INSTITUT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Diplomarbeit

Entwurf und Implementierung eines Konzepts zur Anbindung von Object Request Brokern an eine Netzmanagementplattform

Bearbeiter : Thorsten Vogs

Aufgabensteller : Prof. Dr. Heinz–Gerd Hegering

Betreuer : Dr. Ulf Hollberg (IBM ENC)
Alexander Keller
Dr. Bernhard Neumair

Abgabetermin : 15. Februar 1996

Zusammenfassung

Für das Management von verteilten Systemen gibt es derzeit zwei standardisierte Architekturmodelle, das Internet-Management der IETF und das OSI-Management der ISO. Ein neuer Ansatz ist die Object Management Architecture (OMA) der OMG. Die OMA ist eine Architektur für objektorientierte verteilte Anwendungen, deren Kommunikationsmodell durch CORBA (Common Object Request Broker Architecture) definiert ist. Die zunehmende Bedeutung der OMA für das Systemmanagement zeigt sich sowohl an den Standardisierungsaktivitäten diverser internationaler Gremien, als auch an den Aussagen der Hersteller von Managementsoftware.
Es ist zumindest mittelfristig unwahrscheinlich, daß sich eine der drei Architekturen gegenüber den anderen beiden völlig durchsetzen wird. Dazu sind die Anforderungen im Management und die Stärken und Schwächen der Architekturen zu unterschiedlich. Um alle Ressourcen eines verteilten Systems überwachen und steuern zu können, muß daher die Forderung nach einer integrierten Managementplattform gestellt werden, die als Basiswerkzeug innerhalb der verschiedenen Managementarchitekturen eingesetzt werden kann. Eine solche Plattform ist derzeit nicht kommerziell erhältlich.
In dieser Arbeit wird ein Konzept vorgestellt, mit dem CORBA-konforme Object Request Broker in eine herkömmliche (für Internet- und OSI-Management entwickelte) Managementplattform integriert werden können. Dadurch können die Basisdienste der Plattform für das Management von durch CORBA-Objekte repräsentierten Ressourcen genutzt werden. Systemmanagement in einer CORBA-Umgebung wird somit ermöglicht.
Die Managementplattform kann Ereignismeldungen, die von CORBA-Objekten (z.B. in Fehlersituationen) erzeugt werden, empfangen und verarbeiten. Die Verarbeitung umfaßt das Speichern in Log-Dateien und das Anzeigen an der Benutzeroberfläche. Dadurch werden Ereignisse innerhalb der überwachten CORBA-Umgebung registriert. Außerdem besteht die Möglichkeit, Ereignismeldungen nach bestimmten Kriterien zu filtern und an Managementanwendungen weiterzuleiten, wodurch die Voraussetzung für die Automatisierung des Ereignismanagements geschaffen wird.
Es wird eine Anwendung entwickelt, die Information über die vorhandenen Ressourcen sammelt und diese Managementinformation in der Datenbank der Plattform speichert. Darauf aufbauend zeigt die Plattform ein grafisches Modell der CORBA-Umgebung an der Benutzeroberfläche an. Die Darstellung visualisiert sowohl die Ressourcen, als auch die Beziehungen zwischen ihnen und ihren aktuellen Zustand. Anhand des Modells kann der Benutzer der Managementplattform die Ressourcen überwachen. Zur Aktualisierung der gespeicherten Managementinformation wird ein ereignisgesteuerter Ansatz verwendet.

Inhaltsverzeichnis

Abbildungsverzeichnis

Teil I

Grundlagen

Kapitel 1

Einleitung

Dienste für die inner- und zwischenbetriebliche Kommunikation sind heutzutage für alle großen Unternehmen ein wichtiges Erfolgskriterium. Den Kommunikationsnetzen und Systemen, durch die diese Dienste erbracht werden, kommt eine wichtige Bedeutung zu. Die Aufgabe des Netz- und Systemmanagements liegt in der Stabilisierung dieser Informationsinfrastruktur. Management umfaßt sämtliche Vorkehrungen zur Sicherstellung des effektiven und effizienten Einsatzes eines Rechnernetzes bzw. eines verteilten Systems.
Ein Rahmenwerk für managementrelevante Standards bezeichnet man als Managementarchitektur. Die bekanntesten Managementarchitekturen wurden von der ISO und CCITT/ITU-T (OSI-Management, Telecommunications Management Network) und von der IETF (Internet-Management) entwickelt. Hierbei sind die beiden erstgenannten Architekturen im Bereich der Telekommunikationsnetze dominant und letztere im Bereich der privaten Datennetze. Allerdings befassen sich heute weitaus mehr Gremien mit der Fragestellung, geeignete Managementarchitekturen bzw. Teillösungen zu finden. Beispiele hierfür sind die Open Software Foundation (Distributed Management Environment DME), X/Open (XMP/XOM [X/OPEN 94]), Desktop Management Task Force (DMTF) oder das Network Management Forum (Omnipoints).
Ein weiterer Ansatz, der derzeit zunehmend an Bedeutung gewinnt, ist die Object Management Architecture (OMA) der Object Management Group (OMG). Es handelt sich dabei um eine Architektur für die Kommunikation und Kooperation zwischen Objekten in einer heterogenen vernetzten Umgebung. Das Ziel der OMA ist die Unterstützung allgemeiner verteilter Anwendungen. Das Kommunikationsmodell wird durch die Common Object Request Broker Architecture (CORBA) definiert. Ein Object Request Broker (ORB) stellt dabei Dienste für eine symmetrische Kooperation der verteilten Objekte bereit. Der Ansatz der OMG ist in letzter Zeit auch für das Management aktuell geworden. Dies zeigt sich sowohl in den Forschungsaktivitäten verschiedener Gremien (z.B. X/Open Sysman [X/OPEN 95b] und JIDM [X/OPEN 95a]), als auch bei der Entwicklung von Managementprodukten (z.B. Tivoli TME).
Managementplattformen stellen eine Ablauf- und Entwicklungsumgebung für Anwendungen aus den verschiedenen Funktionsbereichen des Managements dar. Der Aufbau einer Plattform gliedert sich in die Benutzerschnittstelle, Managementanwendungen, Basisanwendungen (wie z.B. Ereignismanagement und Topologiemanagement) und Infrastruktur (Kommunikations- und Informationsbaustein).
Beispiele für Managementplattformen sind IBM Netview für AIX, HP Open View, Cabletron Spectrum, SunNet Manager und Tivoli TME.

Die meisten dieser Produkte verwenden überwiegend SNMP als Managementprotokoll. SNMP (Simple Network Management Protocol) wurde 1988 von der Internet Engineering Task Force (IETF) entwickelt. Das Protokoll und die darauf basierende Architektur des Internet-Management war ursprünglich nur als kurzfristige Zwischenlösung gedacht. Langfristig wollte die IETF die OSI-Management Ansätze einsetzen. Allerdings wurde dieser Plan mittlerweile zugunsten der Standardisierung von SNMPv2 aufgegeben (vgl. auch Anhang A). SNMP-basiertes Internet-Management ist weniger komplex aber auch weniger leistungsfähig als OSI-Management.
OSI-basierte Managementplattformen sind erst seit relativ kurzer Zeit auf breiter Basis kommerziell erhältlich (z.B. TMN Workbench und Support Facility von IBM). Sie werden vor allem für das Management von öffentlichen Daten- und Telefonnetzen eingesetzt. Im Bereich der privaten Datennetze konnten sie sich bisher noch nicht durchsetzen.
Die Zahl der Managementplattformen, die auf CORBA-Technologie basieren, ist noch geringer. Hier ist an erster Stelle die TME-Plattform (Tivoli Management Environment [TIVOLI 94]) zu nennen. Es handelt sich dabei jedoch mehr um ein Spezialwerkzeug für das Systemmanagement als um eine Netzmanagementplattform. Allerdings ist zu bemerken, daß die meisten großen Hersteller von Managementsoftware bereits CORBA-basierte Produkte angekündigt haben.

1.1 Problematik

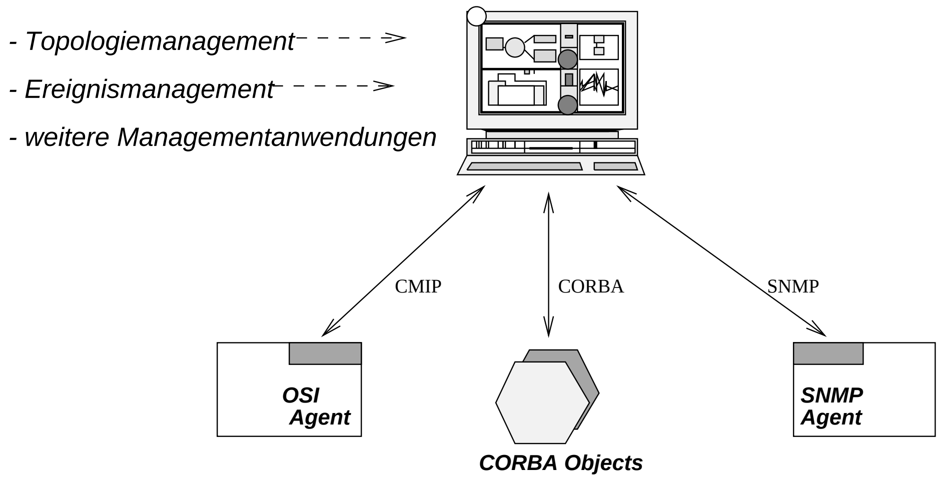

Abbildung 1.1: Eine integrierte Managementplattform

Es beginnt sich abzuzeichnen, daß die OMA neben den „traditionellen“ Architekturen des Internet- und OSI-Management vor allem im System- und Anwendungsmanagement eine Rolle spielen wird. Zumindest mittelfristig ist es unwahrscheinlich, daß sich eine der drei Architekturen gegenüber den anderen beiden völlig durchsetzen wird. Dazu sind die Anforderungen im Management und die Stärken und Schwächen der Architekturen zu unterschiedlich.
Es besteht daher das Problem, daß die Integration und die Interoperabilität der Architekturen ermöglicht werden muß, um das Management eines verteilten Systems mit vertretbarer Komplexität durchführen zu können. Insbesondere muß die Forderung nach einer integrierten

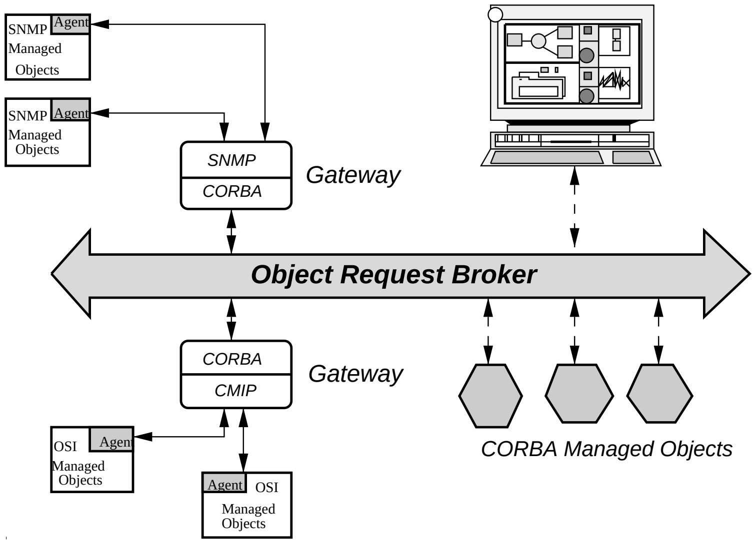

Abbildung 1.2: Interoperabilität zwischen Managementarchitekturen

Managementplattform gestellt werden, die als Basiswerkzeug für Internet-, OSI- und OMA-Management eingesetzt werden kann.
Das in dieser Arbeit entwickelte Konzept liefert die Grundlage für ein integriertes Management aller Ressourcen eines verteilten Systems, indem es die Anbindung eines CORBA-konformen Object Request Brokers an eine für Internet- und OSI-Management entwickelte Plattform ermöglicht. Dadurch können von der Plattform aus Ressourcen überwacht und gesteuert werden, die durch Internet-, OSI- oder OMA-Managementobjekte repräsentiert werden (vgl. Abb. 1.1).
Als nächster Schritt zur Lösung der Problematik kann die Schaffung von Übergängen zwischen den Managementarchitekturen angesehen werden, um die architekturspezifischen Unterschiede für Anwendungen und Benutzer möglichst transparent zu machen. Dies ist das Ziel eines größeren Projekts, in das die vorliegende Arbeit integriert ist (vgl. [KENE S.9]): Von der durch diese Arbeit entstandenen OMA-konformen Plattform sollen sowohl OMA-Objekte, als auch OSI- und SNMP-Managementobjekte überwacht und gesteuert werden. Auf letztere wird dabei mit Hilfe von CORBA/SNMP- bzw. CORBA/CMIP-Gateways zugegriffen (vgl. Abb. 1.2). Die Umsetzung der Informationsmodelle soll durch Spezifikationscompiler erfolgen, die GDMO/ASN.1 nach OMG-IDL (Interface Definition Language - vgl. Kapitel 2) überführen (die Grundlage dafür bilden u.a. die Arbeiten von XoJIDM (vgl. auch Kapitel 2)).

1.2 Vorstellung der Fragestellung

Um die Anbindung eines Object Request Brokers an eine Netzmanagementplattform zu ermöglichen, wurden folgende Fragestellungen untersucht:

1. Wie können die Basisdienste der Managementplattform (Topologie- und Ereignismanagement) für durch OMA-Managementobjekte repräsentierte Ressourcen eingesetzt werden ?

2. Welche managementrelevante Information muß dazu im Informationsbaustein der Plattform vorhanden sein ?

3. Wie ist die Kommunikationsschnittstelle zwischen Managementanwendungen, die auf der Grundfunktionalität der Plattform aufsetzen, und OMA-Managementobjekten sinnvoll zu gestalten ?

Der Schwerpunkt lag dabei auf der Nutzbarmachung der Plattformdienste (Topologiedarstellung incl. Zustandsüberwachung und Verarbeitung von Ereignismeldungen) für das Management von CORBA-Objekten [1].
Für das Topologiemanagement wurden neben einer Discovery-Anwendung und einer Kapselung des Datenbank-APIs der Plattform durch CORBA-Objektklassen Komponenten entwickelt, die eine Aktualisierung der Topologieinformation durch Ereignismeldungen ermöglichen.
Auf der Ebene der Infrastruktur der Managementplattform wurde die Integration dadurch erzielt, daß die Information der CORBA-Discovery in der Plattformdatenbank gespeichert wurde, in der auch die Information anderer Discovery-Anwendungen abgelegt wird (IP, ggf. TMN). Die Datenbank wurde dadurch zu einer integrierten Basis für Managementinformation erweitert.
Für die Verarbeitung von CORBA-Ereignismeldungen durch die Managementplattform wurde ein Konzept entwickelt, das auf dem CORBA-Event-Service basiert. Es wurden Objektklassen entworfen, deren Instanzen Ereignismeldungen in SNMP-Traps umwandeln, damit sie von der Plattform empfangen werden können. Zusätzlich wurde eine Schnittstelle entwickelt, über die CORBA-basierte Managementanwendungen sich für den Empfang von Ereignismeldungen, die durch die Plattform gefiltert wurden, registrieren können. Um ein Fehlermanagement für CORBA-Objekte zu ermöglichen, wurde ein Algorithmus für das Erzeugen von Ereignismeldungen aufgrund von Exceptions erstellt.
Das entworfene Konzept wurde mit der Managementplattform NetView für AIX und dem Object Request Broker DSOM der Firma IBM implementiert.

1.3 Gliederung der Arbeit

Die Arbeit gliedert sich in zwei Teile und den Anhang. Der erste Teil (Kapitel 1 bis 3) behandelt die Grundlagen. In Kapitel 2 wird die Technologie der OMG beschrieben. Es enthält neben einem allgemeinen Überblick über die Architektur und einer Beschreibung des für die Implementierung verwendeten Produkts SOMObjects einen Abschnitt über den derzeitigen Forschungsstand beim Einsatz dieser Technologie für das Management. Kapitel 3 beschreibt am Beispiel von NetView für AIX den Aufbau und die Architektur von Managementplattformen.
Der zweite Teil bildet den Hauptteil der Arbeit. Hier wird das entwickelte Konzept zur Anbindung des Object Request Brokers an die Managementplattform vorgestellt. Kapitel 4 definiert die allgemeine Vorgehensweise, wobei unterschiedliche Alternativen bewertet werden. Der Schwerpunkt des zweiten Teils sind die Kapitel 5 und 6.
In Kapitel 5 wird das Konzept für die Integration von CORBA-Ereignismeldungen in die Event-Management-Dienste der Plattform dargestellt. Kapitel 6 beschreibt die entwickelten Mechanismen zur Gewinnung und Präsentation von OMA-Managementinformation durch

[1] OMA-Objekte werden im folgenden auch als CORBA-Objekte bezeichnet.

die Plattform (Topologiemanagement). Kapitel 7 faßt die Ergebnisse der Arbeit zusammen und nennt Ansätze für weiterreichende Managementanwendungen auf Basis der entwickelten Komponenten.
Der Anhang enthält Information, die für das Verständnis der Arbeit nicht unbedingt notwendig ist, die dem interessierten Leser aber zur weiteren Beschäftigung mit der Thematik dienen kann. Zusätzlich sind die CORBA-IDL-Definitionen der implementierten Komponenten enthalten.

Kapitel 2

Die Object Management Architecture der OMG

2.1 Object Management Architecture (OMA)

Die Arbeit der Object Management Group (OMG) beschäftigt sich mit objektorientierten verteilten Systemen. Die OMG ist ein herstellerübergreifendes Konsortium mit über 500 Mitgliedern. Sie wurde 1989 mit dem Ziel gegründet, Standards für verteilte objektorientierte Anwendungen zu entwickeln. Hierbei stehen vor allem drei Anforderungen im Vordergrund der Bestrebungen: Interoperabilität, Wiederverwendbarkeit und Portabilität der Software.
Die Basisarchitektur für diese Ziele wurde im Object Management Architecture Guide ([OMG 92]) veröffentlicht. Die OMA hat folgende Bestandteile :

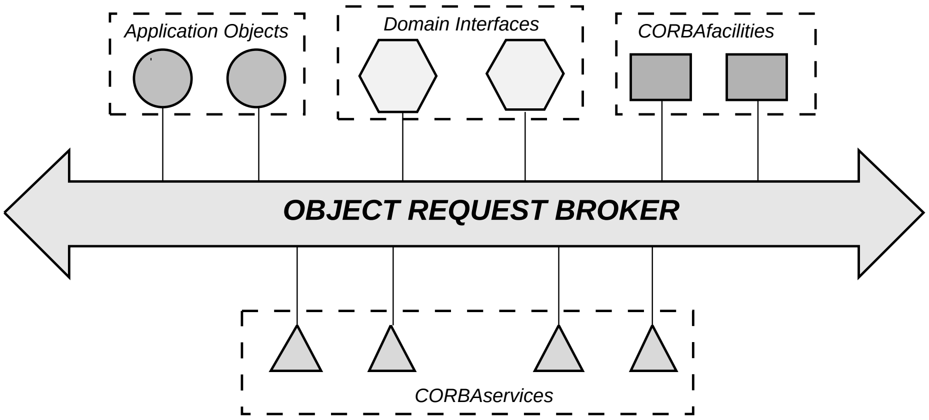

Abbildung 2.1: Die Object Management Architecture der OMG

1. Der *Object Request Broker (ORB)* ist ein Kommunikationsmechanismus für verteilte Objekte. Er ermöglicht flexible Client-Server-Beziehungen zwischen Objekten, indem er Methodenaufrufe von Clients zu Server-Objekten vermittelt und dafür sorgt, daß die Ergebnisse zum Aufrufer zurückgeleitet werden. Die Architektur und Funktion des

ORBs ist in der *Common Object Request Broker Architecture (CORBA)* ([OMG 93a]) festgelegt. CORBA wird im nächsten Abschnitt ausführlicher behandelt.

2. *CORBAservices* sind Dienste, die von speziell dafür bestimmten Objekten erbracht werden. Sie erweitern die Basisfunktionalität des ORBs und ermöglichen die Interoperabilität von verteilten Objekten auf der Anwendungsebene. Der relevante Standard dafür ist die *Common Object Services Specification [OMG 94]*. Momentan sind folgende Dienste standardisiert:

   - Der *Lifecycle Service* definiert, wie Objekte erzeugt, kopiert, bewegt und gelöscht werden können.
   - Der *Persistence Service* ermöglicht das persistente Speichern von Objekten. Dafür können z.B. objektorientierte Datenbanken, relationale Datenbanken und einfache Dateien verwendet werden.
   - Der *Naming Service* dient zur Zuweisung von Namen zu Objekten. Dabei kann auf existierenden Verzeichnisdiensten wie ISO X.500 aufgesetzt werden.
   - Der *Event Service* erlaubt das Erzeugen und Versenden von Nachrichten zwischen Objekten.
   - Der *Concurrency Service* regelt den gleichzeitigen Zugriff auf Objekte durch mehrere Clients.
   - Der *Transaction Service* stellt eine Möglichkeit dar, Transaktionen auf Objekten durchzuführen unter Berücksichtigung der ACID Eigenschaften.
   - Mit dem *Relationship Service* können Assoziationen und Beziehungen zwischen Objekten definiert werden.
   - Der *Externalization Service* behandelt die externe (in einem übertragbaren Format) und interne Darstellung von Objekten.

   Weitere Dienste, wie z.B. Security, Time und Trader befinden sich zur Zeit in der Standardisierungsphase.

3. *CORBAfacilities* sind Dienste, die für viele unterschiedliche Anwendungen nützliche Funktionalität zur Verfügung stellen. Sie werden in zwei Bereiche unterteilt:

   - *Horizontal Facilities* erstrecken sich über mehrere Anwendungsdomänen. Sie werden untergliedert in:
     - *User Interface* definiert den Zugriff auf ein grafisch orientiertes Informationssystem.
     - *Information Management* spezifiziert die Modellierung, Definition, Speicherung, Wiedergewinnung, Verwaltung und Austausch von Information. Hierzu gehören z.B. Mechanismen zum Speichern von Compound Documents.
     - *Systems Management* Services erlauben das Management von verteilten Objektsystemen.
     - *Task Management* ermöglicht die Automatisierung von Tasks in Form von Benutzer- und Systemprozessen.

- *Vertical Facilities* (auch Domain Interfaces genannt) stellen Basisfunktionalität für unterschiedliche Anwendungsdomänen wie z.B. CAD- oder Finanzanwendungen bereit.

Der Unterschied zwischen den CORBAservices und den CORBAfacilities besteht darin, daß letztere anwendungsnäher und spezifisch für bestimmte Anwendungsbereiche sind, während erstere eher eine Erweiterung der Infrastruktur des ORB-Systems darstellen. Die CORBAfacilities befinden sich erst am Anfang der Standardisierungsphase.

4. *Application Objects* bilden die eigentlichen Anwendungen. Sie verwenden die Dienste, die vom ORB, den CORBAservices und den CORBAfacilities zur Verfügung gestellt werden.

Folgende Ziele sollen durch die OMA verwirklicht werden (vgl. [OMG 92] S. 13 ff.):

- Interoperabilität von Anwendungen, die unabhängig voneinander entwickelt wurden.
- Einbindung bereits existierender Anwendungen.
- Wiederverwendbarkeit von existierenden Softwarekomponenten und damit geringere Kosten und weniger Zeitbedarf bei der Enwicklung von neuen Anwendungen.
- Transparenz in Bezug auf Programmiersprachen, Betriebssysteme und Netztechnologien.
- Allgemeine Dienste, die von unterschiedlichen Anwendungen in Anspruch genommen werden können.
- Möglichkeit der Weiterentwicklung und Erweiterung bestehender Anwendungen.

2.2 CORBA

In der *Common Object Request Broker Architecture (CORBA)* werden der Aufbau, die Funktionalität und die Schnittstellen eines ORBs definiert. Damit wird das Kommunikationsmodell der OMA festgelegt. In Abbildung 2.2 ist der Aufbau eines CORBA-konformen ORBs dargestellt.

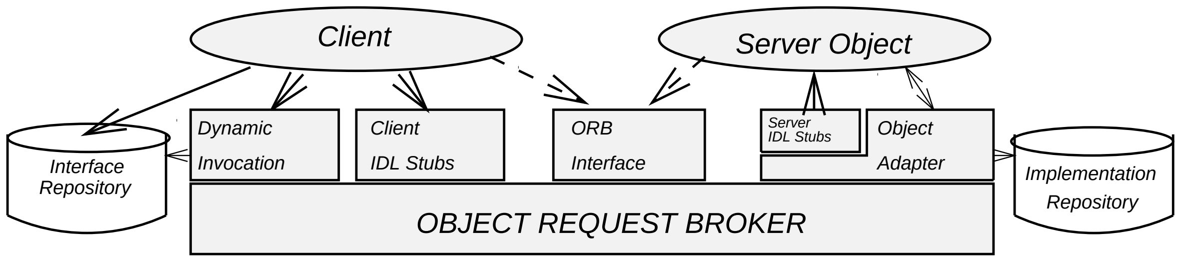

Abbildung 2.2: Die Struktur eines CORBA-konformen ORBs

Der Schwerpunkt des CORBA-Standards liegt in der Beschreibung der Schnittstellen, die ein ORB zur Verfügung stellt. Es werden sowohl die Aufrufschnittstellen beschrieben, über die

Clients auf Objekte zugreifen können, als auch die Schnittstellen auf den Kern des ORBs (*ORB-Interface*).
Jedem Objekt wird durch den ORB ein eindeutiger Identifikator in Form einer Objektreferenz zugewiesen. Anhand dieser Objektreferenz kann der ORB ein Objekt lokalisieren.
Die Beschreibung der Aufrufschnittstellen von Objekten erfolgt in einer eigens dafür definierten Sprache. Diese wird als *Interface Definition Language (IDL)* bezeichnet. Mit dieser C-ähnlichen Sprache wird die Vererbung von Schnittstellenbeschreibungen unterstützt. Die Definition der Syntax und Semantik der IDL sowie ihre Abbildung auf die Programmiersprache C *(Language Mapping)* ist einer der Schwerpunkte der CORBA1.1-Spezifikation. Inzwischen sind zusätzliche Sprachabbildungen für C++ und Smalltalk definiert worden.
Der ORB hat zwei unterschiedliche Schnittstellen für Methodenaufrufe von Clients auf Objekten:

1. *Statische Aufrufschnittstelle:* Hier wird bei der Entwicklung des Client-Programms ein Client-Stub aus der IDL-Beschreibung des Objekts erzeugt und zum Client-Programm dazugebunden.

2. *Dynamische Aufrufschnittstelle:* Clients können zur Laufzeit Aufträge für Objekte zusammensetzen und dem ORB übergeben. Der Client muß dabei die Objektreferenz des adressierten Objekts, die aufzurufende Methode sowie die Parameter für den Methodenaufruf angeben. Als Informationsquelle für die Konstruktion von dynamischen Methodenaufrufen dient das Interface Repository, in dem die IDL-Beschreibung aller Objektklassen gespeichert ist.

Für ein Objekt ist transparent, über welche Schnittstelle eine Methode aufgerufen wird. Ein Aufruf wird über den Object Adapter und das IDL-Skeleton, das den Server-Stub darstellt, übergeben. Der Client muß nicht wissen, in welcher Programmiersprache ein Objekt implementiert ist, auf welchem Netzknoten es sich befindet und ob es zum Zeitpunkt des Aufrufs bereits aktiv ist, oder erst aktiviert werden muß. Dem Client muß nur die Aufrufschnittstelle des Objekts bekannt sein. Dazu dient der Client-Stub bei der statischen und das Interface Repository bei der dynamischen Aufrufschnittstelle.
Der Object Adapter agiert als Bindeglied zwischen dem ORB und den Objekten. Er erfüllt folgende Aufgaben (vgl. [OMG 93a] S. 152):

- Aktivierung von Objekten, die zum Zeitpunkt eines Methodenaufrufs inaktiv sind.

- Generierung von Objektreferenzen.

- Authentisierung von Client-Aufträgen.

- Weiterleitung von Methodenaufrufen an Objekte über IDL-Skeletons.

Diese Funktionalität wird vom sogenannten *Basic Object Adapter (BOA)* erbracht, den jedes CORBA-konforme Produkt implementieren muß. Darüber hinaus sind jedoch auch andere Object Adapter für spezielle Anforderungen möglich.
Über die ORB-Schnittstelle können Clients und Objekte direkt allgemeine Dienste vom ORB nachfragen. Zu diesen Diensten gehört z.B. die Umwandlung von Objektreferenzen in Strings und wieder zurück sowie andere allgemeine Operationen auf Objektreferenzen.

Im *Implementation Repository* befindet sich die Information, die der ORB braucht, um Objekte zu lokalisieren und zu aktivieren. Außerdem können dort weitere implementierungsspezifische Daten, wie z.B. Debugging-Information oder Umgebungsvoraussetzungen, abgespeichert werden. Die CORBA-Spezifikation macht hierzu keine konkreten Vorschriften.
Das *Interface Repository* enthält die IDL-Beschreibungen der Objekte. Über Programmierschnittstellen wird diese Information zur Laufzeit verfügbar gemacht. Die IDL-Beschreibungen werden u.a. zur Konstruktion von dynamischen Methodenaufrufen und zur Typüberwachung von Parametern benötigt.
Die CORBA-Spezifikationen 1.1 und 1.2 dienen als Grundlage für zahlreiche Produktimplementierungen wie z.B. Ionas Orbix, Suns DOE (mittlerweile NEO - *Network Enabled Objects*), Hewlett-Packards Distributed Smalltalk und IBMs SOM/DSOM, das in dieser Diplomarbeit verwendet wurde.
Der größte Mangel von CORBA 1.x ist, daß kein Protokoll für die Interoperabilität zwischen den ORBs verschiedener Hersteller definiert wurde. Da fast jedes CORBA-Produkt eine eigene Implementierungsstrategie verfolgt, können heutige Produkte im allgemeinen nicht zusammenarbeiten. Dieses Problem wurde mit CORBA Version 2.0 [OMG 95] gelöst. Darin wird die Interoperabilität zwischen ORBs über Gateways (sogenannte *Bridges*) und Interoperability-Protokolle wie das GIOP (General Inter ORB Protocol) standardisiert. Damit kann ein ORB auf Objekte zugreifen, die von einem anderen ORB verwaltet werden.
Der nächste Abschnitt beschreibt das in der Diplomarbeit verwendete Produkt SOMObjects von IBM und geht darauf ein, wie die CORBA-Spezifikation dabei implementiert wurde.

2.3 IBM's SOMObjects

2.3.1 SOM: System Object Model

Das System Object Model (SOM) ist eine objektorientierte Technologie zur Erstellung von binären Klassenbibliotheken. Es wurde entwickelt, um eine Reihe von Problemen zu lösen, die bei heutigen objektorientierten Programmiersprachen wie C++ auftreten. Einige dieser Probleme sind:

- Eine Klassenbibliothek, die mit einem bestimmten Compiler übersetzt wurde, kann in der Regel nicht mit Code, der von einem anderen Compiler erzeugt wurde, gebunden werden. Der Grund dafür ist die Inkompatibilität der von verschiedenen Compilern erzeugten internen Daten.

- Objekte, die in einer Programmiersprache implementiert wurden, können nicht von Programmen in einer anderen Sprache verwendet werden. Dies ist insbesondere ein Hindernis für die Entwicklung von wiederverwendbaren Softwarekomponenten und Objekt-Frameworks.

- Häufig müssen Anwendungen neu compiliert werden, wenn sich die Implementierung der verwendeten Klassenbibliotheken ändert, was z.B. der Fall sein kann, wenn ein neues Release der Bibliothek eingesetzt wird.

SOMObjects löst diese Probleme durch ein sprachneutrales Objektmodell, das die Entwicklung von binärkompatiblen Klassenbibliotheken erlaubt. Dabei werden ausgehend von einer

IDL-Beschreibung einer Objektklasse, die dem CORBA-Standard entspricht, Implementierungstemplates für Programmiersprachen generiert.
Die Implementierung einer normalen SOM-Anwendung (deren Objekte nicht verteilt sind) geschieht durch folgende Schritte:

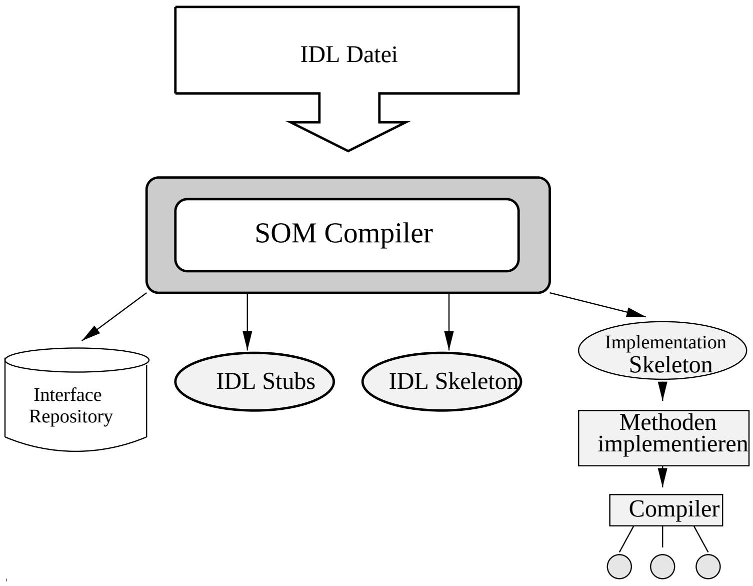

Abbildung 2.3: Der SOM-Entwicklungsprozeß

1. Zuerst werden die Aufrufschnittstellen (die Methoden und die sichtbaren Attribute) für die Objektklassen in IDL definiert.

2. Der SOM-Compiler erzeugt aus den IDL-Dateien Skelette für die Implementierung der Klassen sowie Client- und Server-Stubs in Form von Header-Dateien. Der Compiler kann konfiguriert werden, um Implementierungstemplates für unterschiedliche Programmiersprachen zu erzeugen.

3. Die Methoden in den vom SOM-Compiler erzeugten Skeletten werden durch den Anwendungsprogrammierer implementiert. Dabei können für die einzelnen Klassen unterschiedliche Programmiersprachen verwendet werden.

4. Das Client-Programm, das die Klassen benutzt, wird erstellt.

5. Das Client-Programm wird mit einem Compiler der entsprechenden Programmiersprache übersetzt und die Implementierung der Klassen wird dazugebunden.

6. Das Client-Programm wird ausgeführt.

SOM ist derzeit für AIX, OS/2, Windows und MVS verfügbar.

Die SOM-Laufzeitumgebung

Die SOM-Laufzeitumgebung dient zur Verwaltung der Objekte und als Ausgangspunkt für die Entwicklung von Objektklassen. Sie besteht aus drei Klassen: *SOMObject, SOMClass* und *SOMClassMgr.*

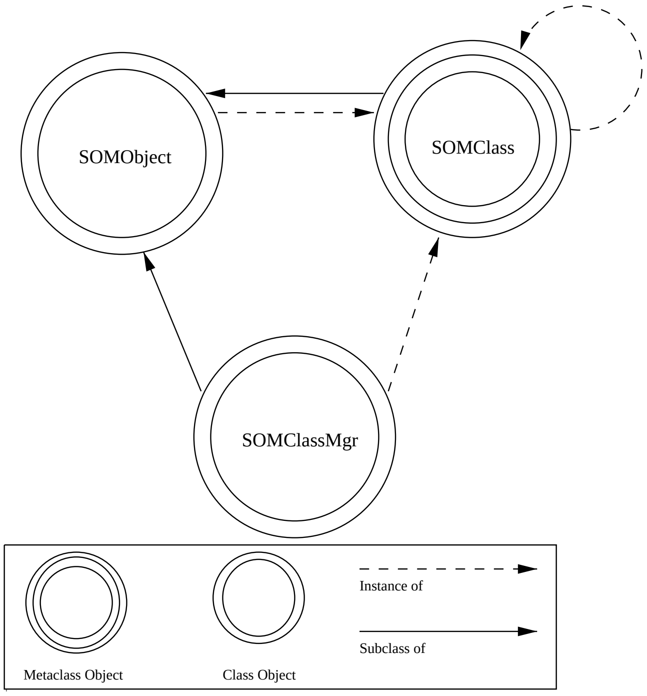

Abbildung 2.4: Die Klassen der SOM-Laufzeitumgebung

Von SOMObject sind alle Objektklassen abgeleitet. Diese Klasse definiert das grundlegende Verhalten aller SOM-Objekte.
Bei SOM werden Klassen zur Laufzeit ihrerseits als Objekte implementiert. Da jede Klasse ein Objekt ist, muß auch SOMObject Instanz einer Klasse sein. Diese Klasse ist SOMClass. SOMClass ist die Wurzel aller *Metaklassen.* Über die Methoden und Attribute der Metaklassen kann zur Laufzeit Information über Klassen gewonnen werden, oder es können z.B. bestimmte Konstruktormethoden für die Objekte einer Klasse definiert werden.
Die dritte Basisklasse der SOM-Laufzeitumgebung ist *SOMClassMgr.* Ein Objekt dieser Klasse wird zur Laufzeit automatisch erzeugt. Dieses *SOMClassMgrObject* erledigt das dynamische Laden der Klassenbibliotheken und DLLs *(Dynamic Linked Libraries)* und registriert alle vorhandenen Klassen.
Die Beziehung zwischen den Klassen der SOM-Laufzeitumgebung ist in Abbildung 2.4 dargestellt.
SOM wurde ursprünglich nicht als CORBA-Implementierung konzipiert, sondern als Framework für die Entwicklung von objektorientierten Anwendungen, wobei der Verteilungsaspekt nicht im Vordergrund stand. Mit DSOM (*Distributed System Object Model*) enthält SOM aber einen CORBA1.1-konformen Object Request Broker.

2.3.2 Die CORBA-Implementierung DSOM

DSOM ermöglicht Interprozessaufrufe zwischen SOM-Objekten, die sich auf der gleichen Maschine *(Workstation DSOM)* oder auf unterschiedlichen Rechnern im Netz befinden *(Workgroup DSOM)*. Die Kommunikation wird über IPC-Mechanismen bzw. LAN-Protokolle (TCP/IP, Netbios und IPX/SPX) realisiert. Durch Ableitung aus der Klasse *SOMSockets* kann die Gruppe der unterstützten Protokolle erweitert werden.
Abbildung 2.5 zeigt die Basisarchitektur von DSOM.

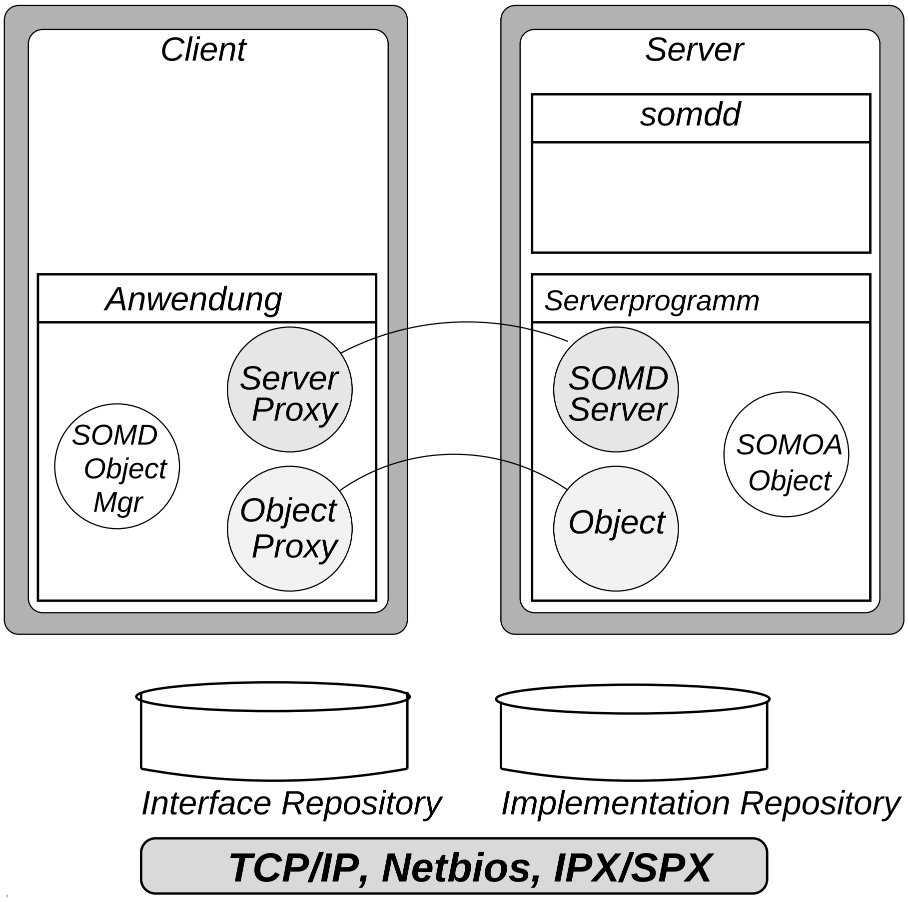

Abbildung 2.5: DSOM - Basisarchitektur

Auf der Serverseite befindet sich der DSOM-Dämonprozeß (somdd). Dieser hat die Aufgabe, Serverprozesse bei Bedarf zu starten und die Verbindungen zwischen Clients und Servern herzustellen.
Die Objekte sind auf Serverprozesse verteilt. Hierzu kann das mitgelieferte Standard-Serverprogramm somdsvr verwendet werden, mit dem SOM-Klassenbibliotheken dynamisch geladen werden können. Es können jedoch auch eigene Serverprogramme mit anwendungsspezifischer Funktionalität eingesetzt werden.
In jedem Serverprozeß gibt es ein Server-Objekt (SOMDServer), das folgende Aufgaben hat:

- Es stellt den Clients über seine Schnittstelle Dienste zum Erzeugen und Löschen von

Objekten im Serverprozeß zur Verfügung.

- Es ist zusammen mit dem Object Adapter für das Erzeugen und Verwalten von Objektreferenzen zuständig.

- Es ist an der Weiterleitung von Methodenaufrufen an Zielobjekte beteiligt.

Der SOM Object Adapter (SOMOA) ist die Schnittstelle zwischen dem Serverprogramm und der DSOM-Laufzeitumgebung. SOMOA nimmt die Aufträge der Clients entgegen und sorgt für ihre Weiterverarbeitung. Dazu gehört das Interpretieren von Objektreferenzen und das Weiterleiten der Methodenaufrufe zu den entsprechenden Zielobjekten.
Die eigentlichen Zielobjekte der Clients befinden sich innerhalb des Serverprogramms. Sie stellen die Methoden zur Verfügung, die von den Clients aufgerufen werden. Die Klassenbibliotheken der Objekte sind in der Regel in Form von Dynamic Linked Libraries (DLLs) auf den Server-Rechnern vorhanden und werden bei Bedarf zu den Serverprogrammen dazugebunden.
Innerhalb eines Client-Programms muß der DSOM Object Manager vorhanden sein. Dieser ist eine Instanz der Klasse *SOMDObjectMgr*. Der Object Manager wird automatisch beim Initialisieren der DSOM-Laufzeitumgebung erzeugt. Der Object Manager ist notwendig, um

- die Server zu finden, die die Klassen der gewünschten Zielobjekte unterstützen,

- Objekte auf Servern zu erzeugen und zu löschen,

- Objektreferenzen in Strings umzuwandeln.

Der Object Manager erbringt zusammen mit dem Server-Objekt die Grundfunktionalität für die Verwaltung der verteilten Anwendungsobjekte. Diese beiden Objekte setzen bei DSOM auf den eigentlichen ORB-Schnittstellen wie dem Object Adapter auf.
Eine weitere zentrale Rolle spielt das Konzept der Proxy-Objekte. Proxy-Objekte sind Stellvertreter für Objekte, die sich eigentlich auf einem Server befinden. Ihre Klassen werden von SOM dynamisch zur Laufzeit generiert. Jede Methode einer Proxy-Klasse wird automatisch so überschrieben, daß ein Aufruf an das entsprechende Objekt auf dem Server weitergeleitet wird. Clients brauchen Proxies sowohl für die eigentlichen Zielobjekte, als auch für das Server-Objekt, um z.B. neue Objektinstanzen zu erzeugen. Ein Proxy-Objekt entspricht einer Objektreferenz auf das Zielobjekt des Proxies.
In Entsprechung zum CORBA-Standard hat DSOM außerdem ein Interface Repository, in dem die IDL-Definitionen sämtlicher Objektklassen gespeichert sind und ein Implementation Repository, in dem die Server und die von ihnen unterstützten Klassen registriert sind.

Die Implementierung von CORBA in DSOM

DSOM Version 2.1 ist ein CORBA1.1-konformer Object Request Broker. Dieser Abschnitt behandelt, wie die Bestandteile der CORBA-Spezifikation bei DSOM implementiert wurden.

- Objektreferenzen:
  Ein beim ORB registriertes Objekt wird durch seine Objektreferenz identifiziert. Bei DSOM kann eine Objektreferenz verschiedene Formate haben, die jedoch alle das gleiche Objekt kennzeichnen:

- Eine Instanz der Klasse *SOMDObject*, die die Lokalisierungsinformation des Objekts beinhaltet.
- Ein Binärformat zum „transportieren“ der Objektreferenz (z.B. als Parameter von Methodenaufrufen).
- Eine String-Form, um die Objektreferenz z.B. in einer Datei speichern zu können.
- Ein Proxy-Objekt, auf dem Clients Methoden aufrufen können.

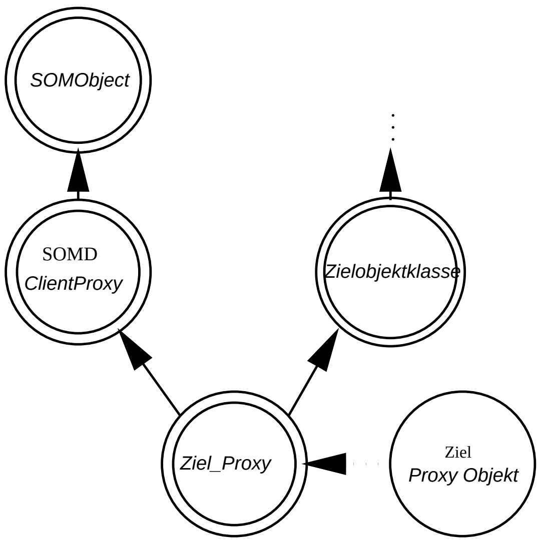

Abbildung 2.6: Erzeugen einer Proxy Klasse

Damit ein Client Methoden auf einem Zielobjekt aufrufen kann, muß er Zugriff auf ein entsprechendes Proxy-Objekt haben. Clients erhalten Proxy-Objekte als Rückgabeparameter von diversen Methodenaufrufen. Das eigentliche Erzeugen der Proxies im Adressraum des Clients erfolgt automatisch zur Laufzeit (vgl. auch Kapitel 6). Dazu muß zunächst eine Proxy-Klasse generiert werden. Dies geschieht mittels Multiple Inheritance von der Klasse *SOMDClientProxy* und der Klasse des Zielobjekts. Das Erzeugen einer Proxy-Klasse ist in Abbildung 2.6 dargestellt.

Bei der Proxy-Klasse wird jede von der Klasse des Zielobjekts geerbte Methode automatisch so überschrieben, daß ein Aufruf zu dem entfernten Zielobjekt weitergeleitet wird. Dadurch werden Methodenaufrufe, die der Client auf seinem lokalen Proxy-Objekt macht, in Wirklichkeit von dem Zielobjekt auf dem Server ausgeführt.

Zur Eindeutigkeit von Objektreferenzen ist zu sagen, daß zwischen Objekten und Referenzen eine 1:n Beziehung vorliegt. Das heißt, eine Referenz zeigt zwar immer auf genau ein Objekt, ein Objekt kann aber durch mehrere Referenzen repräsentiert werden. Es ist z.B. denkbar, daß zwei unterschiedliche Objektreferenz-Strings auf das gleiche SOM-Objekt zeigen (zur String-Form von Objektreferenzen vgl. auch Kapitel 6).

DSOM Version 2.1 ist nicht CORBA2.0-konform und ist daher nicht mit anderen CORBA-Implementierungen interoperabel. Dies bedeutet, daß DSOM-Objektreferenzen nicht von den ORBs anderer Hersteller interpretiert werden können.

- Interface Definition Language (IDL):
  Die SOM-IDL umfaßt den Sprachumfang der CORBA-IDL. Die IDL-Definition eines Objekts (Interface Declaration) besteht aus folgenden Bestandteilen:

  - *Datentypen.* Es können sowohl die Basis-IDL-Typen (string, short, long, unsigned short, unsigned long, float, double, char, boolean, octet und any) als auch zusammengesetzte Datentypen verwendet werden.
  - *Operationen.* Definitionen von Methoden, deren Parameter zusätzlich zu ihrem Typ die Kennzeichnung *in*, *out* oder *inout* haben.
  - *Attributes* sind die nach außen zugänglichen Instanzvariablen der Objektklasse.
  - *Exceptions* sind Datenstrukturen, die bei „Ausnahmefällen" innerhalb von Methodenaufrufen an den Aufrufer zurückgegeben werden.

  Man kann jedoch in einer sogenannten *Implementation Section* der IDL-Datei zusätzliche Information mit angeben, die mit der CORBA-IDL nicht spezifiziert werden kann. Beispiele dafür sind:

  - Private Instanzvariablen, die zwar von den Methoden einer Klasse verwendet werden können, auf die Clients jedoch keinen direkten Zugriff haben und die somit eigentlich nicht Bestandteil der Schnittstelle der Klasse sind.
  - *Passthru statements*, mit denen veranlaßt werden kann, daß Anweisungen wie z.B. *#include* oder *typedef*-Statements in die erzeugten Templates oder Header-Dateien eingefügt werden. Ein Anwendungsbeispiel ist das Einfügen von Typdefinitionen, die nicht in IDL-Format vorliegen, in die Implementierungstemplates.
  - Sogenannte *Modifiers* wie der Name der DLL, die die Klasse enthält, den Namen der Metaklasse der Klasse oder die Kennzeichnung von ererbten Methoden, die überschrieben werden. Der Name der DLL muß deshalb in der IDL-Datei (und somit im Interface Repository) angegeben werden, damit von der SOM-Laufzeitumgebung die DLL bei Bedarf (z.B. bei einem Request zum Erzeugen eines Objekts in einem Serverprozeß) geladen werden kann.
  - Ein wichtiger Bestandteil einer SOM-IDL Definition ist die Releaseorder der Methoden einer Klasse. Durch sie wird die Reihenfolge der Methoden in den vom Compiler erzeugten Datenstrukturen festgelegt. Durch eine konsistente Releaseorder kann erreicht werden, daß die Implementierung einer Klasse sich ändern kann, ohne daß die Client-Programme neu compiliert werden müssen.

  In CORBA 1.1 wird nur eine Sprachabbildung zwischen IDL und C definiert. SOMObjects unterstützt diese Abbildung, hat aber zusätzlich ein Language Mapping für C++.

- Client-Aufrufschnittstellen:
  DSOM unterstützt sowohl die statische als auch die dynamische Aufrufschnittstelle.

  Die statische Aufrufschnittstelle wird mit den durch den SOM-Compiler erzeugten Client-Stubs realisiert. DSOM ermöglicht jedoch auch dynamische Methodenaufrufe, die von Clients zur Laufzeit konstruiert und abgesetzt werden können (*Dynamic Invocation Interface*). Dadurch können Anwendungen Methoden auf Objekten aufrufen, ohne daß bei der Übersetzung ein Client-Stub für die Objektklasse dazugebunden wurde. Ein dynamischer Methodenaufruf besteht aus der Objektreferenz des Zielobjekts, der Methode, die aufgerufen werden soll, und den Parametern für den Methodenaufruf.

Dynamische Methodenaufrufe müssen explizit konstruiert werden, indem die Parameter für den Aufruf in eine speziell dafür vorgesehene Datenstruktur, die sogenannte NVList (Named Value List) eingetragen werden. Als Quelle für die Beschreibung der Methoden und ihrer Parameter dient das Interface Repository.

Dynamische Methodenaufrufe können entweder synchron (mit *invoke*) oder asynchron (mit *send*) ausgeführt werden.

- ORB-Schnittstelle:
  DSOM implementiert die ORB-Schnittstelle. Über sie erfolgt die Umwandlung zwischen Objektreferenzen und Strings. Außerdem dient sie zum Erzeugen von NVLists im Zusammenhang mit dem Dynamic Invocation Interface.

- Server:
  CORBA definiert vier Alternativen für die Zuordnung von Objekten zu Serverprogrammen:

  1. Ein *Shared Server* beinhaltet mehrere Objekte.
  2. Ein *Unshared Server* ist lediglich für ein einziges Objekt zuständig.
  3. Ein *Server per Method* erzeugt für jeden Methodenaufruf einen eigenen Prozeß.
  4. Ein *Persistent Server* wird im Gegensatz zu den anderen drei Möglichkeiten nicht automatisch aktiviert, wenn ein Methodenaufruf für ihn eintrifft, sondern muß explizit durch ein entsprechendes Kommando gestartet werden.

  DSOM startet ein Serverprogramm, wenn ein Client eine Verbindung zu dem entsprechenden Server aufnehmen will. Dabei kann der Anwendungsprogrammierer jeden der vier CORBA-Aktivierungsmechanismen implementieren (vgl. [IBM 94f] S. 6 - 69).

  DSOM stellt ein generisches Serverprogramm (somdsrv) zur Verfügung. Dieses registriert sich automatisch bei der DSOM-Laufzeitumgebung, wenn es gestartet wird.

  Im Implementation Repository werden alle Server und die von ihnen unterstützten Objektklassen registriert. Dazu kann entweder ein Kommando (regimpl) oder eine Programmierschnittstelle verwendet werden. Dadurch kann der ORB die durch die Server implementierten Objekte lokalisieren,(de-)aktivieren und Methodenaufrufe an sie weiterleiten. Der Aufbau des Implementation Repository wird in Kapitel 6 näher beschrieben.

- Object Adapter:
  Der Object Adapter ist die Schnittstelle zwischen den Servern und dem ORB. DSOM hat den *Basic Object Adapter* (BOA) aus der CORBA-Spezifikation implementiert. Allerdings existiert dieser nur als abstrakte Klassendefinition und kann nicht instanziiert werden. Eine Unterklasse des BOA ist der SOM Object Adapter (SOMOA). Dieser stellt neben der Möglichkeit, Server und ihre Objekte beim ORB zu registrieren (durch Erzeugen von Objektreferenzen), zusätzliche SOM-spezifische Funktionalität bereit, um Methodenaufrufe an Zielobjekte zu vermitteln. Der SOMOA ruft dabei für jeden Request die Methode *somdDispatchMethod* des SOMDServers auf, die den Methodenaufruf

an das eigentliche Zielobjekt weiterleitet. Durch Überschreiben der Methode *somdDispatchMethod* kann anwendungsspezifische Funktionalität in die Verarbeitung von Requests integriert werden (s. auch Kapitel 5).

2.3.3 Die SOM-Frameworks

SOMObjects enthält neben DSOM eine Reihe von Frameworks. Ein Framework ist eine Klassenbibliothek, die Objekte für eine bestimmte Aufgabe zur Verfügung stellt. Es wird zunächst gezeigt, inwieweit die SOM-Frameworks konform zu den von der OMG standardisierten CORBAservices sind. Anschließend werden ihre Kernbestandteile kurz erläutert.

CORBAservice	SOM-Framework	Konformität
Naming	-	-
Event	ja	nein
Persistence	ja	nein
Lifecycle	SOMDServer und SOMDObjectMgr	nein
Interface Repository	ja	ja
-	Replication	-
-	CollectionClasses	-

Die Aufführung des Interface Repository als CORBAservice ist eigentlich nicht korrekt, weil es ein integraler Bestandteil der CORBA-Architektur ist. Bei SOM wird es allerdings als eigenes Framework betrachtet. Zusammenfassend kann man sagen, daß die derzeitige Version von SOM (2.1) keinen der grundlegenden CORBAservices implementiert hat und statt dessen eigene Dienste mit teilweise ähnlicher Funktionalität bereitstellt. Das liegt darin begründet, daß die Frameworks vor den entsprechenden OMG-Standards entwickelt wurden.

Das SOM Interface Repository Framework

Das Interface Repository ist eine Datenbank, in der die IDL-Definitionen der Objektklassen gespeichert werden können. CORBA 1.2 nennt u.a. folgende Gründe für das Anlegen einer solchen Datenbank (vgl. [OMG 93] S. 128):

- Der ORB braucht die IDL-Definitionen für die Interpretation von Methodenaufrufen. Insbesondere um:
  - Parameter zu überprüfen,
  - die Korrektheit von Vererbungshierarchien sicherzustellen,
  - die Interoperabilität zwischen ORBs zu unterstützen.
- Außerdem kann das Interface Repository von Clients für verschiedene andere Aufgaben eingesetzt werden. Beispielsweise um:
  - Komponenten für eine CASE-Umgebung bereitzustellen (z.B. Interface Browser),
  - aus den Schnittstellendefinitionen Language Bindings zu erzeugen (z.B. IDL-Compiler)
- Desweiteren spielt das Interface Repository eine wichtige Rolle beim *Dynamic Invocation Interface.*

Die IDL-Definitionen der Klassen werden durch den SOM-Compiler im Interface Repository registriert. Das Interface Repository selbst ist in Form von „flat files" realisiert, die in der Umgebungsvariable SOMIR angegeben werden. Diese Dateien ergeben zusammen eine logische Datenbank. Durch entsprechendes Setzen der SOMIR-Variable kann man erreichen, daß unterschiedliche Anwendungen ihr eigenes separates Interface Repository verwenden oder verschiedene Sichten auf die Daten innerhalb eines Interface Repositories haben.
Der Zugriff auf eine Information im Interface Repository erfolgt über die Methoden der Objektklassen des Interface Repository Frameworks.

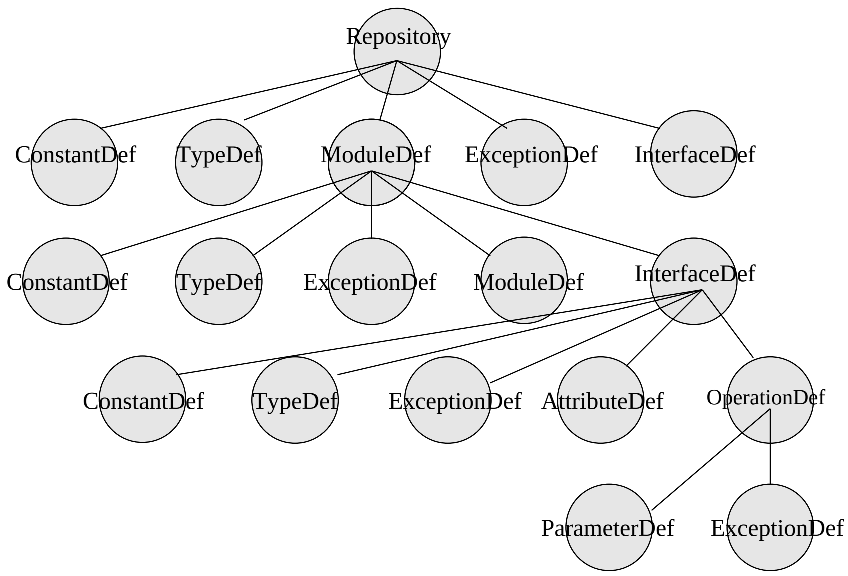

Abbildung 2.7: Die Containment-Beziehungen im Interface Repository

Die Klassen des Interface Repository Frameworks entsprechen im wesentlichen den Bestandteilen einer IDL-Quelldatei:

1. *Contained*. Alle Objekte, die im Interface Repository enthalten sind, müssen von dieser Klasse abgeleitet sein. Die Klasse hat folgende Methoden:
   - *describe* liefert die IDL-Spezifikation der (von *Contained* abgeleiteten) Klasse
   - *within* liefert alle Container-Objekte, in denen das Objekt enthalten ist
2. *Container*. Alle Objekte im Interface Repository, die andere Objekte enthalten, müssen von dieser Klasse abgeleitet sein. Die Klasse hat die Methoden:
   - *contents* liefert den „Inhalt" (d.h. die enthaltenen Objekte) des Containers
   - *describe_contents* beschreibt den Inhalt des Containers (die entsprechenden IDL-Definitionen)
   - *lookup_name* lokalisiert ein bestimmtes Interface-Repository-Objekt anhand seines Namens

3. *ModuleDef.* Eine Instanz dieser Klasse existiert für jedes Modul, das in einer IDL-Datei definiert ist. Ein Modul ist ein eindeutiger Namensraum für IDL-Konstrukte.
4. *InterfaceDef.* Eine InterfaceDef-Instanz existiert für jede SOM-Klasse.
5. *AttributeDef.* Eine Instanz dieser Klasse existiert für jedes Attribut einer Klasse.
6. *OperationDef.* Jede Methode einer Klasse wird durch ein OperationDef-Objekt repräsentiert.
7. *ParameterDef.* Diese Objekte geben Auskunft über die Parameter von Methodenaufrufen.
8. *TypeDef.* Eine Instanz dieser Klasse existiert für jede typedef, struct, union und enum Datenstruktur in einer IDL-Datei.
9. *ConstantDef*-Objekte repräsentieren die in IDL-Deklarationen enthaltenen Konstanten.
10. *Exception.* Ein Objekt existiert für jede definierte Ausnahme, die bei einem Methodenaufruf auftreten kann.
11. *Repository.* Eine Instanz dieser Klasse repräsentiert das gesamte Interface Repository.

Die Klassendefinition (in IDL) aller beim ORB registrierten Objekte muß im Interface Repository gespeichert sein, weil der ORB diese Information zur Laufzeit benötigt.
Die Information im Interface Repository ist jedoch nicht nur dem ORB selbst zugänglich, sondern (über die beschriebene Schnittstelle) jeder SOM-Anwendung.
Speziell für Managementanwendungen ist es wichtig, die Definition der Aufrufschnittstellen von Managementobjekten zur Laufzeit ermitteln zu können (z.B. für dynamische Methodenaufrufe). Das Interface Repository kann deshalb für CORBA-basierte Managementanwendungen die gleiche Rolle spielen, wie die häufig von OSI-Plattformen zur Verfügung gestellten *Metadata*-Repositories, die GDMO-Definitionen für Anwendungen zugänglich machen (vgl. [IBM 95c], [HP]).

Das SOM Event Management Framework

Das Event Management Framework beinhaltet eine Reihe von Klassen für das Versenden und Empfangen von asynchronen Ereignismeldungen.
Eine Anwendung, die Ereignismeldungen von Objekten empfangen will, initialisiert zunächst ein Objekt der Klasse *Event Manager (EMAN)*. Anschließend registriert sie sich für alle Arten von Ereignissen, die sie empfangen will. Der Event Manager wartet daraufhin in einer Endlosschleife auf die entsprechenden Ereignismeldungen und verarbeitet diese weiter.
Es gibt verschiedene Arten von Ereignissen, die folgendermaßen kategorisiert sind:

- *Timer Events.* Ein Ereignis dieser Art wird vom Event Manager erzeugt, wenn eine vorher festgesetzte Zeit verstrichen ist. Der Zeitraum zwischen zwei Events wird bei der Registrierung durch die Anwendung angegeben.
- *Sink Events.* Ein Sink Event wird erzeugt, wenn eine Ein- oder Ausgabeaktivität auf einem definierten Event Sink festgestellt wird. Ein Event Sink kann ein Socket, ein File Descriptor oder eine Message Queue sein.

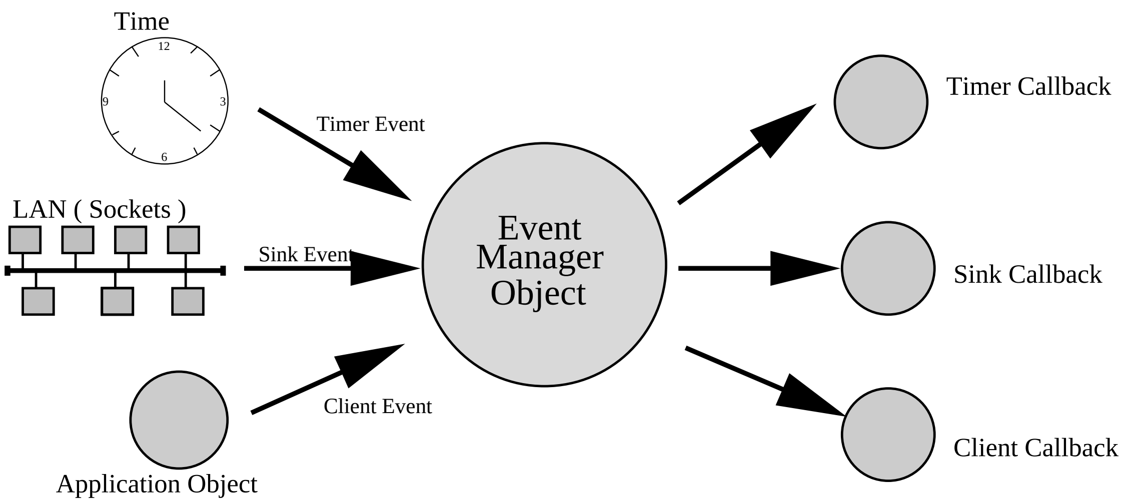

Abbildung 2.8: Das Event Management Framework

- *Client Events.* Diese Ereignismeldungen werden von den Objekten einer Anwendung verursacht. Die Anwendungsobjekte erzeugen diese Ereignisse und schicken sie an den Event Manager zur Weiterverarbeitung. Ein Client Event gehört einem bestimmten Typ an. Dabei können von der Anwendung beliebige Event-Typen mit unterschiedlichen Namen definiert und beim Event Manager registriert werden. Der Datentyp für die Daten eines Client Events ist: void *. Folgendes Codebeispiel zeigt das Erzeugen eines Client Events durch eine Anwendung (vgl. auch [IBM 94f S.12-4]):

```
clientEvent = SOMEClientEventNew();
_somevSetEventClientType (clientEvent, ev, "Example");
_somevSetEventClientData(clientEvent, ev, "any data");
/* assuming that "Example" is already registered with EMan*/
/* enqueue the above event with EMan*/
_someQueueEvent(some_gEMan, ev, clientEvent);
```

- *Work Procedure Events.* Dies sind eigentlich keine Events, sondern Hintergrundprozeduren, die immer dann ausgeführt werden, wenn gerade keine Ereignismeldungen verarbeitet werden müssen.

Die Weiterverarbeitung der Ereignismeldungen erfolgt durch *Callbacks.* Ein Callback ist eine Prozedur, die jedesmal automatisch ausgeführt wird, wenn eine Ereignismeldung der entsprechenden Art beim Event Manager eintrifft. Ein Anwendungsprogramm muß für jede Event-Klasse deren Ereignismeldungen sie empfangen will, eine Callback-Prozedur definieren. Hinsichtlich der Aktionen, die in den Callback-Prozeduren ausgeführt werden können, bestehen keine Beschränkungen.
Genauso wie eine Anwendung ihr Interesse an bestimmten Ereignismeldungen jederzeit registrieren kann, kann sie dies auch jederzeit über bestimmte Funktionsaufrufe wieder rückgängig machen.

Das SOM Replication Framework

Das Replication Framework unterstützt das Vorhandensein identischer Kopien eines Objekts in unterschiedlichen Adressräumen. Wenn der Zustand einer dieser Kopien verändert wird, werden die entsprechenden Updates automatisch auch auf allen anderen Kopien des Objekts vollzogen. Damit wird ein konsistenter Zustand aller replizierten Objekte erreicht.

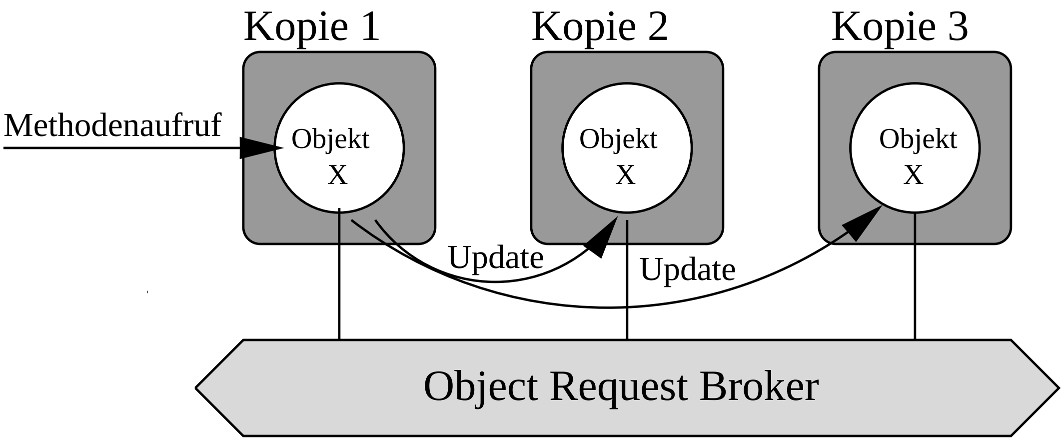

Abbildung 2.9: Das Replication Framework

Damit die Objekte einer Klasse repliziert werden können, muß im Vererbungsbaum der Klasse die Klasse *SOMRReplicbl* auftreten. Außerdem müssen alle Methoden der Klasse, die die Instanzvariablen eines entsprechenden Objekts verändern, auf eine bestimmte Art und Weise überschrieben werden, damit die Zustandsänderungen auf den Kopien des Objekts nachvollzogen werden können.
Für die Ausbreitung der Zustandsänderungen gibt es zwei Alternativen:

- Beim *Operation Logging* wird jeder Methodenaufruf, der den Zustand eines replizierten Objekts verändert, auch auf allen Kopien nachvollzogen.
- Beim *Value Logging* wird der Zustand eines Objekts nach einem Methodenaufruf kodiert und der Zustand der Kopien des Objekts wird anschließend aktualisiert.

Der Unterschied besteht darin, daß bei der ersten Alternative die Berechnung der Methode wiederholt ausgeführt werden muß, während beim zweiten Fall lediglich die Werte der Attribute aktualisiert werden. Bei Methoden, deren Ausführung aufwendig ist (z.B. hoher Verbrauch an Rechenzeit) ist daher das *Value Logging* besser geeignet.
Für die Realisierung der Zustandsupdates sind u.a. die Dienste des Event Management Frameworks nötig.

Das SOM Metaclass Framework

Das Metaclass Framework umfaßt eine Reihe von vordefinierten Metaklassen, die entweder direkt als Metaklasse für Anwendungsklassen dienen können, oder als Ausgangspunkt für benutzerdefinierte Metaklassen.
Folgende vordefinierte Metaklassen sind im Framework enthalten:

- *SOMMBeforeAfter* - damit können Methoden definiert werden, die automatisch ausgeführt werden, wenn eine Methode auf einer Instanz dieser Klasse aufgerufen wird.
- *SOMMSingleInstance* - mit Hilfe dieser Metaklasse können Klassen definiert werden, die höchstens eine Instanz haben.
- *SOMMTraced* - die Methoden der Klassenobjekte dieser Metaklasse geben nach jedem Aufruf eine Meldung mit den Parametern und dem Rückgabewert aus.
- *SOMRReplicable* und *SOMRReplicableObject* - dies sind die Metaklassen für die Objekte des Replication Frameworks.

Das SOM Persistence Framework

Mit dem Persistence Framework kann der Zustand von Objekten persistent gespeichert werden. Dadurch wird erreicht, daß Objekte länger verfügbar sind, als die Lebensdauer des Prozesses, in dem sie erzeugt wurden.
Die Speicherung der Objekte erfolgt in Dateien. Komplexere Aspekte der Datenhaltung wie Recovery, Transaktionen oder Zugriffssynchronisation werden nicht unterstützt.
Allerdings ist das Persistence Framework wie die meisten SOM-Klassen erweiterbar, so daß es durchaus denkbar ist, daß mit einer entsprechenden Anpassung z.B. auch objektorientierte Datenbanken zum Speichern der Objekte herangezogen werden können.

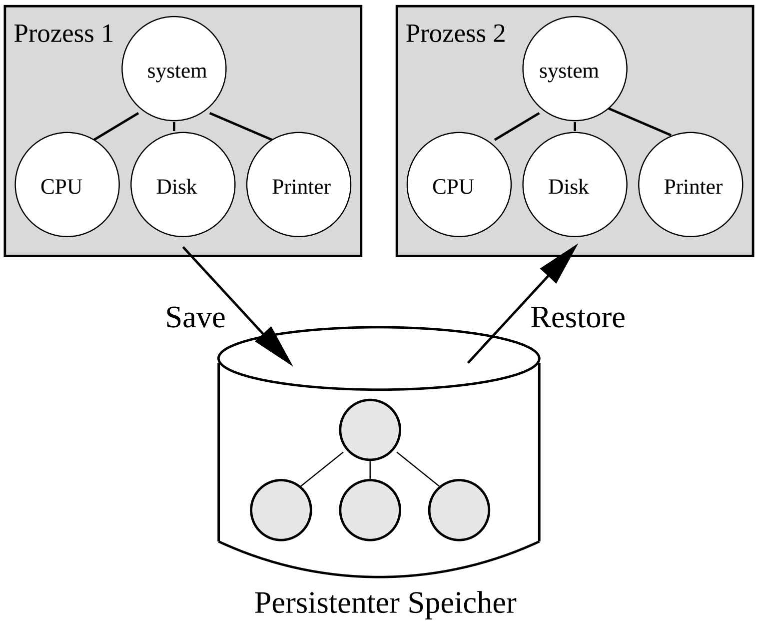

Abbildung 2.10: Persistence Framework

Objekte, deren Zustand persistent gespeichert werden soll, müssen die Klasse *SOMPPersistentObject* in ihrem Vererbungsbaum haben. Das Persistence Framework kann Objekte von beliebiger Komplexität speichern und wiederherstellen. Das heißt, es können Objekte gespeichert werden, die ihrerseits wiederum Objekte als Daten enthalten. Man spricht hierbei von

Embedded Objects. Wenn ein Objekt gespeichert wird, werden normalerweise alle anderen Objekte, die in ihm enthalten sind, automatisch mitgespeichert - sofern dies vom Anwendungsprogrammierer nicht explizit ausgeschlossen wurde.
Jedem persistenten Objekt muß ein eindeutiger Identifikator *(persistent ID)* zugeordnet werden. Diese ID gibt an, wo und wie das Objekt gespeichert werden soll. Dabei wird folgendes Format verwendet:

```
<IOGroupManagerClassName>:<IOGroupName>:<GroupOffset>
```

Der IOGroupManagerClassName gibt an, wie das Objekt gespeichert werden soll, z.B. im Ascii- oder Binärformat. Eine IOGroup beinhaltet alle Objekte, die zusammen in einer Datei abgespeichert werden sollen. Hier wird der entsprechende Dateiname angegeben. Der GroupOffset schließlich identifiziert das Objekt innerhalb seiner IOGruppe und gibt an, wo in der Datei das Objekt gespeichert werden soll.
Die persistentID muß spezifiziert werden, bevor das Objekt abgespeichert werden kann; anhand ihr kann das Objekt später auch wieder reinitialisiert werden.

Das SOM Collection Class Framework

Das Sortiment der SOM Frameworks wird durch das Collection Class Framework abgerundet. Dieses erfüllt zwar keine direkte vordefinierte Funktion wie die anderen Frameworks, es stellt dem Anwendungsprogrammierer aber eine Vielzahl von nützlichen Klassen zur Verfügung. Dabei handelt es sich um Datenstrukturen, die für die Organisation und Strukturierung von anderen Objekten herangezogen werden können.
Die meisten der heute beim Programmieren üblichen Datenstrukturen sind dabei implementiert. Beispiele sind:

- Hash Tables
- Dictionaries (key, value)
- Priority Queues
- Linked Lists

Zusätzlich gibt es sogenannte *Iteratoren*, mit denen Clients die verschiedenen Objekte innerhalb der Datenstrukturen aufsuchen können.

2.4 Einsatz von ORBs im Systemmanagement

Der Einsatz von CORBA für Systemmanagementaufgaben ist derzeit ein aktuelles Forschungsgebiet. Dieser Abschnitt stellt zunächst einige Stärken und Vorteile dieser Architektur dar, die sie für das Management von verteilten Systemen attraktiv machen. Anschließend wird ein kurzer Überblick über derzeitige Forschungs- und Standardisierungsaktivitäten auf dem Gebiet gegeben.

2.4.1 Vorteile von CORBA-basierter Managementsoftware

Neben den allgemeinen Vorteilen der OMA (vgl. Abschnitt 2.1) sind speziell beim Einsatz dieser Architektur für das Netz- und Systemmanagement folgende Aspekte positiv zu beurteilen:

- Man kann die gleiche Architektur sowohl für verteilte Anwendungen als auch für deren Management verwenden. Die Managementanwendungen sind Spezialfälle von verteilten Anwendungen.
- Managementfunktionalität kann leicht direkt in die verteilten Anwendungen (evtl. sogar in die Betriebssysteme von Rechnern) integriert werden.
- Durch die Möglichkeit der Verteilung kann man Teilaspekte von Managementanwendungen durch speziell dafür geeignete Systemressourcen (Hard- und Software) realisieren. Die Granularität der Verteilung sind dabei die Objekte einer Anwendung.
- Vorhandene CORBAfacilities und CORBAservices können für Managementzwecke genutzt werden. Spezielle Facilities für das Systemmanagement befinden sich in der Standardisierungsphase.
- Managementanwendungen, die unter Verwendung der OMG-IDL erstellt wurden, sind beliebig wiederverwendbar, erweiterbar und interoperabel mit anderen Anwendungen. Im Idealfall gibt es Managementframeworks für verschiedene Teilaufgaben, aus denen man sich im „Baukastenprinzip" eine für den Einzelfall angemessene Lösung zusammenbauen kann.
- Im Rahmen der vorhandenen Language Mappings kann die Programmiersprache für die Implementierung von Managementanwendungen frei gewählt werden. Für SNMP und CMIP müssen ganz spezielle APIs verwendet werden. Die Auswahl der Implementierungssprache ist eingeschränkter als bei CORBA.

2.4.2 Forschungsaktivitäten

Der Einsatz von CORBA-basierter Technologie zur Lösung von Managementproblemen ist ein neues Forschungsgebiet. Die momentan auf diesem Gebiet laufenden Standardisierungsaktivitäten befassen sich einerseits mit der Definition von speziellen Objektdiensten für das Systemmanagement und andererseits mit der Interoperabilität von OMG-Technologie und den herkömmlichen Managementarchitekturen (Internet- und OSI-Management).

Common Facilities für das Systemmanagement

Die *X/Open Systems Management Working Group (XoTGsysMan)* hat in einer vorläufigen Spezifikation (vgl. [X/OPEN 95b]) allgemeine Dienste für das Systemmanagement definiert. Diese Common Facilities sollen ein Rahmenwerk für die Entwicklung von Managementanwendungen in einer OMG-Umgebung zur Verfügung stellen (vgl. Abb. 2.11).
Die Spezifikation umfaßt folgende Dienste:

- Managed Sets
  Managed Sets sind Zusammenfassungen von Objekten in Gruppen. Dadurch können

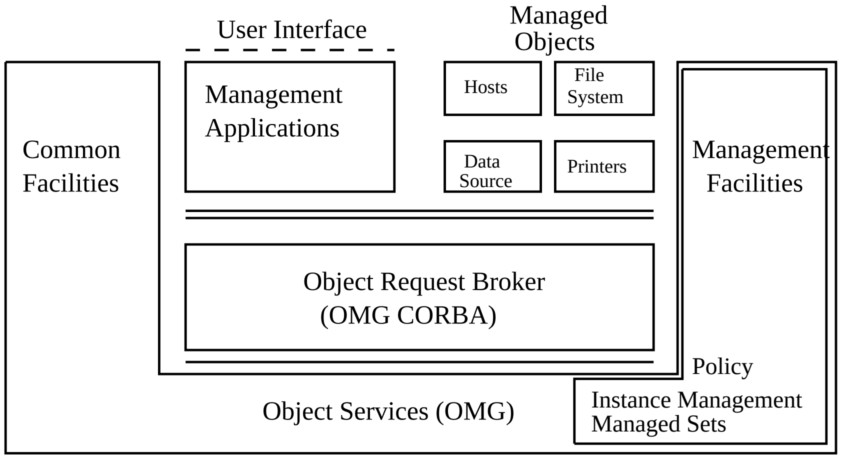

Abbildung 2.11: Systems Management Rahmenwerk von X/Open

Managementressourcen logisch zusammengefaßt und gemeinsam administriert werden. Managed Sets stellen u.a. Methoden zum Hinzufügen und Löschen von Mitgliedern und zum Einholen von Information über die Objekte innerhalb der Gruppe zur Verfügung.

- Instance Management Service
  Der Instance Management Service dient zum Erzeugen und Verwalten von Managed Objects. Für jeden Typ von Managed Object (sog. Managed Instance) gibt es einen Instance Manager. Dieser übernimmt das Lifecycle Management (Erzeugen, Löschen etc.) für diesen Typ von Managed Object.

- Policy Management Service
  Der Policy Management Service erlaubt die Definition und Überwachung der Einhaltung von Policies (Zielvorgaben) für Gruppen von Objekten. Damit können bestimmte Regeln für das Management von Ressourcen aufgestellt werden.

X/Open sieht in diesen Diensten nur eine Untermenge aller nötigen Systems Management Common Facilities. Weitere Kandidaten, die für eine zukünftige Spezifikation identifiziert wurden, sind Dienste für Prozeß-Management, Scheduling und Event-Management sowie Language Bindings für Sprachen, die gewöhnlich zum Schreiben von Skripts für die Systemadministration verwendet werden (z.B. Perl).

Interoperabilität zwischen Managementarchitekturen

Die *Joint Inter-Domain Management (JIDM)* Working Group befaßt sich mit der Kooperation von Managementarchitekturen. Das Ziel dieser von X/Open und dem Network Management Forum gemeinsam eingerichteten Forschungsgruppe ist, die Interoperabilität von Managementsystemen zu ermöglichen, die auf den Protokollen CMIP, CORBA und SNMP basieren.
Die Interoperabilität zwischen den verschiedenen Managementarchitekturen soll durch Abbildungen zwischen den Spezifikationssprachen für Managementinformation und durch Mechanismen zur Protokollumsetzung erreicht werden.

1. Specification Translation:
   Eine Voraussetzung für Interoperabilität ist, daß die Managementinformation einer Architektur im Informationsmodell der anderen Architekturen dargestellt werden kann. Dafür werden Algorithmen und Werkzeuge für die Abbildung zwischen den Spezifikationssprachen benötigt. In [X/OPEN 95a] werden Algorithmen für die Abbildung zwischen OMG-IDL, GDMO und Internet-SMI definiert.

   Als Beispiel für die Specification Translation wird auf der folgenden Seite die GDMO-Definition der Managed-Object Klasse *top* und die entsprechende IDL-Definition gezeigt.

Zunächst die GDMO-Definition:

```
top      MANAGED OBJECT CLASS
CHARACTERIZED BY
topPackage      PACKAGE
        BEHAVIOUR
        topBehaviour;
        ATTRIBUTES
        objectClass GET,
        nameBinding     GET ;;;
CONDITIONAL PACKAGES
        packagesPackage PACKAGE
        ATTRIBUTES      packages        GET;
        REGISTERED AS {joint-iso-ccitt ms(9) smi(3) part2(2) package(4)   16};
        PRESENT  IF  "any registered package, other than this package has been
                      instantiated";

        allomorphicPackage      PACKAGE
        ATTRIBUTES
        allomorphs      GET;
        REGISTERED AS {joint-iso-ccitt ms(9) smi(3) part2(2) package(4)  17};
        PRESENT  IF  "if an object supports allomorphism" ;

REGISTERED AS  { joint-iso-ccitt ms(9) smi(3) part2(2) managedObjectClass(3) 14};
```

Ein GDMO-IDL Compiler gemäß JIDM erzeugt daraus folgende IDL-Definition:

```
module X.721{
interface top : OSIMgmt::ManagedObject {

   X721Att::ObjectClassType objectClass_Get ()
             raises (CMIP_ATTRIBUTE_ERRORS);
   X721Att::NameBindingType nameBinding_Get ()
             raises (CMIP_ATTRIBUTE_ERRORS);
   X721Att::PackagesType packages_Get ()
             raises (CMIP_ATTRIBUTE_ERRORS);
   X721Att::AllomorphsType allomorphs_Get ()
             raises (CMIP_ATTRIBUTE_ERRORS);
 };
}
```

Man sieht, daß die `ATTRIBUTES` der GDMO-Definition abgebildet werden auf IDL get-Methoden. Die IDL-Form der ASN.1-Datentypen ist in einem speziellen Modul (X721Att) definiert.

2. *Interaction Translation*:
   Interaction Translation ist die Protokollumsetzung zwischen CORBA, CMIP und SNMP. Die Umsetzung wird durch Gateways zwischen den Protokollwelten erreicht.

Durch ein Protokollgateway zwischen CORBA und CMIP kann eine CORBA-basierte Managementanwendung beispielsweise mit einem GDMO-Objekt interagieren, ohne zu wissen, daß dieses einer anderen Protokollwelt (Domain) angehört. Ebenso kann ein OSI-Manager auf ein CORBA-Objekt zugreifen.

Einen Ansatz zur Spezifikation eines CORBA/CMIP-Gateways findet man in [HIERRO].

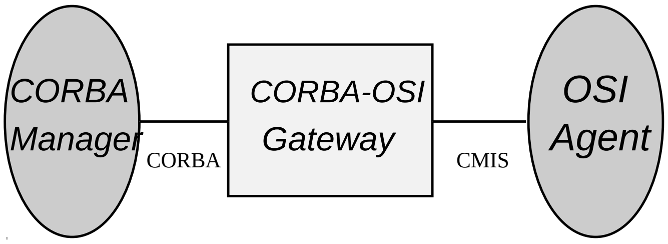

Abbildung 2.12: OSI-CORBA Protokollgateway

Kapitel 3

Die Managementplattform NetView für AIX

Dieses Kapitel gibt zunächst einen allgemeinen Überblick über Aufbau und Architektur von Managementplattformen. Die Darstellung ist dabei an [HEGE 93] angelehnt. Danach werden die einzelnen Bausteine konkret am Beispiel von NetView für AIX diskutiert.

3.1 Managementplattformen

Managementplattformen stellen eine Infrastruktur bereit, in die verschiedene Managementanwendungen eingebettet werden können. Im Gegensatz zu konventionellen Managementwerkzeugen, die häufig auf bestimmte Anwendungsbereiche beschränkt sind (z.B. Fehlermanagement) zeichnen sie sich u.a. durch folgende Merkmale aus:

- Sie sind nicht auf das Management von Ressourcen bestimmter Hersteller beschränkt.
- Sie umfassen mehrere Funktionsbereiche.
- Die Speicherung von Managementinformation erfolgt in Form von objektorientierten Datenmodellen.
- Sie stellen Programmierschnittstellen bereit, mit denen Managementanwendungen erstellt und in die Plattform integriert werden können.
- Sie haben eine grafische Benutzeroberfläche.

Abbildung 3.1 zeigt im Überblick den allgemeinen funktionalen Aufbau einer Managementplattform.
Die Infrastruktur hat folgende Bestandteile:

- Ein Kernsystem sorgt für die Kommunikation und Koordination der übrigen Bausteine. Es steuert den Datenfluß und überwacht die Zusammenarbeit der Module.
- Ein Kommunikationsbaustein ermöglicht die Kommunikation mit Fremdsystemen wie Netzkomponenten, Endsystemen oder Anwendungen.
- Der Informationsbaustein speichert Managementinformation in Form eines objektorientierten Datenmodells.

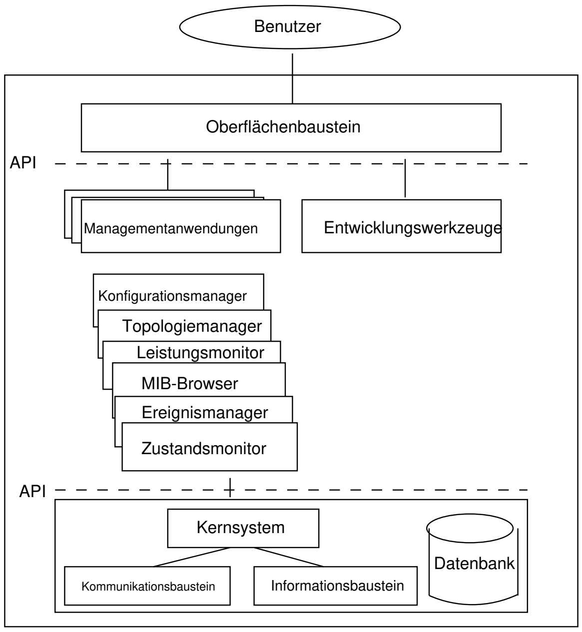

Abbildung 3.1: Die Architektur von Managementplattformen

- Ein Oberflächenbaustein stellt das Netz grafisch dar und ermöglicht den Benutzern Zugang zu Managementanwendungen.

Darüberhinaus enthalten die meisten Managementplattformen eine Reihe von Basisanwendungen. Ein Zustandsmonitor dient zur Überwachung von Ressourcen. Ein Ereignismanager verarbeitet ankommende Ereignismeldungen. Er filtert sie, verteilt sie an Anwendungen, zeigt sie dem Benutzer an und speichert sie in Log-Dateien. Ein Konfigurationsmanager unterstützt die Konfiguration von Ressourcen von der Plattform aus. Ein Topologiemanager überwacht die Netztopologie, wobei neue Objekte selbstständig erkannt und in das vorhandene Topologiemodell eingefügt werden. Ein Leistungsmonitor dient der langfristigen Sammlung von Leistungsdaten, die in periodischen Abständen von den Ressourcen abgefragt werden.
Entwicklungswerkzeuge, wie z.B. Compiler für Managementinformation oder Anwendungsgeneratoren, ermöglichen die Erweiterung der Funktionalität der Plattform.

3.2 Die Architektur von NetView für AIX

Dieser Abschnitt behandelt anhand des soeben eingeführten Architekturmodells detailliert den Aufbau der Managementplattform NetView für AIX (nachfolgend abkürzend als NetView bezeichnet). Zur weiteren Beschäftigung mit der Plattform sei auf die im Literaturverzeichnis aufgeführten Handbücher verwiesen.

3.2.1 Infrastruktur

Kommunikationsbaustein

NetView unterstützt die Managementprotokolle SNMP[1] und CMOT[2]. Als Programmierschnittstellen dienen ein SNMP-API[3] sowie XMP/XOM[4], mit dem sowohl SNMP- als auch CMOT-Anwendungen programmiert werden können.
XMP-basierte Interaktionen mit Agenten werden über die *Communications Infrastructure* abgewickelt. Diese besteht aus den Hintergrundprozessen *pmd* und *orsd* und aus der ORS-Datenbank. Diese Komponenten wirken zusammen, um den Zugriff auf Managed Objects zu ermöglichen.
Der *Postmaster Dämon (pmd)* enthält einen SNMP- und einen CMOT-Stack. Er dient als Schaltzentrale und vermittelt die ein- und ausgehenden Management-PDUs zwischen den Managementanwendungen und den Agenten. Für diese Routing-Funktion verwendet er die Information in der ORS-Datenbank (*Object Registration Services*). Die ORS-Datenbank ist ein globales Verzeichnis aller Agenten. In ihr ist gespeichert, welche Objekte ein Agent verwaltet, welche Aktionen auf den Objekten ausgeführt werden können und welches Managementprotokoll (CMOT oder SNMP) für den Zugriff auf die Objekte verwendet werden muß. Dadurch wird erreicht, daß beim Zugriff auf OSI-Objekte nur die Objektklasse und die Objektinstanz angegeben werden muß, nicht aber der zuständige Agent, die Adresse oder der Protokollstack.

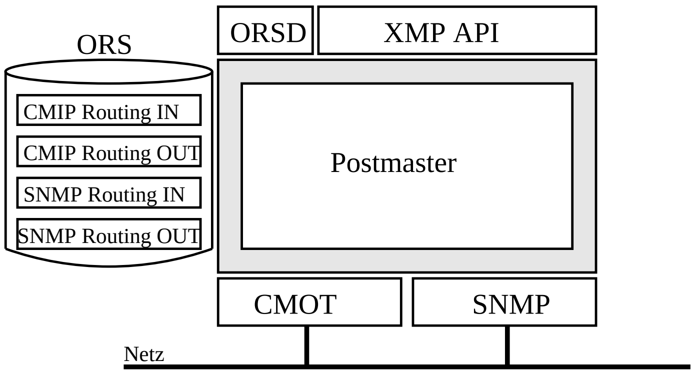

Abbildung 3.2: Der Kommunikationsbaustein von NetView

Die Konsistenz der Information in der ORS-Datenbank wird durch den orsd-Prozess sichergestellt. Er steuert u.a., wieviel Information in einem Cache gespeichert wird, und wie oft Garbage Collection durchgeführt werden soll.
Neben der Routing-Funktion übernimmt pmd auch die Verwaltung der CMOT-Verbindungen *(association management)* und die Steuerung der Timeouts und Retries bei SNMP.
Theoretisch ist es denkbar, auch die Zugriffsfunktion auf einen ORB in den Postmaster zu integrieren, was jedoch aus den in Kapitel 4 genannten Gründen nicht durchgeführt wurde.

[1] Simple Network Management Protocol
[2] CMIP (Common Management Information Protocol) over TCP/IP
[3] Application Programming Interface
[4] X/Open Management-Protocols-Application-Program-Interface / X/Open OSI-Abstract-Data Manipulation API

Informationsbaustein

Der Informationsbaustein von NetView besteht aus einer Reihe von Datenbanken, in denen managementrelevante Information gespeichert wird:

- Die IP-Topologie-Datenbank, in der Information über alle Ressourcen gespeichert ist, die mit der IP-Discovery-Funktion entdeckt wurden.
- In der GTM-Datenbank wird Information von Topologieanwendungen gespeichert, die mit dem *General Topology Manager (GTM)* kommunizieren (vgl. 3.3.4).
- Die Objekt-Datenbank, in der für jede Ressource, die NetView bekannt ist, ein Eintrag enthalten ist [5]. Ein Auszug aus einer NetView-Objekt-Datenbank befindet sich in Anhang B.

Außerdem gibt es eine Reihe weiterer Datenbanken u.a. zum Abspeichern von SNMP-Traps und Darstellungsinformation für die Benutzeroberfläche. Für eine genauere Betrachtung sei auf den Anhang der Arbeit verwiesen.

3.2.2 Basisanwendungen

Discovery und Topologiemanagement

Das Topologiemanagement für IP-Netze umfaßt die Discovery-Funktion und das Statuspolling von Komponenten. Die Discovery-Funktion dient zum Sammeln von Konfigurationsinformation über das Netz und die darin enthaltenen Ressourcen und die Darstellung der Information an der grafischen Oberfläche.
NetView hat eine Reihe von Prozessen für das Topologiemanagement von IP-adressierbaren Komponenten. Für das Topologiemanagement von Ressourcen, die andere Protokolle verwenden, können eigene Anwendungen mit dem *General Topology Manager* erstellt werden. Der *General Topology Manager* wird im Abschnitt über die Anwendungsentwicklung behandelt.
Folgende Aufgaben werden durch die IP-Discovery erfüllt:

- Entdeckung neuer Netzkomponenten.
- Polling, um Status- und Konfigurationsänderungen von Ressourcen festzustellen.

Diese Anforderungen werden durch eine Reihe von kooperierenden Prozessen erfüllt. Die zentrale Rolle bei der IP-Discovery spielt der *netmon* Dämon. Dieser „entdeckt“ IP-Komponenten. Dafür verwendet er u.a. den ARP-Cache[6], ICMP (ping)[7], die MIBs[8] von bereits bekannten Komponenten per SNMP-get (z.B. Routing-Table-Information), oder das weiter unten besprochene *Seed File*. *Netmon* übernimmt außerdem das Polling, um Konfigurations- und Statusänderungen festzustellen.
Für das Eintragen der neuen Komponenten in die IP-Topologie-Datenbank und die Objekt-Datenbank ist der Prozess *ovtopmd* zuständig.

[5] Nicht zu verwechseln mit der vorher erwähnten ORS-Datenbank, in der Lokalisierungsinformation für OSI-Objekte gespeichert ist.
[6] Address Resolution Protocol
[7] Internet Control Message Protocol
[8] Management Information Base

Die Anwendung *ipmap* sorgt für die Konsistenz zwischen der Information in der Topologie-Datenbank und der grafischen Oberfläche. Sie ist z.B. dafür zuständig, die entsprechenden Symbole zu erzeugen, wenn eine neue Komponente entdeckt wurde.
Um für die ermittelten IP-Adressen die entsprechenden Komponentennamen (z.B. hostname) zu ermitteln, wird entweder der *Domain Name Service* (DNS) verwendet oder die /etc/hosts Dateien der Komponenten. Der so ermittelte Name wird dann auf dem der Komponente entsprechenden Symbol (Icon) angezeigt.
Für die effiziente und effektive Durchführung des Topologiemanagements ist eine Konfiguration der SNMP-Parameter und der Polling-Intervalle nötig. Sonst könnte es entweder sein, daß die entsprechende Information in der Plattform nicht aktuell genug ist, oder daß durch die Discovery-Funktion eine zu große Netzlast erzeugt wird.
NetView bietet die Möglichkeit, die SNMP-Parameter sowohl für einzelne Netzknoten, als auch für Gruppen von Komponenten (durch Angabe von Wildcards in den IP-Adressen) unterschiedlich einzustellen. Zu den konfigurierbaren Parametern gehören die *community names* (für GET- und SET-Requests) sowie die Timeout- und Retry-Werte. Desweiteren kann angegeben werden, ob für den Zugriff auf bestimmte Ressourcen ein Proxy-Agent verwendet werden soll.
Außerdem können zusätzlich die Werte für das Topologie- und Statuspolling eingestellt werden. Neben dem Einstellen von festen Polling-Intervallen besteht auch die Möglichkeit, ein *Auto Adjusting Polling Intervall* zu verwenden. Dabei hängt die Polling-Frequenz davon ab, wieviele neue Komponenten im letzten Polling-Zyklus entdeckt wurden.
Normalerweise ist das Discovery-Gebiet beschränkt bis zu den von der Managementstation aus jeweils nächsten Routern. Es besteht jedoch die Möglichkeit der Ausdehnung oder Beschränkung durch *Seed Files*. Ein Seed File kann Komponentennamen, IP-Adressen oder Bereiche von IP-Adressen enthalten. Dadurch kann erreicht werden, daß bei entsprechender Konfigurierung des *netmon* Dämons und der Polling-Parameter die Komponenten im Seed File entweder ausschließlich oder zusätzlich zum normalen Discovery-Gebiet kontrolliert werden.

Das Ereignismanagement

Das Ereignismanagement hat die Aufgabe, Ereignismeldungen entgegenzunehmen und zu verarbeiten. NetView unterscheidet zwischen internen *Map Events*, die eintreten, wenn ein Benutzer oder eine Anwendung den Status der geöffneten Map ändert, und *Network Events* (SNMP-Traps oder CMOT-Notifications), die Ereignisse im Netz signalisieren. Außerdem werden Ereignismeldungen zur internen Kommunikation zwischen den NetView-Prozessen eingesetzt.
SNMP-Traps werden vom trap-Dämon (trapd) empfangen. Er speichert sie in seiner Log-Datei und verteilt sie an Anwendungen weiter, die sich über das SNMP-API dafür registriert haben. Anschließend leitet er sie an den Postmaster weiter, der sie wiederum dem Filter-Dämon (ovesmd) übergibt (vgl. Abb. 3.3).
Der Filter-Dämon kann Ereignismeldungen nach bestimmten Kriterien filtern. Die gefilterten Ereignismeldungen werden an die Anwendungen weitergeleitet, die sich dafür registriert haben (z.B. die nvevents-Anwendung zum Anzeigen der Ereignismeldungen an der grafischen Oberfläche oder Benutzeranwendungen). Für Benutzeranwendungen, die gefilterte Ereignismeldungen empfangen wollen, stehen spezielle Filterfunktionen zur Verfügung. Filtermöglichkeiten gibt es sowohl für XMP-basierte Managementanwendungen, als auch für Anwendungen, die das SNMP-API verwenden.

Das Speichern der gefilterten Ereignismeldungen übernimmt der Log-Dämon (ovelmd). Seine Log-Datei (ovevent.log) dient u.a. als Informationsquelle für das Anzeigen der Event-History. CMOT-Event-Reports werden direkt vom Postmaster empfangen und an den Filter-Dämon weitergeleitet.

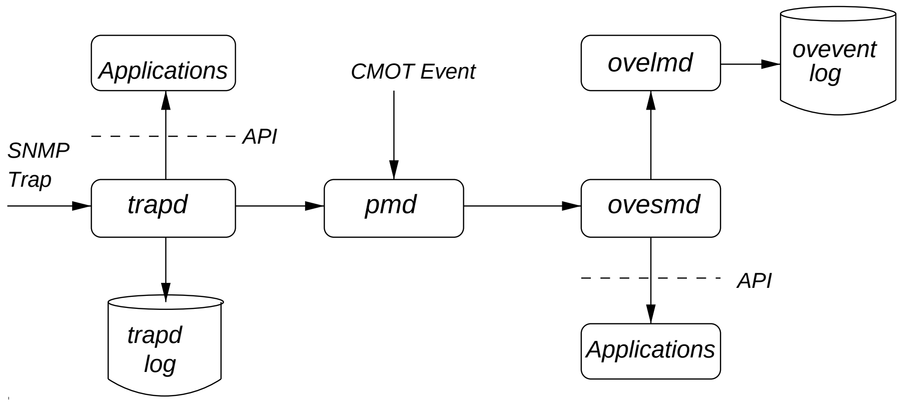

Abbildung 3.3: Verarbeitung von Ereignismeldungen

Die *nvevents* Anwendung zeigt Ereignismeldungen an der Benutzeroberfläche an. Dabei können Ereignisse entweder in Form von Event-Karten oder im Listenformat angezeigt werden. Die Darstellung der Ereignisse erfolgt in sogenannten *Dynamic Workspaces*. Diese werden automatisch aktualisiert, wenn neue Ereignismeldungen eintreffen. Es können mehrere *Dynamic Workspaces* gleichzeitig geöffnet werden, wobei jeder Ereignisse nach unterschiedlichen Filterkriterien anzeigen kann (z.B. Event-Quelle, Event-Kategorie, etc.).
Zusätzlich zu den Filtermöglichkeiten bestehen weitere Konfigurationsmöglichkeiten für das Ereignismanagement (bei SNMP-Traps) :

- Man kann neue Enterprise-Specific-Traps definieren (z.B. für CORBA-Ereignisse).
- Die Information, die von der *nvevents* Anwendung beim Eintreffen einer bestimmten Ereignismeldung angezeigt wird, kann konfiguriert werden.
- Es können bestimmte Aktionen beim Eintreffen von Ereignismeldungen automatisch ausgeführt werden (z.B. Shell-Skripts). Dadurch können Aufgaben des Fehlermanagements automatisiert werden.
- Ereignismeldungen können in Kategorien gruppiert werden, um z.B. bessere Filtermöglichkeiten zu bieten.
- Beim Eintreffen von Ereignismeldungen können Botschaften in Fenstern angezeigt werden. Beispielsweise kann dadurch einem Operateur mitgeteilt werden, was er bei einem bestimmten Ereignis zu tun hat.
- Traps können durch einen eigenen Prozess (*tralert*) in NetView-S/390-Alerts umgewandelt werden und über die *NetView-Service-Point* Anwendung an eine NetView-Anwendung auf einem Hostrechner geschickt werden.

Für CMOT-Event-Reports beschränken sich die Dienste des Ereignismanagements auf Empfang und Filtern.
Um die Dienste der Plattform für das CORBA-Ereignismanagement nutzen zu können, wurden die in Kapitel 5 beschriebenen Schnittstellen zu den soeben genannten Komponenten entwickelt.

Weitere Basisanwendungen

NetView stellt dem Benutzer noch eine Vielzahl zusätzlicher Funktionalität zur Verfügung, von der hier noch zwei typische Managementplattformanwendungen genannt werden.
Schwellwert- und Leistungsüberwachung :
Die Überwachung von Schwellwerten und die Messung von Leistungsdaten sind wichtig, um die Verfügbarkeit von Netzkomponenten und die aktuelle Dienstqualität bestimmen zu können. Damit können Fehler und Leistungsengpässe frühzeitig erkannt und verhindert werden. Diese Aufgaben erfüllt bei NetView (für den SNMP-Bereich) der *MIB-Data-Collector (SNMP-Collect)*. Mit dieser Anwendung können MIB-Daten von bestimmten Netzknoten periodisch durch Polling abgefragt und in einer Datei gespeichert werden. Außerdem besteht die Möglichkeit, auf einer MIB-Variable einen Schwellwert zu definieren, bei dessen Überschreitung der *MIB-Data-Collector* einen Trap auslöst. Die gesammelten Daten können grafisch dargestellt werden. Zusätzlich besteht die Möglichkeit, die Information in einer relationalen Datenbank zu speichern, um flexible Abfragen darauf zu machen.
MIB-Browser :
Der MIB-Browser erlaubt direkten Zugriff auf die Managementinformation von Komponenten. Ein Benutzer kann sich per Mausklick durch die baumartige Struktur der MIBs bewegen und die aktuellen Werte der MIB-Variablen abfragen. Es werden sowohl die Standard-Internet-MIBs als auch herstellerspezifische MIBs unterstützt (die in die MIB-Datenbank der Plattform geladen werden müssen).

3.3 Anwendungsentwicklung für NetView

NetView stellt mehrere Programmierschnittstellen für die Entwicklung von Anwendungen zur Verfügung. Es gibt APIs für den Zugriff auf Managementprotokollstacks (SNMP, CMOT), für die Basisdienste der Plattform (Topologiemanagement und Event Filtering) und für die Benutzeroberfläche der Plattform.

3.3.1 XMP/XOM

XMP (X/Open Management Protocols Application Program Interface) ist ein API, mit dem Anwendungen auf die Management-Protokolle CMIP (bzw. CMOT bei NetView) und SNMP zugreifen können. Die Parameter für die XMP-Funktionsaufrufe werden mit *XOM (X/Open OSI-Abstract-Data-Manipulation API)* erstellt, einem API für die Repräsentation von ASN.1-Datentypen in C.
Die Funktionsaufrufe von XMP unterstützen sowohl die CMIP-Operationen, als auch die von SNMP:

CMIP Operation	SNMP Operation	XMP Funktionsaufruf
Action	-	mp_action_req(), mp_action_rsp()
Cancel get	-	mp_cancel_get_req(), mp_cancel_get_rsp()
Create	-	mp_create_req(), mp_create_rsp()
Delete	-	mp_delete_req(), mp_delete_rsp()
Get	Get	mp_get_req(), mp_get_rsp()
Set	Set	mp_set_req(), mp_set_rsp()
Notification	Trap	mp_event_report_req(), mp_event_report_rsp()
-	Get next	mp_get_next_req(), mp_get_next_rsp()

Außer den Protokolloperationen hat XMP noch zusätzliche Funktionsaufrufe zum Einrichten und Verwalten von Kommunikationsbeziehungen. Abbildung 3.4 zeigt eine typische Umgebung für XMP-basierte Managementanwendungen, die (bei NetView über den Postmaster-Dämon) auf SNMP- und CMOT-Protokollstacks zugreifen.

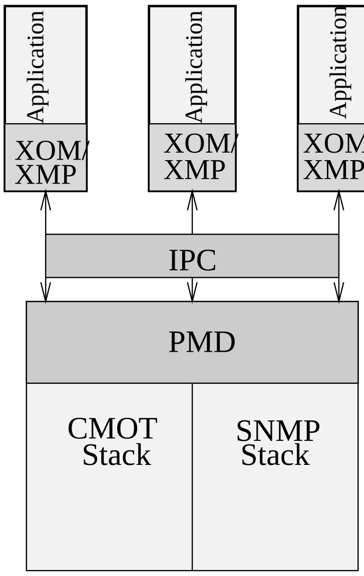

Abbildung 3.4: Das XMP/XOM - API

Das XMP-API ist nicht protokollunabhängig. Es können zwar die gleichen Funktionsaufrufe verwendet werden, um mit SNMP- und CMIP-Agenten zu kommunizieren, aber es werden je nach Managementprotokoll unterschiedliche Funktionsargumente gebraucht. Eine Managementanwendung muß also wissen, welches Protokoll ein bestimmter Agent unterstützt.
Die Argumente für die XMP-Funktionsaufrufe werden in Form von sogenannten OM-Objekten erzeugt. Bei diesen Objekten handelt es sich um Repräsentationen von ASN.1-Datentypen in C, die mit XOM erstellt werden. OM-Objekte setzen sich aus Attributen zusammen und unterstützen Vererbung. Z.B. gibt es eine OM-Klasse *GET-ARGUMENT* mit zwei Subklassen: *MP_C_SNMP_GET_ARGUMENT* und *MP_C_CMIS_GET_ARGUMENT*.
OM-Klassendefinitionen werden in sogenannten *Packages* zusammengefaßt. Es gibt zwei Arten von Packages:

- *Management-Service-Packages* definieren die Datenstrukturen für die XMP-Parameter (z.B. MP_C_CMIS_GET_ARGUMENT). Neben einer allgemeinen *Common-Package* gibt es eine *SNMP-Package* und eine *CMIP-Package*.
- *Management-Contents-Packages* beinhalten OM-Klassen für in ASN.1 spezifizierte Managed-Object-Definitionen. NetView beinhaltet die *DMI-Package* für die Managed-Object-Definitionen aus [ISO 10165-2]: *Management Information Services-Structure of Management Information Part 2: Definition of Management Information.*

XMP/XOM wird für SNMP-basierte Managementanwendungen aufgrund seiner großen Komplexität nur selten verwendet. Es ist jedoch momentan das einzige standardisierte API für CMIP.[9]Die Komplexität ist auch der Hauptgrund, warum es nicht sinnvoll ist, XMP/XOM als API für einen ORB zu verwenden (vgl. Kapitel 4).

3.3.2 SNMP-API

NetView hat ein spezielles SNMP-API. Mit diesem API ist die Anwendungsentwicklung unkomplizierter als mit XMP/XOM. Deshalb wird es in der Praxis bevorzugt verwendet, wenn Managementanwendungen die SNMP-Dienste der Plattform benutzen sollen. Dieser Abschnitt gibt einen Überblick über die Funktionalität des APIs.

Verwaltung von Kommunikations-Sessions

Es gibt Funktionsaufrufe zum Einrichten und Terminieren von SNMP-Sessions [10] mit Agenten. Außerdem können Anwendungen spezielle Sessions mit dem Trap-Dämon (*trapd*) und dem Event-Filter-Dämon (*ovesmd*) einrichten, um ungefilterte bzw. gefilterte Traps zu empfangen. Für jede Session wird eine bestimmte Datenstruktur angelegt, die als Input-Parameter für diverse andere Funktionsaufrufe dient.

PDU-Management

Mit diesen Funktionsaufrufen können Datenstrukturen für SNMP-PDUs[11] und Variable-Bindings (Paare aus Variablennamen und Variablenwert) erzeugt, initialisiert und gelöscht werden. Der Typ der zu erzeugenden PDU wird dabei als Parameter mit angegeben.

Kommunikationsoperationen

Das Versenden von PDUs kann blockierend oder nicht-blockierend erfolgen. Im ersteren Fall blockiert die Anwendung nach dem Abschicken der SNMP-PDU solange, bis entweder eine Antwort oder ein Timeout-Signal eintrifft. Das Timeout-Intervall und gegebenenfalls die Anzahl der Retries (wiederholte Sendungen) können beim Einrichten der SNMP-Session als Parameter angegeben werden.

[9] Ein C++ API für CMIP wird momentan gerade vom Network Management Forum und von X/OPEN standardisiert [NMF 96].

[10] SNMP ist verbindungslos. Mit „Session" ist daher nicht eine Session im Sinne des OSI-Referenzmodells, sondern eine Kommunikationsbeziehung zwischen einer Managementanwendung und einem Agent gemeint.

[11] Protocol Data Unit

Beim nicht-blockierenden Versenden von PDUs wird eine Callback-Funktion angegeben, die automatisch aufgerufen wird, wenn eine Antwort-PDU eintrifft. Bis zum Eintreffen der Antwort kann die Anwendung andere Aktionen ausführen.
Es gibt Funktionsaufrufe zum Empfangen von PDUs (z.B. Traps). Dabei kann eine Anwendung angeben, ob sie PDUs von allen ihren offenen SNMP-Sessions empfangen und diese mittels Callbacks verarbeiten will, oder ob nur PDUs von einer bestimmten Session (und somit von einem bestimmten Agent) empfangen werden sollen.

Weitere Funktionsaufrufe

Zusätzlich gibt es Funktionsaufrufe, mit denen explizit die wiederholte Versendung noch ausstehender Requests eingeleitet werden kann, sowie spezielle Aufrufe, um SNMP-MIB-Tabellen auszulesen. Hierbei kann entweder ein einzelnes Feld der Tabelle oder die ganze Tabelle mit einem einzigen Funktionsaufruf ausgelesen werden (dabei wird der Aufruf intern auf eine Folge von *get-next*-Operationen abgebildet).

3.3.3 Filter-API

Das NetView-Filter-API ermöglicht die Definition von Filterregeln. Dadurch können Managementanwendungen die Ereignismeldungen, die sie empfangen wollen, nach bestimmten Kriterien spezifizieren. Es kann nach folgenden Kriterien gefiltert werden:

- CMOT-Notifications:
  - Managed Object Class
  - Distinguished Name (des sendenden Objekts)
  - Event Type
  - Event Time (Zeit der Versendung des Events)
  - Event Logged Time (Zeit, zu der die Ereignismeldung geloggt wurde)
  - Häufigkeit des Auftretens einer Ereignismeldung
- SNMP-Traps
  - Enterprise ID
  - IP-Adresse des Agenten
  - Event Time
  - Event Logged Time
  - Generic Trap Type
  - Specific Trap Type
  - Häufigkeit des Auftretens

Ein Filterausdruck wird durch einen String dargestellt, der aus Schlüsselworten, Vergleichsoperatoren, Werten und den logischen Operatoren NOT, AND und OR besteht. Die Schlüsselworte entsprechen dabei den oben genannten Filterkriterien.
Mit dem NetView-Filter-API kann man Filterausdrücke erstellen, mit einem Namen und einer Beschreibung versehen und in Dateien abspeichern, um sie später wiederverwenden zu können.
Die Registrierung von Managementanwendungen zum Empfangen von gefilterten Events läuft in folgenden Schritten ab:

- Für das Empfangen von gefilterten Traps mit dem SNMP-API.
  1. Erzeugen der Filterregel bzw. Auslesen einer vordefinierten Regel aus einer Filterdatei.
  2. Eröffnen einer Session mit dem Event-Filter-Dämon (ovesmd). Dabei wird die Filterregel als Argument übergeben.
- Mit dem XMP/XOM-API.
  1. Erzeugen der Filterregel.
  2. Konvertierung der Filterregel in eine XOM-Datenstruktur mit einem speziellen Funktionsaufruf.
  3. Registrierung der Anwendung zum Empfangen von Events.

Mit dem XMP/XOM-API können gefilterte SNMP-Traps und CMOT-Notifications empfangen werden.

3.3.4 Der General Topology Manager

NetView verfügt über Anwendungen für das Topologie-Management von IP-Netzen. Diese Anwendungen umfassen Discovery (*netmon*), Einträge in der IP-Toplogie-Datenbank (*ovtopmd*) und Präsentation der Topologie (*ipmap*).
Für das Topologie-Management von nicht-IP-Netzen dient der *General Topology Manager*. Diese Komponente der Plattform beinhaltet eine Reihe von Prozessen und ein abstraktes Datenmodell für die Modellierung von Netztopologien. Damit können Discovery-Anwendungen für nicht-IP-Netze erstellt werden (mit „nicht-IP“ ist gemeint, daß entweder ein anderes Schicht-3-Protokoll verwendet wird, oder daß die Topologie auf einer anderen Schicht dargestellt wird).

Komponenten

Eine auf dem *General Topology Manager (GTM)* basierende Topologie-Anwendung hat folgende Komponenten:

- Discovery-Anwendung - man muß eine Anwendung schreiben, die die Topologie ermittelt und Topologieänderungen sowie evtl. Statusänderungen feststellen kann.
- *gtmd* - die Discovery-Anwendung teilt die Information über die ermittelte Topologie dem *general topology manager daemon* mit. Dafür stehen eine Schnittstelle in Form von Enterprise-Specific-Traps und ein API zur Verfügung. Die Topologie-Information muß in Form des weiter unten beschriebenen abstrakten Datenmodells übermittelt werden. Der *gtmd* erzeugt für die empfangene Information Einträge in der GTM-Datenbank und der Objekt-Datenbank der Plattform.
- *xxmap* - diese Anwendung dient zur automatischen Präsentation der Information in der GTM-Datenbank an der Oberfläche der Plattform.
- *noniptopod* - dieser Dämon kann optional im Discovery-Prozeß verwendet werden. Er empfängt alle Traps vom *netmon*-Dämon, die die Entdeckung eines neuen IP-Knotens

anzeigen. Über einen in [IBM 94b S.289] beschriebenen Mechanismus überprüft er daraufhin, ob auf diesem Netzknoten Protokollinstanzen vorhanden sind, für die es eine eigene Discovery-Anwendung gibt. Falls dies der Fall ist, wird die Discovery-Anwendung automatisch gestartet, um Topologie-Information von diesem Netzknoten abzufragen.

Datenmodell zur Modellierung der Topologie

Innerhalb der GTM-Datenbank wird jedes Topologie-Modell durch einen eindeutigen Protokoll-Identifikator (in Form einer MIB-Object-ID) gekennzeichnet. Die Topologieinformation selbst wird mit einem aus der Graphentheorie abgeleiteten generischen Datenmodell dargestellt. Die wichtigsten Elemente des Modells werden hier kurz beschrieben. Für eine genauere Beschreibung siehe [IBM 94b S. 263ff.]

- *Vertex* - ein Vertex entspricht einem Knoten. Vertices bilden die unterste Ebene innerhalb der Enthaltenseinsrelationen des Topologie-Modells. Ein Vertex kann beispielsweise eine Netz-Interface-Karte eines Computers repräsentieren.
- *Graph* - ein Graph kann Vertices oder andere Graphen enthalten. Graphen können z.B. verwendet werden, um ein Netzwerk, eine Anwendung oder eine Workstation zu repräsentieren.
- *Arc* - Arcs stellen die Verbindungen zwischen Vertices oder Graphen dar. Sie können beispielsweise verwendet werden, um Protokollverbindungen oder Dienst-Benutzungsrelationen zu modellieren.

Abbildung 3.5 zeigt ein Beispiel für die Modellierungsmöglichkeiten mit diesen Komponenten.

Kommunikation mit dem General Topology Manager

Es gibt zwei Möglichkeiten, wie eine Discovery-Anwendung mit dem gtmd kommunizieren kann, um die Information in der GTM-Datenbank zu erzeugen:

1. über spezielle Enterprise-Specific-Traps
2. über eine Socket-Verbindung und das GTM-API

Die Funktionen des GTM-API können in folgende Gruppen unterteilt werden:

- Erzeugen von GTM-Objekten
- Löschen von GTM-Objekten
- Statusänderung von Vertices und Arcs
- Änderung von Attributwerten der Objekte
- Auslesen von Informationen aus der GTM-Datenbank

Eine genaue Beschreibung der einzelnen Funktionsaufrufe findet man in [IBM 94c]. Die Trap-Schnittstelle bietet im wesentlichen die gleiche Funktionalität, hat jedoch keine Möglichkeit zum Auslesen von Information aus der Datenbank.

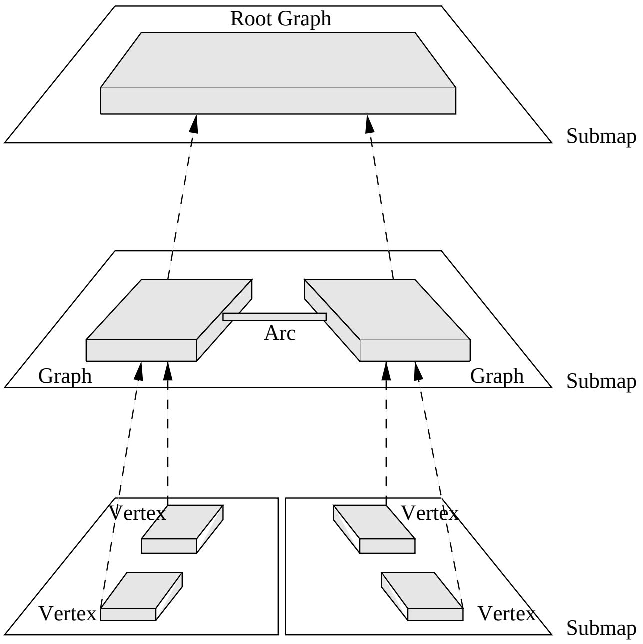

Abbildung 3.5: Beispiel für ein GTM-Informationsmodell

Der Vorteil des GTM besteht darin, daß die Präsentation der Topologiemodelle an der Benutzeroberfläche automatisch geschieht (durch die *xxmap*-Anwendung). Dadurch wird weniger Entwicklungsaufwand benötigt.
Die Alternative zum GTM besteht darin, direkt Einträge in der Objekt-Datenbank der Plattform zu erzeugen und die Modelle mit dem End-User-API grafisch darzustellen. Dies ist zwar aufwendiger, erlaubt aber mehr Flexibilität bei der Darstellung als der GTM.
Der GTM wurde verwendet, um die in Kapitel 6 beschriebene Discovery-Anwendung für CORBA-Ressourcen zu implementieren.

3.4 Zusammenfassung

NetView ist eine Managementplattform, deren Infrastruktur die Kommunikation mit Internet- und OSI-Managementobjekten ermöglicht. Basisanwendungen sind aber nur für Internet-Management vorhanden. Aus diesem Grund ist die Plattform als Werkzeug für OSI-Management nur sehr begrenzt einsetzbar. Hierzu kann die *TMN WorkBench and Support-Facility* von IBM verwendet werden. Es handelt sich dabei um eine Plattform für das Management von Netzen nach den *Telecommunications Management Network* Standards, die sich wiederum an OSI orientieren.
Die Funktionalität der TMN-Plattform ist zu umfangreich, als daß sie hier beschrieben werden könnte.[12] Es sei jedoch erwähnt, daß sie teilweise (Benutzeroberfläche und Informationsbaustein) in NetView integriert ist.

[12] Für eine Einführung vgl. [IBM 95d].

Teil II

Ein Konzept zur Anbindung von Object Request Brokern an eine Netzmanagementplattform

Kapitel 4

Definition der Vorgehensweise

Dieses Kapitel nennt zunächst die grundlegenden Anforderungen für die Anbindung eines Object Request Brokers an eine Managementplattform. Danach werden unterschiedliche Lösungsansätze beschrieben und bewertet und schließlich werden die konkreten Teilaufgaben für den gewählten Lösungsweg herausgearbeitet.

4.1 Anforderungen

Durch die Anbindung eines Object Request Brokers an eine Managementplattform soll erreicht werden, daß die Plattform als Werkzeug für folgende grundlegende Aufgabe eingesetzt werden kann:

- Ressourcen eines verteilten Systems (Anwendungskomponenten, Drucker, Benutzer eines Rechners etc.) werden durch CORBA-Objekte repräsentiert.
- Managementanwendungen überwachen und steuern den Zustand der Objekte und somit auch der Ressourcen.

Um diese Zielsetzung zu erfüllen, müssen folgende Voraussetzungen erfüllt sein.

1. Es muß eine Kommunikationsmöglichkeit vorhanden sein, über die Managementanwendungen auf CORBA-Objekte zugreifen können.
2. In der Plattform muß eine Informationsbasis vorhanden sein, in der ein Modell des zu überwachenden Systems und der darin enthaltenen Ressourcen gespeichert ist.
3. Es muß eine Möglichkeit gefunden werden, wie die grundlegenden Basisdienste der Plattform für das Management von CORBA-Objekten eingesetzt werden können.

4.2 Alternative Lösungsansätze

Der Object Request Broker ermöglicht den Zugriff auf alle CORBA-Objekte, die bei ihm registriert sind (die Registrierung erfolgt durch Erzeugen einer Objektreferenz). Client-Anwendungen können über die Aufrufschnittstellen des ORBs (statisch oder dynamisch - vgl. Kapitel 2) die Methoden der Objekte aufrufen.

Eine Managementanwendung (die eventuell selbst aus verteilten Objekten besteht) kann also auf jedes Managementobjekt zugreifen, wenn sie eine Referenz für dieses Objekt hat. Die möglichen Informationsquellen, um vorhandene Objektreferenzen zu ermitteln, werden in Kapitel 6 beschrieben.
Eine Anwendung, die auf CORBA-Objekte zugreifen soll, kann in jeder Programmiersprache implementiert werden, für die der verwendete ORB eine IDL-Sprachabbildung anbietet. Dabei spielt es keine Rolle, in welcher Sprache die Objekte implementiert sind.
Eine Forderung, die an die Kommunikationsschnittstelle von Managementplattformen häufig gestellt wird, ist, daß sie ein protokollunabhängiges API für den Zugriff auf Managementobjekte zur Verfügung stellen soll (vgl. z.B. [HEGE 93 S.305]). Unter diesem Gesichtspunkt wurde von X/Open das XMP-API (X/Open Management Protocols Application Program Interface) standardisiert (vgl. [X/Open 94]). XMP kann als Programmierschnittstelle für CMIP und SNMP verwendet werden. Es bietet dabei ein gewisses Maß an Transparenz bezüglich der verwendeten Managementprotokolle, da für die Protokolloperationen teilweise die gleichen Funktionsaufrufe verwendet werden können. Die Transparenz ist jedoch nur begrenzt, da je nach Protokoll unterschiedliche Parameter eingesetzt werden müssen. Die Parameter der XMP-Funktionsaufrufe werden mit XOM (X/Open OSI-Abstract-Data-Manipulation API) erzeugt, einem API für die Repräsentation von ASN.1-Datentypen in C.
Prinzipiell ist es denkbar, einen ORB direkt in die Kommunikationsinfrastruktur einer Managementplattform zu integrieren und das XMP-API als „protokollunabhängige“ Programmierschnittstelle für CMIP, SNMP und CORBA zu verwenden. Im folgenden wird kurz beschrieben, welche Schritte dafür nötig sind. Anschließend wird die Alternative bewertet und begründet, warum sie nicht implementiert wurde.
Die Kommunikationsinfrastruktur von NetView für AIX besteht aus folgenden Komponenten (vgl. Kapitel 3):

- Der Postmaster-Dämon enthält einen SNMP- und einen CMOT-Stack und vermittelt die PDUs zwischen den Managementanwendungen und den Agenten.
- Die ORS-Datenbank (Object Registration Services) ist ein Verzeichnis von bekannten Agenten und den von ihnen verwalteten OSI-Managementobjekten.

Die Integration des ORBs unter das XMP-API müßte folgende Schritte beinhalten:

- Das XMP/XOM-API wird verwendet, um Methoden auf CORBA-Objekten aufzurufen. Dazu sind eine *CORBA-Management-Service-Package* und diverse *Management-Contents-Packages* nötig, die nach XOM konvertierte IDL-Datentypen beinhalten. Die *Management-Contents-Packages* können durch einen Compiler generiert werden, der eine IDL-XOM-Sprachabbildung durchführen kann.
- Der Postmaster wandelt die für CORBA-Objekte bestimmten XMP-Funktionsaufrufe in CORBA-Methodenaufrufe um (dafür ist es sinnvoll, die dynamische Aufrufschnittstelle zu verwenden, weil sonst für jede Objektklasse ein Stub benötigt wird).
- Die ORS-Datenbank kann verwendet werden, um Adressierungsinformation für CORBA-Objekte zur Verfügung zu stellen. Da die „CORBA-Adresse“ eines Objekts seine Referenz ist, ist hier eine Abbildung zwischen Objektreferenzen (z.B. in String-Form) und benutzerfreundlichen Namen denkbar (also eine Funktionalität, die der eines Name-Service gleicht).

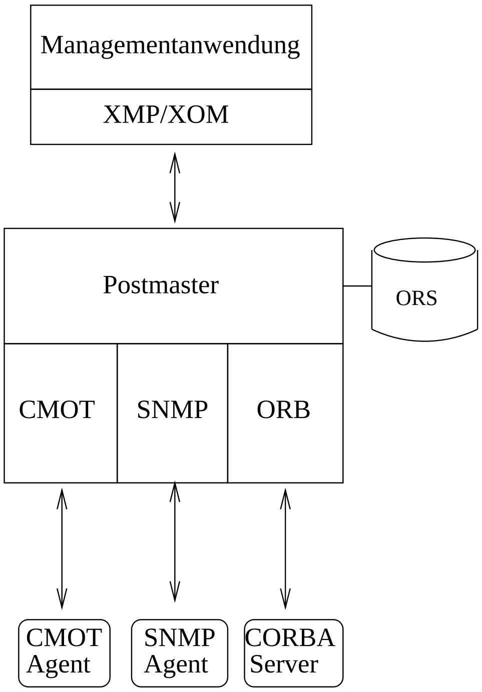

Abbildung 4.1: Integration des ORBs in die Kommunikationsinfrastruktur

Folgende Argumente sprechen gegen diesen Lösungsansatz:

- Die Entwicklung von Managementanwendungen mit XMP/XOM ist kompliziert und umständlich. Aus diesem Grund wird das API auch für die Kommunikation mit SNMP-Agenten in der Praxis höchst selten verwendet. Der Vorteil, den XMP/XOM bietet (ein gewisses Maß an Transparenz) kann den Nachteil der komplizierten Anwendungsentwicklung nicht ausgleichen.[1]

- Der Empfang und die Weiterleitung von CORBA-Ereignismeldungen durch den Postmaster ist, wenn überhaupt, nur schwer realisierbar. Im Gegensatz zu SNMP und CMIP ist eine CORBA-Ereignismeldung kein Protokollelement mit definierter PDU-Struktur, sondern kann ein beliebiger Methodenaufruf auf einem Consumer-Objekt sein (vgl. [OMG 93c], Kapitel 5). Daraus folgt, daß der Code des Postmasters jedesmal erweitert werden müßte, wenn neue Ereignismeldungen definiert werden.

- Die gesamte Integration kann nur durch Erweiterung des Quellcodes von NetView vorgenommen werden. Es müssen zumindest folgende Programmteile erweitert werden:
  - Der Postmaster, um XMP-Funktionsaufrufe in CORBA-Methodenaufrufe umzuwandeln und um CORBA-Ereignismeldungen zu empfangen und diese an den Filterprozeß weiterzuleiten.

[1] XMP hat seine „Existenzberechtigung“ momentan dadurch, daß es das einzige standardisierte API für CMIP ist. Da jedoch ein neues C++ API für CMIP standardisiert wird (vom Network Management Forum und X/Open [NMF 96]) ist es nicht unwahrscheinlich, daß XMP/XOM bald nicht mehr „aktuell“ sein wird. Dies ist ein weiteres Gegenargument dafür, XMP/XOM als API für den ORB zu verwenden.

- Der Filterprozeß (ovesmd), um CORBA-Ereignismeldungen empfangen und verarbeiten zu können, was besonders schwierig ist, weil diese kein vordefiniertes Format haben.
- Der Log-Manager (ovelmd), um CORBA-Ereignismeldungen speichern zu können.
- Die Zugriffsmechanismen des Postmasters auf die ORS-Datenbank, um sie für die Adressierung von CORBA-Objekten verwenden zu können.

Aufgrund der genannten Nachteile wurde dieser Lösungsansatz nicht realisiert. Statt dessen wurde zur Lösung der Aufgabenstellung folgendes Prinzip angewendet (vgl. Abb. 4.2):

- Anwendungen zum Überwachen und Steuern von CORBA-Objekten werden grundsätzlich als normale CORBA-Anwendungen implementiert, die über den ORB auf die entsprechenden Objekte zugreifen. Zur Implementierung kann jede Programmiersprache verwendet werden, für die der verwendete ORB eine IDL-Sprachabbildung anbietet.
- Als Schnittstelle zur Managementplattform wurden Adapter-Objekte entwickelt, die die Dienste der Plattform (Topologiemanagement und Ereignisverarbeitung) kapseln und somit für CORBA-Anwendungen zugänglich machen.

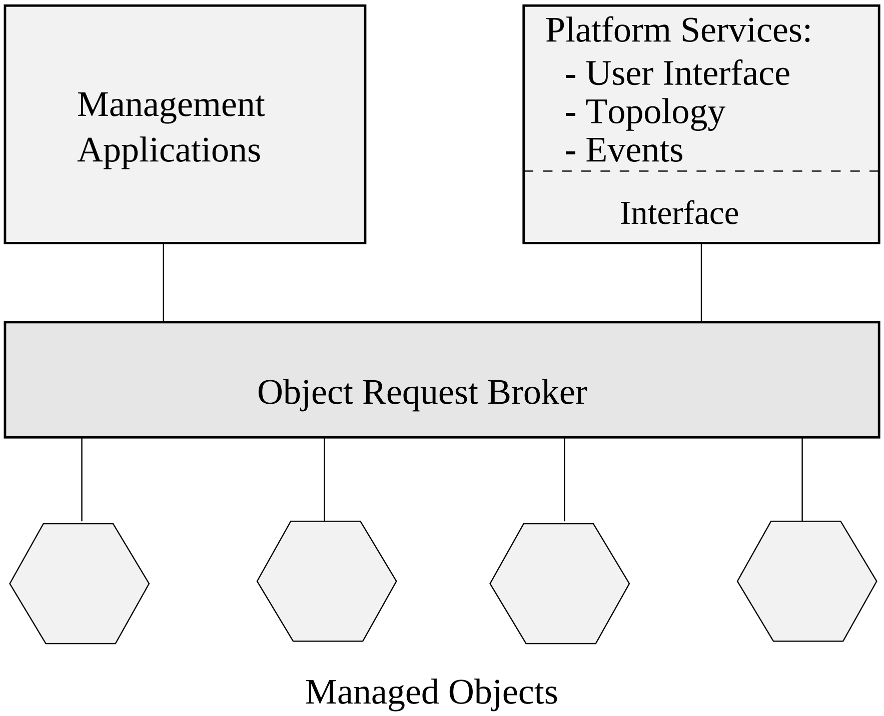

Abbildung 4.2: Konzept zur Lösung der Aufgabenstellung

Als alternativen Lösungsansatz für das Problem der heterogenen Managementarchitekturen und der protokollunabhängigen Programmierschnittstelle sei auf die Arbeiten der JIDM-Gruppe (vgl. Kapitel 2, [X/Open 95a]) verwiesen. Dort wird die Interoperabilität zwischen Managementarchitekturen durch Protokollgateways (CORBA, CMIP, SNMP) angestrebt.

Durch solche Gateways wird das Problem der Programmierschnittstelle für Managementanwendungen zumindest in gleichem Maße wie durch XMP/XOM gelöst.

4.3 Konkretisierung der Teilaufgaben

Mit dem gewählten Lösungsansatz ergeben sich folgende Teilaufgaben:

4.3.1 Erzeugen einer Informationsbasis

Neben einer Kommunikationsschnittstelle benötigen Managementanwendungen Information über die zu verwaltenden Ressourcen. Hierbei können zwei Aspekte unterschieden werden.

- Information über die vorhandenen Ressourcen. Hierzu muß in den Datenbanken der Plattform ein abstraktes Modell des zu überwachenden Systems gespeichert sein und aktuell gehalten werden. Um dieses Modell zu erstellen, wurde eine Reihe von Mechanismen entwickelt, die in Kapitel 6 beschrieben werden.

- Information über die managementrelevanten Eigenschaften der Ressourcen. Im Fall von CORBA-Managementobjekten befindet sich diese Information in der IDL-Beschreibung der Objektklassen. Zur Gewinnung der Information gibt es zwei Möglichkeiten:

  - Es ist schon zur Übersetzungszeit einer Anwendung bekannt, mit welchen Objekten sie kommunizieren wird und sie beinhaltet daher aus den entsprechenden IDL-Beschreibungen erzeugte Stubs.
  - Zur Laufzeit kann die Information über Objektklassen aus dem Interface-Repository gewonnen werden und es können dynamische Methodenaufrufe ausgeführt werden.

4.3.2 Integration der Basisdienste

Es muß eine Möglichkeit gefunden werden, wie die Basisdienste der Plattform für das Management von CORBA-Objekten eingesetzt werden können. Es wurden zwei Dienste identifiziert, für die eine Integration möglich ist:

- Die Verarbeitung von Ereignismeldungen. Hierzu zählt das Filtern und Weiterleiten an bestimmte Anwendungen, das Speichern in Log-Dateien und das Anzeigen an der Benutzeroberfläche. Um diese Dienste für CORBA-Ereignismeldungen nutzen zu können, wurden die in Kapitel 5 beschriebenen Verfahren und Schnittstellen entwickelt.

- Die Darstellung der vorhandenen Ressourcen und ihrer Beziehungen zueinander (Topologiemanagement) an der Oberfläche der Plattform. Hinzu kommt die Visualisierung von Zustandsänderungen durch Farbänderung von Symbolen. Voraussetzung dafür ist, daß in den Datenbanken der Plattform ein Abbild des zu verwaltenden Systems gespeichert ist. Die Mechanismen, die entwickelt wurden, um die Visualisierungsdienste von NetView für die Darstellung der Ressourcen einer CORBA-Umgebung nutzen zu können, werden in Kapitel 6 beschrieben.

Bei diesen beiden Diensten kann eine Integration bzw. eine Anbindung in dem Sinne stattfinden, daß ein relevanter Anteil an vorhandenem „Plattformcode" zur Lösung der Aufgabenstellung verwendet werden kann. Bei anderen NetView-Basisdiensten wie z.B. dem MIB-Browser oder dem Leistungsmonitor (snmpCollect) besteht hingegen keine Möglichkeit, sie für CORBA-Objekte einzusetzen, denn es sind speziell für SNMP geschriebene Anwendungen. Programme, die eine ähnliche Funktionalität für CORBA erbringen, müßten vollständig neu implementiert werden. Man kann sie allerdings in die NetView-Oberfläche integrieren und sie mit Schnittstellen zur Plattformdatenbank versehen.

Kapitel 5

Integration des Ereignismanagements

5.1 Anforderungen

Wie in Kapitel 4 gezeigt wurde, besteht ein wichtiger Aspekt der Anbindung des ORBs an die Plattform darin, daß CORBA-Ereignismeldungen von der Plattform empfangen und verarbeitet werden können. Dadurch können die Dienste der Plattform wie Filtern, Speichern in Log-Dateien und Anzeigen an der Bedienoberfläche genutzt werden.

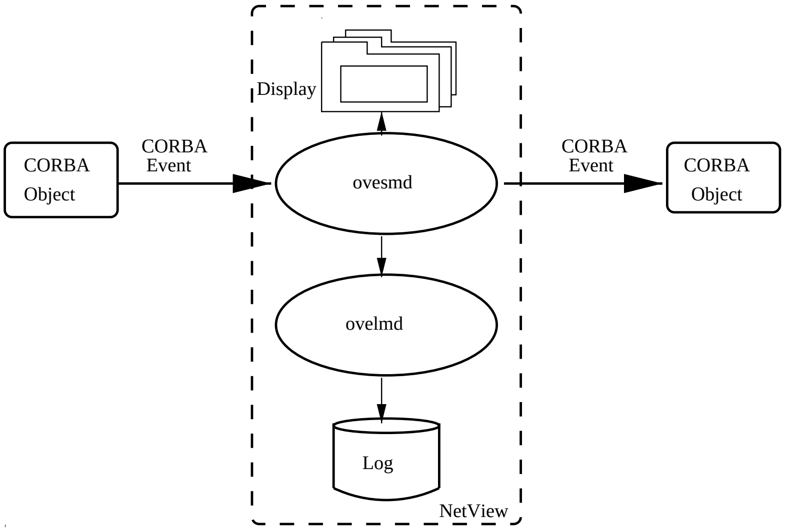

Abbildung 5.1: Verarbeitung von CORBA-Events durch die Plattform

Für das Ereignismanagement werden zwei Schnittstellen zwischen der CORBA-Umgebung und der Managementplattform benötigt:

- Eine Schnittstelle für den Empfang und die Verarbeitung von CORBA-Ereignismeldungen durch die Event-Management-Prozesse der Plattform (vgl. Kapitel 3). Dadurch können sie gefiltert, gespeichert und an der Oberfläche angezeigt werden.

- Darüberhinaus wird eine Schnittstelle benötigt, über die CORBA-Anwendungen sich für den Empfang von gefilterten Ereignismeldungen bei der Plattform registrieren können.

NetView ist nicht in der Lage, CORBA-Ereignismeldungen direkt zu empfangen und zu verarbeiten. Es wurde daher eine Lösung entwickelt, die auf Adapter-Objekten und Event-Gateways basiert.
Das Konzept zur Lösung der Anforderungen beruht auf der Annahme, daß das Erzeugen und Versenden von CORBA-Ereignismeldungen gemäß der *Event Service Specification* [OMG 93c] erfolgt. Das SOM Event Management Framework ist nicht konform zu diesem Standard und wurde deshalb nicht verwendet. Der CORBA-Event-Service wird im folgenden Abschnitt beschrieben. Zum Vergleich mit dem SOM-Framework sei auf Kapitel 2 verwiesen. Der Hauptunterschied besteht darin, daß das SOM-Framework keine Event-Channel-Objekte und keine typisierten (d.h. mit beliebigen IDL-Datentypen als Parameter) Ereignismeldungen kennt.

5.2 Der CORBA-Event-Service

Dieser Abschnitt gibt einen Überblick über die Komponenten des CORBA-Event-Service. Zur genaueren Betrachtung sei auf [OMG 93c] verwiesen.
Der CORBA-Event-Service ermöglicht asynchrone Kommunikation von Objekten mittels Ereignismeldungen. Dabei werden zwei unterschiedliche Rollen für Objekte definiert: Supplier und Consumer. Supplier erzeugen Ereignismeldungen und versenden sie an Consumer. Consumer empfangen die Ereignismeldungen und verarbeiten sie weiter. Die Zuordnung von Supplier und Consumer erfolgt durch Austauschen von Objektreferenzen.

5.2.1 Push- und Pull-Kommunikation

Es gibt zwei unterschiedliche Kommunikationsmechanismen für Ereignismeldungen: Das *Push*-Modell und das *Pull*-Modell.
Beim *Push*-Modell benachrichtigt der *PushSupplier* den *PushConsumer* von sich aus beim Eintreten eines Ereignisses. Er tut dies, indem er die Methode *push* des *PushConsumer*-Objekts aufruft.
Beim *Pull*-Modell geht die Initiative von Consumer aus. Der *PullConsumer* verlangt die Übermittlung einer Ereignismeldung, indem der die Methode *pull* des *PullSuppliers* aufruft.

5.2.2 Event-Channels

Event-Channels sind Objekte, die zur Entkopplung der Kommunikation zwischen Supplier- und Consumer-Objekten eingesetzt werden. Supplier versenden Ereignismeldungen an einen Event-Channel, der sie wiederum an Consumer weiterleitet. Der Event-Channel selbst hat damit sowohl die Rolle eines Consumers als auch eines Suppliers.
Über einen Event-Channel können beliebig viele Supplier und Consumer ohne direkte Interaktion kommunizieren. Für die Kommunikation zwischen einem Supplier und einem Event-Channel bzw. einem Consumer und einem Event-Channel kann das Push-Modell oder das Pull-Modell angewendet werden.
Die IDL-Definition eines Event-Channels besteht aus folgenden Methoden:

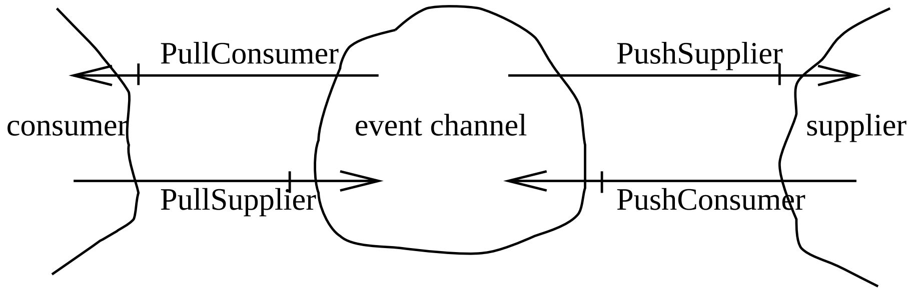

Abbildung 5.2: Funktionsweise von Event-Channels

```
interface EventChannel {
  ConsumerAdmin for_consumers();
  SupplierAdmin for_suppliers();
  void destroy();
};
```

Die `for_consumers`-Methode liefert ein `ConsumerAdmin`-Objekt. Dieses Objekt kann von Consumern verwendet werden, um sich selbst beim Event-Channel zu registrieren. Die Klasse `ConsumerAdmin` hat folgende IDL-Definition:

```
interface ConsumerAdmin {
  ProxyPushSupplier obtain_push_supplier();
  ProxyPullSupplier obtain_pull_supplier();
};
```

Die Methode `obtain_pull_supplier` liefert ein Proxy-Objekt für einen PullSupplier (der eigentliche Supplier ist der Event-Channel). Von diesem Objekt kann der Consumer durch Aufruf der `pull`-Methode die Übermittlung einer Ereignismeldung anfordern.
Analog erfolgt die Registrierung von Suppliern beim Event-Channel. Die Methode `destroy` des Event-Channels dient zum Zerstören des Objekts.
In einer CORBA-Umgebung können beliebig viele Event-Channel-Objekte existieren. Die Hauptfunktion des Event-Channels ist die Entkopplung von der direkten Interaktion zwischen Supplier und Consumer. Weitere Funktionalität, wie z.B. Filtern oder Speichern von Ereignismeldungen, ist implementierungsabhängig und wird im Event-Service nicht spezifiziert. Es werden nur die nach außen sichtbaren Schnittstellen eines Event-Channel-Objekts definiert.

5.2.3 Generische und typisierte Kommunikation

Zusätzlich zur Differenzierung zwischen *Push-* und *Pull-*Modell wird beim CORBA-Event-Service zwischen generischer und typisierter Kommunikation von Ereignismeldungen unterschieden.

Bei der generischen Kommunikation haben die *Push-* und *Pull*-Methoden der Supplier- bzw. Consumer-Objekte jeweils nur einen Parameter vom Datentyp *any*, in dem alle Daten der Ereignismeldung verpackt werden.
Bei der typisierten Event-Kommunikation erfolgt die Übermittlung von Ereignismeldungen durch Methodenaufrufe auf Objekten, die eine „anwendungsspezifische" IDL-Definition haben können[1]. Für die Methoden dieser Objekte werden folgende Beschränkungen gefordert (vgl. [OMG 93c S. 17f.]):

- *TypedConsumer:*
  - es sind nur *in*-Parameter zulässig
  - es sind keine Rückgabewerte erlaubt
- *TypedSupplier:*

  Ein TypedSupplier-Interface wird aus einem bestehenden TypedConsumer-Interface erzeugt. Dabei werden folgende Abwandlungen der Methoden vorgenommen:

  - Für jede Methode *X* wird eine Methode *pull_X* definiert, bei der alle *in*-Parameter in *out*-Parameter umgewandelt werden.
  - Für jede Methode *X* wird eine Methode *try_X* definiert, bei der alle *in*-Parameter in *out*-Parameter umgewandelt werden, und die entweder *true* oder *false* als Rückgabewert hat, je nachdem, ob der Supplier eine Ereignismeldung vorrätig hat oder nicht.

Die Übermittlung von Ereignismeldungen erfolgt durch den Aufruf der Methoden, die die Objekte zur Verfügung stellen.
Die Zuordnung zwischen Suppliern und Consumern ist bei der typisierten Kommunikation komplizierter als bei der generischen. Folgende Schritte sind z.B. für die Registrierung eines *PullConsumers* bei einem *TypedEventChannel* nötig (vgl. Abb. 5.3):

- Der PullConsumer ruft die Methode *for_consumers* des *TypedEventChannel* auf und erhält daraufhin eine Referenz auf ein Objekt der Klasse *TypedConsumerAdmin*.
- Auf diesem Objekt ruft er dann die Methode *obtain_typed_pull_supplier* auf. Diese Methode hat einen Parameter *supported_interface* vom IDL-Typ *Key*. Über diesen Parameter gibt der Consumer die gewünschte Aufrufschnittstelle des *TypedProxyPullSuppliers* an, dessen Referenz er durch den Methodenaufruf erhält.
- Durch Aufruf der Methode *get_typed_supplier* des *TypedProxyPullSupplier* erhält der PullConsumer eine Referenz auf ein Objekt, auf dem er die entsprechenden *pull*-Methoden für die Übermittlung der Ereignismeldungen aufrufen kann.

5.3 Lösungskonzept

In diesem Abschnitt wird das generelle Konzept für die Verarbeitung von CORBA-Ereignismeldungen durch die Managementplattform NetView vorgestellt. Eine Beschreibung der Implementierung erfolgt im nächsten Abschnitt.

[1] Die Vorteile des typisierten Modells werden in 5.3.1 beschrieben.

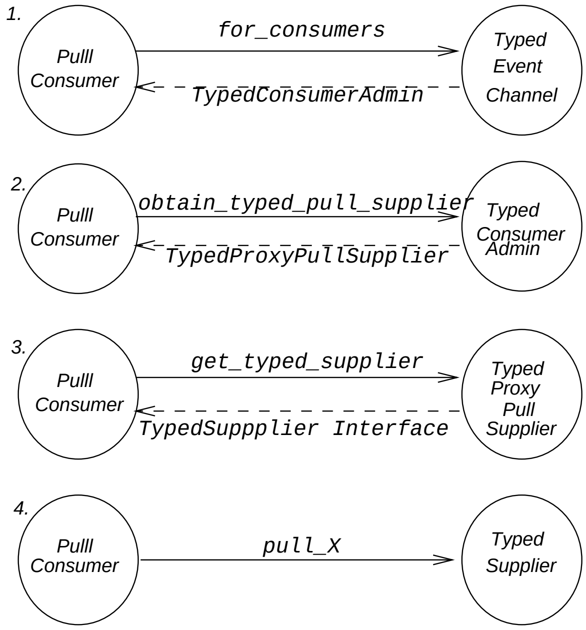

Abbildung 5.3: Registrierung bei einem TypedEventChannel

Damit eine Kommunikation zwischen Suppliern von Systems-Management-Events und der Managementplattform stattfinden kann, muß die Plattform aus CORBA-Sicht die Rolle eines Consumers einnehmen. Außerdem soll sie die Supplier-Rolle für gefilterte Ereignismeldungen übernehmen können.
Als Voraussetzungen für die Entwicklung des Lösungskonzepts wird zunächst definiert, welche der im CORBA-Event-Service definierten Kommunikationsmechanismen für die Übermittlung von Systems-Management-Events sinnvoll sind und welche Rolle Event-Channels bei der Lösung der Anforderungen spielen können.

5.3.1 Systems-Management-Events

Vorteile typisierter Event-Kommunikation

Bei der generischen Event-Kommunikation besteht die Übermittlung einer Ereignismeldung darin, daß ein Push-Supplier die Methode *push* eines Push-Consumers aufruft, für den er eine Objektreferenz hat. Wenn das Pull-Modell angewendet wird, ruft der Pull-Consumer die Methode *pull* des Pull-Suppliers auf.
Diese beiden Methoden haben jeweils nur einen einzigen Parameter vom IDL-Datentyp *any*. Diese Methode ist nicht differenziert genug, wenn man unterschiedliche Arten von Ereignismeldungen versenden und empfangen will. So muß z.B. ein generischer PushConsumer alle Daten akzeptieren, die ihm als Parameter beim Aufruf der *push*-Methode übergeben werden. Für Systems-Management-Events ist das Modell der typisierten Kommunikation aus folgenden Gründen besser geeignet:

- Systems-Management-Events können abgegrenzt werden von anderen Ereignismeldungen wie z.B. Operating-System-Events oder Groupware-Events.

- Die Systems-Management-Events können ihrerseits untergliedert werden in beispielsweise Fault-Events, Topology-Events, Security-Events etc. Es können spezielle TypedSupplier und TypedConsumer für die unterschiedlichen Arten von Ereignismeldungen spezifiziert werden.
- Dadurch, daß eine Ereignismeldung durch einen Methodenaufruf realisiert wird, und daher einen Namen und genau definierte Parameter hat, besteht die Möglichkeit, Rückschlüsse auf ihre Bedeutung zu ziehen, was bei der allgemeinen Push-Methode nicht der Fall ist.

Gründe für den Einsatz von Event-Channels

Der CORBA-Event-Service schreibt die Verwendung von Event-Channel-Objekten nicht vor. Man könnte also Ereignismeldungen direkt „an die Plattform" verschicken. Dennoch ist es aus folgenden Gründen sinnvoll, für das Versenden von Systems-Management-Events einen oder mehrere spezielle Event-Channels zu verwenden:

- Supplier, die Ereignismeldungen verschicken wollen, brauchen nur eine Objektreferenz für den Event-Channel. Sie verschicken alle Ereignismeldungen an den Event-Channel. Es ist transparent für sie, ob die Ereignismeldungen von der Managementplattform oder von anderen Consumern empfangen werden.
- An einen Event-Channel können sich beliebig viele Consumer-Objekte anschließen, um Ereignismeldungen zu empfangen. Auf der Seite der Managementplattform kann es mehrere Consumer für unterschiedliche Event-Arten geben. Es können sich auch Consumer, die zu verschiedenen Managementplattformen gehören, an einen Event-Channel anschließen. Bei der direkten Kommunikation ohne Event-Channel müßte ein Client eine Ereignismeldung an jeden Consumer einzeln verschicken.
- Bei der Kommunikation zwischen Suppliern und dem Event-Channel und zwischen Consumern und dem Event-Channel kann zwischen dem Pull- und dem Push-Modell ausgewählt werden. Es ist z.B. denkbar, daß die Supplier alle Ereignismeldungen durch Push-Methoden an den Event-Channel übertragen. Der Event-Channel leitet aber nur besonders kritische Ereignismeldungen durch Push-Kommunikation an die Plattform weiter. Weniger kritische holt die Plattform sich nur dann aus dem Event-Channel (durch Pull-Methoden), wenn sie freie Verarbeitungskapazität hat.
- Durch entsprechende Implementierung besteht die Möglichkeit, Managementfunktionalität in den Event-Channel zu verlagern. Denkbar ist z.B. eine Event-Channel-Implementierung mit Filter- und Log-Möglichkeiten, um die Plattform bei der Verarbeitung der Events zu entlasten.
- Es kann eine Zuordnung von Event-Channels zu verschiedenen Managementdomänen stattfinden. Beispiele für Domänen können Managementszenarien (Anwendungs-, Netz- oder Systemmanagement), Funktionsbereiche (Konfigurations-, Leistungs- oder Fehlermanagement, etc.) oder geographische Regionen bzw. Verwaltungsbereiche sein. Die vorhandenen Event-Channels können in einem Name-Service registriert sein, um sie für Supplier und Consumer (Managementplattformen oder andere) zugänglich zu machen.

Ein Aspekt, der beim Einsatz von Event-Channel-Objekten zu berücksichtigen ist, besteht in der Zuordnung von Suppliern, Consumern und Event-Channels. Wenn es, wie oben angedeutet, mehrere Event-Channels gibt, könnte z.B. folgendes Szenario auftreten:
Eine Managementanwendung will alle Security-Events aus der Managementdomäne X (z.B. ein bestimmter Verwaltungsbereich) empfangen. An welche Event-Channels muß sie sich „anschließen“ um diese Ereignismeldungen zu empfangen ?
Bei der Implementierung wurde dieses Problem dadurch gelöst, daß alle managementrelevanten Ereignismeldungen an *einen* bestimmten Event-Channel geschickt und von da an die Plattform weitergeleitet werden. Managementanwendungen können sich dann bei der Plattform für den Empfang von bestimmten gefilterten Ereignismeldungen registrieren.
Ein ähnliches Problem tritt auch bei der Zuordnung von Suppliern zu Event-Channels auf. Woher weiß ein Supplier, der Ereignismeldungen an bestimmte Consumer verschicken will, an welchen Event-Channel er sie weiterleiten soll ?
Als Lösungsansatz für diese Problematik kann der Einsatz von *Trader*-Diensten (vgl. dazu Kapitel 6) dienen. Event-Channels können sich mit einer Beschreibung des von ihnen angebotenen Dienstes (z.B. Namen (Name-Service) der registrierten Supplier und Consumer, unterstützte Ereignismeldungen bei *TypedEventChannels* , „Quality of Service“ etc.) beim Trader registrieren. Supplier und Consumer können dann den Trader nutzen, um die für ihre Anforderungen geeigneten Event-Channels zu finden.

5.3.2 Verarbeitung von Ereignismeldungen durch die Plattform

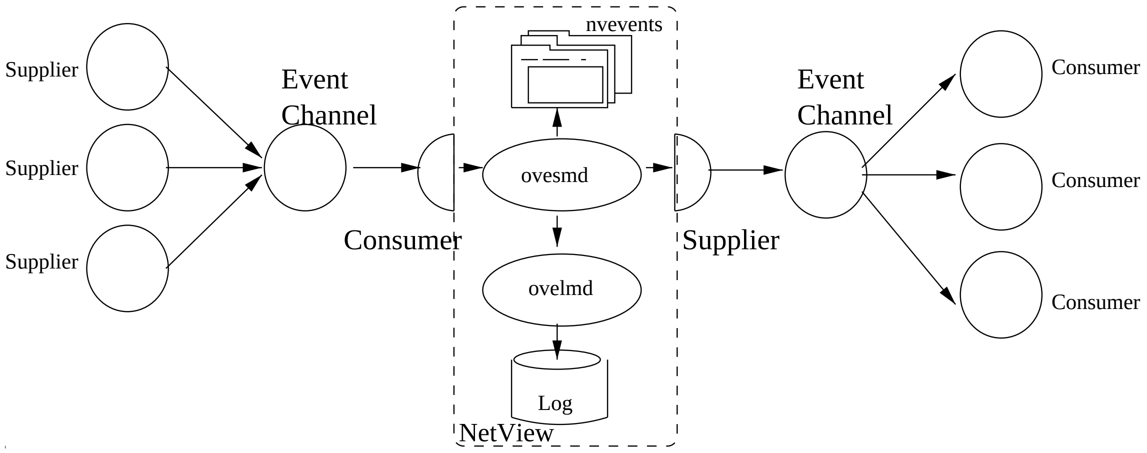

Abbildung 5.4: Konzept für das Event-Management

Um die Event-Management-Dienste der Plattform für CORBA-Ereignismeldungen verwenden zu können, werden folgende Schnittstellen benötigt (vgl. Abb. 5.4):

- Consumer-Objekte, die die Systems-Management-Events von einem oder mehreren Event-Channels empfangen und sie an die Plattform weiterleiten. Dabei müssen die Ereignismeldungen in ein „Format“ umgewandelt werden, das von den Prozessen der Plattform verarbeitet werden kann. Die Alternativen dazu sind SNMP-Traps oder CMOT-Notifications. Eine vergleichende Bewertung dieser beiden Möglichkeiten erfolgt im nächsten Abschnitt.

- Supplier-Objekte, die von der Plattform gefilterte Events an dafür registrierte Consumer weiterleiten. Hierbei können wiederum Event-Channels als „Vermittler“ zwischen einem Supplier und mehreren Consumern eingesetzt werden. Diese Supplier-Objekte empfangen von dem Filterprozeß der Plattform SNMP-Traps bzw. CMOT-Notifications (die aus CORBA-Events erzeugt wurden), wandeln sie um in CORBA-Events und übergeben sie an den entsprechenden Event-Channel. Es gibt für jeden aktiven Filter (SNMP/CMOT) ein Supplier-Objekt. Dieses hat eine ähnliche Funktionalität wie ein Event-Forwarding-Discriminator in der OSI-Welt.

Um die benötigten Schnittstellen zur Verfügung zu stellen, wurden die im nächsten Abschnitt beschriebenen Komponenten entwickelt.

5.4 Implementierung

5.4.1 IDL-Definitionen für Systems-Management-Events

Aus den bereits genannten Gründen ist es sinnvoll, für Systems-Management-Events das Modell der typisierten Event-Kommunikation zu verwenden. Es gibt zwei Alternativen für die Modellierung von Ereignismeldungen:

- als eigene *Objekte*
- als *Operationen* von Supplier- bzw. Consumer-Objekten

Im Rahmen der Implementierung wurden beide Möglichkeiten getestet. Es wurden Ereignismeldungen zur Aktualisierung der Topologiedatenbank der Plattform entwickelt, die zum Beispiel von den Server-Objekten oder dem Implementation-Repository ausgesendet werden (vgl. Kapitel 6).

Ereignismeldungen als Objekte

Der Unterschied der beiden Modellierungsarten sei am folgenden Beispiel verdeutlicht. Es handelt sich um eine Ereignismeldung, die das Erzeugen eines Objekts in einem Server anzeigt:

```
interface object_created : generic_event {
  attribute string server_id;
  attribute string object_reference;
  attribute string object_class;

  void notify(...,
   in string server_id,
   in string object_reference,
   in string object_class);
};
```

Die Klasse *object_created* ist eine Unterklasse von *generic_event* und erbt von dieser die allgemeinen Attribute einer Ereignismeldung wie z.B. Event-Name, Zeitpunkt, Name des Quell-Rechners, etc.[2] Diese werden um weitere spezifische Attribute ergänzt: der Implementation-

[2] Einen Ansatz für die Definition eines allgemeinen Formats für Systems-Management-Events findet man in [X/OPEN 94b].

Repository-Identifikator des Servers, die Objektreferenz des neu erzeugten Objekts und der Name seiner Klasse.
In der Implementierung der Methode *notify* erfolgt die Verarbeitung der Ereignismeldung. Die Parameter der Methode entsprechen den Attributen der Event-Klasse (geerbte und eigene Attribute).
Bei diesem Modell erfolgt das Versenden einer Ereignismeldung (Push-Modell) in folgenden Schritten:

- Der Client, der die Ereignismeldung versenden will (beim obigen Beispiel das Server-Objekt, das das neue Objekt erzeugt hat), erhält eine Referenz (ein Proxy-Objekt) auf ein Objekt der Event-Klasse *object_created* (z.B von einem Event-Channel).
- Der Client ruft die Methode *notify* auf mit den entsprechenden Parametern.
- Die Implementierung der *notify*-Methode verarbeitet die Ereignismeldung.

Die Instanz der Event-Klasse ist in diesem Fall das TypedConsumer-Interface. Der Vorteil der Modellierung von Ereignismeldungen als Objektklassen besteht darin, daß eine Vererbungshierarchie (Vererbung der Attribute und Überschreiben der *notify*-Methode) aufgebaut werden kann.

Ereignismeldungen als Methoden von Supplier- und Consumer-Objekten

Die zweite Alternative besteht darin, eine Ereignismeldung als Methode eines Supplier- bzw. Consumer-Objekts zu modellieren.
Es wurde eine Objektklasse *Event_consumer* definiert, deren Methoden verschiedene im Rahmen der Aufgabenstellung verwendete Ereignismeldungen darstellen.

```
interface Event_consumer : somf_MCollectible
{
  void object_created(in string ev_type,
                      in string host,
                      in string server_id,
                      in string server_alias,
                      in string objclass,
                      in string objref);

  void object_deleted(...);
  void server_added(...);
  void server_deleted(...);
  void server_active(...);
  void server_inactive(...);
  void AnException(...);
}
```

Ein Objekt, das die Event_consumer-Schnittstelle anbietet, ist ein *TypedPushConsumer* für die darin definierten Events. Ein Client kann Ereignismeldungen erzeugen, indem er die Methoden eines Event_consumer-Objekts aufruft, für das er eine Referenz hat. Die Verarbeitung der Ereignismeldungen erfolgt in der Implementierung der Methoden.

Vergleich und Bewertung

Der Vorteil der Modellierung von Ereignismeldungen als Objektklassen ist, daß eine Vererbungshierarchie erzeugt werden kann (vgl. Abbildung 5.5)[3]. Neue Ereignisklassen können durch Vererbung und Spezialisierung (durch Hinzufügen neuer Attribute) aus bereits vorhandenen erzeugt werden.

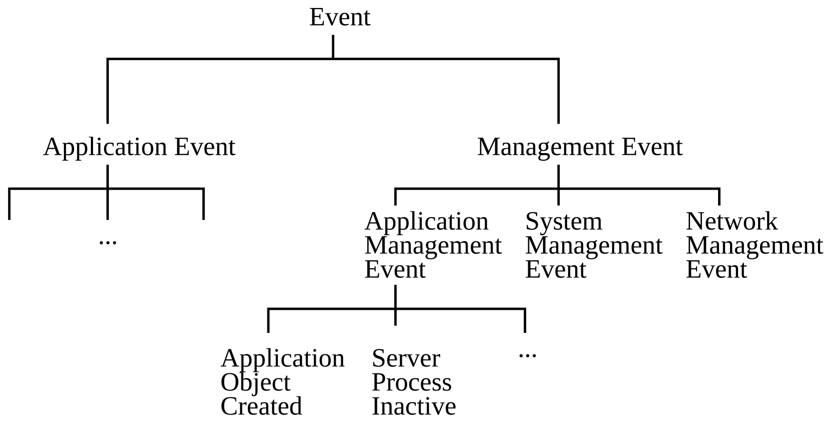

Abbildung 5.5: Vererbungshierarchie von Event-Klassen

Bei der Implementierung des Prototypen wurde allerdings die zweite Alternative gewählt, bei der jeweils mehrere Ereignismeldungen in der IDL-Definition eines Consumers zusammengefaßt werden. Dadurch wurde die Entwicklung der anstelle von Event-Channels eingesetzten Event_Dispatcher-Objekte vereinfacht. Ein Event_Dispatcher ist ein erweiterter Event_consumer, der empfangene Ereignismeldungen an andere Event_consumer weiterleitet.

5.4.2 Die Klasse Event_Dispatcher

Wie im vorigen Abschnitt erklärt wurde, besteht die Aufgabe von Event-Channel-Objekten darin, Events von Suppliern zu empfangen und sie an registrierte Consumer weiterzuleiten. Die CORBA-Event-Service Spezifikation enthält eine große Anzahl von IDL-Definitionen für die Verwaltung und Benutzung von Event-Channel-Objekten. Dadurch wird eine breite Vielfalt von Kommunikationsmodellen (Push/Pull, generisch/typisiert) ermöglicht.
Eine vollständige Implementierung der im CORBA-Event-Service definierten Schnittstellen hätte den Rahmen der Aufgabenstellung gesprengt und wurde daher nicht durchgeführt. Statt dessen wurde die Objektklasse *Event_Dispatcher* entwickelt. Ein Event_Dispatcher kann die gleiche Funktionalität erbringen wie ein Event-Channel, unter folgenden Voraussetzungen:

- Es wird typisierte Event-Kommunikation verwendet - mit definierten Supplier- und Consumer-Objekten (wie oben beschrieben).
- Es wird ausschließlich das Push-Modell verwendet.

Die Klasse Event_Dispatcher ist von der Klasse Event_consumer abgeleitet und stellt zusätzliche Methoden zum Registrieren und Deregistrieren von Consumer-Objekten bereit.

[3] Einen anderen Ansatz dafür findet man in [HOFFNER 94a].

```
interface Event_Dispatcher : Event_consumer
{
  void register_consumer (in Event_consumer consumer);
  void consumer_disconnect (in Event_consumer consumer);
};
```

Dadurch, daß der Event_Dispatcher von der Klasse *Event_consumer* abgeleitet wurde, ist er ein Push-Consumer für die darin definierten Events.
Ein Event_Dispatcher hat eine Liste der Objektreferenzen aller registrierten Consumer. Für diese Liste wurde die Collection-Class somf_TDeque verwendet (eine Liste von Objekten). Die Funktionalität eines Event_Dispatchers besteht darin, daß er jede Ereignismeldung, die er durch den Aufruf einer seiner von *Event_consumer* geerbten Methoden empfängt, an alle registrierten Consumer weiterleitet.
Der Event_Dispatcher dient somit einerseits als zentrales Consumer-Objekt für alle Supplier von Systems-Management-Events und ist andererseits der Supplier für alle seine registrierten Consumer-Objekte (vgl. Abb. 5.6).

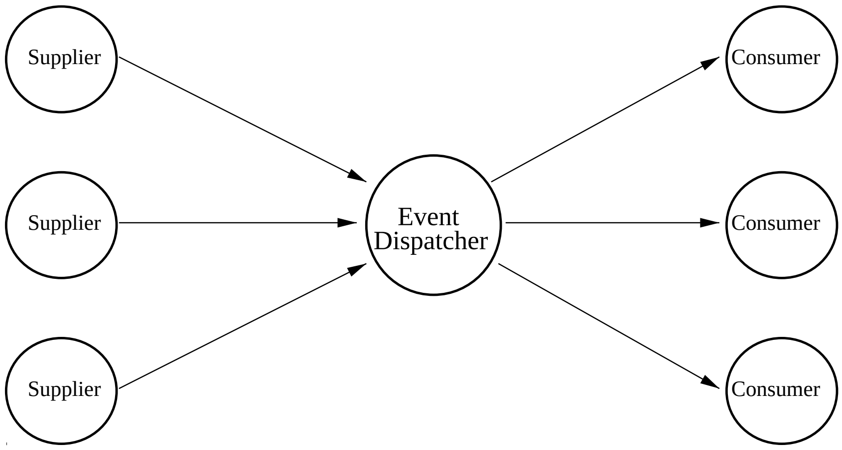

Abbildung 5.6: Die Weiterleitung von Ereignismeldungen durch Event_Dispatcher

5.4.3 Empfang von Ereignismeldungen durch die Managementplattform

Damit CORBA-Ereignismeldungen von der Managementplattform verarbeitet werden können, wurden spezielle Event-Consumer-Objekte entwickelt:

- Ein oder mehrere *EMS_Event_consumer* empfangen sämtliche managementrelevanten CORBA-Events und wandeln sie in SNMP-Traps um, damit sie von der Plattform verarbeitet werden können.

- *GTM_Event_consumer* empfangen alle topologierelevanten Ereignismeldungen (die z.B. das Erzeugen oder Löschen von Objekten anzeigen) und aktualisieren daraufhin die GTM-Datenbank der Plattform. Es wurden zwei verschiedene Topologiemodelle in der GTM-Datenbank erzeugt. Für die Aktualisierung dieser Modelle wurde jeweils ein Consumer-Objekt implementiert (vgl. Kapitel 6).

NetView kann CORBA-Ereignismeldungen nicht direkt verarbeiten. Damit trotzdem die *Event-Management-Services (EMS)* der Plattform genutzt werden können, müssen die CORBA-Events in SNMP-Traps oder CMOT-Notifications umgewandelt werden.
Bei einem Vergleich dieser beiden Alternativen werden die Vorteile von CMOT deutlich:

1. Möglicher Informationsgehalt der Ereignismeldungen.

   Die Bestandteile eines *CMIS-Event-Report-Argument* sind besser geeignet, um die Information eines CORBA-Events darin zu „verpacken“ als eine SNMP-Trap-PDU. Eine Betrachtung der beiden PDU-Typen macht dies deutlich. Ein *CMIS-Event-Report-Argument* besteht aus folgenden Komponenten (die Angabe des Typs bezieht sich auf den jeweiligen XOM-Datentyp):

   - managed Object-Class (Typ: String)
   - managed Object-Instance (Typ: Object)
   - event-Time (Typ: Object)
   - event-Info (Typ: Any)

   Eine SNMP-Trap-PDU hat neben den „üblichen“ Bestandteilen einer SNMP-PDU (request-id, error-status, error-index) folgende Komponenten:

   - enterprise (Objektidentifikator der Wurzel der herstellerspezifischen MIB im Internet-Registrierungsbaum)
   - agent-addr (IP-Adresse des sendenden Agenten)
   - generic trap
   - specific trap
   - time (Zeit seit dem letzten Aktivieren des Agenten (in Zehntelsekunden))
   - Variable Bindings (bestehend aus dem Object-Identifier der MIB-Variable und ihrem Wert)

   Die CMOT-PDU hat zwei wichtige Vorteile gegenüber einer SNMP-Trap-PDU. Der erste ist die Information über die Quelle des Events. Sowohl bei OSI als auch bei CORBA ist der Versender einer Ereignismeldung eine Objektinstanz. Die CMOT-PDU enthält sowohl die Objektklasse als auch den Distinguished-Name des sendenden OSI-Objekts. Bei einer CMOT-Notification, die aus einem CORBA-Event erzeugt wird, ist es sinnvoll, die Objektreferenz (in String-Form) als Identifikator für die Objektinstanz anzugeben. Als Identifikator für die Objektklasse kann ein eindeutiger Name oder die Interface-Repository-ID verwendet werden. Eine SNMP-Trap-PDU enthält hingegen lediglich die IP-Adresse des Hosts, auf dem sich der sendende Agent befindet, als Information über die Quelle der Ereignismeldung. Die Konsequenz daraus sind eingeschränkte Filtermöglichkeiten (s.u.).

   Der zweite Vorteil von CMOT betrifft die anwendungsspezifischen Daten, die in einer Event-Report-PDU verschickt werden können. Die *event-Info*-Komponente eines *CMIS-Event-Report-Argument* ist vom XOM-Datentyp *Any*. Dies bedeutet, daß die übergebene XOM-C-Datenstruktur einem beliebigen ASN.1-Datentyp entsprechen kann. Die Werte der Variable-Bindings einer SNMP-Trap-PDU sind eingeschränkt auf einige wenige ASN.1-Datentypen (die der Value-Komponente eines Variable-Bindings entsprechende

C-Datenstruktur des NetView-SNMP-API ist sogar nur eine union aus u_char *, long * und ObjectID *).

Da bei der Modellierung von CORBA-Ereignismeldungen durch Methodenaufrufe die Parameter beliebige IDL-Datentypen sein können, kann es bei der Umwandlung in SNMP-PDUs leicht zu Problemen kommen, weil nicht alle IDL-Datentypen durch die ASN.1-Datentypen der Internet-SMI dargestellt werden können. Dies gilt allgemein für die Umwandlung von IDL nach Internet-SMI. Aus diesem Grund wurde auch die Übersetzung von IDL nach Internet-SMI bei der Arbeit der JIDM-Gruppe ([X/OPEN 95a]) nicht definiert. Es gibt jedoch definierte Abbildungsvorschriften für IDL nach GDMO/ASN.1, die bei der Umwandlung von CORBA-Event-Parametern angewendet werden können.

2. Filtermöglichkeiten

   Auch die Filtermöglichkeiten, die das NetView-Filter-API für CMOT-Event-Report-PDUs bietet, entsprechen eher den Anforderungen bei der Filterung von CORBA-Ereignismeldungen, als die Möglichkeiten bei SNMP-Traps. Insbesondere die Filtermöglichkeiten nach der Quelle der Ereignismeldung (Objektklasse und Objektinstanz bei CMOT / IP-Adresse des sendenden Hosts bei SNMP) sind bei CMOT besser. Für einen genauen Vergleich der Filtermöglichkeiten sei auf Kapitel 3 verwiesen.

Trotz der gerade beschriebenen Vorteile von CMOT wurden bei der Implementierung des Prototypen CORBA-Ereignismeldungen aus folgenden zwei Gründen in Enterprise-Specific-Traps umgewandelt:

- NetView kann CMOT-Event-Reports im Gegensatz zu SNMP-Traps nicht an der Oberfläche anzeigen.

- Die Implementierung war einfacher, weil das SNMP-API anstelle von XMP/XOM verwendet werden konnte.

Der Empfang und die Umwandlung von CORBA-Ereignismeldungen erfolgt durch Instanzen der Klasse *EMS_Event_consumer*.
Die Klasse EMS_Event_consumer ist von Event_consumer abgeleitet. Die Methoden wurden dahingehend überschrieben, daß beim Aufruf einer Event-Methode eine entsprechende SNMP-Trap-PDU aufgebaut und an NetView geschickt wird. Voraussetzung dafür ist, daß bei der Plattform für jede CORBA-Ereignismeldung ein Enterprise-Specific-Trap definiert ist. NetView bietet die Möglichkeit, eigene Enterprise-Specific-Traps zu definieren.
Ein EMS_Event_consumer registriert sich bei seiner Initialisierung beim Event_Dispatcher und empfängt daraufhin von diesem alle managementrelevanten Ereignismeldungen. Zur Umwandlung der Ereignismeldungen in Traps und zum Versenden dieser Traps an NetView verwendet der EMS_Event_consumer ein Event_Adapter-Objekt. Ein Event_Adapter hat folgende IDL-Definition:

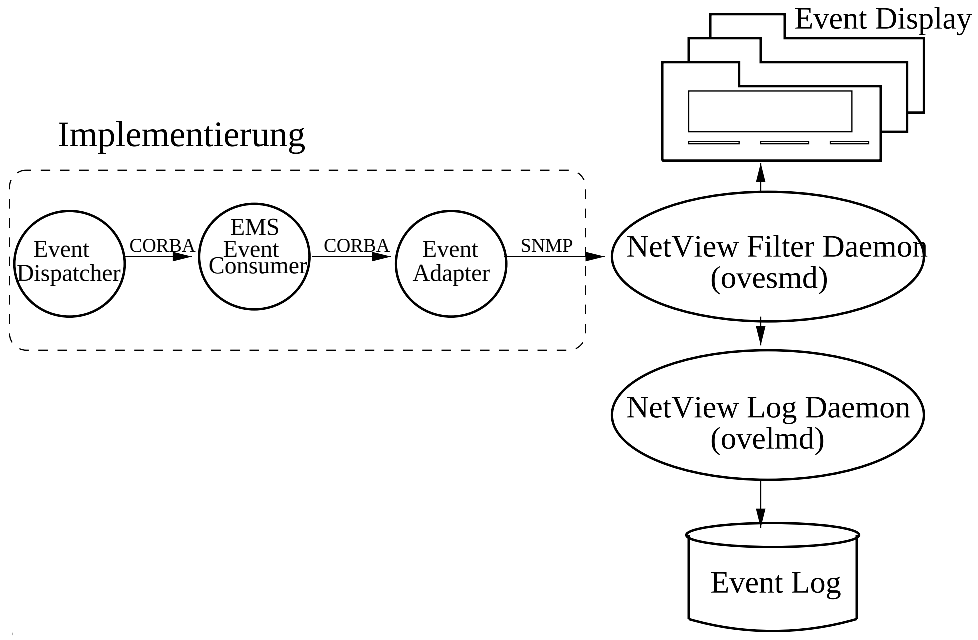

Abbildung 5.7: Umwandlung von CORBA-Events in SNMP-Traps

```
interface Event_Adapter : SOMObject
{
  void create_pdu(in long spec_id, in string source_host);
  void add_arg_long(in long arg);
  void add_arg_string(in string arg);
  void send_pdu();
 }
```

Eine Trap-PDU wird zuerst mit `create_pdu` erzeugt und initialisiert. Dabei wird u.a. die specific-ID des Traps und die dem *host*-Parameter des CORBA-Events entsprechende IP-Adresse als Agenten-Adresse eingesetzt. Danach werden mit den *add_arg*-Methoden die Parameter des CORBA-Events in Variable-Bindings „verpackt" und schließlich wird der Trap abgeschickt. Die Methoden der Klasse Event_Adapter wurden mit dem NetView-SNMP-API implementiert.

Neben der Klasse EMS_Event_consumer wurde die Klasse GTM_Event_consumer entwickelt. Diese Klasse ist von Event_consumer abgeleitet, und die Event-Methoden wurden überschrieben. Ein GTM_Event_consumer empfängt alle topologierelevanten CORBA-Ereignismeldungen und aktualisiert daraufhin die GTM-Datenbank von NetView. Dabei verwendet er die in Kapitel 6 beschriebene CORBA-Schnittstelle für das GTM-API.

5.4.4 Empfang von gefilterten Ereignismeldungen durch CORBA-Anwendungen

Durch die Umwandlung von CORBA-Ereignismeldungen in SNMP-Traps können diese von der Plattform empfangen und verarbeitet werden. Damit ist ein wichtiger Teilaspekt reali-

siert, um die Event-Management-Dienste der Plattform für die Überwachung der CORBA-Umgebung einsetzen zu können. Darüberhinaus ist es jedoch für die Realisierung CORBA-basierter Managementanwendungen notwendig, daß die Anwendungen die von der Plattform gefilterten Ereignismeldungen empfangen können. Es mußte also eine Möglichkeit gefunden werden, um gefilterte SNMP-Traps, die aus CORBA-Events erzeugt wurden, wieder in CORBA-Events zurückzutransformieren. Zusätzlich wurde eine Schnittstelle entwickelt, über die CORBA-basierte Managementanwendungen sich zum Empfangen von gefilterten Ereignismeldungen registrieren können.
Aus CORBA-Sicht ist die Managementplattform ein Supplier von gefilterten Ereignismeldungen. Die Supplier-Rolle wird von sogenannten EFD_Supplier-Objekten übernommen. Der Name wurde gewählt, weil diese CORBA-Objekte eine ähnliche Funktionalität haben wie ein Event-Forwarding-Discriminator in der OSI-Welt (aus Sicht der Empfänger von gefilterten Events). Ein EFD_Supplier wandelt gefilterte SNMP-Traps (die aus CORBA-Events erzeugt wurden) zurück in CORBA-Events und leitet sie an einen Event_Dispatcher weiter, der sie an seine registrierten Consumer „verteilt“. Die Klasse EFD_Supplier hat folgende IDL-Definition:

```
interface EFD_Supplier : SOMObject
{
  attribute string filter;
  void set_Dispatcher(in Event_Dispatcher dispatcher);
  oneway void activate();
}
```

Beim Erzeugen eines EFD_Suppliers erhält dieser eine Objektreferenz auf einen Event_Dispatcher. Nach dem Erzeugen wird das Filter-Attribut gesetzt. Die Registrierung von Consumer-Objekten wird beim Event_Dispatcher vorgenommen. Nach dem Aktivieren des EFD_Suppliers leitet dieser alle Ereignismeldungen, die den Filter passieren, an den Event_Dispatcher weiter.
Der EFD_Supplier filtert die Ereignismeldungen nicht selbst, sondern verwendet dazu die Filtermöglichkeiten der Plattform. Beim Aktivieren des Suppliers werden folgende Aktionen ausgeführt:

- Das Attribut „filter“ wird umgewandelt in einen SNMP-Filterstring. Es könnte auch direkt der SNMP-Filterstring beim Erzeugen des EFD_Suppliers übergeben werden. Allerdings sind gewisse Abbildungen sinnvoll, um die Verwendung von SNMP transparent zu machen. Zumindest ist eine Abbildung zwischen dem Namen des CORBA-Events und der Enterprise-Specific-ID des Traps nötig. Eine Abbildungstabelle kann als Datei, als eigenes Objekt oder innerhalb der Klasse EFD_Supplier realisiert sein. Dadurch, daß eigentlich nicht die CORBA-Events selbst gefiltert werden, sondern SNMP-Traps, ist man bei der Definition des Filter-Attributs auf solche Filterausdrücke beschränkt, die in SNMP-Filter umgewandelt werden können. Die Beschränkungen der SNMP-Filtermöglichkeiten wurden im vorigen Abschnitt bereits genannt.

- Es wird eine SNMP-Session mit dem Filter-Dämon von NetView (ovesmd) aufgebaut. Dabei wird der aus dem CORBA-Filter erzeugte SNMP-Filter als Argument übergeben. Eine Callback-Funktion wird jedesmal automatisch ausgeführt, wenn ein SNMP-Trap den Filter passiert. Diese Callback-Funktion wandelt den SNMP-Trap um in die entsprechende CORBA-Ereignismeldung (der Identifikator dafür ist die Specific-ID des Traps).

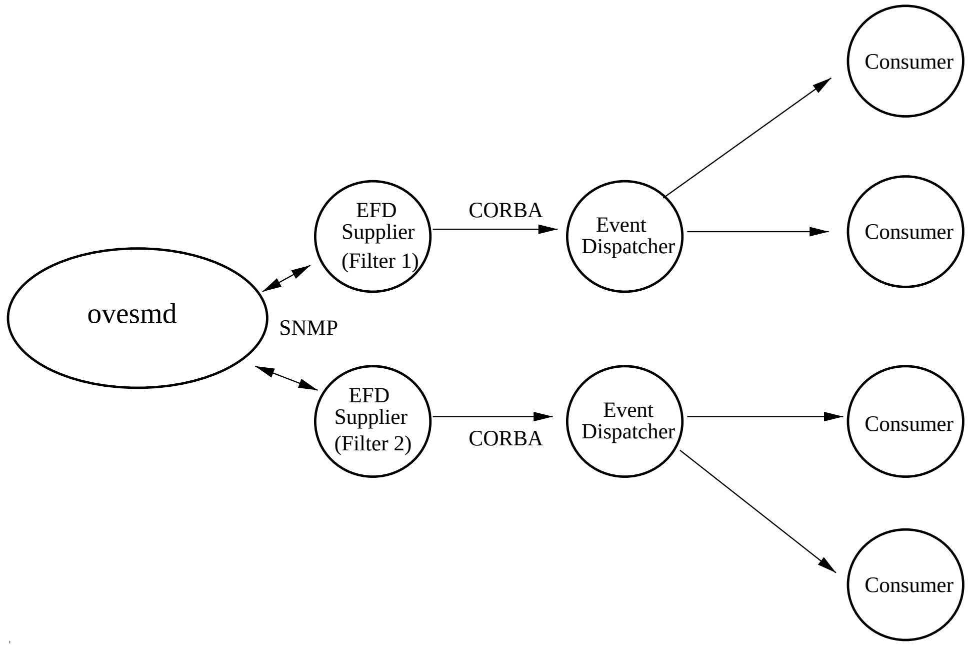

Abbildung 5.8: Event-Filterung durch EFD_Supplier-Objekte

> Die Parameter werden den Variable-Bindings des Traps entnommen. Anschließend wird die Ereignismeldung an den Event_Dispatcher weitergeleitet.

Sobald der SNMP-Filter aktiviert ist, werden in einer Endlosschleife Traps empfangen und durch die Callback-Funktion verarbeitet. Das bedeutet, daß die `activate`-Methode blockiert und daß, während der Filter aktiv ist, keine anderen Methoden auf dem EFD_Supplier aufgerufen werden können. Aus diesem Grund wurde die `activate`-Methode als `oneway` spezifiziert, damit der Aufrufer der Methode nicht ebenfalls blockiert.

5.4.5 Fehlermeldungen aufgrund von Exceptions

Wenn während einem Methodenaufruf auf einem CORBA-Objekt eine Fehlersituation eintritt, erhält der Aufrufer eine Fehlermeldung in Form einer Exception als Rückgabewert. Exceptions sind in IDL definierte Datenstrukturen, die Information über die aufgetretenen Fehler beinhalten.
CORBA unterscheidet zwischen *System_Exceptions* und *User_Exceptions*. Erstere zeigen interne Systemfehler des ORBs an, während letztere vom Benutzer objektklassenspezifisch definiert werden können. Die *System_Exceptions* sind im CORBA-Standard vordefiniert (vgl. [IBM 94f S. 4-25]).
Es wurde ein Verfahren entwickelt, damit beim Auftreten von Exceptions automatisch Ereignismeldungen erzeugt werden, die durch die oben beschriebenen Mechanismen an NetView weitergeleitet werden. Damit kann jede Fehlersituation, die während einem Methodenaufruf auf einem beliebigen Objekt auftritt, von der Plattform registriert werden, wodurch die Grundlage für das Fehlermanagement einer CORBA-Umgebung gegeben ist.

Das entwickelte Verfahren beruht auf folgendem DSOM-spezifischen Mechanismus der Weiterleitung von Methodenaufrufen von Clients an Zielobjekte (vgl. Abb. 5.9):

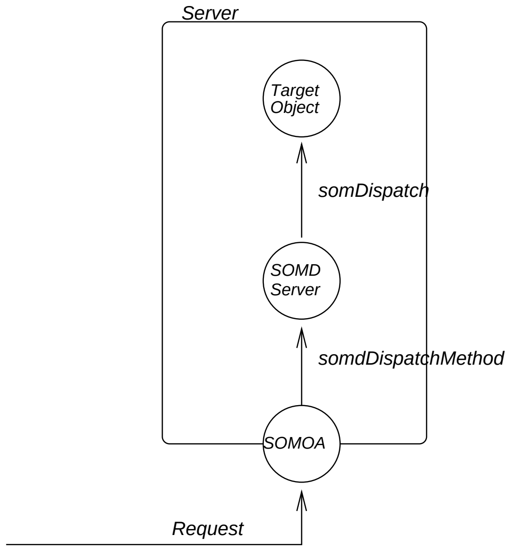

Abbildung 5.9: Die Verarbeitung von Methodenaufrufen bei DSOM

- Ein Client macht einen Methodenaufruf auf einem Proxy-Objekt (oder einen dynamischen Methodenaufruf).
- Der ORB lokalisiert anhand der Objektreferenz den Server, in dem sich das Zielobjekt befindet.
- Beim Server erreicht der Methodenaufruf zunächst den Object Adapter (SOMOA).
- Der SOMOA ermittelt anhand der Objektreferenz, für welches SOM-Objekt der Methodenaufruf bestimmt ist. Dazu ruft er die Methode *somdSOMObjectFromRef* des SOMDServer-Objekts auf.
- Anschließend ruft der SOMOA die Methode *somdDispatchMethod* des SOMDServer auf. Die Parameter der Methode sind u.a. das Zielobjekt und der Identifikator der Methode, die auf diesem aufgerufen werden soll.
- Der SOMDServer führt mittels *somDispatch* den Methodenaufruf auf dem Zielobjekt durch.

Es wurde eine Subklasse von SOMDServer entwickelt, bei der u.a. die Methode *somdDispatchMethod* überschrieben wurde (vgl. auch Kapitel 6). Jedesmal, wenn ein Methodenaufruf auf einem Zielobjekt durchgeführt wird (mit *somDispatch*), wird jetzt überprüft, ob bei dem Methodenaufruf eine Exception aufgetreten ist. Wenn ja, wird eine Ereignismeldung an den Event_Dispatcher verschickt, der sie wiederum an die Plattform weiterleitet. Durch Setzen von Filter-Attributen beim Server-Objekt könnte man erreichen, daß nur bestimmte Exceptions Ereignismeldungen bewirken.

Es besteht also grundsätzlich die Möglichkeit, anwendungsspezifische Funktionalität in die Verarbeitung von Methodenaufrufen zu integrieren. Dazu muß nur eine einzelne Methode (*somdDispatchMethod* des SOMDServer) überschrieben werden.
Dieses Prinzip kann nicht nur für die Fehlerüberwachung eingesetzt werden. Drei weitere Ansätze seien hier genannt:

- Performance Messungen:
  Das Server-Objekt kann die Zeit messen, die es selbst zum Ausführen von *somDispatch* und der aufgerufenen Methode benötigt. Danach können bei Überschreiten einer kritischen Zeitdauer Ereignismeldungen ausgelöst werden. Schwellwerte können objektspezifisch und methodenspezifisch (durch die *methodId*) mittels Setzen von Attributen beim Server-Objekt definiert werden.

- Accounting:
  Es kann der *Principal* jedes Methodenaufrufs überprüft werden. Der Principal ist eine Datenstruktur, die in dem *Environment*-Parameter von Methodenaufrufen übermittelt wird. Sie enthält den Namen des Rechners und den Namen des Benutzers, der den Methodenaufruf ausgelöst hat. Dadurch ist ein Accounting mit der Granularität einzelner Methodenaufrufe möglich. Die Accounting-Daten können in einer Datei oder in internen Datenstrukturen des Server-Objekts gespeichert und von Clients abgefragt werden.

- Sicherheitsüberprüfungen:
  Das Überprüfen des Principals kann auch dazu genutzt werden, Zugriffsbeschränkungen für einzelne Methoden eines Objekts zu definieren und durchzusetzen. Dabei wird vor dem Aufruf einer Methode mit *somDispatch* überprüft, ob der in dem *Principal* angegebene Benutzer Zugriffsrecht auf die Methode des speziellen Objekts hat. Die Zugriffsrechte können zur Laufzeit beim Server-Objekt definiert und geändert werden.

Allerdings ist zu bedenken, daß die Ausführung der eigentlichen Methodenaufrufe um so langsamer werden wird, je größer der „Overhead“ durch die integrierte Managementfunktionalität ist. Aus diesem Grund ist es sinnvoll, die zusätzlichen Funktionen durch Setzen von Attributen bei den Server-Objekten „an- und ausschaltbar“ zu machen.
Die in diesem Kapitel beschriebenen Mechanismen zur Verarbeitung von CORBA-Ereignismeldungen durch die Managementplattform spielten auch bei der Integration des Topologiemanagements eine Rolle. Es wurden u.a. Komponenten entwickelt, die eine ereignisgesteuerte Aktualisierung von CORBA-Topologiedaten in der Plattformdatenbank ermöglichen. Das nächste Kapitel beschreibt diese Komponenten.

Kapitel 6

Integration des Topologiemanagements

6.1 Anforderungen

Eine wichtige Funktion von Managementplattformen besteht darin, Information über die zu überwachenden Ressourcen zu ermitteln, zu speichern und auf Grund dieser Information ein abstraktes Modell an der Benutzeroberfläche darzustellen.
Diese Aufgabe kann in folgende Teilaspekte zerlegt werden:

- Die Information über die vorhandenen Ressourcen wird ermittelt und in der Plattformdatenbank abgespeichert (Discovery-Funktion).
- Da der Bestand an vorhandenen Ressourcen in einem verteilten System sich dynamisch ändern kann, wird die Information aktualisiert.
- Ein Modell wird an der Benutzeroberfläche grafisch dargestellt.
- Zustandsänderungen von Ressourcen werden erkannt und durch Farbänderungen der entsprechenden Symbole visualisiert.

Es wurden Komponenten entworfen und implementiert, die das Topologiemanagement für die Ressourcen einer CORBA-Umgebung ermöglichen. Die Datenbank der verwendeten Managementplattform (NetView für AIX) wird somit zu einer integrierten Informationsbasis für Internet-, OSI- und OMA-Managementobjekte.

6.2 Informationsmodelle für die Ressourcen einer CORBA-Umgebung

Für die weiteren Betrachtungen wird von folgendem grundlegenden Aufbau eines Systems, das aus verteilten und miteinander kommunizierenden Objekten besteht, ausgegangen:

- Verteilte Objekte können untereinander in Referenzbeziehungen [1] stehen.
- Die Objekte befinden sich in Serverprozessen.

[1] D.h. ein Objekt kann Referenzen auf andere Objekte haben und mit ihnen kommunizieren.

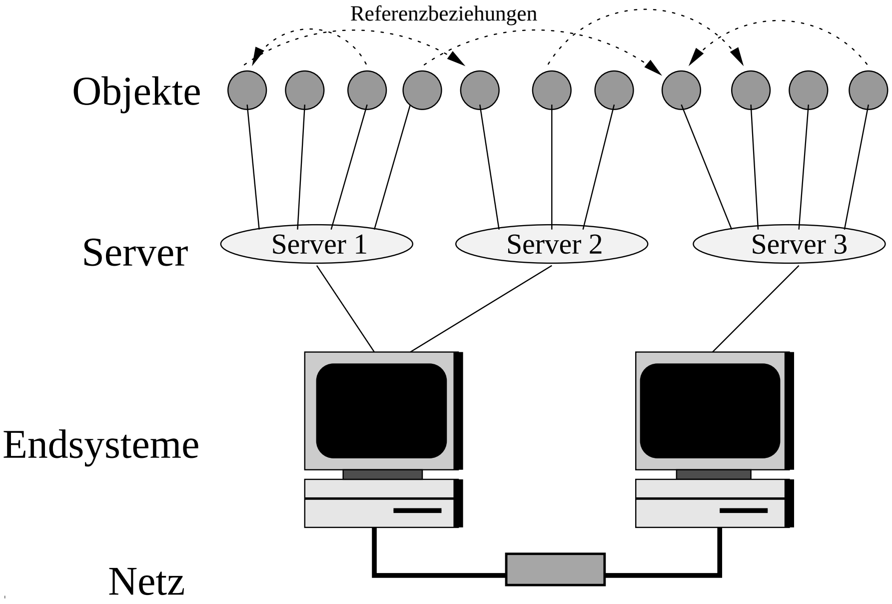

Abbildung 6.1: Schichten einer CORBA Umgebung

- Die Serverprozesse laufen auf Rechnern.
- Die Rechner sind durch ein Netz miteinander verbunden.

Aufbauend auf dieser Grundstruktur wurden Modelle erzeugt, die an der Benutzeroberfläche der Plattform dargestellt werden können. Hierbei war zu berücksichtigen, daß Managementplattformen Topologieinformation durch einen Baum von Submaps präsentieren. Die Modelle mußten also eine entsprechende hierarchische Strukur haben. Die entworfenen Modelle umfassen nicht nur die Ressourcen selbst (wie Rechner, Serverprozesse und Objekte), sondern auch die Beziehungen zwischen ihnen (die in Form von Verbindungen an der grafischen Oberfläche visualisiert werden). Im Gegensatz zur IP-Topologie, die Verbindungen auf der Netzwerkschicht darstellt, sind bei der Darstellung einer CORBA-Umgebung andere Beziehungen zwischen Ressourcen interessant. Aus Sicht der Überwachung von verteilten Anwendungen sind dies die Referenzbeziehungen zwischen den verteilten Objekten, die wiederum abgebildet werden können auf Beziehungen zwischen den Serverprozessen, in denen die Objekte enthalten sind.
Man sieht also, daß je nach Managementszenario (Netz-, System- oder Anwendungsmanagement) unterschiedliche Anforderungen an die Topologiedarstellung gestellt werden. Die grafischen Darstellungsmöglichkeiten der Plattform (hierarchischer Baum von Submaps) ermöglichen kein Modell, das für alle Managementszenarien zufriedenstellend ist. Es wurden daher zwei Topologiemodelle entwickelt und Discovery-Anwendungen dafür implementiert. Eins für das Szenario Systemmanagement und eins für das Anwendungsmanagement. Das Szenario Netzmanagement wurde nicht berücksichtigt, da dieser Bereich durch die IP-Topologiefunktion bereits abgedeckt wird. Die Modelle und ihre Unterschiede werden im folgenden erklärt.

6.2.1 Ein Informationsmodell für das Systemmanagement

Ausgangspunkt für das Informationsmodell ist die Menge der vorhandenen Endsysteme. Innerhalb der einzelnen Systeme sind die Systemressourcen (wie z.B. Dateisysteme, Benutzer, installierte Software etc.) interessant, die durch Managementobjekte repräsentiert werden. Die Managementobjekte sind in Serverprozessen enthalten.
Ein sinnvoller Aspekt ist die Gruppierung von Endsystemen nach bestimmten Kriterien (z.B. geographische Einheiten oder Policy-Regionen). Damit ergibt sich folgendes Informationsmodell, das ausgehend von einem Wurzel-Symbol (Root Icon) in einer Hierarchie von Submaps präsentiert wird:

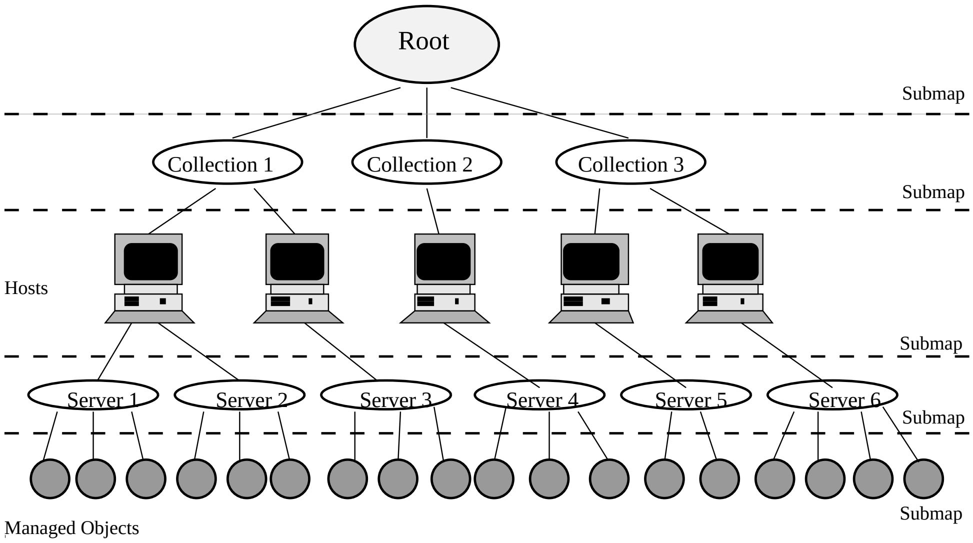

Abbildung 6.2: Informationsmodell für das Systemmanagement

Beim Anklicken eines Symbols für eine Collection (Gruppierung von Rechnern) gelangt man in eine Submap, in der für jeden Rechner, der in der Collection enthalten ist, ein Symbol vorhanden ist. Ebenso werden in der unter einem Rechner-Symbol liegenden Submap alle auf diesem Rechner vorhandenen Serverprozesse gezeigt. Schließlich befindet sich unter jedem Server-Symbol eine Submap mit Symbolen für alle Managed Objects in diesem Server.

6.2.2 Ein Informationsmodell für das Anwendungsmanagement

An die grafische Darstellung zur Überwachung verteilter Anwendungen werden andere Anforderungen gestellt als beim Systemmanagement, obwohl für beide Szenarien die gleiche CORBA-Basisarchitektur verwendet wird. Aus diesem Grund wurde ein zweites Modell entwickelt, mit dem die Beziehungen zwischen den Serverprozessen einer verteilten CORBA-Anwendung überwacht werden können. Eine Beziehung liegt dann vor, wenn ein Objekt, das sich in einem Server befindet, eine Referenz auf ein Objekt hat, das in einem anderen Server enthalten ist. Zwischen den Symbolen der entsprechenden Serverprozesse wird dann eine Verbindung erzeugt. Die Mechanismen zum Erkennen der Referenzbeziehungen werden weiter unten beschrieben.
Es ergibt sich das in Abbildung 6.3 gezeigte Modell:

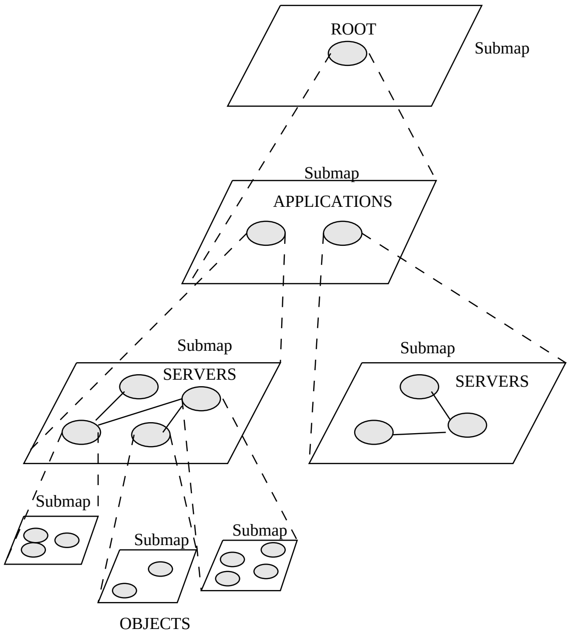

Abbildung 6.3: Informationsmodell für das Anwendungsmanagement

Im Gegensatz zu dem für das Systemmanagement entwickelten Modell treten hier keine Rechner auf. Der Grund dafür ist, daß an der Benutzeroberfläche immer nur Verbindungen zwischen Symbolen innerhalb einer Submap dargestellt werden können. Wenn die Submaps mit den Serverprozessen unterhalb den Symbolen für die Rechner liegen würden, könnten nur Verbindungen zwischen den Serverprozessen eines Rechners angezeigt werden. Statt dessen werden die Serverprozesse nach den Anwendungen, denen sie angehören, gruppiert. In einer übergeordneten Submap befindet sich ein Symbol für jede Anwendung. Unterhalb einem Anwendungssymbol liegt eine Submap mit sämtlichen Serverprozessen, aus denen die Anwendung besteht.
Damit man trotzdem erkennen kann, auf welchem Rechner sich ein Serverprozess befindet, kann der Name des Rechners neben der Bezeichnung des Serverprozesses im Label des entsprechenden Serversymbols angegeben werden.
Als Alternative wäre auch ein Modell denkbar, bei dem direkt die Referenzbeziehungen zwischen den einzelnen Objekten dargestellt werden. Das hätte allerdings zur Folge, daß alle Objekte, aus denen eine Anwendung besteht, in einer Submap dargestellt werden müßten.

6.3 Informationsquellen

Dieser Abschnitt beschreibt die Informationsquellen, die verwendet wurden, um die vorhandenen Ressourcen zu ermitteln, und die Modelle in der Datenbank der Plattform zu erzeugen. Die Beschreibung bezieht sich dabei auf DSOM-Komponenten. Man kann nicht uneingeschränkt davon ausgehen, daß eine andere CORBA-Implementierung die gleichen Informationsquellen zur Verfügung stellt.

Anschließend werden noch einige weitere Informationsquellen beschrieben, die DSOM zur Zeit nicht zur Verfüngung stellt, die aber prinzipiell in einer CORBA-Umgebung eingesetzt werden können.

6.3.1 Informationsbedarf

Folgende Basisinformation wird benötigt:

- Aus welchen Rechnern besteht die DSOM-Umgebung ?
- Welche Serverprozesse laufen auf einem Rechner ?
- Welche Objekte sind in einem Serverprozeß vorhanden ?
- Welche Proxy-Objekte gibt es in einem Serverprozeß ?

Proxy-Objekte entsprechen bei DSOM Objektreferenzen. Die Information, welche Proxy-Objekte in einem Server vorhanden sind, wird benötigt, um die Beziehungen zwischen den Servern zu ermitteln. Dabei wird folgendes Prinzip angewendet:

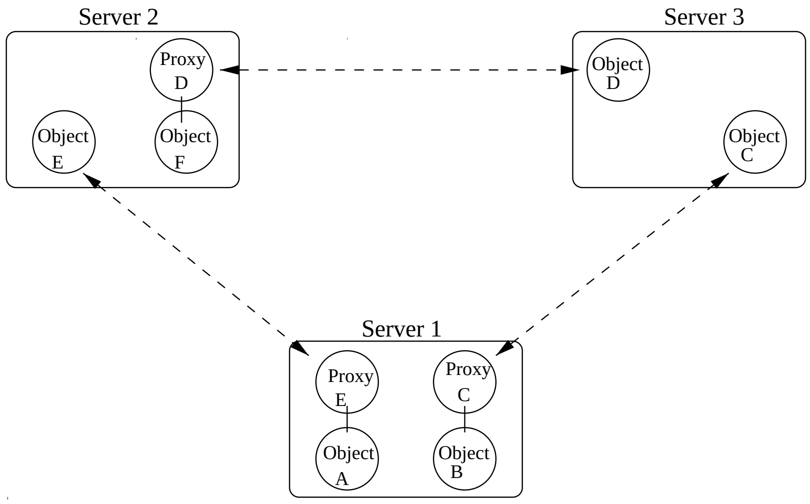

Abbildung 6.4: Beziehungen zwischen Serverprozessen

- Ein Serverprozeß enthält normalerweise Objekte, auf denen Clients Methoden aufrufen können.
- Ein Serverprozeß kann aber auch Proxy-Objekte für Objekte in anderen Serverprozessen enthalten. Diese Proxy-Objekte werden beispielsweise in den Instanzvariablen von in dem Server befindlichen Objekten gehalten. Ein Objekt, das ein Proxy-Objekt als Instanzvariable hat (oder ein Proxy-Objekt als Rückgabewert eines Methodenaufrufs erhält), ist im Besitz einer Referenz auf das Zielobjekt des Proxies (vgl. Abb. 6.4).
- Wenn man ermitteln kann, welche Proxy-Objekte sich in einem Server befinden und auf welchen Servern sich deren Zielobjekte befinden, können die Beziehungen zwischen den Serverprozessen grafisch dargestellt werden (z.B. durch Pfeile).

- Die Darstellung der Beziehungen zwischen den Serverprozessen kann zusammen mit der Protokollierung und Visualisierung der Serverzustände (laufend oder nicht) dazu verwendet werden, Abhängigkeiten im Fehlerfall von verteilten CORBA-Anwendungen zu ermitteln.

6.3.2 Implementation Repository

Eine grundlegende Informationsquelle ist das CORBA Implementation Repository. Bei DSOM ist das Implementation Repository eine Datei, in der sämtliche Serverprozesse registriert sind. Ein Eintrag im Implementation Repository hat folgende Bestandteile:

- Ein vom System generierter eindeutiger Identifikator für den Server.
- Ein Name, der vom Benutzer zugewiesen wird.
- Der Pfadname des Serverprogramms.
- Die Klasse des Server-Objekts (z.B. SOMDServer).
- Eine Datei, in der *ReferenceData* gespeichert wird, die beim Erzeugen von Objektreferenzen angegeben werden kann.
- Angaben darüber, ob der Server mehrere Threads haben kann.
- Der Name des Rechners, auf dem sich der Server befindet.

Zusätzlich ist eine Liste der Objektklassen vorhanden, die durch den Server unterstützt werden. Das Implementation Repository wird benötigt, um Serverprozesse bei Bedarf starten zu können (durch den DSOM Dämon somdd) und um die Verbindung zwischen Clients und Servern herstellen zu können. Ein Client kann nur dann die Verbindung mit einem Server aufnehmen, wenn der Server im Implementation Repository des Client-Rechners registriert ist. Aus diesem Grund wird in der Regel ein zentrales Implementation Repository in einem verteilten Dateisystem (NFS, etc.) gehalten.
Als zukünftige Alternative zum Implementation Repository könnte ein Trader-Service eingesetzt werden, in dem Objekte und die von ihnen angebotenen Dienste registriert sind.
Das Implementation Repository wird in jedem DSOM-Prozeß (egal ob Client oder Server) zur Laufzeit durch ein Objekt (*SOMD_ImplRepObject*) repräsentiert. Durch die Methoden dieses Objekts können sämtliche Einträge in Form von *ImplementationDef*-Objekten ermittelt werden. Dadurch kann man herausfinden, aus welchen Serverprozessen und Rechnern die DSOM-Umgebung besteht.

6.3.3 Server-Objekte

Damit die vorhandenen Objektinstanzen der Managementplattform bekannt gemacht werden können, müssen sie an einem einer Discovery-Anwendung zugänglichen Ort registriert sein. Die entwickelte Lösung besteht darin, daß das Server-Objekt, das in jedem DSOM-Server vorhanden ist, eine Liste aller in dem Prozeß vorhandenen und beim ORB registrierten Objekte beinhaltet. Diese Liste enthält die Objektreferenzen (in String-Form) und die Namen der Objektklassen. Außerdem wird eine Liste der Objektreferenzen aller im Server vorhandenen Proxy-Objekte geführt. Die Listen sind über die Aufrufschnittstellen des Server-Objekts nach außen zugänglich.

Damit die Objekte automatisch bei ihrer Erzeugung registriert werden, wurde eine Unterklasse des Standardservers SOMDServer entworfen und implementiert.
Diese Lösung wurde entwickelt, weil keine Möglichkeit besteht, direkt beim ORB alle registrierten Objekte abzufragen und weil kein Naming Service vorhanden war, in dem Objekte registriert werden können.

6.3.4 Objektreferenzen

Die Beziehungen zwischen den Serverprozessen werden, wie oben beschrieben, aufgrund der in einem Server vorhandenen Proxy-Objekte ermittelt.
Dazu muß man herausfinden, auf welchem Server sich das Zielobjekt eines Proxies befindet. Bei DSOM kann diese Information aus der Objektreferenz (in String-Form) eines Proxy-Objekts gewonnen werden. Eine DSOM-Objektreferenz hat folgendes Format:

```
SOM|2|3090d76e-182ff100-7f-00-10005a4fd4e9|test|12|202b7f5830b6fca53b154500
```

Dieser String repräsentiert eine Objektreferenz für ein Objekt der Klasse „test“ . Das Format hat eine feste Definition mit durch senkrechte Striche getrennten Komponenten. Neben dem Namen der Objektklasse (test) und einem Identifikator für die Objektinstanz (`202b7f5830b6fca53b154500`) enthält der String auch den Implementation-Repository-Identifikator des Servers, auf dem sich das Objekt befindet: `3090d76e-182ff100-7f-00-10005a4fd4e`.
Da die Objektreferenz eines Proxy-Objekts stets identisch ist mit der seines Zielobjekts, können die Beziehungen zwischen den Servern ermittelt werden.
Das gezeigte Format für Objektreferenz-Strings ist spezifisch für die aktuelle Version (2.1) von DSOM. Im Rahmen der Standardisierung von CORBA 2.0 wurden aber auch Formate für interoperable Objektreferenzen definiert. So ist z.B. beim *Internet Inter-ORB Protocol (IIOP)* eine Struktur für Objektreferenzen festgelegt, in der u.a. der Rechner des Zielobjekts angegeben wird (vgl. [OMG 95 S.77 f.]). Dieses Format für Objektreferenzen wird verwendet, wenn die Anforderung eines Methodenaufrufs zwischen zwei unterschiedlichen ORBs transportiert wird. Jeder CORBA2.0-konforme ORB muß das Objektreferenz-Format interpretieren können. Damit können Clients auf Objekte zugreifen, die von ORBs verschiedener Hersteller verwaltet werden.

6.3.5 Weitere Informationquellen

Die hier genannten Informationsquellen wurden bei der Implementierung nicht verwendet, da sie von DSOM zur Zeit nicht zur Verfügung gestellt werden. Sie können jedoch prinzipiell eingesetzt werden, um die vorhandenen Ressourcen in einer CORBA-Umgebung zu ermitteln.

1. *Naming Service.* Der von der OMG standardisierte Naming Service [OMG 93b] ermöglicht die Zuordnung von Namen zu Objekten und die Speicherung dieser *NameBindings* durch einen hierarchisch aufgebauten Verzeichnisdienst, der selbst in Form von Objekten realisiert ist (sogenannte *NamingContexts*). Durch den Dienst kann eine Auflösung von Namen und den zugehörigen Objektreferenzen vorgenommen werden. Objekte können also aufgrund ihres Namens adressiert werden. Über Methoden der NamingContext-Objekte können alle darin enthaltenen *NameBindings* abgefragt werden. Somit kann man die Namen und Objektreferenzen aller registrierten Objekte ermitteln.

2. *Trader Service.* Der Trader Service dient zum Auffinden von bestimmten Diensten. Ein Diensterbringer kann sich beim Trader registrieren, wobei er dem Trader Information über den angebotenen Dienst übergibt. Diese Information wird vom Trader gespeichert. Wenn ein Client eine bestimmte Art von Dienst in Anspruch nehmen will, wendet er sich an den Trader und übergibt ihm eine Beschreibung der gewünschten Funktionalität. Der Trader vergleicht die Beschreibung mit der Information über die bei ihm registrierten Dienste und wenn er einen Dienst findet, der für die Wünsche des Clients „akzeptabel“ ist, erhält der Client eine Referenz auf den Erbringer des Dienstes. Der Trader Service kann für das Anwendungsmanagement verwendet werden, um herauszufinden, welche Dienste es in einer CORBA-Umgebung gibt und welche Objekte die Erbringer dieser Dienste sind. Es gibt momentan keinen gültigen OMG-Standard für einen Trader Service. Als Literaturhinweis sei [ISO/IEC 13235] genannt. Dort findet man eine OMG-IDL Spezifikation der ODP Trading Function.

6.4 Discovery und Aktualisierung des Informationsbestands

6.4.1 Die Discovery-Anwendung

Damit eine Discovery-Anwendung als reine CORBA-Anwendung implementiert werden konnte, wurde eine aus DSOM-Objekten bestehende Schnittstelle für das GTM-API der Plattform entwickelt (vgl. 6.5.1). Durch Methodenaufrufe auf diesen Objekten können „NetView-Objekte“ in der Datenbank erzeugt werden.
Um die vorhandenen CORBA-Ressourcen zu ermitteln, wurden die oben genannten Informationsquellen verwendet.

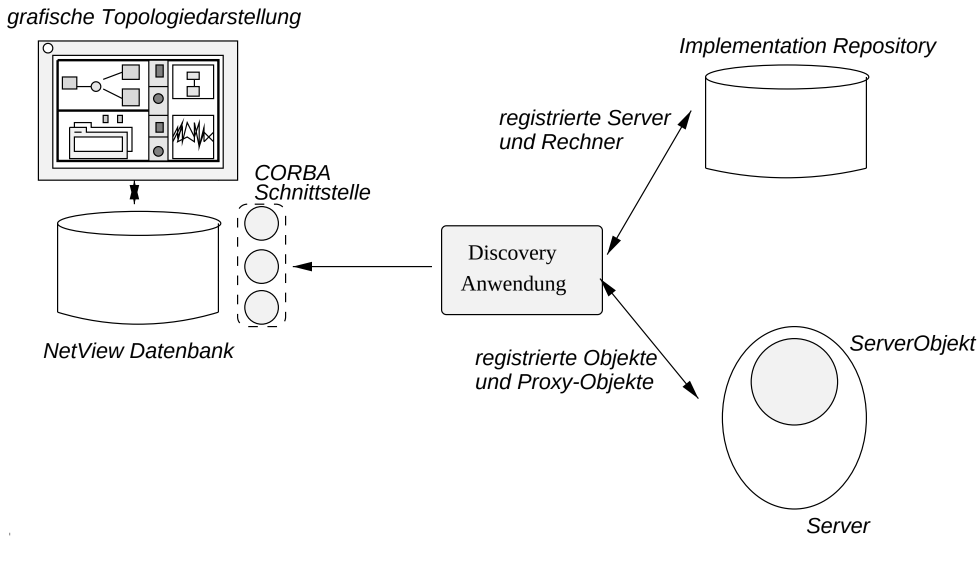

Abbildung 6.5: Discovery Anwendung

Die für die CORBA-Ressourcen erzeugten NetView-Objekte beinhalten neben intern verwendeten Attributen folgende Information:

- Objekte für Rechner: Name des Rechners.
- Objekte für Server: Implementation Repository ID, Server-Alias (als Label des Symbols).
- Objekte für CORBA-Objekte: Objektreferenz in String-Form, Name der Klasse (Der Klassenname wird als Label für das Symbol verwendet, da kein *Naming Service* vorhanden war).
- Objekte für Proxies: Die gleichen Attribute wie bei den CORBA-Objekten und zusätzlich eine Kennzeichnung, daß es sich um ein Proxy-Objekt handelt.

Zusätzlich enthält jedes Objekt Attribute, die seinen aktuellen Zustand kennzeichnen. Die Zustandsattribute werden von NetView in Farbkodierungen der entsprechenden Symbole umgesetzt.
Die Speicherung von zusätzlicher Information in den NetView-Objekten ist möglich.

6.4.2 Aktualisierung der Topologiedaten aufgrund von Ereignismeldungen

Der Informationsbestand wird aktualisiert, wenn eines der folgenden Ereignisse eintritt:

- Hinzufügen oder Löschen von Server-Definitionen im Implementation Repository.
- Erzeugen oder Löschen von Objekten und Proxy-Objekten in einem Serverprozeß.
- Zustandsänderungen von Ressourcen (z.B. Starten eines Serverprozesses).

Prinzipiell kann für diese Aufgabe ein Mechanismus verwendet werden, der dem in der IP-Discovery verwendeten Polling entspricht. Konkret bedeutet dies, daß das Implementation Repository regelmäßig nach Änderungen durchsucht wird und daß bei den Server-Objekten die Listen der vorhandenen Objekte und Proxies abgefragt werden. Der bekannte Nachteil des Pollings ist der durch die regelmäßigen Abfragen erzeugte Netzverkehr. Der Netzverkehr wäre sogar noch höher als bei der IP-Discovery, da man nicht nur die Rechner „pollen“ müßte, sondern jeden einzelnen Serverprozeß.
Um die Nachteile des Pollings zu vermeiden, wurde eine Lösung konzipiert, die auf Ereignismeldungen basiert:

- Die Instanzen der entwickelten Serverobjekt-Klasse erzeugen Ereignismeldungen, wenn Objekte oder Proxy-Objekte in dem entsprechenden Serverprozeß erzeugt oder gelöscht werden.
- Instanzen einer Subklasse des *SOMD_ImplRepObjects* verschicken Ereignismeldungen, wenn Server registriert oder entfernt werden.
- Ein Serverprogramm, das an Stelle von *somdsvr* verwendet werden kann, erzeugt eine Ereignismeldung, wenn der Server gestartet bzw. terminiert wird.

Die Ereignismeldungen werden mit den in Kapitel 5 beschriebenen Mechanismen an die Managementplattform verschickt und dort von speziellen Consumer-Objekten empfangen. Diese Objekte interpretieren die Ereignismeldungen und nehmen die entsprechenden Änderungen in der NetView-Datenbank vor (Erzeugen oder Löschen von Objekten bzw. Änderung von Zustandsattributen).

Da zwei getrennte Topologiemodelle erstellt wurden, gibt es auch zwei Consumer-Objekte für Topology-Events. Diese empfangen zwar die gleichen Ereignismeldungen, interpretieren sie aber anders. Beispielsweise muß beim Systemmanagement-Modell nicht nur ein neuer Server in das Topologiemodell integriert werden, sondern evtl. auch ein neuer Rechner, wenn ein Eintrag im Implementation Repository erfolgt. Beim Anwendungsmanagement-Modell kommen Rechner jedoch gar nicht vor.

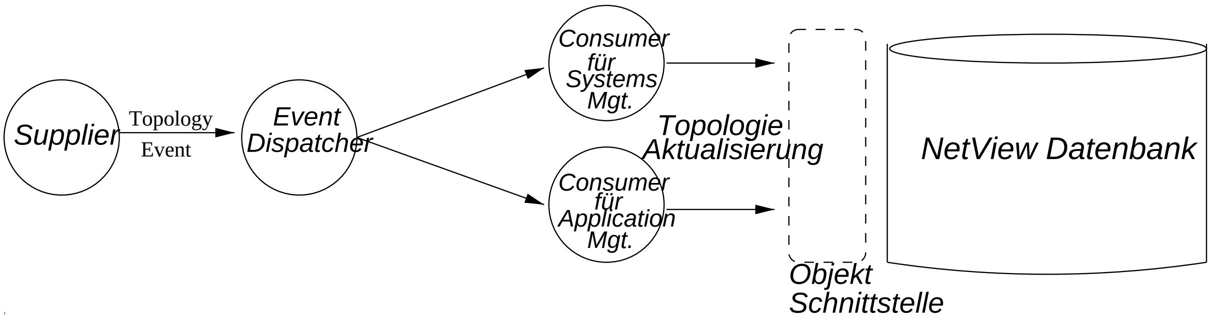

Abbildung 6.6: Aktualisierung der Plattformdatenbank durch Ereignismeldungen

6.5 Implementierung der Komponenten

Dieser Abschnitt beschreibt die Implementierung der für das Topologiemanagement entwickelten Komponenten.

6.5.1 Schnittstelle zur Datenbank der Plattform

Es gibt zwei Alternativen, um bei NetView Topologiemodelle zu erzeugen und grafisch darzustellen:

1. Es kann der in Kapitel 3 beschriebene General Topology Manager (GTM) verwendet werden.

2. Es können direkt Objekte in der NetView Objekt-Datenbank erzeugt werden.

Der Vorteil des GTM ist, daß der Programmierer der Topologieanwendung sich nicht um die grafische Darstellung des Topologiemodells kümmern muß. Dies wird durch die *xxmap*-Anwendung automatisch ausgeführt.
Wenn der GTM nicht verwendet wird, müssen sowohl die Objekte, als auch die Submaps und Symbole für ihre grafische Darstellung erzeugt werden. Dabei muß direkt mit dem User Interface API gearbeitet werden. Der GTM erspart also ein nicht unerhebliches Maß an Entwicklungsaufwand.
Der geringere Entwicklungsaufwand muß jedoch mit einer reduzierten Flexibilität bei der Topologiedarstellung erkauft werden. Wenn unterschiedliche Topologie-Views auf die gleichen CORBA-Ressourcen erstellt werden sollen (wie hier das Systemmanagement- und das Anwendungsmanagementmodell) müssen zwei getrennte Modelle in der GTM-Datenbank erzeugt werden (mit eigenem Protokollidentifikator), auch wenn die GTM-Objekte die gleichen realen Ressourcen repräsentieren.

Bei Verwendung des User Interface API können hingegen einem Objekt beliebig viele Symbole in unterschiedlichen Submap-Hierarchien zugeordnet werden. Konzeptionell ist die direkte Arbeit mit der Objekt-Datenbank und der grafischen Oberfläche also sinnvoller. Bei der Entwicklung des Prototypen wurde allerdings der GTM verwendet.
Der GTM hat ein API, mit dem Objekte in der zugehörigen Datenbank erzeugt werden können. Außerdem können Managementanwendungen über dieses API auf die Information in der Datenbank zugreifen.
CORBA-basierte Managementanwendungen bestehen in der Regel aus verteilten Objekten. Damit diese Objekte unabhängig von ihrer Lokation Einträge in der GTM-Datenbank erzeugen können und Zugriff auf die darin gespeicherte Information haben, wurde eine aus CORBA-Objektklassen bestehende „Kapselung" für dieses API entwickelt.
Für jede Komponente des abstrakten GTM-Datenmodells wurde eine Objektklasse definiert. Über die Methoden der Klassen können die entsprechenden GTM-Objekte erzeugt und gelöscht werden. Normalerweise müssen beim Erzeugen eines GTM-Objekts viele „NetView-interne" Parameter angegeben werden, wie z.B. der Protokollidentifikator oder der Typ des gewünschten Symbols. Bei den entwickelten Objektklassen wurden möglichst viele dieser Parameter bereits intern vordefiniert, um die NetView-spezifischen Details bei der Benutzung des APIs möglichst transparent zu machen.
Als Beispiel sei hier die IDL-Definition der Klasse *Vertex* gezeigt.

```
interface Vertex : SOMObject
{
 void create_in_box(in string boxName,
                    in string vertexName,
                    in string label);

 void create_in_graph(in string graphName,
                      in string vertexName,
                      in string label);

 void delete_vertex(in string vertexName);

 void change_vertex_status(in string vertexName,
                           in string vertexStatus);
}
```

Die Kapselung der Funktionsaufrufe zum Auslesen von Information aus der GTM-Datenbank wurde nicht mehr implementiert. Dazu müssen nämlich die teilweise komplexen Datenstrukturen der Rückgabeparameter dieser Funktionsaufrufe nach CORBA-IDL „gemappt" werden. Dies ist zwar prinzipiell möglich, hätte jedoch einen zu großen Zeitaufwand erfordert.
In der GTM-Datenbank werden neben der CORBA-Topologie auch die Topologiemodelle anderer NetView-Anwendungen gespeichert, wie z.B. die der TMN-Plattform, des SAP-R/3-Managers und anderer. CORBA-Managementanwendungen können prinzipiell also auch auf diese Information zugreifen. Z.B. ist es denkbar, daß im Fehlerfall einer verteilten CORBA-Anwendung die Ursache nicht nur bei den CORBA-Komponenten selbst gesucht wird, sondern daß auch die in der GTM-DB gespeicherten Zustandsattribute von mit TMN überwachten Netzkomponenten berücksichtigt werden. Der vermeintliche Anwendungsfehler kann in Wirk-

lichkeit nämlich auch darin begründet liegen, daß die Kommunikationsinfrastruktur nicht funktioniert.

6.5.2 Erweiterung des Server-Objekts

Um alle existierenden Objektinstanzen ermitteln zu können, wurde eine Subklasse von *SOMD-Server* entwickelt, die u.a. folgende Funktionalität bietet:

- Die Instanzen der Klasse haben Listen aller in dem Serverprozeß existierenden Objekte und Proxy-Objekte.
- Zusätzlich zum Eintragen in die Listen wird beim Erzeugen bzw. Zerstören eines neuen Objekts bzw. Proxies eine Ereignismeldung an den Event_Dispatcher (vgl. Kapitel 5) verschickt, der sie an die Consumer-Objekte der Plattform weiterleitet.
- Falls bei einem Methodenaufruf auf einem Objekt innerhalb des Servers ein Fehler (Exception) auftritt, wird eine Ereignismeldung erzeugt (vgl. Kapitel 5).

Als Grundlage für diese Server-Klasse diente der in [IBM 95a] entwickelte ITSCSOMDServer. Diese Klasse ist von SOMDServer abgeleitet und hat zusätzliche Methoden, um alle in dem Server vorhandenen Klassenobjekte und Objektinstanzen abfragen zu können:

```
interface ITSCSOMDServer : SOMDServer
{
  exception ITSCSERV_ERR {long errCode; char Reason[80];};
  long GetAllRegisterClasses( out sequence<string> clsList)
                                            raises (ITSCSERV_ERR);

  long GetAllUserClasses( out sequence<string> clsList)
                                      raises (ITSCSERV_ERR);

  long GetAllAffinityGroup(in string class_name,
             out sequence<string> clsList) raises (ITSCSERV_ERR);

  long GetClassStatus(in string class_name) raises (ITSCSERV_ERR);

  long GetClassTree(in string class_name,
             out sequence<string> clsList) raises (ITSCSERV_ERR);

  long GetInstancesForClass (in string class_name,
             out sequence<RefStruct::refData> clsList)
             raises (ITSCSERV_ERR);

  long GetClassAncestors (in string class_name,
             out sequence<string> clsList) raises (ITSCSERV_ERR);

  void verifyList(in RefLinkList refLList);
```

```
  sequence<RefStruct::refData> GetRefData();

/*additional methods*/
  sequence<RefStruct::refData> GetProxyData();

  void AddProxyToList(in string ref, in string class);

  void DeleteProxyFromList(in string ref, in string class);

  void object_created(in string server,
                      in string objref,
                      in string class);

  void object_deleted(in string objref, in string class);
}
```

- *GetAllRegisterClasses* liefert eine Liste aller Klassen, die momentan beim SOMClass-MgrObject des Serverprozesses registriert sind.
- *GetAllUserClasses* berücksichtigt nur die benutzerdefinierten Klassen, keine internen SOM-Klassen.
- *GetAllAffinityGroup* liefert eine Liste aller Klassen, die sich in der selben Bibliothek befinden wie die angegebene Klasse.
- *GetClassStatus* stellt fest, ob die angegebene Klasse momentan geladen ist.
- *GetClassTree* liefert den Vererbungsbaum einer Klasse in Form einer Liste aller ihrer Superklassen.
- *GetClassAncestors* liefert die unmittelbaren Superklassen einer Klasse.
- *GetRefData* liefert die Objektreferenzen aller Objekte des Serverprozesses, die beim ORB registriert sind.
- *GetInstancesForClass* kann verwendet werden, um zu einer bestimmten Klasse alle existierenden Objektinstanzen festzustellen.
- *verfiyList* überprüft die Aktualität der Liste aller Objektreferenzen.

Die Implementierung dieser Methoden ist in [IBM 95a S.71ff.] beschrieben. Die Klasse wurde um folgende Methoden erweitert:

- *GetProxyData* liefert eine Liste der Objektreferenzen aller registrierten Proxy-Objekte.
- *AddProxyToList, DeleteProxyFromList* werden verwendet, um Proxy-Objekte beim Server-Objekt zu registrieren.
- *object_created* , *object_deleted* dienen zum Versenden von Ereignismeldungen, wenn neue Objekte in dem Serverprozeß erzeugt wurden.

Um Objekte (keine Proxies) bei ihrer Erzeugung automatisch zu registrieren, wurde die Methode *somdRefFromSOMObj* des *SOMDServer* überschrieben. Diese Methode wird vom Object Adapter (SOMOA) jedesmal aufgerufen, wenn in dem Serverprozeß ein neues Objekt erzeugt wird (z.B. durch *somdCreateObject*). Sie erzeugt eine Referenz für das neue Objekt und registriert es beim ORB. Die überschriebene Methode ruft zunächst die entsprechende Parent-Methode auf, trägt die Referenz des neuen Objekts dann in die dafür bestimmte Liste des Server-Objekts ein und erzeugt eine Ereignismeldung. Alternativ zu *somdRefFromSOMObj* hätte auch die Methode *somdCreateObject* überschrieben werden können. Allerdings ist *somdRefFromSOMObj* „allgemeingültiger" weil damit auch Objekte registriert werden können, die z.B. über spezielle Metaklassen erzeugt werden.
Die Zerstörung von Objekten wird durch Überschreiben der Methode *somdDeleteObject* automatisch protokolliert, wobei der entsprechende Eintrag in der Liste gelöscht wird und ebenfalls eine Ereignismeldung verschickt wird.
Die Ereignismeldungen werden, wie in Kapitel 5 beschrieben, an das Event_Dispatcher-Objekt geschickt und von diesem an die der Plattform „vorgeschalteten" Consumer-Objekte weitergeleitet.
Das Registrieren der Erzeugung und Zerstörung von Proxy-Objekten ist komplizierter, weil Proxy-Objekte innerhalb eines Serverprozesses auf unterschiedliche Arten erzeugt werden können:

- Durch die Methode *somdNewObject* des *SOMDObjectMgr*.
- Aus einem Objektreferenz-String (durch die Methode *object_to_string* des *ORB*).
- Als Return-Werte oder Ausgabeparameter von Methodenaufrufen.

Die eigentliche Konstruktion von Proxy-Objekten geschieht durch interne SOM-Funktionsaufrufe, die nicht für den Anwendungsprogrammierer zugänglich sind. Man kann also nicht durch Überschreiben von bestimmten Methoden erreichen, daß die Erzeugung bzw. Zerstörung von Proxies automatisch registriert wird.
Eine Lösungsmöglichkeit besteht darin, benutzerdefinerte Proxy-Klassen zu entwickeln, deren Instanzen sich bei ihrer Erzeugung automatisch beim Server-Objekt registrieren. Dies stellt aber einen Mehraufwand bei der Anwendungsentwicklung dar, weil nicht nur die Anwendungsklassen selbst implementiert werden müssen, sondern auch die Proxy-Klassen. Da die Implementierung von Proxy-Klassen jedoch grundsätzlich nach einem bestimmten Muster erfolgt (vgl. [IBM 94b S.66f]) könnte man sie auch durch ein entsprechendes Werkzeug automatisch generieren lassen.
Bei der Implementierung des Prototypen wurden Methoden des SOMDObjectMgr überschrieben, um Proxies zu registrieren, die mit diesen Methoden erzeugt bzw. gelöscht werden.

6.5.3 Erweiterung des Object Manager

Die Klasse SOMDObjectMgr stellt Methoden zum Erzeugen und Löschen von Proxy-Objekten bereit. Es wurde eine Subklasse des Object Managers entwickelt, bei der diese Methoden überschrieben wurden. Es handelt sich um die Methoden *somdNewObject, somdDestroyObject* und *somdReleaseObject*.
In der Implementierung dieser Methoden wird zusätzlich zum Aufruf der entsprechenden Parent-Methode das Proxy-Objekt beim Server-Objekt registriert (bzw. aus der Liste entfernt) und eine Ereignismeldung erzeugt.

6.5.4 Erweiterung der Implementation Repository Schnittstelle

Durch die entwickelten Subklassen des SOMDServer und SOMDObjectMgr können die Objekte und die Proxies in den Serverprozessen ermittelt werden. Die Information, aus welchen Serverprozessen (und Rechnern) die CORBA-Umgebung besteht, kann aus dem Implementation Repository gewonnen werden.
Für den Zugriff von Anwendungen auf das Implementation Repository dient das in jeder DSOM-Anwendung vorhandene *SOMD_ImplRepObject*. Über die Methoden dieses Objekts können Anwendungen sowohl die bereits existierenden Server-Definitionen auslesen, als auch neue Server registrieren und vorhandene Definitionen löschen.
Es wurde eine Subklasse von *ImplRepository* entwickelt. Dabei wurden die Methoden *add_impldef* und *delete_impldef* derart überschrieben, daß beim Eintrag bzw. beim Löschen einer Server-Definition automatisch eine Ereignismeldung erzeugt wird.

6.5.5 Serverprogramm

Um überwachen zu können, ob ein bestimmter Server gestartet ist oder nicht, wurde das Programm *managed_server* entwickelt, das anstelle von *somdsvr* eingesetzt werden kann.
Dieser Server verschickt eine Ereignismeldung, wenn er gestartet wird und wenn er terminiert (durch einen entsprechenden Exit-Handler). Aufgrund dieser Ereignismeldungen werden die Zustandsattribute des dem jeweiligen Server entsprechenden NetView-Objekts gesetzt, die wiederum in Farbkodierungen des Symbols an der Benutzeroberfläche umgesetzt werden.

6.5.6 Polling des *somdd*

Um DSOM-Server bei Bedarf zu starten und um die Verbindungen zwischen Clients und Servern herstellen zu können, muß auf jedem Server-Rechner eine Instanz des DSOM-Dämons (*somdd*) laufen.
Der Zustand dieser Prozesse auf den einzelnen Rechnern muß also überwacht werden. Dazu wurde ein Programm entwickelt, das für jeden im Implementation Repository registrierten Rechner feststellt, ob auf ihm der *somdd*-Prozess läuft. Dieses Programm verwendet dazu die in [IBM 95a S.55] beschriebene Klasse *PingSdd*.
Das Programm kann als Hintergrundprozeß in regelmäßigen Abständen ausgeführt werden und somit alle *somdd*-Prozesse überwachen.
Eine Alternative zu diesem Polling-Mechanismus besteht darin, ein spezielles Start-Skript für den *somdd*-Prozess zu schreiben, das entsprechende Ereignismeldungen verschickt.

6.5.7 Integration des DSOM-Server-Managers in die NetView-Oberfläche

Der DSOM-Server-Manager ist ein Kommando, mit dem Serverprozesse verwaltet werden können. Seine Funktionalität umfaßt im wesentlichen das Starten und Stoppen von Servern (vgl. [IBM 94f S. 6-59]). Dabei müssen sich die Server nicht auf dem gleichen Rechner befinden, auf dem das Kommando aufgerufen wird.
Der DSOM-Server-Manager wurde in die NetView-Oberfläche integriert. Dadurch ist es möglich, Server zu starten und zu stoppen, indem man das Symbol des Servers anklickt und anschließend einen bestimmten Menüpunkt auswählt.

Kapitel 7

Zusammenfassung und Ausblick

7.1 Zusammenfassung

Der Einsatz der Object Management Architecture der OMG für das Management von Netzen, Systemen und Anwendungen ist momentan ein aktuelles Forschungsgebiet. Es beginnt sich jedoch abzuzeichnen, daß die OMA neben den „traditionellen" Managementarchitekturen Internet und OSI zukünftig vor allem im System- und Anwendungsmanagement eine Rolle spielen wird. Dies kann man an den Standardisierungsaktivitäten auf diesem Gebiet und an den Aussagen von Produktherstellern erkennen.
Momentan liegt ein Mangel an Managementplattformen vor, die diese neuen Konzepte unterstützen. Außerdem ist es zumindest mittelfristig unwahrscheinlich, daß sich eine der Architekturen gegenüber den anderen beiden völlig durchsetzen wird. Dazu sind die Anforderungen im Management und die Stärken und Schwächen der Architekturen zu unterschiedlich.
Um trotzdem ein integriertes Management aller Ressourcen eines verteilten Systems zu ermöglichen, muß die Forderung nach einer Managementplattform gestellt werden, die als Basiswerkzeug für Internet-, OSI- und OMA-Management eingesetzt werden kann.
In dieser Arbeit wurde ein Konzept entwickelt, mit dem CORBA-konforme Object Request Broker in eine herkömmliche Netzmanagementplattform integriert werden können. Der Schwerpunkt lag dabei auf der Nutzbarmachung der Plattformdienste (Verarbeitung von Ereignismeldungen und Topologiedarstellung inkl. Zustandsüberwachung) für das Management von CORBA-Objekten.
Die Integration des Ereignismanagements (Kapitel 5) ermöglicht, daß CORBA-Ereignismeldungen von der Plattform empfangen werden können. Dadurch können sie an der Benutzeroberfläche angezeigt, nach bestimmten Kriterien gefiltert und in Log-Dateien gespeichert werden. Außerdem können CORBA-Anwendungen sich bei der Plattform für den Empfang von bestimmten gefilterten Ereignismeldungen registrieren. Das ist die Grundvoraussetzung für eine Automatisierung des Ereignismanagements.
Das Konzept beruht auf dem CORBA-Event-Service und verwendet spezielle IDL-Definitionen für Management-Events sowie Event-Channel Objekte zum Vermitteln der Ereignismeldungen. Die Verarbeitung durch die Plattform wird durch Umwandlung in SNMP-Traps (bzw. CMIP-Event-Reports) ermöglicht.
Fehlermeldungen werden automatisch beim Eintreten von Exceptions während Methodenaufrufen auf Objekten erzeugt und an die Plattform geschickt. Dadurch ist die Grundlage für das Fehlermanagment in einem aus verteilten Objekten bestehenden System realisiert.

Die Integration des Topologiemanagements (Kapitel 6) ermöglicht, daß ein Modell des CORBA-Systems an der Benutzeroberfläche der Plattform dargestellt und überwacht werden kann. Hierfür wird zunächst durch eine Discovery-Anwendung Information über die Ressourcen des CORBA-Systems gesammelt und in der Plattformdatenbank abgespeichert. Die geeigneten Informationsquellen für die Discovery wurden dabei teilweise selbst implementiert. Die Plattformdatenbank wird durch die CORBA-Discovery zu einer integrierten Informationsbasis erweitert, in der sowohl die IP- (bzw. SNMP), als auch die OSI- (durch TMN-Discovery) und CORBA-Ressourcen registriert sind.
Zur Präsentation der Topologieinformation an der Benutzeroberfläche wurden verschiedene Modelle entwickelt. Neben einem Modell für das Systemmanagement, dessen zentrale Bestandteile die vorhandenen Rechner und die in ihnen enthaltenen Managementobjekte sind, wurde auch ein Modell für die Überwachung von CORBA-Anwendungen entworfen. Damit können die Referenzbeziehungen zwischen den verteilten Objekten dargestellt werden. Die grafische Darstellung der Topologie wurde durch die Integration eines DSOM-spezifischen Werkzeugs zum Verwalten von Serverprozessen (Starten, Stoppen, etc.) in die Plattformoberfläche ergänzt.
Für die Aktualisierung der Topologie- und Zustandsdaten wurde anstelle des von SNMP bekannten Pollings ein ereignisgesteuerter Ansatz entwickelt, so daß überflüssiger Netzverkehr vermieden wird.
Zusammenfassend kann man sagen, daß NetView für AIX mit den in dieser Arbeit entwickelten und implementierten Erweiterungen nicht mehr nur für Internet- und OSI-Management sondern auch als Basiswerkzeug für OMA-Management eingesetzt werden kann.

7.2 Ausblick

Die folgende Liste soll als Anregung dienen, wie die entworfenen Komponenten weiterentwickelt werden können, bzw. wie zusätzliche CORBA-spezifische Managementfunktionalität in die Plattform integriert werden kann:

- Die Kapselung des GTM-API durch SOM-Klassen kann vervollständigt werden, damit SOM-Objekte unabhängig von ihrer Lokation auf die Information in der GTM-Datenbank zugreifen können (vgl. Kapitel 6).
- Die zur Weiterleitung von Ereignismeldungen verwendeten Event_Dispatcher können um Filter- und Log-Funktionen erweitert werden.
- Bei dem entwickelten Konzept spielt das DSOM-Serverobjekt, das in jedem Server vorhanden ist, eine wichtige Rolle. Es ist für die Registrierung von Objekten und Proxies, für das Versenden von Ereignismeldungen und für die Fehlerüberprüfung (Exceptions) bei Methodenaufrufen zuständig. Speziell der in Kapitel 5 präsentierte Algorithmus zur Fehlerüberprüfung kann, wie dort schon beschrieben wurde, auch für die Integration von Performance-Messungen, Accounting und Security-Funktionen in das Serverobjekt verwendet werden. Die Aktivierung und Konfiguration dieser Funktionen und der Zugriff auf die von ihnen gelieferten Daten sollte möglichst interaktiv von der Plattformoberfläche aus erfolgen können.
- CORBA-Events können in CMIP-Event-Reports statt in SNMP-Traps umgewandelt werden, um die in Kapitel 5 genannten Vorteile nutzen zu können.

- Die bestehenden Klassen für die Umwandlung von SNMP-Traps in CORBA-Events können erweitert werden, um auch CMIP-Event-Reports in CORBA-Events umzuwandeln. Dadurch kann die Grundlage für eine Anwendung geschaffen werden, die Ereignismeldungen aus unterschiedlichen Managementszenarien wie z.B. LAN-Management (SNMP), WAN-Management (CMIP) und System- bzw. Anwendungsmanagement (CORBA) empfangen und korrelieren kann (vgl. auch Abb. 7.1).

- Eine weitere wichtige Komponente wäre ein Werkzeug, mit dem man interaktiv von der Plattformoberfläche aus Methodenaufrufe auf Objekten durchführen kann. Hierfür ist eine Funktionalität denkbar, die der eines SNMP-MIB-Browsers ähnlich ist und die durch einen Interface-Repository-Browser mit eingebautem Zugriff auf das *Dynamic Invocation Interface* des ORBs realisiert wird. Um dynamische Methodenaufrufe ausführen zu können, muß die Objektreferenz des Zielobjekts bekannt sein. Diese ist in der Plattformdatenbank (in String-Form) für jedes CORBA-Objekt gespeichert. Diese Zugriffsfunktion auf Objekte bildet die Grundlage für die Konfiguration von Ressourcen, die durch CORBA-Objekte repräsentiert werden.

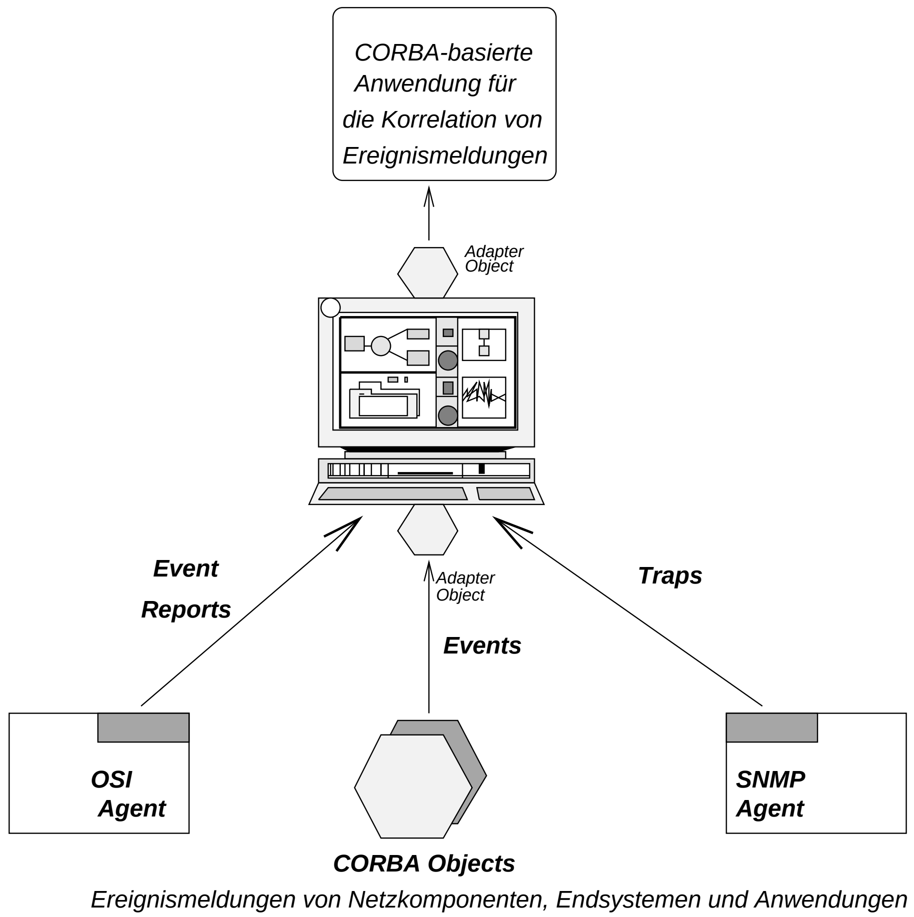

Abbildung 7.1: Integriertes Ereignismanagement

Anhang A

Alternative Ansätze im Netz- und Systemmanagement

1988 wurde von der Internet Engineering Task Force (IETF) das Simple Network Management Protocol (SNMP) entwickelt. SNMP war ursprünglich nur als kurzfristige Zwischenlösung gedacht. Langfristig wollte die IETF das OSI-Protokoll CMIP (Common Management Information Protocol) einsetzen. Allerdings wurde dieser Plan mittlerweile zugunsten der Standardisierung von SNMPv2 aufgegeben. SNMP-basiertes Internet Management ist weniger komplex, aber auch weniger leistungsfähig als OSI-Management.
SNMPv1 ist heute das am weitesten verbreitete Managementprotokoll und hat sich vor allem im Bereich der privaten Datennetze zu einem Industriestandard entwickelt. Auch die meisten Managementplattformen, die heute erhältlich sind, verwenden SNMP, um auf managementrelevante Information zuzugreifen. In öffentlichen Datennetzen ist hingegen die OSI-Managementarchitektur verbreitet.
SNMP-basiertes Internet-Management hat trotz seiner weiten Verbreitung einige Schwächen:

1. Mit SNMP ist zwar das Überwachen von Ressourcen möglich (durch Polling von der Managementstation); die Steuermöglichkeiten sind jedoch sehr begrenzt.

2. Das Protokoll ist ineffizient bei der Übertragung großer Datenmengen. Bei der Abfrage von großen Tabellen (z.B. Wegewahltabellen) sind sehr viele get-requests nötig. Dadurch wird die Netzlast erhöht.

3. Asynchrone Ereignisse werden zu wenig unterstützt. Der einfache Trap-Mechanismus erlaubt kein komplexes ereignisgesteuertes Management. Durch die ständigen Statusabfragen, die deshalb nötig sind, wird die Netzlast erhöht.

4. Das Sicherheitskonzept ist nicht ausreichend. Die Authentifizierung erfolgt lediglich durch community-strings, die leicht abhörbar bzw. fälschbar sind. Aus diesem Grund wird SNMP in sicherheitskritischen Umgebungen, wie z.B. bei Banken, nur selten für Steuerungszwecke eingesetzt.

5. Durch das verwendete Transportprotokoll UDP (ein verbindungsloser Datagrammdienst) kann es zu Datenverlusten kommen.

6. Die Heterogenität der Netzkomponenten spiegelt sich in der Vielzahl der herstellerspezifischen MIBs (Management Information Base) wider. Im Gegensatz zum objektorientierten OSI-Informationsmodell ist keine Vererbung und Wiederverwendbarkeit von Managementinformation möglich.

Zusammenfassend kann man sagen, daß SNMP zwar für das Management einfacher Netzkomponenten geeignet ist, aber nicht für die Anforderungen ausreicht, die das Management großer Corporate Networks oder Client-Server-Systeme mit sich bringt.
Einige der Mängel des Internet-Managements wurden in SNMP-Version2 behoben. Diese Verbesserungen umfassen u.a. (vgl. z.B. [SEITZ] S.49 ff.):

- Es wurde ein Authentifikationsmechanismus zum Erlangen eines höheren Sicherheitsgrads eingeführt. Hierfür wird der *Message Digest 5* Algorithmus (MD5) verwendet. Zusätzlich ist die Verschlüsselung von Nachrichten nach dem *Data Encryption Standard* (DES) möglich.

- Eine Erweiterung der möglichen Managementinformation. Es wurden die neuen MIB-Teilbäume *security(5)* und *snmpV2(6)* eingeführt. Außerdem können mittels sogenannter *Textual Conventions* neue SMI-Datentypen definiert werden. Desweiteren ist durch die Einführung von *Notification Type Macros* die Definition von Meldungen möglich, die sowohl von einem Agenten als auch von einem Manager ausgehen können.

- Die Kommunikation zwischen Managementstationen ist möglich. Dadurch können Stationen realisiert werden, die einerseits als Manager und andererseits als Agent tätig sind. In einer *Manager-to-Manager MIB* ([RFC 1451]) wird festgelegt, wann der Informationsaustausch zwischen Managern stattfinden muß.

- Es wurde eine neue Operation mit dem Namen *Get-Bulk* definiert. Diese ist eine Weiterentwicklung der Get-Next Operation. Die wichtigste Verbesserung betrifft den Transport der Antwortdaten eines Requests. Im Gegensatz zu einer normalen Get- oder Get-Next Operation besteht hier keine Beschränkung bezüglich der Menge der Antwortdaten mehr.

SNMPv2 hat zwar einen Teil der Schwächen von SNMP beseitigt, ist aber auch um einiges komplexer als seine Vorgängerversion. In der Praxis ist es momentan noch sehr wenig verbreitet.
OSI-Management ist zwar wesentlich leistungsfähiger als Internet-Management, aber auch komplexer, was der Hauptgrund für seine geringe Verbreitung ist. Trotzdem gibt es einige Managementplattformen, die diese Standards unterstützen (z.B. TMN Workbench und Support Facility von IBM). Ein Mangel herrscht jedoch vor allem an Agentensystemen und Netzwerkelementen, die eine CMIP Schnittstelle implementiert haben. Die meisten CMIP-fähigen Komponenten sind dem WAN-Bereich zuzuordnen, wie z.B. Multiplexer oder ATM-Switches (vgl. [PETRIK]).

Anhang B

Der Informationsbaustein von NetView für AIX

Der Informationsbaustein von NetView für AIX besteht aus einer Reihe von Datenbanken, in denen managementrelevante Information gespeichert wird. Dies sind:

- IP-Topologie-Datenbank
- Objekt-Datenbank
- GTM-Datenbank
- Map-Datenbank
- Trap-Datenbank

Die *IP-Topologie* Datenbank enthält Information über die Komponenten und Objekte, die während der IP-Discovery ermittelt wurden.
Hier ein Auszug aus einer IP-Topologie-Datenbank:

```
CLASS    OBJECT ID  OBJECT                  STATUS     IP ADDRESS
TOPOINFO  514       IP Internet
NETWORKS  1521      129.187.13            Unmanaged    129.187.13.0
          525       129.187.214           Marginal     129.187.214.0
          553       129.187               Marginal     129.187.0.0
          555       131.159               Up           131.159.0.0
          569       129.187.236           Up           129.187.236.0
SEGMENTS  1522      129.187.13.Segment1   Unmanaged
          554       129.187.Segment1      Marginal
          556       131.159.Segment1      Up
NODES     1515/1514 bro1cz.lrz-muenchen.de Up          129.187.4.251
          1519/1518 hp20.lrz-muenchen.de  Up           129.187.16.33
          1519/1520 hp20.lrz-muenchen.de  Up           129.187.13.23
```

Die Objekte der IP-Topologie-Datenbank sind IP-Netze, Segmente und Netzknoten. Als Attribute sind neben den internen Identifikatoren die IP-Adressen und der momentane Status der Objekte enthalten.

In der GTM-Datenbank ist Information über die Topologie von nicht-IP Protokollen gespeichert. Die Einträge in dieser Datenbank werden vom *General Topology Manager* vorgenommen, über dessen APIs man Anwendungen für die Discovery von nicht-IP Protokollen erstellen kann (vgl. Kapitel 3).
Die Objekt-Datenbank beinhaltet Informationen über alle Ressourcen, die über die IP-Discovery und über den General Topology Manager entdeckt wurden. Außerdem können Einträge durch Benutzerprozesse in dieser Datenbank vorgenommen werden. Im Vergleich zur IP-Topologie-Datenbank ist die Information wesentlich detaillierter. Die Objekt-Datenbank ist die zentrale Informationsbasis der Plattform, in der für jede Ressource, die ihr bekannt ist, ein Objekt enhalten ist. Dabei spielt es keine Rolle, über welche Discovery-Funktion (IP- oder selbstentwickelte) die Ressource entdeckt wurde. Der GTM erzeugt für jedes Objekt, das in seiner GTM-Datenbank erzeugt wird, automatisch auch ein Objekt in der Objekt-Datenbank. Als Beispiel ist hier ein Teil der Attribute gezeigt, die für einen durch die IP-Discovery entdeckten Rechner in der Objekt-Datenbank gespeichert werden:

```
OBJECT: 516

FIELD ID        FIELD NAME                      FIELD VALUE
10              Selection Name                  "ibmhegering1"
11              IP Hostname                     "ibmhegering1"
14              OVW Maps Exists                 2
15              OVW Maps Managed                2
19              IP Status                       Normal(2)
22              isIPRouter                      FALSE
33              vendor                          IBM(1)
43              isNode                          TRUE
45              isComputer                      TRUE
46              isConnector                     FALSE
47              isBridge                        FALSE
48              isRouter                        FALSE
49              isHub                           FALSE
52              isWorkstation                   TRUE
67              isIP                            TRUE
68              isSNMPSupported                 TRUE
70              SNMP sysDescr                   "SunOS 4.1.4"
71              SNMP sysLocation                "UCDavis Departement"
72              SNMP sysContact                 "support"
73              SNMP sysObjectID                "1.3.6.1.4.1.3.1.1"
74              SNMPAgent                       IBM X Station(8)
78              isMLM                           FALSE
79              isSYSMON                        FALSE
80              isSIA                           FALSE
84              TopM Interface Count            1

106             XXMAP Protocol List             "IP"
527             IP Name                         "ibmhegering1"
576             default IP Symbol List          13
```

Neben dem Namen des Rechners, der Anzahl der Maps, in denen ein Symbol für ihn enthalten ist und Klassifizierungsattributen (z.B. isRouter) sind auch die Werte bestimmter MIB-Variablen gespeichert (z.B. sysDescr).
Die Map-Datenbank enthält Information über die grafische Darstellung der Maps. Dazu gehören die Plazierung der Symbole, welche Symbole zu welchem Objekt gehören und die Kennzeichnungen (Labels) der Symbole. Für jede Map wird dabei eine eigene Datenbank geführt. Deren Inhalt kann durch Benutzer aktualisiert werden (z.B. durch Hinzufügen neuer Symbole) oder durch interne Netview-Prozesse (z.B. Discovery).

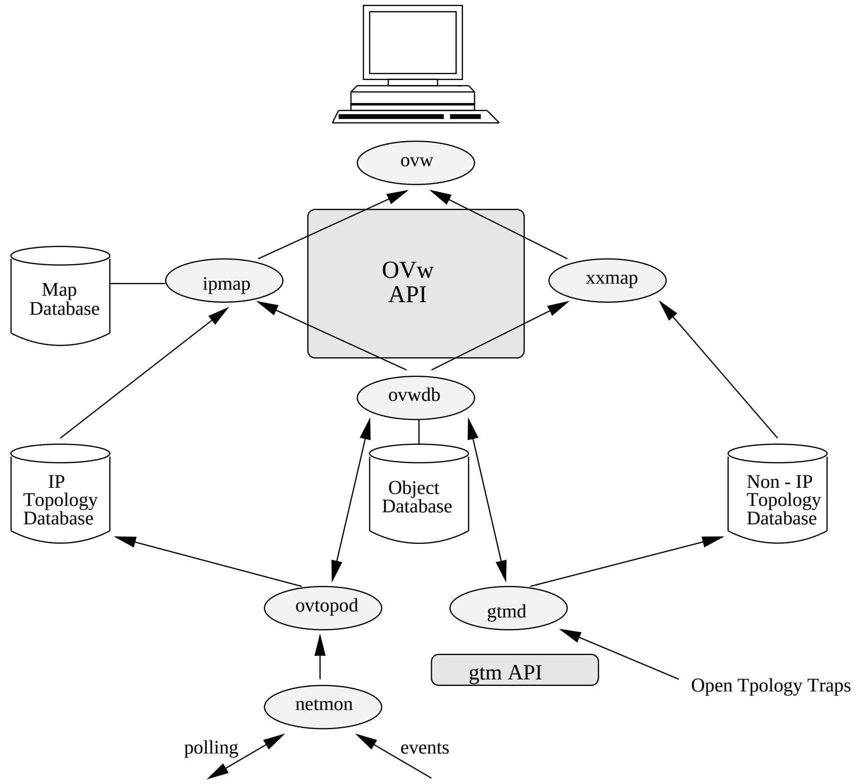

Abbildung B.1: Die NetView-Datenbanken

Weitere Datenbanken: Außerdem gibt es noch eine Trap-Datenbank, in der alle empfangenen Traps vom trapd Prozeß gespeichert werden, sowie eine Datenbank, in der Information über MIB-Variablen gespeichert ist, die von der *snmpCollect* Anwendung gesammelt wurde.
Einsatz von relationalen Datenbanken: Normalerweise sind alle Datenbanken von Netview für AIX als einfache Dateien realisiert. Folgende Information kann jedoch optional auch in relationalen Datenbanken gespeichert werden:

- IP Topologie
- snmpCollect Daten
- Trap Datenbank

Hierfür werden die relationalen Datenbanken von Ingres, Informix, Oracle, Sybase und IBMs DB2/6000 unterstützt.

Anhang C

IDL-Spezifikation der entwickelten Komponenten

C.1 Die Schnittstelle zur NetView GTM-Datenbank

Die Klassen „kapseln" das GTM-API damit DSOM-Clients unabhängig von ihrer Lokation Objekte in der Datenbank erzeugen können. Es wurde eine Klasse für jede Komponente der GTM-MIB entwickelt.

C.1.1 Die Klasse GTM

Die Klasse dient zum Öffnen und Schließen von Sessions mit dem GTM-Dämon (gtmd).

```
#include <somobj.idl>

interface GTM : SOMObject
{

  void NvotInit();

  void NvotDone();

 #ifdef __SOMIDL__
        implementation
        {
        dllname="gtm.dll";
        };
 #endif /* __SOMIDL__ */
};
```

C.1.2 Die Klasse Graph

Damit können Graphen in der GTM-Datenbank erzeugt und gelöscht werden.

```
#include <somobj.idl>
```

```
interface Graph : SOMObject
{
  void create_as_root(in string graphName,
                      in string label);

  void create_g_in_graph(in string parentName,
                         in string graphName,
                         in string label);

  void delete_graph(in string graphName);

#ifdef __SOMIDL__
implementation
{
dllname="gtm.dll";
};
#endif /* __SOMIDL__ */
};
```

C.1.3 Die Klasse Box

Die Klasse repräsentiert die Komponente `Box` des GTM-Datenmodells.

```
#include <somobj.idl>

interface Box : SOMObject
{
void create_b_in_graph(in string parentName,
                       in string boxName,
                       in string label);

void delete_box(in string boxName);

#ifdef __SOMIDL__
        implementation
        {
        dllname="gtm.dll";
        };
#endif /* __SOMIDL__ */
};
```

C.1.4 Die Klasse Arc

Die Klasse Arc dient zum Erzeugen und Löschen von Arcs in der GTM-Datenbank.

```
#include <somobj.idl>

interface Arc : SOMObject
```

```
{
    void create_arc_in_graph(in string graphName,
                             in string aEndpointName,
                             in string zEndpointName,
                             in string label);

    void delete_arc (in string aEndpointName,
                     in string zEndpointName);

 #ifdef __SOMIDL__
        implementation
        {
        dllname="gtm.dll";
        };
#endif /* __SOMIDL__ */
};
```

C.1.5 Die Klasse Vertex

Diese Klasse stellt Methoden zum erzeugen und löschen von Vertices in der GTM-Datenbank und zum ändern ihres Status zur Verfügung.

```
#include <somobj.idl>

interface Vertex : SOMObject
{
void create_in_box(in string boxName,
                   in string vertexName,
                   in string label);

void create_v_in_graph(in string graphName,
                       in string vertexName,
                       in string label);

void delete_vertex(in string vertexName);

void change_vertex_status(in string vertexName,
                          in string vertexStatus);


#ifdef __SOMIDL__
        implementation
        {
        dllname="gtm.dll";
        };
#endif /* __SOMIDL__ */
};
```

C.1.6 Die Klasse Sap

Diese Klasse repräsentiert die Service Access Points des GTM-Datenmodells.

```
#include <somobj.idl>

interface Sap : SOMObject
{
void create_providing_sap(in short vertexProtocol,
                          in string vertexName,
                          in short sapProtocol,
                          in string sapName);

void create_using_sap(in short vertexProtocol,
                      in string vertexName,
                      in short sapProtocol,
                      in string sapName);

void delete_providing_sap(in short vertexProtocol,
                          in string vertexName,
                          in short sapProtocol,
                          in string sapName);

void delete_using_sap(in short vertexProtocol,
                      in string vertexName,
                      in short sapProtocol,
                      in string sapName);


         #ifdef __SOMIDL__
        implementation
        {
        dllname="gtm.dll";
        };
#endif /* __SOMIDL__ */

};
```

C.2 Die Komponenten für das Ereignismanagement

C.2.1 Die Klasse Event_consumer

Dies ist die Basisklasse für alle Empfänger von Ereignismeldungen.

```
#include <somobj.idl>
#include <snglicls.idl>
#include <mcollect.idl>
```

```
interface Event_consumer : somf_MCollectible
{
        /*indicates creation of dsom object*/
        void object_created(in string ev_type,
                            in string host,
                            in string server_id,
                            in string server_alias,
                            in string objclass,
                            in string objref);

        /*indicates deletion of dsom object*/
        void object_deleted(in string ev_type,
                            in string host,
                            in string server_id,
                            in string server_alias,
                            in string objclass,
                            in string objref);

        /*indicates creation of a proxy*/
         void proxy_created(in string ev_type,
                            in string host,
                            in string server_id,
                            in string server_alias,
                            in string proxyclass,
                            in string proxyref);



         /*indicates deletion of a proxy*/
          void proxy_deleted(in string ev_type,
                             in string host,
                             in string server_id,
                             in string server_alias,
                             in string proxyclass,
                             in string proxyref);




        /*indicates that a new server has been
                  added to the implementation repository*/
        void server_added(in string ev_type,
                          in string host,
                          in string server_id,
                          in string server_alias);

        /*indicates that a server has been deleted
```

```
                   from the implementation repository*/
        void server_deleted(in string ev_type,
                            in string host,
                            in string server_id,
                            in string server_alias);

        /*indicates that a server-process has been activated*/
        void server_active(in string ev_type,
                           in string host,
                           in string server_id,
                           in string server_alias);

        /*indicates that a server-process has bee deactivated*/
        void server_inactive(in string ev_type,
                             in string host,
                             in string server_id,
                             in string server_alias);


         void AnException(in string ev_type,
                          in string host,
                          in string server_id,
                          in string server_alias,
                          in string objclass,
                          in string objref,
                          in string method,
                          in string exceptionname);


#ifdef __SOMIDL__
        implementation
        {
          dllname="consumers.dll";
          metaclass= SOMMSingleInstance;

        };
#endif /*__SOM_IDL__*/
};
```

C.2.2 Die Klasse Event_Dispatcher

Die Instanzen dieser Klasse werden anstelle von Event-Channels zum vermitteln von Ereignismeldungen verwendet.

```
#include <Event_consumer.idl>
#include <snglicls.idl>
```

```
interface Event_Dispatcher : Event_consumer
{
void register_consumer(in Event_consumer consumer);
void consumer_disconnect(in somf_MCollectible consumer);

#ifdef __SOMIDL__
implementation
{
dllname = "consumers.dll";
metaclass = SOMMSingleInstance;

object_created: override;
object_deleted: override;
proxy_created: override;
proxy_deleted: override;
server_added: override;
server_deleted: override;
server_active: override;
server_inactive: override;
AnException: override;
somInit: override;
somUninit: override;
};
#endif /*__SOMIDL__*/
};
```

C.2.3 Die Klasse EMS_Event_consumer

Die Instanzen dieser Klasse empfangen CORBA-Ereignismeldungen und wandeln sie in SNMP-Traps um.

```
#include <Event_consumer.idl>
#include <snglicls.idl>

interface EMS_Event_consumer : Event_consumer
{

/* no methods added*/

#ifdef __SOMIDL__
implementation
{
dllname = "consumers.dll";
metaclass = SOMMSingleInstance;
```

```
object_created: override;
object_deleted: override;
proxy_created: override;
proxy_deleted: override;
server_added: override;
server_deleted: override;
server_active: override;
server_inactive: override;
AnException: override;
};
#endif /*__SOMIDL__*/
};
```

C.2.4 Die Klasse Event_Adapter

Die Klasse wird zum Konvertieren von CORBA-Ereignismeldungen in SNMP-Traps verwendet.

```
#include <somobj.idl>

interface Event_Adapter : SOMObject
{

void create_pdu(in long spec_id, in string source_host);

void add_arg_long(in long arg);

void add_arg_string(in string arg);

void send_pdu();


#ifdef __SOMIDL__
    implementation
    {
    dllname = "consumers.dll";
    };
#endif /* __SOMIDL__ */
};
```

C.2.5 Die Klasse GTM_Event_consumer

Die Instanzen dieser Klasse aktualisieren die Topologieinformation aufgrund von Ereignismeldungen.

```
#include <Event_consumer.idl>
#include <snglicls.idl>
```

```
interface GTM_Event_consumer : Event_consumer
{

/* no methods addded*/

#ifdef __SOMIDL__
implementation
{
dllname = "consumers.dll";
metaclass = SOMMSingleInstance;

object_created: override;
object_deleted: override;
proxy_created: override;
proxy_deleted: override;
server_added: override;
server_deleted: override;
server_active: override;
server_inactive: override;
somInit: override;
somUninit: override;
};
#endif /*__SOMIDL__*/
};
```

C.2.6 Die Klasse EFD_Supplier

Die Instanzen dieser Klasse dienen als Quelle für von der Plattform gefilterte Ereignismeldungen.

```
#include <somobj.idl>

interface Event_Dispatcher;

interface EFD_Supplier : SOMObject
{
attribute string filter;

void set_Dispatcher(in Event_Dispatcher dispatcher);

oneway void activate();

#ifdef __SOMIDL__
implementation
{
dllname = "efd.dll";
```

```
filter: noset;
};
#endif /*__SOMIDL__*/
};
```

C.3 Die Erweiterung der DSOM-Laufzeitkomponenten

C.3.1 Die Erweiterung der Klasse ITSCSOMDServer

Dies ist eine Erweiterung der Klasse ITSCSOMDServer aus [IBM 95a].

```
#ifndef ITSCSOMDServer_idl
#define ITSCSOMDServer_idl

#include <somobj.idl>
#include <somdserv.idl>
#include <somdtype.idl>
#include <snglicls.idl>
#include "reflitem.idl"
#include "refstrct.idl"
interface RefLinkList;
interface Repository;

interface ITSCSOMDServer : SOMDServer
{

    /*-------------------------------*/
    /*       Exceptions              */
    /*-------------------------------*/

  exception ITSCSERV_ERR {long errCode;
                        char Reason[80];};

    /*-------------------------------*/
    /*           Methods             */
    /*-------------------------------*/



    long GetAllRegisterClasses( out sequence<string> clsList)
                                raises (ITSCSERV_ERR);

    long GetAllUserClasses( out sequence<string> clsList)
                            raises (ITSCSERV_ERR);

    long GetAllAffinityGroup(in string class_name,
                             out sequence<string> clsList)
```

```
                              raises (ITSCSERV_ERR);

    long GetClassStatus(in string class_name)
                        raises (ITSCSERV_ERR);

    long GetClassTree(in string class_name,
                      out sequence<string> clsList)
                      raises (ITSCSERV_ERR);

    long GetInstancesForClass (in string class_name,
            out sequence<RefStruct::refData> clsList)
            raises (ITSCSERV_ERR);

    long GetClassAncestors (in string class_name,
                            out sequence<string> clsList)
                            raises (ITSCSERV_ERR);

    sequence<RefStruct::refData> GetRefData();

    sequence<RefStruct::refData> GetProxyData();

    void AddProxyToList(in string ref, in string class);

    void DeleteProxyFromList(in string ref, in string class);

    void verifyList(in RefLinkList refLList);

    void object_created(in string server,
                        in string objref,
                        in string class);

    void object_deleted(in string objref,
                        in string class);

    /*------------------------------*/
    /*       Implementation         */
    /*------------------------------*/

#ifdef __SOMIDL__

  implementation
  {
    passthru C_h = "#include \"refstrct.h\"";
    passthru C_xh = "#include \"refstrct.xh\"";

    // Class Modifiers
    releaseorder: GetRefData,
```

```
                  GetAllRegisterClasses,
                  GetAllUserClasses,
                  GetAllAffinityGroup,
                  GetClassStatus,
                  GetClassTree,
                  GetInstancesForClass,
                  GetClassAncestors,
                  verifyList,
                  object_created,
                  object_deleted
                 ;

    dllname = "itscserv.dll";
    metaclass = SOMMSingleInstance;

            /*------------------------------*/
            /*      Method Overrides        */
            /*------------------------------*/

    // Method modifiers
    somdRefFromSOMObj:  override;
    somdSOMObjFromRef:  override;
    somdDispatchMethod: override;
    somInit:            override;
    somUninit:          override;
    somdCreateObj:      override;
    somdDeleteObj:      override;
            /*------------------------------*/
            /*     Instance Variables       */
            /*------------------------------*/
    Repository  repo;
    RefLinkList refLList; /*list of objects*/
    RefLinkList proxyList; /*list of proxies*/

  };
  #endif /* __SOMIDL__ */
};
#endif
```

C.3.2 Die Klasse MgmtObjectMgr

Die Instanzen dieser Klasse registrieren erzeugte Proxy-Objekte beim Serverobjekt (ITSC-SOMDServer oder Subklasse).

```
#ifndef MgmtObjectMgr_idl
```

```
#define MgmtObjectMgr_idl

#include <somobj.idl>
#include <somdom.idl>

interface MgmtObjectMgr : SOMDObjectMgr
{

  implementation
{
  somdNewObject: override;
  somdDestroyObject: override;
  somdReleaseObject: override;

  dllname = "MgmtObjectMgr.dll";
};
};
#endif
```

C.3.3 Die Klasse MImplRepository

Diese Erweiterung der Schnittstelle zum Implementation Repository sendet Ereignismeldungen beim registrieren und löschen von Servern aus.

```
#ifndef MImplRepository_idl
#define MImplRepository_idl

#include <implrep.idl>

interface MImplRepository : ImplRepository
{
void Server_Added(in string server_host,
                  in string server_id,
                  in string server_alias);

void Server_Deleted(in string server_host,
                    in string server_id,
                    in string server_alias);

 implementation
 {

   add_impldef: override;
   delete_impldef: override;
   dllname = "mimplrep.dll";
   };
};
#endif
```

Literaturverzeichnis

[ABECK 94] Sebastian Abeck. *Management Plattforms: Integration Techniques.* Proceedings of the 1st Workshop of the HP European Academic Network Management Forum, June 94.

[ABECK 95] Sebastian Abeck. *Integrationstechniken im Netzmanagement.* GI/ITG Fachtagung Kommunikation in verteilten Systemen, February 95.

[BERNSTEIN] Lawrence Bernstein, C.M. Yuhas. *Truce in Protocol Wars.* Journal of Network and Systems Management, Vol. 1, No. 2, 1993

[CMIP RUN a] CMIP Run: A quarterly Newsletter covering Network Management Technology and Events. *Comparing SNMP and CMIP.* CMIP Run Vol.4, Number 1

[CMIP RUN b] CMIP Run Newsletter. *XOM and XMP for Programmers.*

[GERING] Michael Gering. *CMIP versus SNMP.* In: H.-G. Hegering, Y.Yemini (eds.): Integrated Network Management III, North Holland Publ. 1993.

[HEGE 93] Heinz-Gerd Hegering, Sebastian Abeck. *Integriertes Netz- und Systemmanagement.* Addison Wesley, 1993. 1. Auflage.

[HEGE 94] Heinz-Gerd Hegering. *Management von Client/Server-basierten Systemumgebungen.* CW FOCUS. Computerwoche Verlag, München, August 94.

[HIERRO] Juan J. Hierro. *Common Facilities for OSI Management.* Telefonica Investigacion y Desarollo. June 20th, 1995.

[HOFFNER 94a] Hoffner, Y. *Monitoring in Distributed Systems.* Architecture Report: APM.1008.01, October 1994, APM Ltd., Poseidon House, Castle Park, Cambridge.

[HOFFNER 94b] Hoffner, Y. *Management in Object-Based Federated Distributed Systems.* Architecture Report: APM.1018.01, October 1994, APM Ltd., Poseidon House, Castle Park, Cambridge.

[HOFFNER 94c] Hoffner, Y. *Visualisation of Distributed Systems.* Architecture Report: APM.1019.01, October 1994, APM Ltd., Poseidon House, Castle Park, Cambridge.

[HP] Hewlett Packard. *Developing Applications for the Distributed Management Platform HP Open View - Rev 4.0.* Lehrgangsunterlagen.

[IBM 93] IBM. *Examples of Using Netview/6000 APIs.* IBM Redbook GG24-4059-00. Juli 93

[IBM 94a] IBM. *SOMObjects: A practical Introduction to SOM and DSOM.* IBM Redbook GG24-4357-00. July 94

[IBM 94b] IBM. *Netview for AIX Programmer's Guide Version 3.* September 94.

[IBM 94c] IBM. *Netview for AIX Programmer's Reference Version 3.* September 94.

[IBM 94d] IBM. *Netview for AIX Administrator's Guide Version 3.* September 94.

[IBM 94e] IBM. *Netview for AIX Administrator's Reference Version 3.* September 94.

[IBM 94f] IBM. *SOMObjects Developer Toolkit: Users Guide Version 2.1.* October 94.

[IBM 94g] IBM. *SOMObjects Developer Toolkit: Quick Reference Guide Version 2.1.* October 94.

[IBM 95a] IBM *SOMObjects: Management Utilities for Distributed SOM.* IBM Redbook GG24-4479-00. April 95

[IBM 95b] IBM. *Netview for AIX Version4 Release Notes.* Mai 1995.

[IBM 95c] IBM. *IBM TMN WorkBench for AIX: Programmer's Guide.* September 19995.

[IBM 95d] IBM. *TMN Products for AIX: General Information.* September 1995.

[ISO/IEC 10165-2] *Information Technology - Open Systems Interconnection - Structure of Management Information -Part 2: Definition of Management Information:* 1992

[ISO/IEC 13235] *1994/ Draft ODP Trading Function.*

[KENE] Alexander Keller, Bernhard Neumair. *Sytems Management Middleware: Verteilte objektorientierte Technologien für das Systemmanagement.* In: Dieter Wall, editor, Proceedings der 11. GI Fachtagung Organisation und Betrieb von DV Versorgungsstrukturen. Vieweg, November 1995.

[NEUM] Bernhard Neumair. *Integriertes Management von Endanwendersystemen.* CW Focus. Computerwoche Verlag July 1995.

[NMF 96] Network Management Forum. *TMN C++ Application Programming Interface.* Issue 1.0. Draft 6 - For Public Comment. January 19996.

[OMG 92] Object Management Group. *Object Management Architecture Guide.* Revision 2.0. Richard Mark Soley, Ph.D. (ed.). OMG TC Document 92.11.1

[OMG 93a] Object Management Group. *The Common Object Request Broker: Architecture and Specification.* Revision 1.2. OMG Document Number 93.xx.yy

[OMG 93b] Joint Object Services Submission. *Naming Service Specification.* OMG TC Document 93.5.2, Revised May 14, 1993

[OMG 93c] Joint Object Services Submission. *Event Service Specification.* OMG TC Document 93.7.3, Revised July 2, 1993

[OMG 94] Object Management Group. *Common Object Services Specification (COSS)*, Volume 1. OMG TC Document 94-1-1 (1994).

[OMG 95] Object Management Group. *CORBA 2.0 Interoperability - Universal Networked Objects*. OMG TC Document 95.3.xx, March 20, 1995.

[ORFALI] Robert Orfali, Dan Harkey, Jeri Edwards. *Essential Client/Server Survival Guide.* John Wiley & Sons Inc., 1994

[PETRIK] Claudia Petrik. *Hohe Ziele - Anspruch und Wirklichkeit von Omnipoint.* Gateway September 1994.

[PÜRCK] Christoph Pürckhauer. *SOM als Basis wiederverwendbarer Komponenten.* Objekt Spektrum Mai/Juni 95.

[RFC 1451] Manager-to-Manager MIB.

[RUTT] Tom Rutt. *Comparison of the OSI management, OMG and Internet management Object Models.* A Report of the Joint XOpen/NM Forum Inter-Domain Management Task force. March 94.

[SEITZ] Jochen Seitz. *Netzwerkmanagement.* International Thomson Publishing 1994.

[SLOMAN] Morris Sloman (ed.). *Network and Distributed Systems Management.* Addison Wesley, 1994.

[STEVENS] W. Richard Stevens. *UNIX Network Programming.* P T R Prentice-Hall, 1990. ISBN 0-13-949876-1.

[TIVOLI 94] Tivoli Systems Inc. *TME 2.0: Technology Concepts And Facilities.* 1994.

[TIVOLI a] Tivoli Systems Inc. *Object Orientation Brings Advantages to Distributed Systems Management.* A White Paper.

[TIVOLI b] Tivoli Systems Inc. *A Perspective On Standards for Distributed Systems Management.* A White Paper.

[TIVOLI c] Tivoli Systems Inc. *A Comparison between Network Management and Systems Management for Today's Client/Server Environment.* A White Paper.

[TIVOLI d] Tivoli Systems Inc. *Tivoli Applications Management Specification (AMS).* A Tivoli Systems Position Paper.

[WARNE 95] John Warne. *Event Management for Large-Scale Distributed Systems.* Architecture Report: APM.1633.01, November 1995, APM Ltd., Poseidon House, Castle Park, Cambridge.

[X/OPEN 94] X/Open Company. *Systems Management: Management Protocols API (XMP).* X/Open CAE Specification. Document Number: C306

[X/OPEN 94b] X/Open Company. *Event Services for Systems Management - A White Paper.* October 1994.

[X/OPEN 95a] X/Open Company. *Inter-Domain Management Specifications: Specification Translation.* X/Open Preliminary Specification (Draft April 17, 1995).

[X/OPEN 95b] X/Open Company. *Systems Management: Common Management Facilities, Volume 1.* X/Open Document Number: P421.