory
            DESCRIPTION
"An IP Addres of an IP gateway."
 ::= { ipGatewayEntry 2 }


-- Section 1.2 - SNMP Parameters

snmpParms       OBJECT IDENTIFIER ::= { tisPlatform 2 }

-- These are various parameters dealing with SNMP on the
-- WinNT/Caribou platform

--
-- ** 1.2.1 SNMP Agent Contact
--
snmpAgentContact OBJECT-TYPE
SYNTAX  DisplayString
ACCESS  read-write
STATUS  mandatory
DESCRIPTION
"This is the name of someone to call about the box.
...
```

Anhang D

Die CORBA - Schnittstelle

D.1 AdaptorServer.idl

```
// Copyright (c) 09/29/98, by Sun Microsystems, Inc.
// All rights reserved.

// "@(#)AdaptorServer.idl 3.1 98/09/29 SMI"


// __________________________________________________________________
//
// JAVA PACKAGE
//
// __________________________________________________________________
#pragma javaPackage "com.sun.jaw.impl.adaptor.iiop.internal"

// __________________________________________________________________
//
// MODULE DEFINITION
//
// __________________________________________________________________
module CorbaServer     {

  // ---------------------------------------------------------------
  // typedef declaration
  // ---------------------------------------------------------------
        //
        // String list
        //
        typedef sequence <string>        StringListType;

        //
        // object serialization
        //
        typedef sequence <octet>        SerializedObjectType ;

        //
        // serialized object list
        //
        typedef sequence <SerializedObjectType>
                SerializedObjectListType ;

  // ---------------------------------------------------------------
  // Exception declaration
  // ---------------------------------------------------------------
        exception JdmkServerRuntimeException
            { SerializedObjectType JavaObjectException ; } ;
        exception JdmkIllegalAccessException
            { string JavaStringException ; } ;
        exception JdmkInstantiationException
            { string JavaStringException ; } ;
        exception JdmkClassNotFoundException
            { string JavaStringException ; } ;
        exception JdmkServiceNotFoundException
            { string JavaStringException ; } ;
        exception JdmkInstanceAlreadyExistException
            { string JavaStringException ; } ;
        exception JdmkInstanceNotFoundException
            { string JavaStringException ; } ;
        exception JdmkInvalidSerializedListException
            { string JavaStringException ; } ;
        exception JdmkPropertyNotFoundException
            { string JavaStringException ; } ;
        exception JdmkInvocationTargetException
            { SerializedObjectType JavaObjectException ;
              string JavaStringException ; } ;
        exception JdmkInvalidPropertyValueException
            { string JavaStringException ; } ;
        exception JdmkNoSuchMethodException
            { string JavaStringException ; } ;
        exception JdmkCommunicationException
            { string JavaStringException ; } ;

// ==================================================================
//
// INTERFACE DEFINITION
//
// ==================================================================
interface EvtRcvClient {
        void handleEvent(
           in SerializedObjectType  event,
           in string source,
           in string method,
```

```
           in string listener)
        raises ( JdmkInvalidSerializedListException,
                 JdmkCommunicationException );
} ;


interface AdaptorIIOP {


  // ---------------------------------------------------------------
  // DESCRIPTION
  //     Gets handles on managed objects controlled by the remote
  //     managed object
  //     server. The method enables any of the following be to obtained:
  //         - All the managed objects
  //         - A subset specified through a query
  //         - A specific instance
  //     When the class name and the instance name are empty, it means
  //     that all the
  //     objects are to be selected (and filered if a query is specified).
  //
  // PARAMETERS
  //     - name: The instance name of the object to be retrieved.
  //     - query: The query to be applied for selecting managed  objects.
  //              It's a serialized java object.
  //              Its class is com.sun.jaw.reference.query.QueryExp
  //
  // RETURN
  //     A list containing the selected managed objects.
  //
  // EXCEPTIONS
  //     - JdmkInstanceNotFoundException
  //     - JdmkServiceNotFoundException
  //     - JdmkInvalidSerializedListException
  //
  // For more information, see 'getObject' in 'Interface
  // com.sun.jaw.impl.adaptor.rmi.AdaptorServer'
  //
  // ---------------------------------------------------------------
        StringListType getObject(
                        in string name,
                        in SerializedObjectType query)

        raises (        JdmkInstanceNotFoundException,
                        JdmkServiceNotFoundException,
                        JdmkServerRuntimeException,
                        JdmkInvalidSerializedListException);

  // ---------------------------------------------------------------
  // DESCRIPTION
  //     Allows the value of a specific property within a managed object
  //     to be obtained.
  //
  // PARAMETERS
  //     - name: The names of the managed objects from within which
  //     the property
  //          is to be retrieved.
  //     - property: The name of the property to be retrieved.
  //
  // RETURN
  //     The value of the retrieved property.
  //     It's a serialized java object.
  //     Its class is java.lang.Object
  //
  // EXCEPTIONS
  //     - InstanceNotFoundException,
  //     - JdmkPropertyNotFoundException,
  //     - JdmkServiceNotFoundException,
  //     - JdmkInvocationTargetException,
  //     - JdmkInvalidSerializedListException
  //
  // For more information, see 'getValue' in
  //'Interface com.sun.jaw.impl.adaptor.rmi.AdaptorServer'
  //
  // ---------------------------------------------------------------
        SerializedObjectType getValue(
                        in string name,
                        in string property)

        raises (        JdmkInstanceNotFoundException,
                        JdmkPropertyNotFoundException,
                        JdmkServiceNotFoundException,
                        JdmkInvocationTargetException,
                        JdmkServerRuntimeException,
                        JdmkInvalidSerializedListException );

  // ---------------------------------------------------------------
  // DESCRIPTION
  //     Allows the value of a specific indexed property within a managed
  //     object to be obtained.
  //
  // PARAMETERS
  //     - name: The names of the objects from within which the property is
  //             to be retrieved.
  //     - property: The name of the property to be retrieved.
  //     - pos: The position in the index of the value to be retrieved.
  //
  // RETURN
  //     The value of the retrieved property.
  //     It's a serialized java object.
  //     Its class is java.lang.Object
  //
  // EXCEPTIONS
  //     - JdmkInstanceNotFoundException
  //     - JdmkPropertyNotFoundException
  //     - JdmkServiceNotFoundException
  //     - JdmkInvocationTargetException
  //     - JdmkInvalidSerializedListException
  //
  // For more information, see 'getIndexedValue' in
  // 'Interface com.sun.jaw.impl.adaptor.rmi.AdaptorServer'
  //
  // ---------------------------------------------------------------
        SerializedObjectType getIndexedValue(
                        in string name,
                        in string property,
                        in long pos)

        raises (        JdmkInstanceNotFoundException,
                        JdmkPropertyNotFoundException,
                        JdmkServiceNotFoundException,
                        JdmkInvocationTargetException,
                        JdmkServerRuntimeException,
                        JdmkInvalidSerializedListException);

  // ---------------------------------------------------------------
  // DESCRIPTION
  //     Allows the values of several properties within a managed object
  //     to be obtained.
  //
  // PARAMETERS
  //     - name: The names of the objects from within which the properties
  //              are to be retrieved.
  //     - propertyIdList: A list of the properties to be retrieved.
  //
  // RETURN
  //      The values of the retrieved properties.
  //     It's a serialized java object.
  //      Its class is com.sun.jaw.reference.common.PropertyList
  //
```

```
// EXCEPTIONS
//      - JdmkInstanceNotFoundException
//      - JdmkServiceNotFoundException
//      - JdmkInvalidSerializedListException
//
// For more information, see 'getValues' in
// 'Interface com.sun.jaw.impl.adaptor.rmi.AdaptorServer'
//
// ----------------------------------------------------------------
      SerializedObjectType getValues(
                      in string name,
                      in SerializedObjectListType propertyIdList)

      raises (        JdmkInstanceNotFoundException,
                      JdmkServiceNotFoundException,
                      JdmkServerRuntimeException,
                      JdmkInvalidSerializedListException );

// ----------------------------------------------------------------
// DESCRIPTION
//     Sets the value of a specific property of a managed object.
//     The value must support the Serializable interface.
//
// PARAMETERS
//     - name: The name of the m-bean within which the
// property is to be set.
//     - id: The name of the property to be set.
//     - value: The value that the property is to be set to.
// It's a serialized java object.
//           It's class is java.lang.Object
//     - op: The Java class name of the operator to be applied
// to the property.
//          The class must implement the OperatorSrvIf interface.
//
// RETURN
//     The value that the property has been set to.
// It's a serialized java object.
//     It's class is java.lang.Object
//
// EXCEPTIONS
//    - JdmkInstanceNotFoundException
//    - JdmkInvocationTargetException
//    - JdmkIllegalAccessException
//    - JdmkServiceNotFoundException
//    - JdmkPropertyNotFoundException
//    - JdmkInvalidPropertyValueException
//    - JdmkClassNotFoundException
//    - JdmkInstantiationException
//    - JdmkInvalidSerializedListException
//
// For more information, see 'SetValue' in
// 'Interface com.sun.jaw.impl.adaptor.rmi.AdaptorServer'
//
// ----------------------------------------------------------------
      SerializedObjectType SetValue(
                      in string name,
                      in string id,
                      in SerializedObjectType value,
                      in string op)

      raises (        JdmkInstanceNotFoundException,
                      JdmkInvocationTargetException,
                      JdmkIllegalAccessException,
                      JdmkServiceNotFoundException,
                      JdmkPropertyNotFoundException,
                      JdmkInvalidPropertyValueException,
                      JdmkClassNotFoundException,
                      JdmkInstantiationException ,
                      JdmkServerRuntimeException,
                      JdmkInvalidSerializedListException);

// ----------------------------------------------------------------
// DESCRIPTION
//     Sets the value of a specific indexed property of a managed object.
//     The value must support the Serializable interface.
//
// PARAMETERS
//     - name: The name of the m-bean within which the
// property is to be set.
//     - id: The name of the property to be set.
//     - value: The value that the property is to be set to.
// It's a serialized java object.
//             It's class is java.lang.Object
//     - op: The Java class name of the operator to be applied
// to the property.
//          The class must implement the OperatorSrvIf interface.
//     - pos: The position in the index of the value to be set.
//
// RETURN
//     The value that the property has been set to.
// It's a serialized java object.
//     It's class is java.lang.Object
//
// EXCEPTIONS
//     - InstanceNotFoundException
//     - JdmkInvocationTargetException
//     - JdmkIllegalAccessException
//     - JdmkServiceNotFoundException
//     - JdmkPropertyNotFoundException
//     - JdmkInvalidPropertyValueException
//     - JdmkClassNotFoundException
//     - JdmkInstantiationException
//     - JdmkInvalidSerializedListException
//
// For more information, see 'setIndexedValue' in
// 'Interface com.sun.jaw.impl.adaptor.rmi.AdaptorServer'
//
// ----------------------------------------------------------------
      SerializedObjectType SetIndexedValue(
                      in string name,
                      in string id,
                      in SerializedObjectType value,
                      in string op,
                      in long pos)

      raises (        JdmkInstanceNotFoundException,
                      JdmkInvocationTargetException,
                      JdmkIllegalAccessException,
                      JdmkServiceNotFoundException,
                      JdmkPropertyNotFoundException,
                      JdmkInvalidPropertyValueException,
                      JdmkClassNotFoundException,
                      JdmkInstantiationException,
                      JdmkServerRuntimeException,
                      JdmkInvalidSerializedListException) ;

// ----------------------------------------------------------------
// DESCRIPTION
//     Sets the value of several properties within a managed object.
//     The value must support the Serializable interface.
//
// PARAMETERS
//     - name: The name of the object within which the properties
//             are to be set.
//     - modif: A list of the properties to be set and the values to which
//             they are to be set.
//      It's a serialized java object.
//      It's class is com.sun.jaw.reference.common.ModificationList
//
```

```
// RETURN
//     The values of the properties that were set.
// It's a serialized java object.
//     It's class is com.sun.jaw.reference.common.PropertyList
//
// EXCEPTIONS
//     - JdmkInstanceNotFoundException
//     - JdmkServiceNotFoundException
//     - JdmkInvalidSerializedListException
//
// For more information, see 'setValues' in
// 'Interface com.sun.jaw.impl.adaptor.rmi.AdaptorServer'
//
// ----------------------------------------------------------------
      SerializedObjectType setValues(
                    in string name,
                    in SerializedObjectType modif)

      raises (       JdmkInstanceNotFoundException,
                    JdmkServiceNotFoundException,
                    JdmkServerRuntimeException,
                    JdmkInvalidSerializedListException);


// ----------------------------------------------------------------
// DESCRIPTION
//     Returns the name of the domain controlled
// by the managed object server.
//
// PARAMETERS
//     No parameter
//
// RETURN
//     Returns the name of the domain controlled
// by the managed object server
//
// EXCEPTIONS
//     No exception
//
// For more information, see 'getDomain' in
// 'Interface com.sun.jaw.impl.adaptor.rmi.AdaptorServer'
//
// ----------------------------------------------------------------
      string getDomain()

      raises (       JdmkServerRuntimeException);

// ----------------------------------------------------------------
// DESCRIPTION
//     Allows a Java object of a particular class to be instantiated in a
//     remote managed object server.
//
// PARAMETERS
//     - className: The Java class name of the object to be created.
//
// RETURN
//     No  return parameter
//
// EXCEPTIONS
//    - JdmkIllegalAccessException
//    - JdmkInstantiationException
//    - JdmkClassNotFoundException
//    - JdmkServiceNotFoundException
//    - JdmkInvocationTargetException
//
// For more information, see 'newObj' in
// 'Interface com.sun.jaw.impl.adaptor.rmi.AdaptorServer'
//
// ----------------------------------------------------------------
      void newObj (
                    in string className)

      raises (       JdmkIllegalAccessException,
                    JdmkInstantiationException,
                    JdmkClassNotFoundException,
                    JdmkInvocationTargetException,
                    JdmkServerRuntimeException,
                    JdmkServiceNotFoundException) ;

// ----------------------------------------------------------------
// DESCRIPTION
//      Creates a persistent instance of a managed
//      object in the remote object
//      server. When calling the method, you can optionally
//      provide the class name of
//      the Java implementation to be used for instantiating
//      the new object.
//
// PARAMETERS
//     - impl: The name of the Java implementation to be
// used on the server.
//     - name: The name of the managed object to be created.
//     - plist: The list of initial values of the properties of the
// new managed object.
//           It's a serialized java object.
//              It's class com.sun.jaw.reference.common.ModificationList
//
// RETURN
//     The newly created managed object.
// It's an Object name and a serialized java object.
//     It's class is com.sun.jaw.reference.common.PropertyList
//
// EXCEPTIONS
//     - JdmkIllegalAccessException
//     - JdmkClassNotFoundException
//     - JdmkServiceNotFoundException
//     - JdmkInstanceAlreadyExistException
//     - JdmkInstantiationException
//     - JdmkInvocationTargetException
//     - JdmkInvalidSerializedListException
//
// For more information, see 'newDBMO' in
// 'Interface com.sun.jaw.impl.adaptor.rmi.AdaptorServer'
//
// ----------------------------------------------------------------
      string newDBMO(
                         in string impl,
                         in string name,
                         in SerializedObjectType plist )

      raises (       JdmkIllegalAccessException,
                    JdmkClassNotFoundException,
                    JdmkServiceNotFoundException,
                    JdmkInstanceAlreadyExistException,
                    JdmkInstantiationException,
                    JdmkInvocationTargetException,
                    JdmkServerRuntimeException,
                    JdmkInvalidSerializedListException ) ;

// ----------------------------------------------------------------
// DESCRIPTION
//     Creates an instance of a managed object in the remote
//     object server.
//     When calling the method, you can optionally provide
//     the class name of
//     the Java implementation to be used for instantiating
//     the new object.
//
// PARAMETERS
//     - impl: The name of the Java implementation to be
```

```
//     used on the server.
//     - name: The name of the managed object to be created.
//     - plist: The list of initial values of the properties of the
//     new managed object.
//     It's a serialized java object.
//     It's class is com.sun.jaw.reference.common.ModificationList
//
// RETURN
//     The newly created managed object. It's an Object
//     name and a serialized java object.
//     It's class is com.sun.jaw.reference.common.PropertyList
//
// EXCEPTIONS
//     - JdmkIllegalAccessException
//     - JdmkClassNotFoundException
//     - JdmkServiceNotFoundException
//     - JdmkInstanceAlreadyExistException
//     - JdmkInstantiationException
//     - JdmkInvocationTargetException
//     - JdmkInvalidSerializedListException
//
// For more information, see 'newMO' in
// 'Interface com.sun.jaw.impl.adaptor.rmi.AdaptorServer'
//
// ----------------------------------------------------------------
      string newMO(
                      in string impl,
                      in string name,
                      in SerializedObjectType plist )

      raises (        JdmkIllegalAccessException,
                      JdmkClassNotFoundException,
                      JdmkServiceNotFoundException,
                      JdmkInstanceAlreadyExistException,
                      JdmkInstantiationException,
                      JdmkInvocationTargetException,
                      JdmkServerRuntimeException,
                      JdmkInvalidSerializedListException) ;



// ----------------------------------------------------------------
// DESCRIPTION
//     Allows a Java object of a particular class to be instantiated in a
//     remote managed object server.
//
// PARAMETERS
//     - className:  The Java class name of the object to be created.
//     - aLoader: The name of a class loader to be used.
//
// RETURN
//
// EXCEPTIONS
//     - JdmkIllegalAccessException
//     - JdmkInstantiationException
//     - JdmkClassNotFoundException
//     - JdmkServiceNotFoundException
//
// For more information, see 'newObj' in
// 'Interface com.sun.jaw.impl.adaptor.rmi.AdaptorServer'
//
// ----------------------------------------------------------------
      void newNamedObj(
                      in string className,
                      in string aLoader)

      raises (        JdmkIllegalAccessException,
                      JdmkInstantiationException,
                      JdmkClassNotFoundException,
                      JdmkServerRuntimeException,
                      JdmkServiceNotFoundException) ;

// ----------------------------------------------------------------
// DESCRIPTION
//      Creates an instance of a managed object in the remote
//     object server.
//      When calling the method, you can optionally provide
//     the class name of
//      the Java implementation to be used for instantiating
//     the new object.
//
// PARAMETERS
//     - impl: The name of the Java implementation to be used
//     on the server.
//     - name: The name of the managed object to be created.
//     - plist: The list of initial values of the properties of the
//     new managed object.
//         It's a serialized java object.
//         It's class com.sun.jaw.reference.common.ModificationList
//     - aLoader: The name of a class loader to be used.
//
// RETURN
//     The newly created managed object. It's an Object name
//     and a serialized java object.
//     It's class is com.sun.jaw.reference.common.PropertyList
//
// EXCEPTIONS
//     - JdmkIllegalAccessException
//     - JdmkClassNotFoundException
//     - JdmkServiceNotFoundException
//     - JdmkInstanceAlreadyExistException
//     - JdmkInstantiationException
//     - JdmkInvocationTargetException
//     - JdmkInvalidSerializedListException
//
// For more information, see 'newMO' in
// 'Interface com.sun.jaw.impl.adaptor.rmi.AdaptorServer'
//
// ----------------------------------------------------------------
      string newMOAndLoader(
                      in string impl,
                      in string name,
                      in SerializedObjectType plist,
                      in string aLoader)

      raises (        JdmkIllegalAccessException,
                      JdmkClassNotFoundException ,
                      JdmkServiceNotFoundException ,
                      JdmkInstanceAlreadyExistException ,
                      JdmkInstantiationException ,
                      JdmkInvocationTargetException ,
                      JdmkServerRuntimeException,
                      JdmkInvalidSerializedListException) ;



// ----------------------------------------------------------------
// DESCRIPTION
//      Creates a persistent instance of a managed object in the
//     remote object
//      server. When calling the method, you can optionally
//     provide the class
//      name of the Java implementation to be used for instantiating
//     the new
//      object.
//
// PARAMETERS
//     - impl: The name of the Java implementation to be used
//     on the server.
```

```
//     - name: The name of the managed object to be created.
//     - plist: The list of initial values of the properties of the
//     new managed object.
//         It's a serialized java object.
//         It's class com.sun.jaw.reference.common.ModificationList
//     - aLoader: The name of a class loader to be used.
//
// RETURN
//     The newly created managed object. It's an Object name
//     and a serialized java object.
//     It's class is com.sun.jaw.reference.common.PropertyList
//
// EXCEPTIONS
//     - JdmkIllegalAccessException
//     - JdmkClassNotFoundException
//     - JdmkServiceNotFoundException
//     - JdmkInstanceAlreadyExistException
//     - JdmkInstantiationException
//     - JdmkInvocationTargetException
//     - JdmkInvalidSerializedListException
//
// For more information, see 'newDBMO' in
// 'Interface com.sun.jaw.impl.adaptor.rmi.AdaptorServer'
//
// ---------------------------------------------------------------
      string newDBMOAndLoader(
                      in string impl,
                      in string name,
                      in SerializedObjectType plist,
                      in string aLoader)

      raises (        JdmkIllegalAccessException ,
                      JdmkClassNotFoundException ,
                      JdmkServiceNotFoundException ,
                      JdmkInstanceAlreadyExistException ,
                      JdmkInstantiationException ,
                      JdmkInvocationTargetException ,
                      JdmkServerRuntimeException,
                      JdmkInvalidSerializedListException) ;

// ---------------------------------------------------------------
// DESCRIPTION
//      Adds a named object under the control of the remote CMF.
//      Use this method with care, because it moves an instance
//      remotely. All methods of this instance are executed locally on
//      the remote agent.
//
// PARAMETERS
//     - object: The object to be added to the remote repository.
//     It's a serialized java object.
//               It's class is  java.lang.Object
//     - logicalName: The logical name of the object.
//
// RETURN
//
// EXCEPTIONS
//     - JdmkServiceNotFoundException
//     - JdmkInstanceAlreadyExistException
//     - JdmkInvalidSerializedListException
//
// For more information, see 'addObject' in
// 'Interface com.sun.jaw.impl.adaptor.rmi.AdaptorServer'
//
// ---------------------------------------------------------------
      void addObject(
                      in SerializedObjectType object,
                      in string logicalName)

      raises (        JdmkServiceNotFoundException,
                      JdmkInstanceAlreadyExistException,
                      JdmkServerRuntimeException,
                      JdmkInvalidSerializedListException );

// ---------------------------------------------------------------
// DESCRIPTION
// Deletes an instance of a managed object in the remote object server.
//
// PARAMETERS
//     - name: The name of the managed object to be deleted.
//
// RETURN
//
// EXCEPTIONS
//    - JdmkInvocationTargetException
//    - JdmkServiceNotFoundException
//    - JdmkInstanceNotFoundException
//
// For more information, see 'deleteMO' in
// 'Interface com.sun.jaw.impl.adaptor.rmi.AdaptorServer'
//
// ---------------------------------------------------------------
      void deleteMO(
                      in string name)

      raises (        JdmkInvocationTargetException,
                      JdmkServiceNotFoundException,
                      JdmkServerRuntimeException,
                      JdmkInstanceNotFoundException );

// ---------------------------------------------------------------
// DESCRIPTION
//      Allows a listener for a managed object to be added.
//     Transparently to
//      the caller, the adaptor creates and registers a listener
//     for the object
//      within the managed object server. The managed object
//     to be listened to
//      is specified by the <VAR>mo</VAR> parameter.
//      When the remote object fires an event by calling a method
///     of the remote listener, the corresponding method in the
//     local listener
//      is called.
//
// PARAMETERS
//      mo: The managed object to be listened to.
//      listen: The listener to be created in the agent.
//      receiver: The name of the receiver to be called for
//     forwarding the event.
//
// RETURN
//      A reference to the created listener.
//
// EXCEPTIONS
//      - JdmkInstanceNotFoundException
//      - JdmkIllegalAccessException
//      - JdmkServiceNotFoundException
//      - JdmkClassNotFoundException
//      - JdmkInstantiationException
//
// ---------------------------------------------------------------
      string  addListener(
                      in string mo,
                      in string listen,
                      in string receiver)

      raises (        JdmkInstanceNotFoundException,
                      JdmkIllegalAccessException,
                      JdmkServiceNotFoundException,
                      JdmkClassNotFoundException,
                      JdmkServerRuntimeException,
```

```
                    JdmkInstantiationException );

// ----------------------------------------------------------------
// DESCRIPTION
//      Allows a listener for a managed object to be removed.
//
// PARAMETERS
//      ref: A reference of the listener to be removed.
//
// RETURN
//     No return parameter
//
// EXCEPTIONS
//     No exception
//
// ----------------------------------------------------------------
      void removeListener(in string ref)

      raises (        JdmkServerRuntimeException) ;

// ----------------------------------------------------------------
// DESCRIPTION
//     Allows any method to be applied to a remote object.
//
// PARAMETERS
//     - objName: The name of the remote object.
//     - pfName: The name of the method to be applied.
//     - params: An array containing the parameters to be
//     passed to the method.
//              It's a serialized java object
//               It's class is java.lang.Object
//     - signature: The signature of the method to be called.
//
// RETURN
//     The value of the method applied.
//     It's a serialized java object.
//     It's class is java.lang.Object
//
// EXCEPTIONS
//     - Jdmk InstanceNotFoundException,
//     - JdmkInvocationTargetException,
//     - JdmkServiceNotFoundException,
//     - JdmkNoSuchMethodException,
//     - JdmkIllegalAccessException
//     - JdmkInvalidSerializedListException
//
// For more information, see 'invokePerform' in
// 'Interface com.sun.jaw.impl.adaptor.rmi.AdaptorServer'
//
// ----------------------------------------------------------------
      SerializedObjectType invokePerform(
                    in string name,
                    in string pfName,
                    in SerializedObjectListType params,
                    in StringListType signature)

      raises (        JdmkInstanceNotFoundException,
                    JdmkInvocationTargetException,
                    JdmkServiceNotFoundException,
                    JdmkNoSuchMethodException,
                    JdmkIllegalAccessException,
                    JdmkServerRuntimeException,
                    JdmkInvalidSerializedListException);

// ----------------------------------------------------------------
// DESCRIPTION
//     Returns the version of this class.
//
// PARAMETERS
//     None
//
// RETURN
//     Returns the version of this class.
//
// For more information, see 'getClassVersion' in
// 'Interface com.sun.jaw.impl.adaptor.rmi.AdaptorServer'
//
// ----------------------------------------------------------------
      string getClassVersion() ;

// ----------------------------------------------------------------
// DESCRIPTION
//      Returns the service name  of this IIOP object
//
// PARAMETERS
//     None
//
// RETURN
//      Returns the service name  of this IIOP object
//
// For more information, see 'getServiceName' in
// 'Interface com.sun.jaw.impl.adaptor.rmi.AdaptorServer'
//
// ----------------------------------------------------------------
      string getServiceName() ;

// ----------------------------------------------------------------
// DESCRIPTION
//     Returns the name of the protocol (iiop)
//
// PARAMETERS
//     None
//
// RETURN
//     Returns the name of the protocol (iiop)
//
// For more information, see 'getProtocol' in
// 'Interface com.sun.jaw.impl.adaptor.rmi.AdaptorServer'
//
// ----------------------------------------------------------------
      string getProtocol() ;



} ; // END OF interface AdaptorIIOP


} ; // END OF module CorbaServer
```

D.2 GkStatistics.idl

```
module GkStatisticsApp
{
   // ---------------------------------------------------------------
   // typedef declaration
   // ---------------------------------------------------------------
   //
   // String list
   //
   typedef sequence <string>  StringListType;

   //
   // object serialization
   //
   typedef sequence <octet>   SerializedObjectType ;

   //
   // serialized object list
   //
   typedef sequence <SerializedObjectType>   SerializedObjectListType ;

   interface GkStatistics
   {
      // ---------------------------------------------------------------
      // DESCRIPTION
      //     Get value of GkStatsCurrentNoOfCalls-Property, found in GkStatistics-H-Bean
      //     registered in JDHK-Agent.
      //
      // PARAMETERS
      //     - none
      //
      // RETURN
      //     long (Attribute "GkStatsCurrentNoOfCalls")
      //
      // EXCEPTIONS
      //     - none
      //
      // ---------------------------------------------------------------
      long getGkStatsCurrentNoOfCalls();
      // ---------------------------------------------------------------
      // DESCRIPTION
      //     Set value of gkStatsCurrentNoOfCalls-Property, found in GkStatistics-H-Bean
      //     registered in JDHK-Agent.
      //
      // PARAMETERS
      //     - NewGkStatsCurrentNoOfCalls  : New Value for GkStatsCurrentNoOfCalls Attribute
      //
      // RETURN
      //     No  return parameter
      //
      // EXCEPTIONS
      //
      // ---------------------------------------------------------------
      void setGkStatsCurrentNoOfCalls(in long NewGkStatsCurrentNoOfCalls);
      // ---------------------------------------------------------------
      // DESCRIPTION
      //     Get value of GkStatsCurrentBandwidth-Property, found in GkStatistics-H-Bean
      //     registered in JDHK-Agent.
      //
      // PARAMETERS
      //     - none
      //
      // RETURN
      //     long (Attribute "GkStatsCurrentBandwidth")
      //
      // EXCEPTIONS
      //     - none
      //
      // ---------------------------------------------------------------
      long getGkStatsCurrentBandwidth();
      // ---------------------------------------------------------------
      // DESCRIPTION
      //     Set value of GkStatsCurrentBandwidth-Property, found in GkStatistics-H-Bean
      //     registered in JDHK-Agent.
      //
      // PARAMETERS
      //     - NewGkStatsCurrentBandwidth  : New Value for GkStatsCurrentBandwidth Attribute
      //
      // RETURN
      //     No  return parameter
      //
      // EXCEPTIONS
      //
      // ---------------------------------------------------------------
      void setGkStatsCurrentBandwidth(in long NewGkStatsCurrentBandwidth);
   };
};
```

Anhang E

Implementierung der MIB-Gruppe “GKStatistics”


```
package tis20.impl;

// Copyright (c) 07/24/98, by Sun Microsystems, Inc.
// All rights reserved.

// "@(#)SystemImpl.java 1.1 98/07/24 SMI"

import java.io.*;
import java.util.*;
import java.net.*;
import java.lang.*;
import java.text.*;

// Dependency on Java Dynamic Management Kit.
import com.sun.jaw.impl.adaptor.snmp.*;
import com.sun.jaw.reference.agent.cmf.*;
import com.sun.jaw.reference.common.*;
import com.sun.jaw.snmp.agent.*;
import com.sun.jaw.snmp.common.*;

// Dependency on Java Native Interface
import jni.*;

// Dependency on Generated Files
import tis20.mibgen.*;

// Other dependencies
import tools.*;
import trial.*;

public class GkStatisticsImpl extends GkStatistics {

   public GkStatisticsImpl(SnmpMib myMib, Framework cmf) {
      super(myMib);
      init(myMib, cmf);

      // Get these Objects to insert and remove Tableentries
   GkStatisticsMIB = myMib;
   GkStatisticsCMF = cmf;
}

private static GkStatEntryImpl GkStatEntry[] = new GkStatEntryImpl[10];

private void init(SnmpMib myMib, Framework cmf) {
   // Initialize the system description using some system properties
   //
   try {
      GkStatsCurrentNoOfCalls = getGkStatsCurrentNoOfCalls();
      GkStatsCurrentBandwidth = getGkStatsCurrentBandwidth();
      initGkStatTable(myMib, cmf);
   } catch (SecurityException e) {
      // Do not process the exception
   } catch (SnmpStatusException e) {
      // Do not process the exception
   }
   startUpTime= java.lang.System.currentTimeMillis();
}

/**
* Getter for the "GkStatsCurrentNoOfCalls" variable.
*/
public Integer getGkStatsCurrentNoOfCalls() throws SnmpStatusException {
   String oid = "1.3.6.1.4.1.2206.5.1.3.4.1";
   //GkStatsCurrentNoOfCalls = new Integer(nativeBase.getIntNative(tools.ConvertTools.String2Int(oid)));
   return GkStatsCurrentNoOfCalls;
}

/**
 * Setter for the "GkStatsCurrentNoOfCalls" variable.
 */
public  void setGkStatsCurrentNoOfCalls(Integer x) throws SnmpStatusException {
   String oid =  "1.3.6.1.4.1.2206.5.1.3.4.1.0";
   GkStatsCurrentNoOfCalls = x;
   //nativeBase.setNative(tools.ConvertTools.String2Int(oid), x.intValue());
}

/**
 * Checker for the "GkStatsCurrentNoOfCalls" variable.
 */
public  void checkGkStatsCurrentNoOfCalls(Integer x) throws SnmpStatusException {
   //
   // Add your own checking policy.
   //
}

/**
* Getter for the "GkStatsCurrentBandwidth" variable.
*/
public Integer getGkStatsCurrentBandwidth() throws SnmpStatusException {
   //int[] mOIDintArray =  {1,3,6,1,4,1,2206,5,1,1,1,2,0};
   String oid =  "1.3.6.1.4.1.2206.5.1.3.4.2";
   //GkStatsCurrentBandwidth = new Integer(nativeBase.getStringNative(tools.ConvertTools.String2Int(oid)));
   return GkStatsCurrentBandwidth;
}

/**
 * Setter for the "GkStatsCurrentBandwidth" variable.
 */
public  void setGkStatsCurrentBandwidth(Integer x) throws SnmpStatusException {
   //int[] mOIDintArray =  {1,3,6,1,4,1,2206,5,1,1,1,2,0};
   String oid =  "1.3.6.1.4.1.2206.5.1.3.4.2.0";
   GkStatsCurrentBandwidth = x;
   //nativeBase.setNative(tools.ConvertTools.String2Int(oid), x.intValue());
}

/**
 * Checker for the "GkStatsCurrentBandwidth" variable.
```

```
 */
public  void checkGkStatsCurrentBandwidth(Integer x) throws SnmpStatusException {
   //
   // Add your own checking policy.
   //
}

private void initGkStatTable(SnmpMib myMib, Framework cmf) {

   //Loading the GkStatTable
   //to initialize, it's better to use the oid as an integer-array
   int[][] oid = {{1,3,6,1,4,1,2206,5,1,3,4,4,1,1}, {1,3,6,1,4,1,2206,5,1,3,4,4,1,2}
      ,{1,3,6,1,4,1,2206,5,1,3,4,4,1,3}, {1,3,6,1,4,1,2206,5,1,3,4,4,1,4}
      ,{1,3,6,1,4,1,2206,5,1,3,4,4,1,5}};
   String[] StartOid = {"1.3.6.1.4.1.2206.5.1.3.4.4.1.1", "1.3.6.1.4.1.2206.5.1.3.4.4.1.2"
      ,"1.3.6.1.4.1.2206.5.1.3.4.4.1.3", "1.3.6.1.4.1.2206.5.1.3.4.4.1.4"
      ,"1.3.6.1.4.1.2206.5.1.3.4.4.1.5"};

   int entries = 0;
   int NrOfRows;
   int tempInt = 0;

   GkStatEntry[entries] = new GkStatEntryImpl(myMib, cmf, GkStatTable);

   boolean NoMoreRows = false;

   // Get the GkStatIndex
   try {
      //tempInt = nativeBase.getNextIntNative(oid,0);
      GkStatEntry[entries].setGkStatIndex(new Integer(tempInt));
      NoMoreRows = (StartOid[0].equals(tools.ConvertTools.Int2String(oid)[0]));
   } catch (Exception e) {
      e.printStackTrace();
   }

   while (!NoMoreRows) {
      entries++;
      GkStatEntry[entries] = new GkStatEntryImpl(myMib);
      try {
         //tempInt = nativeBase.getNextIntNative(oid,0);
         GkStatEntry[entries].setGkStatIndex(new Integer(tempInt));
         NoMoreRows = (StartOid[0].equals(tools.ConvertTools.Int2String(oid)[0]));
      } catch (IllegalArgumentException iae) {
         // no more rows !??
         NoMoreRows = true;
      } catch (Exception e) {
         e.printStackTrace();
      }
   }

   NrOfRows = entries;

   // Get the GkStatNoOfCalls
   // inputOID now at first Element of second column
   entries = 0;
   while (entries != NrOfRows) {
      int tempGkStatNoOfCalls = 0;
      try {
         //tempGkStatNoOfCalls = nativeBase.getNextIntNative(oid,1);
         //GkStatEntry[entries].setGkStatNoOfCalls(new Integer(tempGkStatNoOfCalls));
      } catch (Exception e) {
         e.printStackTrace();
      }
      entries++;
   }

   // Get the GkStatMaxNoOfCalls
   // inputOID now at first Element of third column
   entries = 0;
   while (entries != NrOfRows) {
      int tempGkStatMaxNoOfCalls = 0;
      try {
         //tempGkStatMaxNoOfCalls = nativeBase.getNextIntNative(oid,2);
         //GkStatEntry[entries].setGkStatMaxNoOfCalls(new Integer(tempGkStatMaxNoOfCalls));
      } catch (Exception e) {
         e.printStackTrace();
      }
      entries++;
   }

   // Get the GkStatAverageLength
   // inputOID now at first Element of fourth column
   entries = 0;
   while (entries != NrOfRows) {
      int tempGkStatAverageLength = 0;
      try {
         //tempGkStatAverageLength = nativeBase.getNextIntNative(oid,3);
         //GkStatEntry[entries].setGkStatAverageLength(new Integer(tempGkStatAverageLength));
      } catch (Exception e) {
         e.printStackTrace();
      }
      entries++;
   }

   // Get the GkStatPeakBandwidth
   // inputOID now at first Element of fifth column
   entries = 0;
   while (entries != NrOfRows) {
      int tempGkStatPeakBandwidth = 0;
      try {
         //tempGkStatPeakBandwidth = nativeBase.getNextIntNative(oid,4);
         //GkStatEntry[entries].setGkStatPeakBandwidth(new Integer(tempGkStatPeakBandwidth));
      } catch (Exception e) {
         e.printStackTrace();
      }
      entries++;
   }

   entries = 0;
   while (entries != NrOfRows) {
      try {
         //Register in GkStatTable
         GkStatTable.addEntry(GkStatEntry[entries]);
      } catch (Exception e) {
         e.printStackTrace();
      }
      try {
         //Register in Core management Framework
         GkStatEntry[entries].addInCmf(cmf);
      } catch (Exception e) {
         e.printStackTrace();
      }
      entries++;
   }

   // Only for test without running gatekeeper !!!
   try {
      GkStatEntry[0].setGkStatIndex(new Integer(1));
      /*
      GkStatEntry[0].setGkStatNoOfCalls(new Integer(0));
      GkStatEntry[0].setGkStatMaxNoOfCalls(new Integer(0));
      GkStatEntry[0].setGkStatAverageLength(new Integer(0));
      GkStatEntry[0].setGkStatPeakBandwidth(new Integer(0));
      */
      try {
         //Register in GkStatTable
         GkStatTable.addEntry(GkStatEntry[entries]);
      } catch (Exception e) {
         e.printStackTrace();
      }
```

```
      try {
         //Register in Core management Framework
         GkStatEntry[entries].addInCmf(cmf);
      } catch (Exception e) {
         e.printStackTrace();
      }
   } catch (SnmpStatusException e) {
      // Proceed ?
   }
}

public void performCreateGkStatEntry(int NewGkStatIndex)
{
   String[] oid = {"1.3.6.1.4.1.2206.5.1.3.4.4.1.1", "1.3.6.1.4.1.2206.5.1.3.4.4.1.2"
      ,"1.3.6.1.4.1.2206.5.1.3.4.4.1.3", "1.3.6.1.4.1.2206.5.1.3.4.4.1.4"
      ,"1.3.6.1.4.1.2206.5.1.3.4.4.1.5"};

   GkStatEntryImpl GkStatEntry =
   new GkStatEntryImpl(GkStatisticsMIB, GkStatisticsCMF, GkStatTable);

   int NewGkStatNoOfCalls = 0;
   int NewGkStatMaxNoOfCalls = 0;
   int NewGkStatAverageLength = 0;
   int NewGkStatPeakBandwidth = 0;

   try {
      int[] LegalGkStatIndex = new int[48];
      for (int i = 0; i<LegalGkStatIndex.length; i++) {
         LegalGkStatIndex[i] = i+1;
      }

      // Check index range
      if (!tools.ArrayTools.isElement(NewGkStatIndex, LegalGkStatIndex)) {
         throw new ArrayIndexOutOfBoundsException();
      }


      // Set the M-bean properties
      GkStatEntry.setGkStatIndex(new Integer(NewGkStatIndex));

      // get the Index and oid
      SnmpIndex testIndex = GkStatTable.buildSnmpIndex(GkStatEntry);
      SnmpOid testOid = GkStatTable.buildOidFromIndex(testIndex);

      // concatenate base-oid with index
      oid[0] = oid[0] + "." + testOid.toString();
      oid[1] = oid[1] + "." + testOid.toString();
      oid[2] = oid[2] + "." + testOid.toString();
      oid[3] = oid[3] + "." + testOid.toString();
      oid[4] = oid[4] + "." + testOid.toString();

      // convert in useable format for the native interface
      int[][] tempOid = tools.ConvertTools.String2Int(oid);

      // set values in gatekeeper
      nativeBase.setNative(tempOid[0], NewGkStatIndex);

      //Register in PlatformMiscUserTable
      System.out.print("Register entry in GkStatTable ...");
      GkStatTable.addEntry(GkStatEntry);
      System.out.println(" done.");

      //Register in Core Management Framework
      System.out.print("Register entry in CMF ...");
      GkStatEntry.addInCmf(GkStatisticsCMF);
      System.out.println(" done.");
   } catch (SnmpStatusException snmpe) {
      System.out.println("Index already used, please check entries for free index!");
   } catch (ArrayIndexOutOfBoundsException aioobe) {
      System.out.println("Wrong index (must be between 1..48), please retry");
   } catch (NullPointerException npe) {
      System.out.println("Entries not complete, please retry");
   } catch (Exception e) {
      e.printStackTrace();
   }
}

// Private variables.
//
private SnmpMib GkStatisticsMIB;
private Framework GkStatisticsCMF;
}
```

Anhang F

Die Agenten

F.1 Master Agent

```
package trial;

import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.net.*;

import com.sun.jaw.reference.agent.cmf.*;
import com.sun.jaw.reference.common.*;
import com.sun.jaw.reference.agent.services.*;
import com.sun.jaw.impl.agent.services.security.*;
// Adaptors
import com.sun.jaw.impl.adaptor.security.*;
//import com.sun.jaw.impl.adaptor.generic.AdaptorServer;
// Discovery
import com.sun.jaw.impl.agent.services.jawdiscovery.*;
import com.sun.jaw.impl.agent.services.jawdiscovery.DiscoveryResponder;
//M-Let
import com.sun.jaw.impl.agent.services.mlet.MLetSrv;
// Cascading
import com.sun.jaw.impl.agent.services.cascading.*;

import com.sun.jaw.impl.adaptor.generic.AdaptorServer;

//import tis20.impl.*;
//import trial.discovery.monitor.DiscMonitor;

public class MasterAgent \{

   public static void main(String args[]) \{
      Framework  cmf;
      MoRepSrvIf rep;
      Class      repSrv;
      String     repName;
      String     mtdName;
      String     domain;
      String     mletName;

      Debug.setOnAll();
      //Debug.setOn(9);
```

```
//Debug.parseDebugProperties();
//Debug.println("Default debug is on");

// Create a security manager which allows loading of classes
// containing native code.
//
java.lang.System.setSecurityManager(new AgentSecurityManager());

repName     = "com.sun.jaw.impl.agent.services.light.RepositorySrv";
mtdName     = "com.sun.jaw.impl.agent.services.light.MetaDataSrv";
mletName    = "com.sun.jaw.impl.agent.services.mlet.MLetSrv";

try \{
   repSrv = Class.forName(repName);
   rep    = (MoRepSrvIf)repSrv.newInstance();
   cmf    = new Framework((MoRepSrvIf)rep, null);
   domain = cmf.getDomain();

   //
   // Create a metadata service
   //
   cmf.newObject(mtdName, domain + ":" + ServiceName.META, null);


   // To add Authentification for HTTP-Adaptor
   AuthInfo user1 = new AuthInfo("siemens", "hicom");

   //Start the Discovery-Responder
   DiscoveryResponder disc =
   (DiscoveryResponder)
   cmf.newObject("com.sun.jaw.impl.agent.services.jawdiscovery.DiscoveryResponder"
                , domain + ":com.sun.jaw.impl.agent.services.jawdiscovery.DiscoveryResponder"
                ,null);


   String AdaptorTypes[] = \{ "html", "rmi" \};
   int AdaptorPorts[] = \{ 8092, 1098 \};
   int NrOfAdaptors = AdaptorTypes.length;
   AdaptorServer AdaptorServers[] = new AdaptorServer[NrOfAdaptors];
```

```
   for (int i = 0; i < NrOfAdaptors; i++) \{
      String SpecialAdaptor = new String("com.sun.jaw.impl.adaptor."
             +AdaptorTypes[i] + ".AdaptorServerImpl");
      String SpecialAdaptorMO = new String("Adaptor(s):"
                                          + "com.sun.jaw.reference.client.adaptor.AdaptorMO"
                                          + ".protocol=" + AdaptorTypes[i]
                                          + ",port="
                                          + AdaptorPorts[i]);
      Debug.println(SpecialAdaptor + " to CMF with name \\n\\t" + SpecialAdaptorMO);
      AdaptorServers[i] = (AdaptorServer)cmf.newObject(SpecialAdaptor, SpecialAdaptorMO, null);
      if (AdaptorTypes[i].equals("http")) \{
         ((com.sun.jaw.impl.adaptor.html.AdaptorServerImpl)AdaptorServers[i])
 .addUserAuthenticationInfo(user1);
      \}
   \}

     // Check the Discovery Search Service
     try \{
        int multicastPort = 9000;
        String multicastGroup = "224.224.224.224";
        int ttl = 1;

        DiscoveryClient discoveryClient = new DiscoveryClient();
        discoveryClient.setMulticastGroup(multicastGroup);
        discoveryClient.setMulticastPort(multicastPort);
        discoveryClient.setTimeToLiveInt(ttl);
        discoveryClient.performStart();
        Vector result = discoveryClient.performFindAdaptors();
        Enumeration adaptorList = result.elements();
        while (adaptorList.hasMoreElements()) \{
           DiscoveryResponse response = (DiscoveryResponse) adaptorList.nextElement();
           Vector adaptors = response.objectList;
           Enumeration enum = adaptors.elements();
           while (enum.hasMoreElements()) \{
              System.out.println("Adaptor="+enum.nextElement().toString());

           \}
        \}
        discoveryClient.performStop();
     \} catch (Exception e) \{
        e.printStackTrace();
     \}

   // Use M-Let Service to load TestClasses i.e. Discovery-Monitor
```

```
      // Create Instance of MLet-Server
      MLetSrv mlet = (MLetSrv) cmf.newObject(mletName, domain
                                             + ":"
                                             + "com.sun.jaw.impl.mo.MLetSrvMO"
                                             , null);

      //Load DiscMonitor into Master-Agent
      Vector Beans = mlet
  .performLocalLoadURL("http://pn62007308.mch.pn.siemens.de/DiscMonitor.html");
      //Vector Beans = mlet.performLocalLoadURL("file:/jdmk/trial/DiscMonitor.html");
      if (Beans.isEmpty()) \{
         System.out.println("No Beans found!");
      \} else \{
         cmf.newObject("trial.discovery.monitor.DiscMonitor",domain + ":DiscMonitor",null);
      \}


      // get FirstSubAgent with cascading service

      RemoteAgent FirstSubAgent = (RemoteAgent)
                       cmf.newObject("com.sun.jaw.impl.agent.services.cascading.RemoteAgent",
                                     "Adaptor(s)" + ":" +
                                     "com.sun.jaw.impl.agent.services.cascading.RemoteAgentMO" +
                                     ".protocol=" + "rmi" +
                                     ",port=" + "1096"
                                     , null);

      RemoteAgent SecondSubAgent = (RemoteAgent)
                       cmf.newObject("com.sun.jaw.impl.agent.services.cascading.RemoteAgent",
                                     "Adaptor(s)" + ":" +
                                     "com.sun.jaw.impl.agent.services.cascading.RemoteAgentMO" +
                                     ".protocol=" + "rmi" +
                                     ",port=" + "1097"
                                     , null);

   \} catch (Exception e) \{
      e.printStackTrace();
   \}
 \}
\}
```

F.2 FirstSubAgent

```
package trial;

import java.util.Vector;
import java.net.*;

import com.sun.jaw.reference.agent.cmf.*;
import com.sun.jaw.reference.common.*;
import com.sun.jaw.reference.agent.services.*;
import com.sun.jaw.impl.agent.services.security.*;
import com.sun.jaw.impl.adaptor.security.*;
import com.sun.jaw.impl.adaptor.generic.AdaptorServer;
import com.sun.jaw.impl.agent.services.jawdiscovery.DiscoveryResponder;

import com.sun.jaw.impl.adaptor.snmp.* ;

import tis20.mibgen.*;
import tis20.impl.*;
```

```
public class FirstSubAgent \{

   static AdaptorServerImpl snmpAdaptor= null;

   public static void main(String args[]) \{
      Framework  cmf;
      MoRepSrvIf rep;
      Class      repSrv;
      String     repName;
      String     mtdName;
      String     mibClassName;
      String     domain = null;

      //Debug.setOnAll();
      repName     = "com.sun.jaw.impl.agent.services.light.RepositorySrv";
      mtdName     = "com.sun.jaw.impl.agent.services.light.MetaDataSrv";
      mibClassName= "tis20.mibgen.SBCS_TIS2_MIB_Impl";

      try \{
```

```
      repSrv = Class.forName(repName);
      rep    = (MoRepSrvIf)repSrv.newInstance();
      cmf    = new Framework((MoRepSrvIf)rep, null);
      //domain = new String("FirstSubAgent");//cmf.getDomain();
      domain = cmf.getDomain();
      //
      // Create a metadata service
      //
      cmf.newObject(mtdName, domain + ":" + ServiceName.META, null);
      //Start the Discovery-Responder
      DiscoveryResponder disc =
      (DiscoveryResponder)
      cmf.newObject("com.sun.jaw.impl.agent.services.jawdiscovery.DiscoveryResponder"
                    , domain + ":"
  + "com.sun.jaw.impl.agent.services.jawdiscovery.DiscoveryResponder"
                    ,null);
      //Create Adaptors
      String snmpName = null;
      String AdaptorTypes[] = \{ "html", "rmi", "http", "snmp", "iiop" \};

      int AdaptorPorts[] = \{ 8082, 1096, 8081, 8083, 8085 \};
      int NrOfAdaptors = AdaptorTypes.length;
      AdaptorServer AdaptorServers[] = new AdaptorServer[NrOfAdaptors];
      for (int i = 0; i < NrOfAdaptors; i++) \{
         String SpecialAdaptor = new String("com.sun.jaw.impl.adaptor."
   +AdaptorTypes[i] + ".AdaptorServerImpl");
         String SpecialAdaptorMO = new String("Adaptor(s):"
                                          + "com.sun.jaw.reference.client.adaptor.AdaptorMO"
                                          + ".protocol=" + AdaptorTypes[i]
                                          + ",port=" + AdaptorPorts[i]
                                          );
         if (AdaptorTypes[i].equals("snmp")) \{
            snmpName = SpecialAdaptorMO;
         \}
         Debug.println("Adding" + SpecialAdaptor
 + " to CMF with name \\n\\t" + SpecialAdaptorMO);
         AdaptorServers[i] = (AdaptorServer)
 cmf.newObject(SpecialAdaptor, SpecialAdaptorMO, null);
         \}

         //
         // Create the MIB II (RFC 1213)
         //
         //String mibName= new String("snmp:SBCS_TIS2_MIB_Impl");
         String mibName= new String(domain + ":SBCS_TIS2_MIB_Impl");
         Debug.println("Adding SBCS_TIS2_MIB to CMF with name \\n\\t" + mibName);
         SBCS_TIS2_MIB_Impl mib =  (SBCS_TIS2_MIB_Impl)
 cmf.newObject(mibClassName, mibName, null);
         //
         // Bind the SNMP adaptor to the MIB in order to make the MIB accessible
         // through the SNMP protocol.
         // If this step is not performed, the MIB will still live in the JDMK agent:
         // its objects will be addressable through HTML/RMI/IIOP but not SNMP.
         //
         mib.setSnmpAdaptorName(snmpName);
      \} catch (Exception e) \{
         e.printStackTrace();
      \}
   \}

   static public AdaptorServerImpl getSnmpAdaptor() \{
      return snmpAdaptor;
   \}

   public void performShutdownAgent(DiscoveryResponder disc) \{
      disc.performStop();
      try \{
         System.exit(0);
      \} catch (Exception e) \{
         e.printStackTrace();
      \}
   \}
\}
```

F.3 SecondSubAgent

```
package trial;
import java.util.Vector;
import java.net.*;
import com.sun.jaw.reference.agent.cmf.*;
import com.sun.jaw.reference.common.*;
import com.sun.jaw.reference.agent.services.*;
import com.sun.jaw.impl.agent.services.security.*;
import com.sun.jaw.impl.adaptor.security.*;
import com.sun.jaw.impl.agent.services.jawdiscovery.DiscoveryResponder;
import com.sun.jaw.impl.adaptor.generic.AdaptorServer;
import com.sun.jaw.impl.adaptor.snmp.* ;
import tis20.mibgen.*;
import tis20.impl.*;

public synchronized class SecondSubAgent \{

   static AdaptorServerImpl snmpAdaptor = null;

   public static void main(String args[]) \{
      Framework  cmf;
      MoRepSrvIf rep;
      Class      repSrv;
      String     repName;
      String     mtdName;
      String     mibClassName;
      String     domain;

      //Debug.setOnAll();
      // Create a security manager which allows loading of classes
      // containing native code.
      //
      java.lang.System.setSecurityManager(new AgentSecurityManager());

      repName     = "com.sun.jaw.impl.agent.services.light.RepositorySrv";
      mtdName     = "com.sun.jaw.impl.agent.services.light.MetaDataSrv";
      mibClassName= "tis20.mibgen.SBCS_TIS2_MIB";

      try \{
         repSrv = Class.forName(repName);
         rep    = (MoRepSrvIf)repSrv.newInstance();
         cmf    = new Framework((MoRepSrvIf)rep, null);
         domain = cmf.getDomain();
         //
         // Create a metadata service
         //
         cmf.newObject(mtdName, domain + ":" + ServiceName.META, null);
         //Start the Discovery-Responder
         //DiscoveryResponder disc =
         //(DiscoveryResponder)
         cmf.newObject("com.sun.jaw.impl.agent.services.jawdiscovery.DiscoveryResponder"
```

```
                        , domain
                        + ":com.sun.jaw.impl.agent.services.jawdiscovery.DiscoveryResponder"
                        ,null);
            // To add Authentification for HTTP-Adaptor
            AuthInfo user1 = new AuthInfo("siemens", "hicom");
            AuthInfo user2 = new AuthInfo("root", "admin");
            //Create Adaptors
            String snmpName = null;
            //String AdaptorTypes[] = \{ "html", "rmi", "http", "snmp", "iiop" \};
            //int AdaptorPorts[] = \{ 8087, 1097, 8086, 8089, 8085 \};
            String AdaptorTypes[] = \{ "html", "rmi", "http", "snmp" \};
            int AdaptorPorts[] = \{ 8087, 1097, 8086, 8089 \};
            int NrOfAdaptors = AdaptorTypes.length;
            AdaptorServer AdaptorServers[] = new AdaptorServer[NrOfAdaptors];
            for (int i = 0; i < NrOfAdaptors; i++) \{
               String SpecialAdaptor = new String("com.sun.jaw.impl.adaptor."
                                               +AdaptorTypes[i] + ".AdaptorServerImpl");
               String SpecialAdaptorMO = new String("Adaptor(s):"
                                                 + "com.sun.jaw.reference.client.adaptor.AdaptorMO"
                                                 + ".protocol=" + AdaptorTypes[i]
                                                 + ",port="
                                                 + AdaptorPorts[i]);
               if (AdaptorTypes[i].equals("snmp")) \{
                  snmpName = SpecialAdaptorMO;
               \}
               Debug.println(SpecialAdaptor + " to CMF with name \\n\\t" + SpecialAdaptorMO);
               AdaptorServers[i] = (AdaptorServer)cmf
.newObject(SpecialAdaptor, SpecialAdaptorMO, null);
               if (AdaptorTypes[i].equals("http")) \{
                  ((com.sun.jaw.impl.adaptor.http.AdaptorServerImpl)AdaptorServers[i])
        .addUserAuthenticationInfo(user1);
               \}
               if (AdaptorTypes[i].equals("html")) \{
                  ((com.sun.jaw.impl.adaptor.html.AdaptorServerImpl)AdaptorServers[i])
        .addUserAuthenticationInfo(user1);
               \}
            \}
            //
            // Create the MIB II (RFC 1213)
            //
            String mibName= new String("snmp:SBCS_TIS2_MIB");
            Debug.println("Adding SBCS_TIS2_MIB to CMF with name \\n\\t" + mibName);
            SBCS_TIS2_MIB mib=  (SBCS_TIS2_MIB) cmf
        .newObject(mibClassName, mibName, null);
            //
            // Bind the SNMP adaptor to the MIB in order to make the MIB accessible
            // through the SNMP protocol.
            // If this step is not performed, the MIB will still live in the JDMK agent:
            // its objects will be addressable through HTML/RMI/IIOP but not SNMP.
            //
            mib.setSnmpAdaptorName(snmpName);
         \} catch (Exception e) \{
            e.printStackTrace();
         \}
      \}
      static public AdaptorServerImpl getSnmpAdaptor() \{
         return snmpAdaptor;
      \}
   \}
```

Abkürzungsverzeichnis

Abkürzung	**Bedeutung**
A	
ACL	Access Control List
AMS	Administration Maintenance Server
API	Application Programming Interface
C	
C-Bean	Client Bean
CORBA	Common Object Request Broker Architecture
COS	Common Object Service
CMF	Core Management Framework
CRAM	Challenge Response Authentification Mechanism
D	
DLL	Dynamic Link Library
F	
FMA	Flexible Management Agenten
I	
IDL	Interface Definition Language
IIOP	Internet InterORB Protocol
IPC	Inter Process Communication
J	
JAR	Java Archive
JDK	Java Development Kit
JDMK	Java Dynamic Management Kit
JMAPI	Java Management API
JNI	Java Native Interface
JVM	Java Virtual Machine
M	
M-Bean	Management Bean
MD5	Message Digest 5
MIB	Management Information Base

MIBGEN	Management Information Base Generator
MLET	Management Applet (Service)
MO	Managed Object
MOGEN	Managed Object Generator

S

OEM	Original Equipment Manufacturer
OID	Object Identifier
OMG	Object Management Group
ORB	Object Request Broker

R

RMI	Remote Method Invokation

S

so	shared object library
SQL	Simple Query Language
SSL	Secure Socket Layer
SXA	SNMP Extension Agent

T

TIS	Telephony Internet Server

U

UTF	Unicode Text Form

Literaturverzeichnis

[agentX] WHITE, MATT: *An Overview of the AgentX Protocol.* The Simple Times, 6(1):1–6, March 1998.

[BL 94] BERNERS-LEE, T.: *RFC 1630: Universal Resource Identifiers in WWW: A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World-Wide Web.* RFC, IETF, Juni 1994.

[BLCo 95] BERNERS-LEE, T. und D. CONNOLLY: *RFC 1866: Hypertext Markup Language — 2.0.* RFC, IETF, November 1995.

[BLFF 96] BERNERS-LEE, T., R. FIELDING und H. FRYSTYK: *RFC 1945: Hypertext Transfer Protocol — HTTP/1.0.* RFC, IETF, Mai 1996.

[BLMM 94] BERNERS-LEE, T., L. MASINTER und M. MCCAHILL: *RFC 1738: Uniform Resource Locators (URL).* RFC, IETF, Dezember 1994.

[ct97/3] SCHWARZ, MARTIN: *COM, SOM und CORBA - oder die Suche nach dem Software-Esperanto.* c´t, 3:256, März 1997.

[ct98/18] OSTERMANN, KLAUS: *Programmieren mit Swing - GUI Entwicklung mit Java.* c´t, 18:168, 1998.

[ct98/20] PIEMONT, CLAUDIA: *Geistreiche Verbindungen - Intelligente Geräte in dezentralen Netzen.* c´t, 20:198, 1998.

[flan98] FLANAGAN, DAVID: *JAVA in a Nutshell.* O´Reilly, 2 Auflage, 1998.

[HeAb 94] HEGERING, H.-G. und S. ABECK: *Integrated Network and System Management.* Addison-Wesley, 1994.

[jdmk2] MICROSYSTEMS, SUN: *Java Dynamic Management Kit 2.0 - Programming Guide.* Sun Microsystems Inc., Februar 1998.

[jdmk3b] MICROSYSTEMS, SUN: *Java Dynamic Management Kit 3.0 Beta - Programming Guide.* Sun Microsystems Inc., August 1998.

[jdmkOnline] JAVASOFT: *JDMK-Onlinehilfe.* Technischer Bericht Sun Microsystems, May 1997.

[jidl96] MICROSYSTEMS, SUN: *Java IDL.* Sun Microsystems Inc., 1996/97.

[jni97] JAVASOFT: *Java Native Interface Specification.* JavaSoft, May 1997. 1.1.

[KCK 97] Klensin, J., R. Catoe und P. Krumviede: *RFC 2195: IMAP/POP AUTHorize Extension for Simple Challenge/Response*. RFC, IETF, September 1997.

[KPS 95] Kaufman, C., R. Perlman und M. Spencer: *Network Security*. Prentice Hall, Inc., 1995.

[Moun 97] Mountzia, M.-A.: *Flexible Agents in Integrated Network and Systems Management*. Dissertation, Technische Universität München, Dezember 1997.

[mur98] Murray, James D.: *Windows NT, SNMP*. O'Reilly, January 1998.

[Myer 94] Myers, J.: *RFC 1734: POP3 AUTHentication command*. RFC, IETF, Dezember 1994.

[MyRo 96] Myers, J. und M. Rose: *RFC 1939: Post Office Protocol — Version 3*. RFC, IETF, Mai 1996.

[NeMa 95] Nebel, E. und L. Masinter: *RFC 1867: Form-based File Upload in HTML*. RFC, IETF, November 1995.

[NiePeck97] Peck, Patrick Niemeyer & Joshua: *JAVA - Expedition ins Programmierreich*. O´Reilly, 1 Auflage, 1997.

[omg91] Group, Object Management: *The Common Object Request Broker, Architecture and Specification, OMG und X/Open 1991/2*. OMG-Dokument 91.12.1, 1991/92.

[omgsec96] 06-20, OMG Document Numer: orb/96: *Common Secure Interoperability*. OMG, 1996.

[Ragg 96] Raggett, D.: *RFC 1942: HTML Tables*. RFC, IETF, Mai 1996.

[RDK+ 97] Rekhter, Y., B. Davie, D. Katz, E. Rosen und G. Swallow: *RFC 2105: Cisco Systems' Tag Switching Architecture Overview*. RFC, IETF, Februar 1997.

[ReKa 96] Rekhter, Y. und D. Kandlur: *RFC 1937: "Local/Remote" Forwarding Decision in Switched Data Link Subnetworks*. RFC, IETF, Mai 1996.

[Sieg 96] Siegel, Jon: *CORBA Fundamentals and Programming*. John Wiley & Sons, Inc., 1996.

[sieJan98] Siemens: *Telephony Internet Server FDB 4*, January 1998.

[sieMar98] Siemens: *Telephony Internet Server FDB 5*, March 1998.

[Sole 95] Soley, Richard Mark (Herausgeber): *Object Management Architecture Guide*. John Wiley & Sons, Inc., Dritte Auflage, Juni 1995.

[SoMa 94] Sollins, K. und L. Masinter: *RFC 1737: Functional Requirements for Uniform Resource Names*. RFC, IETF, Dezember 1994.

[ssl96] Alan O. Freier, Philip Karlton, Paul C. Kocher: *The SSL Protocol*. Netscape Communications Corporation, 3. Auflage, March 1996. Expires 9/96.

[Stal 93] STALLINGS, WILLIAM: *SNMP, SNMPv2, and CMIP: The Practical Guide to Network Management Standards.* Addison-Wesley, 1993.

[Tane 97] TANENBAUM, ANDREW S.: *Computernetzwerke.* Prentice Hall, Inc., 1997.

[w3c97] RAGGETT, DAVE: *HTML 3.2 Reference Specification*, January 1997.

[waldo98] WALDO, JIM: *JINI Architecture Overview.* Sun Microsystems Inc., 1998.

[YNAD 97] YERGEAU, F., G. NICOL, G. ADAMS und M. DUERST: *RFC 2070: Internationalization of the Hypertext Markup Language.* RFC, IETF, Januar 1997.