# INSTITUT FÜR INFORMATIK

**DER**

**LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN**

## Fortgeschrittenenpraktikum

# Implementierung des OMG Notificationservice für das MASA-Java-Agentensystem


| | |
|---|---|
| Bearbeiter: | Oliver Maul |
| Aufgabensteller: | Prof. Dr. H.-G. Hegering |
| Betreuer: | Stephen Heilbronner |
| | Boris Gruschke |

# *Inhalt*


**1 EINLEITUNG ..... 7**

1.1 AUFGABENSTELLUNG ..... 7
1.2 OMG NOTIFICATIONSERVICE ..... 7
*1.2.1 Komponenten ..... 8*
*1.2.2 Structured Events ..... 9*
*1.2.3 Event Filtering ..... 10*
*1.2.4 Quality of Service ..... 10*
1.3 MOBILE AGENT SYSTEM ARCHITECTURE (MASA) ..... 11
*1.3.1 Komponenten ..... 12*
*1.3.2 Kommunikation ..... 12*

**2 ARCHITEKTUR ..... 13**

2.1 OBJEKTMODELL ..... 13
2.2 DIE HILFSKLASSEN ..... 14
*2.2.1 AppFilter ..... 14*
*2.2.2 http ..... 14*
*2.2.3 NSOptions ..... 14*
*2.2.4 QoS ..... 14*
*2.2.5 IID ..... 14*
*2.2.6 EventQueueEx ..... 14*
2.3 PERSISTENZ ..... 15
2.4 QUEUES ..... 15
2.5 CONNECTIONS ..... 15
2.6 LEBENSZYKLUS DER CHANNEL OBJEKTE ..... 15
2.7 DIE STANDARD FILTER CONSTRAINT SPRACHE ..... 16
*2.7.1 Filter Objekt ..... 16*
*2.7.2 Constraint ..... 16*

**3 DIE AGENTENOBERFLÄCHE ..... 17**

3.1 FILTERVERWALTUNG ..... 17
3.2 TEST-PUSHCONSUMER ..... 18
3.3 TEST-PROXYSUPPLIER ..... 19

**4 INSTALLATION ..... 19**

4.1 DAS PROPERTYFILE ..... 20

**5 STAND DER IMPLEMENTIERUNG ..... 23**

5.1 VORAUSSETZUNGEN ..... 23
5.2 ANDERE NOTIFICATION SERVICE-IMPLEMENTIERUNGEN ..... 23
*5.2.1 DSTC CORBA Notification Service ..... 23*
*5.2.2 OOC ORBacus Notify ..... 23*

**ANHANG A: VERZEICHNISBAUM DES INSTALLATIONSPAKETS ..... 24**

**ANHANG B: IDL-QUELLTEXTE NACH SPEZIFIKATION DER OMG ..... 25**

IDL-QUELLTEXT MODUL „COSNOTIFICATION“ ..... 25
IDL-QUELLTEXT MODUL „COSNOTIFYCHANNELADMIN“ ..... 27
IDL-QUELLTEXT MODUL „COSNOTIFYCOMM“ ..... 31
IDL-QUELLTEXT MODUL „COSNOTIFYFILTER“ ..... 32

**ANHANG C: IDL-QUELLTEXT DES NOTIFICATIONSERVICEAGENT ..... 34**

**LITERATUR ..... 35**

# Abbildungsverzeichnis

Abbildung 1: Architektur des NotificationService (Quelle: [1], Seite 22) ..... 7
Abbildung 2: Struktur eines Structured Event (Quelle: [1], Seite 31) ..... 9
Abbildung 3: Übersicht über das MASA-System (Quelle: Homepage „Flexible Agent“) ..... 12
Abbildung 4: NotificationService-Agent Architektur ..... 13
Abbildung 5: Hauptfenster des Agenten ..... 17
Abbildung 6: Filter Admin Fenster ..... 18
Abbildung 7: Push Consumer Fenster ..... 18
Abbildung 8: Push Supplier Fenster ..... 19

# 1 Einleitung

## *1.1 Aufgabenstellung*

Die im Rahmen dieses Fortgeschrittenen-Praktikums zu bearbeitende Aufgabe bestand in der Realisierung einer Implementierung des CORBA NotificationService, wie er in [1] spezifiziert ist. Als Laufzeitumgebung war das an der Forschungs- und Lehreinheit entwickelte MASA Agentensystem vorgegeben ([3]).

## *1.2 OMG NotificationService*

Zur Einführung sollen zunächst die wichtigsten Eigenschaften des CORBA NotificationService dargestellt werden. Für eine ausführliche Darstellung sei auf [1], Kapitel 2 verwiesen.
Zweck des NotificationService ist die Bereitstellung eines Kommunikationsdienstes für verteilte Anwendungen. Die Kommunikation wird dabei durch Kanäle, den sogenannten Event Channels

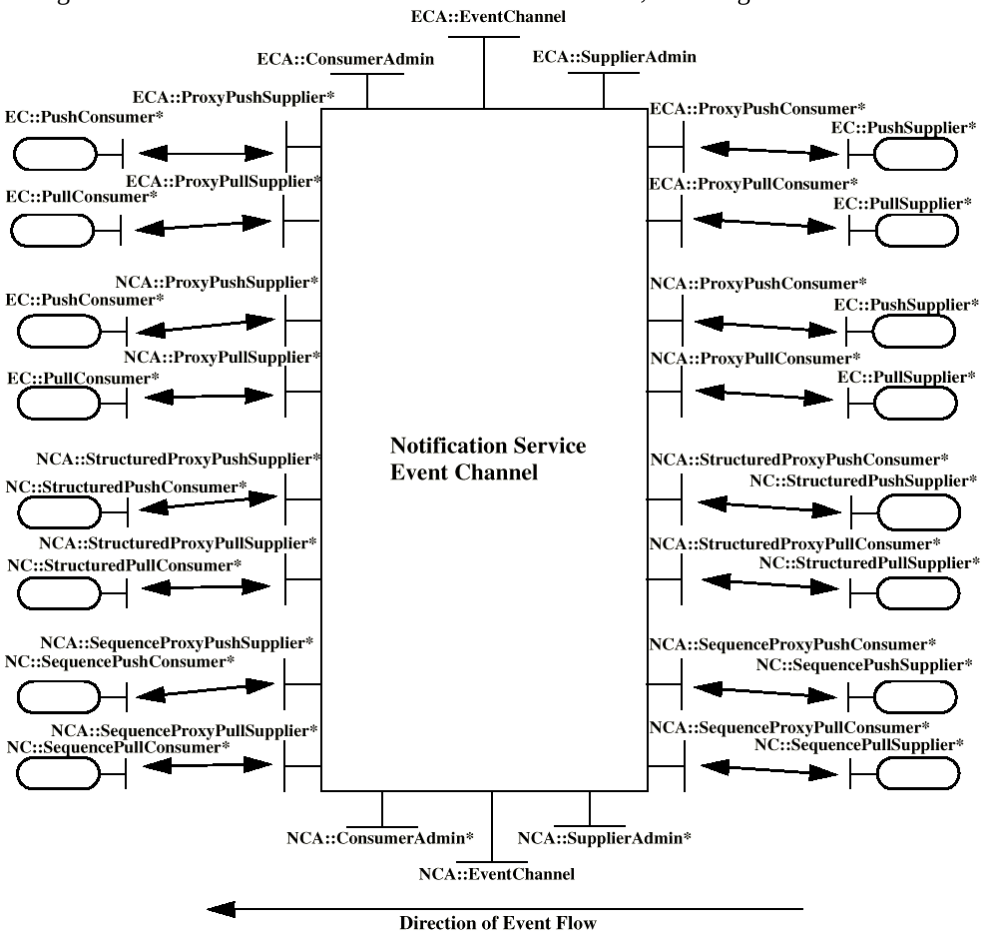


**Abbildung 1: Architektur des NotificationService (Quelle: [1], Seite 22)**

durchgeführt. Durch die Indirektion die der NotificationService in die Kommunikation bringt, werden ebenfalls folgende Eigenschaften mit eingebracht.

Der Mechanismus ist entkoppelt. Der Sender einer Nachricht muß den oder die Empfänger nicht kennen.
Die Kommunikation findet asynchron statt. Das bedeutet, daß der Sender einer Nachricht nicht blockiert wird bis die Nachricht beim Empfänger angekommen ist. Um den Verlust von Nachrichten dennoch zu verhindern, bietet der NotificationService diesbezüglich diverse QoS-Parameter an. Eine explizite Bestätigung des Empfangs ist aber nicht vorgesehen. Wird es gewünscht, muß es in der Anwendung implementiert werden.
Die Indirektion des NotificationService bringt außerdem die Möglichkeit des Multicasting mit sich. Sobald sich mehr als ein Empfänger an einen EventChannel anmeldet, werden alle Nachrichten die von Sendern in den Kanal geschickt werden nicht nur an einen Empfänger weitergeleitet. Vielmehr werden alle eingehenden Nachrichten dupliziert und an alle Empfänger gesendet.

### 1.2.1 Komponenten

Der NotificationService besteht aus einer Menge von Komponenten. Die Modularität trägt zur Skalierbarkeit bei. So kann der NotificationService über mehere Systeme verteilt werden.

Die Kernkomponente bildet der **EventChannel**. Der Name wurde aus dem EventService übernommen, was aber nicht bedeutet, daß beide Komponenten dieselbe Funktionalität besitzen. Allerdings wurde beim NoificationService auf "Abwärtskompatibilität" zum EventService geachtet. Das EventChannel-Interface besitzt sämtliche Methoden des EventChannel-Interface aus dem EventService.
Der EventChannel übernimmt die Pufferung von eingehenden Nachrichten und verteilen diese an seine ConsumerAdmins. Er kann Supplier- und ConsumerAdmins erzeugen. Dabei vergibt er jeweils eindeutige IDs, durch welche die Referenz des entsprechenden Objekts wiedergewonnen werden kann. Jeder EventChannel erzeugt zudem automatisch je einen Consumer- und einen SupplierAdmin mit der ID 0, die sogenannten DefaultAdmins.

Der **SupplierAdmin** dient der Gruppierung von Verbindungen zu den Sendern eines EventChannels. Dadurch ist es möglich gemeinsame Eigenschaften von Verbindungen im bündeln. Er kann Proxys erzeugen und vergibt dabei eindeutige IDs. Er leitet die Nachrichten, die er von seinen Proxys erhält an den EventChannel weiter nachdem er seine Filter erfolgreich angewendet hat.

Der **ConsumerAdmin** dient der Gruppierung von Verbindungen zu den Empfängern eines EventChannels. Dadurch ist es möglich gemeinsame Eigenschaften von Verbindungen im ConsumerAdmin zu bündeln. Er kann Proxys erzeugen und vergibt dabei eindeutige IDs. Er verteilt die Nachrichten, die er vom EventChannel erhält an seine ProxyObjekte.

Die **Proxys** dienen als Bindeglied zwischen dem NotificationService und dessen Clients. Proxys können Filter zugewiesen werden, die nur für die Verbindung gelten sollen.

**Filter** enthalten eine Menge von Constraints und werden an Admin- und Proxy-Objekte gebunden. Jedes Admin- und Proxy-Objekt mit assoziierten Filtern wendet diese auf jede Nachricht, die sie erreicht, an. Nur solche Nachrichten die alle Constraints mindestens eines Filters erfüllen, werden weitergeleitet. Alle anderen werden verworfen.

### 1.2.2 Structured Events

Das Structured Event, wie es im NotificationService verwendet wird, ist ein Kompromiß zwischen typisierter und untypisierter Eventkommunikation. Durch die feste Datenstruktur, die der Dienst und die Clients kennen, ist das Reagieren auf bestimmte Events also insbesondere das Filtern möglich.

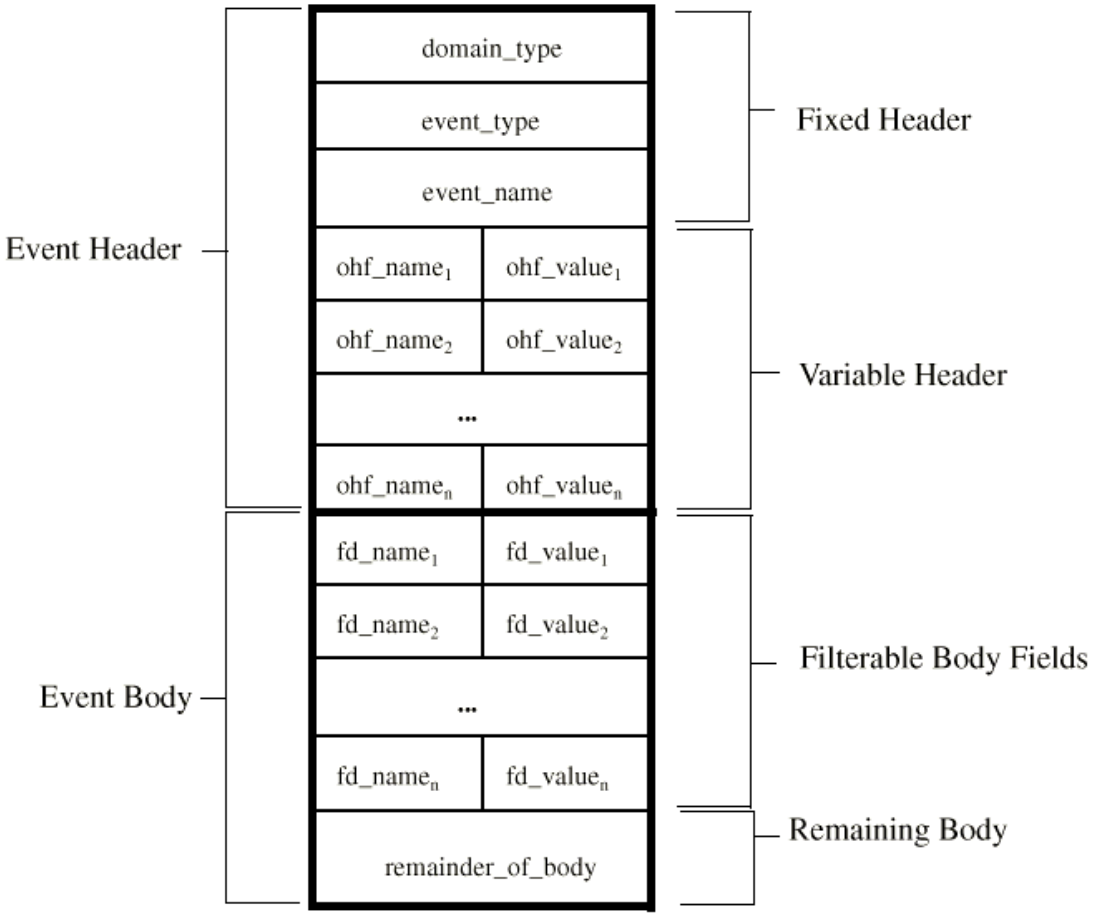


**Abbildung 2: Struktur eines Structured Event (Quelle: [1], Seite 31)**

Ein kleiner Teil des Headers, der Fixed Header, ist in jedem Event zwingend vorhanden. Darin werden nur die drei Felder domain_type, event_type und event_name angegeben. Alle drei Felder sind vom Typ String. Domain_type enthält den vertikalen Industriebereich, also z.B. finance oder telecom. event_type und event_name spezifizieren die Nachricht genauer innerhalb des domain_type. Der event_type kategorisiert das Event innerhalb der Domain eindeutig, z.B. "CommunicationsAlarm". Der variable Teil kann standardisierte per-Nachricht-QoS-Eigenschaften enthalten. Hier werden immer Schlüssel-Wert-Paare verwendet, z.B. ("EventReliability", "0"). Der Schlüssel ist vom Typ String, der Wert vom Typ Any.

Der Eventbody ist ebenso wie der Header zweigeteilt. Der variable Teil besteht ebenfalls aus Schlüssel-Wert-Paaren, die zur Filterung der Nachricht im NotificationService bestimmt sind. Der feste Teil besteht aus einem Feld vom Typ Any. Hierin ist das eigentliche Event, wie es im EventService verwendet wird.

### 1.2.3 Event Filtering

Der eindeutig wichtigste Fortschritt gegenüber dem EventService ist die Möglichkeit der Clients Events vorzuselektieren. Diese Eigenschaft wird durch Filter Objekte realisiert, die aus mindestens einem Constraint bestehen. Ein Constraint besteht aus einer Liste von Eventtypen für die das Constraint gelten soll und einem String mit einem boolschen Ausdruck, der in einer erweiterten Version der OMG Constraint Language beschrieben werden. Die OMG Constraint Language ist im Anhang des OMG Trader Service ([6]) als BNF abgedruckt.

Jedes Admin- und Proxyobjekt im NotificationService verfügt über Operationen zum manipulieren einer Liste von Filterobjekten.

Diese Filterobjekte können sich im gleichen Prozeß wie der Eventchannel befinden oder aber in einem getrennten Prozeß, evtl. sogar auf einem anderen System. Dabei ist allerdings der eventuell entstehende Mehraufwand zum Übertragen der Events zum Filterobjekten zu beachten. Denn jede Nachricht muß zur Evaluierung des Filters zum Filter übertragen werden.

Bei der Constraintsprache sei auf die Spezifikation [1], Kapitel 2.4 verwiesen. Sie erlaubt Vergleichsoperatoren (<, >, ==), Rechenoperatoren (+, -, *, /) und für den Zugriff auf das zu evaluierenden Events den Referenzoperator $.
Da ein Structured Event verschiedene komplexe Felder (Unions, Structs, Anys) besitzen kann, benötigt man noch einige Operatoren für den Zugriff auf solche Elemente.
Bei einem Struct kann mittels . auf dessen Elemente zugegriffen werden. Z.B. $event_name
Bei einem Array kann mittels [] auf dessen Elemente zugegriffen werden. Z.B. $.test[1]
Die Länge eines Arrays läßt sich mit dem speziellen Element _length ermitteln. Z.B. $.test._length
Bei einer Liste von Schlüssel-Wert-Paaren kann mittels () auf dessen Elemente zugegriffen werden. Z.B. $.Eventheader.variable_header(Priority)

Um den Namen des Events auf Gleichheit mit dem String "Probealarm" zu testen würde man folgendes Constraint verwenden:

```
$.EventHeader.fixed_header.event_name == ’ProbeAlarm‘ bzw.
                $event_name == 'ProbeAlarm'
```

Um zu prüfen ob die Priorität des aktuellen Events größer als 1000 ist würde man folgendes Constraint verwenden:

```
$.EventHeader.variable_header(Priority) > 1000
```

### 1.2.4 Quality of Service

Der NotificationService besitzt Standardinterfaces zur Kontrolle der QoS-Eigenschaften.

QoS-Eigenschaften existieren auf den verschiedenen Ebenen eines Channels: EventChannel, Admin, Proxy, Event.

**Reliability**

Mit der Connection Reliability wird festgelegt, ob versucht werden soll eine abgebrochene Verbindung erneut herzustellen. Die Event Reliability bestimmt, ob Events mit dem Best Effort Verfahren zugestellt werden sollen oder ob sie persistent gespeichert werden sollen bis sie zugestellt worden sind, um einen möglichen Verlust der Events bei einem Fehler im Channel zu vermeiden.

**Priority**

Durch eine Priorität, die jedem Event zugewiesen werden kann, ist es möglich die Zustellreihenfolge zu beeinflussen. Standardmäßig stellt der Notification Service die Events der Priorität entsprechend zu.
Dem QoS-Parameter Priority kann ein Wert zwischen –32767 und 32767 zugewiesen werden. Dieser Wert wird allen Events, die keine Priorität im Header enthalten, zugewiesen.
Die Priorität wird von der Order Policy, Discard Policy und den Filtern ausgewertet.

**Expiry Times**

Es gibt zwei Expiry Time Parameter. *StopTime* gibt einen Endzeitpunkt an, ab dem das Event nicht mehr zugestellt werden soll. *Timeout* gibt eine Zeitspanne an, wie lange das Event gültig ist.

**Earliest Delivery Time**

Mit diesem Parameter ist es möglich im voraus Events zu generieren, die aber erst zu einem bestimmten Zeitpunkt zugestellt werden sollen.

**Order Policy**

Hiermit ist die Zustellpolicy eines Proxys konfigurierbar. Es gibt folgende definierte Konstanten:
- *AnyOrder* Jede Zustellpolicy ist erlaubt
- *FifoOrder* Events werden in der Reihenfolge zugestellt in der sie ankommen
- *Priority Order* Events werden der Priorität entsprechend zwischengespeichert, so daß Events mit einer höheren Priorität vor solchen mit einer niedrigeren Priorität zugestellt werden.
- *Deadline Order* Events werden der Lebensdauer entsprechend zwischengespeichert, so daß Events mit einer kürzeren Lebensdauer vor solchen mit einer höheren Lebensdauer zugestellt werden.

**Discard Policy**

Dieser Parameter bestimmt in welcher Reihenfolge der Event Channel bei einem internen Speicherüberlauf Events verwerfen soll. Es gibt folgende definierte Konstanten:
- *Any Order* Jedes Event kann verworfen werden
- *Fifo Order* Das zuerst empfangene Event wird zuerst verworfen
- *Lifo Order* Das zuletzt empfangene Event wird zuerst verworfen
- *Priority Order* Events mit niedriger Priorität werden vor Events mit höherer Priorität verworfen
- *Deadline Order* Events mit kürzerer Lebensdauer werden vor Evens mit höherer Lebensdauer verworfen

## *1.3 Mobile Agent System Architecture (MASA)*

Zum Abschluß der Einführung soll nun noch die in [3] entwickelte „Mobile System Agent Architecture“ (MASA) kurz vorgestellt werden. MASA stellt eine plattformunabhängige Laufzeit- und Kommunikationsarchitektur für mobile Agenten dar. Abbildung 3 liefert eine Übersicht der MASA-Architektur.

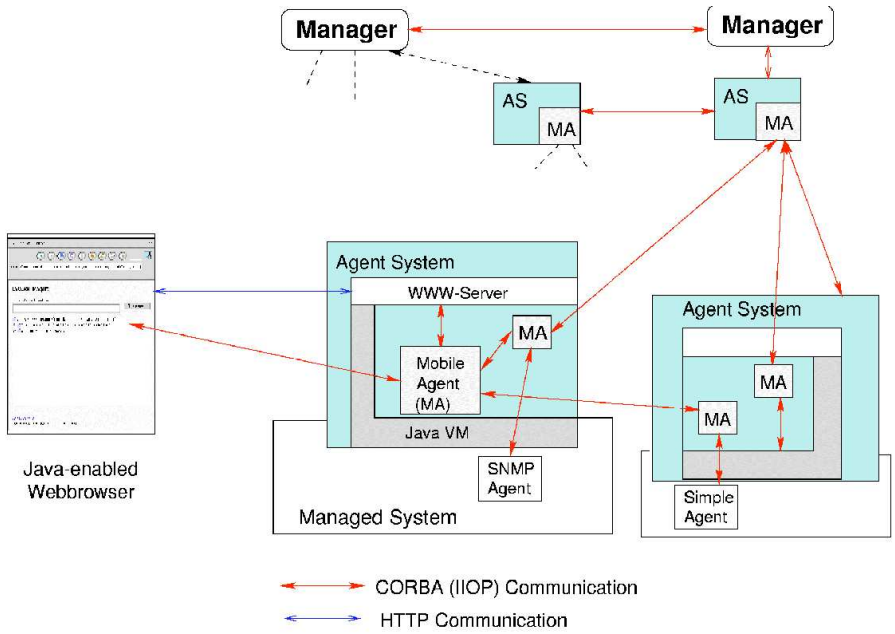


**Abbildung 3: Übersicht über das MASA-System (Quelle: Homepage „Flexible Agent“)**

### 1.3.1 Komponenten

Auf einem zu managenden Endsystem (*managed system*) wird ein Agentensystem (*agent system*) ausgeführt, das die Laufzeitumgebung für alle Agenten (*mobile agents*) auf einem Endsystem darstellt. Agentensystem und Agenten sind in der Programmiersprache Java implementiert.

Basis des Agentensystems und Abstraktionsschicht von plattformabhängigen Teilen des Endsystems ist die Java Virtual Machine, die Java-eigene Laufzeitumgebung. Das Agentensystem wird in der JVM ausgeführt. Die Agenten laufen innerhalb des Agentensystems ab, wodurch sie dessen Kontrolle unterworfen sind. Weiterhin ist ein eigener Webserver Teil eines jeden Agentensystems. Dieser dient als Dienstschnittstelle zwischen Browsern und dem Agentensystem bzw. den Agenten.

### 1.3.2 Kommunikation

Agenten können mit anderen Agenten kommunizieren und selbständig zu anderen Agentensystemen migrieren. Dabei erfolgt die gesamte Kommunikation über CORBA ([9]). Zur Steuerung eines Agenten muß dieser ein eigenes, wiederum in Java implementiertes, Applet mitbringen. Will ein Benutzer auf einen Agenten zugreifen, fordert er vom Webserver des Agentensystems über das HTTP-Protokoll das Applet des Agenten an. Dieses wird dann auf seinen Webbrowser übertragen und dort ausgeführt. Im folgenden kommuniziert dann das Applet via CORBA direkt mit dem zu steuernden Agenten.

# 2 Architektur

Die Klassenstruktur der Implementierung lehnt sich stark an die Modul Struktur der IDL Spezifikation an. Der stationäre Agent registriert sich unter dem Namen "NotificationService" beim MASA.

## *2.1 Objektmodell*

In der folgenden Abbildung wird die Modularisierung des Notification Service bzw. die des MASA Agenten dargestellt. Die Verbindungen stellen Beziehungen dar, die in der Implementierung durch Klassenattribute realisiert sind. Die Kardinalitäten geben an, wieviele Instanzen der Beziehungen zwischen den Klassen bestehen. Zwischen EventChannelFactory und EventChannel besteht z.B. eine 1:n Beziehung. Die EventChannelFactory hat also n Referenzen zu n EventChannels und jeder EventChannel hat eine Referenz auf die EventChannelFactory.

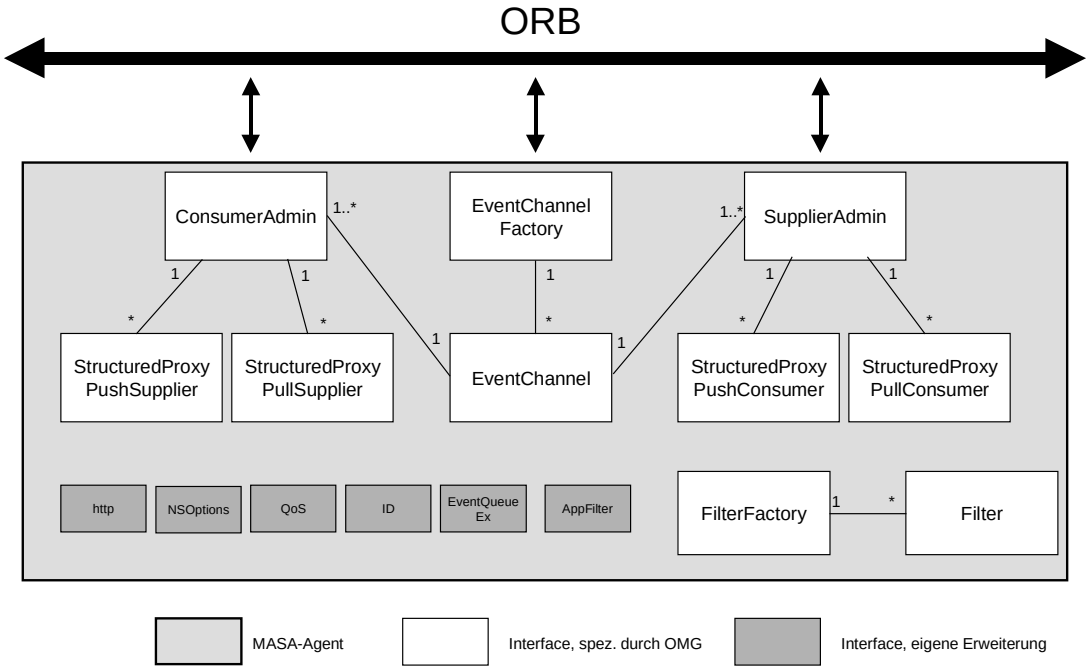


**Abbildung 4: NotificationService-Agent Architektur**

Zu jedem Interface der Spezifikation gibt es eine entsprechende Implementierungsklasse. Zusätzlich wurden einige zusätzliche Klassen gebildet, in denen z.B. die Queue (siehe 2.4) gekapselt wird. Die Beziehungen der Klassen http, NSOptions, QoS, ID, EventQueueEx und AppFilter wurden weggelassen, um die Übersichtlichkeit zu erhalten.
Die Filter erscheinen hier zwar eigenständig, werden aber nach der Erzeugung durch die FilterFactory und deren Konfiguration einem Objekt eines Channels zugewiesen. Dies liegt in der Zuständigkeit der Anwendung.

Die Abbildung 4 zeigt den gesamten NotificationService-Agenten. Er bildet eine Einheit und wird als stationärer Agent in einer MASA-Laufzeitumgebung ausgeführt. Mit dem Start des Agenten wird eine EventChannelFactory- und eine FilterFactory-Instanz erzeugt und je nach Konfiguration des Agenten beim NamingService registriert. Es ist auch möglich direkt über die Methoden getECF() und getFF() des Agenten diese Referenzen zu beziehen.

## *2.2 Die Hilfsklassen*

Zusätzlich zu den Klassen, die die Interfaces des NotificationService implementieren gibt es folgende Hilfsklassen.

### 2.2.1 AppFilter

AppFilter wird von den Admin-Objekten und Proxys verwendet um zu überprüfen ob ihre Filter ein gegebenes Event matchen oder nicht. AppFilter bekommt dazu das Event und eine Menge von Filtern mit. Es wendet dann der Reihe nach alle Filter auf das Event an. Ist mindestens ein Filter erfüllt, liefert die statische Methode app true ansonsten false.

### 2.2.2 http

Diese Klasse implementiert einen eigenen http-Server, der anfangs zu Testzwecken genutzt wurde und nun nur noch aus "historischen Gründen" vorhanden ist. Der Server ist über die Konfigurationsdatei steuerbar. Er liefert Informationen über die Eventchannel und die Anzahl der Events in ihnen.

### 2.2.3 NSOptions

Hier werden die aktuellen Konfigurationsparameter statisch vorgehalten, damit alle anderen Objekte direkten Zugriff darauf haben.

### 2.2.4 QoS

Die QoS-Klasse implementiert die Verwaltung der QoS des NotificationService. Es gibt unter anderem die Methoden get_qos, set_qos und validate_qos. Diese Methoden wurden hier gekapselt, um eine Redundanz zu verhindern, die durch mehrfache Implementierung in EventChannel, Admins und Proxys aufgetreten wäre.

### 2.2.5 IID

IID enthält ein Verzeichnis von IDs und Objektreferenzen. Diese Klasse wird in EventChannel und den Admin-Objekten verwendet um die IDs der erzeugten Admins bzw. Proxys zu verwalten.

### 2.2.6 EventQueueEx

Diese Klasse implementiert die Queue die von EventChannel, den Admins und auch den Proxys verwendet wird. Die Bedeutung der Queue wird in 2.4 erläutert.

## 2.3 Persistenz

Der Notification Service verlangt an einigen Stellen die Gewährleistung von dauerhaft gespeicherten Daten, so z.B. der QoS-Parameter Event Reliability. Aber auch die Speicherung der Channelstrukturen verlangt einen solchen Mechanismus.
Die Wahl ist auf die in Java vorhandenen Serialisierungsmöglichkeiten gefallen. Sie erlaubt es den Zustand von Objekten z.B. in eine Datei zu speichern. Jedes Objekt wird in eine eigene Datei gespeichert. Das Verzeichnis das hierzu verwendet wird ist über den Parameter dumpdir in der Konfigurationsdatei zu bestimmen. Beim Starten des Notification Service werden die Channel-Strukturen anhand der Dateien wiederhergestellt.

## 2.4 Queues

Die Events, die den Channel durchlaufen, werden von Proxy Consumern also stellvertretenden Verbrauchern entgegengenommen. Als erstes werden sie in der Eingangsqueue dieses Proxys eingereit. Genauso wie die Proxy Consumer besitzt jedes Channel-, Admin und Proxy Supplier-Objekt eine solche Eingangsqueue. Im Fall des Channel-Objekts ist diese Queue zugleich die zentrale Queue des gesammten Channels. Alle Events, die den Channel durchlaufen, gelangen in diese Queue.

Die Queues sollen dafür sorgen, daß die Threads die die Events einliefern möglichst wenig mit internen Aufgabe des Zielobjekts aufgehalten werden. Für weitere Aufgaben, die ein Objekt mit den Events zu erledigen hat, besitzt jedes Objekt einen oder mehrere Threads.

Die Queue, die in den genannten Objekten verwendet wird, ist in der Klasse EventQueueEx implementiert. Die Events werden dabei in einer einfach verketteten Liste organisiert. Jedes Listenelemente enthält neben dem Event, den Dateinamen in den es persistent gespeichert wird, falls es sich um die Queue des EventChannel handelt und es die QoS erfordert. Dieser Dateinamen wird durch den Java Ausdruck

```
String.valueOf(System.currentTimeMillis()+"."+(System.currentTimeMillis()%1000))
```

bestimmt. Weiter enthält jedes Listenelement den Zeitpunkt des Eintreffens und die IDs der ConsumerAdmins und Proxys die das Event noch nicht erhalten haben. Sobald keine IDs mehr vorhanden sind, wird das Listenelement und damit das Event aus der Queue entfernt. Ein eigener Discard-Thread sorgt alle 10 Sekunden dafür, daß Events der QoS entsprechend gelöscht werden, falls die StopTime oder der Timeout überschritten wurde.

## 2.5 Connections

Die Verbindungen zwischen den Proxyobjekten und den Consumern bzw. Suppliern können mittels des QoS-Parameters Connection Reliability persistent gespeichert werden. Dabei versucht das Proxy-Objekt eine Verbindung wieder aufzubauen, falls es durch z.B. Netzwerkproblemen zu einem Verbindungsfehler gekommen ist.

## 2.6 Lebenszyklus der Channel Objekte

Ein Channel existiert vom Anlegen an bis er explizit mit der Methode destroy() zerstört wird. Er überlebt auch Neustarts des Notification Service.

Administrations-Objekte haben einen Lebenszyklus vom Erzeugen mittels der Channel-Methode create_supplier_admin() bzw. create_consumer_admin() bis der Chanel zerstöt wird.

Proxy Objekte werden normalerweise von den Clients des NotificationService erzeugt und werden zerstört wenn sich der Client über die Methode disconnect() abmeldet. Um sich wieder mit dem Channel zu verbinden, muß ein neues Proxy-Objekt erzeugt werden.

## *2.7 Die Standard Filter Constraint Sprache*

Ein zentrales neues Merkmal des Notification Service gegenüber dem Event Service ist die Möglichkeit der Filterung von Events an vielen Stellen im Channel. So kann jedes Proxy-Objekt und die Consmer- sowie Supplier-Admin-Objekte mit Filtern ausgestattet werden.
Sind einem solchen Objekt mehrere Filter Objekte zugeordnet, werden deren Resultate über eine Disjunktion verknüpft. Die Ergebnisse der Constraints in einem Filter werden konjugiert.

### 2.7.1 Filter Objekt

Ein Filter Objekt repräsentiert eine Menge von Constraints. Als Eingabe erhält ein Filter ein Structured Event. Auf ein Event werden die mit dem Filter assoziierten Constraints angewendet und deren Ergebnisse konjungiert

### 2.7.2 Constraint

Die Constraints eines Filter Objekts werden in einer erweiterten Form der TCL (Trading Constraint Language) ausgedrückt. Diese Sprache erlaubt es einfache Bedingungen zu definieren, die erfüllt sein müssen um das entsprechende Event im Notification Service weiterzuleiten.

Ein einfaches Beispiel eines Constraint:

```
$event_type == 'CommunicationsAlarm'
```

Das Symbol $ steht für eine Referenz auf das aktuelle Event. Mit `.EventHeader.event_type` wird der Pfad zu der Variablen event_type im Event Header beschrieben. Der linke Teil dieses Constraints steht somit für den Event Typ des aktuellen Events, der hier mit der Konstanten `'CommunicationsAlarm'` auf Gleichheit verglichen wird. Wird diese Bedingung erfüllt liefert dieses Constraint den Wahrheitswert TRUE an das Filter-Objekt zurück.

# 3 Die Agentenoberfläche

Das GUI lehnt sich stark an die Struktur des Notification Service an. Die Hauptseite ist in zwei Hälften unterteilt. Links werden die EventChannels als Baum angezeigt. Rechts befindet ich ein Textfeld in dem Hinweise zu allen Aktionen ausgegeben werden. Jeder Knoten des Baums besitzt ein Kontextmenü, das über die rechte Maustaste erreicht werden kann. Die Menüs der einzelnen Knotentypen (EventChannel, Admin, Proxy) bieten entsprechende Funktionen zum Verwalten des Knotens. Das Menü des EventChannelFactory-Knotens bietet z.B. einen Menüpunkt "Create EventChannel". Ein EventChannel-Menü beinhaltet unter anderem einen "QoS..."-Menüpunkt.

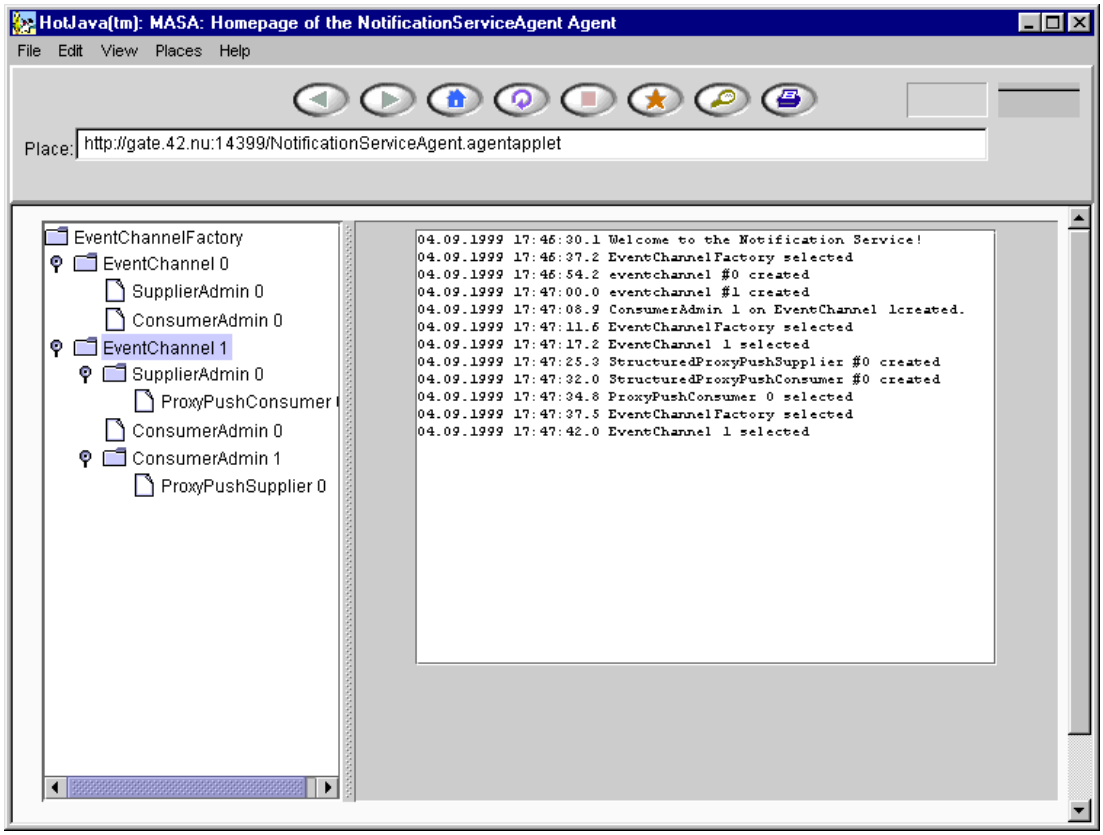


**Abbildung 5: Hauptfenster des Agenten**

Die Knoten des Baums sind nach dem Schema "<Knotentyp><KnotenID>" benannt.

## *3.1 Filterverwaltung*

Über die Filterverwaltung, die bei jedem Objekt eines Channels außer bei dem EventChannel-Objekt selbst verfügbar ist, könne Filter und deren Constraints konfiguriert werden. Jedes Constraint wird dabei auf syntaktische Korrektheit hin überprüft. Es können nur korrekte Constraints gespeichert werden.

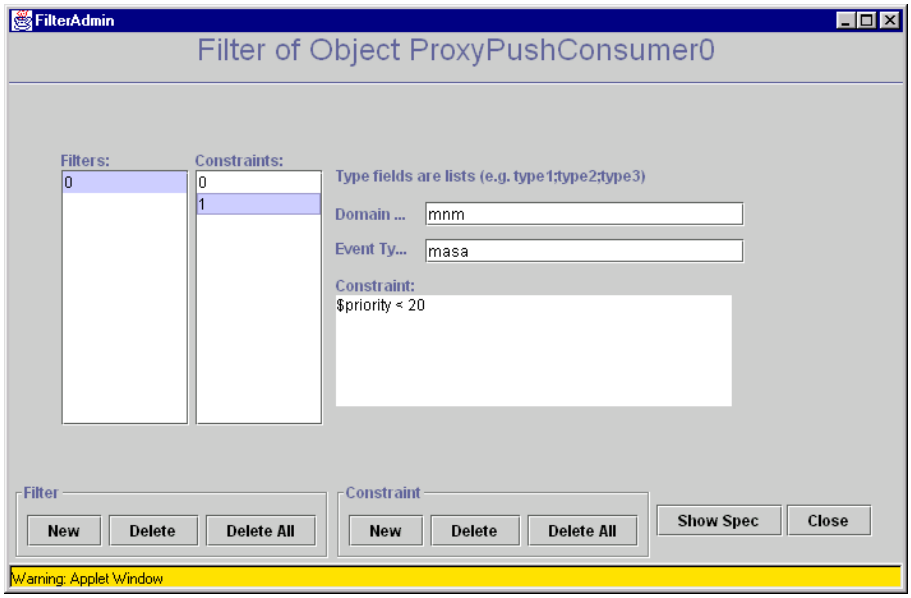


**Abbildung 6: Filter Admin Fenster**

## *3.2 Test-PushConsumer*

Der Test-PushConsumer erlaubt es einen PushConsumer zu simulieren, dessen Empfangsbereitschaft mittels dem "Suspend" Button ausgesetzt werden kann (Flußkontrolle).

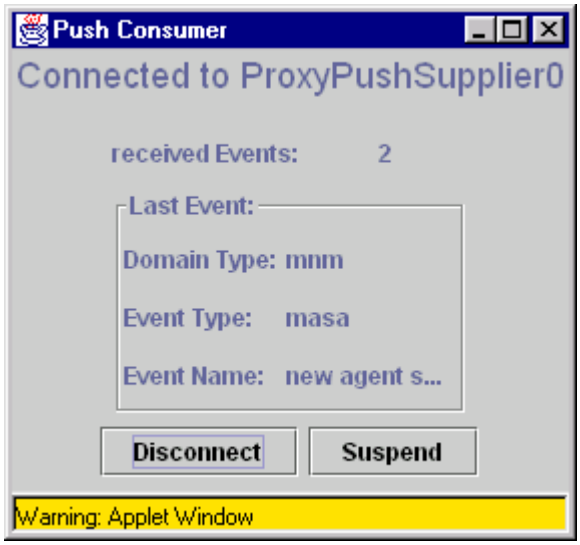


**Abbildung 7: Push Consumer Fenster**

Normalerweise würde ein PushConsumer alle Events die bei seinem ProxyPushSupplier ankommen automatisch von diesem zugestellt bekommen. Dadurch kann der PushConsumer überlastet werden.

Z.B. könnte dem Prozeß der Speicher ausgehen oder aber der PushConsumer könnte durch die Evententgegennahme keine Rechenzeit mehr für andere Tätigketen haben. Will ein Push Consumer eine solche Überlastung verhindern, kann er seinen ProxyPushSupplier über die Methoden suspend_connection() und resume_connection() steuern.

### 3.3 Test-ProxySupplier

Über den Test Push Supplier können Events in einen Notification Channel gepusht werden. Diese Events sind von einfacher Struktur. Es können nur die Event Header Felder Domain Type, Event Type und Event Name bestimmt werden. Als Event Body wird immer ein String mit dem Inhalt "" verwendet.

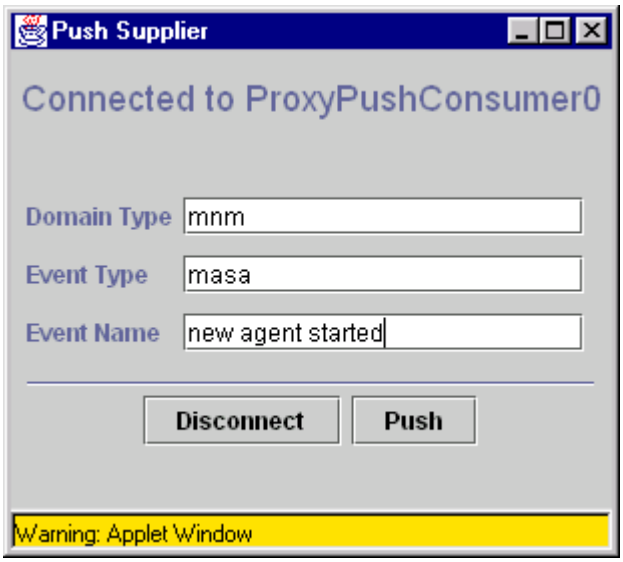


Abbildung 8: Push Supplier Fenster

# 4 Installation

Die Übersetzung und Installation geschieht ausschließlich durch die Makefiles. Um die Installation so einfach wie möglich zu gestalten, wurde die Makefilestruktur vom MASA übernommen

Als einzigster Parameter muß in der Datei Makefile.DEF folgende Zeile angepaßt werden:

```
export MASA_INSTALL_PATH    :=  $(HOME)/Masa/system/install
```

Die Variable MASA_INSTALL_PATH muß den Pfad zum Masa Installationsverzeichnis enthalten.

Die Installation des Notification Service geschieht in drei Schritten:

Mit dem Aufruf

```
make proddir
```

wird ein Produktionsverzeichnis angelegt in dem die Übersetzung durchgeführt wird.
Anschließend wird der Übersetzungsvorgang durch

make

gestartet. Hierbei werden die IDL Module in Java Klassen übersetzt und die class-Dateien erzeugt.
Nach dem Übersetzen werden die erzeugten Java Klassen in das Installationsverzeichnis von MASA kopiert und teilweise in Java Archiven zusammengefaßt.

make install; make install_doc

## 4.1 Das Propertyfile

Das Properyfile hat den Namen **genPropertiesFile.NotificationService,** liegt im Verzeichnis bin des NotificationService und enthält Properties, die das Verhalten des Dienstes beeinflussen.

Properties in dieser Datei haben stets das Format *Key = Value*.

Die Datei wird beim Installieren des Dienstes in das Installationsverzeichnis von MASA kopiert.

Die verfügbaren Properties werden nun hier kurz erläutert.

**`de.unimuenchen.informatik.mnm.NotificationService.notificationserviceiorfile`**

Gibt den Namen einer Ausgabedatei für die IOR der EventChannelFactory an. Sie kann von einem Client verwendet werden um die Objektreferenz der EventChannelFactory zu erzeugen.

**`de.unimuenchen.informatik.mnm.NotificationService.filterfactoryiorfile`**

Gibt den Namen einer Ausgabedatei für die IOR der FilterFactory an. Sie kann von einem Client verwendet werden um die Objektreferenz der FilterFactory zu erzeugen.

**`de.unimuenchen.informatik.mnm.NotificationService.dumpdir`**

Gibt den Namen eines Verzeichnisses an, in dem der Dienst Objekte speichert.

**`de.unimuenchen.informatik.mnm.NotificationService.name`**

Gibt den Namen an, unter dem sich die EventChannelFactory registriert.

**`de.unimuenchen.informatik.mnm.NotificationService.filterfactoryname`**

Gibt den Namen an, unter dem sich die FilterFactory registriert.

**`de.unimuenchen.informatik.mnm.NotificationService.bind`**

true/false: Soll am osagent registriert werden?

**`de.unimuenchen.informatik.mnm.NotificationService.namingservice`**

true/false: Soll am Namingservice registriert werden?

**de.unimuenchen.informatik.mnm.NotificationService.http**

true/false: Soll der "historische" interne http-Server gestartet werden?

**de.unimuenchen.informatik.mnm.NotificationService.httpport**

Portnummer, die der "historische" interne http-Server verwenden soll.

**de.unimuenchen.informatik.mnm.NotificationService.nsspecurl**

URL der Spezifikation des NotificationService. Sie wird im GUI des Agenten verwendet, um den Benutzer dorthin zu verweisen, wenn er fragen zur TCL hat.

**de.unimuenchen.informatik.mnm.NotificationService.debug**

0-4: Debuglevel. 0 = kein Debugoutput, 1 = nur EventChannelFactory, 2 = 1 + EventChannel, 3 = 2 + Supplier- und ConsumerAdmin, 4 = kompletter Debugoutput

# 5 Stand der Implementierung

Bisher sind vor allem die Grundfunktionalität sowie die Implementierung der Filter Constraint Sprache, die komplett umgesetzt ist, im Vordergrund gestanden. Die QoS-Parameter StartTime, discardPolicy sind noch nicht implementiert, d.h. deren Werte haben keinen Einfluß auf das Verhalten des Notification Service.

## *5.1 Voraussetzungen*

Der NotificationService-Agent verlangt, daß folgende Software Komponenten installiert sind:

- Java Development Kit 1.2
- Swing 1.1 Beta 2
- Orbacus 3.1.2
- JavaCC 1.1

## *5.2 Andere Notification Service-Implementierungen*

Während von mir der Notification Service als MASA Agent implementiert wurde, arbeiteten auch andere Firmen und Organisationen an einer solchen Implementierung.

### 5.2.1 DSTC CORBA Notification Service

Die DSTC ist ein australisches Joint Venture aus 24 Mitgliedern, das von der Regierung unterstützt wird. Zu den Mitgliedern gehört unter anderem Boing, Fujitsu und Sun Microsystems. Die erste Version brachte DSTC [5] im Herbst 1998 heraus. Mittlerweile gibt es die Version 1.1 als Evaluierungsversion zum Download. Es ist in Java geschrieben und beinhaltet eine graphische Administrationsoberfläche. Die aktuelle Version unterstützt bereits Event Filterung sowie das Push und Pull Modell.

### 5.2.2 OOC ORBacus Notify

Anfang 1999 kam auch OOC mit einer eigenen Implementierung ORBacus Notify [4] heraus. Es ist in C++ geschrieben, um eine maximale Performance zu gewährleisten, und unterstützt Event Filterung. Außer Structured Events werden auch Anys und Sequences bereitgestellt. Allerdings fehlt dem ORBacus Notify noch das Pull Modell, Typed Events und die administrativen Eigenschaften wie MaxConsumers, MaxSuppliers und MaxQueueLength.

# Anhang A: Verzeichnisbaum des Installationspakets

```
INSTALL
Makefile
Makefile.DEF
README
RELEASE_NOTES
notificationservice.properties
PRODUCTION.default/
        Makefile
src/
   NotificationService/
       AppFilter.java
       AsyncInputStream.java
       ConstraintParser
       ConsumerAdminImpl.java
       EventChannelFactoryImpl.java
       EventChannelImpl.java
       EventQueueEx.java
       ExtObjectInputStream.java
       ExtObjectOutputStream.java
       FilterFactoryImpl.java
       FilterImpl.java
       Http.java
       ID.java
       Makefile
       NSOptions.java
       NotificationServer.java
       QoS.java
       StructuredProxyPullConsumerImpl.java
       StructuredProxyPullSupplierImpl.java
       StructuredProxyPushConsumerImpl.java
       StructuredProxyPushSupplierImpl.java
       SupplierAdminImpl.java
       nsim.java
       ConstraintParser/
            ASTAddNode.java
            ASTAndNode.java
            ASTComponentNode.java
            ASTConstraint.java
            ASTDivNode.java
            ASTEQNode.java
            ASTExistComponentNode.java
            ASTExistIdentNode.java
            ASTExprInNode.java
            ASTExprTwiddleNode.java
            ASTFalseNode.java
            ASTGENode.java
            ASTGTNode.java
            ASTLENode.java
            ASTLTNode.java
            ASTMulNode.java
            ASTNENode.java
            ASTNotNode.java
            ASTNumberNode.java
            ASTOrNode.java
            ASTStringNode.java
            ASTSubtractNode.java
            ASTTrueNode.java
            Makefile
            Node.java
            SimpleNode.java
            SymbolTable.java
            TCL.jjt
            TCLParserWrapper.java
            TypeConverter.java
  NotificationServiceAgent/
       FilterAdm.java
       Makefile
       MyPullConsumer.java
       MyPullSupplier.java
       MyPushConsumer.java
       MyPushSupplier.java
       NSCreateEC.java
       NSQoS.java
       NotificationServiceAgentApplet.java
       NotificationServiceAgentStationaryAgent.java
       comboItem.java
       nsInfo.java
       idl/
          CosEventChannelAdmin.idl
          CosEventComm.idl
          CosNotification.idl
          CosNotifyChannelAdmin.idl
          CosNotifyComm.idl
          CosNotifyFilter.idl
          CosTrading.idl
          Makefile
          NotificationServiceAgent.idl
          TimeBase.idl
```

# Anhang B: IDL-Quelltexte nach Spezifikation der OMG

## *IDL-Quelltext Modul „CosNotification“*

```
module CosNotification {
    // The following two are the same, but serve different purposes.
    typedef CosTrading::PropertySeq OptionalHeaderFields;
    typedef CosTrading::PropertySeq FilterableEventBody;
    typedef CosTrading::PropertySeq QoSProperties;
    typedef CosTrading::PropertySeq AdminProperties;

    struct EventType {
        string domain_name;
        string type_name;
    };

    typedef sequence<EventType> EventTypeSeq;

    struct PropertyRange {
        CosTrading::PropertyName name;
        CosTrading::PropertyValue low_val;
        CosTrading::PropertyValue high_val;
    };

    typedef sequence<PropertyRange> PropertyRangeSeq;

    enum QoSError_code {
        UNSUPPORTED_PROPERTY,
        UNAVAILABLE_PROPERTY,
        UNSUPPORTED_VALUE,
        UNAVAILABLE_VALUE,
        BAD_PROPERTY,
        BAD_TYPE,
        BAD_VALUE
    };

    struct PropertyError {
        QoSError_code code;
        PropertyRange available_range;
    };

    typedef sequence<PropertyError> PropertyErrorSeq;

    exception UnsupportedQoS { PropertyErrorSeq qos_err; };
    exception UnsupportedAdmin { PropertyErrorSeq admin_err; };

    // Define the Structured Event structure
    struct FixedEventHeader {
        EventType event_type;
        string event_name;
    };

    struct EventHeader {
        FixedEventHeader fixed_header;
        OptionalHeaderFields variable_header;
    };

    struct StructuredEvent {
        EventHeader header;
        FilterableEventBody filterable_data;
        any remainder_of_body;
    }; // StructuredEvent

    typedef sequence<StructuredEvent> EventBatch;

    // The following constant declarations define the standard
```

```
// QoS property names and the associated values each property
// can take on. The name/value pairs for each standard property
// are grouped, beginning with a string constant defined for the
// property name, followed by the values the property can take on.

const string EventReliability = "EventReliability";
const short BestEffort = 0;
const short Persistent = 1;

const string ConnectionReliability = "ConnectionReliability";
// Can take on the same values as EventReliability

const string Priority = "Priority";
const short LowestPriority = -32767;
const short HighestPriority = 32767;
const short DefaultPriority = 0;

const string StartTime = "StartTime";
// StartTime takes a value of type TimeBase::UtcT when placed
// in an event header. StartTime can also be set to either
// TRUE or FALSE at the Proxy level, indicating whether or not the
// Proxy supports the setting of per-message stop times.

const string StopTime = "StopTime";
// StopTime takes a value of type TimeBase::UtcT when placed
// in an event header. StopTime can also be set to either
// TRUE or FALSE at the Proxy level, indicating whether or not the
// Proxy supports the setting of per-message stop times.

const string Timeout = "Timeout";
// Timeout takes on a value of type TimeBase::TimeT

const string OrderPolicy = "OrderPolicy";
const short AnyOrder = 0;
const short FifoOrder = 1;
const short PriorityOrder = 2;
const short DeadlineOrder = 3;

const string DiscardPolicy = "DiscardPolicy";
// DiscardPolicy takes on the same values as OrderPolicy, plus
const short LifoOrder = 4;

const string MaximumBatchSize = "MaximumBatchSize";
// MaximumBatchSize takes on a value of type long

const string PacingInterval = "PacingInterval";
// PacingInterval takes on a value of type TimeBase::TimeT


interface QoSAdmin {

QoSProperties get_qos();

void set_qos ( in QoSProperties qos)
    raises ( UnsupportedQoS );

void validate_qos (
        in QoSProperties required_qos,
        out PropertyRangeSeq available_qos )
    raises ( UnsupportedQoS );

}; // QosAdmin

// Admin properties are defined in similar manner as QoS
// properties. The only difference is that these properties
// are related to channel administration policies, as opposed
// message quality of service

const string MaxQueueLength = "MaxQueueLength";
// MaxQueueLength takes on a value of type long

const string MaxConsumers = "MaxConsumers";
```

```
    // MaxConsumers takes on a value of type long

    const string MaxSuppliers = "MaxSuppliers";
    // MaxSuppliers takes on a value of type long


    interface AdminPropertiesAdmin {

        AdminProperties get_admin();

        void set_admin (in AdminProperties admin)
            raises ( UnsupportedAdmin);

    }; // AdminPropertiesAdmin

};
```

## *IDL-Quelltext Modul „CosNotifyChannelAdmin“*

```
module CosNotifyChannelAdmin {

    exception ConnectionAlreadyActive {};
    exception ConnectionAlreadyInactive {};

    // Forward declarations
    interface ConsumerAdmin;
    interface SupplierAdmin;
    interface EventChannel;
    interface EventChannelFactory;

    interface ProxyConsumer :
            CosNotification::QoSAdmin,
            CosNotifyFilter::FilterAdmin {
        readonly attribute SupplierAdmin MyAdmin;

        CosNotification::EventTypeSeq obtain_subscription_types();

        void validate_event_qos (
                in CosNotification::QoSProperties required_qos,
                out CosNotification::PropertyRangeSeq available_qos)
            raises (CosNotification::UnsupportedQoS);

    }; // ProxyConsumer

    interface ProxySupplier :
            CosNotification::QoSAdmin,
            CosNotifyFilter::FilterAdmin {

        readonly attribute ConsumerAdmin MyAdmin;

        attribute CosNotifyFilter::MappingFilter priority_filter;
        attribute CosNotifyFilter::MappingFilter lifetime_filter;

        CosNotification::EventTypeSeq obtain_offered_types();

        void validate_event_qos (
                in CosNotification::QoSProperties required_qos,
                out CosNotification::PropertyRangeSeq available_qos)
            raises (CosNotification::UnsupportedQoS);

    }; // ProxySupplier



    interface ProxyPushConsumer :
            ProxyConsumer,
```

```
        CosNotifyComm::NotifyPublish,
        CosEventComm::PushConsumer {

    void connect_any_push_supplier (
            in CosEventComm::PushSupplier push_supplier)
        raises(CosEventChannelAdmin::AlreadyConnected);

}; // ProxyPushConsumer



interface StructuredProxyPushConsumer :
        ProxyConsumer,
        CosNotifyComm::StructuredPushConsumer {

    void connect_structured_push_supplier (
            in CosNotifyComm::StructuredPushSupplier push_supplier)
        raises(CosEventChannelAdmin::AlreadyConnected);

}; // StructuredProxyPushConsumer



interface ProxyPullSupplier :
        ProxySupplier,
        CosNotifyComm::NotifySubscribe,
        CosEventComm::PullSupplier {

    void connect_any_pull_consumer (
            in CosEventComm::PullConsumer pull_consumer)
        raises(CosEventChannelAdmin::AlreadyConnected);

}; // ProxyPullSupplier



interface StructuredProxyPullSupplier :
        ProxySupplier,
        CosNotifyComm::StructuredPullSupplier {

    void connect_structured_pull_consumer (
            in CosNotifyComm::StructuredPullConsumer pull_consumer)
        raises(CosEventChannelAdmin::AlreadyConnected);

}; // StructuredProxyPullSupplier



interface ProxyPullConsumer :
        ProxyConsumer,
        CosNotifyComm::NotifyPublish,
        CosEventComm::PullConsumer {

    void connect_any_pull_supplier (
            in CosEventComm::PullSupplier pull_supplier)
        raises(CosEventChannelAdmin::AlreadyConnected,
        CosEventChannelAdmin::TypeError );

}; // ProxyPullConsumer



interface StructuredProxyPullConsumer :
        ProxyConsumer,
        CosNotifyComm::StructuredPullConsumer {

    void connect_structured_pull_supplier (
            in CosNotifyComm::StructuredPullSupplier pull_supplier)
        raises(CosEventChannelAdmin::AlreadyConnected,
        CosEventChannelAdmin::TypeError );

}; // StructuredProxyPullConsumer
```

```
interface ProxyPushSupplier :
        ProxySupplier,
        CosNotifyComm::NotifySubscribe,
        CosEventComm::PushSupplier {

    void connect_any_push_consumer (
            in CosEventComm::PushConsumer push_consumer)
        raises(CosEventChannelAdmin::AlreadyConnected,
        CosEventChannelAdmin::TypeError );

    void suspend_connection()
        raises(ConnectionAlreadyInactive);

    void resume_connection()
        raises(ConnectionAlreadyActive);

}; // ProxyPushSupplier



interface StructuredProxyPushSupplier :
        ProxySupplier,
        CosNotifyComm::StructuredPushSupplier {

    void connect_structured_push_consumer (
            in CosNotifyComm::StructuredPushConsumer push_consumer)
        raises(CosEventChannelAdmin::AlreadyConnected,
        CosEventChannelAdmin::TypeError );

    void suspend_connection()
        raises(ConnectionAlreadyInactive);

    void resume_connection()
        raises(ConnectionAlreadyActive);

}; // StructuredProxyPushSupplier

typedef long ProxyID;
typedef sequence <ProxyID> ProxyIDSeq;

enum ClientType {
    ANY_EVENT,
    STRUCTURED_EVENT,
    SEQUENCE_EVENT
};

typedef long AdminID;
typedef sequence<AdminID> AdminIDSeq;

exception AdminNotFound {};
exception ProxyNotFound {};

interface ConsumerAdmin :
        CosNotification::QoSAdmin,
        CosNotifyComm::NotifySubscribe,
        CosNotifyFilter::FilterAdmin,
        CosEventChannelAdmin::ConsumerAdmin {

    readonly attribute AdminID MyID;
    readonly attribute EventChannel MyChannel;

    attribute CosNotifyFilter::MappingFilter priority_filter;
    attribute CosNotifyFilter::MappingFilter lifetime_filter;

    readonly attribute ProxyIDSeq pull_suppliers;
    readonly attribute ProxyIDSeq push_suppliers;

    ProxySupplier get_proxy_supplier (
            in ProxyID proxy_id )
```

```
        raises ( ProxyNotFound );

    ProxySupplier obtain_notification_pull_supplier (
            in ClientType ctype,
            out ProxyID proxy_id);

    ProxySupplier obtain_notification_push_supplier (
            in ClientType ctype,
            out ProxyID proxy_id);

}; // ConsumerAdmin

interface SupplierAdmin :
        CosNotification::QoSAdmin,
        CosNotifyComm::NotifyPublish,
        CosNotifyFilter::FilterAdmin,
        CosEventChannelAdmin::SupplierAdmin {

    readonly attribute AdminID MyID;
    readonly attribute EventChannel MyChannel;

    readonly attribute ProxyIDSeq pull_consumers;
    readonly attribute ProxyIDSeq push_consumers;

    ProxyConsumer get_proxy_consumer (
            in ProxyID proxy_id )
        raises ( ProxyNotFound );

    ProxyConsumer obtain_notification_pull_consumer (
            in ClientType ctype,
            out ProxyID proxy_id);

    ProxyConsumer obtain_notification_push_consumer (
            in ClientType ctype,
            out ProxyID proxy_id);

}; // SupplierAdmin

interface EventChannel :
        CosNotification::QoSAdmin,
        CosNotification::AdminPropertiesAdmin,
        CosEventChannelAdmin::EventChannel {

    readonly attribute EventChannelFactory MyFactory;

    readonly attribute ConsumerAdmin default_consumer_admin;
    readonly attribute SupplierAdmin default_supplier_admin;

    ConsumerAdmin new_for_consumers( out AdminID id );

    SupplierAdmin new_for_suppliers( out AdminID id );

    ConsumerAdmin get_consumeradmin ( in AdminID id )
        raises (AdminNotFound);

    SupplierAdmin get_supplieradmin ( in AdminID id )
        raises (AdminNotFound);

    AdminIDSeq get_all_consumeradmins();
    AdminIDSeq get_all_supplieradmins();

}; // EventChannel

typedef long ChannelID;
typedef sequence<ChannelID> ChannelIDSeq;

exception ChannelNotFound {};

interface EventChannelFactory {

    EventChannel create_channel (
            in CosNotification::QoSProperties initial_qos,
```

```
                in CosNotification::AdminProperties initial_admin,
                out ChannelID id)
            raises(CosNotification::UnsupportedQoS,
            CosNotification::UnsupportedAdmin );

        ChannelIDSeq get_all_channels();

        EventChannel get_event_channel ( in ChannelID id )
            raises (ChannelNotFound);

    }; // EventChannelFactory

};
```

## *IDL-Quelltext Modul „CosNotifyComm“*

```
module CosNotifyComm {

    exception InvalidEventType { CosNotification::EventType type; };

    interface NotifyPublish {

        void offer_change (
                in CosNotification::EventTypeSeq added,
                in CosNotification::EventTypeSeq removed )
            raises ( InvalidEventType );

    }; // NotifyPublish

    interface NotifySubscribe {

        void subscription_change(
                in CosNotification::EventTypeSeq added,
                in CosNotification::EventTypeSeq removed )
            raises ( InvalidEventType );

    }; // NotifySubscribe

    interface StructuredPushConsumer : NotifyPublish {
        // NOTE: If transactional semantics are desired, a transactional
        // variant of this interface should be implemented

        void push_structured_event(
                in CosNotification::StructuredEvent notification)
            raises(CosEventComm::Disconnected);

        void disconnect_structured_push_consumer();

    }; // StructuredPushConsumer

    interface StructuredPullConsumer : NotifyPublish {
        void disconnect_structured_pull_consumer();
    }; // StructuredPullConsumer

    interface StructuredPullSupplier : NotifySubscribe {
        // NOTE: If transactional semantics are desired, a transactional
        // variant of this interface should be implemented

        CosNotification::StructuredEvent pull_structured_event()
            raises(CosEventComm::Disconnected);
        CosNotification::StructuredEvent try_pull_structured_event(
                out boolean has_event)
            raises(CosEventComm::Disconnected);

        void disconnect_structured_pull_supplier();
```

```
    }; // StructuredPullSupplier

    interface StructuredPushSupplier : NotifySubscribe {
        void disconnect_structured_push_supplier();
    }; // StructuredPushSupplier

};
```

## *IDL-Quelltext Modul „CosNotifyFilter“*

```
module CosNotifyFilter {

    typedef long ConstraintID;

    struct ConstraintExp {
        CosNotification::EventTypeSeq event_types;
        string constraint_expr;
    };

    typedef sequence<ConstraintID> ConstraintIDSeq;
    typedef sequence<ConstraintExp> ConstraintExpSeq;

    struct ConstraintInfo {
        ConstraintExp constraint_expression;
        ConstraintID constraint_id;
    };

    typedef sequence<ConstraintInfo> ConstraintInfoSeq;

    struct MappingConstraintPair {
        ConstraintExp constraint_expression;
        any result_to_set;
    };

    typedef sequence<MappingConstraintPair> MappingConstraintPairSeq;

    struct MappingConstraintInfo {
        ConstraintExp constraint_expression;
        ConstraintID constraint_id;
        any value;
    };

    typedef sequence<MappingConstraintInfo> MappingConstraintInfoSeq;

    typedef long CallbackID;
    typedef sequence<CallbackID> CallbackIDSeq;

    exception UnsupportedFilterableData {};
    exception InvalidGrammar {};
    exception InvalidConstraint {ConstraintExp constr;};
    exception DuplicateConstraintID {ConstraintID id;};

    exception ConstraintNotFound {ConstraintID id;};
    exception CallbackNotFound {};

    exception InvalidValue {ConstraintExp constr; any value;};


    interface Filter {

        readonly attribute string constraint_grammar;

        ConstraintInfoSeq add_constraints (
                in ConstraintExpSeq constraint_list)
            raises (InvalidConstraint);

        void modify_constraints (
                in ConstraintIDSeq del_list,
```

```
                in ConstraintInfoSeq modify_list)
            raises (InvalidConstraint, ConstraintNotFound);

        ConstraintInfoSeq get_constraints(
                in ConstraintIDSeq id_list)
            raises (ConstraintNotFound);

        ConstraintInfoSeq get_all_constraints();

        void remove_all_constraint();

        void destroy();

        boolean match ( in any filterable_data )
            raises (UnsupportedFilterableData);

        boolean match_structured (
                in CosNotification::StructuredEvent filterable_data )
            raises (UnsupportedFilterableData);

        CallbackID attach_callback (
                in CosNotifyComm::NotifySubscribe callback);

        void detach_callback ( in CallbackID callback)
            raises ( CallbackNotFound );

        CallbackIDSeq get_callbacks();

};  // Filter


interface MappingFilter {

    readonly attribute string constraint_grammar;

    readonly attribute CORBA::TypeCode value_type;

    readonly attribute any default_value;

    MappingConstraintInfoSeq add_mapping_constraints (
            in MappingConstraintPairSeq pair_list)
        raises (InvalidConstraint, InvalidValue);

    void modify_mapping_constraints (
            in ConstraintIDSeq del_list,
            in MappingConstraintInfoSeq modify_list)
        raises (InvalidConstraint, InvalidValue,
            ConstraintNotFound);

    MappingConstraintInfoSeq get_mapping_constraints (
            in ConstraintIDSeq id_list)
        raises (ConstraintNotFound);

    MappingConstraintInfoSeq get_all_mapping_constraints();

    void remove_all_mapping_constraints();

    void destroy();

    boolean match ( in any filterable_data,
            out any result_to_set )
        raises (UnsupportedFilterableData);

    boolean match_structured (
            in CosNotification::StructuredEvent filterable_data,
            out any result_to_set)
        raises (UnsupportedFilterableData);

    boolean match_typed (
            in CosTrading::PropertySeq filterable_data,
            out any result_to_set)
        raises (UnsupportedFilterableData);
```

```
}; // MappingFilter


interface FilterFactory {

    Filter create_filter (
            in string constraint_grammar)
        raises (InvalidGrammar);

    MappingFilter create_mapping_filter (
            in string constraint_grammar,
            in any default_value)
        raises(InvalidGrammar);

}; // FilterFactory


typedef long FilterID;
typedef sequence<FilterID> FilterIDSeq;

exception FilterNotFound {};


interface FilterAdmin {

    FilterID add_filter ( in Filter new_filter );

    void remove_filter ( in FilterID filter )
        raises ( FilterNotFound );

    Filter get_filter ( in FilterID filter )
        raises ( FilterNotFound );

    FilterIDSeq get_all_filters();

    void remove_all_filters();

}; // FilterAdmin


};
```

# Anhang C: IDL-Quelltext des NotificationServiceAgent

```
module NotificationServiceAgent {

  interface NotificationServiceAgent
  {
    CosNotifyChannelAdmin::EventChannelFactory getECF();
    CosNotifyFilter::FilterFactory getFF();
    string getProperty(in string name);
  };
};
```

# Literatur


[1] OMG: Notification Service, OMG Document Number telecom/98-01-01, 1998

[2] NEC Systems Laboratory: A CORBA-based Notification Service, 1997

[3] Kempter, Bernhard: „Entwurf eines Java/CORBA-basierten Mobilen Agenten“; Diplomarbeit; Technische Universität München; 1998

[4] ORBacus Notify: htp://www.ooc.com/notify/

[5] DSTC CosNotification: http://www.dstc.com/Products/CORBA/Notification_Service/

[6] OBJECT MANAGEMENT GROUP: CORBAservices – Trader Service, OMG Document Number formal/97-05-01, 1997

[7] OMG: „IDL/Java Language Mapping“; OMG Document Number 97-03-01

[8] OMG: The Common Object Request Broker“, OMG Document Number 97–02–25

[9] Boris Gruschke, Stephen Heilbronner, Helmut Reiser: Mobile Agent System Architecture,1999

[10] Sun Microsystems: Java Platform 1.2 Beta 4 API Specification, 1998