INSTITUT FÜR INFORMATIK

DER
LUDWIG-MAXIMILIANS-UNIVERSITÄT
MÜNCHEN

Diplomarbeit

Christian Coehn

Entwurf eines verteilten Managementsystems für eine Email-Infrastruktur

Aufgabensteller	:	Prof. Dr. Heinz-Gerd Hegering
Betreuer	:	Boris Gruschke
		Stephen Heilbronner
Abgabedatum	:	17. August 1998

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

München, den 17. August 1998

.....
(Unterschrift des Kandidaten)

Zusammenfassung

Die *Total Cost of Ownership* eines Email-Systems hängt nicht von den Anschaffungskosten ab, sondern wird entscheidend vom Aufwand für den Betrieb des Systems bestimmt. Um diesen Aufwand niedrig zu halten, ist ein integriertes Management des Dienstes erforderlich. Hierzu wird ein einheitliches Managementmodell benötigt, welches mächtig genug ist, die Managementinformation und –funktionalität der verteilten Anwendung Email zu beschreiben.

In dieser Diplomarbeit wird ein Modell für die Beschreibung einer Email-Infrastruktur und deren Management entwickelt. Im Rahmen einer Top-Down Modellierung stützt sich das Modell auf generische Basisklassen für verteilte Systeme des *Reference Model for Open Distributed Processing* (RM-ODP). Ausgehend von der OSI-Definition einer Email-Architektur werden die Basisklassen für die Modellierung und das Management dieser Architektur verfeinert. Als Notationssprache kommt die objektorientierte *Object Modeling Technique* (OMT) zum Einsatz, welche unabhängig von bestehenden Management-architekturen ist.

Das entstandene Modell ist generisch und läßt sich sowohl auf verschiedene Email-Architekturen, als auch auf die verschiedenen Informationsmodelle der Management-architekturen abbilden.

In dieser Arbeit wird ein Ausschnitt des Modells auf Basis der *Common Object Request Broker Architecture* (CORBA) für den MTA *sendmail* prototypisch implementiert. Mittels eines CASE-Tools werden aus dem Objektmodell IDL-Beschreibungen der Managementschnittstellen erzeugt. Der Agent wird in Java mit Hilfe der CORBA Entwicklungs- und Laufzeitumgebung *Visibroker for Java* erstellt und stützt sich auf die vom MNM-Team entwickelte *Mobile Agent System Architecture* (MASA).

Ein zusätzlich entwickeltes Management-Applet ermöglicht den Zugriff auf den Agenten über eine graphische Oberfläche.

Inhaltsverzeichnis

1 Einführung 1
1.1 Die Bedeutung von Email 1
1.2 Aufgabenstellung 1
1.3 Vorgehensweise 2

2 Anforderungsanalyse für das Email-Management 5
2.1 Definition einer Email-Infrastruktur 5
2.1.1 Aufbau eines Message Handling Systems 5
2.1.2 Aufbau einer Nachricht 9
2.2 Managementaspekte eines MHS 10
2.2.1 Grundfunktionalität 10
2.2.1.1 Konfiguration 10
2.2.1.2 Fehler 10
2.2.1.3 Leistung 11
2.2.1.4 Abrechnung 12
2.2.1.5 Sicherheit 13
2.2.2 Weitere Funktionalität 14
2.2.3 Integration mit dem Systemmanagement 16

3 State of the Art im Emailmanagement 19
3.1 Herstellerspezifische Ansätze 19
3.1.1 Sendmail 19
3.1.2 Netscape Messaging Server 20
3.1.3 Weitere Produkte 23
3.2 Normierte Ansätze der IETF 23
3.2.1 Network Services Monitoring MIB 23
3.2.2 Mail Monitoring MIB 25
3.2.3 Message Tracking MIB 27
3.2.4 Bewertung 29
3.3 Normierte Ansätze der ISO 30
3.4 Einsatzszenario - die BMW AG 33
3.4.1 Infrastruktur 33
3.4.2 Management des Systems 34
3.5 Zusammenfassung 35

4 Entwicklung eines Managementmodells auf Basis von RM-ODP 37
4.1 RM-ODP als generische konzeptuelle Plattform 37
4.2 Modellierung mittels OMT und StP 40

4.2.1 Object Modeling Technique (OMT) ... 40
4.2.2 Software through Pictures (StP) ... 42
4.3 Modellierung der Email-Infrastruktur ... 44
4.3.1 Designentscheidungen ... 44
4.3.2 Enterprise Viewpoint ... 44
4.3.3 Informational Viewpoint ... 45
4.3.4 Computational Viewpoint ... 48
4.3.4.1 ODP-Basisklassen für den Computational Viewpoint ... 48
4.3.4.2 Modellierung des Computational Viewpoint ... 50
4.3.4.2.1 Der Message Transfer Agent (MTA) ... 55
4.3.4.2.2 Die lokale Mailbox und der Message Store (MS) ... 64
4.3.4.2.3 Der User Agent (UA) ... 66
4.3.4.2.4 Der Client ... 67
4.3.4.2.5 Der Directory Service ... 68
4.3.4.2.6 Die Benutzerverwaltung (MUser/MHSUser) ... 69
4.3.4.2.7 Die Modellierung einer Email und deren Fluß im System ... 70
4.3.4.2.8 MHSHistory ... 73
4.3.4.2.9 Zusammenfassung ... 73
4.3.5 Engineering Viewpoint ... 73
4.3.5.1 ODP-Basisklassen für den Engineering Viewpoint ... 73
4.3.5.2 Modellierung des Agentensystems ... 75

5 Prototypische Implementierung ... 79
5.1 Motivation ... 79
5.1.1 Das Problem unerwünschter kommerzieller Emails ... 79
5.1.2 Zugangskontrolle für nomadische Systeme ... 80
5.2 Konfiguration der Zugangskontrolle in sendmail 8.9 ... 81
5.3 Entwicklungsumgebung ... 82
5.3.1 Java ... 82
5.3.2 CORBA ... 84
5.3.3 Die *Mobile Agent System* Architektur ... 85
5.4 Der Agent MTAAccessDB ... 86
5.4.1 Funktionalität des Agenten ... 86
5.4.2 IDL-Schnittstelle ... 86
5.4.3 Implementierung in Java ... 88
5.4.4 Start des Agenten ... 88
5.5 Das Management Applet ... 89

6 Zusammenfassung und Ausblick ... 93

7 Anhang ... 97

7.1 OMT-Modelle....97
7.2 Implementierungsbeispiele....101

8 Abkürzungsverzeichnis....107

9 Literaturverzeichnis....109

Abbildungsverzeichnis

Abbildung 1 - OSI Message Handling Environment 6
Abbildung 2 - Message Handling System 6
Abbildung 3 - Nachrichtenfluß 8
Abbildung 4 - Envelope und Content 9
Abbildung 5 – Netscape Mission Control Architektur (aus [Nets98]) 21
Abbildung 6 – Netscape Mission Control Desktop (aus [Nets98]) 21
Abbildung 7 – Netscape Proxy-Agent 22
Abbildung 8 – Network Services Monitoring MIB 24
Abbildung 9 – Mail Monitoring MIB 26
Abbildung 10 – Message Track Request 28
Abbildung 11 - TMN Schichtenmodell 30
Abbildung 12 - TMN Email Modell (aus [X460]) 31
Abbildung 13 - Email-Infrastruktur BMW AG 34
Abbildung 14 – Softwareentwicklung mit ODP 39
Abbildung 15 – OMT Klasse 41
Abbildung 16 – OMT Assoziation 41
Abbildung 17 – OMT Generalisierung 41
Abbildung 18 – OMT Aggregation 41
Abbildung 19 - Der StP Desktop 42
Abbildung 20 - StP Objektmodelleditor 43
Abbildung 21 - Informational Viewpoint 45
Abbildung 22 - ODP Basisklassen zum Computational Viewpoint (nach [Muel98]) 49
Abbildung 23 – Email-Komponenten im Computational Viewpoint 50
Abbildung 24 – Managementschnittstellen im Computational Viewpoint 51
Abbildung 25 - Kompletter Computational Viewpoint der Email-Infrastruktur 53
Abbildung 26 - ODP Basisklassen des Engineering Viewpoints (aus [Muel98]) 74
Abbildung 27 - Engineering Viepoint der prototypischen Implementierung 76
Abbildung 28 - Bibliotheksszenario 81
Abbildung 29 - Object Management Architecture (OMA) 84
Abbildung 30 - MASA Agentensystem und Agenten (aus [Kemp98]) 85
Abbildung 31 - Die Klasse MTAAccessDB im StP Klasseneditor 87
Abbildung 32 - Die Oberfläche des Management Applets 90
Abbildung 33 - ARM API (aus [Tivo98b]) 95

1 Einführung

1.1 Die Bedeutung von Email

Das Gut Information ist dabei, sich zum wichtigsten Wirtschaftsfaktor zu entwickeln. Parallel zum Anwachsen der Informationsmenge wächst auch der Bedarf, Information möglichst effektiv zu organisieren, zu verteilen und immer schneller zu transportieren. Die elektronische Post, Email genannt, spielt hierbei eine unternehmenskritische Schlüsselrolle und hat ihren Siegeszug längst angetreten. Im Gegensatz zu anderen Diensten, wie etwa elektronischer Dateiübertragung (z.B. FTP) oder Hypertext-Informationssystemen (z.B. WWW), deren Nutzung sich bei Mitarbeitern außerhalb IT-Kreisen gerade erst durchsetzt, ist der Umgang mit Email in fast allen Unternehmen zur Selbstverständlichkeit geworden.

Email erlaubt - im Idealfall - den schnellen, sicheren und unkomplizierten Transport von Informationen aller Art - von kurzen Texten bis hin zu multimedialen Daten wie Video oder Audio. Kommunikation via Email ist im Gegensatz zum Telefonieren asynchron. Der Partner muß also nicht direkt erreicht werden, sondern die zugestellte Information kann nach Bedarf abgerufen werden. Vielfach wird der Fluß von Email weiter automatisiert: Die Möglichkeiten reichen von einfachen Operationen, wie Umleitung bei Nichterreichbarkeit, automatischer Vorsortierung und Antwort bis zur Abbildung von Unternehmensprozessen auf ein regelbasiertes System (Workflow).

Eine Umfrage[1] der Unternehmensberatung Ernst & Young unter mehr als 400 amerikanischen Personalchefs hat ergeben, daß Email das Telefon als bevorzugtes Kommunikationsmedium überrundet hat. 36% der Befragten bevorzugten Email gegenüber dem Telefon (26%) oder persönlichen Meetings (15%). Ein Grund für die starke Akzeptanz ist die oben erwähnte Anpaßbarkeit von Email-Lösungen an Unternehmensbedürfnisse, die andere Internet-applikationen, wie das WWW noch nicht erreicht haben („While e-mail has emerged to address the communication challenges of today's organization, the Internet has not fully emerged as a solution for adressing broader business issues").

1.2 Aufgabenstellung

Wie alle Anwendungen im IT-Bereich ist auch die Applikation Email in den vergangenen 10 Jahren starken Änderungen unterworfen worden. Zunächst, zumindest im kommerziellen Bereich, häufig nur als firmeninternes Kommunikationsmedium eingesetzt, sollen Nachrichten heute weltumspannend in wenigen Minuten ausgetauscht werden können. Die Architektur eines Mailsystems hat sich dementsprechend von monolithischen Großrechnersystemen zu einer stark verteilten Client/Server-Anwendung geändert.

Mit zunehmender Heterogenität und Komplexität ist der Aufwand zum Betrieb von Email-Infrastrukturen jedoch sprunghaft angestiegen. Die Komplexität resultiert aus einer Vielzahl von Faktoren: Emailverteilung erfolgt nicht in Echtzeit, sondern nach dem

[1] http://www.ey.com/press/releases/040798.htm

Store-and-Forward Prinzip. Fehler treten deshalb zeitlich verzögert auf und sind oft schwer zu lokalisieren. Email wird außerdem durch eine Vielzahl von unterschiedlichen Technologien realisiert: Eine Nachricht kann auf ihrem Weg mehrmals Adreßformate und Inhalte wechseln. Email stützt sich weiterhin auf andere, oftmals selbst verteilte, Anwendungen ab. Ein Beispiel hierfür sind Verzeichnisdienste, wie X.500 oder DNS, deren Management ebenfalls sehr komplex ist.

Aus der Komplexität und damit der Kostenintensivität des Systems resultiert der Wunsch nach integriertem Management. Dies wurde mittlerweile sowohl von Produktherstellern, als auch von Standardisierungsgremien, wie der ISO oder der IETF aufgegriffen. Leider wiederholen sich gerade im Bereich des Anwendungsmanagements die Fehler aus dem Gebiet des Netzmanagements. Während über die Modellierung von Komponenten, wie etwa Routern oder Hubs aus Managementsicht heute Übereinstimmung besteht, ist es keineswegs klar, was eine verteilte Anwendung letzlich ausmacht und wie sie für Managementzwecke abstrahiert werden muß. Dies hat zur Folge, daß die entsprechenden Organisationen erneut Managementmodelle entwickeln, die über kein einheitliches Informations- und Kommunikationsmodell verfügen, sondern hauptsächlich auf das Management der eigenen Email-Technologie zugeschnitten sind. Die entsprechenden ISO-Standards sind auf das Management von X.400 Systemen ausgerichtet, die IETF Drafts dienen hauptsächlich dem Management von RFC822-kompatiblen Systemen. Anbieter von Email-Anwendungen gehen häufig eigene Wege und liefern völlig proprietäre Werkzeuge zum Management ihrer Produkte.

Aufgabe der Diplomarbeit ist es nun, die verteilte Anwendung Email selbst zu modellieren und Schnittstellen für deren Management zu definieren. Dabei wird insbesondere Wert auf ein Informationsmodell gelegt, welches unabhängig von bestehenden Managementlösungen und -technologien ist. Im Sinne einer effizienten Implementierung und der Integration von bereits bestehenden Lösungen ist es aber auch erforderlich, das generische Modell problemlos wieder auf vorhandene Architekturen abbilden zu können.

Das *Reference Model of Open Distributed Processing* (RM-ODP) ist ein objektorientierter Ansatz zur Modellierung von verteilten Systemen. Der Standard definiert fünf verschiedene Sichten auf eine Anwendung, anhand derer diese Arbeit die Applikation Email in einem Top-Down Verfahren modelliert. Als Modellierungssprache kommt hierbei die *Object Modeling Technique* (OMT) zum Einsatz.

Um die leichte Abbildbarkeit auf einen bestehenden Managementansatz wie CORBA zu demonstrieren, werden anschließend mit Hilfe eines CASE-Tools IDL-Schnittstellen für eine Managementklasse generiert. Die Implementierung der Schnittstellen erfolgt durch einen in Java programmierten Agenten.

1.3 Vorgehensweise

Nach der kurzen Einführung dieses ersten Kapitels werden im *zweiten Kapitel* die Komponenten einer Email-Infrastruktur, sowie deren Interaktion herausgearbeitet. Ausgehend von der definierten Infrastruktur werden Anforderungen an das Management von Email abgeleitet. Alle später vorgestellten Lösungsansätze werden an diesen Anforderungen gemessen.

Im *dritten Kapitel* wird der momentane Istzustand von Email-Management untersucht. Hierzu werden werden ein kommerzielles und ein nichtkommerzielles Produkt, sowie die Ansätze der

IETF und der ISO betrachtet. Das Kapitel schließt mit der Beschreibung einer Infrastruktur und deren Management im Produktiveinsatz.

Im *vierten Kapitel* wird ein eigenes Managementmodell für eine Email-Infrastruktur, basierend auf dem RM-ODP Modell entwickelt. In einem ersten Schritt (Computational Viewpoint) werden die Komponenten der Infrastruktur und ihre Managementschnittstellen modelliert, in einem zweiten Schritt (Engineering Viewpoint) wird die Abbildung auf ein bestehendes Agentensystem herausgearbeitet. Die Klassen des Modells werden jeweils in OMT spezifiziert.

Im *fünften Kapitel* wird ein Ausschnitt des Modells in einen Agenten umgesetzt. Nach einer kurzen Vorstellung der zugrundeliegenden Techniken wie CORBA und Java wird auf die Implementierungsdetails eingegangen.

Das *sechste Kapitel* schließt die Diplomarbeit mit einem Rückblick ab und beschäftigt sich in einem Ausblick damit, wie das entwickelte Modell in bestehende, nicht objektorientierte Managementarchitekturen integriert werden kann.

2 Anforderungsanalyse für das Email-Management

Ziel der Diplomarbeit ist es, ein objektorientiertes, generisches Modell einer Email-Infrastruktur und ihrer Managementfunktionalität zu gewinnen. Dies wirft eine Reihe von Fragen auf, welche vorab geklärt werden müssen: Was ist eine Email-Infrastruktur? Welche Komponenten lassen sich definieren und welche Funktionalität erbringen Sie? Was versteht man unter dem Management von Email und welche Anforderungen ergeben sich daran? Die folgenden Abschnitte gehen diesen Fragen nach.

2.1 Definition einer Email-Infrastruktur

2.1.1 Aufbau eines Message Handling Systems

Ein Email-System arbeitet nach dem Store-and-Forward Prinzip. Daraus ergeben sich automatisch die beiden Hauptfunktionalitäten des Systems:

- *Nachrichtenübertragung*: die asynchrone, nicht in Echtzeit verlaufende Übertragung von Information zwischen zwei Kommunikationspartnern über DV-Systeme.
- *Nachrichtenspeicherung*: die automatisierte Speicherung von Information, welche durch Nachrichtenübertragung weitergegeben wurde.

Untersucht man nun, aus welchen Komponenten sich eine Email-Infrastruktur zusammensetzt, so findet man, trotz unterschiedlichen Herstellern, Standards und Möglichkeiten verschiedener Systeme weitgehend identische Basiskomponenten.

Definition und Interaktion dieser Komponenten gehen auf Standards der ITU (früher CCITT) und der ISO zurück. Das X.400 Mailsystem der ITU war Grundlage für die Modellierung eines Email-Dienstes durch die ISO: MOTIS. 1988 paßte die ITU das X.400 System schließlich so an, daß es mit MOTIS übereinstimmte.

Um die zentralen Komponenten einer Email-Architektur zu gewinnen, geht die OSI in ihrer X.402-Empfehlung [X402]. nach einem Top-Down Verfahren vor und definiert zuerst den Begriff des sogenannten *Message Handling Environment* (MHE).

Die funktionalen Einheiten dieser Umgebung sind:

- *MHS*: Das Objekt, welches die Informationsübertragung ermöglicht, wird *Message Handling System* (MHS) genannt.
- *Benutzer*: Das Ziel des MHS ist die Informationsübertragung zwischen Nutzern. Eine einzelnes Objekt (also z.B. eine Person), welches die Informationsübertragung in Anspruch nimmt wird *Benutzer* bzw. *User* genannt.

- *Verteiler*: Mit Hilfe des MHS können Benutzer Informationen sowohl an weitere, einzelne Benutzer, als auch an vorher definierte Gruppen von Benutzer übertragen. Das Objekt, welches eine solche Gruppe definiert, wird *Verteiler*, bzw. *Distribution List* genannt.

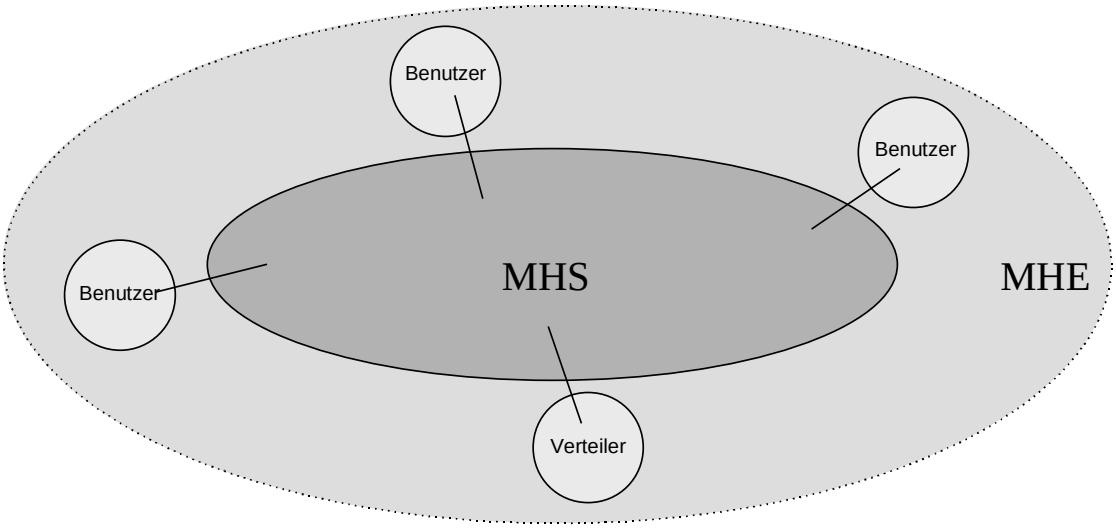


Abbildung 1 - OSI Message Handling Environment

Das Message Handling Environment wird nun schrittweise weiter verfeinert. Von Interesse ist dabei vor allem das Message Handling System:

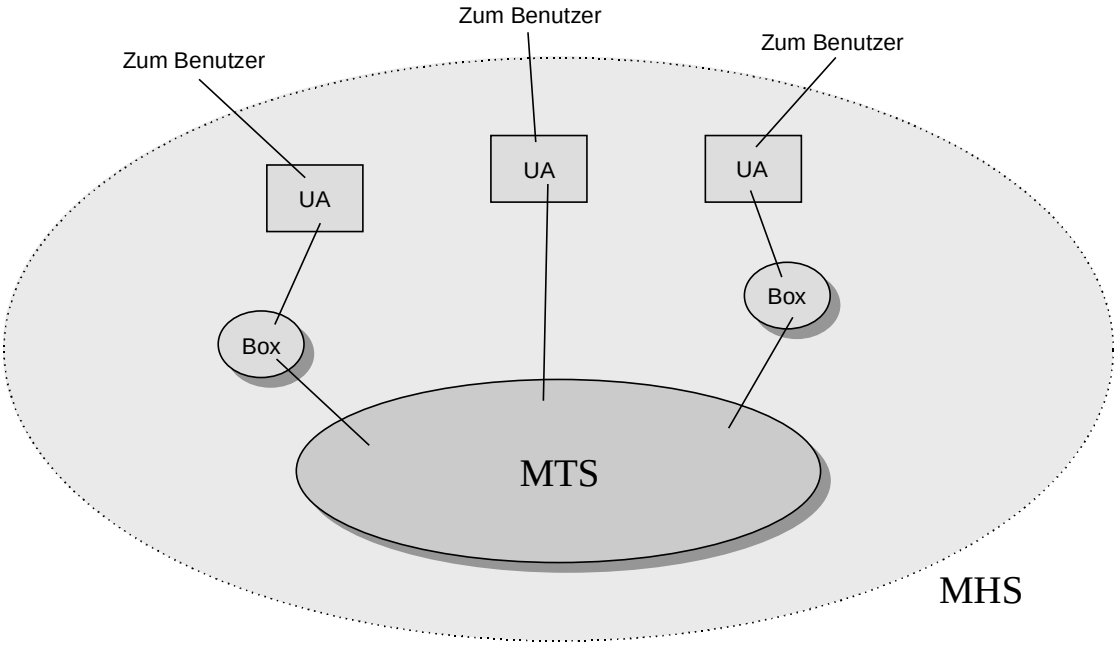


Abbildung 2 - Message Handling System

Die Komponenten des MHS sind:

- *MTS*: Das MHS überträgt Informationen an einzelne Benutzer und Benutzergruppen. Die funktionale Einheit, welche tatsächlich für die Übertragung verantwortlich ist, wird *Message Transfer System* (MTS) genannt.
- *UA*: Die funktionale Einheit, mit welcher der Benutzer in Kontakt tritt, um Nachrichten zu bearbeiten, wird *Benutzeragent* oder *User Agent* (UA) genannt. Bei einem UA kann es sich um ein einfaches Mailprogramm ebenso wie um das Frontend einer Workflow-Applikation handeln.
- *Box*: Ein typischer Benutzer muß die Informationen, die er erhält, abspeichern können. Das funktionale Objekt, welche einem einzelnen Benutzer diese Möglichkeit bietet wird *Mailbox* (*Box*) genannt. Jede Mailbox ist mit einem UA verbunden, aber nicht jeder UA ist notwendigerweise mit einer Mailbox assoziiert.
- *AU*: Die funktionale Einheit, welche das MHS mit anderen Nachrichtenübertragungs-systemen, wie etwa dem Telefaxdienst verbindet, also Gatewaying-Funktionalität bietet, wird *Access Unit* (*AU*) genannt.

Das Message Transfer System setzt sich nun ausschließlich aus Objekten einer Klasse zusammen, den *Message Transfer Agenten* (MTA). Der MTA definiert sich als kleinstes Glied in einer Store-and-Forward Kette, durch die eine Nachricht vom Absender zum Empfänger laufen muß. Damit der MTA Nachrichten ablegen kann, benötigt er einen lokalen Zwischenspeicher. Dieser Zwischenspeicher wird *Nachrichtenspeicher* oder *Message Store* (MS) genannt.

Wie bereits erwähnt, finden sich die dargestellten Komponenten auch in weiteren, unabhängig von der OSI-Modellierung entstandenen Architekturen. Eine Ausnahme bildet hierbei lediglich die Definition des Benutzeragenten, welcher in der OSI Modellierung zu monolithisch ausfällt. Dort hat der UA drei Aufgaben zu erfüllen:

- Er interagiert mit dem Benutzer, dient also dem Lesen, Eingeben und Beantworten von Emails.
- Er kommuniziert mit dem Nachrichtentransfersystem, also der Summe der Transferagenten, über Zustellung and Annahme von Emails.
- Er verwaltet den Nachrichtenspeicher, löscht also alte Nachrichten oder sortiert sie zwischen verschiedenen Ordnern um.

Für die weitere Modellierung wird das Objekt UA deshalb in ein Frontend und ein Backend aufgespalten: Das Bearbeiten von Mails, also die Benutzerschnittstelle und die Interaktion mit dem Nachrichtentransfersystem wird durch ein neues Objekt *Client* realisiert, das Verwalten der Mailbox bleibt hingegen beim Benutzeragenten.

Zusammengefaßt ergibt sich damit folgendes Bild für den Fluß einer Information durch das System:

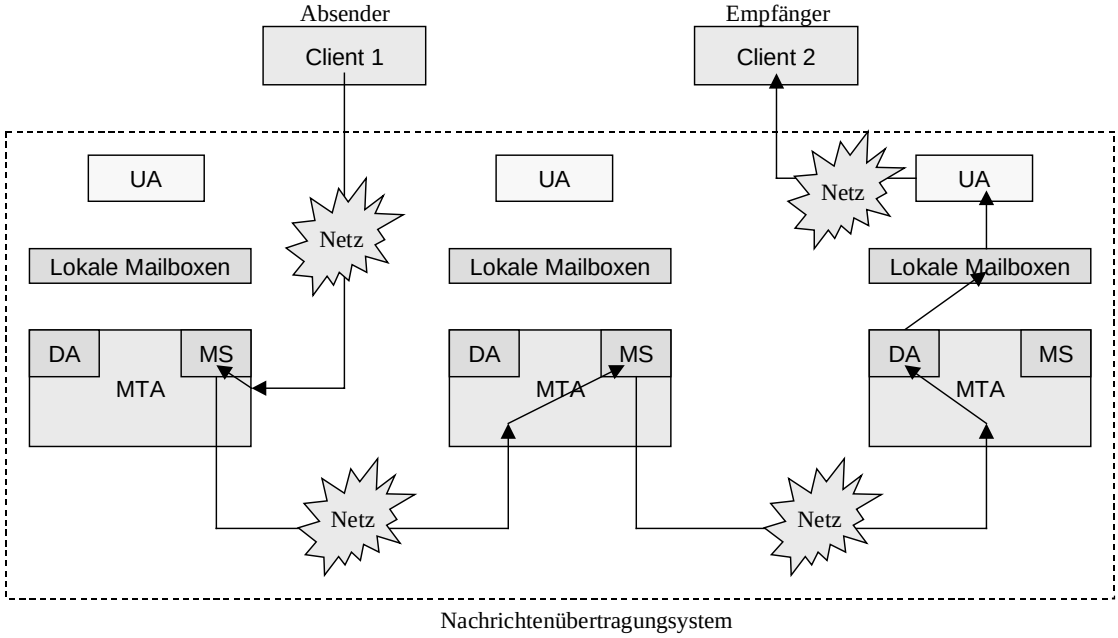


Abbildung 3 - Nachrichtenfluß

Hat der MTA eine Information von einem Benutzeragenten oder einem anderem MTA erhalten, so prüft er die Syntax des Nachrichtenumschlags (siehe 2.1.2). Ist die Syntax, wie etwa die Notation der Empfängeradresse fehlerhaft, so wird entweder die Annahme der Nachricht ganz verweigert oder die Nachricht mit einem Kommentar an den Absender zurückgeschickt. Ist die Syntax des Umschlags in Ordnung, so speichert der MTA die Nachricht die Nachricht in seinem MS ab. In einem nächsten Schritt ermittelt der Transferagent, ob der Empfänger lokal ist. Falls ja, so kann die Information direkt in die Mailbox des Empfängers geschrieben werden. Ist der Empfänger wie in obiger Abbildung dargestellt nicht lokal, so ermittelt der MTA die Route zu diesem durch Abfrage eines Verzeichnisses. Anschließend wird die Mail dem ermittelten MTA zugestellt (*Relaying*), wo der Prozeß erneut beginnt.

Der letzte MTA der obigen Kette erkennt die Mail schließlich als lokal und schreibt sie mit Hilfe eines *Delivery Agents* (DA) in die lokale Mailbox des Empfängers. Von dort holt der Client des Empfängers über den UA die Mail ab.

Alle MTAs halten ihre Aktionen dabei sowohl im Nachrichtenumschlag, als auch in Logdateien fest. Diese Daten können für eine spätere Fehlersuche ausgewertet werden.

Beim oben erwähnten Delivery Agent handelt es sich um ein Hilfsobjekt, welches der MTA zum Schreiben einer Nachricht in eine lokale Mailbox oder zum Abwickeln der Kommunikation mit weiteren MTAs heranziehen kann. Das Konzept des DAs entstammt der Internet Mailarchitektur [RFC822]; in der X.400 Architektur greift der MTA direkt auf die Mailboxen zu.

2.1.2 Aufbau einer Nachricht

Bisher wurde wurden Emails einfach als „Informationen“ bezeichnet, aber noch nicht genauer charakterisiert. Zu diesem Zweck wird nun die von einem Benutzer zum anderen übertragene Information als *Nachricht* oder *Message* definiert. Eine Nachricht setzt sich aus zwei Komponten zusammen:

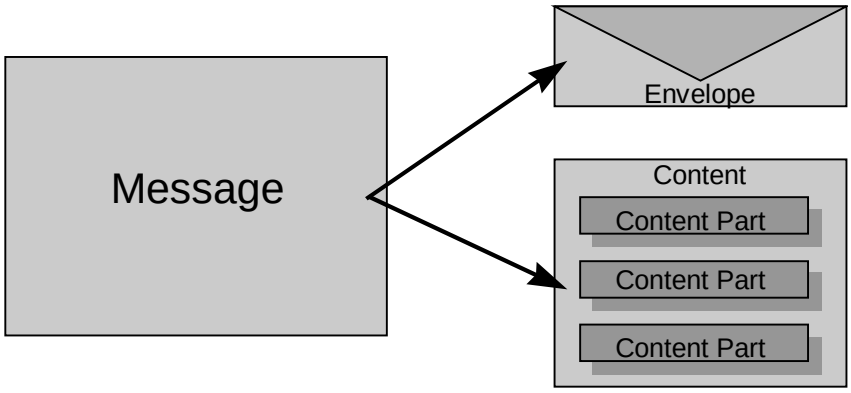


Abbildung 4 - Envelope und Content

- *Envelope*: Das Informationsobjekt, welches Steuerinformationen beinhaltet, also z.B. den Nachrichtenabsender und mögliche Empfänger ausweist, den bisherigen Übertragungsweg angibt, zur Bestimmung des weiteren Nachrichtenweges herangezogen wird und den Nachrichteninhalt oder –typ (*Content-Type*) beschreibt, heißt *Umschlag* (*Envelope*) einer Nachricht. Der Umschlag kann bei jeder Übertragung im MTS neu geschrieben werden.
- *Content*: Das Informationsobjekt, welches Nutzdaten enthält und vom MTA weder untersucht, noch geändert wird, heißt *Nachrichteninhalt* oder *Content*. Der MTA hat allerdings die Möglichkeit, die Darstellung des Informationsobjektes zu ändern; dieser Vorgang wird *Konvertierung* (*Conversion*) genannt.

Zusätzlich zu den allgemeinen Nachrichten, lassen sich noch zwei weitere, spezielle Typen von Nachrichten, die *Benachrichtigungen* definieren: Man versteht darunter vom MTS generierte Nachrichten, welche Auskunft über den Fortschritt bzw. den endgültigen Status einer Nachricht geben: Man unterscheidet dabei zwischen Bestätigungen, den sogenannten *Delivery Reports* und Fehlermeldungen, den *Non-Delivery Reports*.

Sender und Empfänger einer Nachricht werden durch *Adressen* charakterisiert. Adressen sind Datenstrukturen, welche es dem MTS erlauben, den Adressaten innerhalb einer physischen oder logischen Struktur zu lokalisieren. Der Aufbau von Adressen ist nicht genormt, sondern für verschiendene Email-Architekturen unterschiedlich implementiert. So verwendet das X.400 Nachrichtensystem sogenannte O/R Adressen, also eine Liste von Attribut/Werte-Paaren (z.B. G=Christian/S=Coehn/O=Informatik/PRMD=uni-muenchen/C=DE), während die Adressen in [RFC822] Systemen nach dem Format Benutzer bei Rechner bei Subdomäne bei Domäne (coehn@informatik.uni-muenchen.de) aufgebaut sind.

2.2 Managementaspekte eines MHS

Um eine auf obigem Modell aufbauende Email-Infrastruktur effizient verwalten, einfach adminstrieren und problemlos skalieren zu können, ist es erforderlich, das Modell um Managementanspekte zu verfeinern.

In den folgenden Abschnitten wird im Hinblick auf die verschiedenen, funktionalen Managementaspekte untersucht, welche Anforderungen an die Komponenten eines MHS für ein integriertes Management zu stellen sind. In einem weiteren Abschnitt werden weitergehende Managementoperationen vorgestellt, die auf der Basisfunktionalität aufbauen.

In den weiteren Kapiteln werden bereits existierende Lösungen, sowie das zu entwickelnde Modell daran gemessen, inwieweit sie die festgelegten Anforderungen erfüllen können.

2.2.1 Grundfunktionalität

2.2.1.1 Konfiguration

Das Verhalten der oben definierten Komponenten der Email-Infrastruktur wird durch eine Menge von Parametern beschrieben. Unter Konfigurationsmanagement versteht man nun die Abfrage und Modifikation dieser Parameter von einer beliebigen, oftmals zentralen Stelle im Netz. Dabei soll es keine Rolle spielen, ob die Konfigurationsparameter der Komponten lokal, oder zentral in einem Verzeichnis gespeichert sind.

Beispiele für zu setzende Parameter sind:

- Routingstrategien, anhand derer ein MTA das nächste Glied der Store-and-Forward Kette bestimmt. Hierunter ist zum Beispiel die Angabe von Verzeichnisdiensten, die Angabe von Parametern zum Zugriff auf Verzeichnisse, die Angabe von Backupstrategien beim Ausfall von Verzeichnisdiensten oder die Festlegung von statischen Routen und Relays zu verstehen.
- In- und Außerbetriebnahme von Mailhosts.
- Festlegen von Konfigurationsparametern aller Art aus den Bereichen Leistungs- und Sicherheitsmanagement.

Beispiele für abzufragende Parameter sind erneut die Routingstrategien: Hierzu sollten die gemanagten Komponenten, wie z.B. MTAs, Informationen über Ihre Nachbarn im MTS liefern können. Auf diese Weise kann die Mailserver-Topologie ermittelt werden und Routingstrategien graphisch visualisiert werden.

2.2.1.2 Fehler

Ziel des Fehlermanagements ist die zeitige Erkennung von anomalen Zuständen in der Email-Infrastruktur. Der Operateur sollte dabei an der Managementkonsole über den Fehler informiert werden.

Ebenfalls zum weiteren Umfeld des Fehlermanagements gehört die Fehlerkorrelation, also das automatische Erkennen von Folgefehlern. Dadurch sollen Eventbursts – das Melden von

Dutzenden oder Hunderten von Fehlern an der Managementkonsole – vermieden werden. So ist es z.B. wünschenswert, daß das Versagen eines MS mit gleichzeitig auftretenden Alarmmeldungen eines MTA in Verbindung gebracht werden kann.

Ein Problem beim Fehlermanagement auf Dienstebene ist, daß das Versagen von Komponenten durch Probleme auf der System- oder Netzebene ausgelöst werden kann. Daten aus diesen Ebenen liegen dem Dienstmanagement aber meistens nicht vor. Für eine wirksame Überwachung des Dienstes Email ist deshalb eine Integration von Dienst- und Systemmanagement unbedingt erfolderlich (siehe Abschnitt 2.2.3).

Beispiele für Anforderungen an das Fehlermanagement sind:

- Erkennung von Fehlern beim Einspeisen von Nachrichten ins System, bedingt z.B. durch Protokollinkompatibilität, fehlerhafte Adressierung, fehlgeschlagene Authentifizierung oder nicht unterstützte Formate.
- Fehlererkennung beim Zwischenspeichern von Nachrichten, bedingt durch Überlastung oder Ausfall eines MS, Probleme beim Konvertieren des Nachrichtenkörpers; Vorbeugung durch ständige Überwachung von Ressourcen.
- Fehlererkennung beim Relaying, also dem Transport zwischen MTAs, bedingt etwa durch schlechte oder nicht zustandekommende Verbindungen, Ausfall oder Fehlkonfiguration der Gegenstelle, Probleme beim Konvertieren von Adressen und Routingproblemen (fehlerhaft, nicht vorhanden oder Routingschleife). Umgehen der Fehler durch automatisches Umschalten auf Backup-Systeme und Ausweichrouten. Entlastung überforderter Systeme durch Load Balancing der MTAs.
- Fehlererkennung bei der Auslieferung von Mails bzw. dem Abholen von Mails durch Benutzer, ausgelöst durch Probleme mit Mailboxen, ausgefallenen oder fehlerhaft konfigurierten Clients.
- Überwachung der Schnittstellen zu externen Systemen.
- Integration mit Systemmanagement, sowie Eventkorrelierungs- und Troubleticket Systemen.

2.2.1.3 Leistung

Die Qualität einer Email-Infrastruktur wird immer mit mehreren Kriterien bemessen. Während das Fehlermanagement die Fehlerfreiheit und Stabilität des angebotenen Dienstes garantieren soll, soll das Leistungsmanagement dem Anwender die zügige Abarbeitung und Auslieferung seiner Emails garantieren. Dem Betreiber soll das Leistungsmanagement außerdem Auslastungsdaten aller Komponenten der Infrastruktur liefern, um Engpässe zu erkennen, vorhandene Hardware optimal nutzen und Planungsdaten für Erweiterungen zu gewinnen.

Was bedeutet aber in diesem Zusammenhang „zügige“ Abarbeitung und wann ist eine Komponente „optimal“ ausgelastet? Beim Bewerten der Leistung eines Dienstes wird zu diesem Zweck ein Satz von Dienstgüteparametern (*Quality of Service*, QoS), definiert. Beispiele hierfür sind z.B. die maximale Laufzeit einer Nachricht oder die maximale Auslastung eines Systems.

Anschließend muß die Managementlösung den Email-Verkehr im System, sowie die Auslastungsdaten der Komponten überwachen. Wird ein QoS-Kriterium verletzt, so kann die

Managementlösung versuchen, die Dienstgüte durch Ändern von Einstellungen wieder herzustellen. Ist dies nicht möglich, so muß der Adminstrator informiert werden. Hierfür wird ein Alarm- oder Benachrichtigungskonzept benötigt.

Beispiele für Paramter aus dem Leistungsmanagement von Dienstkomponenten, wie MTAs sind:

- Übertragungsdauer für eine Nachricht. Dabei soll zwischen der Gesamtlaufdauer und der Verweildauer auf den einzelnen routenden Systemen unterschieden werden, um Engpässe erkennen zu können.
- Anzahl der offenen Verbindungen im MTS.
- Anzahl der transferierten Nachrichten.
- Volumen der transferierten Nachrichten.
- Anzahl der abgelehnten oder fehlgeschlagenen Transfers.

Für langfristige Planungen und Trendanalysen ist die Archivierung der gesammelten Daten nötig.

2.2.1.4 Abrechnung

Das Abrechnungsmanagement muß für eine Email-Infrastruktur mehrere Aufgaben leisten.

An erster Stelle steht dabei die klassische Accountingfunktion, d.h. der Servicebetreiber will die durch den Benutzer verursachten Kosten erfassen und abrechnen. Nachdem eine abzurechnende Einheit, wie etwa eine Nachricht, oder ein festgelgtes Volumen definiert wurde, sammelt das Managementsystem alle von den Komponenten gelieferten Daten, wie etwa Anzahl der Nachrichten, Größe der Nachrichten, Zeitpunkt des Versands oder Nachrichten-Priorität, ordnet diese Benutzern zu und erstellt Abrechnungen.

Die Protokollierung von Routen und die benutzerbezogene Speicherung des Ressourcenverbrauchs, erlaubt aber noch weitere Anwendungsmöglichkeiten. Die Probleme, welchen Email-Betreiber heute mit Sorge entgegensehen, sind zum einen das Versenden sehr großer Attachments und zum anderen das gehäufte Auftreten von Massenmailings, welche oftmals Werbezwecken dienen. Durch ständige Überwachung der aus dem System einlaufenden Abrechnungsdaten lassen sich plötzlich auftretende Unregelmäßigkeiten erkennen. Dies erlaubt dem Administrator, auch im Rahmen des Sicherheitsmanagements, frühzeitig auf den Verursacher solcher Störungen zuzugehen.

Die oben angesprochene Protokollierung von Routen erlaubt noch eine weitere Auswertung, die vor allem dann interessant ist, wenn in einem Unternehmen bereits viel mit Email kommuniziert wird, oder Arbeitsabläufe auf einen Email-basierten Workflow abgebildet wurden. In diesem Fall kann mit Hilfe der Managementlösung versucht werden, den physikalischen Nachrichtenfluß auf logische Unternehmensstrukturen abzubilden und ein Kommunikationsprofil zwischen z.B. Mitabeitern oder Abteilungen zu erstellen. Die Auswertung eines solchen Profils kann unter Umständen helfen, Unterschiede zwischen vorgesehenen und tatsächlichen Kommunikationswegen aufzudecken und zu minimieren.

Abrechnungsmanagement für eine Email-Infrastruktur soll also die folgenden Möglichkeiten bieten:

- Definition von Abrechnungseinheiten.
- Benutzerbezogene Erfassung der von den Komponenten gelieferten Daten, wie z.B. Anzahl- und Volumen von Nachrichten, Zeitraum (z.B. Hauptzeit/Nebenzeit) des Versands, in Anspruch genommener Ressourcen, wie Plattenplatz und CPU-Zeit, Umrechung in Abrechnungseinheiten und Erstellen der Abrechnung.
- Erkennen von Unregelmäßigkeiten und Peaks bei Nachrichtenanzahl und –Volumen
- Protokollierung von Absender, Empfänger und Route einer Email, sowie darauf aufbauend eine Verkehrsanalyse.

2.2.1.5 Sicherheit

Sicherheitsmanagement hat die Aufgabe, die Datensicherheit und –integrität innerhalb der Email-Infrastruktur sicherzustellen. Dabei müssen drei Bereiche unterschieden werden:

Der erste Bereich umfaßt dabei den Schutz von Integrität und Vertraulichkeit der vermittelten Emails selbst. Wie im Architekturbild im Abschnitt 2.1.1 gezeigt, läuft eine Email bei der Auslieferung über mehrere, im Netz verteilte, Mailhosts. Die Kommunikation zwischen den beiden Mailhosts ist dabei in der Regel ungeschützt. Um nun Verlust oder Verdopplung, der Verfälschung oder dem Abhören von Nachrichten entgegenzuwirken, muß die Email-Infrastruktur Mechanismen, wie Authentifizierung von Komponenten beim Verbindungsaufbau, Verschlüsselung, digitale Signaturen, Zertifikatverwaltung usw. bereitstellen.

Aufgabe der Managementlösung ist es nun, den Einsatz von Sicherheitsmechanismen, durch Definition von Policies oder Visualisierung von Optionen, so einfach wie möglich zu machen. Sicherheitsmanagement ist dabei in diesem Punkt eine Spezialisierung des in Abschnitt 2.2.1.1 besprochenen Konfigurationsmanagements.

Der zweite Bereich, den das Sicherheitsmanagement umfaßt, ist die Sicherung der Infrastruktur selbst. Viele Komponenten der Infrastruktur, wie z.B. MTAs sind einerseits komplexe Programme, in denen immer wieder Fehler entdeckt werden und stehen andererseits direkt mit der Außenwelt in Verbindung. Sie sind damit häufig der erste Punkt für Angreifer, welche die Sicherheit des Systems kompromittieren wollen. So sind z.B. beim MTA *sendmail* immer wieder Sicherheitslücken bekannt geworden, die es Angreifern erlaubten, Superuser-Rechte auf dem Mailhost zu erlangen. Die Aufgabe einer Managementlösung muß es hier sein, solchen Angriffen durch komfortable Konfiguration von Sicherheitsoptionen der Komponenten vorzubeugen und eventuelle Unregelmäßigkeiten im Betrieb, die auf einen Angriffsversuch schließen lassen, zu entdecken und als Alarme weiterzumelden.

Der letzte Bereich läßt sich als Metamanagement bezeichnen: er umfaßt das Sicherheitsmanagement der Managementlösung selbst. Der Zugriff auf die oftmals sensiblen Daten des Managementsystems, oder das Steuern der Infrastruktur über das System, darf nur autorisiertem Personal möglich sein. Es muß also ein Zugriffssystem realisiert werden, welches wie der erste Bereich, Methoden wie Authentifizierung, Verschlüsselung und Benutzerpolicies einsetzt.

Zusammengefaßt sollte eine Managementlösung im Bereich Sicherheit die folgenden Möglichkeiten bieten:

- Zentrale Steuerung von Verschlüsselung, Signaturen, Zertifikaten, etc.

- Umsetzung von Firmenpolicies bezügl. Email-Sicherheit.
- Visualisierung der sicherheitsrelevanten Optionen.
- Konfiguration sicherheitsrelevanter Optionen der Komponenten MTA, UA, MS, wie etwa:
  - Festlegung von Authentifizierungsmechanismen (Paßwortübermittlung im Klartext, verschlüsselte Übermittlung, Einmal-Paßwörter, Challenges, etc.)
  - Abschalten potentiell sicherheitsgefährdener Protokolloptionen (wie etwa der EXPN oder VRFY Optionen bei RFC822 SMTP-MTAs).
  - Festlegen von Zugriffsrechten für MTA und UA selbst.
- Im Zusammenspiel mit Fehler- und Leistungsmanagement die Erkennung von Unregelmäßigkeiten, die eventuell auf Angriffen von außen beruhen.
- Zugriff auf Managementlösung nur durch authorisiertes Personal.

2.2.2 Weitere Funktionalität

Bei der bisherigen Diskussion von Managementanforderungen anhand der Funktionsbereiche wurde fast ausschließlich auf Anforderungen an die einzelnen Komponenten einer Mailarchitektur eingegangen.

Aufbauend darauf lassen sich jedoch weitergehende Szenarien ableiten, in deren Mittelpunkt der Dienst Email steht. Derartige Szenarien lassen sich auch nicht mehr den streng getrennten Managementfunktionsbereichen zuordnen, vielmehr werden mehrere Bereiche berührt.

Ein solches Szenario, welches auch bei der Erstellung des Managementmodells im Computational und Engineering Viewpoint des ODP-Modells (vgl. und 4.3.5) wieder aufgegriffen wird, ist das *Message Tracking*.

Unter **Message Tracking** versteht man eine Erweiterung des Message Handling Systems: Wurden bisher nur Statusnachrichten vom Typ Delivery oder Non-Delivery erzeugt, soll das System nun eine Nachricht von der Quelle bis zur Senke verfolgen und jederzeit Auskunft über den Ort und Status von Emails geben können. Auch nach erfolgter Zustellung einer Mail sollen noch Anfragen möglich sein, etwa „über welche MTAs ist meine Mail gelaufen“ oder „warum hat die Zustellung so lange gedauert“.

Durch Message Tracking werden werden Anwender und Betreuer des Systems auf folgende Weise unterstützt:

- Rückmeldungen und Empfangsbestätigungen für Benutzer. Haben die Benutzer Zugriff auf das Message Tracking System, um ihre eigenen Nachrichten verfolgen zu können, so entlastet dies die Hotline und steigert das Vertrauen in die Email-Infrastruktur.
- Überprüfung der Email-Infrastruktur und einfache Fehlerisolierung, insbesondere nach Umstellungen bei Hardware oder Routing. Dabei kann sowohl der Weg von „verschwundenen“, als auch von falsch ausgelieferten Emails nachvollzogen werden.
- Ermittlung von nachrichten- und benutzerspezifischen Daten, die, wie bereits erwähnt, als Grundstein für Performancemessungen und Accounting dienen können.

- Unterstützung des Sicherheitsmanagements – potentielle Sicherheitslücken können präzise lokalisiert werden. Wird das System z.B. mit Nachrichten überflutet, läßt sich der Nachrichtenursprung erkennen.

Um ein effizientes Message Tracking zu realisieren, sollte das Managementsystem die folgenden Möglichkeiten bereit halten:

- Ein Benutzer muß eine Anfrage mit mehreren Kriterien stellen können. Beispiele hierfür sind z.B. die Message-ID, Absender, Empfänger, Betreff oder ein Zeitraum. Dabei muß es dem Benutzer einerseits möglich sein, mit Hilfe von „Wildcards“ eine möglichst umfassende Suchanfrage zu stellen, andererseits muß er das Suchergebnis durch Kombination der Kriterien auch einschränken können.
- Je nach Berechtigung (Benutzer/Administrator) darf eine Anfrage nur die eigenen, oder alle Nachrichten umfassen.
- Das System muß skalierbar sein und auch bei hohem Nachrichtenaufkommen schnelle Antwortzeiten garantieren. Dies bedingt eine genaue Planung des Tracking Systems, also wie die Tracking-Informationen gespeichert werden (Datenstruktur, Indizierung), und wo die Informationen vorgehalten werden – bei den einzelnen MTAs (Pull-Konzept) oder einer zentralen Managementstation (Push-Konzept). Für alle möglichen Alternativen sind die unterschiedlichen Auswirkungen auf die Antwortzeiten und den Ressourcenverbrauch, wie Netzlast oder Plattenplatz des Tracking-Systems zu ermitteln.
- Das System soll optional nicht nur auf Anfrage von Benutzern und Administratoren aktiv werden, sondern das „Hängenbleiben“ einer Email erkennen und dem Administrator anzeigen (Frühwarnfunktion).
- Der Einfluß des Tracking-Systems auf die Glieder der Store-and-Forward Kette, die MTAs, muß minimal bleiben. CPU oder I/O-intensive Operationen, welche die Hauptaufgabe, den Nachrichtentransport selbst, in seiner Performanz negativ beeinflussen, sind nicht wünschenswert.
- Das Message Tracking System darf nicht auf Komponenten eines bestimmten Herstellers zugeschnitten sein, sondern muß mit einer heterogenen Email-Welt, also auch Gatewaying, zurechtkommen.

Ein weiteres Beispiel für eine dienstumfassende Managementaufgabe ist die **Benutzerverwaltung**. Benutzer, welche Nachrichten über das Mailsystem empfangen möchten, müssen dem System bekannt gemacht werden, also angelegt werden. Dazu ist in der Regel die Vergabe einer eindeutigen Adresse ebenso erforderlich, wie die Bereitstellung einer Mailbox, eine Zugangsberechtigung zum UA und die eventuelle Eintragung in Verteiler.

Auch an die Benutzerverwaltung werden mehrere Anforderungen gestellt:

- Die Benutzerdaten müssen, auch wenn sie im System verteilt gespeichert werden, zentral zugänglich sein. Änderungen sollen ausschließlich an einem definierten Punkt geschehen und automatisch zu den relevanten Komponenten propagiert werden.
- Die Benutzerverwaltung soll sich direkt auf ggf. bereits bestehende Benutzerdaten (z.B. in Verzeichnisdiensten) abstützen oder diese zumindest automatisiert einlesen können. Dabei ist vor allem die Übernahme von Authentifizierungsdaten gefragt (*Single Signon*).

- Die Benutzerverwaltung soll das Setzen von Optionen, wie etwa Limits bezüglich Mailanzahl oder –größe getrennt für jeden einzelnen Benutzer, aber auch für komplette, flexibel zu definierende Benutzergruppen erlauben

Eine weitere, bereichs- und sogar anwendungsübergreifende Aufgabe ist das **Routingmanagement**. Die Konfiguration des Mailroutings erfolgt im Idealfall an einer graphischen Benutzeroberfläche:

- Ein Topologiedient ermittelt durch Abfragen der MTA-Konfiguration und des Verzeichnisdienste die logische Struktur des Systems und visualisiert sie an einer Managementstation.
- Mailrouten könnten bei einer solchen Anwendung durch Graphen dargestellt werden und intuitiv geändert werden.
- Stößt ein MTA an seine Leistungsgrenzen oder fällt er komplett aus, so soll das System auf Backuprouten umstellen und dies dem Operateur wiederum melden.

Es gibt eine Vielzahl weiterer Managementanforderungen. Denkbar ist z.B. der Einsatz von *Policies* [Heil98, Slo94], also Vorgaben, welche Benutzern und Administratoren des Systems auferlegt werden und welche automatisch überwacht werden sollen. Die zwangsweise Verwendung von *Corporate Message Recovery Keys* (CMRK) zur Verschlüsselung stellt z.B. eine solche Policy dar. Ein weiteres Einsatzgebiet von Policies wäre die Definition des Verhaltens des Email-Systems gegenüber nomadischen oder unbekannten Teilnehmern (vgl. [Radi98]).

Allen Szenarios gemeinsam ist die Definition von Dienstparametern, welche irgendwann auf Komponentenparameter abgebildet werden müssen. Diese Abbildung findet wiederum auf dem bereits erwähnten Computational und Engineering Viewpoint des ODP Managementmodells statt. Dabei werden im Computational Viewpoint alle Komponenten mit einer Managementgrundfunktionalität ausgestattet, welche die Erfüllung der gestellten Anforderungen garantieren soll. Im Engineering Viewpoint wird detailliert auf die Verteilung und Zusammenspiel von Managementschnittstellen und auf die Realisierung der Konzepte des Computational Viewpoints eingegangen.

2.2.3 Integration mit dem Systemmanagement

Da verteilte Anwendungen Netz- und Systemressourcen in Anspruch nehmen, reicht es für eine Managementanwendung nicht aus, die Email-Infrastruktur isoliert zu betrachten. Benötigt werden Schnittstellen zum klassischen Systemmanagement, um auftretende Fehlerzustände mit Fehlern der Ausführungsumgebung korellieren zu können.

Einbezogen werden muß also ein Prozeßmanagement, welches kontinuierlich prüft, ob unterstützende Prozesse noch laufen. Ebenfalls überwacht werden müssen Ressourcen wie Plattenspeicher, CPU-Last und Netzlast, Belegung von Ports, Qualität von Verbindungen und vieles mehr.

Wenn das Management der verteilten Anwendung Email nur bei bereits bestehendem Systemmanagement möglich und sinnvoll ist, stellt sich auch die berechtigte Frage, wie weit man den

Anforderungen des Mailmanagements durch Ausdehnung einer bereits bestehenden Systemmanagementlösung gerecht werden kann.

In der Praxis (siehe Abschnitt 3.4) zeigt sich jedoch, daß der alleinige Einsatz von Methoden aus dem Systemmanagement keineswegs das fehlerfreie Funktionieren einer verteilten Anwendung garantieren kann. Klassisches Management kann nur feststellen, ob entsprechende Prozesse laufen, ob genügend Ressourcn verfügbar, ob Netzwerkverbindungen bestehen und ob Daten ausgetauscht werden. Ohne weitere Kenntnis über die Semantik der Daten können Werkzeuge zum System- und Netzmanagement weder garantieren, daß das System tatsächlich läuft, noch können sie Fehler erkennen, welche sich in der Anwendung ereignen – gerade diese Fehler gilt es jedoch zu beseitigen bzw. zu vermeiden.

3 State of the Art im Emailmanagement

Aufsetzend auf der Anforderungsanalyse werden im folgenden Kapitel bereits bestehende Managementansätze untersucht. Die Abschnitte 3.1.1 und 3.1.2 betrachten dabei die proprietären Löaungen einer freien und einer kommerziellen Email-Lösung. In einem zweiten Schritt sollen die Managementmodelle bekannter Gremien beleuchtet werden. Von der IETF, der ISO, der DMTF und der OMG bieten nur die ersten beiden Lösungen zum Email-Management, welche in den Abschnitten 3.2 und 3.3 vorgestellt werden. Abschließend wird ein konkretes Mail-Management-Szenario in Abschnitt 3.4 kurz vorgestellt.

3.1 Herstellerspezifische Ansätze

3.1.1 Sendmail

Im Internet ist *sendmail* der Standard-MTA. Eric Allman begann die Entwicklung des Programms als Student an der Berkeley Universität in Kalifornien bereits 1979, um Emails zwischen dem ARPAnet und UUCP-basierenden Netzen austauschen zu können. Als im ARPAnet 1980 die schrittweise Umstellung von NCP auf TCP vollzogen wurde, hatte dies eine explosionsartige Vermehrung von Hosts und die Umstellung vom flachen (xxx-university) zum heutigen, hierarchischen (xxx.university.edu) Adreßraum zur Folge.

Als Protokoll zur Übermittlung von Nachrichten war FTP nun nicht mehr geeignet: 1982 wurde in [RFC821] das *Simple Mail Transfer Protocol* (SMTP) veröffentlicht und *sendmail* wurde fester Bestandteil von BSD Unix.

Die Version von *sendmail*, die heute von jedem Unix-Hersteller ausgeliefert wird und bei den meisten Internet Servivce Providern als MTA eingesetzt wird, ist 8.8.8. Seit kurzem ist die neue Version 8.9 von *sendmail* verfügbar. Aus Managementgesichtspunkten unterscheiden sich beide Versionen nicht, die neuen Funktionen von Version 8.9 wurden jedoch in der prototypischen Agentenimplementierung (siehe Kapitel 5) berücksichtigt.

Zum Zeitpunkt der Entstehung von *sendmail* war der Bedarf eines integrierten Managements nicht vorhanden. Die einzige Möglichkeit an Daten für Fehler- oder Leistungsanalyse oder Abrechung zu gelangen, bestand in der Auswertung von Logdateien.

Heute ist das ständig zunehmende Nachrichtenvolumen ebenso ein Problem, wie die Unzulänglichkeiten des SMTP-Protokolls, welchem die fehlende Unterstützung für z.B. internationale Zeichensätze, Binäranhänge, Multimedia, Authentifizierung und Verschlüsselung mit immer neuen Zusätzen, wie z.B. MIME (vgl. [RFC1521, RFC1522]), PGP/MIME (vgl. [RFC2015]), usw., nachgerüstet werden soll. Obwohl der Managementbedarf für die Anwendung Email heute sogar von der IETF erkannt wird, bietet das *sendmail* Programm auch in seiner neuesten Version keinerlei Werkzeuge[2] zur Überwachung des MTAs oder zur Einbindung in eine bestehende Managementplattform.

Zur Überwachung seiner Funktionen bietet *sendmail* die folgenden Möglichkeiten:

[2] Mehrere Anfragen in Newsgroups nach Umsetzungen der IETF MIBs oder nach proprietären Managementwerkzeugen für sendmail blieben, ebenso wie eine Suche im WWW, erfolglos.

- Logging mittels *syslog*: Sämtliche Meldungen des MTAs werden an den Syslog-Dämon übermittelt. Der Syslog-Dämon schreibt diese Nachrichten in Logdateien oder übermittelt sie an eine zentrale Stelle. Die Menge der protokollierten Nachrichten läßt sich dabei in verschiedenen Schritten einstellen. Greift man die übermittelten Informationen an der Syslog-Schnittstelle ab (vgl. [Coeh97]), so lassen sich zumindest Fehlermeldungen in SNMP Notifications umsetzen und an eine zentrale Stelle übermitteln. Ein weitergehendes Management ist nicht möglich.
- Shell- und Perlskripte: Teil der *sendmail* Distribution ist das Programm *mailstats*, sowie einige Skripte, die einfach Statistiken, wie etwa Anzahl und Volumenen der ein- und ausgehenden Emails erzeugen.

Ebenso einfach wie die Log- und Reportingmöglichkeiten von *sendmail* sind die Konfigurationsmöglichkeiten: Die Grundeinstellungen des Programms werden über eine einzige Datei konfiguriert. Weiterhin stützt sich *sendmail* für Routinginformationen auf den Verzeichnisdienst DNS (vgl. [RFC1035]) ab. Durch Änderung der Routinginformationen bzw. entsprechende Backup-Routen können Fehler durch Überbrücken von MTAs kompensiert werden. Zentrales Fehler- oder Konfigurationsmanagement, wie Überprüfen der Queues, Setzen von Schaltern, Rerouten oder Neuerzeugen von Messages, ist jedoch nicht möglich.

Betrachtet man zusammenfassend die Möglichkeiten von *sendmail* im Bezug auf Email-Management, so schneidet das Programm sehr schlecht ab. Daß es trotzdem soweit verbreitet ist, dürfte auch an der historisch gewachsenen Installationsbasis liegen – zieht man jedoch in Betracht, daß vereinfachte und zentrale Administration auch in den neuesten *sendmail* Versionen keine Rolle spielt, so scheint der Druck durch die Anwender, also die Email-Betreiber, in diesem Bereich eine Verbesserung zu erzielen, nicht allzu groß zu sein. Eventuell ist man sich hier noch nicht der Arbeitserleichterung und Kostenersparnis durch Email-Management bewußt.

3.1.2 Netscape Messaging Server

Die Firma Netscape wurde 1994 von Jim Clark und Mark Andreessen gegründet. War der Netscape Internet Browser anfangs das einzige Produkt, so ist die Firma seit ihrem Börsengang 1995 mit einer Vielzahl von Produkten auf dem Markt vertreten.

Anders als Microsoft oder Lotus, welche mit ihren Produkten Exchange und Notes monolithische Messaging- und Groupwarelösungen anbieten, bietet Netscape bereits eine modulare Lösung von verschiedenen Severprodukten an, welche alle auf offenen Internetstandards basieren. Da die Netscape Software zudem weit verbreitet ist, wurde sie exemplarisch als Vertreter eines kommerziellen Produkts untersucht

Netscapes Produktpalette umfaßt Directory-Server, Zertifikats-Server, Email-Server, sowie Server für Calendering- und Workflowfunktionen.

Zur zentralen Administration sämtlicher Internet Server und Clients setzt Netscape auf seine Lösung Mission Control [Nets98], welche im Laufe des zweiten oder dritten Quartals 1998 verfügar sein soll.

Die Architektur von Mission Control stellt Netscape folgendermaßen dar:

Abbildung 5 – Netscape Mission Control Architektur (aus [Nets98])

Dreh- und Angelpunkt dieser Architektur ist ein zentraler Verzeichnisdienst, in dem nicht nur Benutzer, Gruppen, Listen sowie Zugriffsrechte vorgehalten werden, sondern auch die Daten aller im Netz arbeitenden Server. Das zentrale Verzeichnis, auf das mittels LDAP zugegriffen wird, ersetzt somit lokale Konfigurationsdateien bzw. Registrierungseinträge.

Weitere Komponenten dieser Architektur sind lokale, also auf dem Serverhost laufende „HTTP-Control-Agents“. Diese Agenten übernehmen Aufgaben, wie Starten und Stoppen eines Service, Prozeß- und Ressourcenkontrolle, sowie weitere serverspezifische Konfiguration. Wie der Name andeutet, setzt Netscape zwischen den Control Agent und der Mission-Control-Konsole das Hypertext Transfer Protocol (HTTP) ein.

Die in Java programmierte Mission-Control-Konsole dient als Benutzerschnittstelle für den Operateur. Sie kann von jedem Server im Netz gestartet werden und läuft auch im Webbrowser.

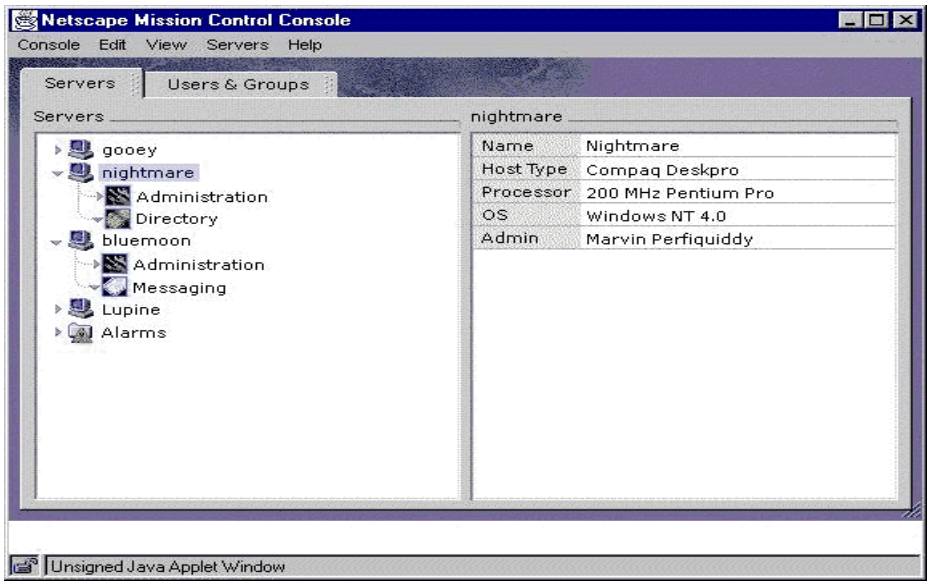


Abbildung 6 – Netscape Mission Control Desktop (aus [Nets98])

Die Konsole zeigt alle im Directory Service registrierten Anwendungen an und erlaubt die Steuerung derselben über die Control Agents. Für nicht interaktive oder regelmäßig wiederkehrende Managementaufgaben werden Kommandozeilentools und Skripten zur Verfügung gestellt.

Zur Einbindung von nicht-Netscape Anwendungen in das Mission-Control-Framework wird Netscape ein Software Developers Kit zur Verfügung stellen, welches die Entwicklung eigener Agenten erlauben soll.

Unabhängig von Mission Control ist der Email-Server von Netscape auch mit einem SNMP-Agenten ausgestattet. Die Netscape Email MIB orientiert sich dabei an der in Abschnitt 3.2.2 vorgestellten *Mail Monitoring MIB*.

Soll der Netscape Email-Agent, mit dem auf einem Server vorinstallierten Hauptagent koexistieren, so wird ein SNMP-Proxy installiert, welcher alle SNMP Anfragen für den Netscape-Teilbaum an den Email-Agenten abzweigt und die restlichen Anfragen an den Hauptagenten durchreicht:

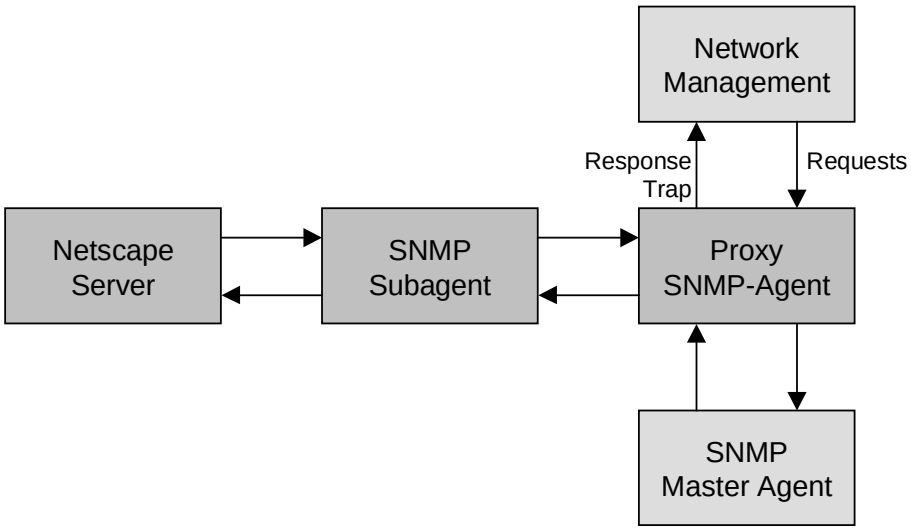


Abbildung 7 – Netscape Proxy-Agent

Wie ist nun dieser Managementansatz zu bewerten? Als erstes gilt festzuhalten, daß Netscape mit Mission Control und SNMP zwei unterschiedliche Ansätze zur Verwaltung eines Email-Servers zur Verfügung stellt. Mission Control leistet dabei für Email kaum mehr, als einfaches Prozeß- und Ressourcenmonitoring. Ob die Control-Agents für Email in ihrer Leistung erweitert werden, kann im jetzigen (Ankündigungs-)stadium von Mission Control noch nicht gesagt werden.

Die SNMP Komponente in Netscapes Managementlösung erlaubt eine Darstellung von Auslastung und Performanz, kann aber die in Kapitel 2 gestellen Anforderungen nicht erfüllen.

Die Interoperabilität zwischen Netscapes SNMP-Agenten und Mission Control ist leider nicht befriedigend. So gibt es keine Gateways oder Proxies, die es einer SNMP-Management-Station möglich machen, mit Netscapes HTTP-Agenten zu kommunizieren. Umgekehrt kann

die Mission Control Console nur eingeschränkt auf die Daten der SNMP-Agenten aus eigenem Haus zugreifen (ausschließlich Prozeßmonitoring).

So positioniert Netscape sein Mission Control also als low-end Tool für die täglichen Administrationsaufgaben („Mission Control provides centralized user, security, and configurationmanagement“) komplementär zu bestehenden Managementlösungen („Enterprise management solutions such as HP OpenView, Tivoli TME, [...] provide complimentary high-end management functionality [...]“)[3].

3.1.3 Weitere Produkte

Im Rahmen der Produktuntersuchung wurden noch weitere Produkte, wie z.B. *CDC Intratore* [CDS98] oder der *Lotus Notes Domino Sever* [Lotu98] evaluiert.

Beide Produkte verfügen sowohl über proprietäre Managementschnittstellen, als auch über SNMP-Implementierungen. Da die Produkte im Bereich Management keine anderen Ansätze als der Netscape Messaging Server bieten, werden sie nicht detaillierter vorgestellt.

Erwähnenswert ist lediglich, daß Lotus die Unterstützung des *Java Management API* (JMAPI) [Sun97, Schi98] angekündigt hat. Weitere Details wurden jedoch noch nicht bekanntgegeben.

3.2 Normierte Ansätze der IETF

Innerhalb der *Internet Engineering Task Force* (IETF) arbeiten mehrere Arbeitsgruppen an dem Management von verteilten Systemen. Die *Mail and Directory Management* (MADMAN) Gruppe beschäftigt sich dabei insbesondere mit Email- und Verzeichnisdiensten. Alle folgenden Ansätze haben gemeinsam, daß sie dem Trend, weg vom Management einzelner Komponenten, hin zum Anwendungsmanagement, folgen.

Obwohl alle hier vorgestellten Lösungen im Rahmen, des von der IETF erarbeiteten und von Marshall T. Rose in [Rose96] ausführlich dokumentierten *Internet Management Frameworks* realisiert werden, nehmen Sie doch für sich in Anspruch, generisch genug zu sein, um nicht nur Internet-Applikationen, sondern auch die entsprechenden ISO-Dienste managen zu können.

3.2.1 Network Services Monitoring MIB

Im Januar 1998 wurde die ursprünglich in [RFC1565] definierte *Network Services Monitoring MIB* (NSM-MIB), nach mehrjähriger Beratung, in [RFC2248] neu aufgelegt. Da die im folgenden Abschnitt vorgstellte Mail Monitoring MIB auf der NSM MIB aufbaut, wird diese hier kurz vorgestellt.

Bei den, in der NSM MIB modellierten, zu überwachenden Objekten, handelt es sich nicht um Rechner, oder Komponenten, sondern um Netzwerkapplikationen im weitesten Sinne. Die MIB enthält also Elemente, die allen gemanagten Anwendungen gemein sind. Dabei verstehen die Autoren Freed & Kille die Objekte ihrer MIB nicht unbedingt als universell

[3] http://www.netscape.com/comprod/at_work/solutions/admin/faq.html

einsetzbar. Sie sollen vielmehr als „Basisklassen“ für eigene, zu überwachende Anwendungen möglichst generisch und einfach zu erweitern sein.

Wie sich am Beispiel der „abgeleiteten Klasse“ *Mail Monitoring MIB* jedoch zeigt, wird dieser Ansatz durch das nicht objektorientierte Informationsmodell von SNMP erschwert.

Damit modelliert die MIB eine Netzwerkanwendung auf folgende Weise:

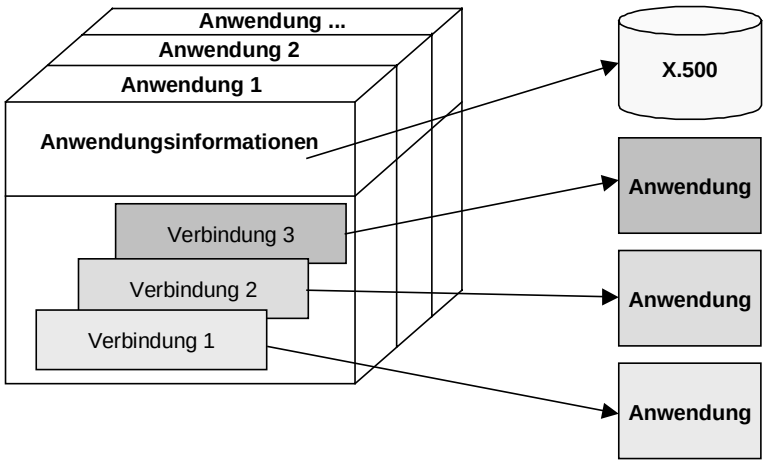


Abbildung 8 – Network Services Monitoring MIB

Die MIB besteht aus einer Tabelle, in der jeder Anwendung 1-x auf dem Agentenhost eine Zeile mit Informationen zugeordnet wird. Für jede Applikation wird außerdem eine weitere Tabelle gepflegt, aus der die momentan offenen Netzverbindungen ersichtlich sind. Damit ergibt sich das zweidimensionale Schema aus Abbildung 8.

Folgende Informationen werden über eine Anwendung geführt:

- **applIndex**: Eindeutiger Index für eine Anwendung
- **applName**: Anwendungsname im Klartext
- **applDirectoryName**: Directory Eintrag (z.B. X.500), wo weitere Informationen zur Anwendung gefunden werden können
- **applVersion**: Version der Anwendung
- **applUptime**: Startzeitpunkt der Anwendung
- **applOperStatus**: gibt den gegenwärtigen Zustand der Applikation, wie z.B. up, down, halted usw. wieder
- **applLastChange**: Zeitpunkt, zu dem der aktuelle Zustand begann
- **applInboundAssociations**: Anzahl der derzeit offenen eingehenden Verbindungen
- **applOutboundAssociations**: Anzahl der derzeit offenen ausgehenden Verbindungen
- **appAccumulatedlInboundAssociations**: Gesamtzahl aller eingehenden Verbindungen

- **applAccumulatedOutboundAssociations**: Gesamtzahl aller ausgehenden Verbindungen
- **applLastInboundActivity**: Zeitpunkt der letzten eingehenden Verbindung
- **applLastOutboundActivity**: Zeitpunkt der letzten ausgehenden Verbindung
- **applRejectedInboundAssociations**: Anzahl der abgelehnten, eingehenden Verbindungen
- **applFailedOutboundAssociations**: Anzahl der abgelehnten, ausgehenden Verbindungen
- **applDescription**: Beschreibung der Applikation
- **applURL**: URL mit weitergehenden Applikationsinformationen

Folgende Informationen werden über die offenen Verbindungen geführt:

- **assocIndex**: Eindeutiger Index der Verbindung
- **assocRemoteApplication**: Name des Systems, auf dem die entfernte Anwendung läuft (z.B. IP-Adresse oder eindeutige Bezeichnung des gemanagten Objektes bei OSI)
- **assocApplicationProtocol**: Verbindungsprotokoll zur entfernten Anwendung
- **assocApplicationType**: Typ der Verbindung (ein- oder ausgehend, mit Client oder Server)
- **assocDuration**: Startzeitpunkt der Verbindung

Bei Betrachtung der geführten Variablen fällt auf, daß tatsächlich nur ein Grundgerüst modelliert wird: Eine Netzwerkanwendung ist ein Programm, welches Verbindungen zu anderen Programmen unterhält. Über die Anzahl der Verbindungen wird eine Statistik geführt.

Alle Variablen der MIB sind als *read-only* deklariert, d.h. selbst einfache steuernde Zugriffe, wie Start oder Stop einer Applikation, sind nicht möglich.

Bei genauerer Betrachtung der MIB fällt außerdem auf, daß bei den Verbindungen einer Anwendung, lediglich Name des Partnersystems und verwendetes Protokoll geführt werden. Insbesondere erfaßt die MIB nicht auf eindeutige Weise, mit welcher Anwendung auf dem entfernten Host über das Netz kommuniziert wird, d.h. es gibt bei den Associations keinen Verweis auf den Applikationsindex (applIndex) der Anwendung auf der entfernten Maschine.

Dies hat zur Folge, daß die MIB lediglich eine Ansammlung von Anwendungen und deren Verbindungsstatistiken bietet, die jedoch in keinem Verhältnis zu Anwendungen auf anderen Hosts stehen. Da die Managementanwendung somit keine Abhängigkeiten zwischen den Applikationen ermitteln kann, ist im Fehlerfall leider auch keine Korrelation möglich.

3.2.2 Mail Monitoring MIB

Wie in Abschnitt 3.2.1 erwähnt, handelt es sich bei der *Network Services Monitoring* MIB nur um ein generisches Template für beliebige Netzwerkapplikationen. Zur Überwachung von MTAs haben die beiden Autoren deshalb der NSM MIB eine weitere, spezialisierte MIB hinzugefügt. Diese *Mail Monitoring MIB* wurde erstmals 1994 in [RFC1566] vorgestellt und jetzt in [RFC2249] überarbeitet.

Der Fluß von Emails wird von den Autoren des RFC wie folgt dargestellt:

- Ein MTA empfängt Nachrichten durch Useragenten, Message Stores, andere MTAs oder Gateways
- Der MTA ermittelt den „next hop“, also das nächste Ziel einer Nachricht. Da eine Mail mehrere Empfänger haben kann, kann sie auch mehrere „next hops“ haben, was u.U. eine oder mehrere Kopien nötig macht.
- Falls nötig, wird das Nachrichtenformat konvertiert
- Die Nachricht wird ans Ziel, also einen weiteren MTA, einen UA, einen MS oder ein Gateway übermittelt.

Kernstück ist hier also der MTA, der in der Mail Monitoring MIB wie folgt modelliert wird:

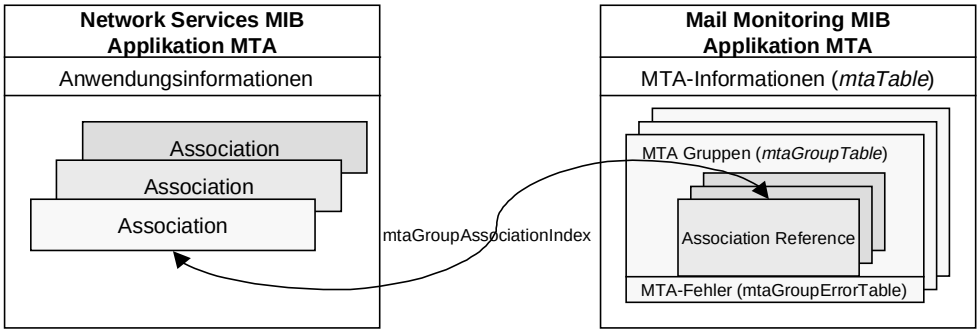


Abbildung 9 – Mail Monitoring MIB

Die Abbildung zeigt, daß sich die *Mail Monitoring MIB* in vier Tabellen gliedert, nämlich in *mtaTable*, *mtaGroupTable*, *mtaGroupAssociationTable* und *mtaGroupErrorTable*, von denen allerdings nur die erste zwingend implementiert werden muß:

- **mtaTable**: Diese Tabelle, die jeder Agent, der die MIB implementiert, bereitstellen muß, umfaßt 12 Variablen für jeden, auf dem Agentenhost laufenden MTA. Dabei handelt es sich um reine Statistik: mtaReceivedMessages, mtaStoredMessages, mtaTransmitted-Messages, mtaReceivedVolume, mtaStoredVolume, mtaTransmittedVolume, mta-ReceivedRecipients, mtaStoredRecipients, mtaTransmittedRecipients, mtaSuccessful-ConvertedMessages, mtaFailedConvertedMessages, mtaFailedConvertedMessages, mta-LoopsDetected.

  Die Bedeutung der Variablen erschließt sich dabei sofort aus deren Namen, auf eine detaillierte Erläuterung wird an dieser Stelle verzichtet.

- **mtaGroupTable**: Diese optionale Tabelle erlaubt das Zusammenstellen von Variablen-gruppen. So kann z.B. die erste Zeile dieser Tabelle fünf statistische Variablen zum Nachrichteneingang, die nächste Zeile 10 Variablen zum ausgehenden Volumen usw. enthalten. Dem Agent steht es dabei völlig frei, welche Kombinationen er aus den, insgesamt 33 zur Verfügung stehenden, Variablen realisiert.

- **mtaGroupAssociationTable**: Diese optionale Tabelle realisiert die Verknüpfung der Mail Monitoring Mib mit der allgemeineren Network Services Monitoring MIB. In der mtaGroupTable stehen u.a. Variablen wie mtaGroupOutboundAssociations zur Verfügung, d.h. es wird z.B. festgehalten, wieviele ausgehende Verbindungen der MTA offen hat. Wie in Abbildung 9 gezeigt, werden diese offenen Verbindungen über die Variable *mtaGroupAssociationIndex* mit den offenen Verbindungen der Anwendung MTA in der Network Services MIB korelliert.
- **mtaGroupErrorTable**: In dieser optionale Tabelle werden die Fehler oben für jede definierte Gruppe mitprotokolliert.

Die Verbindung der Mail Monitoring MIB mit der Network Services Monitoring MIB gleicht deren Schwächen aus: Sofern auf jedem MTA-Host ein Agent läuft, der beide MIBs implementiert, kann sich eine Managementapplikation von MTA-Host zu MTA-Host hangeln und die Abhängigkeiten zwischen den MTAs herausfinden. Beim Aufall eines MTAs kann die Managementapplikation alle mit diesem MTA verbundenen Systeme anzeigen und die Fehler korellieren.

Leider ist der gruppenspezifische Teil der MIB unterspezifiziert. Dadurch, daß es jedem Agent gestattet ist, eigene, völlig willkürlich zusammengesetzte Gruppen zu bilden, muß eine Managementapplikation nicht nur die MIB, sondern auch die herstellerabhängige Implementierung im Agenten kennen – die Generik geht deshalb völlig verloren. Es muß deshalb bezweifelt werden, ob Implementierungen von [RFC2249] mehr als nur die erste Tabelle umfassen werden.

Bei der Betrachtung der MIB fällt außerdem auf, daß lediglich statistische Variablen, wie Anzahl der Nachrichten im Spool, Größe von Nachrichten, Anzahl der Fehler, Anzahl der Konvertierungen, etc. geführt werden. Diese Variablen erlauben ohne weitere Informationen nur einen groben Überblick über den Zustand des Systems. Alle Variablen der MIB sind *read-only* definiert, so daß kein steuernder Einfluß auf den MTA genommen werden kann. Dies hat zur Folge, daß die in Kapitel 2 erwähnten Bereiche des Konfigurations-, Sicherheits- und Abrechnungsmanagement überhaupt nicht abgedeckt werden können.

3.2.3 Message Tracking MIB

Die beiden bisherigen Ansätze haben gemein, daß sie sich nur mit der Prozeßüberwachung und den Netzwerkverbindungen von MTAs beschäftigen. Um das Message Tracking, also die Flußkontrolle von Emails im System realisieren zu können, arbeiten Bruce Ernst und Gordon Jones von der MADMAN Gruppe an einer weiteren MIB: Die *Message Tracking MIB* [ErJo97] ist ein Internet Working Draft, d.h. sie befindet sich noch im Entwurfsstadium.

Die Message Tracking MIB erlaubt ausschließlich, den Weg und den gegenwärtigen Status einer Nachricht zu ermitteln. Sie bietet keine auf dieser Information aufbauenden Dienste, wie etwa Routing-Kontrolle, Loop-Erkennung oder Sicherheits- und Abrechnungs-Mechanismen.

Die MIB setzt sich aus drei Tabellen zusammen:

- **mtaInformationTable**: diese Tabelle enthält Statusinformationen für jeden durch den Agenten überwachten MTA. Neben einem eindeutigen Index und einer Klartextbezeichnung für den MTA, wird der MTA-Typ (z.B. X.400 oder SNMP), der Beginn der Überwachung und ein alternativer Agent, der den MTA evtl. ebenfalls

überwacht, geführt. Wichtig ist an dieser Stelle vor allem der Beginn der Überwachung: Der Manager muß wissen, ob der Agent eine Anfrage nach einer alten Nachricht überhaupt beantworten kann.

- **msgTrackRequestTable**: Um einen Message Tracking Request zu starten, erzeugt der Manager eine neue Zeile in dieser Tabelle und füllt sie entsprechend aus. Die Spalten der Tabelle repräsentieren Suchkriterien, wie etwa Message-Id, Namen im From: oder To: Feld, Subject-Zeile, Nachrichtengröße, Datum, aber auch abstrakte Kriterien, wie Fehlercode, Adresse des Sender- oder Empfänger-MTAs, usw. Der Manager kann dabei beliebig viele Felder besetzen und damit den Umfang seiner Anfrage steuern. Durch Setzen von Kriterien, wie z.B. Fehlercode kann sich der Manager auch alle aufgetretenen Probleme anzeigen lassen, ohne speziell nach einer Nachricht zu suchen.
- **msgTrackResponseTable**: Hat der Agent die Suche beantwortet, so füllt er diese Tabelle mit Informationen zu allen Nachrichten, die den Anfragekriterien entsprechen. Dabei werden Informationen, wie z.B. Nachrichtenstatus (wartet, vermittelt, durch UA abgeholt), Zeit der Vermittlung, Message-Id, Next Hop, Sender, Empfänger, Subject-Zeile usw. bereitgestellt.

Eine typische Anfrage durchläuft den folgenden Zyklus:

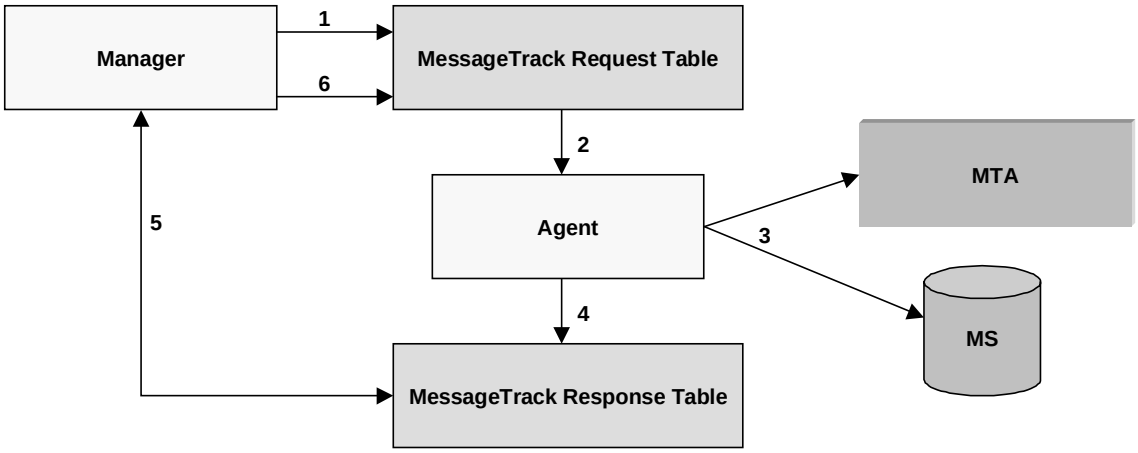


Abbildung 10 – Message Track Request

1. Der Manager erzeugt eine neue Zeile in der msgTrackRequestTable.
2. De Agent wertet die Tabellenzeile aus und formuliert seine Suchanfrage.
3. Der Agent startet seine Anfrage an den/die überwachten MTA(s) und Message Store(s).
4. Der Agent füllt die msgTrackResponseTable mit dem Ergebnis seiner Anfrage. Anschließend setzt er in der Anfragetabelle das Statusfeld der Anfrage auf „bearbeitet“.
5. Der Manager erkennt die Anfrage als bearbeitet und liest das Ergebnis aus.
6. Der Manager löscht seine Anfrage aus der Tracking Tabelle.

Auf Basis der erhaltenen Antwort kann die Management-Applikation weitere Anfragen an andere MTAs stellen und sich auf diese Weise den Weg einer Nachricht durch das System verfolgen.

Der vorgestellte Ansatz zum Message Tracking birgt Probleme in sich. Abgesehen davon, daß SNMP keine Kommunikation der Agenten untereinander ermöglicht, werden die Agenten auch niemals selbst (z.B. mittels *SNMP Notifications*) tätig. Dies hat zur Folge, daß auftretende Probleme im Mailsystem durch die Agenten nicht erkannt bzw. gemeldet werden. Eine frühzeitige Intervention durch einen Systembetreuer ist deshalb nicht möglich. Erst wenn Mails nicht mehr ausgeliefert werden, oder gar verloren gehen, und die Benutzer sich an ihre Administratoren wenden, können diese mittels Message Tracking versuchen, den Fehler nachzuvollziehen und zu beheben.

Ein zweites Problem ist, daß ein Benutzer in der Regel nicht weiß, welcher MTA im System seine Mail entgegen genommen hat. In einem größeren Netz mit komplexem Routing ist diese Information auch für den Administrator nicht immer verfügbar. Doch bei welchem MTA im System soll die Anfrage in diesem Fall beginnen? Der Manager kann nach dem Trial&Error-Prinzip vorgehen und einfach alle MTAs abarbeiten, was jedoch zu langen Antwortzeiten führt. Eine simultane Anfrage an alle Agenten führt wiederum zu hoher Netz- und Systemlast.

3.2.4 Bewertung

Mit der Mail Monitoring MIB und der Message Tracking MIB liegt ein normierter Ansatz für das Management einer Email-Infrastruktur vor. Die Lösung erlaubt es, sich schnell einen einfachen Überblick über den Betriebszustand des Systems zu verschaffen, also ob Mailserver up oder down sind, wie hoch das Nachrichtenaufkommen ist und wie es sich auf die MTAs verteilt. Außerdem kann der Verbleib einzelner Nachrichten gezielt ermittelt werden.

Um sich aus der Fülle der geführten Variablen einen detaillierten Einblick in die Email-Infrastruktur zu verschaffen, ist beim Operateur jedoch erhebliches Expertenwissen nötig. Es wäre denkbar, mit diesem Wissen und dem Abstützen auf weitere MIBs, wie etwa der *RMON MIB* zur Überwachung von Workstations eine Email Managementanwendung auf SNMP-Basis zu programmieren, welche eine Erkennung von Ferhlerzuständen und Performanceproblemen bei MTAs ermöglicht.

Die MIBs erfüllen allerdings nicht die Anforderungen aus den Bereichen Konfigurations-, Abrechnungs- und Leistungsmanagement. Hinzu kommt, daß die IETF das Thema Mailmanagement ausschließlich auf die Komponente MTA beschränkt. Auf dieser Basis läßt sich jedoch eine komplette Email-Infrastruktur nicht mehr abstrahieren und modellieren. Dies spiegelt sich auch darin, daß die Überlegungen aus Abschnitt 2.2.2 zu den Themen Routing- und Topologieservice, Benutzermanagement oder Policies von der IETF überhaupt nicht aufgegriffen werden.

Weitere Probleme beim Einsatz dieser Lösung resultieren aus dem starren Informations- und Rollenmodell von SNMP. Dadurch, daß auf die vorgestellten MIBs nur lesend zugegriffen werden kann, ist es der Managementanwendung im Fehlerfall nicht möglich, z.B. durch Ändern von Konfigurations- oder Routingeinstellungen, das ausgefallene System wieder in Gang zu setzen, oder zu überbrücken. Da die MIBs keine Notifications enthalten, muß die Managementanwendung die Agenten außerdem in regelmäßigen Intervallen pollen. Dies führt

zusammen mit der recht aufwendigen Implementierung des Message Tracking zu nicht unerheblicher Netz- und Systemlast, welche dem Anspruch von SNMP-Management „the impact of adding network management to managed nodes must be minimal“, widerspricht.

Die in 3.2.2 erwähnte Unterspezifizierung der Mail Monitoring MIB macht es außerdem erforderlich, daß alle Komponenten einer, auf den MADMAN-MIBs basierenden Managementlösung, also Managementanwendung und alle Agenten aus einer Hand kommen. Das Erweitern eines bestehenden Management-Frameworks um eine Email-Komponente ist so nur schwer möglich.

3.3 Normierte Ansätze der ISO

Auch die ISO hat ein umfassendes Framework zum Management eines MHS entwickelt. Im Standard [X460] wird eine Überblick über eine enstprechende Managementarchitektur gegeben.

Grundlage der Architektur ist das aus dem Telekommunikationsbereich bekannte *Telecommunications Management Network* (TMN) [M3200]. Das Modell soll dabei insbesondere folgende Schlüsselanforderungen erfüllen:

- Integriertes Management einer Managementdomäne (MD) anstatt dem Management einzelner Netzwerkelemente.
- Problemlose Integration heterogener Komponenten verschiedener Hersteller in einer MD.
- Einfache Erweiterung der Managementfunktionalität.

Hierzu definiert das TMN-Modell fünf hierarchische Schichten von Managementfunktionen und den Informationsaustausch zwischen den Schichten.

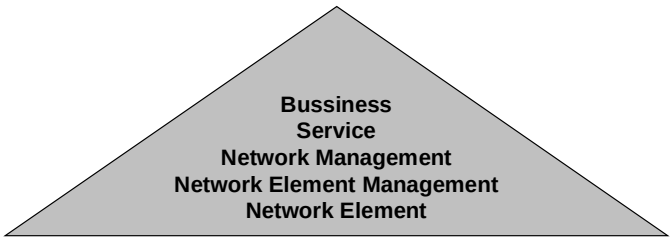


Abbildung 11 - TMN Schichtenmodell

Den einzelnen Schichten kommt dabei die folgende Funktionalität zu:

- *Network Element Layer*: Repräsentation physischer Komponten, wie MTAs, sowie des Datenflusses zwischen diesen Komponenten.
- *Network Element Management Layer*: Diese Schicht besteht aus Managementoperationen, welche auf ganze Gruppen von Managed Objects (MOs) angewandt werden können. Gruppen von MOs definieren sich dabei z.B. als Komponenten eines Herstellers oder Komponenten eines bestimmten Standorts.

- *Network Management Layer*: Auf dieser Ebene werden die einzelnen Komponenten zu einer globalen Sicht des Netzes abstrahiert. Dementsprechend sind hier nur noch Netze, sowie Beziehungen zwischen Netzen sichtbar.
- *Service Management Layer*: Schwerpunkt dieser Schicht ist die Verwaltung vom Netz erbrachter Dienste und die Unterstütung der Business Layer (z.B. durch statistische Informationen)
- *Bussiness Management Layer*: Integration des Netzes in den betriebswirtschaftlichen Ablauf.

Für eine Email-Infrastruktur ergibt sich damit folgende TMN-Modellierung:

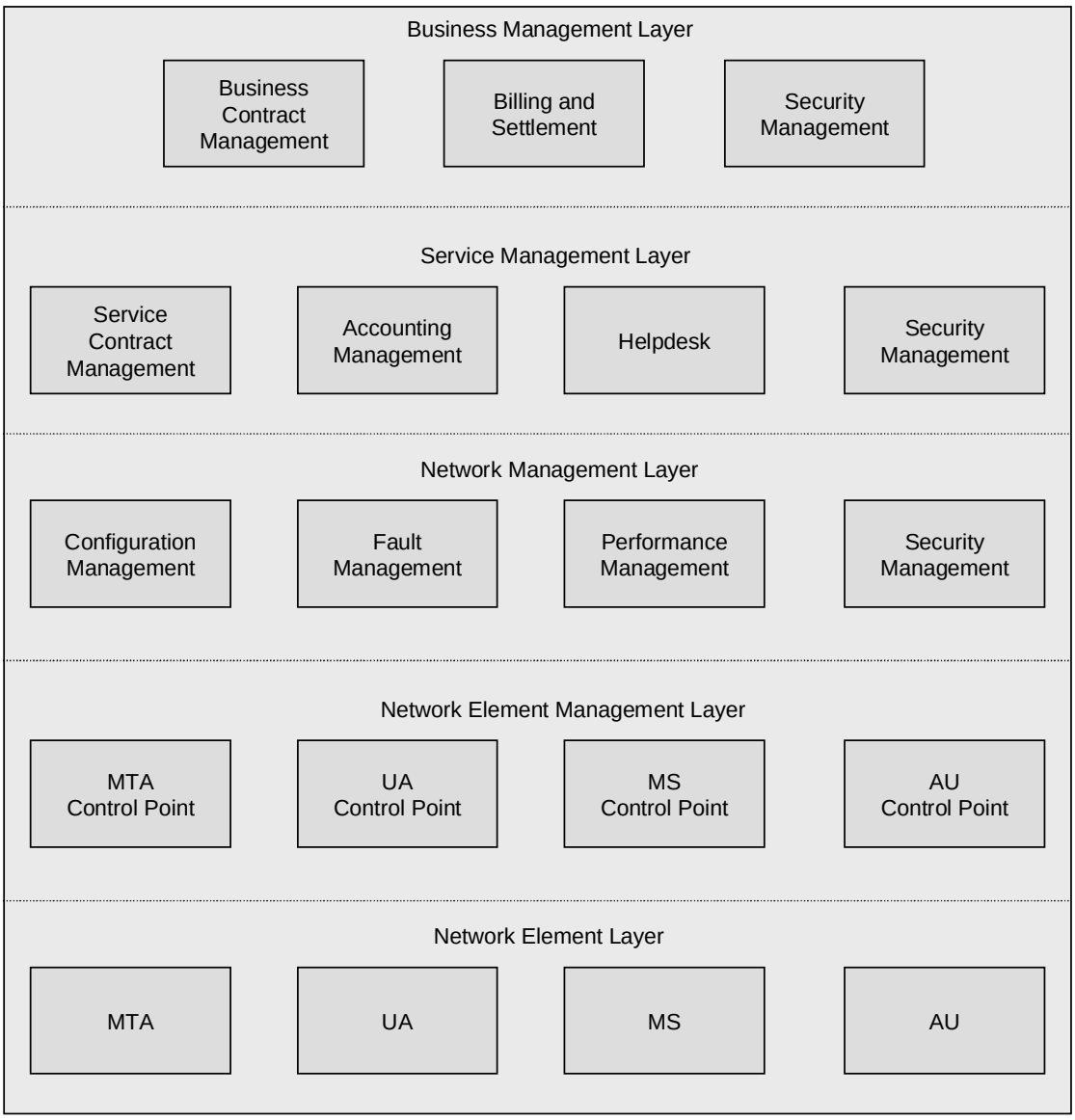


Abbildung 12 - TMN Email Modell (aus [X460])

Für das weitere Management der der Email-Infrastruktur hat die ISO nun generische und komponentenbezogene Managementfunktionalität standardisiert. Dabei wird auf das Informations-, Kommunikations- und Funktionsmodell des OSI-Managements (X.700ff) zurückgegriffen.

Diese Managementfunktionalität wird in weiteren ITU-T Empfehlungen festgehalten:

Name des Standards	ITU-T Bezeich-nung
Managementfunktionen	
Message Handling Systems (MHS) management: Logging Information	X.462
Message Handling Systems (MHS) management: Security management functions	X.463
Message Handling Systems (MHS) management: Configuration management func.	X.464
Message Handling Systems (MHS) management: Fault management functions	X.465
Message Handling Systems (MHS) management: Performance Management func.	X.466
Management von Komponenten	
Message Handling Systems (MHS) management: Message transfer agent entity	X.467
Message Handling Systems (MHS) management: User agent entity	X.468
Message Handling Systems (MHS) management: Message store entity	X.469
Message Handling Systems (MHS) management: Access unit entity	X.470

Eine detailliertere Beschreibung des TMN-Modells und der Folgestandards findet man in [Maur95].

Bewertet man nun diesen Ansatz der ISO, so stellt man als erstes fest, daß er wesentlich ausgereifter und strukturierter als die von verschiedenen Arbeitsgruppen der IETF erarbeiteten Vorschläge zum Mailmanagement ist. Obwohl das TMN-Modell für Netz- und Systemmanagement entwickelt wurde, eignet es sich auch zur Beschreibung von Anwendungsmanagement.

Bei der vorgestellten TMN-Modellierung des Email-Managements handelt es sich allerdings nicht um einen Top-Down Ansatz, sondern um eine Bottom-Up Abstraktion aus den in [X460] erarbeiteten Komponenten. Dies liegt daran, daß das ISO-Modell einer Email-Architektur durch Nachbildung des bereits bestehenden X.400 Standards entstanden ist und die Managementfunktionalität erst später hinzugefügt wurde. Der Ansatz läßt auch ein für das Dienstmanagement außerordentlich wichtiges Problem offen: Die Frage nach der Abbildung zwischen verschiedenen Schichten des Modells. So ist keineswegs geklärt, wann und nach welchem Prinzip Probleme auf der Network Element Schicht auf höhere Schichten eskaliert werden müssen. Wann und wie etwa wird das Versagen eines MTA zum Performanzproblem, das auf der viel abstrakteren Network Management Schicht sichtbar wird.

Was dem ISO-Modell ebenfalls fehlt, ist eine Analogie zum Engineering Viewpoint des RM-ODP, d.h. es werden keinerlei Vorschläge zur Implementierung der Managementfunktionalität gemacht.

Eine kurze Sichtung der Standards X.462-X.467 zeigt eine Fülle von Managementobjekten, die bei der Erstellung des Managementmodells in Kapitel 4 teilweise wieder aufgegriffen werden. Es stellt sich allerdings die Frage, ob eine Implementierung aller von der ISO geforderten Managementobjekte mit überschaubarem Aufwand zu realisieren ist und ob das resultierende Managementsystem die Performanz des Message Handling Systems nicht stark beeinflussen würde. So erkennen die Verfasser von X.460 selbst: „It is essential that the overhead of management not overwhelm the actual message traffic on a network. The percentage of management information comparative to other network traffic should be considered when designing the management services“.

3.4 Einsatzszenario - die BMW AG

3.4.1 Infrastruktur

Wie in allen großen Betrieben ist die heterogene Email-Infrastruktur der BMW AG historisch gewachsen.

Wie aus Abbildung 13 ersichtlich, ist das Produkt *Mailhub* der Firma *Control Data Systems*, als zentraler Mail- und Verzeichnisdienst der Kern der Infrastruktur.

Die Außenanbindung des Mailsystems erfolgt mittels SMTP-Protokoll über einen Firewall-Rechner ins Internet und über X.400 und einen X.400-Provider. Eingehende Mails werden an das Mailhub System zugestellt. Das weitere Routing wird durch Einträge im X.500-Verzeichnis bestimmt. Für die Münchner Benutzer der BMW AG erfolgt die Zustellung an eine der Maschinen HPMUC1 bis HPMUC4 (in der Abbildung sind nur die ersten beiden Maschinen zu sehen). Auf diesen Maschinen läuft die Software *OpenMail* von *Hewlett-Packard*. Bei der Mailhub Maschine handelt es sich um eine HP 9000/K260 mit zwei Prozessoren und 512 MB RAM. OpenMail läuft jeweils auf einer HP 9000/K570 mit vier Prozessoren und 2 Gigabyte RAM. Als Platten werden für beide Systeme EMC^2-Platten von Symetrics über einen externen SCSI-Bus angeschlossen. Lediglich das Betriebssystem und Teile der Anwendungssoftware sind lokal installiert.

Der Zugriff durch die Benutzer auf die HP Maschinen erfolgt i.d.R. über den Netscape Mail-Client, also über offene Internet-Standards, wie SMTP bzw. POP3 oder IMAP4. Der Zugriff auf das Adreßverzeichnis erfolgt über das LDAP-Protokoll. Der Adreßraum in München ist flach, d.h. für interne, wie externe Kommunikation werden Email-Adressen nach dem Schema Vorname.Nachname@bmw.de verwendet.

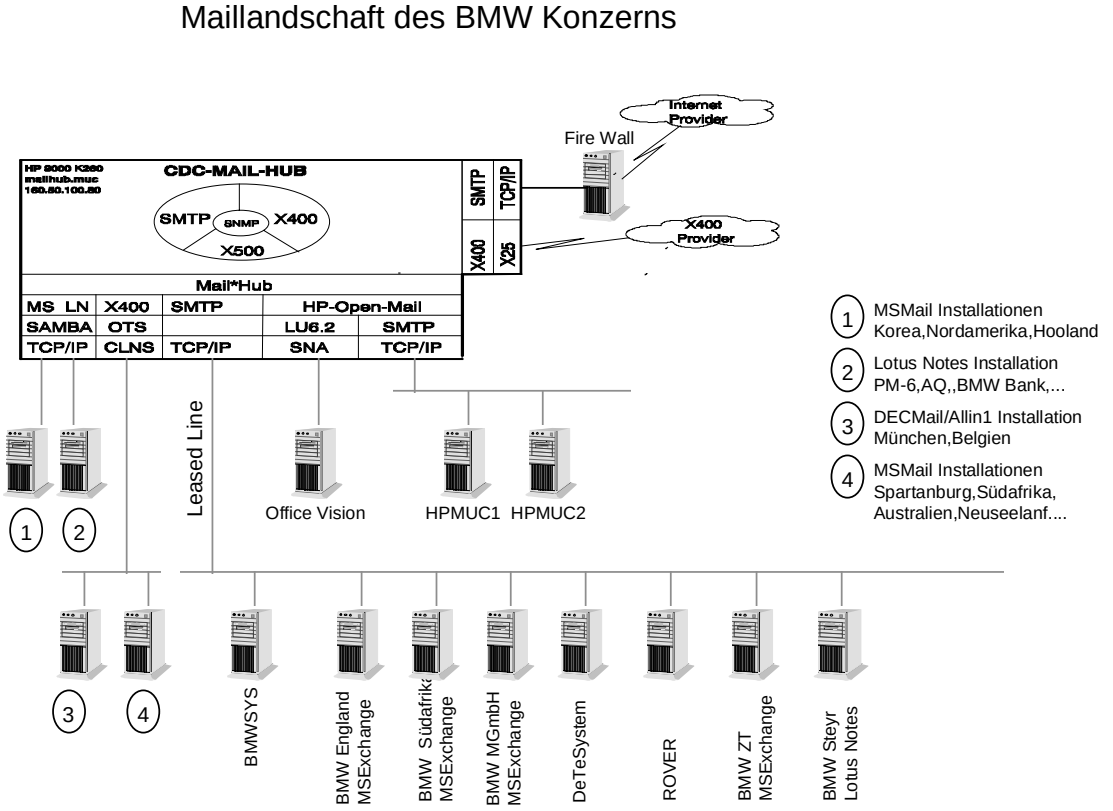


Abbildung 13 - Email-Infrastruktur BMW AG

Weitere Mailsysteme, die ebenfalls noch zum Einsatz kommen, wie Lotus Notes, OfficeVision, MSMail oder DECMail werden über Gateways erreicht.

3.4.2 Management des Systems

Beim Management des Systems muß nach Managementbereichen unterschieden werden. Zum Konfigurationsmanagement gehört das Setzen von Konfigurationsoptionen und in weiterem Sinne das Anlegen und Löschen von Benutzern. Dies wird zentral am Mailhub System erledigt: Änderungen am Benutzerbestand werden durch Skripten automatisch auf alle weiteren Rechner propagiert.

Das Leistungsmanagement wird durch Auswerten von Logdateien mittels Skripten abgewickelt. Dabei werden das Mailaufkommen pro Rechner festgehalten und Verkehrsanalysen erstellt. Diese Zahlen dienen der Planung für den weiteren Ausbau. Benutzerspezifische Daten werden in der Auswertung nicht erfaßt.

Für den Fall eines Totalausfalls wurde auf den OpenMail Rechnern das Produkt *MC ServiceGuard* der Firma HP installiert. Das Produkt erlaubt, beim Ausfall eines Servers auf einen bereitgestellten Backup Server umzuschalten und die Konfiguration zu übernehmen. Für den Mailhub Server gibt es ebenfalls einen identischen Backup-Server, auf welchen bei Systemausfall umgeschaltet wird.

Im Bereich des Fehlermanagements sind die Email-Server als Managed Nodes in eine unternehmensweite, zentrale Managementlösung intergriert. Als Managementplattform wird hierbei *Tivoli* [Tivo98a] der Firma *IBM* eingesetzt. Das Fehlermanagement umfaßt das klassische Prozeßmanagement sowie das Ressourcenmanagement der Mailhosts. Hierzu wurden auf den Mailrechnern Tivoli-Monitore [BMW98], also Agenten installiert, die in regelmäßigen Abständen prüfen, ob Prozesse noch existieren, wie hoch die CPU-Last ist, und zu welchem Grad die Filesysteme gefüllt sind. Ist ein Prozeß nicht mehr vorhanden, so wird er meist neu gestartet. Alle weiteren Probleme lösen eine Ereignis aus, welches die Managementapplikation meldet, ein steuernder Eingriff ist nicht vorgesehen.

Das Tivoli-Management umfaßt die folgenden Bereiche:

- OpenMail Prozesse: Überprüfen verschiedener Prozesse, wie ewa inetd, Notification Server, License Server, Shared Memory Daemon, Database Monitor, etc.
- OpenMail Services: Service Router, Local Delivery Agent, Sendmail Interface, Local Client Interface, Print Server, Directory Synchronization, etc.
- OpenMail Queues: Service Router, Bulletin Board Server, X.400 Interface, etc.
- Diverse Prozesse: Netscape und Samba.
- Filesysteme: Schwellwertüberwachung von /, /tmp, /var, etc.
- Systemauslastung: Schwellwertüberwachung von CPU-Last, Memory, IO, Swap.
- X.500: Prozeßüberwachung von LmGrd, GDServer, LDAP Daemon, etc.
- Netzwerk: Test von Interfaces und weiterer Komponenten.

Sämtliche Tivoli-Monitore sind als Skripten in Perl 5 implementiert. Da Mailhub und OpenMail derzeit über keine direkten Managementschnittstellten verfügen, müssen sie sich normaler Unix Befehle, wie etwa *ps, bdf, omstat, netstat* usw. bedienen.

Dabei ergeben sich zwei Probleme: Die Implementierung in Perl und das Zurückgreifen auf Shellkommandos ist nicht ressourcenschonend und stellt eine nicht unerhebliche Belastung für die überwachten Rechner dar. Aus diesem Grund werden die Monitore typischerweise nur alle 30 Minuten gestartet, wodurch Fehler evtl. zu spät, d.h. erst dann, wenn der Dienst bereits negativ beeinflußt ist, entdeckt werden. Ein weiteres Problem ist, daß schwere Fehlerzustände, wie etwa das Überlaufen eines Filesystems auch die Agenten in Mitleidenschaft zieht, weshalb keine Fehlermeldung an den Manager erfolgt.

3.5 Zusammenfassung

In diesem Kapitel wurden verschiedene Managementansätze für eine Email-Infrastruktur untersucht. Alle besprochenen Lösungsansätze decken jedoch nur Teilaspekte der Forderungen aus Kapitel 2 ab.

Eine mögliche Lösung wäre es, die unterschiedlichen Ansätze zusammenzuführen und zu komplettieren. Dies scheitert jedoch an verschiedenen Informationsmodellen der jeweiligen Managementarchitekturen.

Im nächsten Kapitel wird deshalb auf der Basis von RM—ODP ein unabhängiges, objektorientiertes Modell zum Management einer Email-Infrastruktur entwickelt.

4 Entwicklung eines Managementmodells auf Basis von RM-ODP

Im folgenden Kapitel wird ein Modell zum Management einer Email-Infrastruktur auf Basis von RM-ODP entwickelt. Abschnitt 4.1 gibt einen kurzen Überblick über RM-ODP. Eine Einführung in OMT und das verwendete CASE-Tool StP folgen in Abschnitt 4.2. Die Modellierung der Infrastruktur anhand der fünf ODP-Sichten erfolgt schließlich in Abschnitt 4.3.

4.1 RM-ODP als generische konzeptuelle Plattform

Im Gegensatz zum Management von Komponenten, wie etwa Routern, Switches oder Workstations, welchem trotz einer Vielzahl von unterschiedlichen Managementarchitekturen ein gemeinsames Grundverständnis über die zu managende Ressource und die Managementaufgaben zugrunde liegt, ist die Frage, wie das Management einer verteilten Anwendung zu charakterisieren sei, ein aktuelles und kontrovers diskutiertes Forschungsgebiet.

Die Diskussion um die Modellierung und das Management von verteilten Anwendungen wird durch eine Vielzahl von verfügbaren Managementarchitekturen mit jeweils unterschiedlichen Informations-, Kommunikations- und Organisationsmodellen erschwert. Beispiele für solche Managementarchitekturen sind z.B. das OSI-Management, das Internetmanagement, sowie das Web-Based Enterprise Management (WBEM).

Durch zunehmende Globalisierung und der sich abzeichnenden Konvergenz von Einsatzgebieten mit traditionell unterschiedlichen Managementformen, wie etwa Internet- und Telekommunikationsdiensten, wuchs der Druck, eine konzeptuelle Plattform zu entwickeln, welche es erlaubt, Modelle zur Beschreibung verteilter Anwendungen zu entwickeln, die unabhängig von bestehenden Architekturen sind. Gleichzeitig sollte es mit dem Ziel der einfachen Implementierung aber möglich sein, die so entwickelten, generischen Modelle mit geringem Aufwand auf bestehende Architekturen abzubilden.

Aus diesem Grund begann 1987 die Entwicklung von *Open Distributed Processing* (ODP), bei der die folgenden Ziele realisiert werden sollten:

- Interoperabilität:
  Eine ODP-Herstellerimplementierung sollte mit einer Implementierung eines anderen Herstellers zusammenarbeiten können.
- Portabilität:
  Eine Anwendung, die auf einer ODP-Implementierung eines Herstellers läuft, sollte direkt auf die Implementierung eines anderen Herstellers übertragbar sein.
- Transparenz
  Die Verteiltheit wird vor dem Benutzer verborgen.

ODP wurde 1992 durch die ISO (ISO 10746) und die ITU-T [X901] standardisiert. Die Standards werden als *Reference Model for Open Distributed Processing* (RM-ODP) zusammengefaßt.

Obwohl ODP nicht explizit für Managementzwecke, sondern generell zur Modellierung verteilter Anwendungen entworfen wurde, eignet es sich aufgrund der nachfolgend beschriebenen Eigenschaften und der Tatsache, daß auch Dienstmanagement eine verteilte Anwendung ist sehr gut zum Erstellen eines Managementmodells.

Da die ODP-Konzepte von *Transparencies*, *Functions* und *Viewpoints* zur Modellierung des Email-Managements benutzt werden, werden sie hier kurz vorgestellt. Die Beschreibung orientiert sich dabei an [Joyn97].

- **ODP-Transparencies**: Basis der ODP-Abstraktion bestehender Architekturen ist die Transparenz der Verteiltheit. Dies bedeutet, daß die Verteiltheit der Anwendung verborgen wird. Für den Anwender oder Programmierer soll die Anwendung so erscheinen, als würde sie lokal laufen. Ein Beispiel hierfür ist die Sicht eines Endanwenders auf ein Email-System. Schreibt der Benutzer eine Mail, so interagiert er mit dem Client auf seiner Workstation. Mit dem Abschicken der Mail ist der Vorgang für ihn abgeschlossen. Alle weiteren Vorgänge bleiben vor dem Benutzer verborgen. Der ODP-Standard definiert dabei die folgenden Transparenzen, die aber nicht alle von einer verteilten Anwendung genutzt werden müssen: Access Transparency, Failure Transparency, Location Transparency, Migration Transparency, Persitance Transparency, Relocation Transparency, Replication Transparency und Transaction Transparency.
- **ODP-Functions**: Während die ODP-Transparencies Anforderungen an ein verteiltes System aufstellen, bieten die ODP-Functions eine Basisfunktionalität, die zur Konstruktion des ODP-Systems genutzt werden können. Die ODP-Functions lehnen sich dabei teilweise an Funktionalitäten bereits existierender Architekturen (CORBA) an. Die Funktionen werden grob in vier Bereiche unterteilt:
  - *Management Functions*: Instanziierung, Überwachung, Verschieben und Löschen von Prozessen, Prozeßgruppen, Betriebssystemen und Rechnern.
  - *Coordination Functions*: Stellt Koordinationsfunktionalität zwischen Objekten zur Verfügung. Beispiele hierfür sind Benachrichtigungs-, Gruppierungs-, Migrations- und Transactionsfunktionen.
  - *Repository Functions*: Verwaltet das Object Repository, Beispiele sind *Storage Function, Type Repository Function* oder *Trading Function.*
  - *Security Funtions*: Sicherheitsfunktionen. Beispiele hierfür sind *Access Control Function, Security Audit Function* oder *Key Management Function*.
- **ODP-Viewpoints**: Die ODP-Viewpoints geben dem Entwickler ein Konzept zur Top-Down-Modellierung eines verteilten Systems an die Hand. Sie sind für den Endbenutzer nicht von Belang. Die folgenden fünf Sichten auf ein verteiltes System werden definiert:
  - *Enterprise Viewpoint*: die Unternehmens- bzw. Betreibersicht auf ein System definiert die Anforderungen, Zweck, Einsatzumgebung und Policies einer verteilten Anwendung. In dieser Sicht definierte Anforderungen und Einschränkungen müssen in allen anderen Sichten eingehalten werden.
  - *Information Viewpoint*: Diese Sicht auf ein verteiltes System konzentriert sich auf Informationsfluß und –verarbeitung innerhalb der Anwendung.

- *Computational Viewpoint*: In dieser Sicht wird die Gesamtunktionalität des Systems erstmalig auf Komponenten aufgesplittet. Die verteilte Anwendung wird als eine Menge von Objekten dargestellt, welche über Schnittstellen miteinander kommunizieren. Die tatsächliche Verteilung der Objekte wird dabei noch außeracht gelassen. Zur Modellierung des Computational Viewpoint stellt ODP generische Basisklassen für Objekte und Schnittstellen zur Verfügung. Dabei handelt es sich z.B. um *Computational Objects* oder *Signal- Stream-* bzw. *Operation-Interfaces*.
- *Engineering Viewpoint*: Diese Sicht beschäftigt sich explizit mit der Verteilung der Objekte über das System. Die im Computational Viewpoint definierten Objekte werden dabei auf die kleinsten Einheiten (Basic Engineering Object) des Engineering Viewpoints abgebildet. Diese Einheiten kommunizieren über Kanäle (*Channels*). Auch auf dieser Ebene stellt ODP wieder generische Basisklassen, wie *Node*, *Nucleus*, *Cluster*, etc. zur Verfügung.
- *Technology Viewpoint*: In dieser Sicht wird schließlich die technische Realisierung des Systems festgelegt. Welche Rechen- und Netzkapazität wird benötigt? Welche Hardware kommt zum Einsatz, welche Produkte werden gekauft, in welcher Programmiersprache wird die Lösung realisiert?

Wie gestaltet sich nun der Prozeß der Softwareentwicklung unter Zuhilfenahme der ODP-Konzepte? Alexander Keller und Bernhard Neumair stellen dazu in [KeNe97c] folgendes Vorgehen vor:

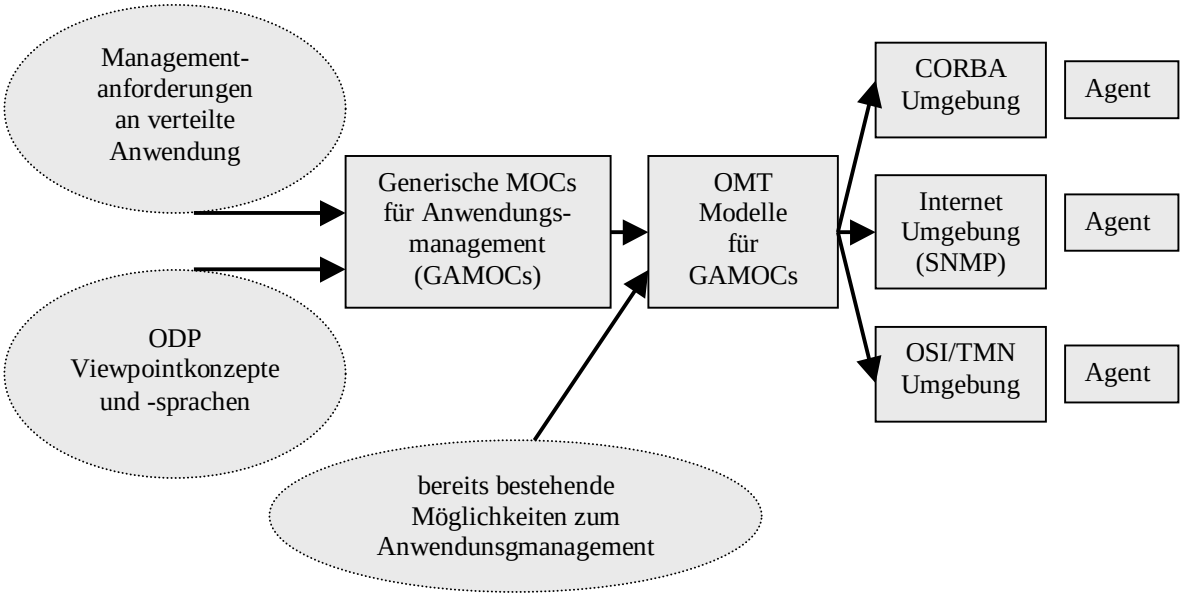


Abbildung 14 – Softwareentwicklung mit ODP

In einem ersten Schritt werden die Managementanforderungen und Einsatzszenarien zusammengetragen. Unter Berücksichtigung der ODP-Konzepte und Viewpoints werden *Generic Managed Object Classes for Application Management* (GAMOCs) generiert.

Die generischen Klassen werden anschließend verfeinert und mit Attributen und Methoden versehen. Dabei handelt es sich um einen Top-Down Ansatz, d.h. die Attribute und Methoden der Klassen resultieren aus der Anforderungsanalyse.

In einem dritten Schritt wird der Top-Down Ansatz durch eine Bottom-Up Analyse komplementiert – zu diesem Zweck werden die managementrelevanten Informationen, welche die verteilte Anwendung bereits liefern kann, untersucht.

Die Diplomarbeit folgt diesem Ansatz. Nach einer Vorstellung der Modellierungssprache OMT und des Case-Tools Software through Pictures (StP) wird, unter Berücksichtigung der ODP-Viewpoints, ein Ansatz für das Management von Emails im Top-Down Verfahren modelliert. Dieser Ansatz wird in Kapitel 5 durch eine Bottom-Up Analyse für einen prototypischen Managementagenten ergänzt.

4.2 Modellierung mittels OMT und StP

4.2.1 Object Modeling Technique (OMT)

Die Heterogenität existierender Managementansätze spiegelt sich auch in der Vielzahl von Notationen zur Beschreibung ihrer Informationsmodelle wieder. So benutzt das Internetmanagement zur Beschreibung seiner *Managed Objects* (MOs) eine Untermenge der Makrosprache ASN.1, das OSI Management nutzt GDMO/ASN.1 für seinen objektorientierten Ansatz und die Schnittstellen von CORBA Objekte werden in CORBA-IDL spezifiziert.

Für eine Umsetzung von ODP eignen sich diese Notationen nur bedingt. Ähnlich wie bei ODP selbst, ist eine generische Beschreibung der MOs gefragt, welche unabhängig vom Informationsmodell der unterliegenden Managementarchitektur ist. Gleichzeitig muß auch hier - im Hinblick auf eine schnelle Implementierung - eine einfache Abbildung auf bestehende Notationen möglich sein.

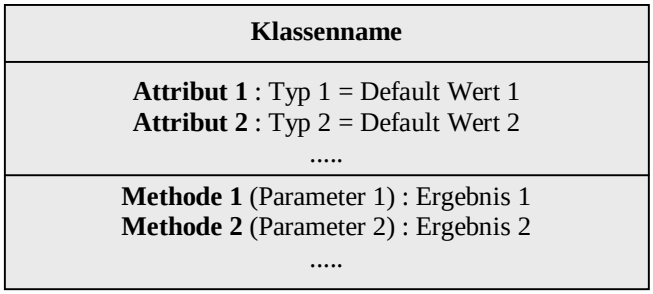
Die objektorientierte Beschreibungssprache *Object Modeling Technique* [Rum91], von James Rumbaugh und General Electric seit 1987 entwickelt, wird diesen Anforderungen gerecht. OMT wurde für eine Vielzahl von Software Engineering Projekten genutzt und ist heute nach [KeNe97c] im Telekommunikationsbereich weit verbreitet.[4]

In dieser Arbeit wird OMT zur graphischen Modellierung der MOs einer Email-Infrastruktur verwendet. Die benutzten Konzepte sind aus dem objektorientierten Software Engineering bekannt:

- **Klassen**: eine Klasse ist eine Menge von Attributen und darauf arbeitenden Funktionen, den Methoden. In OMT wird eine Klasse wie folgt dargestellt:

[4] Die Weiterentwicklung von OMT ist die Unified Modeling Language.

Abbildung 15 – OMT Klasse



- **Assoziation**: Assoziationen beschreiben die Relationen bzw. den Informationsfluß zwischen den Objektklassen. Assoziationen werden als Linien zwischen den Objektklassen dargestellt und oft mit Verben beschriftet. Obwohl diese Beschriftungen oft eine Richtung implizieren, sind Assoziationen in OMT bidirektional:

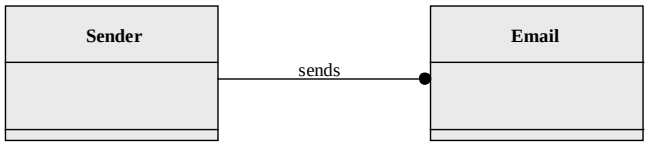


Abbildung 16 – OMT Assoziation

- **Generalisierung**: Die Generalisierung modelliert das bekannte Prinzip der Vererbung. Die Verfeinerung der Unterklassen findet duch Hinzufügen bzw. Überschreiben von Attributen und Methoden statt:

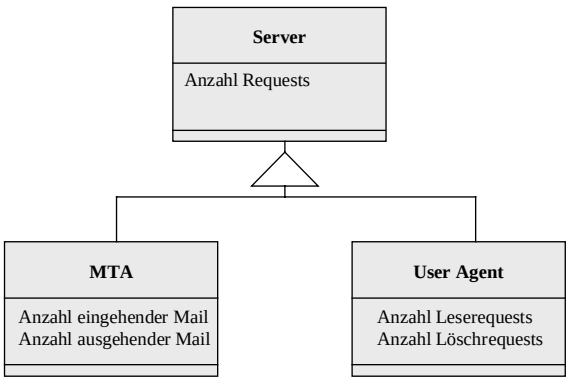


Abbildung 17 – OMT Generalisierung

- **Aggregation**: Die Aggregation modelliert eine Enthaltenseins-Beziehung. Dies erlaubt, ein Objekt aus mehreren Objekten zusammenzusetzen:

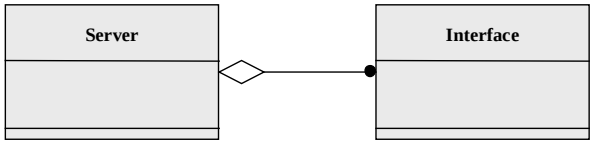


Abbildung 18 – OMT Aggregation

Nach exakter Typisierung der Attribute und Parametrisierung der Methoden können die OMT Objekte auf bestehende Notationen abgebildet werden, d.h. es können z.B: IDL-Schnittstellen erzeugt werden.

4.2.2 Software through Pictures (StP)

Software through Pictures (StP) ist ein CASE-Tool, welches das einfache Erstellen von ODP-Objektmodellen erlaubt. Die Software wurde ursprünglich von der Firma Interactive Development Environments erstellt, welche von der Firma Aonix übernommen wurde.

Zur Modellierung können verschiedene Notationen verwendet werden: Komponenten stehen für OMT, Booch und UML zur Verfügung. Sämtliche Komponenten sind mittels Client-Server-Konzept an den multiuserfähigen StP-Kern gebunden. Alle Komponenten verwalten ihre Daten in einem relationalen DBMS.

In dieser Diplomarbeit wurde StP Version 2.4.2 mit OMT-Modellierung verwendet. Nach dem Aufrufen der Software findet der Anwender den StP Desktop vor, von welchem die einzelnen Werkzeuge gestartet werden können.

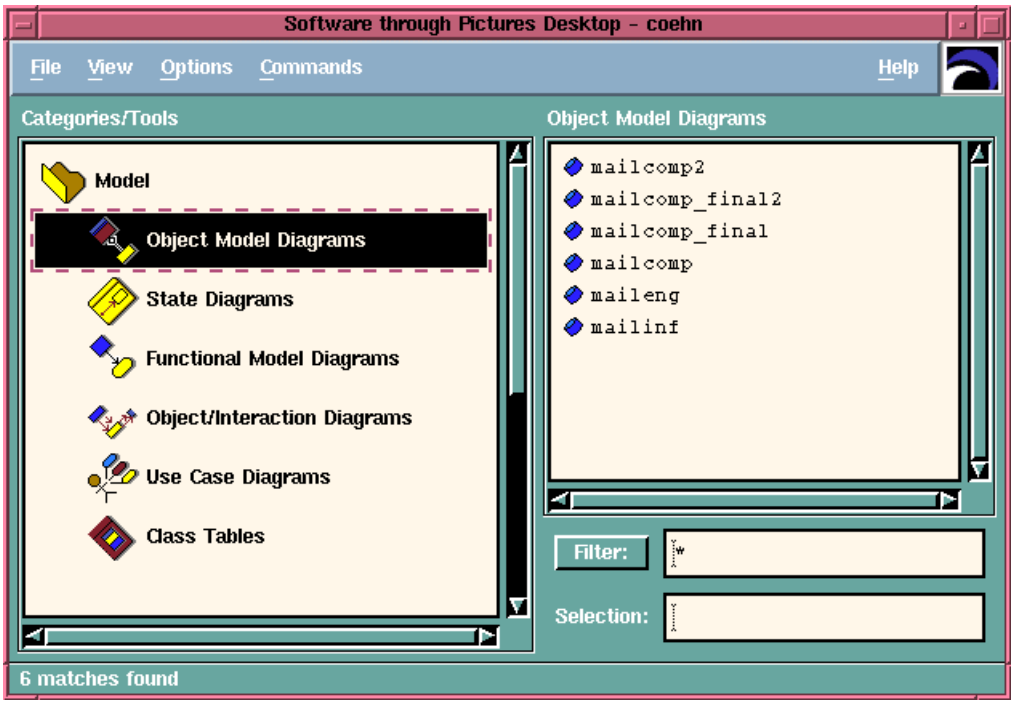


Abbildung 19 - Der StP Desktop

Die wichtigsten Komponenten sind dabei:

- Diagrammeditoren: Mit diesen Editoren können Use Cases, Objektmodelle, Objektinteraktionsmodelle, sowie dynamische und funktionale Modelle erstellt werden.
- Tabelleneditoren: Diese Editoren erlauben das Verändern von Klassen- und Zustandstabellen. Die Attribute und Methoden der Objekte werden hier genau spezifiziert, was für eine spätere Codegenerierung wichtig ist.
- Weitere Werkzeuge: Die hier zur Verfügung gestellten Werkzeuge, wie z.B. Druckfunktionen werden von den Diagramm- und Tabelleneditore benutzt oder dienen der System- und Benutzerverwaltung

StP erlaubt aus den erstellen Objektmodellen Code, wie z.B. C++ Klassen oder IDL Interfaces zu generieren. Die Codegenerierung ist dabei auf die Umsetzung der Objektmodelle beschränkt, dynamische und Funktionalmodelle können nicht ausgewertet werden.

In dieser Diplomarbeit wurde deshalb hauptsächlich der Objektmodelleditor benutzt:

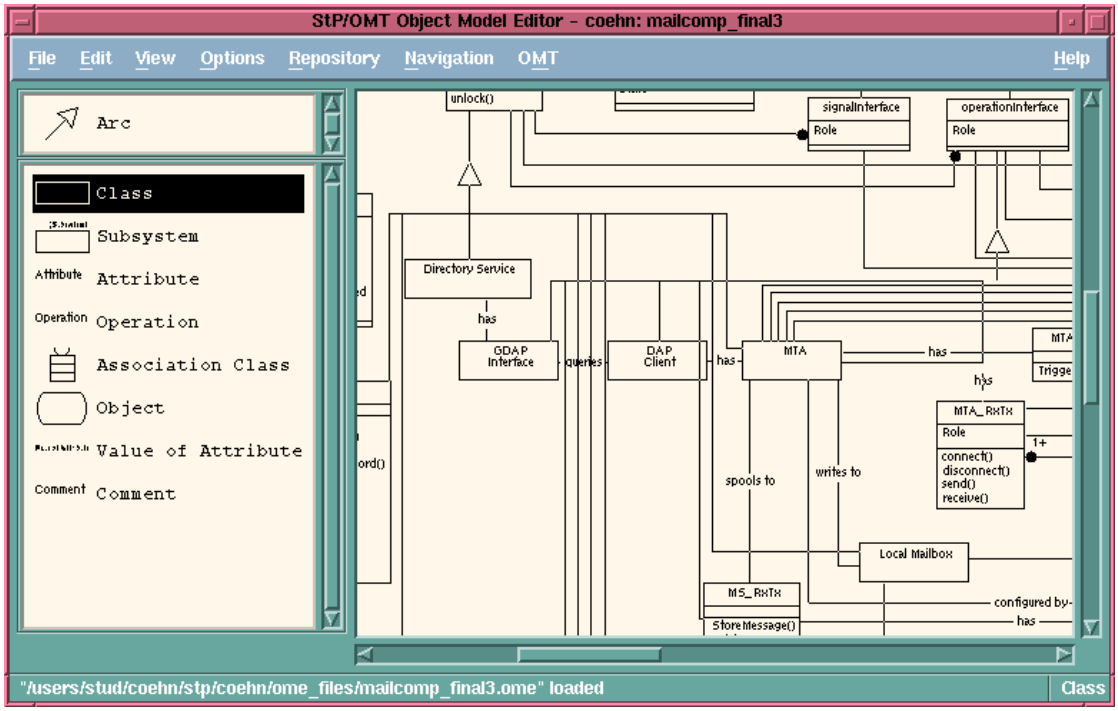


Abbildung 20 - StP Objektmodelleditor

4.3 Modellierung der Email-Infrastruktur

4.3.1 Designentscheidungen

Bei der Konzipierung der einzelnen ODP-Viewpoints gab es jeweils mehrere Möglichkeiten zur Objektmodellierung. Insbesondere trat die Frage auf, inwieweit Objekte des Computational Viewpoints bereits im Informational Viewpoint sichtbar werden müssen.

Email ist eine verteilte Anwendung. Instrumentiert man die verschiedenen Komponenten durch Agenten mit Managementfunktionalität, so ist auch die hieraus resultierende Managementanwendung zwangsläufig wieder verteilt. Auch diese verteilte Managementanwendung kann nun wieder mit ODP modelliert werden. Die Diplomarbeit modelliert jedoch den **Dienst Email und seine Managementschnittstelle** und nicht eine konkrete Managementapplikation.

Dies hat für die Modellierung der ODP-Viewpoints folgende Konsequenzen:

- Enterprise Viewpoint: Hier werden Anforderungen an Email aus Betreibersicht dargestellt. Eine dieser Anforderungen ist das Management des Email-Systems.
- Informational Viewpoint: Es wird der Informationsfluß der Email-Infrastruktur dargestellt. Gleichzeitig werden Managementanforderungen festgehalten, die sich ohne Kenntnis der zugrunde liegenden Komponenten fordern lassen. Der Informationsfluß einer etwaigen Managementanwendung wird nicht modelliert. Komponenten des Mailsystems, wie etwa MTAs, die im Informational Viewpoint einer Managementanwendung sichtbar wären, werden deshalb nicht modelliert, sondern treten erst im Computational Viewpoint auf.
- Computational Viewpoint: Um die Management-Anforderungen aus dem Informational Viewpoint erfüllen zu können, werden die Komponenten der Infrastruktur modelliert und mit Managementfunktionalität ausgestattet.
- Engineering Viewpoint: Im Engineering Viewpoint werden bereits konkrete Entscheidungen im Hinblick auf eine spätere Implementierung getroffen. Da nicht das gesamte Managementmodell, sondern nur ein Ausschnitt daraus implementiert werden, werden nicht alle *Computational Objects*, sondern nur die für die Implementierung relevante Objekte in den Engineering Viewpoint übertragen.
- Technology Viewpoint: Der Technology Viewpoint wird ihm Rahmen der prototypischen Implementierung in Kapitel 5 behandelt.

4.3.2 Enterprise Viewpoint

Aus Unternehmens- bzw. Betreibersicht stellt Email ein zusätzlich eingeführtes Kommunikationsmedium dar, an welches die folgenden Anforderungen gestellt werden:

- Punkt-zu-Punkt Kommunikation.
- Punkt-zu-Mehrpunkt Kommunikation.
- Kommunikation ist asynchron.

- Die übertragenen Kommunikationseinheiten sind mittlere bis große Nachrichten.
- Weitere Forderungen: Dabei handelt es sich um Policies, also um spezielle Forderungen, die vom System umgesetzt werden müssen. Beispiele hierfür sind z.B. Zugangspolicies, Policies bezüglich Zertifizierung und Verschlüsselung oder Policies, die den Ressourcenverbrauch durch einzelne Benutzer regulieren.
- Die Anwendung muß über eine Managementschnittstelle verfügen.

4.3.3 Informational Viewpoint

Informationsfluß- und verarbeitung in einem Email-System, sowie Management-anforderungen an den Dienst Email, werden in Abbildung 21 dargestellt.

Basis ist hier wieder die im Enterprise Viewpoint geforderte Punkt-zu-Punkt bzw. Punkt-zu-Mehrpunkt Kommunikation. Eine Information zu einem bestimmten Thema „fließt“ von einer Quelle zu einem oder mehreren Zielen.

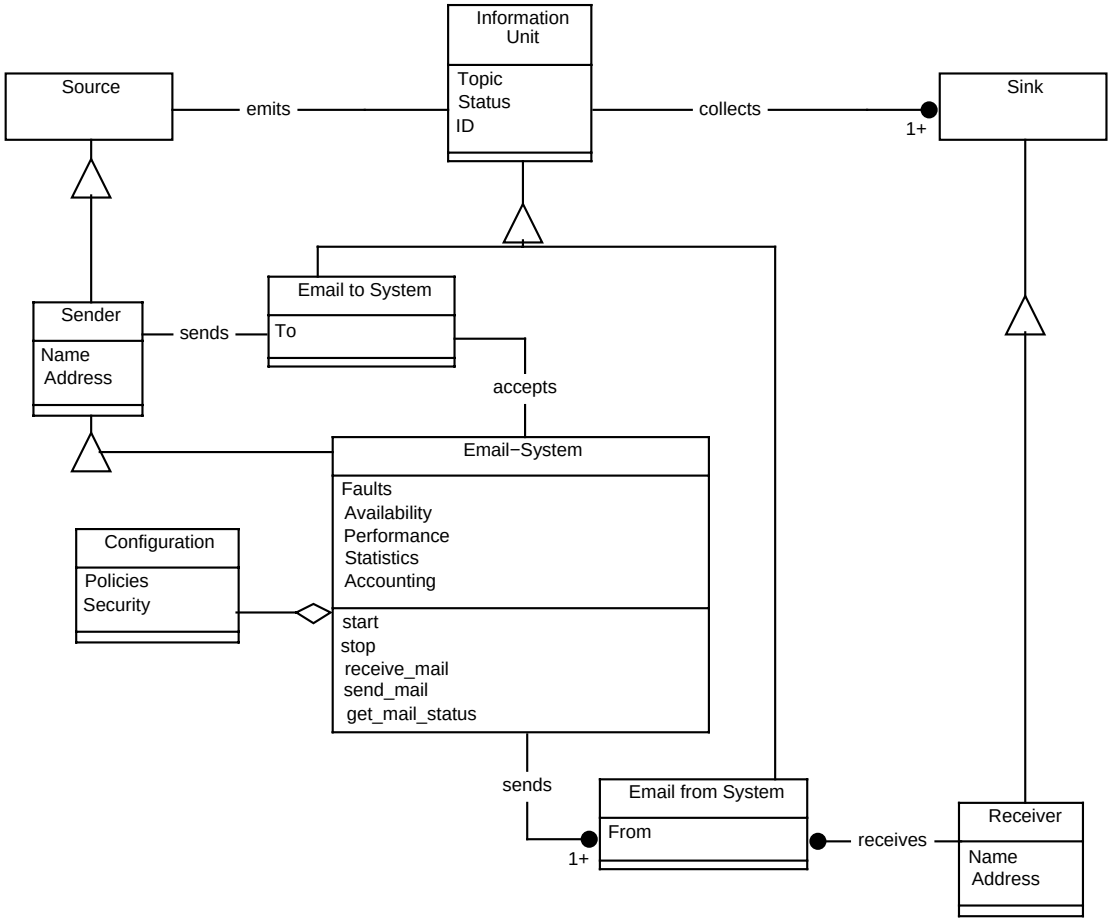


Abbildung 21 - Informational Viewpoint

In der OMT-Modellierung wird dies durch drei abstrakte Basisklassen dargestellt: *Source*, *InformationUnit* und *Sink*. Über *Source* und *Sink* ist zunächst nichts weiter bekannt, Quelle einer Information kann eine Person ebenso wie z.B. ein Meßgerät sein.

Dabei fließt die Information zu einer oder mehreren Senken, was durch die entsprechende OMT-Multiplizität im Diagramm ersichtlich wird. Die Klasse *InformationUnit* verfügt über das Attribut *Topic*, welches verdeutlicht, daß eine Instanz dieses Objekts nur eine ganz konkrete Information übermittelt.

Weitere Attribute von *InformationUnit* sind *Status* und *ID*. *Status* vemittelt hierbei, ob die Information noch unterwegs ist, bereits beim Empfänger angekommen, oder gar verarbeitet ist. *ID* ist, ähnlich wie *Topic*, ein Bezeichner, von dem jedoch Eindeutigkeit über Zeit und Raum gefordert wird.

Konkretisiert man das Modell auf eine Email-Infrastruktur, kann man von den drei Basisklassen weitere Verfeinerungen ableiten: *Sender*, *Receiver*, *Email to System* und *Email from System*.

Die Klasse *Sender* erbt von *Source* und wird um zwei im Email-Verkehr wichtige Attribute verfeinert. *Name* ist eine deskriptive Bezeichnung von *Sender*. Handelt es sich beim Sender um eine natürliche Person, so wird dieses Attribut bei einer Instanziierung in der Regel mit dem Realnamen belegt. Gleichzeitig erhält der Sender als eindeutige Identifizierung eine Emailadresse, welche mit dem Attribut *Address* dargestellt wird.

Die Klassen *Email to System* und *Email from System* schließlich sind eine Verfeinerung der Oberklasse *InformationUnit*. Das Attribut *To* erlaubt dem Sender, ein Ziel für seine Email anzugeben, das Attribut *From* der Klasse *Email From System* zeigt dem Empfänger die Quelle der Information. Beide Email-Klassen erben das Attribut *Topic* von *InformationUnit*. Im Email-Kontext ist darunter die Subject-Zeile einer Mail zu verstehen. Außerdem erben die Emailklassen das Objekte *Status*, welches anzeigt, ob die Information noch unterwegs ist, oder bereits vermittelt werden konnte. Ebenfalls vererbt wird das Attribut *ID*, welches nun als Message-Id zu verstehen ist. Wie bereits von ID gefordert, ist eine Message-Id einmalig über Zeit und Raum und daher geeignet, eine Email zweifelsfrei zu identifizieren.

Zwischen *Sender* und *Receiver* befindet sich die Klasse *Email-System*, welche die Aufgabe hat, die Emails zwischen Sender und Empfänger zu vermitteln. Dazu dienen die beiden Methoden *receive_mail* und *send_mail*. Die Verteiltheit der Email-Infrastruktur spielt bei dieser Betrachtung gemäß der ODP-Forderung nach Transparenz noch keine Rolle (siehe Abschnitt 4.3.1).

Die aus dem Enterprise Viewpoint stammende Forderung nach Asynchronizität der Kommunikation spiegelt sich hier in der entkoppelnden Wirkung von *Email-System* wieder. Sender und Empfänger kommunizieren jeweils getrennt mit dem System, nicht jedoch direkt miteinander. Aus diesem Grund wurde die übertragene Email in zwei Klassen getrennt: Eine *Email to System* unterscheidet sich nicht nur in der Rolle, also gesendet oder empfangen, von *Email from System*, sondern kann tatsächlich verschieden sein, d.h. sogar eine andere Message-Id besitzen. Im Email-System sind Umformungen, Adreßabbildungen, Vervielfachungen und viele weitere Operationen denkbar. *Email-System* kann auch selbst als Sender auftreten, z.B. um Benachrichtigungen zu versenden.

Die verbleibenden Klassen, Methoden und Attribute der Darstellung dienen nicht der Modellierung des Dienstes selbst, sondern der Modellierung der Managementanforderungen. Auch hierbei muß die gebotene ODP-Transparenz beachtet werden: Da Verteiltheit auf dieser

Stufe noch nicht relevant ist, werden ausschließlich Managementanforderungen an den Dienst Email modelliert, das Management von Komponenten - wie etwa MTAs - spielt hier keine Rolle.

Die Klasse *Configuration* steht in einer Enthaltenseinsbeziehung zu *Email-System* und spiegelt die Konfiguration des Dienstes wieder. Das Attribut *Options* spiegelt den klassischen Bereich des Konfigurationsmanagements wieder, d.h. *Options* beinhaltet sämtliche Einstellungen, die systemweit vorgenommen werden. Die Umsetzung von Policies aus dem Enterprise Viewpoint wird durch das Attribut *Policies* abgedeckt. Eine konkretere Umsetzung von System-Policies findet sich erst bei komponentenbezogener Betrachtungsweise, die nicht Aufgabe des Informational Viewpoints ist. Das Attribut *Security* deckt den Bereich des Sicherheitsmanagements ab. Das Attribut repräsentiert alle sicherheitsrelevanten Einstellungen von Email-System.

Um das Management des Dienstes Email zu gewährleisten, stellt *Email-System* weitere Attribute und Methoden zur Verfügung.

Bei den Attributen handelt es sich um:

- *Availability*: Dieses Attribut gibt an, ob der Dienst vollständig, eingeschränkt oder gar nicht verfügbar ist.
- *Faults*: Dieses Attribut deckt das Fehlermanagement ab: Auftretende Fehler werden von Email-System mitprotokolliert und können eingesehen werden. Eine Fehlerbehebung erfolgt dann z.B. durch Änderung von Options in der Klasse *Configuration*.
- *Performance*: Dieses Attribut gibt die Leistung des Email-Systems an.
- *Statistics*: Das Email-System protokolliert vermittelte Emails und erstellt daraus für den Betreiber interessante Statistiken. Zusätzlich zu reinen Leistungsdaten, welche sich ja bereits bei Performance finden, sind z.B. Flußanalysen vorstellbar.
- *Accounting*: Die Email-Infrastruktur verfügt über die Fähigkeit, für alle Teilnehmer eine Abrechnung zu erstellen.

Die folgenden Methoden werden zur Verfügung gestellt:

- *start*/*stop*: Diese Methoden erlauben das Starten und Anhalten des Dienstes.
- *get_mail_status*: Jede Email verfügt über das Attribut Status. Das Email-System kennt den Status jeder Email und erlaubt es deshalb dessen Abfrage, d.h. es ermöglicht ein Message Tracking.

Wie sieht nun ein typischer Ablauf, also das Versenden einer Email aus? Die Objekte der Klassen *Sender* und *Receiver* müssen instanziiert werden. Dabei werden jeweils die Attribute *Name* und *Address* gesetzt. Unter der Instanziierung könnte man sich z.B. das erstmalige Anlegen der Kennungen durch einen Administrator vorstellen. Erst beim Löschen dieser Kennungen würden die Objekte wieder zerstört.

Wenn *Sender* nun eine Nachricht verfaßt, wird ein Übertragungsobjekt *Email to System* erzeugt, das *Attribut To* wird mit der Adresse des Empfängers besetzt. Wurde die Email vom System akzeptiert, so ist der Lebenszyklus von *Email to System* bereits durchlaufen, das Objekt wird zerstört. Die Information wird nun durch Objekte innerhalb von *Email-System* repräsentiert, die noch nicht sichtbar sind. Nach interner Verarbeitung erzeugt *Email-System*

nun ein Objekt *Email From System*. Dieses Objekt bleibt solange bestehen, bis es von *Receiver* abgeholt wird. Die Information befindet sich nun beim Emfpänger und das „Trägerobjekt“ *Email from System* wird zerstört.

Zusammenfassung:

Das entworfene Informationsmodell vermittelt die Grundfunktionalität einer Email-Infrastruktur ebenso, wie die Managementanforderungen an diese Infrastruktur. Die in Kapitel 2.1 vorgestellten Managementbereiche werden durch Attribute und Methoden des Modells abgedeckt.

Im Computational Viewpoint müssen nun die einzelnen Komponten einer Email-Infrastruktur mit entsprechender Managementfunktionalität herausgearbeitet werden. Die Abbildung von Klassen des Informational Viewpoints und deren Attributen und Methoden auf den Computational Viewpoint ist dabei nicht einfach und kann keineswegs automatisch geschehen.

So stößt man etwa beim Attribut *Availablity* der Klasse *Email-System* unwillkürlich auf das Problem der Definition von Verfügbarkeit. Diese Definition ist, ebenso wie die Abbildung auf Komponenten des Computational Viewpoints, Thema mehrerer Dissertationen und kann im Rahmen dieser Diplomarbeit nicht gegeben werden.

4.3.4 Computational Viewpoint

4.3.4.1 ODP-Basisklassen für den Computational Viewpoint

Zur Modellierung des Computational Viewpoints bietet ODP eine Menge von generischen Basisklassen.

Die folgende Beschreibung der Klassen orientiert sich an [Muel98]. Die Attribute der Klassen werden im folgenden Abschnitt am konkreten Beispiel Email erläutert.

- *Computational Object* (*compObject*): Komponente einer verteilten Anwendung, welche an fest definierten und typisierten Schnittstellen Dienste anbieten oder nutzen kann. Die Klasse erbt von einer noch allgemeineren Klasse object, welche lediglich über einen Identifikator und Operationen zum Erzeugen und Vernichten einer Instanz verfügt.
- *Computational Object Template* (*compObjTemplate*): umfaßt nach RM-ODP alle für die Instanziierung des Computational Objects nötigen Informationen. Unter Managementgesichtspunkten handelt es sich dabei um ein Software in einem verteilten Rechensystem. Die Instantiierung des Templates entspricht dem Starten der Software.
- *Computational Interface* (*compInterface*): Punkt, an welchem ein Computational Object seinen Dienst zur Verfügung stellt. Die Klasse erbt von der allgemeinen Oberklasse *interface* und ist Vater der drei speziellen Klassen *signalInterface*, *operationInterface* und *streamInterface*. Für Managementzwecke relevante Daten und Operationen auf einer Komponente werden i.d.R. als *operationInterface* bereitgestellt.

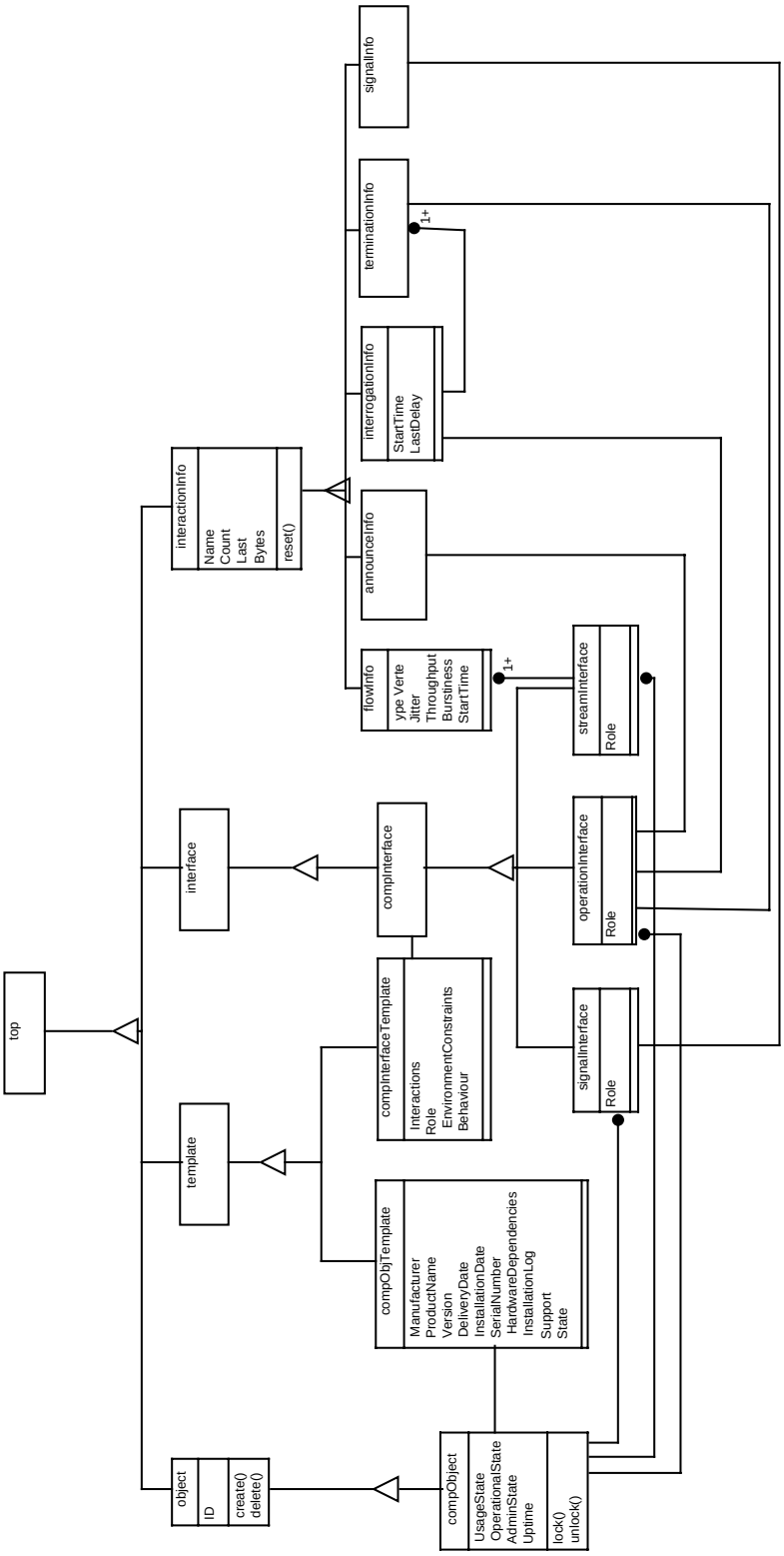


Abbildung 22 - ODP Basisklassen zum Computational Viewpoint (nach [Muel98])

- *Computational Interface Template* (*compInterfaceTemplate*): Nach RM-ODP gehört zu jedem Computational Interface genau ein zugehöriges Template. Es enthält alle nötigen Informationen, um zur Laufzeit ein Computational Object zu instanziieren. Insbesondere enthält es Signaturen der möglichen Interaktionen, die Rolle, das Verhalten und Bedingungen der Schnittstelle an die Umgebung.
- *Interaction Information* (*interactionInfo*): Treten Objekte über ihre Schnittstellen miteinander in Verbindung, so werden Informationen über diese Interaktionen in Objekten dieses Typs gespeichert. Mit diesen Informationen können die Anzahl der offenen Verbindungen, Übertragungsvolumen, Statuscodes, etc. jederzeit ermittelt werden.

4.3.4.2 Modellierung des Computational Viewpoint

Die Modellierung des Computational Viewpoint soll in mehreren Schritten geschehen.

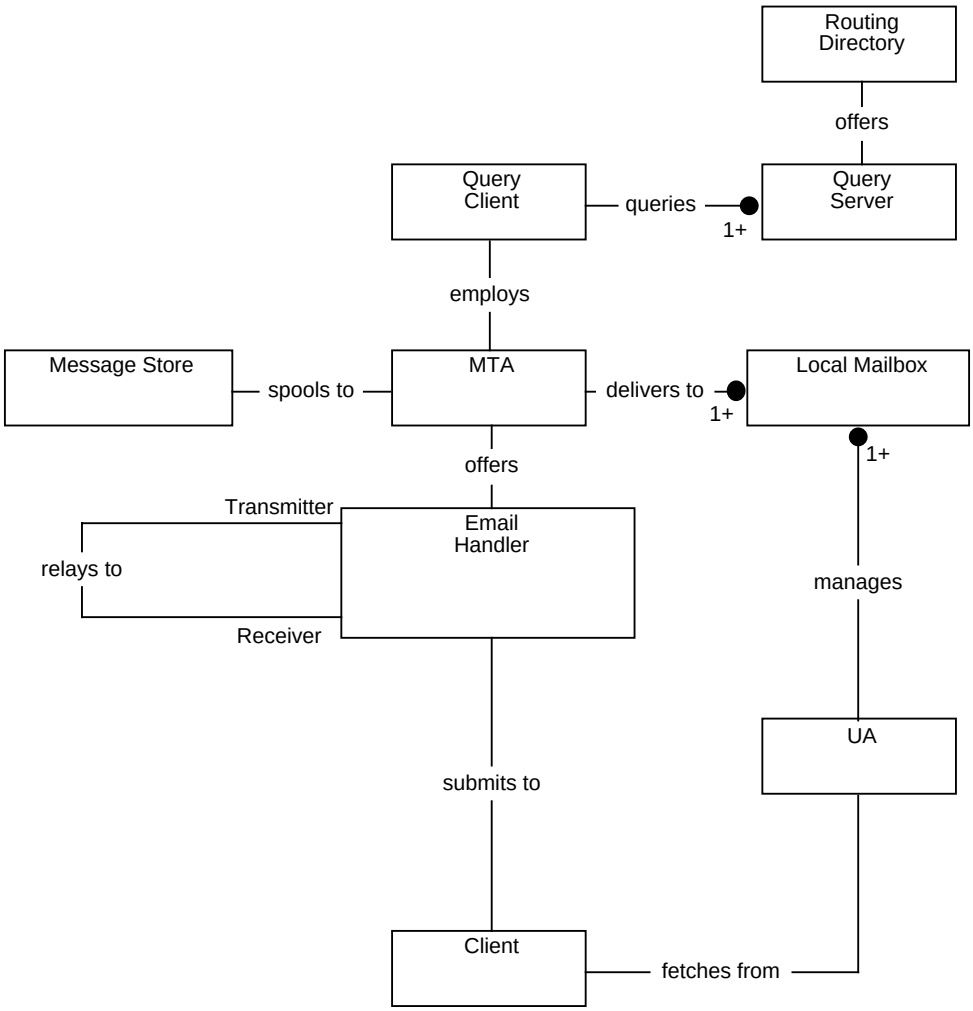


Abbildung 23 – Email-Komponenten im Computational Viewpoint

In einem ersten Schritt werden nur die aus Abschnitt 2.1.1 bekannten Komponenten eines MHS in OMT-Notation übertragen:

Dabei wird noch nicht erläutert, wie die Klassen aus den generischen ODP-Basisklassen gewonnen werden können. Auf die Modellierung des im Informational Viewpoint gezeigten Informationsfluß wurde ebenfalls noch verzichtet, d.h. es es gibt noch keine Klasse, welche eine Email abstrahiert.

Die Klassen bedürfen dabei kaum der Erklärung. *Client* sendet eine Mail an *MTA* und empfängt sie von *UA*. *UA* liest die Nachrichten aus *Local Mailbox* und verwaltet diese, d.h. die Klasse legt neue Ordner an, löscht und verschiebt Mails, etc. *MTA* speichert Nachrichten in *Message Store* zwischen oder versendet sie an andere MTA Objekte. *Routing Directory* bietet Verzeichnis- und Routinginformationen an.

Die Klassen *Query Server*, *Query Client* und *Email Handler* stellen bereits Umsetzungen der vorgestellten ODP Computational Interfaces dar, zwischen welchen Informationen fließen. So verfügt die Klasse *MTA* über ein oder mehrere Interfaces *Email Handler*, über welche Mail versendet oder empfangen wird. Je nachdem, ob *Email Handler* eine Mail ausliefert oder empfängt, tritt das Interface in der Rolle *Receiver* oder *Transmitter* auf.

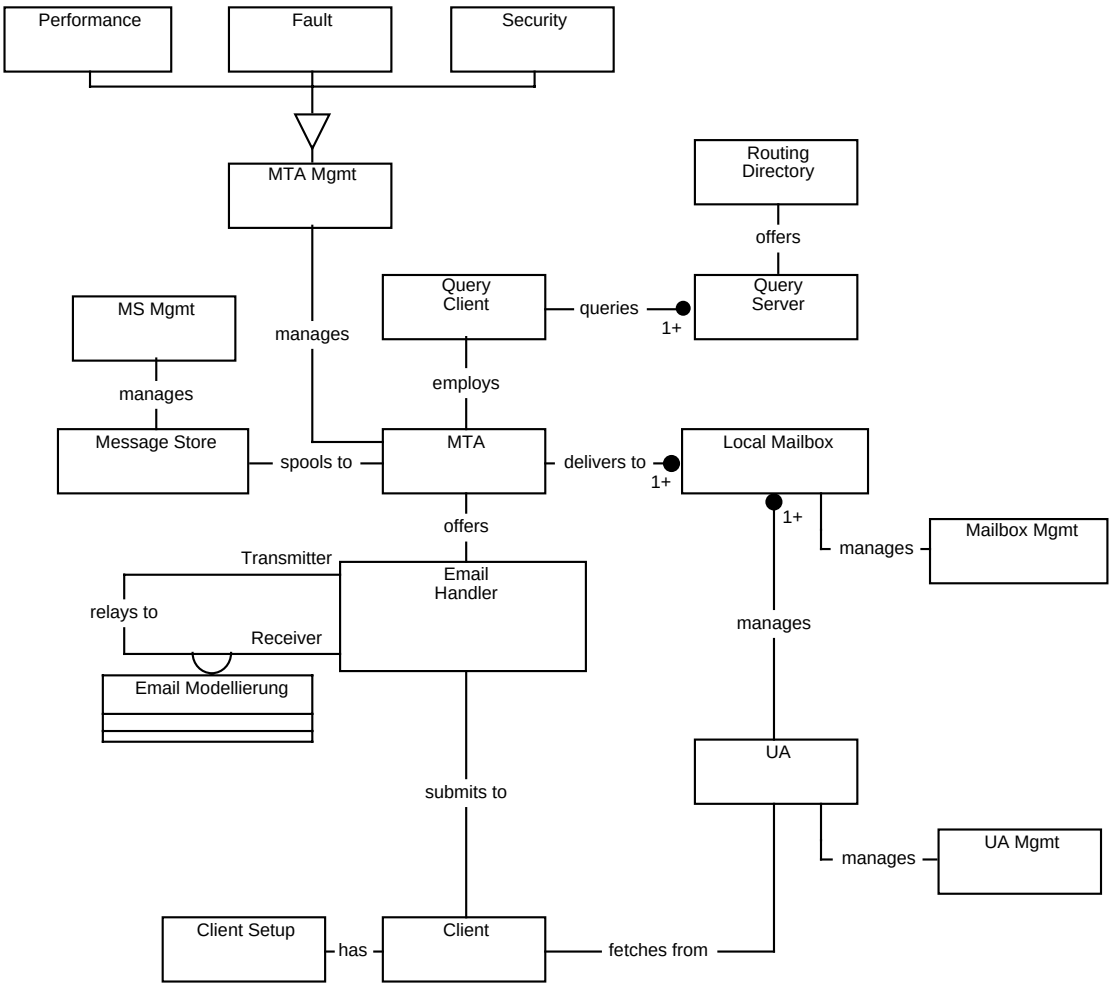


Abbildung 24 – Managementschnittstellen im Computational Viewpoint

Im nächsten Schritt müssen die Komponenten des MHS mit Managementfunktionalität ausgestattet werden.

Abbildung 24 gibt einen groben Überblick über das entstehende Modell. Jede Komponente des Mailsystems erhält ein Interface, welches Managementfunktionalität zur Verfügung stellt. Das MTA-Management ist so komplex, daß verschiedene Funktionsbereiche des Managements über unterschiedliche Schnittstellen abgedeckt werden. Da der Fluß von Emails nicht allein durch Attribute der Managementschnittstellen der Komponenten modelliert werden kann, wird die (noch abstrakte) Klasse *Email Modellierung* eingeführt.

In einem letzten Schritt wird das Modell nun komplettiert. Die durch die ODP-Basisklassen bereitgestellte Funktionalität soll genutzt werden. Ebenso muß die Modellierung einer Email konkretisiert werden. Weiterhin werden die Managementschnittstellen mit Attributen und Methoden versehen.

Alle Komponenten erben von *object* bzw *compObject*. *object* ist eine generische Basisklasse, die, wie bereits erwähnt, lediglich über eine ID, sowie einen Konstruktor und Destruktor verfügt. Die ODP-Klasse *compObject* verfügt über Attribute und Methoden, welche alle von ihr abgeleiten Computational Objects mit einer minimalen Schnittstelle für Statusmanagement versehen:

- *OperationalState*: Dieses Attribut signalisiert die Betriebsbereitschaft eines Computational Objects. Kann das Objekt seine Dienste an den Schnittstellen bereitstellen, so hat das Attribut den Wert Enabled, andernfalls Disabled. Kann der Zustand des Objekts zu einem bestimmten Zeitpunkt nicht ermittelt werden, so ist auch der Wert Unknown erlaubt. Auf das Attribut kann nur lesend zugegriffen werden.
- *UsageState*: Dieses Attribute gibt an, ob ein zur Verfügung gestellter Dienst momentan genutzt wird, ob also ein „Auftrag“ bearbeitet wird. UsageState hat den Wert Idle, wenn das Objekt gerade nicht benutzt wird, Active, wenn Dienste genutzt werden, aber noch Kapazitäten frei sind und Busy, falls keine weiteren Aufträge mehr angenommen werden können. Der Wert Unknown ist auch hier wieder möglich. Auf das Attribut kann nur lesend zugegriffen werden.
- *AdminState*: Der Administrationszustand gibt an, ob das Objekt seine Dienste anbieten darf (unlocked), oder, z.B. wegem administrativen Zugriff gesperrt (locked) ist. Der Zustand Shutting Down signalisiert, daß bereits angenommene Aufträge noch abgearbeitet werden, jedoch keine neuen Aufträge angenommen werden. Das Attribut kann über die entsprechenden Methoden lock() und unlock() modifiziert werden
- *Uptime*: Gibt die Zeit in Sekunden an, die seit der Instantiierung des Objekts verstrichen ist.
- *lock():* Setzt das Attribut *AdminState* auf *Locked* oder *Shutting Down*.
- *unlock():* Setzt das Attribut *AdminState* auf *Unlocked*.

Alle weiteren Komponenten erben direkt von compObject. Auf die Definition einer generischen Vaterklasse Server, wie in [Muel98] vorgenommen, wird hier verzichtet, da sich die verschiedenen Komponenten zu stark unterscheiden und eine generische Serverklasse, die sämtliche Managementdienste in nur einer Schnittstelle bündelt, die Verteilung der vielfältigen Funktionalitäten der Klasse MTA auf mehrere Schnittstellen erschweren würde.

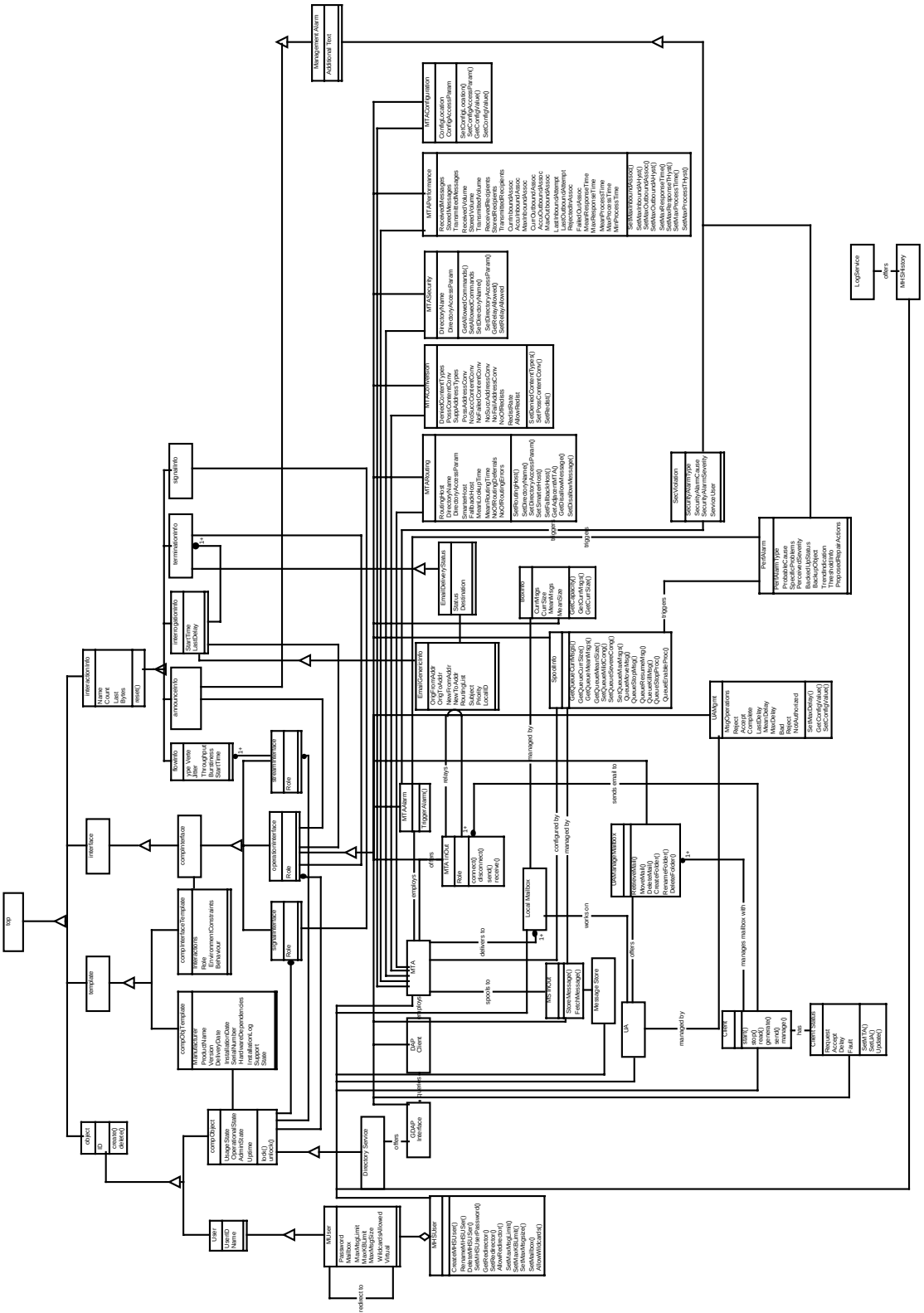


Abbildung 25 - Kompletter Computational Viewpoint der Email-Infrastruktur

Bevor die Managementschnittstellen detailliert besprochen werden, soll die weitere Vorgehensweise kurz erläutert werden.

Wie bereits in Abbildung 24 gezeigt, werden alle Komponenten mit Managementschnittstellen versehen. Auch die Nutzfunktionalität der Komponenten wird nun in Schnittstellen verlagert. Dabei ergibt sich folgende Zuordnung:

- *MTA*: Wie schon erwähnt, ist der MTA der Kern der Email-Infrastruktur. Seine Funktionalität, also das Versenden und Empfangen von Mails stellt er an seiner Schnittstelle *MTA InOut* zur Verfügung. Das Management des MTA gestaltet sich besonders besonders aufwendig und wird deshalb auf mehrere Klassen verteilt. Mit der Aufteilung in die Bereiche Routing, Conversion, Security, Performance und Configuration folgt das Modell dem ISO-Standard [X467] zum MTA-Management. Die Attribute und Methoden der MTA-Klassen wurden dabei zum Großteil direkt aus der Anforderungsanalyse erarbeitet. Einige Attribute der Klasse *MTASecurity* stammen ebenfalls aus [X467], einige Statistik-Attribute der Klasse *MTAPerformance* stammen aus [RFC2249].
- *Message Store*: Der MS ermöglicht das Zwischenspeichern von Nachrichten. Dementsprechend stellt er an einer Schnittstelle *MS InOut* die beiden Methoden *StoreMessage*() und *FetchMessage*() zur Verfügung. Das Management des MS übernimmt die Klasse *SpoolInfo*. Die Methoden der Klasse wurden selbst erarbeitet und setzen ein in [CCT96] vorgestelltes Konzept zum Load Balancing bei MTAs um.
- *Local Mailbox*: In der lokalen Mailbox findet die Speicherung von ausgelieferten Nachrichten statt. Die Attribute und Methoden der zugehörigen Managementklasse Boxinfo wurde wieder aus der Anforderungsanalyse abgeleitet.
- *Client*: Die Klasse wird über das Objekt *Client Status* überwacht. Die Attribute und Methoden resultieren aus dem Wunsch nach zentraler Fehlerüberwachung und automatischer Konfiguration und Verteilung von Client-Software.
- *Directory Service*: Der Directory Service wird nur soweit wie für die Interaktion mit der Email-Infrastruktur nötig modelliert. Hierzu bietet er Verzeichnisinformationen über eine in [Gars98] spezifizierte Schnittstelle *GDAP Interface* an. Der MTA greift die Informationen über die Schnittstelle *DAP Client* ab.
- *Muser/MHSUser*: Wie in Abschnitt 2.2.2 gezeigt, gibt es Managementaufgaben, die sich keiner bestimmten Komponente zuordnen lassen. Die Benutzerverwaltung wird deshalb über die unabhängigen Klassen *MUser* und *MHSUser* modelliert. Die Attribute der Klassen wurden wieder aus den Anforderungen erarbeitet.
- *Email*: Die Modellierung des Flusses einer Email durch das System erfolgt über die Klassen *EmailGenericInfo* und *EmailStatus*. Die Klassen mit ihren Methoden und Attributen wurden als Verfeinerungen aus dem generischen ODP Transaktions/Interaktions-Konzept erarbeitet.

Die folgenden Abschnitte stellen alle Klassen sowie ihre Attribute und Methoden vor. Das Resultat ist ein umfangreicher Katalog von Managementoperationen auf einer Email-Infrastruktur.

4.3.4.2.1 Der Message Transfer Agent (MTA)

Die Klasse *MTA* ist der Kern der Email-Infrastruktur: Ihre Komplexitität spiegelt sich in der Vielzahl der Managementschnittstellen und –operationen wieder.

Die Klasse *MTA* erbt von *compObject* und verfügt damit über deren Attribute und Operationen. Ihre Dienste stellt die Klasse *MTA* an der Schnittstelle *MTA InOut* zur Verfügung. *MTA InOut* ist Unterklasse des in 4.3.4.1 vorgestellten Operational bzw. Computational Interfaces. Die von *MTA InOut* angebotenen Funktionen *connect*(), *disconnect*(), *send*(), *receive*() erlauben das Vermitteln von Mail. Je nachdem, ob die Schnittstelle zum Senden oder Empfangen genutzt wird, bekommt das Attribut *Role* den Wert *Sending* oder *Receiving*. Nachdem *MTA* eine Mail akzeptiert hat, wird sie entweder lokal zugestellt, oder an eine weitere Instanz von *MTA* übergeben.

Um einen MTA effizient verwalten zu können, wird an den Schnittstellen weitere Funktionalität zur Verfügung gestellt. Die Aufteilung der Funktionen orientiert sich dabei an [X467]: Die weiteren Schnittstellen sind:

- *MTARouting*: Die Attribute und Methoden dieser Klasse steuern das Routing des MTA. Hierzu gehört das Festlegen der zu benutzenden Verzeichnisse, ebenso wie die Realisierung von Backupstrategien. Zusätzlich werden Performance- und Fehlerdaten zum Routing gesammelt. Eine Managementapplikation kann diese auswerten und das Routing oder den zu befragenden Verzeichnisdienst entsprechend verändern.
- *MTAConversion*: Die Klasse gibt Auskunft über bzw. steuert das Verhalten des MTA beim Konvertieren von Nachrichtenkörpern und Nachrichtenumschlägen. Weiterhin erfaßt die Klasse Daten zu Nachrichtenexplosionen, wie sie z.B. durch Verteiler ausgelöst werden.
- *MTASecurity*: Die Klasse umfaßt das Sicherheitsmanagement eines MTA. Hierzu gehört das Setzen von Optionen für die Schnittstelle MTA InOut, ebenso wie die in Kapitel 5 implementierte Zugriffskontrolle und die Möglichkeit, Sicherheitsalarme auszulösen.
- *MTAPerformance*: Die Methoden dieser Klasse spiegeln die Leistung eines MTA wieder. Darunter wird hauptsächlich der Durchsatz verstanden. Speziellere Daten, wie z.B. bei Routing oder Konvertierung auftretende Fehler, werden als Attribute direkt bei den entsprechenden Klassen modelliert. Die Klasse verfügt über die Möglichkeit, bei sinkender Performanz Alarme auszulösen.
- *MTAConfiguration*: Die Klasse bietet generische Konfigurationsoptionen, welche sich leicht auf bestehende Implementierungen abbilden lassen.
- *MTAAlarm*: Die Klasse erlaubt dem MTA beim Auftreten vorher definierter Ereignisse, Alarmobjekte zu instanziieren.

Die Benutzerverwaltung wird als Spezialfall des Konfigurationsmanagements durch eine eigene Klasse *MHSUser* abgedeckt. Alle Schnittstellen erben von *operationInterface.*

Die einzelnen Funktionsbereiche des MTA-Managements werden nun detailliert erläutert.

MTARouting

Die Klasse erlaubt, das Routingverhalten des MTAs zu prüfen und zu steuern. Dabei stehen die folgenden Attribute und Methoden zur Verfügung:

- *RoutingHost*: enthält den Namen oder die Adresse eines Rechners, von dem Informationen zum Routing einer Email abgefragt werden können. Der Zugriff ist lesend möglich, Schreibzugriff erfolgt über die entsprechende Methode.
- *DirectoryName*: Name des Verzeichnisses, aus dem Routinginformationen gewonnen werden können. Zugriff lesend, Schreibzugriff über Methode.
- *DirectoryAccessParam*: Parametereinstellungen für den Zugriff auf das Routing-Verzeichnis. Zugriff lesend, Schreibzugriff über Methode.
- *SmarterHost*: enthält den Namen einer weiteren Instanz von *MTA*, an die sämtliche ausgehenden Mails geschickt werden sollen, deren Ziel der MTA nicht selbst feststellen kann. Ist dieses Attribut gesetzt, so werden keinerlei Routinginformationen aus Verzeichnisdiensten gewonnen, d.h. die drei vorherigen Attribute werden nicht ausgewertet. Zugriff lesend, Schreibzugriff über Methode.
- *FallbackHost*: enthält den Namen einer weiteren Instanz von *MTA*, an den Mails geschickt werden, falls zu einer Zieladresse keinerlei Routinginformationen aus einem Verzeichnisdienst gewonnen werden können, oder falls der Verzeichnisdienst nicht erreicht werden kann. Zugriff lesend, Schreibzugriff über Methode.
- *MeanLookupTime*: durchschnittliche Zeit (in einem Zeitintervall t) in Sekunden, die der MTA pro Mailbearbeitungsauftrag auf die Antworten von Verzeichnisdiensten gewartet hat. Zugriff nur lesend.
- *MeanRoutingTime*: Durchschnittliche Zeit (in einem Zeitintervall t) in Sekunden, die der MTA pro Mailbearbeitungsauftrag insgesamt mit dem Routing zugebrach hat. Zugriff nur lesend.
- *NoOfRoutingDeferrals*: Anzahl der Mails (in einem Zeitintervall t), die der MTA aufgrund fehlender Routinginformationen vorübergehend nicht bearbeiten konnte. Zugriff nur lesend.
- *NoOfRoutingErrors*: Anzahl der Mails (in einem Zeitintervall t), die der MTA aufgrund fehlender oder fehlerhafter Routinginformationen (z.B. Routing Loops) endgültig nicht bearbeiten konnte. Zugriff nur lesend.
- *SetRoutingHost*(): Methode zum Setzen von *RoutingHost*.
- *SetDirectoryName*(): Methode zum Setzen von *DirectoryName*:
- *SetDirectoryAccessParam*(): Methode zum Setzen von *DirectoryAccessParam*.
- *SetSmarterHost*(): Methode zum Setzen von *SmarterHost*.
- *SetFallbackHost*(): Methode zum Setzen von *FallbackHost*.
- *GetAdjacentMTA*(): Diese Methode erlaubt das Austesten von MTA-Einstellungen: Sie erwartet als Parameter eine Email-Zieladresse und liefert als Ergebnis entweder Local oder Instanz-ID des Ziel-MTAs. Die Methode stellt damit die Basisfunktionalität eines Topologiedienstes bereit.
- *GetDisallowUser*(): Liefert eine Liste von Absenderadressen, deren Emails der MTA prinzipiell nicht akzeptiert.

- *SetDisallowUser*(): Erlaubt das Setzen oder Löschen von Absendern, deren Mails der MTA nicht akzeptiert. Diese Funktionalität kann z.B. zum Filtern ungewünschter Mails genutzt werden.

MTAConversion

Die Klasse erlaubt, das Verhalten des MTAs bei anfallenden Konvertierungen zu prüfen und zu steuern. Unter den Begriff Konvertierung fallen dabei sowohl Umsetzungen am Nachrichtenkörper (wie etwa Wandlungen des Zeichensatzes), als auch am Nachrichtenumschlag (wie etwa Wandlungen von Adressen). Dabei steht die folgende Funktionalität zur Verfügung:

- *DeniedContentTypes*: Mailinhaltstypen (*Content-Types*), deren Übertragung der MTA nicht akzeptiert. Zugriff lesend, Schreibzugriff über Methode.
- *PossContentConv*: Mögliche Inhaltskonvertierungen (z.B. IASText nach TTX oder 8BITMIME nach 7BITQP), die der MTA unterstützt. Zugriff lesend, Schreibzugriff über Methode.
- *SuppAddressTypes*: Adreßtypen (wie z.B. user@sub.do.main oder path!host!user), deren Syntax der MTA auswerten kann. Zugriff lesend.
- *PossAddressConv*: Mögliche Umsetzungen, die ein MTA zwischen verschiedenen Adreßtypen vornehmen kann. Zugriff nur lesend.
- *NoSuccContentConv*: Anzahl der Inhaltskonvertierungen, die der MTA in einem Zeitintervall t vorgenommen hat. Zugriff nur lesend.
- *NoFailContentConv*: Anzahl der Inhaltskonvertierungen, die in einem Zeitraum t fehlgeschlagen sind. Zugriff nur lesend.
- *NoSuccAddressConv*: Anzahl der Adreßumsetzungen, die der MTA in einem Zeitraum t vorgenommen hat. Zugriff nur lesend. Zugriff nur lesend.
- *NoFailAddressConv*: Anzahl der Adreßumsetzungen, die in einem Zeitraum t fehlgeschlagen sind. Zugriff nur lesend.
- *NoOfRedists*: Anzahl der Adreßumsetzungen, die der MTA in einem Zeitraum t vorgenommen hat, und die das Erzeugen von neuen Mails zur Folge hatten (Verteilerlistenfunktion bzw. *Mailing List Explosion*). Zugriff nur lesend.
- *RedistRate*: Durchschnittliche Verhältnis von Anzahl der Mails, welche durch eine Redistribution erzeugt wurden zur Anzahl der Mails, die diese Redistribution auslösten. Ermittelt über einen Zeitraum t, Zugriff nur lesend.
- *AllowRedist*: Gibt an, ob die Redistributionsfunktion des MTA aktiviert ist, oder nicht. Zugriff lesend, Schreibzugriff über Methode.
- *SetDeniedContentTypes*(): Methode zum Setzen von *ContentTypes*.
- *SetPossContentConv*(): Methode zum Setzen von *ContentConv*.
- *SetRedist*(): Methode zum Setzen von *AllowRedist*.

MTASecurity

Die Klasse ermöglicht das Sicherheitsmanagement eines MTA. Darunter fällt zum einen das Setzen von sicherheitsrelevanten Konfigurationsoptionen, wie etwa dem Erlauben oder Verbieten sicherheitskritischer Kommandos.

Das Sicherheitsmanagement für Email sollte auch Bereiche der Authentifizierung, Zertifizierung und Verschlüsselung von Nachrichten umfassen. Allerdings ist das Management dieses Bereichs so umfangreich, daß es ein eigenes ODP-Modell erfordern würde. Die Klasse *MTASecurity* deckt diese Funktionen deshalb nur rudimentär ab:

- *DirectoryName*: Name des Verzeichnisses, aus dem Directory-Inforrmationen bzgl. Authentifizierungs- oder Zertifizierungsinformationen gelesen werden können. Zugriff lesend, schreibend über Methode.
- *DirectoryAccessParam*: Parametereinstellungen für den Zugriff auf das Routing-Verzeichnis. Zugriff lesend, Schreibzugriff über Methode.
- *GetAllowedCommands*(): Liefert eine Liste aller Operationen, welche im Protokoll an der Schnittstelle *MTA InOut* erlaubt sind. Das SMTP-Protokoll eines Internet-MTAs enthält z.B. Befehle, mit denen sich die Benutzer dieses MTAs auflisten lassen. Mit dieser und der nächsten Methode lassen sich solche Befehle abschalten.
- *SetAllowedCommands*(): Erlaubt das Setzen oder Löschen von Operationen, die an der Schnittstelle *MTA InOut* erlaubt sind.
- *SetDirectoryName*(): Methode zum Setzen von DirectoryName.
- *SetDirectoryAccessParam*(): Methode zum Setzen von DirectoryAccessParam.
- *GetRelayAllowed*(): Liefert für die aktuelle Instanz der Schnittstelle *MTA InOut* eine Liste aller weiterer Instanzen der Objekte *MTA InOut* und Client, welche die Schnittstelle benutzen dürfen.
- *SetRelayAllowed*(): Fügt der Zugriffliste von *MTA InOut* Objekt-IDs, welche die Schnittstelle benutzen dürfen hinzu bzw. entfernt diese wieder. Dadurch kann eine Zugriffskontrolle für die Schnittstelle *MTA InOut* realisiert werden.

SecViolation

Ein weiterer, wichtiger Teil des Sicherheitsmanagements ist das Erkennen und Melden von Sicherheitsalarmen. Zu diesem Zweck wird der MTA mit der Möglichkeit ausgestattet, Benachrichtigungen bzw. Alarme an eine Managementanwendung senden zu können. Diese Funktionalität wird durch die MTA-Schnittstelle *MTAAlarm* mit ihrer Methode *TriggerAlarm*() implementiert. Beim Auftreten eines Sicherheitsalarms wird ein Objekt der Klasse *SecViolation* als Interaktionsinformation zwischen dem MTA und einem Manager instanziiert. Weil eine Vielzahl von weiteren Alarmen denkbar ist, wird *SecViolation* als Spezialisierung der allgemeinen Klasse *ManagementAlarm* definiert. Da die Meldung eines Alarms an den Manager asynchron und ohne Rückmeldung erfolgt, erbt *ManagementAlarm* von der Klasse *announceInfo*, welche wiederum Unterklasse von *interactionInfo* ist. Die Managementanwendung selbst wird in der Darstellung nicht gezeigt. Die Klasse *SecViolation* definiert die folgenden Ausnahmezustände, welche sich an der X.721 Empfehlung der ITU-T [X721] orientieren.

- *Integrity Violation*: Eine Integritätsverletzung tritt dann auf, wenn eine Unterbrechung im Informationsfluß festgestellt wurde und Grund zur Annahme besteht, daß Informationen illegal eingefügt, gelöscht oder verändert werden. Im Email-Bereich subsumiert der Begriff „Information“ dabei sowohl die ausgetauschten Emails selbst, als auch Metainformationen, wie z.B. Routingdaten, die ein MTA einholt. Eine Integritätsverletzung wäre also sowohl bei einem gefälschten Umschlag einer Email, als auch bei gefälschten Routing-Informationen (z.B. durch *DNS-Spoofing*), welche eine Mailumleitung an einen illegalen MTA veranlaßt, gegeben.
- *Physical Violation*: Eine „physische Verletzung“ ist dann gegeben, wenn eine physische Ressource beschädigt wird. Darunter fallen Schäden an der Hardware ebenso, wie z.B. das Ausstecken eines Kabels.
- *Operational Violation*: Dieser Alarm zeigt an, daß ein eingehender Request überhaupt nicht bearbeitet werden konnte. Gründe dafür können völlige Nichtverfügbarkeit, oder Fehlfunktionen von Schnittstellen sein. Ein Beispiel hierfür wäre z.B. das Überfluten von *MTA InOut* mit unsinnigen Daten (*Denial-Of-Service Attacke*).
- *Time Domain Violation*: Dieser Alarm wird durch Ereignisse ausgelöst, die zu einem unerwarteten Zeitpunkt auftreten oder zu diesem Zeitpunkt generell verbotenen sind. Darunter fallen also stark verzögert erbrachte Informationen, oder der Ablauf eines Zertifikats.
- *Security Service Violation*: Dieser Alarm wird durch Angriffe, welche von Sicherheitseinrichtungen erkannt wurden, ausgelöst. Darunter fallen etwa Authentifizierungsfehler, Vertraulichkeitsbrüche, Zugriffsfehler, etc.

Die Klasse *SecViolation* enthält also, in Anlehnung an [X736] die folgenden Attribute:

- *SecurityAlarmType*: Integrity, Physical, Operational, Time Domain oder Security Service.
- *SecurityAlarmCause*: Grund des Alarms, also z.B. Denial-of-Service, Out-of-Hours-Activity, Intrusion Detection, Mail Forged, etc.
- *SecurityAlarmSeverity*: Warning, Minor, Major, Critical und Indeterminate (Auswirkung des Alarms auf das System ist unbekannt).
- *ServiceUser*: Der Benutzer oder das System (falls bekannt), welches den Alarm auslöste.
- *AdditionalText*: Weitere Informationen im Klartext, die den Alarm genauer erläutern.
- *CorrelatedNotification*: Menge von Instanz-IDs weiterer Alarm-Objekts, welche mit der aktuellen Instanz zusammenhängen. Kann vom Manger evtl. zur Eventkorrelierung genutzt werden.

MTAPerformance

Die Klasse *MTAPerformance* implementiert das Leistungsmanagement für den MTA. Bei den Attributen dieser Klasse handelt es sich zum einen um statistische Werte, aus denen sich Rückschlüsse über die Performance des Systems ziehen lassen. Zum anderen handelt es sich um Schwellwerte, wie etwa die maximale Verweilzeit einer Mail bei einem MTA. Beim Überschreiten dieser Schwellwerte wird ein Alarm ausgelöst. Wie bereits bei *MTASecurity*

gesehen, wird ein Alarm als Verfeinerung von *ManagementAlarm* realisiert. In der OMT-Darstellung ist der Alarm für das Leistungsmanagement durch die Klasse *PerfAlarm* implementiert. *PerfAlarm* kommt auch dann dann zum Einsatz, wenn kritische Operationen des MTA fehlgeschlagen sind.

Die Klasse *MTAPerformance* verfügt über die folgenden Attribute:

- *ReceivedMessages*: Die Anzahl der Emails, die der MTA in einem Zeitintervall t an seiner Schnittstelle *MTA InOut* entgegen genommen hat. Zugriff nur lesend.
- *StoredMessages*: Die Anzahl der Emails, die der MTA gegenwärtig in seinem Message Store gespeichert hat. Zugriff nur lesend.
- *TransmittedMessages*: Die Anzahl der Emails, die der MTA in einem Zeitintervall t lokal ausgeliefert, oder an weitere MTAs vermittelt hat. Zugriff nur lesend.
- *ReceivedVolume*: Das Volumen in kB, welches der MTA in einem Zeitintervall t an seiner Schnittstelle *MTA InOut* empfangen hat. Zugriff nur lesend.
- *StoredVolume*: Das Volumen in kB, welches der MTA zur Bearbeitung gegenwärtig in seinem Message Store gespeichert hält. Zugriff nur lesend.
- *TransmittedVolume*: Das Volumen in kB, welches der MTA in einem Zeitintervall t lokal ausgeliefert, oder an weitere MTAs vermittelt hat. Zugriff nur lesend.
- *ReceivedRecipients*: Anzahl der Empfänger, für die der MTA in einem Zeitintervall t Emails entgegengenommen hat. Zugriff nur lesend.
- *StoredRecipients* Anzahl der Empfänger, für die der MTA gegenwärtig Mails zur Bearbeitung gespeichert hat. Zugriff nur lesend.
- *TransmittedRecipients*: Anzahl der Empfänger, an die der MTA in einem Zeitintervall t Mails ausgeliefert hat. Zugriff nur lesend.
- *CurrInboundAssoc*: Anzahl der derzeit offenen Verbindungen für eingehende Mails, d.h. Anzahl der an *MTA InOut* anliegenden Requests. Zugriff nur lesend.
- *AccuInboundAssoc*: Akkumulierte Anzahl der eingehenden Verbindungen über einem Zeitraum t. Zugriff nur lesend.
- *MaxInboundAssoc* Maximale Anzahl von eingehenden Verbindungen. Ist das Maximum erreicht, so werden weitere Requests abgewiesen und ein Alarm ausgelöst. Zugriff lesend, schreibend über Methode.
- *CurrOutboundAssoc:* Anzahl der derzeit offenen Verbindungen für ausgehende Mails. Zugriff nur lesend.
- *AccuOutboundAssoc*: Akkumulierte Anzahl der ausgehenden Verbindungen über einen Zeitraum t. Zugriff nur lesend.
- *MaxOutboundAssoc*: Maximale Anzahl der gleichzeitig geöffneten, ausgehenden Verbindungen. Ist das Maximum erreicht, werden keine weiteren Mails ausgeliefert und ein Alarm ausgelöst. Zugriff lesend, schreibend über Methode.
- *LastInboundAttempt*: Zeitpunkt des letzten eingehenden Requests. Zugriff nur lesend.
- *LastOutboundAttempt*: Zeitpunkt des letzten ausgehenden Requests. Zugriff nur lesend.
- *RejectedInAssoc*: Anzahl der, in einem Zeitintervall t, abgewiesenen, eingehenden Requests. Zugriff nur lesend.

- *FailedOutAssoc*: Anzahl der, in einem Zeitintervall t, abgewiesenen, ausgehenden Requests. Zugriff nur lesend.
- *MeanResponseTime*: Durchschnittliche Zeit in Millisekunden, in einem Zeitintervall t, die zwischen dem Eingehen und der Annahme eines Requests vergeht. Zugriff nur lesend.
- *MaxResponseTime*: Maximale Zeit in Millisekunden, die zwischen einem eingehenden Request und dessen Annahme vergehen darf. Bei Überschreitung wird ein Alarm ausgelöst. Zugriff lesend, schreibend über Methode.
- *MeanProcessTime*: Durchschnittliche Zeit über einem Zeitraum t in Sekunden, welche die Abarbeitung eines Requests, also der Zeit von der Requestannahme bis zur Auslieferung oder Weiterleitung der Email, gedauert hat. Zugriff nur lesend.
- *MaxProcessTime*: Maximale Zeit in Sekunden, welche die Abarbeitung eines Requests in Anspruch nehmen darf. Bei Überschreitung wird ein Alarm ausgelöst. Zugriff lesend, Schreiben über Methode.
- *SetMaxInboundAssoc*(): Methode zum Setzen von MaxInboundAssoc.
- *SetMaxInboundAHys*(): Methode zum Setzen eines Hysteresewertes für Alarme aufgrund der Überschreitung von *MaxInboundAssoc*. Um nach Überschreiten des Alarms einen Eventsturm zu vermeiden, wird eine neuer Alarm wegen Überschreitung erst nach einer Verzögerungszeit ausgelöst. Der Hysteresewert gibt diese Verzögerungszeit in Sekunden an.
- *SetMaxOutboundAssoc*(): Methode zum Setzen von *MaxOutboundAssoc*.
- *SetMaxOutboundAHys*():Methode zum Setzen eines Hysteresewertes für Alarme aufgrund Überschreitung von *MaxOutboundAssoc*.
- *SetMaxResponseTime*(): Methode zum Setzen von *MaxResponseTime*.
- *SetMaxResonseTHys*(): Methode zum Setzen eines Hysteresewertes für Alarme aufgrund Überschreitung von *MaxResponseTime*.
- *SetMaxProcessTime*(): Methode zum Setzen von *MaxProcessTime*.
- *SetMaxProcessTHys*(): Methode zum Setzen eines Hysteresewertes für Alarme aufgrund Überschreitung von *MaxProcessTime*.

PerfAlarm

Die Klasse ist Unterklasse von *ManagementAlarm*, welche von *announceInfo* bzw. *interactionInfo* erbt. Die folgenden Alarme werden definiert:

- *Processing Error Alarm*: Dieser Alarm indiziert, daß beim Bearbeiten eines Auftrags ein Fehler aufgetreten ist.
- *Quality Of Service Alarm*: Dieser Alarm zeigt an, daß eine zuvor definierte QoS-Anforderung nicht erreicht wurde (Schwellerwert unter- oder überschritten).
- *Communication Alarm*: Dieser Alarm tritt auf, wenn die Kommunikation mit einem weiteren Objekt fehlgeschlagen ist. Im Email-Modell gehören dazu vor allem weitere Instanzen von MTAs, sowie des Directory Service.

In Anlehnung an die in ITU-T Empfehlung [X733] definierten Alarme enthält die Klasse PerfAlarm also die folgenden Attribute:

- *PerfAlarmType*: wie oben definiert, *Processing Error, Quality of Service* oder *Communication.*
- *ProbableCause*: Möglicher Grund für den Alarm. Beispiele sind hier beispielsweise *Call Establishment Error* oder *Communications Protocol Error* bei einem Communication Alarm, oder *Corrupt Data* (Mail mit fehlerhaftem Inhalt) oder *Out of Memory* bei einem Processing Error Alarm oder *Congestion* (Kapazitätsgrenze erreicht, siehe 4.3.4.2.2) bzw. *Retransmittion Rate Excessive* bei einem QoS Alarm.
- *SpecificProblems*: Falls eine genauere Fehlerbeschreibung als oben möglich ist, so wird mit diesem Attribut ein Fehlercode (als Menge von Integerzahlen) signalisiert.
- *PerceivedSeverity*: die Schwere des aufgetretenen Fehlers, in verschiedenen Graden. Ist der Alarm vom Typ Quality of Service, so ist der Fehlergrad immer Major.
    - *Cleared*: Zeigt an, daß der Fehler behoben wurde. Das Instanziieren eines Objektes mit Cleared Status invalidiert alle Objekte, die mit identischen PerfAlarmType, ProbableCause und SpecificProblems instanziiert wurden.
    - *Indeterminate*: Zeigt an, daß die Schwere des Fehlers nicht bestimmt werden kann.
    - *Warning*: Dieser Grad zeigt an, daß ein aufgetretener Fehler die Dienstgüte eventuell beeinflussen könnte, aber noch kein Schaden entstanden ist. Bei wiederholtem Auftreten sind evtl. weitere Untersuchungen nötig.
    - *Minor*: Dieser Grad zeigt an, daß ein Fehler aufgetreten ist, der die Dienstgüte potentiell negativ beeinflussen könnte. Um dies zu verhindern, sollte der Fehler beseitigt werden.
    - *Major*: Es ist ein Fehler aufgetreten, der die Dienstgüte negativ beeinflußt. Eine Fehlerbehebung ist dringend erforderlich.
    - *Critical*: Dieser Grad zeit einen Fehler an, der die Dienstgüte erheblich beeinflußt, bzw. für einen Totalausfall verantwortlich ist. Der Fehler muß sofort behoben werden.
- *BackedUpStatus*: Dieser Parameter gibt an, ob das ausgefallene Objekt durch ein Backup-Objekt ersetzt werden konnte, so daß der Dienst nicht negativ beeinflußt wurde. Beispiele hierfür sind z.B. das Umschalten auf einen Ersatzverzeichnisdienst oder das Vermitteln von Mail an einen Fallback-Host.
- *BackupObject*: Falls gesetzt, gibt dieser Parameter den Namen des Backup-Objektes an.
- *TrendIndication*: Falls gesetzt, gibt dieses Attribute bei sich wiederholendem Fehler oder dauerhaftem Über- bzw. Unterschreiten eines Schwellwertes eine Trendinfo an. Mögliche Werte sind More Severe, No Change, Less Severe.
- *ThresholdInfo*: Dieses Attribut wird im Falle der Schwellwertverletzung gesetzt. Es besteht aus 4 Unterparametern:
    - *Triggered Threshold*: Attributname des Schwellwerts aus *MTAPerformance*, der über- oder unterschritten wurde.
    - *Threshold Level*: In *MTAPerformance* gesetzter Schwellwert.

  - *Observed Value*: Tatsächlich gemessener Wert.
  - *Arm Time*: Der Zeitpunkt, zu dem ein Alarm zuletzt „scharf gemacht“ wurde. Falls der Alarm bereits aufgetreten und der Schwellwert dauheraft überschritten ist, also genau die Zeit, die bei den Hysteresewerten in *MTAPerformance* definiert wurde.
- *ProposedRepairActions*: Falls dieses Attribut gesetzt wurde, gibt es mögliche Problemlösungen an. Diese sind entweder als Menge von Integerzahlen oder als Name eines ODP-Objektes, welches instanziiert werden soll, kodiert.
- *AdditionalText*: Weitere Informationen im Klartext, die den Alarm genauer erläutern.
- *CorrelatedNotification*: Menge von Instanz-IDs weiterer Alarm-Objekts, welche mit der aktuellen Instanz zusammengängen. Kann vom Manger evtl. zur Eventkorrelierung genutzt werden.

MTAConfiguration

Objekte dieser Klasse decken die Konfigurationsarbeit ab, die durch keine andere Klasse zum MTA-Management erfaßt werden. Entsprechend generisch sind an dieser Stelle die Operationen. Ein Objekt der Klasse MTAConfiguration virtualisiert eine Menge von Optionen, die durch einen Namen eindeutig charakterisiert sind, und auf welche lesend und/oder schreibend zugegriffen werden kann.

Während die vorhergehenden Klassen Konfigurationsoptionen nach funktionalen Managementaspekten zusammengefaßt haben, werden die Objekte dieser Klasse nach physischen Gesichtspunkten gruppiert, d.h. ein Objekt erlaubt immer die Manipulation von Optionen, welche zusammen gespeichert sind. Dabei spielt es keine Rolle, ob die Optionen in einer Textdatei, einer binären Registrierung oder einem zentralen Konfigurationsverzeichnis gespeichert sind.

Die Klasse *MTAConfiguration* bietet also die folgenden Attribute und Methoden:

- *ConfigLocation*: Pfad und Name des Konfigurationsfiles, bzw. Teilast des Registrierungsbaums bzw. Name des Verzeichnisses. Zugriff lesend, schreiben über Methode.
- *ConfigAccessParam*: Parameter zum Zugriff auf Konfiguration, also z.B. Benutzername, unter welchem der Zugriff erfolgt, oder Parameter zum Zugriff auf den Verzeichnisdienst. Zugriff lesend, Schreiben über Methode.
- *SetConfigLocation*(): Methode zum Setzen von *ConfigLocation*.
- *SetConfigAccessParam*(): Methode zum Setzen von *ConfigAccessParam*.
- *GetConfigValue*(): Liest einen Konfigurationsparameter aus und liefert den Wert im Ergebnis.
- *SetConfigValue*(): Setzt einen Konfigurationsparameter auf einen bestimmten Wert.

4.3.4.2.2 Die lokale Mailbox und der Message Store (MS)

Lokale Mailbox und Message Store dienen dem Speichern von Emails. Der Unterschied zwischen den beiden Komponenten liegt darin, daß der Message Store ein Zwischenspeicher ist, der mit dem MTA mittels seiner Schnittstelle *MS InOut* kommuniziert, während die lokale Mailbox das endgültige Ziel einer Mail darstellt und vom UA verwaltet wird.

Der Message Store eines MTA ist aus ein oder mehreren internen Warteschlangen (*Queues*) aufgebaut. Dabei kann zwischen *Inbound Queues, Outbound Queues* und *Delivery Queues* unterschieden werden. Die Inbound Queue enthält, von Clients, oder anderen MTAs akzeptierte Nachrichten, die vom MTA noch bearbeitet werden müssen. Die Outbound Queue beinhaltet vom MTA bearbeitete Nachrichten, die nun an einen anderen MTA vermittelt werden müssen. Die Delivery Queue enthält schließlich all die Nachrichten, die vom MTA bearbeitet und lokal sind, also in die lokale Mailbox geschrieben werden müssen.

Für das im folgenden beschriebende *Load Balancing* [CCT96], werden die Inbound- und Outbound Queues können nochmals unterteilt. So lassen sich mehrere Inbound Queues nach Priorität der Nachrichten einrichten, d.h. Nachrichten mit hoher Priorität werden in eine Queue einsortiert, welche der MTA bevorzugt abarbeitet. Die Outbound Queues werden nach Empfänger-MTA kategorisiert.

Das Management des Message Store dient nun vor allem dem Leistungsmanagement. Zum einen stellt der MS an einer Managementschnittstelle die üblichen statistischen Daten zur Verfügung, welche Einblick in die Füllung des MS und damit die Auslastung des MTA geben. Zum anderen lassen sich Mails ganz gezielt von einer Queue in eine andere verschieben. Durch Verschieben von Mails in Input Queues mit niedrigerer Priorität wird der lokale MTA selbst entlastet. Durch Abarbeitungsstop einer Outbound Queue, oder Verschieben von Mails aus dieser Queue in eine andere, kann ein entfernter, überlasteter MTA wieder entlastet werden.

Wie kann aber nun ein MTA den anderen MTAs der Email-Infrastruktur eine Überlast bekanntgeben? Zwei Möglichkeiten kommen in Frage: Das Mailprotokoll, welches zwischen den MTAs gefahren wird, läßt eine Übermittlung von Lastzuständen zu. In diesem Fall könnte der überlastete MTA allen benachbarten MTAs über die Schnittstelle *MTA InOut* kontaktieren. Die zweite Möglichkeit wäre eine Meldung der Last an einen Email-Manager, welcher die Topologie der Infrastruktur kennt und die Queues der umgebenden MTAs anpaßt.

Da die erste Lösung die Trennung zwischen Nutz- und Managementinformationen verwischt und außerdem den bereits überlasteten MTA zum Aufbau mehrerer Verbindungen zwingt, wird die zweite Lösung gewählt.

Zur Zustandsüberwachung der Queues und zur Definition von Überlasten (*Congestion*) wird die Klasse *SpoolInfo* als Unterklasse von *operationInterface* eingeführt. Die Meldungen bei Überlast werden wieder durch Instanziierung von *PerfAlarm* als Interaktion zum Manager realisiert. Dabei unterscheiden sich die, durch *MTAPerformance* und die durch *SpoolInfo* ausgelösten Alarme durchaus. *SpoolInfo* löst Alarme als Mitteilung an einen Manager aus, damit dieser automatisch eine bessere Lastenverteilung zwischen den MTAs vornehmen kann. Das System ist zu diesem Zeitpunkt in seiner Performance noch nicht behindert. Von *MTAPerformcance* ausgelöste Alarme signalisieren schwerwiegendere Performanceprobleme (z.B. durch Versagen des Load Balancing), welche in der Regel nicht mehr automatisch bearbeitet werden können, sondern von einem menschlichen Operateur behoben werden müssen.

Die Klasse *SpoolInfo* verfügt über die folgenden Methoden:

- *GetQueueCurrMsgs*(): Liefert zu einer bestimmten Inbound- oder Outbound Queue die Anzahl der derzeit gespeicherten Nachrichten.
- *GetQueueCurrSize*(): Liefert das, in einer bestimmten Inbound- oder Outbound Queue gespeicherte Nachrichtenvolumen in kB.
- *GetQueueMeanMsgs*(): Liefert über einem Zeitintervall t die durchschnittliche Anzahl von Nachrichten, welche in einer bestimmten Queue gespeichert sind.
- *GetQueueMeanSize*(): Liefert über einem Zeitintervall t die durchschnittliche Anzahl von Nachrichten, welche in einer bestimmten Queue gespeichert sind.
- *SetQueueMildCong*(): Setzt die Nachrichtenanzahl für eine Queue, ab der eine erste Warnung erzeugt wird. Beim Erreichen dieses Schwellwerts instanziiert *SpoolInfo* ein Objekt der Klasse *PerfAlarm*. *PerfAlarmType* wird dabei auf *Quality of Service* gesetzt, *Probable Cause* auf *Congestion* und *Perceived Severity* auf *Minor*.
- *SetQueueSevereCong*(): Setzt die Nachrichtenanzahl für eine Queue, ab der der MTA eine erhebliche Überlast signalisiert. Dazu wird wie oben ein *PerfAlarm* Objekt erzeugt, *Perceived Severity* jedoch auf *Major* gesetzt.
- *SetQueueMaxMsgs*(): Definiert die maximale Größe einer Queue. Beim Erreichen dieses Schwellwertes akzeptiert der MTA keine weiteren eingehenden Verbindungen.
- *QueueMoveMsg*(): Verschiebt eine Email von einer Queue in eine andere. Wird diese Methode auf eine Eingangsqueue angewandt, so kann sich der MTA vorübergehend selbst entlasten: Durch das Verschieben von Mails in eine Eingangsqueue mit niedriger Priorität bleibt dem MTA noch Kapazität für das Bearbeiten von Mails mit hoher Priorität. Die „zur Seite gelegten“ Mails werden zu einem späteren Zeitpunkt mit geringerer Last bearbeitet. Das Anwenden der Methode auf eine Ausgangsqueue entspricht einem Rerouting einer Mail und dient dem Überbrücken von ausgefallenen, oder stark überlasteten MTAs.
- *QueueStopMsg*(): Teilt dem MTA mit, eine Nachricht in einer Queue zurückzuhalten und nicht auszuliefern.
- *QueueResumeMsg*(): Teilt dem MTA mit, eine zuvor gestoppte Nachricht auszuliefern.
- *QueueKillMsg*(): Teilt dem MTA mit, eine sich in der Queue befindliche Nachricht zu löschen.
- *QueueStopProc*(): Teilt dem MTA mit, eine bestimmte Queue vorübergehend gar nicht mehr zu bearbeiten.
- *QueueEnableProc*(): Teilt dem MTA mit, die Bearbeitung einer zurvor stillgelegten Queue wiederaufzunehmen.

Stellt ein MTA fest, daß eine Mail für einen Benutzer des lokalen Systems bestimmt ist, so liefert er sie lokal aus, d.h. er schreibt sie in die Mailbox dieses Benutzers. Die Bearbeitung einer Mail ist für den MTA zu diesem Zeitpunkt erfolgreich abgeschlossen, die Mailbox selbst wird vom UA verwaltet.

Die für das Management relevanten Informationen einer Mailbox beschränken sich auf Ort der Lagerung, Füllgrad, sowie dem Setzen von Quota für den Benutzer. Hierzu wird die

Klasse *BoxInfo* als Unterklasse von *operationInfo* eingeführt. Pro lokalem System wird nur eine Instanz von *BoxInfo* erzeugt, auch wenn dort mehrere Benutzermailboxen verwaltet werden.

Da das Setzen von Maillimits bereits durch *MHSUser* (siehe 4.3.4.2.6) abgedeckt wird, verfügt die Klasse lediglich über folgende Attribute:

- *CurrMsgs*: Anzahl der Nachrichten, die zur Zeit in allen Benutzermailboxen des Systems gespeichert sind. Zugriff lesend.
- *CurrSize*: Gesamtgröße aller Nachrichten, die zur Zeit in allen Benutzermailboxen des Systems gespeichert sind. Zugriff lesend.
- *MeanMsgs*: Durchschnittliche Zahl von Nachrichten über einem Zeitintervall t, welche in allen Maiboxen gespeichert sind.
- *MeanSize*: Durschnittliche Gesamtgröße aller Nachrichten, die in einem Zeitintervall t in allen Benutzermailboxen des Systems gespeichert sind. Zugriff lesend.
- *GetCapacity*(): Liefert die Auslastung der gesamten Mailboxkapazität eines Systems in Prozent.
- *GetCurrMsgs*(): Liefert die aktuelle Anzahl der Nachrichten eines bestimmten Benutzers.
- *GetCurrSize*(): Liefert die aktuelle Größe aller Nachrichten eines Benutzers.

4.3.4.2.3 Der User Agent (UA)

Wie bereits in Abschnitt 2.1.1 erwähnt, wurde der Benutzeragent nach ISO-Definition für diese Arbeit in zwei Komponenten aufgetrennt: Der *Client* wird vom Benutzer als Frontend ausgeführt und dient als Schnittstelle zwischen Benutzer und Dienst. Der *User Agent* läuft serverseitig und nimmt die Aufträge des Clients entgegen.

Der UA greift also auf die lokale Mailbox des Users zu und übermittelt die Nachrichten an dessen Client. Auf Wunsch des Benutzers kann die gelesene Nachricht anschließend gelöscht oder in einen anderen Ordner verschoben werden. Für diese Aufgaben verfügt der UA über eine Nutzschnittstelle. Über die Managementschnittstelle bietet der UA weitergehende Informationen an. Dabei handelt es sich in erster Linie um Attribute zum Fehlermanagement. Gleichzeit werden Statistiken über die Anzahl der Requests vorgehalten. Da die Konfiguration eines UA in der Regel nicht sehr komplex (man denke an einen POP3- oder IMAP4- Dämon) ist, bietet der UA an seiner Managementschnittstelle nur eine generische Konfigurationsmethode.

Damit ergibt sich folgendes Klassengebilde: UA erbt direkt von *compObject* und verfügt damit schon über dessen Grundausstattung (*OperationalState, AdminState*, etc). Die durch den Client aufrufbaren Funktionen befinden sich in der Schnittstelle *UAManageMailbox*, welche von *operationInterface* erbt. Die angebotenen Methoden sind *RetrieveMail*(), *MoveMail*(), *DeleteMail*(), *CreateFolder*(), *RenameFolder*() und *DeleteFolder*().

Die Managementschnittstelle *UAMgmt* erbt ebenfalls von *operationInterface* und bietet die folgenden Attribute und Methoden:

- *MsgOperations*: Anzahl aller in einem Zeitintervall t eingetroffenen Requests. Zugriff nur lesend. Unter Requests werden Leseanforderungen für Emails ebenso verstanden, wie Lösch- oder Verschiebeoperationen auf Nachrichten und Ordner.

- *Reject*: Anzahl aller Aufträge, die in einem Zeitintervall t vom UA generell nicht angenommen wurden (z.B. wegen extremer Last oder einem Fehlerzustand).
- *Accept*: Anzahl aller Aufträge, die in einem Zeitintervall t vom UA akzeptiert wurden. Es gilt: *Accept = Request – Reject*. Zugriff lesend.
- *Complete*: Anzahl der Aufträge, welche in einem Zeitintervall t vom UA vollständig abgearbeitet wurden. Dazu zählen sowohl erfolgreich abgearbeitet Requests, als auch solche, die einen Fehler verursachten. Zugriff lesend.
- *LastDelay*: Zeit in ms, die der UA für die Bearbeitung des letzten Auftrags brauchte. Zugriff lesend.
- *MeanDelay*: Durchschnittliche Zeit in ms, die der UA zur Abarbeitung eines Auftrags in einem Intervall t benötigte. Zugriff lesend.
- *MaxDelay*: Maximale Zeit, die der UA zur Abarbeitung eines Requests benötigen darf (Schwellwert). Überschreiten löst einen Performance Alarm aus. Zu diesem Zweck ist die Klasse UAMgmt mit PerfAlarm (siehe 4.3.4.2.1) assoziiert. Zugriff lesend, schreiben über Methode.
- *Bad*: Anzahl, der in einem Zeitintervall t akzeptierten Requests, die der UA als fehlerhaft erkannte und deswegen nicht bearbeitete. Beispiel hierfür ist etwa eine Leseanforderung auf eine Mail, die im Message Store nicht mehr vorhanden ist. Zugriff lesend.
- *Reject*: Anzahl der in einem Zeitintervall t akzeptierten Requests, die der UA aufgrund eines internen Fehlers nicht bearbeiten konnte und deshalb mit einem Fehler quittierte. Beispiel hierfür sind z.B. ein Stack Overflow oder ein physischer Lesefehler bei der Anforderung einer Mail. Zugriff lesend.
- *NotAuthorized*: Anzahl aller in einem Zeitintervall t angenommenen Requests, welche aufgrund fehlender Berechtigungen (z.B. Paßwortproblem) wieder verworfen wurden. Zugriff lesend.
- *SetMaxDelay*(): Methode zum Setzen von MaxDelay.
- *GetConfigValue*(): Liest einen Konfigurationsparameter aus und liefert den Wert im Ergebnis.
- *SetConfigValue*(): Setzt einen Konfigurationsparameter auf einen bestimmten Wert.

Schwere Fehler kann die Klasse *UAMgmt* auch signalisieren, indem sie eine entsprechende Instanz von *PerfAlarm* erzeugt.

4.3.4.2.4 Der Client

Der Client, also die Benutzeroberfläche zum Lesen, Verfassen und Verwalten von Mails ist nicht direkter Bestandteil einer Email-Infrastruktur. Allerdings kann es durchaus interessant seien, die Clients in die Betrachtungen miteinzubeziehen. Führt z.B. auch der Client Protokoll über erfolgreiche und fehlgeschlagene Requests, so läßt sich beim Ausfall eines Servers, wie z.B. eines MTAs sehr schnell analysieren, welche Bereiche (also z.B. Benutzer oder Abteilungen) vom Ausfall stark betroffen sind. Außerdem läßt sich bei außergewöhnlichen Lasten am Server eventuell ein verursachender Client leicht finden. Neben diesen statistischen

Informationen sollte der Client auch aus der Ferne konfigurierbar sein und ein automatisches Update erlauben. Diese Funktionalität wird in einem Interface *ClientStatus* zur Verfügung gestellt.

Client erbt direkt von *compObject*, die operative Schnittstelle *ClientStatus* von *operationInterface*.

Die Klasse *Client* verfüget also über folgende Attribute:

- *start*(), *stop*(), *read*(), *generate*(), *send*(), *manage*(): Vom Benutzer genutzte Funktionalität zum Starten, Beenden des Clients, sowie zum Lesen, Schreiben und Senden von Mails, sowie zur Verwalten der Mailbox über den Benutzeragenten.

Folgende managementspezifischen Attribute und Methoden finden sich in *ClientStatus*:

- *Request*: Request enthält die Anzahl der Aufträge, welche der Client in einem bestimmten Zeitraum t an einen Server, also einen MTA oder UA geschickt hat. Zugriff lesend.
- *Accept*: Dieses Attribut enthält die Anzahl der Aufträge, die korrekt bearbeitet wurden, für die der Client also eine postive Quittung vom Server erhalten hat. Zugriff lesend.
- *LastDelay*: Dieses Attribut enthält die Bearbeitungszeit für den letzten Auftrag aus der Sicht des Clients in Millisekunden. Zugriff lesend.
- *MeanDelay*: Dieses Attribut enthält die mittlere Bearbeitungszeit (in ms) für Aufträge aus Sicht des Clients in einem Zeitraum t. Zugriff lesend.
- *Fault*: Erhält der Client vom Server eine Fehlermeldung über einen fehlgeschlagenen Auftrag, so speichert er die Fehlermeldung in diesem Attribut ab. Neue Fehlermeldungen überschreiben hier immer die letzte Meldung. Zugriff lesend.
- *MTA*: Name des MTA, mit dem der Client zum Versenden von Mails kommuniziert. Zugriff lesend, Schreiben über Methode.
- *UA*: Name des UA, mit dem der Client zum Managen der Mailbox kommuniziert. Zugriff lesend, Schreiben über Methode.
- *SetMTA*(): Methode zum Setzen von MTA.
- *SetUA*(): Methode zum Setzen von UA.
- *Update*(): Veranlaßt den Client zu einem automatischen Update.

4.3.4.2.5 Der Directory Service

Das Message Handling System und der Directory Service greifen in mehreren Bereichen eng ineinander. Beispiele hierfür sind:

- Vergeben von benutzerfreundlichen Namen bzw. Aliasbezeichnungen anstatt schwer zu merkenden O/R oder SMTP Adressen. Zur Speicherung und Auflösung der Namen dient der Verzeichnisdienst.
- Einfache Definition von Verteilern im Verzeichnisdienst.
- Authentifizierung anhand im Verzeichnisdienst gespeicherter Daten.
- Gewinnung von Routinginformationen.

- Ermittlung der vom Empfänger unterstützen Nachrichtentypen durch *Capability* Attribute im Verzeichnisdienst.
- Abspeicherung zentraler Konfigurationsdaten und Policies in Verzeichnissen.

Der Verzeichnisdienst wird als Unterklasse von *compObject* modelliert und stellt seine Funktionalität an der Schnittstelle *GDAP Interface* (*Generic Directory Acess Protocol Interface*) zur Verfügung. Der MTA kann diese Funktionalität über seine Schnittstelle DAP Client in Anspruch nehmen.

Diese Modellierung des Verzeichnisdienstes als „Black Box“ ist natürlich sehr stark vereinfachend: Sowohl der Dienst selbst, als auch das Dienstmanagement sind, wie ein Email-System und dessen Management, eine verteilte Anwendung, die sich in einem ODP/OMT-Modell aus einer Vielzahl von Computational Objects zusammensetzt.

Das Management von Verzeichnisdiensten würde aber den Rahmen dieser Arbeit sprengen und wird im Detail in [Gars98] untersucht.

Für das Management der Email-Infrastruktur reicht es aus, die Klasse *DirectoryService* als Monolith zu modellieren. Die einzigen Managementattribute, mit denen der Verzeichnisdienst in diesem Modell ausgestattet wird, sind die von *compObject* geerbten Statusinformationen. Für das Funktionieren einer Email-Infrastruktur ist natürlich auch die Korrektheit der Daten, die *DirectoryService* liefert, relevant. Fehler in diesem Bereich werden jedoch bereits durch *MTARouting* abgedeckt und nicht beim Verzeichnisdienst modelliert.

4.3.4.2.6 Die Benutzerverwaltung (MUser/MHSUser)

Die Benutzerverwaltung ist zentraler Punkt einer Email-Infrastruktur. Da dort vorgenommene Änderungen, wie z.B. das Anlegen eines Benutzers alle Komponenten des Systems beeinflussen, wurde die Klasse *MHSUser* (*Message Handling System User*) nicht als operative Schnittstelle einer bestehenden Klasse modelliert, sondern als eigenes Objekt, welches direkt von *compObject* erbt.

Ein einzelner Benutzer wird dabei durch die Klasse *MUser* modelliert, der in einer Enthaltensseinsbeziehung zu *MHSUser* steht. Jeder Benutzer hat die Möglichkeit, seine Mails an einen oder mehrere weitere Benutzer umzuleiten. Dies wird durch die Beziehung *redirect_to* modelliert. Ein Benutzer kann auch „virtuell“ sein, d.h. eine Mail an diesen Benutzer hat ausschließlich Verteilerfunktionalität.

Ein Objekt der Klasse *MUser* enthält die folgenden Attribute:

UserID: Von User geerbt. Die Adresse des Benutzers im Email-System.

Name: Von User geerbt. Der Realname des Benutzers.

Password: Das Paßwort des Benutzers.

Mailbox: Mailbox, in welcher die Mails für den Benutzer abgespeichert werden.

MaxMsgLimit: Maximale Anzahl von Mails, die ein Benutzer in seiner Mailbox speichern darf.

MaxKBLimit: Maximale Größe in KB, welche die Mailbox eines Benutzers haben darf.

MaxMsgSize: Maximale Größe, die eine einzelne, vom Benutzer versandte Mail haben darf.

WildcardsAllowed: Steuert, ob und wie (nur innerhalb der Gruppe, innerhalb der Organisation, weltweit) Wildcards in Empfängeradressen aufgelöst werden.

Virtual: *Yes* oder *No*, gibt an, ob der Benutzer real oder virtuell ist. Ein virtueller Benutzer ist eine Gruppenbezeichnung oder eine Verteilerliste, welche für Redistributionen genutzt wird. Ist der Benutzer virtuell, so sind die Attribute *Password* bis *WildCardsAllowed* leer.

Der Zugriff auf alle Attribute ist nur lesend möglich. Zur Verwaltung eines Benutzers dient die Klasse *MHSUSer*, welche die folgenden Attribute und Methoden zur Verfügung stellt:

- *CreateMHSUser*(): Legt einen neuen Benutzer an, macht diesen dem Email- und Verzeichnisdienst bekannt, erzeugt die lokale Mailbox, etc.
- *RenameMHSUser*(): Benennt einen Benutzer um.
- *DeleteMSHUser*(): Löscht einen Benutzer komplett aus dem Mailsystem.
- *SetMHSUserPassword*(): Setzt das Paßwort eines Benutzers neu.
- *GetRedirector*(): Die Methode erlaubt, die aktuelle Umleitung eines Benutzers abzufragen, d.h. sie liefert die Beziehung einer Instanz der Klasse MUser zu anderen Instanzen dieser Klasse.
- *SetRedirector*(): Erlaubt es, vom Benutzer gesetzte Umleitungen, zu ändern oder löschen.
- *AllowRedirector*(): Erlaubt es, die Umleitungsfunktionalität ein- und auszuschalten, ohne mittels *SetRedirector*() Umleitungen löschen und neu anlegen zu müssen.
- *SetMaxMsgLimit*(): Methode zum Setzen von *MaxMsgLimit* eines Benutzers.
- *SetMaxKBLimit*(): Methode zum Setzen von *MaxKBLimit* eines Benutzers.
- *SetMaxMsgSize*(): Methode zum Setzen von *MaxMsgSize* eines Benutzers.
- *SetMailbox():* Methode zum Setzen des Attributes *Mailbox* eines Benutzers.
- *AllowWildcards*(): Methode zum Setzen der Wildcard-Funktionalität eines Benutzers.

4.3.4.2.7 Die Modellierung einer Email und deren Fluß im System

Beim Systemstart werden Objekte, die Managementschnittstellen implementieren, automatisch instanziiert: So bedingt das Starten eines MTA das Erzeugen der Objekte *MTARouting*, *MTAConversion*, *MTASecurity*, *MTAPerformance* und *MTAConfiguration*, das Starten eines UA zieht das Instanziieren eines *UAMgmt* Objektes nach sich und das Aufrufen des Clients durch den Benutzer erzeugt eine Instanz von *ClientStatus*. Das System befindet sich somit in Bereitschaft.

Was passiert nun, wenn ein Benutzer eine Email verfaßt und diese abschickt? Um den Fluß von Emails erfassen zu können, werden Interaktionen an den Schnittstellen der Objekte protokolliert. RM-ODP sieht zu diesem Zweck die Basisklasse *interactionInfo* vor. Wird ein Schnittstellenobjekt, wie z.B. ein *operationInterface* erzeugt, so werden die Anzahl der eingehenden Requests, sowie deren Name, Größe und Zeitpunkt des letzten Requests festgehalten. Je nach Art der eingehenden Requests, ist eine weitere Verfeinerung mittels der drei Unterklassen *flowInfo*, *announceInfo* und *interrogationInfo* möglich.

Um das Ergebnis eines eingehenden Requests ebenfalls festhalten zu können, ist jede Instanz einer *interactionInfo* mit ein oder mehreren Objekten der Klasse *terminationInfo* assoziiert.

RM-ODP erlaubt, mehrere Requests in einem *interactionInfo* Objekt zu „sammeln“, oder für jede Art von Request ein neues Objekt zu instanziieren. Damit ist klar, daß eine Email als Unterklasse von *interactionInfo* bzw. *interrogationInfo* modelliert werden muß.

Zu diesem Zweck enthält das Objektmodell die beiden Klassen *EmailGenericInfo* und *EmailDeliveryStatus*.

EmailGenericInfo verfügt über die folgenden Attribute:

- *Name* (von *interactionInfo* geerbt): „Name“ der Email, also die global eindeutige Message-Id.
- *Count* (von *interactionInfo* geerbt): Da für jeden neuen Email-Request eine neue Instanz von *EmailGenericInfo* erzeugt wird, steht dieser Zähler immer auf 1.
- *Last* (von *interactionInfo* geerbt): Erstellungsdatum der Email.
- *Bytes* (von *interactionInfo* geerbt): Gesamtgröße der Email in Bytes
- *StartTime* (von *interrogationInfo* geerbt): Zeitstempel des Eintreffens der Email.
- *LastDelay* (von *interrogationInfo* geerbt): Dauer bis zur endgültigen Abarbeitung des Requests.
- *OrigFromAddr*: Die Absenderadresse, wie sie in der eingehenden Mail enthalten war.
- *OrigToAddr*: Die Empfängeradresse(n), wie sie in der eingehenden Mail enthalten war(en).
- *NewFromAddr*: Die Empfängeradresse nach einer eventuellen Umformung durch den MTA. Falls keine Umformung stattfand, identisch mit *OrigFromAddr*.
- *NewToAddr*: Der oder die Empfängeradressen nach einer Umwandlung und/oder Redistribution durch den MTA. Falls keine Umwandlung stattfand, identisch mit *OrigToAddr*.
- *RoutingList*: Pfad, den die Email bisher durch das System gegangen ist. Enthält als letzten Eintrag immer den Namen des lokalen MTAs.
- *Subject*: Die Subject-Zeile der Email.
- *Priority*: Priorität, mit welcher die Nachricht vom MTA eingestuft wurde. In der Regel wird die Priorität vom Absender vorgegeben. Der MTA kann diesem Wunsch folgen, muß es jedoch nicht (z.B. bei Überlast).
- *LocalID*: Lokale Auftragsbezeichnung, welche der MTA einem Request vergibt. Im Gegensatz zur Message-Id ist diese Bezeichnung weder über Zeit, noch Raum eindeutig.

EmailDeliveryStatus wird wie *EmailGenericInfo* sofort beim Eintreffen eines Requests erzeugt und verfügt über die folgenden Attribute:

- *Status*: Gegenwärtiger Status der Email. Kann folgende Werte haben:
    - *Receiving*: Sobald ein MTA einen Request annimmt, erzeugt er eine Instanz von EmailDeliveryStatus und setzt Status auf Receiving.

    - *Processing*: Zeigt an, daß der MTA eine Mail komplett empfangen und in einer Inbound Queue gespeichert hat.
    - *Stopped*: Zeigt an, daß eine Mail nicht mehr weiter bearbeitet wird. Dies ist das Resultat einer Operation aus dem Queue Management. Wird die Mail wieder bearbeitet, wechselt Status erneut auf Processing.
    - *Delivered*: Die Mail wurde durch den MTA lokal ausgeliefert.
    - *Transmitting*: Die Mail befindet sich in Übertragung an einen entfernten MTA.
    - *Relayed*: Die Mail wurde an einen entfernten MTA ausgeliefert.
    - *Deferred*: Zeigt an, daß die Mail augrund eines Fehler nicht ausgeliefert werden konnte. Nach einer Wartezeit wird ein weiterer Versuch gestartet.
    - *Failed*: Zeigt an, daß eine Mail aufgrund eines Fehlers endgültig nicht bearbeitet werden konnte und an den Empfänger zurückgeschickt oder verworfen wurde.
- *ProcessingError*: Wird gesetzt, falls eine Email temporär oder dauerhaft nicht bearbeitet werden kann. Wird eine Email nach einem temporären Fehler ausgeliefert, so wird der Attributsinhalt wieder gelöscht.

Ein typischer Ablauf könnte nun wie folgt aussehen: Ein Benutzer schreibt mit seinem Email-Client eine Nachricht und sendet diese ab. Der MTA nimmt die Mail an seiner Schnittstelle *MTA InOut* entgegen und erzeugt je eine Instanz von *EmailGenericInfo* und *EmailDeliveryStatus*. Die Attribute der beiden Objekte werden soweit wie möglich bereits gesetzt. Der Status der Email wechselt nun auf *Processing*. Der MTA untersucht die Zieladresse und findet heraus, daß der Empfänger nicht lokal ist. Also ermittelt er das Routing mit Hilfe des Verzeichnisdienstes. Eventuelle Adreßumwandlungen werden in *EmailGenericInfo* eingetragen. Anschließend trägt sich der MTA in *RoutingList* ein und sendet die Mail an den nächsten MTA. Die Transaktion ist für den MTA abgeschlossen, der Status der Mail wird auf *Relayed* gesetzt und die Attribute der MTA-Klassen werden aktualisiert. Beim nächsten MTA beginnt der Zyklus von neuem.

Durch Befragen sämtlicher *EmailGenericInfo* und *EmailDeliveryStatus* Objekte ist es nun jederzeit möglich, den Weg einer Email durch das System zu verfolgen und den gegenwärtigen Status (z.B. in einer Queue gespeichert, in Übertragung zwischen zwei oder mehreren MTAs, oder ausgeliefert) zu prüfen. Ebenso können die Verweilzeiten bei den MTAs abgerufen und Fehlerzustände festgestellt werden, Adreßumwandlungen können nachvollzogen werden und vieles mehr.

Ist der Datenverlust bei den bisherigen Klassen nicht allzu schwerwiegend, so muß für die Objekte EmailGenericInfo und EmailDeliveryStatus unbedingt Persistenz auch z.B. über einen Systemneustart gefordert werden, damit sich der Weg einer Nachricht über längere Zeit verfolgen läßt.

4.3.4.2.8 MHSHistory

Auf dieKlasse *MHSHistory* wird an dieser Stelle nicht weiter eingegangen, da sie nicht mehr zentraler Teil der Infrastruktur ist.

Die Klasse bietet an ihrer Schnittstelle *LogService* einen Archivierungsdienst an, der benötigt wird, um Daten für Trendanalysen oder Planungszwecke auch langfristig verfügbar zu halten.

4.3.4.2.9 Zusammenfassung

In den Abschnitten zum Computational Viewpoint wurde eine verteilte Email-Infrastruktur abstrahiert, modelliert und mit Managementschnittstellen versehen. Die Möglichkeiten des Modells erlauben die Umsetzung der aus Abschnitt 2.2.1 bekannten Forderungen durch eine Managementanwendung.

Von den weitergehenden Szenarien aus Abschnitt 2.2.2 wurde die Benutzerverwaltung modelliert. Für den dort geforderten Topologiedienst und das Message Tracking wurden Schnittstellen bzw. entsprechende Objekte geschaffen, welche einer Managementanwendung alle benötigten Daten liefern können.

Falls zusätzliche Funktionen benötigt werden, können sie leicht in die bestehenden Klassen integriert werden.

4.3.5 Engineering Viewpoint

4.3.5.1 ODP-Basisklassen für den Engineering Viewpoint

Im Computational Viewpoint des RM-ODP wurde eine verteilte Anwendung in Komponenten zerlegt. Die Aufteilung in Klassen wie MTA, UA oder MS erfolgte dabei nach funktionalen Gesichtspunkten. Die tatsächliche Verteilung und die Kommunikationswege zwischen den Komponenten spielte bei dieser Betrachtung noch keine Rolle. Für eine Implementierung des Modells wird nun eine Plattform benötigt, mit deren Hilfe eine Verteilung und die bereits im Computational Viewpoint spezifizierte Objektkommunikation realisisiert werden kann.

Zur Modellierung einer solchen Plattform stellt das RM-ODP wieder Basisklassen, welche ein maschinenunabhäniges verteiltes System und seine Kommunikationskanäle modellieren bereit.

Die verteilte Plattform des Engineering Models besteht aus einer Anzahl vernetzter Rechner, den *Nodes*. Der *Nucleus* kontrolliert die Rechen-, Speicher- und Kommunikationsressourcen der Nodes, entspricht also in etwa dem Betriebssystem eines Rechners. Gemäß RM-ODP beherbergt ein Node genau einen Nucleus. Wie sich bei der späteren Betrachtung der Agenteninfrastruktur zeigen wird, läßt sich diese Vorgabe nicht halten. Weiterhin beherbergt ein Node mehrere *Capsules*; ein Capsule abtrahiert den traditionallen Prozeßbegriff, d.h. es besizt seinen eigenen, geschützten Adreßraum und nutzt Teile der Speicher- und Rechenressourcen, die der Nucleus verwaltet. Da die Agentenumgebung JAVA-basiert (siehe Kapitel 5) ist, also in einer virtuellen Maschine abläuft, kann auch diese Modellierung nicht vollständig umgesetzt werden.

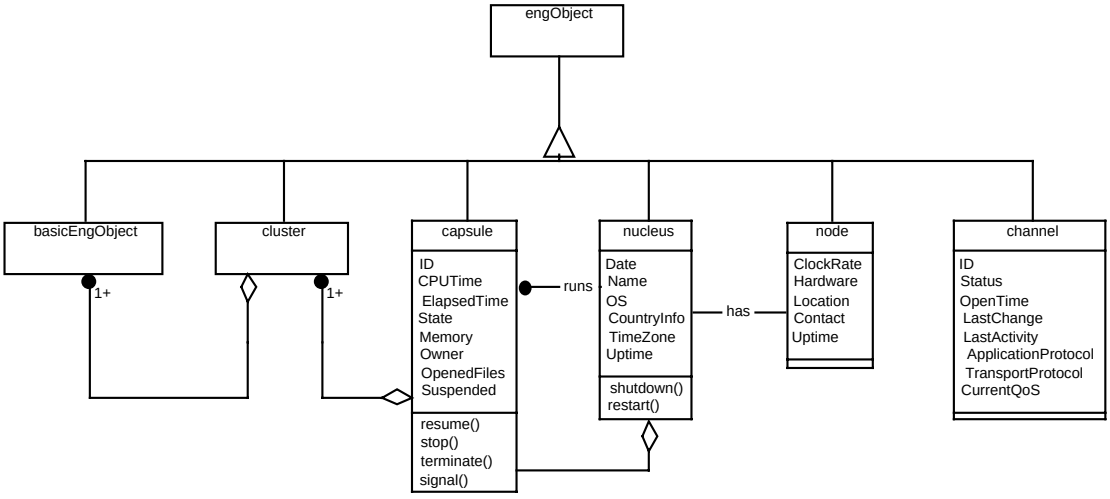


Abbildung 26 - ODP Basisklassen des Engineering Viewpoints (aus [Muel98])

Ein Capsule setzt sich aus mehreren *Clusters* von *Basic Engineering Objects* (BEO) zusammen. Die Objekte des Computational Viewpoint werden zur Laufzeit durch Basic Engineering Objects realisiert. An dieser Stelle wird also die Abbildung vom Computational auf den Engineering Viewpoint vorgenommen, d.h, es wird z.B. sichtbar, ob mehrere Managementschnittstellen zusammen in einem Agentenprozeß realisiert werden, oder ob die Funktionalität einer Schnittstelle sogar auf mehrere kommunizierende Prozesse verteilt wird. Die ODP Vorstellung ist nun, daß mehrere BEOs zusammen ein Cluster bilden, welches allein aber nicht lauffähig ist. Weitere Cluster, wie etwa Libraries oder Startup-Code werden von einem Linker oder Classloader entweder statisch oder dynamisch zur Laufzeit zu einem Capsule gebunden.

Das ODP-Modell definiert außerdem Kommunikationskanäle (*Channels*), welche die Bindungen aus dem Computational Model wiedergeben. Im Computational Viewpoint konnten Objekte, deren Computational Interfaces aneinander gebunden, waren, Interaktionen eingehen. Im Engineering Model werden diese Interaktionen nun auf auf Kommunikationskanäle zwischen den Prozessen abgebildet. Ebenso wird sichtbar, mit welcher Infrastruktur das Engineering Model die Kommunikation unterstützt.

Damit ergibt sich für die Modellierung des Engineering Viewpoints folgendes Ausgangsmodell:

Alle Klassen erben von *engObject*. *node* abstrahiert einen Rechner, was durch die hardwarespezifischen Attribute *ClockRate*, *Hardware*, *Location*, *Contact* und *Uptime* ausgedrückt wird. *node* ist ist mit *nucleus* assoziiert, welche als Klasse über die Attribute *Date*, *Name*, *OS*, *CountryInfo*, *TimeZone* und *Uptime* verfügt. Zusätzlich bietet *nucleus* mittels der Methoden *shutdown*() und *restart*() die Möglichkeit, den Node anzuhalten oder neu zu starten. Das Betriebssystem, also *nucleus* ist einerseits aus mehreren Capsules, also Prozessen (die meist im privilegierten Modus laufen) aufgebaut und verwaltet weitere Prozesse (die dementsprechend nicht privilegiert sind). Ein Capsule verfügt über klassische Attribute aus dem Prozeßmanagement: *ID*, *CPUTime*, *ElapsedTime*, *State*, *Memory*, *Owner*, *OpenFiles* und *Suspended*. Weiterhin kann ein Capsule mittels *stop*() und *resume*() angehalten und wieder gestartet werden, sowie mittels *signal*() Signale empfangen und mittels *terminate*() beendet werden. Wie bereits erwähnt setzt sich *capsule* aus mehren Clustern und

ein Cluster aus mehreren BEOs zusammen, was im Modell durch die entsprechenden Aggregationen ausgedrückt wird. Die Attribute der Klasse *channel* beschreiben Managementinformationen, welche für Kommunikationsverbindungen aller Art angewendet werden können. Jeder Channel besitzt einen eindeutigen Identifikator (*ID*), den Zeitpunkt, zu dem er erzeugt wurde (*OpenTime*), sowie einen Zustand (*Status*). Mögliche Zustände könnten z.B. *up*, *down*, *congested* oder *unknown* sein. Die Dienstgüte des Kanals kann über *CurrentQoS* abgefragt werden. *LastChange* liefert den Zeitpunkt der Konfigurationsänderung eines Kanals, Zeitpunkt der letzten Aktivität wird in *LastAcitvity* gespeichert. Die Attribute *ApplicationProtocol* und *TransportProtocol* geben Auskunft über die eingesetzten Protokolle der Anwendungs- bzw. Transportschicht.

Basierend auf den generischen Basisklassen wird nun das, auf Java und CORBA basierende System flexibler Managementagenten, welche beim MNM-Team zum Einsatz kommt modelliert. Das Modell wird dabei nur soweit präzisiert, wie es für die spätere prototypische Implementierung eines Agenten notwendig ist. Dies bedeutet, daß nur die Klassen *MTASecurity* und *MTA* des Computational Viewpoint auf ein Basic Engineering Object abgebildet wird und daß spezielle Dienste des Agentensystems, wie Naming- oder Eventservice nicht modelliert werden. Eine detaillierte Beschreibung des Agentensystems findet sich in [Kemp98], Managementagenten wurden z.B. in [Allg98, Demm98, Radi98] entwickelt.

4.3.5.2 Modellierung des Agentensystems

In Kapitel 5 wird die Zugangskontrolle zum MTA implementiert. Die folgende Modellierung soll deshalb die Abbildung der Computational Objects *MTASecurity* und *MTA* auf den Engineering Viewpoint, sowie deren Kommunikation über Kanäle demonstrieren.

An der Spitze der Klassenhierarchie steht wieder *engObject* mit den generischen ODP-Unterklassen *basicEngObject*, *cluster*, *capsule*, *nucleus*, *node* und *channel*. Die Klasse *Generic Association* ist ein Hilfskonstrukt: Eigentlich sollte die generische Assoziation zwischen basicEngObject und channel weiter verfeinert werden. Da eine Verfeinerung von Assoziationen in StP jedoch nicht möglich ist, wurde die leere Assoziationsklasse *Generic Association* gebildet, die dann weiter verfeinert wird.

Bei der Umsetzung von Computational Objects auf BEOs sieht [X901] sowohl 1:1 als auch 1:n Abbildungen vor. Die Klasse MTA aus Abbildung 27 stellt eine solche 1:1 Umsetzung der Klasse *MTA* aus Abbildung 25 dar. Die Klassen *MTASecurity Relaying* bzw. die Agentenimplementierung *MTAAccessDB Running Agent* resultieren aus einer 1:n Abbildung der Klasse *MTASecurity* des Computational Viewpoint. Die weiteren aus *MTASecurity* resultierenden BEOs werden im Modell nicht gezeigt.

Wie bereits angedeutet, erfolgt die Implementierung des Agenten mit Java innerhalb einer CORBA-Umgebung. Im weiteren Vorgehen werden aus den generischen ODP-Klassen sowohl ein klassischer Unix-Prozeß als auch die Java-Umgebung abgeleitet. Dies erlaubt, die Kommunikation zwischen Agent und MO zu modellieren.

Die beiden Objekte *Java Class File* und *OBJ Libraries* sind Unterklassen von *cluster*, stellen also jeweils kleinste Einheiten von ausführbarem Code dar. *Java Thread* und *Unix Process* sind Unterklassen von *capsule* und setzten sich aus mehreren Clustern zusammen. Wie bereits erwähnt, stellt diese Modellierung einen kleinen Bruch mit dem ODP-Modell dar. Dort wird

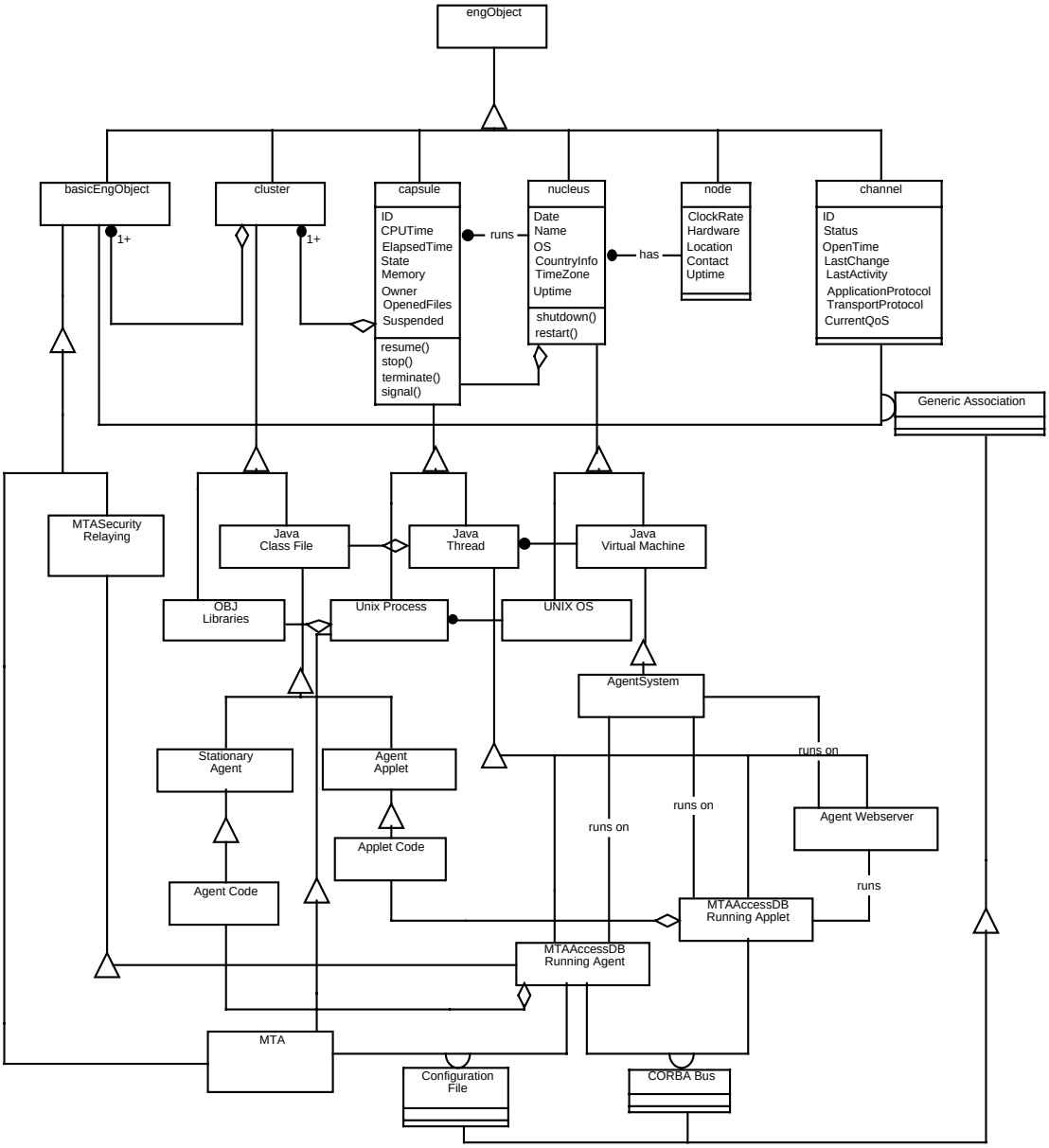


Abbildung 27 - Engineering Viepoint der prototypischen Implementierung

betont, daß ein Capsule einen eigenen Adreßraum besitzt, welcher durch den Nucleus geschützt wird. Dies trifft auf *Java Thread* innerhalb der Java VM jedoch nicht zu.

Unix Process und *Java Thread* laufen jeweils auf einem Betriebssystem, welches eine Verfeinerung von *nucleus* darstellt: Das Betriebssystem für den Unix-Prozeß wird durch die Klasse *UNIX OS* modelliert, für den Thread stellt *Java Virtual Machine* die Ausführungsumgebung dar.

UNIX OS und *Java Virtual Machine* laufen auf einem Node. Die 1:1 Beziehung aus dem RM-ODP zwischen *node* und *nucleus* kann an dieser Stelle nicht eingehalten werden: Auf einem Rechner können schließlich beliebig viele virtuelle Javamaschinen gestartet werden.

MTA kann nun aus den bereits bestehenden Klassen als Verfeinerung von *Unix Process* abgeleitet werden. Die Zusammensetzung aus mehreren *OBJ Libraries* Instanzen ist implizit.

Für den Java-Agenten muß hingegen sowohl *Java Class File* als auch die Ausführungsumgebung noch weiter verfeinert werden. *Stationary Agent* und *Agent Applet* sind Java-Klassen aus [Kemp98]. Die Verfeinerung dieser Klassen zu *Agent Code* und *Applet Code* wird in Kapitel 5 näher beschrieben.

Das Agentensystem stellt eine Art „Betriebssystem“ für Agenten dar: Es startet und beendet Agenten, ermöglicht ihre Kommunikation und erlaubt die Migration von Agenten durch das System. Deshalb wird das Agentensystem, obwohl es selbst ein Thread ist, nicht als Verfeinerung von Java Thread modelliert, sondern von Java Virtual Machine abgeleitet. Die drei Agenten *MTAccessDB Running Agent*, *MTAAccessDB Running Applet* und *Agent Webserver* sind *Java Thread* Verfeinerungen, welche auf *AgentSystem* laufen. Die Agenten setzen sich dabei aus den jeweiligen *Agent Code* bzw. *Applet Code* Klassen zusammen. Die Beziehung zu weiteren *Java Class File* Objekten ist implizit (geerbt).

Die bisherigen Klassen stellen sowohl die Ausführungsumgebung, als auch einen Teil der Vererbungshierarchie der Java-Klassen dar. Wie bereits erwähnt, ist CORBA die Kommunikationsinfrastruktur. Um diesen Kommunikationkanal zwischen den Agenten zu modellieren, wurde eine Assoziation zwischen *MTAAccessDB Running Agent* und *MTAAccessDB Running Applet* mit einer Assoziationsklasse *CORBA Bus* eingefügt. *CORBA Bus* ist eine Verfeinerung der Oberklasse *Generic Association*.

Eine ganz andere Kommunikation findet zwischen dem Agent und seinem MO, dem MTA statt: Zur Steuerung von *MTA* schreibt der Agent in dessen Konfigurationsfile. Dies wird durch die Assoziationsklasse *Configuration File*, welche wieder von *Generic Association* erbt, dargestellt.

5 Prototypische Implementierung

Im folgenden Kapitel wird die Abbildbarkeit des Managementmodells auf eine bestehende Architektur anhand einer prototypischen Implementierung geprüft. Abschnitt 5.1 gibt zwei mögliche Einsatzszenarien und motiviert die Auswahl der implementierten Methoden des Modells. Abschnitt 5.2 und 5.3 geben einen kurzen Überblick über den zum Einsatz kommenden MTA *sendmail* und die Entwicklungsumgebung. Abschnitt 5.4 und 5.5 stellen den entwickelten Agenten und das Managementapplet vor.

5.1 Motivation

5.1.1 Das Problem unerwünschter kommerzieller Emails

Jeder, der heute über das Internet Emails austauscht, kennt das Problem von unverlangt zugesandten Mails. Diese Mails werden millionenfach verteilt und werben meistens für Produkte oder eine Webseite, auf der wiederum Produke und Dienstleistungen angeboten werden. Oftmals handelt es sich um illegale Kettenbriefe oder sonstige dubiose Aufforderungem, die zum schnellen Geld führen sollen.

Im englischen Sprachraum werden diese Art von Emails als *SPAM-Mails*[5] oder *UCE* (*Unsolicited Commercial Email*), das Versenden der Mails dementsprechend als Spamming bezeichnet. Das massenhafte Versenden von Werbenachrichten erregte 1994 erstmals Aufsehen, als die amerikanische Anwaltskanzlei *Canter & Siegel* mehr als 8000 Newsgruppen mit Werbungen für die jährliche Greencard Lotterie der Vereinigten Staaten überflutete.

Im November 1996 fielen die Mailserver des Providers T-Online für drei Tage aus, nachdem Spammer eine Werbemail millionenfach verschickten. 1,3 Millionen Kunden waren für diesen Zeitraum ohne Email-Versorgung.

Unverlangt zugesandte, kommerzielle Email wird von Empfängern und deren Provider also aus folgenden Gründen abgelehnt:

- Die Gebühren für die unverlangte Werbung werden fast ausschließlich vom Empfänger bezahlt. Ein Email-Verteiler an eine Million Adressen kostet den Versender über eine in den USA billig zu habende Standleitung (T1) kaum etwas. Rechnet man dagegen bei den Empfängern nur fünf Sekunden für den Empfang einer Email, so fallen insgesamt 1388 Stunden Onlinezeit bei den Empfängern an, die bezahlt werden müssen.
- Hinzu kommt der Ausfall von Produktivität bei den Empfängern und Systembetreuern, welcher durch das Bearbeiten/Löschen/Verhindern von Werbung entsteht.

[5] Spam ist eine Kurzform für „Spiced Pork and Ham“. Die Übernahme der Bezeichnung auf Werbemails stammt von einem Sketch der englischen Komikertruppe Monty Python. Dort wird in einem Restaurant so aufdringlich für Spam geworben, daß keine normale Kommunikation mehr möglich ist.

- Das Ausliefern von Werbemails nimmt enorme Ressourcen in Anspruch. Die Versender von Emails lierfern oft nicht direkt aus, sondern „kippen“ die Werbung bei einem unbeteiligten System ab, welches die Auslieferung übernimmt.

Wie bereits angdeutet ist es ein großes Problem, daß die Versender von Werbemails die Header ihrer Emails fälschen, um ihre Spuren zu verwischen. Das im Internet verwendete Mailprotokoll SMTP sieht keinerlei Authentifizierungsmaßnahmen zum gesicherten Inverkehrbringen von Emails vor; an Lösungen, wie etwa einem eigenen Mail-Submission-Protokoll [GeKl98], wird noch gearbeitet. Deshalb ist es oft schwierig, den tatsächlichen Verursacher einer Werbemail ausfindig zu machen. Schlimmer noch ist, daß bisher fast alle Systeme im Internet, welche Mail versenden und empfangen, keine Überprüfung vornehmen, ob der Empfänger einer Email tatsächlich ein lokaler Benutzer ist. Dieses bedingungslose Relaying, also das Akzeptieren von Mails nichtlokalen Ursprungs an nichtlokale Empfänger ist z.B. beim *sendmail* MTA bis einschließlich Version 8.8.8 nicht abzuschalten. Dies hat zur Folge, daß nahezu jeder Host im Internet, auf welchem ein MTA läuft als Werbelieferant mißbraucht werden kann – in Verbindung mit gefälschten Mailheadern sieht es sogar so aus, als stammten die Emails vom mißbrauchten System. Die Betroffenen haben doppelten Schaden: Die Kosten für das Relaying der Spam-Mails und die Auseinandersetzung mit den verärgerten Empfängern, welche sich nun an den vermeintlichen Absender wenden.

Da die rechtliche Situation bezüglich einem Verbot von Werbemails nach wie vor nicht geklärt ist, müssen Provider und Email-Betreiber zu technischen Hilfsmitteln greifen, um das Spamming zu unterbinden, eine typische Aufgabe aus dem Securitymanagement der Email-Infrastruktur.

5.1.2 Zugangskontrolle für nomadische Systeme

Als *nomadische Systeme* (NoS) werden sich im Unternehmensnetz bewegende Objekte bezeichnet. Beispiele hierfür sind z.B. Notebooks oder *Personal Digital Assistants* (PDAs). Prinzipiell soll es diesen nomadischen Systemen ermöglicht werden, sich an jeder Stelle im Unternehmen ins Netz einklinken zu können. Aufgabe des Managements ist es dabei, die nomadischen Systeme zu identifizieren, ihnen entsprechend abgestufte Zugriffsrechte einzuräumen und für eine Abrechnung benutzer Ressourcen zu sorgen.

In [Heil98a] wird ein Einsatzszenario für das Management nomadischer Systeme entwickelt. In der in Abbildung 28 dargestellten Bibliothek soll Besuchern mit Notebooks die Möglichkeit gegeben werden, sich an das vorhandene Intranet anzuschließen. Die Benutzung weiterer Dienste, wie etwa das Versenden von Email soll unbekannten, also anonymen Sendern untersagt bleiben. Den Benutzer bekannter NoS, denen einen Account eingerichtet wurde, sollen hingegen erweiterte Zugriffsrechte eingeräumt bekommen: Sie sollen die Dienste WWW und Mail benutzen können.

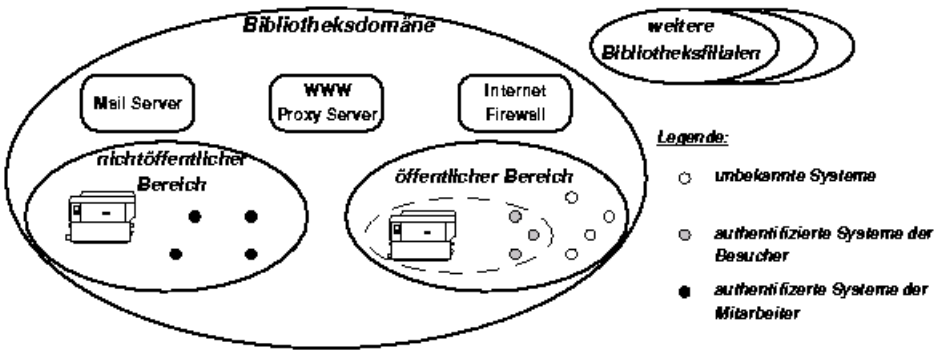


Abbildung 28 - Bibliotheksszenario

In [Radi98] wird ein policybasiertes Konzept zum Management nomadischer Systeme erarbeitet. Ein dort entwickelter *Enforcement Service* überwacht dabei die Durchsetzung der zuvor definierten Policies, indem er beim Auftreten von Events, wie etwa dem Anschluß eines neuen Systems, Operationen an den Managementschnittstellen der überwachten Ressourcen auslöst.

Um MTAs als Managed Objects in obiges Szenario integrieren zu können, müssen sie die Möglichkeit zur Verfügung stellen, Client Requests, basierend auf ihrer IP-Adresse, annehmen oder ablehnen zu können. Der Enforcement Service muß die Liste akzeptierter IP-Adressen außerdem ständig verändern können.

5.2 Konfiguration der Zugangskontrolle in sendmail 8.9

sendmail 8.9 ist seit Juli 98 verfügbar. Die wichtigste Neuerung dieser Version ist die verbesserte Zugangskontrolle, also neue Optionen, welche unkontrolliertes Relaying, basierend auf Domainname oder IP-Adresse des Clients unterbinden sollen. Hierzu zählen:

- Relaying ist in der Grundeinstellung abgeschaltet.
- Vom Absender übermittelte Informationen werden genauer überprüft.
- Bessere Prüfung von Mailheadern.
- Einführung einer *Access Control* Datenbank, welche den Zugriff auf den MTA steuert.

Die Komplexität einer *sendmail*-Konfiguration ist sehr hoch. Dennoch ist der MTA seit den Verbesserungen durch die *IDA* (*Institutionen för Datavetenskap*) deutlich leichter zu konfigurieren geworden. Das direkte Editieren der Konfigurationsdatei *sendmail.cf* kann heute meist vermieden. werden.

Stattdessen erstellt man eine Datei *sendmail.mc* mit Makrodefinitionen, welche das Verhalten von *sendmail* festlegt. Dabei kann man sich auf vorgefertigte Makros für die wichtigsten Konfigurationsaufgaben abstützen.. Anschließend übersetzt ein Makropräprozessor die Konfiguration und erstellt die *sendmail.cf* Datei.

Auch die Benutzung der oben erwähnten *Access Control* Datenbank läßt sich durch ein einfaches Makro aktivieren. Mittels des Befehls **`FEATURE(access_db, hash –o /fs/dir/name)`** gibt man hierzu lediglich den Pfad zur Datenbank in der Definitionsdatei an.

Zur Generierung der Datenbank muß zuerst eine Textdatei erstellt werden. In jeder Zeile dieser Datei steht eine Kombination aus Schlüssel und Wert. Der Schlüssel kann dabei ein Benutzer- oder Domainname bzw. eine IP- oder Netzwerkadresse sein. Der Wert ist ein String, welcher wie folgt aufgebaut sein darf:

OK	Akzeptiert die Mail, selbst wenn andere Konfigurationseinträge die Mail als unberechtigt abweisen würden.
RELAY	Erlaubt einem Domain das Relay durch den lokalen MTA. RELAY impliziert auch OK.
REJECT	Blockt die Email mit einer generischen Fehlermeldung ab.
DISCARD	Verwirft die Email komplett, ohne dem Sender eine Fehlermeldung zu übermitteln.
### beliebiger Text	### entspricht einem gültigen Statuscode aus [RFC821], danach folgt ein beliebiger Text. Ein Beispiel wäre etwa *„550 Sorry, we accept mail from authorized personnel only“*.

Ein Beispiel für eine Textdatei *input.txt* wäre also:

```
cyberspammer.com          550 We dont't accept SPAM Mail!
okay.cyberspammer.com     OK
sendmail.org              RELAY
128.32                    550 Sorry, personnel only
128.32.10.1               OK
128.32.10.2               OK
```

Anschließend ist die Datei mittels **`makemap hash /fs/dir/name <input.txt`** in eine Datenbank zu konvertieren. Die so gesetzten Regeln werden von *sendmail* ohne Neustart sofort beachtet.

Für die prototypische Implementierung eines Management-Agenten wurde wurde die Umsetzung der Methoden *GetAllowedRelay*() und *SetAllowedRelay*() der in Abschnitt 4.3.4.2.1 entwickelten Klasse *MTASecurity* gewählt, welche die Verwaltung der *sendmail* Zugriffsrechte erlauben.

5.3 Entwicklungsumgebung

5.3.1 Java

Die erste Version der Programmiersprache Java wurde Ende 1995 von *Sun Microsystems* veröffentlicht. Aufgrund ihrer Portabilität und Unabhängigkeit von Rechner- oder

Betriebssystemarchitekturen fand die Sprache rasche Verbreitung. Insbesondere die Möglichkeit kleine Anwendungen, sogenannte *Applets*, direkt im WWW-Browser eines Benutzers ausführen zu können, fand bei Programmierern starken Zuspruch.

Einige zentrale Eigenschaften von Java sind ([Flan97]):

- Java ist eine objektorientierte Sprache. Im Gegensatz zu z.B. C++ sind fast alle Typen, welche die Sprache zur Verfügung stellt bereits Objekte. Für den Programmierer stehen bereits Klassen für gängige Basistypen, Container, I/O-Operationen, graphische Oberflächen, Eventhandling uvm. zur Verfügung. Diese Klassen werden in *Packages* zusammengefaßt. Häufig genutze Packages sind z.B. java.io, java.awt oder java.util.
- Java ist architekturunabhängig und portabel. Quellcode wird vom Java Compiler nicht in echten Maschinencode, sondern in sogenannten *Byte Code* übersetzt. Dieser Byte Code wird zur Laufzeit von einem Interpreter, der *Java Virtual Machine* (JVM) ausgeführt. Java-Programme können auf jeder Maschine laufen, welche über eine JVM verfügt.
- Java eignet sich besonders gut für die Entwicklung verteilter Anwendungen, d.h. die Sprache verfügt bereits über Klassen, welche sie netzwerkfähig machen und einen Mechnismus zum Aufrufen entfernter Objekte.
- Java bietet die Möglichkeit, Programme über das Netz auf den Rechner eines Benutzers zu übertragen und dort z.B. im Browser auszuführen. Diese Applets laufen beim Benutzer in einer sogenannten *Sandbox*, einer JVM mit eingeschränkten Rechten.
- Java ist eine dynamische Sprache. Klassen können zur Laufzeit geladen und dynamisch instanziiert werden.
- Java ist eine robuste Sprache. Aufgrund starker Typisierung und dem Fehlen der aus C und C++ bekannten Zeigerarithmetik können häufige Programmierfehler dieser Sprachen ausgeschlossen werden. Java bietet außerdem eine leistungsfähige Ausnahmebehandlung (*Exception Handling*) und nimmt dem Programmierer alle Aufgaben der Speicherverwaltung ab.

Als Implementierungsbasis für die Agenten wurde Java hauptsächlich aus zwei Gründen gewählt. Die *„Write once, run everywhere“* Philosophie der Sprache erlaubt den Einsatz der Managementagenten auf einer Vielzahl unterschiedlicher Plattformen, ohne wesentliche Anpassungen am Code vornehmen zu müssen.

Ein weiterer Entscheidungsgrund für Java war die Netzwerkfähigkeit der Sprache. Zwar kommt statt RMI[6] - der Java eigenen Methode zur Objektkommunikation - CORBA zum Einsatz, doch läßt sich CORBA mit Java einfach implementieren und nutzen.

Ein weiterer Vorteil von Java ist die syntaktische Anlehnung an C und C++, welche ein schnelles Erlernen der Sprache ermöglicht.

Für die Implementierung des Agenten wurde das *Java Development Kit* (JDK) 1.1.6 [Sun98] verwendet.

[6] Remote Method Invocation

5.3.2 CORBA

Die *Common Object Request Broker Architecture* (CORBA) ist ein Ansatz, das objektorientierte Programmierparadigma auf verteilte Systeme zu übertragen. Verteilte Objekte werden nicht länger nur auf einem Rechner, sondern im Netz ausgeführt. Die Architektur erfüllt dabei genau die im RM-ODP formulierte Forderung nach Transparenz, d.h. die Verteiltheit der Objekte im Netz bleibt vom Anwender verborgen: die Interaktion durch gegenseitige Methodenaufrufe ist unabhängig von der zugrundeliegenden Netztechnologie, sowie von Rechner- und Betriebssystemarchitekturen, sowie der gewählten Programmiersprache.

CORBA wurde von einem internationalen Firmenkonsortium, der *Object Management Group* (OMG), entworfen und 1990 erstmals als *Object Management Architecture* (OMA) Guide veröffentlicht.

Die Objektmanagement-Architektur sieht dabei wie folgt aus:

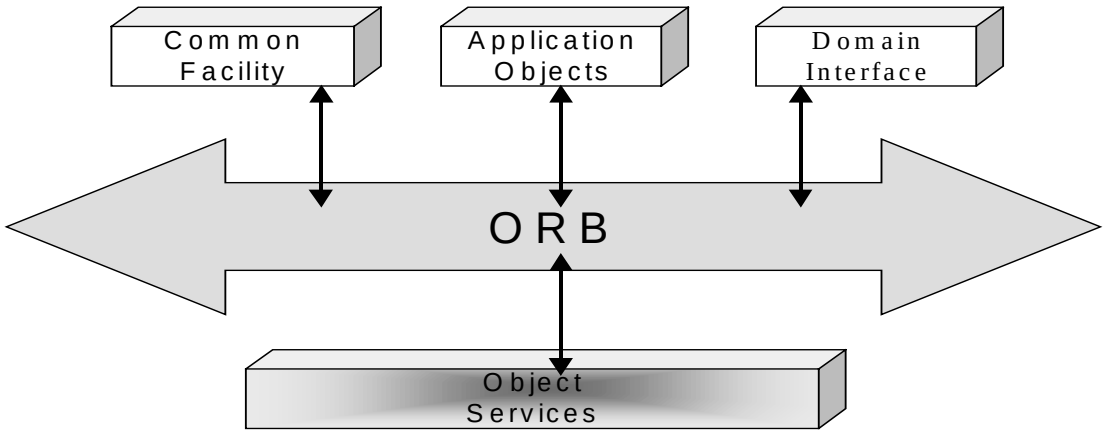


Abbildung 29 - Object Management Architecture (OMA)

CORBA ist eine Client/Server-Architektur. Zentraler Bestandteil ist deshalb der *Object Request Broker* (ORB) [OMG97a]. Der Broker stellt die Vermittlungsfunktionalität, also die Kommunikationsinfrastruktur zwischen Client- und Severobjekten (Application Objects) zur Verfügung. Der ORB ist für die Lokalisierung von entfernten Objekten, ebenso wie für den Methodenaufruf sowie die Übergabe von Argumenten und Ergebnissen bzw. Fehlern zuständig.

Aufbauend auf der Funktionalität des ORB sieht die Standardarchitektur weitere Komponeten vor [OMG97b]. Die *Object Services* (oder *CORBA Services*) decken Gebiete, wie Benennung von Objekten (*Naming Service*), Erzeugen und Vernichten von Objekten (*Lifecycle Service*) oder Ereignisvermittlung (*Event Service*) ab. Die in der Diplomarbeit entwickelten CORBA-Objekte fallen in den Bereich der *Application Objects*.

Zur Beschreibung von Objektschnittstellen definiert der Standard eine eigene Beschreibungssprache, die Interface Definition Language (IDL). Die Sprache ist an C++ angelehnt und beschreibt die öffentlich zugänglichen Attribute und Methoden eines Objektes.

Die Abbildung von IDL Datentypen auf Typen konkreter Programmiersprachen, wie etwa C++ oder Java ist von der OMG ebenfalls standardisiert worden; die Übersetzung der

Schnittstellen in Code wird von einem IDL-Compiler vorgenommen. Der Compiler erzeugt dabei auch sogenannte Stubs, Skeletons und Dyamic Invocation Interfaces (DII). Die Stubs und Skeletons dienen den eigenen Klassen als Schnittstellen zum statischen Aufrufen und Anbieten von CORBA-Services, DIIs bieten ermöglichen dynamische Aufrufe von Services, die zur Kompilierzeit nicht bekannt waren.

Da CORBA in seinem Aufbau die ODP-Anforderungen an ein verteiltes System vollständig erfüllt und somit eine leichte Abbildbarkeit des Email-Objektmodells verspricht, wurde es als Infrastruktur für verteiltes Management gewählt.

Für die Diplomarbeit wurde die ORB-Implementierung und die Entwicklungsumgebung *Visibroker for Java 3.0* [Visi97] des Herstellers *Visigenic* verwendet. Die Software generiert aus den IDL-Schnittstellen Java-Interfaces und erzeugt auch die Stubs und Skeletons in Java, so daß der gesamte Implementierungsprozeß mit dieser Sprache realisiert werden kann.

Für weitere Informationen zu CORBA wird auf einschlägige Literatur, wie z.B. [OHE96] verwiesen.

5.3.3 Die *Mobile Agent System* Architektur

Der entwickelte Agent und das zugehörige Applet basieren auf der in [Kemp98] entwickelten *Mobile Agent System Architecture* (MASA). Diese Architektur lehnt sich an die *Mobile Agent Interoperability Specification* der OMG [OMG97c] an.

MASA implementiert eine Agentenverwaltung, d.h. es bietet Dienste wie Agentenmanagement, Migration von Agenten über das Netz, sowie Naming und Location Services.

Als Kommunikationsinfrastruktur kommt CORBA zum Einsatz:

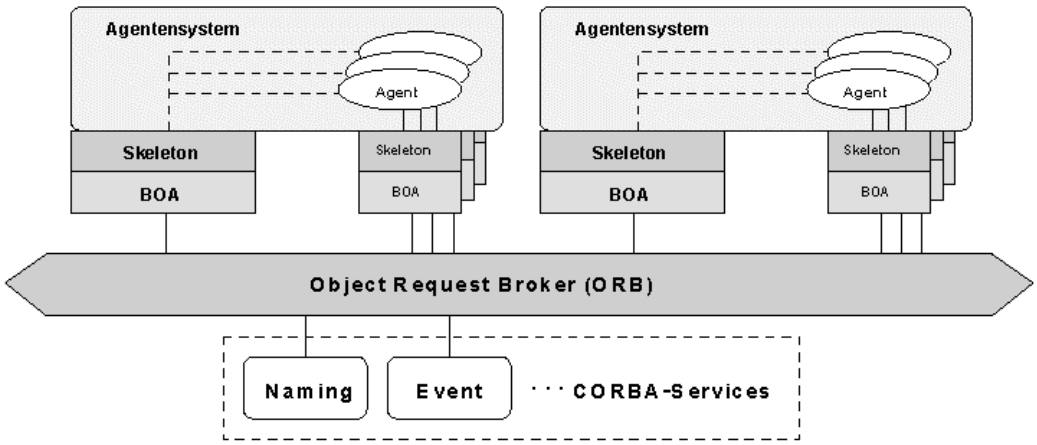


Abbildung 30 - MASA Agentensystem und Agenten (aus [Kemp98])

Alle Agenten haben die Möglichkeit, über eine HTML-Seite mit eingebettetem Applet eine graphische Benutzeroberfläche zur Verfügung zu stellen.

Um die vorhandene Agenteninfrastruktur und die problemlose Kommunikation mit anderen Agenten wie etwa dem bereits ewähnten Policy Enforcement Agenten kommunizieren zu können, wurde die *sendmail* Zugriffssteuerung als stationärer MASA-Agent implementiert.

5.4 Der Agent MTAAccessDB

5.4.1 Funktionalität des Agenten

Die Realisierung der Zugriffssteuerung erfolgt über den Agenten MTAAccessDB, welcher eine Subklasse von *MTASecurity* ist.

Folgende Methoden werden zur Verfügung gestellt:

- *InitializeDatabase*(): Diese Methode erzeugt eine neue, leere Zugriffsliste.
- *RetrieveDatabase*(): Diese Methode liefert die gesamte Zugriffsliste zurück.
- *GetKeyVal*(): Mit dieser Methode läßt sich gezielt nach Zugriffsrechten für einen oder mehrere Client-Systeme suchen.
- *SetKeyVal*(): Der Aufruf dieser Methode setzt oder löscht Zugriffsrechte.
- *UpdateDatabase*(): Diese Methode entspricht einem Commit, d.h. mit *SetKeyVal*() vorgenommene Änderungen werden erst nach dem Aufruf dieser Methode übernommen.

Die Implementierung des Agenten geschieht nun in mehreren Schritten:

- Die oben definierten Funktionen werden in IDL spezifiziert.
- Ein Compiler erzeugt aus den IDL-Schnittstellen Java Interfaces.
- Die Interfaces werden in eigenen Klassen implementiert.
- Der Code wird übersetzt und kann durch das Agentensystem gestartet werden.

Die nächsten Abschnitte beschreiben die einzelnen Schritte.

5.4.2 IDL-Schnittstelle

Um die IDL-Schnittstelle des Agenten automatisch generieren zu können, müssen die obigen Methoden noch genauer spezifiziert werden, d.h. Parameter und Rückgabewerte müssen typisiert werden. StP bietet hierzu als Werkzeug den Class Table Editor an.

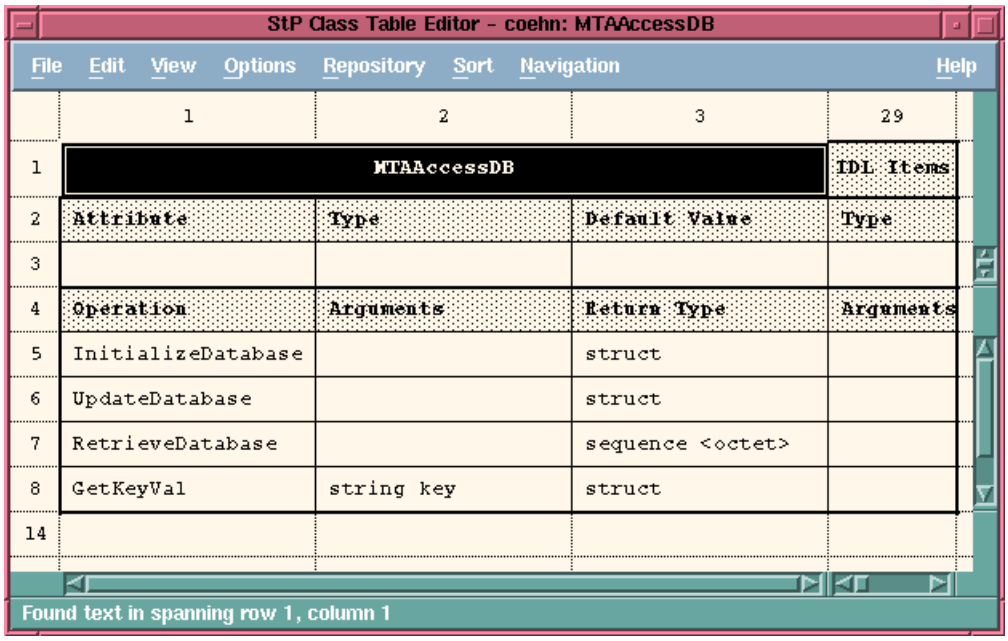


Abbildung 31 - Die Klasse MTAAccessDB im StP Klasseneditor

Anschließend kann die IDL-Schnittstelle durch StP automatisch generiert werden:

```
// StP -- created on Mon Jul 27 11:42:41 1998 for coehn@sunhegering7
from system coehn
#ifndef _MTAAccessDB_idl_
#define _MTAAccessDB_idl_
#include "/proj/fagent/masa_0.2/src/idl/AgentService.idl"
// stp class declarations
interface MTAAccess;
// stp class declarations end

struct retcode {long rc; string var;};
// stp class definition 1
interface MTAAccessDB:agent::AgentService
{
// stp class members
    retcode InitializeDatabase();
    retcode UpdateDatabase();
    string RetrieveDatabase();
    retcode GetKeyVal(in string key, in long part);
    retcode SetKeyVal(in string key, in string val);
// stp class members end
};
// stp class definition end

// stp footer
#endif
// stp footer end
```

Zeile 5 der Schnittstelle wurde nachträglich eingefügt, um von der MASA-Klasse AgentService erben zu können.

5.4.3 Implementierung in Java

Nach dem Erstellen IDL-Schnittstelle wird der *idl2java* Compiler der Visigenic Entwicklungsumgebung aufgerufen. Dieser erzeugt aus der IDL-Schnittstelle eine Vielzahl von Java Klassen.

Dazu gehört eine Java Interface Klasse als Pendant zur IDL-Schnittstelle, ebenso wie weitere unterstützende Klassen, wie etwa Stubs zum Aufrufen weiterer Clientobjekte, Skeletons zum Anbieten der eigenen Servermethoden und sogenannte *Tie-Classes*, welchen einen Workaround für den in Java fehlenden Polymorphismus darstellen.

Zusätzlich zur Interface Klasse erzeugt der Compiler auch eine Implementierungsklasse, in welcher sich bereits die Funktionsrümpfe finden, die nun mit Code ausgefüllt werden müssen.

Wie bereits in Abschnitt 5.3.1 erwähnt, verwendet Java sogenannte Packages zur Gruppierung mehrerer Klassen. Die Klassen des Agenten werden im Package *mtaaccess.agent* zusammengefaßt. Da Packagenamen global eindeutig sein müssen, wurde der Packagename mit der Präfix *de.unimuenchen.informatik.mnm* versehen.

Der Rumpf der Java Klasse *MTAAccessDBStationaryAgent.java* sieht also so aus:

```
package de.unimuenchen.informatik.mnm.mtaaccess.agent;

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

public class MTAAccessDBStationaryAgent extends
    de.unimuenchen.informatik.mnm.masa.agent.StationaryAgent
    implements MTAAccessDBOperations {

// Your work starts here...

}
```

Nachdem die Methoden des Agenten implementiert wurden, kann der Code durch den Java-Compiler *javac* übersetzt werden. Hierbei ist zu beachten, daß die Umgebungsvariablen und insbesondere der CLASSPATH entsprechend gesetzt sind.

5.4.4 Start des Agenten

Vor dem Einsatz muß der Agent konfiguriert werden. Dazu verfügt er über eine *Properties* Datei, in welcher die wichtigsten Einstellungen festgehalten werden:

- *dbDBM*: Diese Einstellung gibt den Pfad zur Datenbank mit den Zugriffsrechten an. Dateiendungen, wie .dbm, .pag oder .dir werden automatisch hinzugefügt.
- *dbSource*: Gibt den Pfad zur Textdatei an, aus welcher die Datenbank generiert wird.

- *makeMap*: Gibt den Befehl inclusive Parameter an, der aus der Textdatei die Datenbank generiert
- *sendmailCf*: Gibt den Pfad zur sendmail.cf Konfigurationsdatei an.

Ein Beispiel für eine Properties Datei, wie sie während der Entwicklung verwendet wurde:

```
dbDBM=/users/stud/coehn/proto/testing/access
dbSource=/users/stud/coehn/proto/testing/access.txt
makeMap=/usr/sbin/makedbm -
sendmailCf=/etc/sendmail.cf
```

Kann der Agent beim Starten seine Properties Datei nicht finden, so prüft er, ob die benötigten Dateien in Standardpfaden liegen. Ist dies nicht der Fall, wirft der Agent eine *Exception* und beendet sich.

Bevor der Agent jedoch gestartet werden kann, muß die MASA Agentenumgebung initialisiert werden. Dies geschieht in mehreren Schritten: Das Skript *vbjtcsh* startet den *osagent*, also die Visigenic Implementierung des ORB mit ausführlichen Meldungen in einer Shell. Anschließend wird der Naming Service durch Eingabe *von vbj –DOORBServices=CosNaming com.visgenic.vbroker.services.CosNaming.ExtFactory myFactory /tmp/naming.log* gestartet. Für den Start aller weiteren Services wurde ein entsprechendes Makefile angelegt. Duch die Eingabe von make client und wird das Agentensystem hochgefahren und der Agent ausgeführt.

Nachdem der Agent gestartet wurde, stellt er seine Methoden zur Manipulation der *sendmail* Konfiguration als CORBA Serverobjekt zur Verfügung. Ein Methodenaufruf aus einem Client sähe z.B. so aus:

```
org.omg.CORBA.Object objref= obj.connectToAgent(name1);
MTAAccessDB obj1= MTAAccessDBHelper.narrow(objref);
String erg = obj1.RetrieveDatabase();
```

Beim Start registriert sich der Agent außerdem beim Webserver der MASA-Umgebung, um seine Funktionalität auch über eine graphische Oberfläche zur Verfügung stellen zu können.

5.5 Das Management Applet

Das Applet zum MTAAccessDB erfüllt zwei wichtige Funktionen. Wird der Agent hauptsächlich zur Abwehr von SPAM-Mail benutzt, so erlaubt das Applet die komfortable Konfiguration der zu blockenden Benutzer oder Domains, ohne sich auf den jeweilige Mailhost einloggen zu müssen.

Wird der Agent für die Durchsetzung von Mail-Policies im Intranet eingesetzt, so dient das Applet hauptsächlich zur Kontrolle der Durchsetzung. Durch die Möglichkeit alle erlaubten IP-Adressen abfragen zu können, dient das Applet auch der Fehlersuche.

Nach dem Start des Agenten genügt es die HTML-Einstiegsseite des MASA-Systems aufzurufen, der Agent ist dort als MTAAccessDB verzeichnet. Durch entsprechenden Klick wird das Applet gestartet.

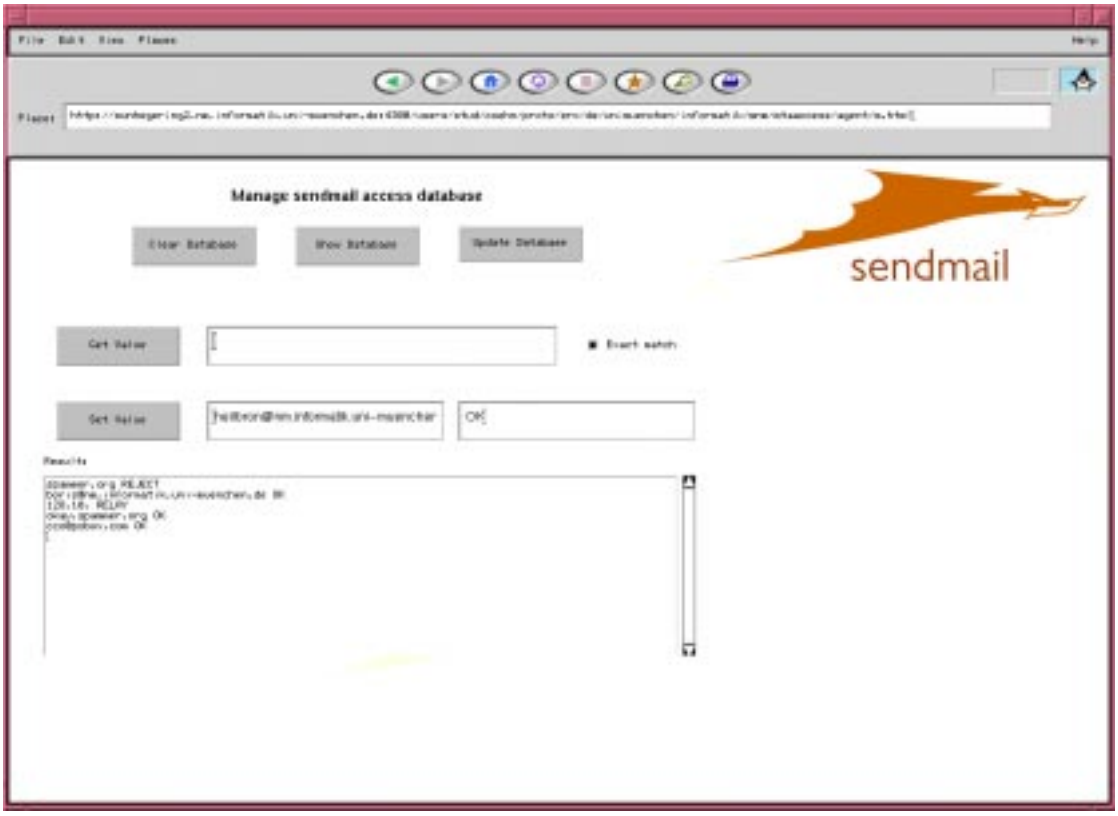


Abbildung 32 - Die Oberfläche des Management Applets

Die Bedienung des Applets erfolgt einfach durch das Drücken der entsprechenden Schaltflächen. Ergebnisse der jeweiligen Operation werden im Textfeld Results ausgegeben. Paßt der Text nicht ganz ins Textfeld, so kann er mit der Laufleiste am rechten Rand auf- und abgeschoben werden.

Die einzelnen Schaltflächen sind:

- *Clear Database*: Löscht die Liste mit den Zugriffsrechten und erzeugt eine neue.
- *Show Database*: Zeigt den kompletten Inhalt der Zugriffsliste im Results Fenster an.
- *UpdateDatabase*: Sämtliche Änderungen an den Zugriffsrechten (auch Clear Database) werden erst wirksam, nachdem UpdateDatabase ausgeführt wurde.
- *GetValue*: Erlaubt die Suche nach einem bestimmten Schlüssel in der Zugriffsliste. Je nachdem, ob Exact Match aktiv ist oder nicht, wird nach genauer Übereinstimmung oder Enthaltenseins-Beziehung gesucht.
- *SetValue*: Erlaubt es der Zugriffliste ein bestimmtes Schlüssel/Werte-Paar hinzuzufügen.

Das Applet ist wieder als Java Klasse implementiert, welches die Methoden des Agenten über CORBA-Aufrufe benutzt:

```
org.omg.CORBA.Object obj = initCORBA ("MTAAccessDB");
      _mtaaccess = MTAAccessDBHelper.narrow (obj);
```

Die Klassen, welche die Funktionalität der Schaltflächen implementieren sind dementsprechend sehr kurz:

```
// Der ClearButton, loescht die Datenbank
      button1 = new java.awt.Button();
      button1.setActionCommand("button");
      button1.setLabel("Clear Database");
      button1.setBounds(96,48,120,40);
      button1.setBackground(new Color(12632256));
      // Event Handler dazu
      button1.addActionListener (new ActionListener() {
      public void actionPerformed (ActionEvent e) {
        retcode rc;
        rc = _mtaaccess.InitializeDatabase();
        resultArea.setText ("");
        resultArea.append (rc.var);
      }
      });
      add (button1);
```

6 Zusammenfassung und Ausblick

In dieser Diplomarbeit wurde das Management der verteilten Anwendung Email untersucht. Der Schwerpunkt der Arbeit lag auf der Erstellung eines generischen Modells für verteiltes Email-Management, der Spezifikation von dienstspezifischen Managementoperationen, sowie der prototypischen Abbildung auf eine bestehende Managementarchitektur.

Hierzu wurden in einem ersten Schritt die Komponenten einer Email-Infrastruktur herausgearbeitet, sowie Managementanforderungen an diese Komponenten und den Dienst als Ganzes formuliert. In Kapitel 3 wurden bestehende Ansätze verschiedener Hersteller und Organisationen untersucht. Die Untersuchung hat gezeigt, daß existierende Managementlösungen stets nur Teilaspekte der Anforderungsanalyse abdecken. Insbesondere fehlt ein Informationsmodell, welches speziell auf die Verteilungsaspekte des Dienstes Email eingeht und die bisherigen Ansätze integriert.

Ein solches Informationsmodell wurde in dieser Arbeit durch Weiterentwicklung und Verfeinerung bestehender Ansätze zur Modellierung verteilter Systeme entwickelt. Das Modell ist objektorientiert und basiert auf den generischen Basisklassen des RM-ODP. Zur Modellierung einer verteilten Anwendung bietet ODP fünf verschiedene Sichten auf diese Anwendung. Für das Management entscheidend sind dabei vor allem der *Computational Viewpoint* und der *Engineering Viewpoint*.

Schwerpunkt der Arbeit war deshalb die Modellierung des Computational Viewpoint der Email-Infrastruktur und die Definition der Managementschnittstellen. Die Ausarbeitung der Schnittstellen erfolgte durch Untersuchung der Anforderungsanalyse und das Aufgreifen bereits bestehender Ansätze, vor allem aus dem OSI-Management. Das Resultat ist ein umfangreicher Katalog von Managementattributen und –methoden. Als Modellierungssprache wurde die mächtige objektorientierte OMT-Notation verwendet. Zur Erstellung der OMT-Diagramme wurde das CASE-Tool StP verwendet.

Das entwickelte Modell nimmt für sich in Anspruch, leicht auf bestehende Management-architekturen abbildbar zu sein. Um diesem Anspruch gerecht zu werden, wurde im Engineering Viewpoint eine mögliche Realisierung in einer OMA-Umgebung modelliert.

Abschließend wurde ein Ausschnitt des Modells aus dem Bereich des Sicherheits-managements als Agent implementiert. Als Kommunikationsinfrastruktur wurde CORBA gewählt, als Implementierungssprache diente Java. Dabei stellte sich CORBA als besonders geeignet zur Umsetzung der ODP-Modelle heraus. Die Eignung von Java zum Entwickeln von Managementagenten kann - aufgrund der Plattformunabhängigkeit und der Integration von Netzwerkfähigkeit - ebenfalls sehr positiv bewertet werden.

Der enstandene Agent setzt auf der am Lehrstuhl entwickelten MASA-Architektur auf. Zusammen mit weiteren Agenten, z.B. für das Management von Switches [Allg98] oder DHCP-Servern [Demm98] ist der Agent außerdem in eine Architektur für Policy-basiertes Management nomadischer Systeme [Radi98] integriert.

Ein interessanter Aspekt, der in dieser Arbeit nicht weiter verfolgt werden konnte, wäre die Definition ganz konkreter Policies für das Management einer Email-Infrastruktur und die Abbildung dieser Policies auf die einzelnen Komponenten mit Hilfe des *Policy Enforcement*

Services. Ebenfalls interessant wären Meta-Policies, die z.B. sich ergebende Sicherheitsfragen beim Einsatz einer verteilten Managementlösung beantworten.

Ein weiterer Bereich, der in dieser Arbeit ebenfalls ausgeklammert wurde, ist die Modellierung einer Management-Anwendung, die auf den in Kapitel 4 entwickelten Schnittstellen aufsetzt. So bietet ein MTA im Modell zwar eine Methode, welche seine Nachbarn liefert; wie jedoch eine Anwendung mit Hilfe dieser (und weiterer) Informationen die Topologie der Infrastruktur visualisiert, wurde offengelassen. Ein weiteres Beispiel für eine wichtige Managementapplikaton ist das Message Tracking. Das Modell stellt Informationen über Fluß und Status von Emails zur Verfügung, legt sich aber nicht auf eine konkrete Implementierung einer zugehörigen Anwendung fest.

Zieht man in Betracht, daß Studien zu modernen objektorientierten Managementarchitekturen bisher vor allem im universitären Bereich zu finden sind, so ist es sicher auch lohnenswert zu untersuchen, inwieweit das entwickelte Modell auf bereits bestehende bzw. historisch gewachsene Managementplattformen, wie etwa das im Kapitel 3 erwähnte Tivoli-System der BMW AG abgebildet werden kann.

Generell ist dazu zu sagen, daß die im Modell definierten Attribute häufig von skalarem Typ sind, also durchaus als MIB-Variablen in einem SNMP-Szenario definiert werden können. Als problematisch stellt sich dabei jedoch die Implementierung von Objektassoziationen oder –aggregationen heraus. Bei der Abbildung der Computational Objects auf Engineering Objects, also Agenten, welche über Kanäle interagieren, wird die Kommunikationsmöglichkeit zwischen diesen Agenten vorausgesetzt. Genau diese Voraussetzung ist bei SNMP und vielen proprietären Lösungen jedoch nicht gegeben.

Dennoch könnten einige Methoden z.B. als Tivoli-Monitore realisiert werden. Interessant sind hierbei z.B. die Übermittlung von Performanzdaten oder die Realisierung der entwickelten Alarme.

Ein Problem, welches beim Email-Management völlig unabhängig von der zugrunde liegenden Managementarchitektur auftritt, ist die Gewinnung der durch das Modell geforderten Daten. Solange die Komponenten der Infrastruktur keine Schnittstelle zum Abgreifen relevanter Informationen bieten, können manche Methoden entweder gar nicht, oder nur unter erheblichem Ressourcenverbrauch implementiert werden. Eine sich abzeichnende Lösung des Problems könnte die 1996 veröffentlichte *Application Respone Measurement* (ARM) Spezifikation [Tivo98b] sein, welche federführend von Tivoli Systems und Hewlett Packard entwickelt wurde und mittlerweile von einer Vielzahl von Herstellern unterstützt wird. Die Spezifikation greift den Transaktionsbegriff auf: Dabei wird festgehalten, daß gerade bei einer verteilten Anwendung, wie z.B. Email eine Transaktion auf Dienstebene nicht mehr eindeutig einer Transaktion auf Netzebene zugeordnet werden kann. Demenstprechend wird es für Fehler- und Leistungsmanagement als wichtig erkannt, Transaktionsdaten direkt an der überwachten Anwendung abgreifen zu können. Damit ergibt sich folgendes Bild für das ARM API:

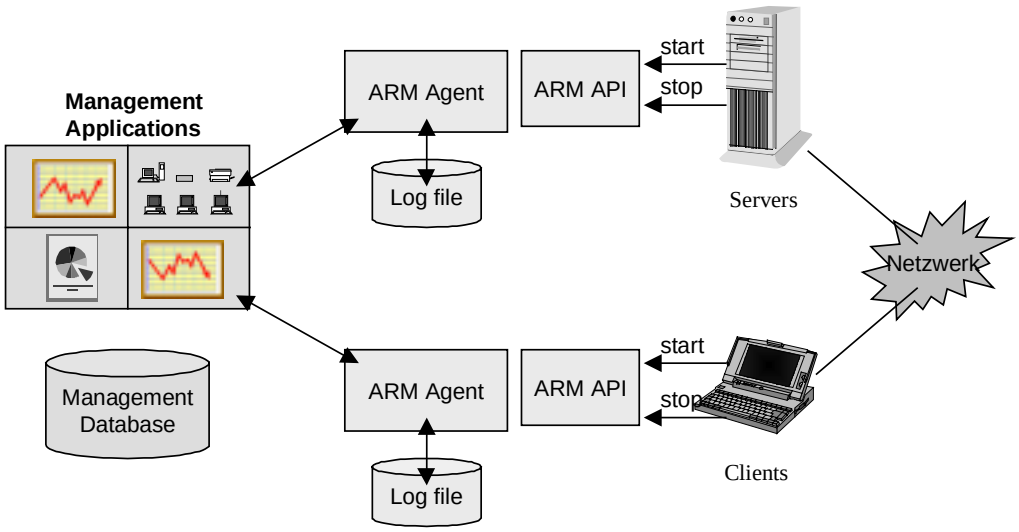


Abbildung 33 - ARM API (aus [Tivo98b])

Mit Hilfe der ARM-Spezifikation verfügen Anwendungen nun über eine Schnittstelle, mit denen sie folgende Informationen weitergeben können:

- Sind Transaktionen fehlgeschlagen?
- Wie lange dauert eine Transaktion?
- Wer benutzt die Anwendung und welche Transaktionen werden wie oft aufgerufen?
- Wo befinden sich Flaschenhälse?

Bezieht man die Spezifikation nun auf eine Email-Infrastruktur, so verfügt man nun über die Möglichkeit, Informationen über jede Transaktion einer Email ressourcenschonend direkt am vom überwachten Programm zu beziehen.

Das Implementieren der Managementklassen und das Realisieren von Message Tracking für eine ARM-fähige Email-Anwendung stellt einen interessanten Ausblick dar.

7 Anhang

7.1 OMT-Modelle

Auf den folgenden drei Seiten befinden sich die OMT-Modelle zum Informational Viewpoint, zum Computational Viewpoint und zum Engineering Viewpoint der Email-Infratstruktur.

7.2 Implementierungsbeispiele

Der Agent MTAAccessDB:

```
package de.unimuenchen.informatik.mnm.mtaaccess.agent;

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

public class MTAAccessDBStationaryAgent extends
    de.unimuenchen.informatik.mnm.masa.agent.StationaryAgent
    implements MTAAccessDBOperations {

  private Properties accessSource, agentProp;
  static final String agentPropertiesFile = "/users/stud/coehn/proto/accessAgent.prop";

  public MTAAccessDBStationaryAgent() throws Exception
      {
      // Konstruktor initialisiert den Agent. Hier muss
      // 1. Die Properties Datei eingelesen werden, dort
      //    steht die Source Datei, die Position der
      //    Datenbank access.db und die Position von
      //    sendmail.cf
      // 2. Die Source Datei eingelesen werden
      System.out.println ("Entering constructor\n");

      FileInputStream in = null;
      int ok = 0;
      accessSource = new Properties();
      agentProp = new Properties();
      try {
         in = new FileInputStream (agentPropertiesFile);
         agentProp.load (in);
      } catch (FileNotFoundException e) {
         System.out.println ("Could not access agent properties file "+
                             agentPropertiesFile + ";" + e.toString());
      } catch (IOException e) {
         System.out.println ("Error reading agent properties file "+
                             agentPropertiesFile + ";" + e.toString());
      } finally {
         if (in != null)
           in.close();
         if (agentProp.getProperty ("dbDBM") == null) {
           System.out.println ("Assuming default database location /etc/mail/acces");
           agentProp.put ("dbDBM", "/etc/mail/access");
         }
         if (agentProp.getProperty ("dbSource") == null) {
           System.out.println ("Assuming default database source location /etc/mail/acces");
           agentProp.put ("dbSource", "/etc/mail/access");
         }
         if (agentProp.getProperty ("sendmailCf") == null) {
           System.out.println ("Assuming default sendmail configuration /etc/sendmail.cf");
           agentProp.put ("sendmailCf", "/etc/sendmail.cf");
         }
         if (agentProp.getProperty ("makeMap") == null) {
           System.out.println ("Will try '/usr/bin/makemap hash' to compile database");
           agentProp.put ("makeMap", "/usr/bin/makemap hash");
         }
      }

      System.out.println ("Config Reader ok, reading database\n");
      // Ok, Config eingelesen, jetzt access Source einlesen
      try {
         ok = 0;
         in = new FileInputStream (agentProp.getProperty ("dbSource"));
         accessSource.load (in);
         ok = 1;
      } catch (FileNotFoundException e) {
         System.out.println ("Could not access database source file "+
                             agentProp.getProperty ("dbSource")+ ";" +
                            e.toString() + "\nCreating new one.");
      }
      catch (IOException e) {
         System.out.println ("Could not read database source file "+
```

```
                        agentProp.getProperty ("dbSource")+ ";" +
                       e.toString() + "\nGiving up.");
      throw new Exception ("Unable to read "+agentProp.getProperty ("dbSource"));
    } finally {
      if (in != null)
        in.close();
      if (ok == 0) {
        try{
          FileOutputStream out = new FileOutputStream (agentProp.getProperty ("dbSource"));
          out.close();
        } catch (IOException e) {
          System.out.println ("Unable to create "+agentProp.getProperty ("dbSource")+
                              "\nGiving up.");
          throw new Exception ("Unable to create "+agentProp.getProperty ("dbSource"));
        }
      }
    }

    // Ok, jetzt noch checken, ob makemap da ist.
    StringTokenizer st = new StringTokenizer (
                         agentProp.getProperty ("makeMap"));
    String firstToken = st.nextToken();
    File mm = new File (firstToken);
    System.out.println ("Ok, checking for makemap: "+firstToken);
    if (!mm.isFile() || !mm.canRead())
      throw new Exception ("makemap not found, giving up.");
    System.out.println ("Constructor done!\n");
  }


public retcode InitializeDatabase()
  {
    // Create scratch database
    accessSource = new Properties();
    return new retcode (0, "Ok.");
  }

public retcode UpdateDatabase()
  {
    // Die Update Methode schreibt die Source DB
    // zurueck und erzeugt mittels des in makeMap
    // gespeicherten Befehls eine neue Datenbank
    // Rueckgabe: retcode.rc = 0, alles ok,
    //                            makeMap Ausgabe in var
    //            retcode.rc = 1, I/O Fehler,
    //                            Exception in var
    //            retcode.rc = 2, Fehler bei Ausfuehrung von makeMap,
    //                            makeMap Ausgabe in var
    //            retcode.rc = 3. Fehler beim Spawnen, Exception in var

    FileWriter out = null;
    StringBuffer s = null;
    String elem = null;
    Process child = null;

    try {
      out = new FileWriter (agentProp.getProperty ("dbSource"));
      Enumeration accessDBEnum = accessSource.propertyNames();
      while (accessDBEnum.hasMoreElements()) {
        elem = (String)accessDBEnum.nextElement();
        s = new StringBuffer();
        s.append (elem);
        s.append (" ");
        s.append (accessSource.getProperty (elem));
        s.append ("\n");
        out.write (s.toString());
      }
      if (out != null)
        out.close();
    } catch (IOException e) {
      return new retcode (1, e.toString());
    }

    // Ok, Source weggeschrieben, jetzt makeMap aufrufen
    s = new StringBuffer("/bin/sh -c ");
    s.append (agentProp.getProperty ("makeMap") + " ");
    s.append (agentProp.getProperty ("dbDBM") + " ");
    s.append ("<" + agentProp.getProperty ("dbSource"));
    //s.append (" 2>&1");
```

```
    System.out.println ("Spawning " + s.toString());
    try {
      child = Runtime.getRuntime().exec(s.toString());
      InputStream cmdOut = child.getInputStream();
      InputStreamReader r = new InputStreamReader (cmdOut);
      BufferedReader in = new BufferedReader (r);
      StringBuffer lines = new StringBuffer();
      String line;
      while ((line = in.readLine()) != null)
        lines.append (line+"\n");
      if (child.waitFor() != 0)
        return (new retcode (2, lines.toString()));
      else
        return (new retcode (0, lines.toString()));
    } catch (IOException e) {
      return (new retcode (3, e.toString()));
    } catch (InterruptedException e) {
      return (new retcode (3, e.toString()));
    }
  }

public String RetrieveDatabase()
  {
    // Ganze Datenbank als String schicken
    String s = "";
    String elem = null;
    Enumeration accessDBEnum = accessSource.propertyNames();
    while (accessDBEnum.hasMoreElements()) {
      elem = (String)accessDBEnum.nextElement();
      s = s.concat (elem);
      s = s.concat (" ");
      s = s.concat (accessSource.getProperty (elem));
      s = s.concat ("\n");
    }
    return s;
  }

public retcode GetKeyVal(String key, int part)
  {
    // Falls part = 0, exakten Match,
    // ansonsten contains match
    // Rueckgabe: retcode.rc = 0, kein Treffer
    //            retcode.rc = x, x Treffer, Ergebnis in var
    StringBuffer s = new StringBuffer();
    String elem, elemlower, keylower = null;
    String var = new String();
    retcode rcode = new retcode(0, null);
    int matches = 0;
    if (part == 0) {
      var = accessSource.getProperty (key);
      if (var != null) {
        rcode.rc = 1;
        rcode.var = var;
      }
      return (rcode);
    }

    Enumeration accessDBEnum = accessSource.propertyNames();
    keylower = new String (key);
    keylower = keylower.toLowerCase();
    while (accessDBEnum.hasMoreElements()) {
      elem = (String)accessDBEnum.nextElement();
      elemlower = elem.toLowerCase();
      // Jetzt Containment pruefen
      if (elemlower.indexOf (keylower) != -1) { // Match !
        s.append (elem);
        s.append (" ");
        s.append (accessSource.getProperty (elem));
        s.append ("\n");
        matches++;
      }
    }
    if (matches > 0) {
      rcode.rc = matches;
      rcode.var = s.toString();
    }
    return rcode;
  }
```

```
  public retcode SetKeyVal(String key, String val)
    {
      if (key == null || val == null)
        return new retcode (1, "Nothing to do!");
      else if (key.equals(""))
        accessSource.remove (key);
      else
        accessSource.put (key, val);
      return new retcode(0, "Ok.");
    }

  public void cleanUp()
    {
    }

}
```

Das Management-Applet:

```
package de.unimuenchen.informatik.mnm.mtaaccess.agent;

/*
        A basic extension of the java.applet.Applet class
 */



import java.awt.*;
import java.awt.event.*;
import java.applet.*;


public class mta extends de.unimuenchen.informatik.mnm.masa.agent.AgentApplet
{

  private MTAAccessDB _mtaaccess;

  public void init()
    {
      // CORBA Object Request Broker initialisieren
      // und versuchen, den Agenten zu erreichen.
      try {
        org.omg.CORBA.Object obj = initCORBA ("MTAAccessDB");
        _mtaaccess = MTAAccessDBHelper.narrow (obj);
      }
      catch (java.net.MalformedURLException e) {
        resultArea.append ("Oops: " + e.toString());
      }
      catch (java.io.IOException e) {
        resultArea.append ("Oops: " + e.toString());
      }


      // Controls initialisieren
      setLayout(null);
      setSize(645,491);
      label1 = new java.awt.Label("Manage sendmail access database",Label.CENTER);
      label1.setBounds(132,0,360,40);
      label1.setFont(new Font("Helvetica", Font.BOLD, 14));
      add(label1);
      resultArea = new java.awt.TextArea();
      resultArea.setBounds(12,288,624,192);
      add(resultArea);
      label2 = new java.awt.Label("Result:");
      label2.setBounds(12,264,100,24);
      add(label2);

      // Der ClearButton, loescht die Datenbank
      button1 = new java.awt.Button();
      button1.setActionCommand("button");
      button1.setLabel("Clear Database");
      button1.setBounds(96,48,120,40);
      button1.setBackground(new Color(12632256));
      // Event Handler dazu
      button1.addActionListener (new ActionListener() {
```

```
  public void actionPerformed (ActionEvent e) {
    retcode rc;
    rc = _mtaaccess.InitializeDatabase();
    resultArea.setText ("");
    resultArea.append (rc.var);
  }
});
add (button1);

// Der Update Button, teilt Agenten mit, Source
// rauszuschreiben und neu zu compilieren
button2 = new java.awt.Button();
button2.setActionCommand("button");
button2.setLabel("Update Database");
button2.setBounds(408,48,120,36);
button2.setBackground(new Color(12632256));
// Event Handler dazu
button2.addActionListener (new ActionListener() {
  public void actionPerformed (ActionEvent e) {
    retcode rc;
    resultArea.setText ("");
    rc = _mtaaccess.UpdateDatabase();
    switch (rc.rc) {
    case 0:
      resultArea.append ("Ok, database updated!\n");
      break;
    case 1:
      resultArea.append ("I/O error while updating database:\n" +
                         rc.var);
      break;
    case 2:
      resultArea.append ("The database compiler returned an error:\n" +
                         rc.var);
      break;
    case 3:
      resultArea.append ("Error calling database compiler:\n" + rc.var);
    }
  }
});
add (button2);

// Der Get Value Button, holt sich einen Eintrag
// aus der Source Database des Agenten
button3 = new java.awt.Button();
button3.setActionCommand("button");
button3.setLabel("Get Value");
button3.setBounds(24,144,120,40);
button3.setBackground(new Color(12632256));
// Event Handler dazu
button3.addActionListener (new ActionListener() {
  public void actionPerformed (ActionEvent e) {
    retcode rc;
    resultArea.setText ("");
    rc = _mtaaccess.GetKeyVal (textField1.getText(),
                               checkbox1.getState() ? 0 : 1);
    if (rc.rc == 0)
      resultArea.append ("No matches.\n");
    else
      resultArea.append (rc.rc + "matche(s):\n" +
                         rc.var);
  }
});
add (button3);

// Der Set Button, traegt einen Eintrag in die
// Source Database ein oder loescht diesen
button4 = new java.awt.Button();
button4.setActionCommand("button");
button4.setLabel("Set Value");
button4.setBounds(24,216,120,40);
button4.setBackground(new Color(12632256));
// Event Handler dazu
button4.addActionListener (new ActionListener() {
  public void actionPerformed (ActionEvent e) {
    resultArea.setText ("");
    retcode rc;
    rc = _mtaaccess.SetKeyVal (textField2.getText(),
                               textField3.getText());
    resultArea.append (rc.var);
```

```
        }
      });
      add (button4);

      // Show Database holt die gesamte TextDB in
      // Sourceform und zeigt sie im Resultbutton an
      button5 = new java.awt.Button();
      button5.setActionCommand("button");
      button5.setLabel("Show Database");
      button5.setBounds(252,48,120,40);
      button5.setBackground(new Color(12632256));
      // Event Handler dazu
      button5.addActionListener (new ActionListener() {
        public void actionPerformed (ActionEvent e) {
          resultArea.setText ("");
          retcode rc;
          String db;
          db = _mtaaccess.RetrieveDatabase ();
          resultArea.append (db);
        }
      });
      add (button5);

      textField1 = new java.awt.TextField();
      textField1.setBounds(168,144,336,40);
      textField1.setFont(new Font("Helvetica", Font.PLAIN, 12));
      add(textField1);
      textField2 = new java.awt.TextField();
      textField2.setBounds(168,216,228,40);
      textField2.setFont(new Font("Helvetica", Font.PLAIN, 12));
      add(textField2);
      textField3 = new java.awt.TextField();
      textField3.setBounds(408,216,228,40);
      textField3.setFont(new Font("Helvetica", Font.PLAIN, 12));
      add(textField3);
      checkbox1 = new java.awt.Checkbox("Exact match");
      checkbox1.setBounds(528,144,100,40);
      add(checkbox1);
      checkbox1.setState(true);
    }

  //{{DECLARE_CONTROLS
  java.awt.Label label1;
  java.awt.TextArea resultArea;
  java.awt.Label label2;
  java.awt.Button button1;
  java.awt.Button button2;
  java.awt.Button button3;
  java.awt.Button button4;
  java.awt.Button button5;
  java.awt.TextField textField1;
  java.awt.TextField textField2;
  java.awt.TextField textField3;
  java.awt.Checkbox checkbox1;
}
```

8 Abkürzungsverzeichnis

API	Application Programming Interface
ARM	Application Response Measurement
ASN.1	Abstract Syntax Notation 1
AU	Access Unit
BEO	Basic Engineering Object
CASE	Computer Aided Software Engineering
CMRK	Corporate Message Recovery Key
CORBA	Common Request Object Broker Architecture
DA	Delivery Agent
DAP	Directory Access Protocol
DHCP	Dynamic Host Configuration Protocol
DII	Dynamic Invocation Interface
DL	Distribution List
DMTF	Desktop Management Task Force
DNS	Domain Name Service
EXPN	Expand-Option
FTP	File Transfer Protocol
HTTP	Hypertext Transfer Protocol
IDA	Institutionen för Datavetenskap
IDL	Interface Definition Language
IETF	Internet Engineering Taskforce
ISO	International Standardization Organization
ITU	International Telecommunication Union
JDK	Java Development Kit
JMAPI	Java Management API
JVM	Java Virtual Machine
MADMANG	Mail and Directory Management Group
MASA	Mobile Agent System Architecture

MASIF	Mobile Agent System Interoperability Facilities
MD	Managementdomäne
MHE	Message Handling Environment
MHS	Message Handling System
MIB	Management Information Base
MO	Managed Object
MS	Message Store
MTA	Message Transfer Agent
MTS	Message Transfer System
NOS	Nomadische Systeme
NS	Nachrichtenspeicher
OMA	Object Management Architecture
OMG	Object Management Group
OMT	Object Modeling Technique
OO	Objektorientiert
ORB	Object Request Broker
OSI	Open Systems Interconnection
QOS	Quality of Service
RMI	Remote Method Invocation
RM-ODP	Reference Model for Open Distributed Processing
SMTP	Simple Mail Transfer Protocol
SNMP	Simple Network Management Protocol
SPAM	Spiced Pork and Ham
STP	Software Through Pictures
TMN	Telecommmunications Management Network
UA	User Agent
UCE	Unsolicited Commercial Email
UML	Unified Modeling Language
VRFY	Verify-Option
WBEM	Web Based Enterprise Management
WWW	World Wide Web

9 Literaturverzeichnis

[Allg98] Peter Allgeyer, „Entwurf einer Managementschnittstelle für einen PC-basierten Switch“, Diplomarbeit, Technische Universität München, August 1998

[Aoni96] Aonix, *Software through Pictures/Object Modeling Technique – Getting Started with StP/OMT Release 3.0*, Februar 1996

[Aoni96a] Aonix, *Software through Pictures/Object Modeling Technique –Creating OMT Models Release 3.0*, Februar 1996

[Aoni97] Aonix, *StP Core – Fundamentals of StP Release 2.4*, September 1997

[ArGo98] Ken Arnold, James Gossling, *The Java Programming Language, Second Edition*, Addison-Wesley, 1998

[BMW98] Gabriele Wappler, „BMW Systems Management, Beschreibung der HP Openmail Monitore“, Interner Bericht, März 98

[CCT96] Deokjai Choi, Taesang Choi, Adrian Tang, „Issues in Enterprise E-Mail Management“, In *IEEE Communications Magazine*, April 1996

[CDS98] Control Data Systems Inc., „IntraStore Server 98“, Technischer Bericht, http://intrastore.cdc.com/www, 1998

[CoAl97] Bryan Costales, Eric Allman, *sendmail, Second Edition*, O’Reilly, 1997

[Coeh97] Christian Coehn, „Entwicklung eines SNMP-Agenten zur Auswertung und Weiterleitung von Syslog-Meldungen“, Fortgeschrittenenpraktikum, Ludwig-Maximilians-Universität München, November 1997

[Demm98] Simone Demmel, „Implementierung eines CORBA-basierten Managementagenten für DHCP Server“, Fortgeschrittenenpraktikum, Ludwig-Maximilians-Universität München, 1998

[ErJo97] Bruce Ernst, Gordon Jones, „Internet Draft: Message Tracking MIB“, Technischer Bericht, IETF, 1997

[Flan97] David Flanagan, *Java in a Nutshell, Second Edition*, O’Reilly, 1997

[Gars98] Markus Garschhammer, „Entwicklung eines Managementkonzepts für LDAP-integrierte Verzeichnisdienste“, Diplomarbeit, Technische Universität München, August 1998

[GeKl98] R. Gellens, J. Klensin, „Internet Draft: Message Submission“, Technischer Bericht, IETF, Juli 98

[HeAb93] Heinz-Gerd Hegering, Sebastian Abeck, *Integriertes Netz- und Systemmanagement*, Addison Wesley, 1. Auflage, 1993

[Heil98a] Stephen Heilbronner, „Requirement for Policy-based Management of Nomadic Computing Systems“, In *Proceedings of the 8th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM 98)*, Newark, DE, USA, Oktober 1998

[Joyn97] Ian Joyner, „Open Distributed Processing: Unplugged!“, http://homepages.tig.com.au/~ijoyner/ODPUnplugged.html, 1997

[Kemp98] Bernhard Kempter, „Entwurf eines JAVA/CORBA-basierten Mobilen Agenten“, Diplomarbeit, Technische Universität München, August 1998

[KeNe97c] Alexander Keller, Bernhard Neumair, „Using ODP as a Framework for CORBA-based Distributed Applications Management“, In *Proceedings of the IFIP/IEEE Joint International Conference on Open Distributed Processing (ICODP) and Distributed Platforms (ICDO)*, Toronto, CAN, DE, Mai 1997

[Lotu98] Lotus Inc., „Lotus Domino Management Solutions“, White Paper, http://www.lotus.com/home.nsf/welcome/administration, 1998

[M3200] „Maintenance: Telecommunications Management Network --- TMN Services: Overview“, Technischer Bericht, ITU-T, Oktober 1992

[Maur95] Johann Maurer, „Entwicklung eines auf standardisierten Management-konzepten basierenden Anwendungsmanagements am Beispiel von X.400, Diplomarbeit, Technische Universität München, Mai 1995

[Muel98] Tobias Müller, „CORBA-basiertes Management von UNIX-Workstations mit Hilfe von ODP-Konzepten“, Diplomarbeit, Technische Universität München, Februar 1998

[Nets98] Netscape Inc., „Netscape Mission Control“, White Papier, http://www.netscape.com/missioncontrol/index.html, 1998

[OHE96] Robert Orfali, Dan Harkey, Jeri Edwards, *The Essential Distributed Objects Survival Guide*, John Wiley and Sons, Inc., 1996

[OMG97a] OMG, „A Discussion of the Object Management Architecture“, Technischer Bericht, Januar 1997

[OMG97b] OMG, „CORBAservices: Common Object Servies Specification“, Technischer Bericht, November 1997

[OMG97c] GMD Fokus, „Mobile Agent System Interoperability Facilities Specification“, Technischer Bericht, Object Management Group, 1997

[Radi98] Igor Radisic, „Konzeption eines Policy-basierten Konfigurations-managements für nomadische Systeme in Intranets“, Diplomarbeit, Ludwig-Maximilians-Universität München, August 1998

[RFC1035] P. Mockapetris, „Domain Names, Implementation and Specification“, Technischer Bericht, IETF, November 1987

[RFC1521] N. Borenstein, N. Freed, „MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies“, Technischer Bericht, IETF, September 1993

[RFC1522] K. Moore, „MIME (Multipurpose Internet Mail Extensions) Part Two: Message Header Extensions for Non-ASCII Text, Technischer Bericht, IETF, September 1993

[RFC1565] S. Kille, N. Freed, „Network Services Monitoring MIB“, Technischer Bericht, IETF, Januar 1994

[RFC1566] S. Kille, N. Freed, „Mail Monitoring MIB“, Technischer Bericht, IETF, Januar 1994

[RFC2015] M. Elkins, „MIME Scurity with Pretty Good Privacy (PGP)“, Technischer Bericht, IETF, Oktober 1996

[RFC2248] N. Freed, S. Kille, „Network Services Monitoring MIB“, Technischer Bericht, IETF, Januar 1998

[RFC2249] N. Freed, S. Kille, „Mail Monitoring MIB“, Technischer Bericht, IETF, Januar 1998

[RFC706] Jonathan B. Postel, „On the Junk Mail Problem“, RFC706, Internet Activities Board, November, 1975

[RFC821] Jonathan B. Postel, „Simple Mail Transfer Protocol“, Technischer Bericht, IETF, August 1982

[RFC822] Jonathan B. Postel, „Standard for the Format of ARPA Network Text Messages“, Technischer Bericht, IETF, August 1982

[Rose96] Marshall T. Rose, „The Simple Book: An Introduction to Networking Management“, Prentice Hall 1996

[Rum91] James Rumbaugh, *Object-Oriented Modeling and Design*, Prentice Hall, 1991

[Schi98] Christian Schiller, „Evaluierung von JMAPI als Basis integrierter Managementanwendungen“, Diplomarbeit, Technische Universität München, August 1998

[Slo94] Morris S. Sloman, *Network and Distributed Systems Management*, Addison Wesley, 1994

[Sun97] Sun Microsystems Inc., „The Java Management API (JMAPI) Specification“, White Paper, http://java.sun.com/products/JavaManagement/index.html, 1997

[Sun98] Sun Microsystems Inc., „The Source for Java Technology“, http://www.javasoft.com, 1998

[Tivo98a] Tivoli Systems Inc., „Tivoli – The Power to Manage“, http://www.tivoli.com, 1998

[Tivo98b] Tivoli Systems Inc., „Tivoli and Application Management“, White Paper, http://www.tivoli.com/o_products/html/body_map_wp.html, 1998

[Visi97] Visigenic, *Visibroker for Java Reference Manual 3.0*, September 1997

[X400] „Message Handling Services: Message Handling System and Service Overview“, Recommendation X.400, International Telecommunication Union, März 1993

[X402] „Message Handling Systems: Overall architecture“, Recommendation X.402, International Telecommunication Union, September 1992

[X460] „Message Handling Systems (MHS) Management: Model And Architecture“, Recommendation X.460, International Telecommunication Union, April 1995

[X467] „Message Handling Systems (MHS) Management: Message transfer agent entity“, Recommendation X.460, International Telecommunication Union, April 1995

[X721] „Information Technology – Open Systems Interconnection – Structure of Management Information: Definition of Management Information“, Recommendation X.721, International Telecommunication Union, 1992

[X733] „Information Technology – Open Systems Interconnection - Systems Management: Alarm Reporting Function“, Recommendation X.733, International Telecommunication Union, 1992

[X736] „Information Technology – Open Systems Interconnection – Systems Management: Security Alarm Reporting Function“, Recommendation X.736, International Telecommunication Union, 1992

[X901] „Reference Model Of Open Distributed Processing“, Recommendation X.901, International Telecommunication Union, Juni 1995