Freie Universität Berlin
Fachbereich Mathematik und Informatik
Institut für Informatik

Diplomarbeit

Entwurf und Implementierung des CORBA Notification Service

Alphonse Bendt

31. Mai 2003

Betreut durch Prof. Dr.-Ing. Klaus-Peter Löhr und Dr. Gerald Brose

Diplomarbeit von:
Alphonse Bendt
Wernigeroder Str. 26
10589 Berlin

Inhaltsverzeichnis

1. Einleitung

Große Netzwerke wie das Internet ermöglichen die Entwicklung von verteilten Applikationen, die die Interaktion von Personen und Systemen auf dem gesamten Globus erlauben. Die realisierten Anwendungen sind vielfältig.

Zum Teil erfordern technische Gründe die Verteilung von Systemen. So werden Systeme zur Gewährleistung von Fehlertoleranz und Ausfallsicherheit oft redundant ausgelegt. Ausfallsicherheit gegen physische Einwirkung, wie bspw. Feuer oder Erdbeben, erfordert es, Rechnersysteme an weit entfernten Standorten zu positionieren. Um hohe Rechenkapazitäten anzubieten, kann es kostengünstiger und einfacher sein, mehrere kleine Systeme zu verbinden, als ein großes und dementsprechend teures System zu verwenden. Anwendungen spiegeln oft die Arbeitsabläufe der realen Welt wider. In Zeiten eines globalisierten Alltags scheint es daher nur natürlich, dass auch weltweit verteilte Systeme miteinander agieren, um Dienstleistungen zu erbringen. Manche Ressourcen, auf die ein Rechner zugreift, sind an bestimmte geographische Orte gebunden. So existieren Systeme, die an Messpunkten Messdaten aufnehmen, um beispielsweise deutschlandweit die Luftqualität zu messen. Andere Systeme sind an den Aufenthaltsort bestimmter Personen gebunden. Der Extremfall sind mobile Systeme, die oft ihren geographischen Aufenthaltsort ändern können.

Die Entwicklung und Pflege von Software ist ein komplexer, schwieriger Vorgang. Dieser Prozess wird noch schwieriger, wenn Verteilungsaspekte berücksichtigt werden müssen. Ein Netzwerk kann überlastet und damit sehr langsam sein. Es können Nachrichten verlorengehen. Um mit all diesen Problemen umzugehen, ist zusätzliche Programmlogik notwendig. Ein weiteres Problem stellt die Gewährleistung von Sicherheit dar. Einerseits sollen alle Ressourcen gemeinsam genutzt werden können, andererseits sollen Ressourcen vor unerwünschtem Zugriff auf private Daten oder Hardware geschützt werden.

Die heute verbreiteten Programmiersprachen wie C/C++ oder Java bieten dem Anwendungsentwickler lediglich ein Abstraktionsniveau auf Ebene zuverlässiger Netzwerkverbindungen zu einem anderen Rechner. Die Kommunikation mittels eines einfachen Nachrichtenparadigmas (*send*, *receive*) steht im Gegensatz zum prozeduralen bzw. objektorientierten Paradigma. Die Programmierung gestaltet sich außerdem mühsam und fehlerträchtig.

Um Applikationen auf einem höheren Abstraktionsniveau entwickeln zu können, existieren verschiedene Konzepte, die die Entwicklung verteilter Anwendungen nach dem gleichen Programmiermodell wie lokale Anwendungen ermöglichen. Die Applikation kommuniziert dabei mit einer Softwareschicht, die als *Middleware* bezeichnet wird, die die Kommunikation mit den Primitiven

send, *receive* transparent abwickelt. Ein bekanntes Beispiel für eine Middleware, die es erlaubt, prozedural zu entwickeln, ist das *Remote Procedure Call* (RPC) Modell von Sun. Schließlich existieren Systeme wie CORBA oder DCOM, mit deren Hilfe Applikationen nach dem objektorientierten Paradigma entwickelt werden können.

Die Verwendung aufrufbasierter Kommunikation hat in bestimmten Einsatzgebieten Nachteile. Ein alternatives, wichtiges Kommunikationsmodell ist der Austausch von Nachrichten. Im Gegensatz zum einfachen Nachrichtenparadigma der Netzwerkebene, handelt es sich hier um Nachrichten auf Applikationsebene. Der bekannteste Dienst dieser Art ist e-Mail. Sie dient zur direkten Kommunikation zwischen Menschen. Benachrichtigungsdienste bieten eine ähnliche Funktionalität zur Kommunikation zwischen Applikationen, da e-Mail dazu weniger geeignet ist. Benachrichtigungssysteme werden auch als *Enterprise Messaging System* oder *Message oriented Middleware* (MOM) bezeichnet.

Ein Benachrichtigungssystem erlaubt Applikationen, Informationen in Form von Nachrichten auszutauschen. Eine Nachricht enthält benutzerdefinierte und administrative Daten. Diese können vom Benachrichtigungssystem ausgewertet werden, um die Nachricht zuzustellen. Der benutzerdefinierte Anteil einer Nachricht enthält beliebige anwendungsspezifische Daten, die üblicherweise Teil einer geschäftlichen Transaktion sind oder über bestimmte Ereignisse informieren.

Durch Einsatz einer MOM wird der Aufwand, Nachrichten über das Netzwerk an andere Applikationen zuzustellen, für den Benutzer transparent. Darüber hinaus bieten solche Systeme meist noch Fehlertoleranz, Lastverteilung, Skalierbarkeit und eine transaktionale Semantik.

Aufgabenstellung

Im Rahmen einer Diplomarbeit soll Entwurf und Implementierung des Notification Service für die freie CORBA-Implementierung JacORB [37] erfolgen. Die Implementierung soll sich dabei auf zwei Aspekte konzentrieren:

- Filterung: Die in der Spezifikation vorgesehenen Filterungsmöglichkeiten und die Filtersprache sind vollständig zu implementieren.
- Performanz: Die Basisfunktionalität der Benachrichtigungsvermittlung soll hochperformant und mit anderen Java-Implementierungen des Dienstes [47] vergleichbar sein. Auch die Implementierung der Filterfunktionalität soll unter Performanzgesichtspunkten erfolgen.

Die dienstgütebezogenen Aspekte der Notification Service-Spezifikation können, müssen aber nicht vollständig implementiert werden. Bei der Implementierung ist auf ein gut dokumentiertes Design und sowohl lesbaren als auch wartbaren Code zu achten. Benachrichtigungsdienste sind ein Forschungsgebiet, auf dem es in den letzten Jahren viele Aktivitäten gegeben hat. Die schriftliche

Arbeit soll in einer Einführung einen ausführlichen Vergleich aktueller industrieller Messaging-Plattformen wie *Java Messaging Service* (JMS) [38], *Message oriented Middleware* (MOM) wie IBM MQSeries und Forschungsprototypen wie SIENA [57] enthalten.

JacORB

JacORB ist eine freie Implementierung des CORBA-Standard, die am Fachbereich Informatik der Freien Universität Berlin entstanden ist. Die Entwicklung wird von Dr. Gerald Brose koordiniert und vom genannten Fachbereich und der Xtradyne Technologies AG unterstützt. JacORB steht unter der *GNU LESSER GENERAL PUBLIC LICENSE* (LGPL) zur Verfügung.

Weitere Vorgehensweise

Die Arbeit gliedert sich in einen theoretischen und einen praktischen Teil. Der praktische Anteil bezieht sich auf die Implementierung des CORBA Notification Service für JacORB, die im Rahmen dieser Diplomarbeit erstellt wurde. Der JacORB Notification Service ist auf der Webseite von JacORB erhältlich.

Zunächst werden in Kapitel 2 die den Benachrichtigungssystemen zugrundeliegenden Konzepte erläutert. Dazu werden die Anforderungen konkretisiert, die den Einsatz eines Benachrichtigungssystems motivieren. Nach der Vorstellung relevanter Technologien wird das Benachrichtigungskonzept modellhaft beschrieben.

Dieses Modell dient dazu, existierende nachrichtenbasierte Middleware in Kapitel 3 miteinander zu vergleichen. Dabei wird ein Überblick über die verfügbare Funktionalität der einzelnen Systeme gegeben.

Der praktische Teil der Arbeit beginnt in Kapitel 4 mit der Identifikation von Problembereichen der Implementierung. Anschließend werden mögliche Lösungsmuster diskutiert.

Nach der Festlegung einer Architektur erfolgt in Kapitel 5 die Beschreibung der Implementierung des Systems.

Schließlich wird in Kapitel 6 evaluiert, inwieweit die Anforderung der Performanz erreicht werden konnte. Hierzu wurden Tests mit anderen Implementierungen des Notification Service durchgeführt.

2. Benachrichtigungsdienste

Middlewareplattformen wie CORBA, Java RMI oder Microsoft DCOM basieren auf dem *Remote Procedure Call* (RPC) Modell [auch *Synchronous Method Invocation* (SMI)]. Das RPC-Modell bietet eine virtuelle Monoprozessorsicht auf ein verteiltes Szenario. Das bedeutet, dass ein Methodenaufruf auf einem entfernten Objekt semantisch und syntaktisch einem Aufruf auf einem lokalen Objekt äquivalent ist.[1] Wie bei einem lokalem Aufruf blockiert der Client so lange, bis der Aufruf abgeschlossen ist. Eine verteilte Applikation kann im Prinzip nach dem gleichen Paradigma wie eine lokale Applikation entwickelt werden. RPC eignet sich daher besonders für die horizontale Integration von Applikationen in verteilten Transaktionen (vgl. Abbildung 2.1 auf der nächsten Seite). Als erfolgreiches Beispiel können hier die n-tier Architekturen und insbesondere Komponenten-Architekturen wie *Enterprise Java Beans* (J2EE), *CORBA Components Model* (CCM) oder *DCOM* genannt werden.

Sobald jedoch Applikationen vertikal miteinander gekoppelt werden sollen, treten folgende Nachteile in den Vordergrund [20] (vgl. Abbildung 2.2 auf der nächsten Seite):

- Enge Kopplung von Client und Server.
  Client und Server sind bezüglich ihrer Lebensdauer eng miteinander gekoppelt. Der Server muss stets verfügbar sein, um eine Anfrage zu bearbeiten. Falls der Server nicht verfügbar ist, schlägt der Aufruf fehl. Um diese Fehler zu erkennen und zu verarbeiten, ist zusätzliche Applikationslogik auf Clientseite nötig, was die Codekomplexität erhöhen kann.

  Durch direkte Kenntnis voneinander sind Client und Server zusätzlich logisch eng miteinander gekoppelt. Dadurch wird es schwierig, einzelne Komponenten außerhalb ihres ursprünglichen Nutzungskontextes wiederzuverwenden.

- Synchrone Kommunikation.
  Bei synchroner Kommunikation ist ein Nachrichtenaustausch automatisch mit einer Prozess-Synchronisation der Teilnehmer verbunden. Der Client wird während des entfernten Operationsaufrufs blockiert, bis der Server die Ergebnisse liefert. Dabei wird die Parallelität von Client und Server schlecht ausgenutzt.

[1] Der Vollständigkeit halber sei darauf hingewiesen, dass lokale und entfernte Aufrufe sich oft doch, und zwar in teilweise ganz erheblichen Einzelheiten unterscheiden [6].

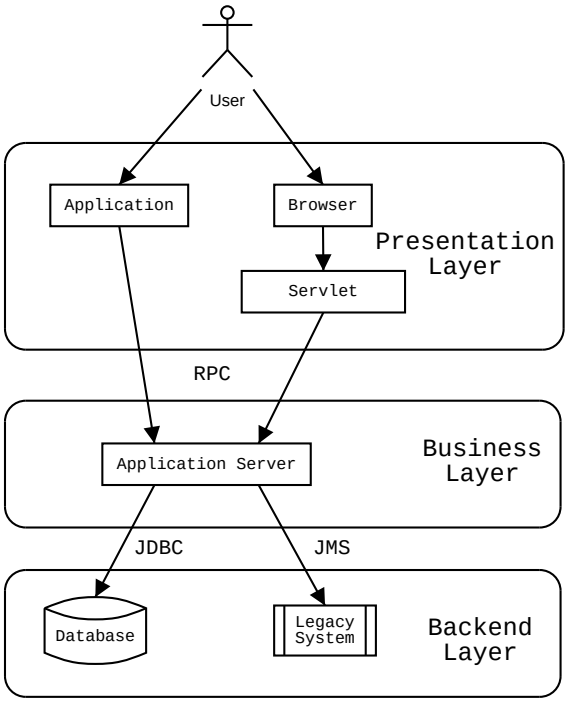

Abbildung 2.1.: Integration horizontaler Schichten

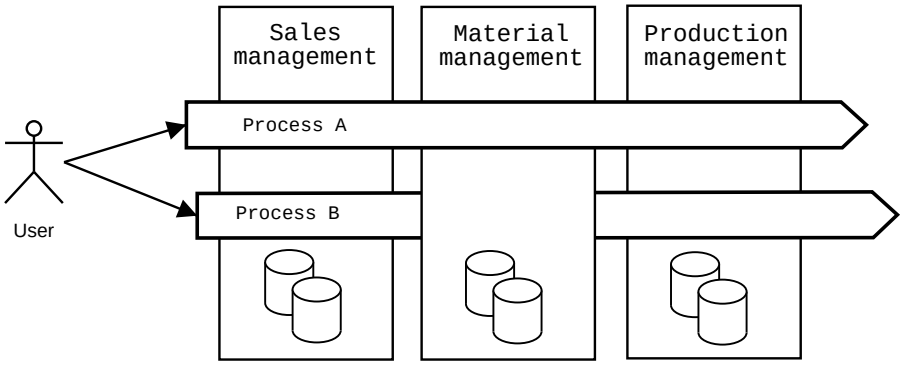

Abbildung 2.2.: Integration vertikaler Schichten

- Punkt-zu-Punkt Verbindungen.
  Ein Aufruf ist üblicherweise an ein einzelnes Ziel adressiert. Multicast Kommunikation wird nicht unterstützt.

Bei wachsender Anzahl der Kommunikationspartner sind RPC-basierende Applikationen anfällig gegen Störungen der zugrundeliegenden Verbindungen oder Nicht-Verfügbarkeit der Kommunikationspartner. Schwankende Latenzzeiten für die Netzwerkkommunikation mit nur einem der Kommunikationspartner können die Gesamtperformanz eines Systems verringern. Der Totalausfall einer Netzwerkverbindung oder eines Subsystems erfordert komplexe Fehlererkennungs- und Fehlerbehandlungsmechanismen, damit nicht das ganze System funktionsunfähig wird.

Insbesondere ist synchrone Kommunikation nicht geeignet im Fall von mobilen Komponenten bzw. Komponenten, die häufig nicht verfügbar sind.

Diese Nachteile gelten auch für horizontal gekoppelte Applikationen. Da dort die Funktionalität der höheren Schichten abhängig von der Funktionalität der niedrigeren Schichten ist, wird durch Ausfall einer tieferen Schicht die ganze Applikation funktionsunfähig. Dies kann am Beispiel einer J2EE-Webapplikation verdeutlicht werden. Webapplikationen bestehen üblicherweise aus einem Web-Tier, einem Applicationserver und einer Datenbank. Im Web-Tier laufen Servlets, die mittels RMI mit Enterprise Java Beans im Applicationserver kommunizieren. Die Beans kommunizieren wiederum mit einer Datenbank. Ist die Verbindung zwischen Webserver und Applicationserver gestört, ist das gesamte System funktionsunfähig. Die Funktionsfähigkeit der Servlets ist eng an die Verfügbarkeit der Beans gekoppelt. Der Einsatz von RPC zur Kommunikation zwischen Servlets und Beans und die damit verbundene Kopplung stellt daher keine zusätzliche Einschränkung dar.

Eine andere Situation besteht bei vertikal gekoppelten Systemen. Jedes der Systeme kann üblicherweise autark funktionieren. Die Nicht-Verfügbarkeit einzelner Kommunikationspartner wirkt sich lediglich auf die Teile der Funktionalität aus, die eine direkte Rückmeldung oder Aktion auf eine Nachricht erfordert. Ein Beispiel für dieses Szenario ist eine Anwendung in einem Unternehmen mit verteilten Standorten. Angenommen, es existieren die Standorte Vertrieb, Produktion und Lager. Sobald der Vertrieb eine Bestellung aufgibt, beauftragt der Vertrieb das Lager mit der Auslieferung. Das Lager prüft, ob eingehende Aufträge sofort ausgeführt werden können. Falls Waren nicht am Lager sind, wird ein Produktionsauftrag an die Produktion gesandt. Das System bleibt funktionsfähig, wenn einzelne Komponenten ausfallen. So könnte der Vertrieb bei Nicht-Verfügbarkeit des Lagers trotzdem Bestellungen aufnehmen. Wenn das Lager zu einem späteren Zeitpunkt wieder verfügbar ist, kann es über die inzwischen angefallenen Bestellungen informiert werden.

Wenn in Systemen alle Partner wechselseitig miteinander kommunizieren müssen, bedeutet dies, dass die Partner n:m Beziehungen zueinander haben. Solche Beziehungen sind schwierig zu verstehen, zu verwalten und zu erweitern. Um eine Dienstgüte und Fehlertoleranz zu erreichen, ist eine aufwendige Logik auf Applikationsebene notwendig. Der Einsatz eines Benachrichtigungs-

systems kann helfen, ein System zu restrukturieren. Dadurch wird der Aufbau des Systems verständlicher und Administration und Erweiterung des Systems gestalten sich einfacher. Durch den Einsatz bestehender Benachrichtigungssysteme kann Entwicklungsaufwand gespart werden. Außerdem ist ein bestehendes Benachrichtigungssystem meist stabiler und leistungsfähiger als eine Eigenentwicklung, was sich auch positiv auf die Performanz und Stabilität der Applikation auswirken kann.

Letztendlich kann durch Einsatz eines Benachrichtigungssystems mit standardisierter Schnittstelle auch die Integrierbarkeit einer Applikation verbessert werden, da der eingesetzte Benachrichtigungsdienst einfach gegen eine andere Implementierung mit gleicher (standardisierter) Schnittstelle ausgetauscht werden kann.

2.1. Anwendungsszenarien

Im vorangegangenen Kapitel wurde ein Beispiel für ein typisches Einsatzgebiet von Benachrichtigungsdiensten gegeben. Im folgenden werden weitere Anwendungsszenarien von Benachrichtigungsdiensten skizziert, um zu verdeutlichen, in welchen Gebieten Benachrichtigungsdienste zum Einsatz kommen können. Aus den verschiedenen Anwendungsszenarien ergeben sich Anforderungen, die ein Benachrichtigungsdienst erfüllen muss, um flexibel eingesetzt werden zu können.

eCommerce Anwendungen Beispiele dafür sind Online-Auktionssysteme wie eBay, oder Reservierungssysteme für Fluggesellschaften, Bahnbetreiber etc. Diese Applikationen sind durch ein hohes Datenaufkommen charakterisiert. An der Kommunikation sind vergleichsweise wenige Parteien beteiligt, die sich aber an weit voneinander entfernten Orten befinden.[2] Die Kommunikationspartner sind oft nur an speziellen Untermengen der verfügbaren Informationen interessiert.

News Dies umfasst netzbasierte Sportnachrichten, Börsenticker, Notfallmeldungen und sonstige Benachrichtigungen. Diese Applikationsklasse ist ebenfalls im Bereich der Wide Area Networks angesiedelt.

Interaktive Netzwerkspiele Als Hauptanforderung stellt diese Applikationsklasse die geringe Latenzzeit der Kommunikation. Der Verlust einzelner Datenpakete ist üblicherweise nicht als kritisch anzusehen.

Echtzeitsysteme Steuerung von z. B. Flugzeugelektronik. Echtzeitsysteme benötigen im Gegensatz zu interaktiven Netzwerkspielen nicht unbedingt minimale Latenzzeiten. Wesentlich wichtiger sind garantierte Antwortzeiten [vgl. 29].

Komplexe Software-Systeme Diese Applikationsklasse hat als zentrale Anforderung die Dienstgüte und Zuverlässigkeit eines Benachrichtigungssystem.

[2] Weit entfernt im Sinne der geographischen Distanz als auch der Netzwerktopologie.

Um die bei solchen Systemen komplexen Datentypen filtern zu können, sind mächtige Filter auf Anwendungsebene notwendig. Im Gegensatz zu Applikationsklassen, die in Wide Area Networks angesiedelt sind, werden die Systeme dieser Applikationsklasse üblicherweise in einem lokalen Netzwerk betrieben. Daher können auch große Datenmengen schnell übertragen werden.

Die beschriebenen Szenarien haben unterschiedliche Anforderungen an ein Benachrichtigungssystem. Als zentrale Anforderungen ergeben sich:

Filtermöglichkeiten Ein Benachrichtigungsdienst muss Möglichkeiten bieten, den Clients eine Auswahl der Nachrichten zu erlauben.

Skalierbarkeit Ein Benachrichtigungsdienst ist mit zahlreichen, parallel agierenden Clients verbunden. Idealerweise ist der Rechenaufwand für den Dienst linear abhängig von der Anzahl der Clients.

Geringe Latenzzeiten Latenzzeit bezeichnet in diesem Kontext die Verzögerung zwischen Absenden einer Nachricht und dem Empfang bei den einzelnen Empfängern.

Echtzeitanforderungen Bei Echtzeitsystemen ist die Korrektheit des Systemverhaltens nicht nur von den logischen Ergebnissen der Berechnungen abhängig, sondern auch vom Zeitpunkt, an dem diese Ergebnisse ausgegeben werden. Um in einem Echtzeitsystem eingesetzt werden zu können, muss ein Benachrichtigungsdienst daher deterministische und statistische Antwortzeiten garantieren.

Zuverlässigkeit der Auslieferung Nach einem erfolgreichen synchronen Aufruf hat der Aufrufer die implizite Kenntnis, dass der Aufruf bearbeitet wurde (falls kein Fehler aufgetreten ist). Beim einfachen Nachrichtenversand entfällt dies. Daher muss die Möglichkeit bestehen, beim Versand der Nachricht anzugeben, ob diese garantiert ausgeliefert werden muss oder nicht.

Teilweise können sich die Anforderungen gegenseitig behindern. So steht beispielsweise eine zuverlässige Auslieferung im Widerspruch zu minimaler Latenzzeit, da eine Nachricht, nachdem sie vom Benachrichtigungsdienst entgegengenommen wurde, zunächst zeitaufwendig persistent gemacht werden muss, bevor sie weiterverabeitet wird. Ein flexibler Benachrichtigungsdienst sollte daher Einstellmöglichkeiten bieten, um optimal an ein bestimmtes Einsatzszenario angepasst zu werden.

2.2. Relevante Technologien

Die eingangs genannten Probleme *synchrone Kommunikation*, *Unicastadressierung* und *enge Kopplung* treten in zahlreichen Anwendungsgebieten der Informatik auf. Dementsprechend existieren bereits tragfähige Lösungen und Konzepte in Forschung und Praxis.

In diesem Abschnitt wird ein Überblick über Technologien gegeben, die relevante Lösungsansätze realisieren. Ziel ist es, das Konzept eines Benachrichtigungsdienstes klar zu definieren und gegenüber anderen Technologien abzugrenzen.

2.2.1. Asynchrone Aufrufe

Bei Verwendung asynchroner Aufrufe blockiert der Client nur so lang, bis die Nachricht versendet wurde. Dies ist im Allgemeinen der Fall, bis die Nachricht im Sendepuffer der darunterliegenden Laufzeitumgebung abgelegt wurde. Der Client blockiert demnach nur kurze Zeit. Damit existieren folgende Vorteile:

- Client und Server arbeiten zeitlich entkoppelt.
- Durch die zeitliche Entkopplung und nicht-blockierende Kommunikation kann Parallelität besser genutzt werden.
- Um auf das Ergebnis des asynchronen Aufrufs zuzugreifen, kann der Client

  1. ein Callback-Interface registrieren, das vom Laufzeitsystem aufgerufen wird, sobald das Ergebnis verfügbar ist, oder
  2. zu einem späteren Zeitpunkt aktiv nach dem Rückgabewert fragen (Polling).

Diese Vorgehensweise kann helfen, den Ressourcenbedarf eines Systems zu verringern und wird daher speziell in echtzeit- oder ereignisgesteuerten Systemen angewendet. Auf der anderen Seite haben asynchrone Methodenaufrufe folgende Nachteile:

- Im Gegensatz zu der asynchronen Kommunikation findet bei synchroner Kommunikation keine Prozess-Synchronisation der Kommunikationspartner statt. Daher sind ggfs. zusätzliche Mechanismen zur Prozess-Synchronisation notwendig, falls dieses Verhalten notwendig ist.
- Ist eine hohe Zuverlässigkeit der Kommunikation notwendig, so muss auf Client- als auch auf Serverseite eine Pufferung der Nachrichten stattfinden. Sind diese Puffer gefüllt, so muss wiederum blockiert werden. Dies widerspricht dem asynchronen Konzept. Alternativ können Nachrichten verworfen werden.
- Der Informationsfluss in Applikationen, die asynchrone Kommunikation verwenden, ist im Allgemeinen schwieriger verständlich und damit auch schwieriger wartbar, als in Applikationen, die synchrone Kommunikation verwenden. Die Codestruktur spiegelt nicht die Informationsflüsse wider.

Ein spezieller Fall asynchroner Aufrufe ist die sog. *one-way* Kommunikation. Dabei ist der Aufrufer nicht am Ergebnis des Aufrufs interessiert bzw. der Aufruf liefert kein Ergebnis. Im Gegensatz zu *two-way* Kommunikation kann der Aufrufer keine Aussage darüber treffen, ob der Aufruf erfolgreich ausgeführt wurde.

In CORBA existiert seit Version 2.4 das *Asynchronous Messaging Interface* (AMI), das asynchrone Kommunikation ermöglicht. Das AMI erlaubt sowohl das Programmiermodell mit Callbacks als auch mit Polling.[3] Außerdem besteht die Möglichkeit, das *Dynamic Invocation Interface* (DII) zu verwenden. Durch Verwendung des DII sind asynchrone und one-way Methodenaufrufe möglich [siehe 7, 259f].

Asynchrone Aufrufe, wie sie bei CORBA (AMI, DII) oder COM+ eingesetzt werden können, sind ein relativ systemnahes Konzept. Asynchrone Aufrufe können die Komplexität einer Applikation erhöhen, da die Fehlerbehandlung weiterhin auf Applikationsebene behandelt werden muss. Asynchrone Aufrufe sind keine Möglichkeit, um zur Entkopplung der Lebensdauer von Client und Server beizutragen. Darüber hinaus ist auch mit asynchronen Methodenaufrufen keine Multicast-Adressierung möglich.

2.2.2. Zentralisierte Dienste

Diese Systeme sind dadurch gekennzeichnet, dass eine lokale, zentrale Instanz (Message Bus oder Event Queue) existiert, die Nachrichten zwischen den beteiligten Komponenten vermittelt. In diese Kategorie fallen graphische Nutzerschnittstellen wie das X-Windowsystem, das Java Event-Modell und nachrichtenbasierte Entwicklungsumgebungen, wie z. B. HP SoftBench, DEC FUSE und Sun ToolTalk [9, S. 7]. Im folgenden Abschnitt wird exemplarisch das Java Event-Modell beschrieben.

Java Event-Modell

Das *Java Event-Modell* findet vor allem in Nutzerprogrammen mit graphischer Oberfläche (GUI) Anwendung, ist jedoch flexibel genug, um auch in anderen Bereichen Einsatz zu finden. Das Java Event-Modell ist eine Anwendung des Entwurfsmusters *Observer* [19, 287f].

Im Java Event-Modell bezeichnet ein Ereignis die Zustandsänderung einer beliebigen Applikationskomponente. So werden beispielsweise Ereignisse ausgelöst, wenn der Nutzer eine Texteingabe in ein Eingabefeld macht, die Maus bewegt oder mit der Maus auf ein Fensterelement klickt. Ereignisse werden als Objekte vom Typ *EventObject* modelliert. Die elementare Klasse *EventObject* enthält lediglich die Information über den Auslöser (Event Source) eines Ereignisses. Dies kann z. B. ein Texteingabefeld sein. In abgeleiteten Klassen können weitere Informationen verfügbar sein. So enthält die Klasse *KeyEvent* beispiels-

[3] Siehe [63] für einen Überblick und Gegenüberstellung des synchronen und des asynchronen Programmiermodells mit CORBA.

weise die zusätzliche Information, welcher Tastendruck das Ereignis ausgelöst hat.

Interessierte Komponenten (sog. *Event Listener*) können sich für bestimmte Ereignisse registrieren. Die Komponenten müssen dazu ein vorgegebenes Callback-Interface (die Namenskonvention lautet: <*EventType*>EventListener) implementieren und sich durch Aufruf einer bestimmten Methode registrieren. Sobald ein Ereignis auftritt, werden alle registrierten Beobachter durch Aufruf einer Methode des Callback-Interface benachrichtigt.

Durch Erweiterung um einen Fernaufrufmechanismus wie RMI kann das Java Event-Modell auch in verteilten Applikationen eingesetzt werden. Das Java Event-Modell kann dabei helfen, flexibel erweiterbare Applikationen zu erstellen.

Letztendlich kann das Java Event-Modell jedoch nicht als echter Benachrichtigungsdienst angesehen werden, da keine echte Entkopplung zwischen Sender und Empfänger stattfindet. Die Empfänger haben direkte Kenntnis von den Sendern. Dadurch wird eine starke Abhängigkeit zwischen Sender und Empfänger geschaffen, wodurch Sinn und Zweck eines Benachrichtigungsdienstes untergraben werden [9, S. 13].

Außerdem erfolgt die Benachrichtigung der einzelnen Beobachter üblicherweise sequentiell und nicht nebenläufig.

2.2.3. Internet-Technologien

Internet-Applikationen sind dezentral administriert und weit verteilt. Die Probleme, die den Einsatz eines Benachrichtigungsdienstes motivieren, treten daher auch in diesem Kontext auf. Die Betrachtung einiger Beispiele zeigt skalierbare Lösungen, die zwar nicht universell einsetzbar sind, jedoch wertvolle Ideen zur Realisierung eines Benachrichtigungssystems beisteuern können.

USENET News

Das *USENET News System* (Netnews) ist ein gutes Beispiel für eine Applikation, die auf Nutzerebene skalierbare n:m-Kommunikation realisiert.

Das USENET standardisiert eine Menge von Protokollen, um Nachrichten in einem dezentralen Netzwerk von News-Servern auszutauschen. Artikel sind in hierarchisch organisierten Newsgroups strukturiert. Ein individueller News-Server speichert lokal alle Artikel, die er von benachbarten News-Servern erhalten hat. Damit ist der Zugriff von Clients aus, die mit diesem News-Server verbunden sind, sehr schnell. USENET benötigt keinen zentralen Server. Wenn ein Client einen neuen News-Artikel auf einem News-Server veröffentlicht, wird der Artikel von seinem an alle benachbarten News-Server weitergeleitet, diese leiten die Nachricht wiederum an alle ihre Nachbarn weiter. In dieser Weise wird die Nachricht durch ein weltumspannendes Netz propagiert.

Ein News-Artikel ist ähnlich wie eine e-Mail modelliert. Er besteht aus einem sog. *Header* und einem *Body*. Der Header ist eine Liste von Namen/Wertpaaren

und enthält Informationen, wie z. B. den Namen und e-Mailadresse des Absenders, Newsgroupnamen und einen eindeutigen Bezeichner. Die Informationen aus dem Header werden für die Propagation des Artikels zwischen den Servern verwendet. Der Body enthält den (aus Sicht des News-Server) unstrukturierten Nachrichtentext.

Das USENET stellt eine stabile, ausgereifte und skalierbare Technologie dar. Dem Einsatz als Grundlage für Benachrichtigungsdienste stehen jedoch einige Probleme im Weg: Die Filtermöglichkeiten sind nicht sehr elaboriert. Newsgroupname und Betreff der einzelnen Artikel gliedern die Nachrichten zwar, in der Praxis ist diese Gliederung jedoch für die meisten Nutzer zu grob. Daher bieten die meisten Newsreader dem Endnutzer Filterfunktionen an, um aufgrund der Headerinformationen und des Artikeltextes bestimmte Artikel auszuwählen.

Der Dienst ist außerdem relativ schwergewichtig. Es kann mehrere Tage dauern, bis ein Newsartikel alle News-Server weltweit erreicht hat.

IP Multicast

IP Multicast ist ein Mechanismus zur Gruppenkommunikation auf Netzwerkebene. In einem IP-Netzwerk werden Kommunikationspartner durch einen eindeutigen Bezeichner, die sog. IP-Adresse, identifiziert. Die IP-Adresse wird verwendet, um Datenpakete vom Absender zum Empfänger zu routen. Eine Multicast-Adresse bezeichnet eine Gruppe von Kommunikationsendpunkten, die jeweils aus mehreren Rechnern bestehen können. Der Absender schickt Datenpakete nur einmal an eine Multicast-Adresse. In einem LAN funktioniert dies noch recht einfach, da alle Multicast-Teilnehmer an demselben Übertragungsmedium angeschlossen sind. Für die Kommunikation in Weitverkehrsnetzen wurden spezielle Routingverfahren entwickelt. Der MBone ist ein Multicast-Netzwerk, welches auf die existierende Internet-Infrastruktur aufsetzt. Multicast=PPakete werden von jedem Sender über eine virtuelle Baumstruktur zu den Gruppenteilnehmern geschickt. Die Daten werden nur an den Knoten vervielfältigt, die Multicast-Teilnehmer in mehreren Ästen haben. Damit wird jedes Datenpaket in jedem Netzabschnitt nur einmal übertragen.

Multicast-Pakete basieren auf IP-Paketen. Unterschiedlichste Anwendungen können daher Multicast nutzen. Interessant ist Multicast vor allem für datenintensive Anwendungen wie Videostreaming, da dort das übertragene Datenvolumen erheblich reduziert werden kann.

Es existieren jedoch keine Filtermöglichkeiten für Multicast. Ein interessierter Nutzer kann sich lediglich in einer Multicast-Gruppe anmelden, empfängt dann alle Pakete, die an diese Gruppe versendet wurden und kann sich später wieder abmelden. Aufgrund der fehlenden Filtermöglichkeiten kann Multicast daher alleine nicht als Benachrichtigungsdienst verwendet werden. Multicast wird jedoch als Transportmechanismus für Benachrichtigungsdienste verwendet.

Multicast ist vor allem für Applikationen interessant, die keinen zentralen Server verwenden und für den Fall, dass Skalierbarkeit im WAN eine besondere

Rolle spielt.

2.2.4. Verteilte nachrichtenbasierte Systeme

Es existieren verschiedene nachrichtenbasierte Systeme, die auf einer verteilten Infrastruktur aufsetzen. JEDI, Elvin und Keryx sind einige Beispiele für solche Systeme. Manche von ihnen sind frei erhältlich, andere sind kommerziell verfügbar. Der Autor Carzaniga [9] kategorisiert diese Systeme anhand ihrer Möglichkeiten, Entscheidungen über die Weiterleitung von Nachrichten durch Auswertung der einzelnen Nachrichten zu treffen.

Channel-based Subscription

Der einfachste Subskriptionsmechanismus wird üblicherweise als Kanal bezeichnet. Sender benachrichtigen Empfänger, indem sie die Nachrichten in einem Kanal veröffentlichen. Der Benachrichtigungsdienst liefert die Nachrichten an jeden Interessenten aus, der am betreffenden Kanal angemeldet ist.

Anwendungen, die Channel-based Subscription realisieren, sind beispielsweise der CORBA Event Service und Mailinglisten.

Durch die nicht vorhandenen Filtermechanismen kann dieser Subskriptionsmechanismus auch mit Hilfe von Multicast realisiert werden. Eine Multicast-Group entspricht dabei einem Kanal.

Subject-based Subscription

Dieses Konzept erweitert die Channel-based Subscription. Benachrichtigungen enthalten dabei ein wohldefiniertes Attribut, das Thema oder Betreff (Subject). Der Benachrichtigungsdienst kann anhand des Themas eine Auswahl treffen, welche Benachrichtigungen weitergeleitet werden und welche nicht. Das USENET ist eine praktische Anwendung dieses Konzepts. Die Konfiguration eines News-Servers legt fest, welche Newsgroup an welchen benachbarten Server weitergeleitet werden soll. Ebenso können Endnutzer auswählen, an welcher Newsgroup sie interessiert sind und laden daraufhin alle Artikel aus dieser Newsgroup herunter.

Voraussetzung für Subject-based Subscription ist ein festgelegtes Vokabular für die Beschreibung von Nachrichten durch ein Thema. Meist wird dazu ein hierarchischer Namensraum verwendet. Wenn die Struktur der Nachrichten sich schlecht auf diese vorgegebene Hierarchie abbilden lässt, zeigt sich jedoch die Unflexibilität dieses Ansatzes.[4]

Subject-based Subscription lässt sich auf Basis von Channel-based Subscription realisieren. Dabei kommt pro Subject ein eigener Kanal zum Einsatz.

[4] Usenet Postings lassen sich beispielsweise teils nicht eindeutig, einer einzelnen Newsgroup zuordnen. Das Posting kann dann in mehreren Newsgroups veröffentlicht werden (*Crossposting*).

Content-based Subscription

Wenn die ganze Nachricht einer wohldefinierten Struktur unterliegt, können komplexe Filter innerhalb von Subskriptionen verwendet werden. Ein Benachrichtigungsdienst kann anhand eines Filterausdrucks komplexe Entscheidungen treffen, welche Benachrichtigungen weitergeleitet werden sollen. Da die ganze Nachricht zur Evaluierung durch den Dienst zur Verfügung steht, wird diese Form auch Content-based Subscription (inhaltsbasierte Subskription) genannt.

Die Nachrichten sind selbstbeschreibend. Dadurch ist dieses System sehr flexibel. Es wird kein gemeinsames Vokabular mehr benötigt.

Anwendungen, die Content-based Subscription realisieren, sind beispielsweise der CORBA Notification Service, Java JMS und SIENA.

2.2.5. Datenbanken

Tatsächlich kann ein Datenbank-System (DBMS) als Grundlage für ein nachrichtenbasiertes System verwendet werden. Datenbanken bieten mächtige Abfragesprachen und sind meist fehlertolerant realisiert.

Datenbanktrigger sind dazu geeignet, um ein nachrichtenbasiertes System zu realisieren. Mit Hilfe eines Trigger kann man Bedingungen und Aktionen deklarieren.

Das Nachrichtenformat kann auf das flexible Typsystem der darunterliegenden Datenbank aufsetzen. Datenbanken verfügen über zahlreiche Typen und sind damit für viele Anwendungsgebiete geeignet. Durch die Verwendung von Relationen können auch rekursive Datentypen realisiert werden.

Das Publizieren neuer Nachrichten löst Datenbanktrigger aus. Zur Filterung der Beobachterprofile kann die mächtige Abfragesprache der Datenbank verwendet werden. Mit Hilfe von SQL können Abfragen auf jedem Feld der Nachricht erfolgen. Mit Hilfe eines DBMS kann also Content-based Subscription realisiert werden.

Um mehrere Datenbank-Instanzen miteinander zu einer dezentralisierten Architektur zu koppeln, ist der Einsatz einer Middleware zur Kommunikation notwendig. Zahlreiche DBMS bieten proprietäre Replikationsmechanismen an, um den Datenbestand auf mehrere Serverinstanzen zu verteilen. Die Zielsetzung dieser Mechanismen ist vor allem die Zuverlässigkeit. Es ist daher schwierig, mit Hilfe dieser Mechanismen geringe Latenzzeiten zu erreichen.

Letztendlich kann die Auslieferung der Benachrichtigungen unter transaktionalem Schutz erfolgen, wodurch auch hohe Ansprüche an die Zuverlässigkeit eines Dienstes befriedigt werden können.

Insgesamt sind DBMS für hochperformante, nachrichtenbasierte Anwendungen ungeeignet. Die Persistenzmechanismen sind zu langsam. Die Triggermechanismen skalieren nicht [54]. Ein DBMS wird daher eher als reiner Persistenzmechanismus innerhalb eines Benachrichtigungsdienstes zum Einsatz kommen.

2.3. Modell eines Benachrichtigungsdienstes

Ein Benachrichtigungsdienst dient als Vermittler von Informationen zwischen Herausgebern und Interessenten. Herausgeber und Interessenten sind im Allgemeinen verteilte Komponenten einer komplexen Anwendung. Falls ein Herausgeber beabsichtigt, ein bestimmtes Informationsangebot zu veröffentlichen, kündigt er dies bei dem Benachrichtigungsdienst an (*advertisement*). Die Ankündigung enthält Informationen über Aufbau und Inhalt der Nachrichten, die der Herausgeber veröffentlichen möchte (vgl. Schritt 1 in Abbildung 2.3).

Interessenten können das Informationsangebot einsehen (Schritt 2) und ein Profil der für sie interessanten Informationen definieren (Schritt 3). Der Benachrichtigungsdienst speichert die Subskription des Interessenten zur späteren Verwendung.

Bei vielen Systemen sind die Schritte 1 und 2 nicht vorhanden bzw. nicht zwingend erforderlich. Ein Interessent beginnt in diesem Fall direkt mit dem dritten Schritt.

Der Herausgeber schickt seine Informationen in einem vorgebenen Nachrichtenformat an den Benachrichtigungsdienst (vgl. Schritt 1 in Abbildung 2.4 auf der nächsten Seite). Anhand der registrierten Profile kann der Benachrichtigungsdienst feststellen, zu welchen Profilen die Informationen passen (Schritt 2 und 3) und die Nachrichten den entprechenden Interessenten zustellen (Schritt 4).

Der Benachrichtigungsdienst kann zuverlässige Auslieferungsmodi bieten. Die Auslieferung erfolgt aus Sicht des Herausgebers asynchron. Die Schnittstellen zum Benachrichtigungsdienst sind sprach- und plattformunabhängig. Der Dienst ist in all diesen Aspekten flexibel konfigurierbar, um an unterschiedlichste Einsatzgebiete angepasst werden zu können.

Die Lebensdauer von Interessenten und Herausgeber wird entkoppelt. Außerdem wird durch Abstraktion von der darunterliegenden Middleware von einem bestimmten Transportmechanismus abstrahiert. Dies erlaubt, eine Applikation stärker problemorientiert zu entwickeln.

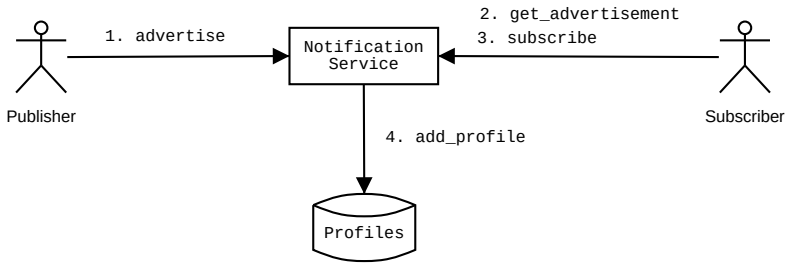

Abbildung 2.3.: Modell eines Benachrichtigungsdienstes: Ankündigung und Subskription

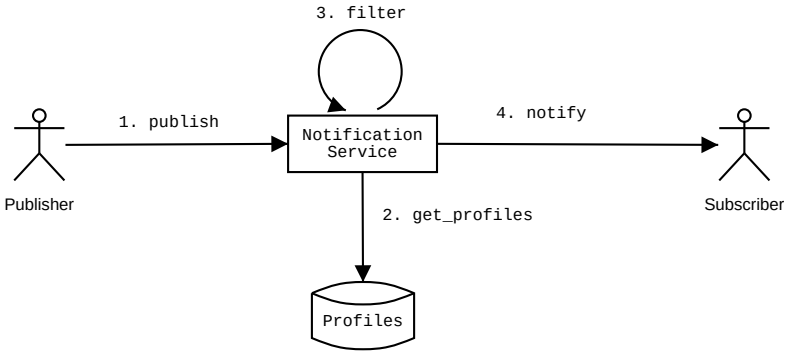

Abbildung 2.4.: Modell eines Benachrichtigungsdienstes: Veröffentlichung und Benachrichtigung

2.3.1. Definitionen

Die Betrachtung typischer Anwendungsszenarien ergibt typische Anforderungen an einen Benachrichtigungsdienst (siehe Abschnitt 2.1). Die Untersuchung der relevanten Technologien zeigte, dass manche der betrachteten Systeme einige der Anforderungen bereits effektiv lösen (vgl. Abschnitt 2.2). Auffallend ist, dass keine einheitliche Terminologie verwendet wird, was den Vergleich der unterschiedlichen Systeme erschwert. Daher sollen an dieser Stelle einige Begriffe konkretisiert werden, die im Zusammenhang mit Benachrichtigungsdiensten verwendet werden. Im Anschluss daran sollen für Benachrichtigungssysteme relevante Konzepte erläutert, und wo sinnvoll, dem RPC Modell gegenübergestellt werden.

Begriffe

Ereignis (Event) In ereignisbasierten Systemen unterscheidet Hinze *zeitliche* Ereignisse und Ereignisse, die sich auf *Zustandsänderungen* beziehen [30]. Ein zeitliches Ereignis kennzeichnet das Erreichen eines bestimmten Zeitpunkts oder das Verstreichen eines bestimmten Zeitintervalls. Diese Art von Ereignissen spielt in verteilten Applikationen eine geringe Rolle. Die andere Art von Ereignissen wird von Objekten ausgelöst, deren Zustand sich ändert. Ein Objekt kann dabei als Zustandsautomat angesehen werden. Sobald eine Transition von einem Zustand in den nächsten stattfindet, hat ein Ereignis stattgefunden. Diese Art von Ereignissen wird von Hinze als *aktives Ereignis* bezeichnet. Ein *passives Ereignis* tritt im Zusammenhang von aktiven Ereignissen und Zeit auf. Ein passives Ereignis wird dabei als Abfolge bestimmter aktiver Ereignisse $e_1 \ldots e_n$ in einem bestimmten Zeitintervall t bezeichnet. Ein passives Ereignis ist also aus mehreren aktiven Ereignissen *zusammengesetzt*.

Der Versuch, auf einen Rechnerport zuzugreifen, an dem kein Serverprozess aktiv ist, kann beispielsweise als aktives Ereignis angesehen werden. Finden nun innerhalb eines bestimmten Zeitintervalls mehrere dieser ak-

tiven Ereignisse statt, so könnte dies ein Hinweis auf einen Portscan sein. Das passive Ereignis „Portscan“ lässt sich also als eine Abfolge von aktiven Ereignissen ausdrücken.

Quelle (Event Producer/Event Source) Das Objekt, welches das Ereignis auslöst, wird als *Quelle* bezeichnet. *Event Producer* oder *Event Source* sind synonyme Bezeichnungen.

Beobachter (Event Consumer/Event Sink/Observer) Die Objekte, die am Eintritt bestimmter Ereignisse interessiert sind, werden üblicherweise als *Beobachter* bezeichnet. Im Kontext verschiedener Systeme findet man auch die Begriffe *Event Consumer*, *Event Sink* oder *Observer*.

Benachrichtigung (Notification) Ein Ereignis kann auftreten, ohne dass Beobachter existieren, die davon Notiz nehmen. Im Gegensatz dazu werden interessierte Beobachter vom Eintritt eines Ereignisses *benachrichtigt*. Die Begriffe „Ereignis“ und „Benachrichtigung“ werden oft synonym verwendet, obwohl sie zwei unterschiedliche Fakten beschreiben.

Die Begriffe „Quelle“ und „Beobachter“ sind an das Entwurfsmuster *Beobachter* angelehnt [19, 287f]. Das Entwurfsmuster entkoppelt Sender und Empfänger auf syntaktischer Ebene und erlaubt eine einfache Art der Multicast adressierung. Diese Art der Interaktion ist auch als *publish-subscribe* (publiziere und abonniere) bekannt. Der Begriff wird üblicherweise auch verwendet, um die Interaktion von Quelle und Beobachter bei Verwendung eines Benachrichtigungssystems zu beschreiben. Ein Benachrichtigungsdienst kann also als Instanz des Beobachtermusters angesehen werden.

Bei existierenden Systemen kommt üblicherweise ein weiterer Kommunikationsstil zum Einsatz: *point-to-point*. Bei point-to-point wird über eine Warteschlange kommuniziert. Diese ist üblicherweise FIFO geordnet. Jede Nachricht wird nur einem Empfänger zugestellt und anschließend aus der Warteschlange entfernt. Manche Systeme lassen es zu, dass die Empfänger die Nachrichten in der Warteschlange betrachten, ohne sie daraus zu entfernen. Mit point-to-point ist jedoch keine n:m-Kommunikation möglich. Damit kann point-to-point nicht als echtes Benachrichtigungssystem angesehen werden.

Wichtige Konzepte

Ein Ereignis hat immer eine Quelle, es können jedoch mehrere Beobachter benachrichtigt werden. Daher spricht man auch von n:m Kommunikation, da ein Beobachter auch solche Benachrichtigungen erhalten kann, die aus unterschiedlichen Quellen stammen. Im Gegensatz dazu kann ein Client in einer aufrufbasierten Architektur zu einem Zeitpunkt lediglich 1:1 kommunizieren.

Bei RPC-Kommunikation hält der Client eine Referenz auf einen Server. Die Laufzeitumgebung des Servers hält ebenfalls mindestens für die Dauer des Aufrufs Informationen über den Client, um das Ergebnis des Aufrufs zurückgeben zu können. In ereignisbasierten Systemen hält der Beobachter keine Referenz

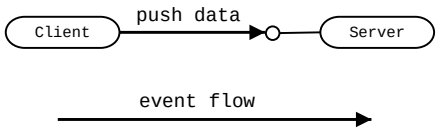

Abbildung 2.5.: Client Push

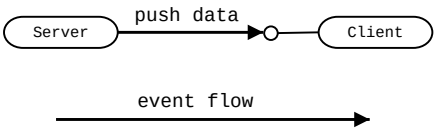

Abbildung 2.6.: Server Push

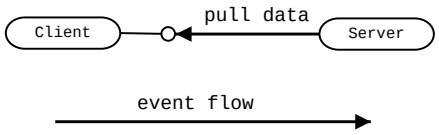

Abbildung 2.7.: Server Pull

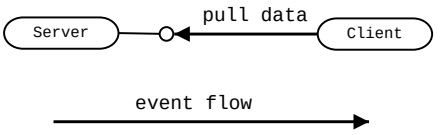

Abbildung 2.8.: Client Pull

auf die Quelle des Ereignisses. Die Quelle benötigt jedoch eine Referenz, um die Benachrichtigung zu versenden. Daher sprechen die Autoren Brose et al. hier von einem impliziten Aufruf [7, S. 470]. Das aktive Versenden einer Nachricht durch einen Aufruf wird üblichweise als *push* bezeichnet (Abbildung 2.5). Der gleiche Begriff wird verwendet, wenn der Benachrichtigungsdienst die Nachricht aktiv dem Empfänger zustellt (Abbildung 2.6).

Alternativ bieten viele Benachrichtigungssysteme an, dass der Beobachter explizit nach neuen Ereignissen fragen kann (*pull* oder Polling, Abbildung 2.8). Korrespondierend kann der Benachrichtigungsdienst das Pull-Modell verwenden, um neue Nachrichten bei der Quelle abzuholen (Abbildung 2.7).

Client/Server-Architekturen verwenden oft sog. Session-Konzepte. Kontextinformationen werden über mehrere Aufrufe hinweg gespeichert und stehen bei der Bearbeitung der einzelnen Operationen zur Verfügung. Im Gegensatz dazu enthalten Benachrichtigungen stets alle relevanten Informationen, die zur Auswertung des auslösenden Ereignisses notwendig sind, da keine Annahmen über den Kontext gemacht werden können.

Als wichtigster Unterschied von ereignis- und aufrufbasierten Architekturen nennen Brose et al. die Entkopplung der Beteiligten. Ereignisquelle und Beobachter müssen weniger voneinander wissen als Client/Server-Paare. Damit werden die Entwurfsziele des Entwurfsmusters *Vermittler* (Mediator) erreicht [19, 385f]:

- Das Vermittlermuster entkoppelt Kollegen (kooperierende Objekte). Ein Vermittler fördert die lose Kopplung zwischen den Kollegen. Die Kollegen können unabhängig voneinander variiert und wiederverwendet werden.
- Das Protokoll zwischen den Objekten wird vereinfacht. Ein Vermittler ersetzt n:m-Beziehungen durch 1:n-Beziehungen zwischen dem Vermittler und den Kollegen. 1:n-Beziehungen sind einfacher zu verstehen, zu verwalten und zu erweitern.
- Es abstrahiert davon, wie Objekte zusammenarbeiten. Die Vermittlung zwischen den Objekten wird zu einem eigenständigen und unabhängigen

Konzept gemacht und in ein Objekt gekapselt. Dadurch steht die Interaktion der Kollegenobjekte wieder im Vordergrund und es kann von ihrem individuellen Verhalten abgesehen werden. Dies kann helfen zu klären, wie Kollegenobjekte interagieren.

Diese Entkopplung im Vergleich zu RPC findet, wie von den Autoren Brose et al. ausgeführt, auf unterschiedlichen Ebenen statt:

- räumlich
- zeitlich
- syntaktisch
- semantisch

Die **räumliche** Entkopplung findet durch die Verwendung eines Vermittlers statt. In ereignisbasierten Systemen wird dieser Vermittler unter anderem als Channel (z. B. CORBA EventChannel), Queue (z. B. Java Swing) oder Router bezeichnet. Der Vermittler nimmt Benachrichtigungen von Quellen entgegen und leitet diese an Beobachter weiter. Damit müssen Quelle und Beobachter wechselseitig nicht mehr ihre Identität kennen. Beide haben nur noch eine 1:n-Beziehung zum Vermittler.

Die **zeitliche** Entkopplung wird ebenfalls durch Einsatz des Vermittlers erreicht und bedeutet, dass nicht alle Beobachter zum Zeitpunkt des Ereignisses verfügbar sein müssen. Lediglich der Vermittler muss für die Quelle verfügbar sein. Abhängig von der Verfügbarkeit eines Beobachters kann dieser zu einem weitaus späteren Zeitpunkt über das Ereignis benachrichtigt werden. Zu diesem Zeitpunkt muss die Quelle nicht mehr verfügbar sein.

Die **syntaktische** Entkopplung findet statt, da Quelle und Beobachter lediglich die generische Programmierschnittstelle des verwendeten Benachrichtigungsdienstes verwenden. Damit wird die unabhängige Entwicklung, Wiederverwendung und Portierung von Quelle und Beobachter wesentlich vereinfacht.

Da ein Ereignis lediglich Daten enthält, werden Quelle und Beobachter auch **semantisch** entkoppelt. Die Quelle braucht keine Annahmen darüber zu treffen, auf welche Art die Benachrichtigungen verarbeitet werden.[5] Genau genommen kann sie keine Annahmen darüber treffen, ob die Benachrichtigung überhaupt zur Kenntnis genommen wird.

Die Realisierung dieser Konzepte kann eine effektive Lösung für die eingangs beschriebenen Probleme (synchrone Kommunikation, enge Kopplung und Unicastverbindungen) erreicht werden.

[5] Die Autoren Brose et al. weisen darauf hin, dass die Beobachter jedoch sehr wohl ein Verständnis über die Semantik der Benachrichtigungen haben müssen, um diese sinnvoll verarbeiten zu können

2.3.2. Kategorisierung/Vergleichskriterien

Die Vielzahl der Anwendungsgebiete von Benachrichtigungsdiensten macht es schwierig, einen universell einsetzbaren Benachrichtigungsdienst zu realisieren. Daher existieren in der Praxis entsprechende Systeme in verschiedenen Ausprägungen, um die unterschiedlichen Anforderungen zu erfüllen. Um nun verschiedene Systeme vergleichen zu können, sind Vergleichskriterien notwendig. Die Autoren Carzaniga und Wolf schlagen folgende Vergleichskriterien vor [14]:

- Datenmodell. Welche Datentypen können innerhalb von Nachrichten verwendet werden? Datentypen, Format und Struktur.
- Welche Mechanismen sind vorhanden, um Beobachtern eine Auswahl der Nachrichten zu erlauben? Wie ist Syntax und Semantik der Filtersprache? Welcher Bereich der einzelnen Benachrichtigungen ist für die Evaluierung eines Filters sichtbar?
- Topologie der beteiligten Knoten. Wie werden Knoten eingebunden oder entfernt? Mit welchen Protokollen kommunizieren die Knoten? Ist eine Authentisierung der Knoten möglich?
- Welche Kommunikationsprotokolle existieren zwischen Knoten und Clients? Welche Basisfunktionalität und welche Administrationsoperationen können verwendet werden?
- Interne Architektur der Knoten. Physikalische Komponenten und logische Komponenten wie Matching- und Routing-Algorithmen oder Datenstrukturen.

In Anlehung an diese Arbeit folgt eine Kategorisierung der Kriterien.

Datenmodell

Struktur Bei einem Benachrichtigungsdienst ist das Datenmodell wichtig, in dem die Benachrichtigungen veröffentlicht werden. Zentral ist die Strukturierung der Daten.

Im einfachsten Fall sind die Nachrichten nicht strukturiert. Lediglich Sender und Empfänger haben Kenntnis der Struktur und des Inhalts einer Nachricht. Es können vom Benachrichtigungsdienst keinerlei Entscheidungen aufgrund einzelner Nachrichten durchgeführt werden.

Listen von einfachen Datentypen, z. B. *String*, sind eine weitere mögliche Struktur, die vor allem bei den älteren Internet-Protokollen wie e-Mail und USENET-News eingesetzt wird.

Strukturierte Recordtypen, die sich namensbasiert oder positionsbasiert dereferenzieren lassen, sind bereits deutlich näher an übliche Datentypen aus Programmiersprachen angelehnt.

Schließlich lassen sich rekursive Strukturen einsetzen. So können Nachrichten beispielsweise in Form von LISP-Ausdrücken formuliert werden. Heutzutage weit verbreitet ist die Verwendung von XML-Dokumenten.

Datentypen Im einfachen Fall können nur vordefinierte Wertebereiche (Domains) in Benachrichtigungen verwendet werden. Dies ist einfach zu realisieren, jedoch nicht sehr flexibel. Vordefinierte Wertebereiche bieten die Möglichkeiten zu Optimierungen der eingesetzten Filter- und Routing-Algorithmen, da mehr Annahmen über die Nachrichten getroffen werden können.

Eine mächtige Möglichkeit besteht darin, selbstdefinierte getypte Strukturen zu verwenden. Ein Datenmodell, das in einer Applikation verwendet wird, kann dadurch auch einfacher im Benachrichtigungsdienst eingesetzt werden.

Einschränkungen Das Datenmodell eines Benachrichtigungsdienstes kann zahlreiche Einschränkungen gegenüber dem Typsystem der verwendeten Programmiersprache aufweisen. So kann die Gesamtnachrichtengröße oder die Anzahl der verfügbaren Attribute beschränkt sein.

Ursachen der Einschränkungen können in den zugrundeliegenden Transportprotokollen liegen. Falls beispielsweise die Anforderung besteht, eine Nachricht in einem einzelnen Netzwerkpaket zu versenden, dürfen die Nachrichten eine bestimmte Größe nicht überschreiten.

Weitere Einschränkungen können bei den verwendbaren Datentypen existieren. So ist in einem System denkbar, dass nur Datentypen eingesetzt werden können, die eine bestimmte Schnittstelle unterstützen oder von einer speziellen Basisklasse abgeleitet sind.

Bei rekursiven Typen kann eine Obergrenze für die Anzahl bzw. Tiefe der Verschachtelungsebenen existieren.

Subskriptionsmodell

Der zweite zentrale Aspekte eines Benachrichtigungsdienstes ist das Subskriptionsmodell. Das Subskriptionsmodell beschreibt die Mechanismen, mittels derer ein Beobachter eine Auswahl der für ihn interessanten Benachrichtigungen beschreibt.

Sichtbarkeitsbereich Im engen Zusammenhang mit der Struktur einer Nachricht steht der Sichtbarkeitsbereich einer Subskription. Dies beschreibt, welcher Teil einer Nachricht durch eine Subskription evaluiert werden kann. Im einfachsten Fall existiert ein einzelnes, global bekanntes Attribut, das zur Evaluierung zur Verfügung steht (Channel-based Subscription). Nur wenig flexibler ist die Einführung eines einzelnen Attributes pro Nachricht, das evaluiert werden kann (Subject-based Subscription). Am flexibelsten ist die Möglichkeit, die gesamte Publikation evaluieren zu können (Content-based Subscription).

Publikation neuer Nachrichtentypen Um eine Subskription zu formulieren, benötigt ein Beobachter Kenntnis über die Struktur der zur Verfügung stehenden Nachrichten. Anhand dieser Informationen kann sich der Beobachter einen ersten Überblick über das verfügbare Informationsspektrum machen. Die Strukturinformationen werden verwendet, um Filterausdrücke formulieren zu können.

Die Struktur der Nachrichten ist bei Subject-based Subscription und Channel-based Subscription vorgegeben, daher sind dort keine Mechanismen zur Publikation neuer Nachrichtentypen sinnvoll. Die Struktur ist dem Client implizit bekannt.

Bei Systemen, die Content-based Subscription verwenden, ist diese Struktur jedoch nicht vorgegeben. Daher wird das publish-subscribe Modell um den Begriff *advertise* erweitert. So wird Quellen ermöglicht, Meta-Informationen über die Nachrichten zu veröffentlichen, die sie zu versenden beabsichtigen.

Ein Beobachter kann anhand dieser Informationen beispielsweise entscheiden, ob es für ihn interessant ist, sich an dem Benachrichtigungsdienst anzumelden. Die Informationen können auch innerhalb eines Benachrichtigungsdienstes verwendet werden. So beschreiben die Autoren Hinze und Bittner einen Algorithmus, der aus Subskriptionsinformationen interessierter Beobachter eine Struktur generiert, um konkrete Nachrichten effizient den einzelnen Subskriptionen zuzuordnen. Der Algorithmus benötigt dazu Attribute und Wertebereiche der einzelnen Nachrichtentypen, wie sie innerhalb der Publikationsinformationen zur Verfügung stehen können [32].

Mächtigkeit der Abfragesprache Je breiter der Sichtbarkeitsbereich der Subskription wird, desto wichtiger ist eine mächtige Abfragesprache, um eine effektive Auswahl der Nachrichten treffen zu können.

Abfragesprachen können deklarativ oder imperativ sein. Einfache Sprachen unterstützen nur einfache Prädikate, die implizit disjunktiv oder konjunktiv verknüpft sind. Mächtigere Sprachen erlauben es, einfache Ausdrücke mit Verknüpfungsoperatoren zu komplexeren Ausdrücken zusammenzusetzen.

Kommunikationsprotokolle

Ereignisquellen, Vermittler und Beobachter bilden einen Graphen, dessen Kanten die Kommunikationsverbindungen sind (vgl. Abschnitt 2.3.2). Man kann Client/Server und Server/Server Verbindungen unterscheiden. Insbesondere existieren unterschiedliche Möglichkeiten, wie ein Client auf die Schnittstellen des Benachrichtigungsdienstes zugreifen kann.

In lokalen Systemen werden dazu üblicherweise *Shared Memory* oder *Interprocess Communication* (IPC) Techniken eingesetzt, zum Zugriff auf entfernte Systeme sinnvollerweise eine Middlewaretechnik wie CORBA oder RMI.

Verwendet ein Benachrichtigungsdienst mehrere Server, kann ein Server-/Server Protokoll zum Einsatz kommen. Einerseits versenden die Server die Nachrichten, die sie von ihren Clients erhalten. Andererseits kann ein Server Informationen über die Subskriptionen seiner Clients versenden. Andere Server

können aufgrund dieser Informationen bereits eine Vorauswahl der Nachrichten treffen, die sie an diesen Server senden.

Topologie der beteiligten Knoten

Dieser Abschnitt beschreibt, in welchen Topologien die an einem Benachrichtigungssystem beteiligten Knoten (Quelle, Beobachter und Vermittler) miteinander verbunden sind.

Zentralisierte Architektur Benachrichtigungsdienste mit zentralisierter Architektur setzen einen Message-Server ein. Der Message-Server, auch Message-Router oder Message-Broker genannt, ist dafür verantwortlich, Nachrichten von Clients entgegenzunehmen und sie den anderen Clients zuzustellen. Der Message-Server entkoppelt den Sender von den Empfängern.

Ein Client kennt nur den Message-Server. Clients können sich am Server an- und abmelden, ohne dass andere Clients davon Notiz nehmen müssen.

In Abbildung 2.9 ist der schematische Aufbau einer zentralisierten Architektur gezeigt. Wie zu erkennen, führt der Einsatz einer zentralisierten Architektur zu einer minimalen Anzahl an Verbindungen. Trotzdem können alle Beteiligten miteinander kommunizieren.

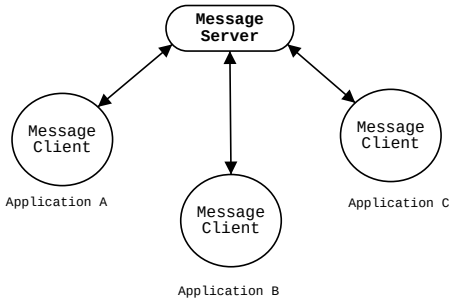

Abbildung 2.9.: Zentralisierte Architektur

Dezentralisierte Architekturen In dezentralisierten Architekturen kommt üblicherweise IP Multicast als Transportprotokoll zum Einsatz. Ein Benachrichtigungsdienst, der auf eine dezentralisierte Architektur setzt, verwendet keinen zentralen Server. Ein Teil der Serverfunktionalität (Persistenz, Transaktionen, Sicherheit) wird in die Clients ausgelagert. Das Zustellen (Routing) der Nachrichten wird an das Mulicastprotokoll der Netzwerkschicht delegiert (vgl. Abb. 2.10 auf der nächsten Seite).

Im Gegensatz zur zentralisierten Architektur benötigt eine dezentralisierte Architektur keinen zentralen Server für das Routing der Nachrichten. Sollte weitere Funktionalität wie Persistenz oder Dienstgüte notwendig sein, so wird diese zum Client ausgelagert.

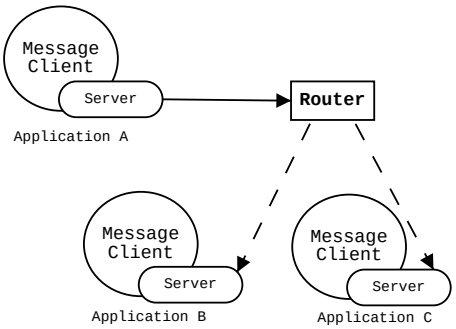

Abbildung 2.10.: Dezentralisierte Architektur

Hybride Architekturen Hybride Architekturen kombinieren Aspekte zentralisierter und dezentralisierter Architekturen. So könnten beispielsweise Clients mit einem lokalen Server verbunden sein (zentralisierte Architekur), der seinerseits wiederum mit weiteren Servern über Multicast kommuniziert (dezentralisierte Architektur).

Die verschiedenen Server sind dabei in unterschiedlichen Topologien miteinander verbunden. Zwischen den Servern kann ein spezielles Server/Server Protokoll zum Einsatz kommen.

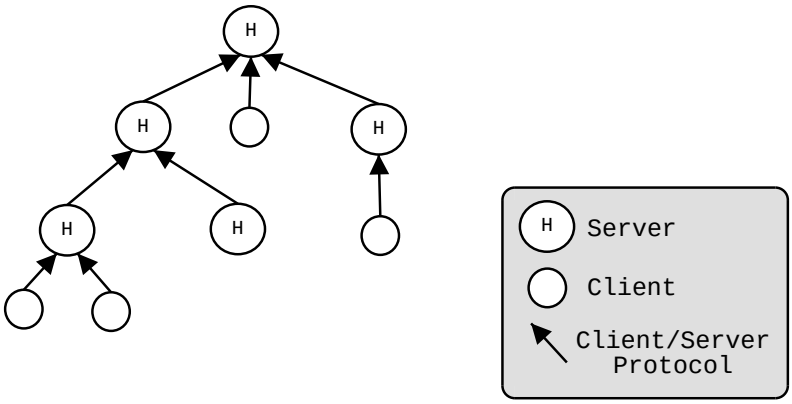

Abbildung 2.11.: Hierarchische Server-Topologie

Hierarchische Topologie Jeder Server verfügt über eine bestimmte Anzahl von Clients. Dies können Quellen, Beobachter oder weitere Server sein. Der Server kann wiederum Client eines übergeordneten Servers sein. Damit bilden die Server einen gerichteten zyklischen Graphen oder Baum (siehe Abbildung 2.11).

Diese Topologie ist einfach zu realisieren, da das Client/Server Protokoll auch für die Kommunikation zwischen den Servern eingesetzt werden kann. Sie ist die natürliche Erweiterung der zentralisierten Architektur. Ein Server muss lediglich um die Clientfunktionalität erweitert werden. Der Server meldet sich dann bei einem anderen Server als Client an. Der Server sendet alle Benachrichtigungen, die er von seinen Clients erhält, an seinen übergeordneten Server

und leitet alle Benachrichtigungen, die er von seinem übergeordnetem Server erhält, an seine Clients weiter.

Die Konfiguration solcher Topologien gestaltet sich ebenfalls als sehr einfach. Mehrere autarke Subnetze können einfach miteinander verbunden werden, indem die Server der Subnetze sich an einem gemeinsamen Toplevel-Server anmelden. Diese Topologie wird im USENET News System, im JEDI System und bei Keryx angewendet.

Mit wachsender Anzahl von Teilnehmern wächst die Last für die höher im Baum angeordneten Server. Außerdem stellen die Wurzelserver einen kritischen Ausfallpunkt (Single Point of Failure) dar.

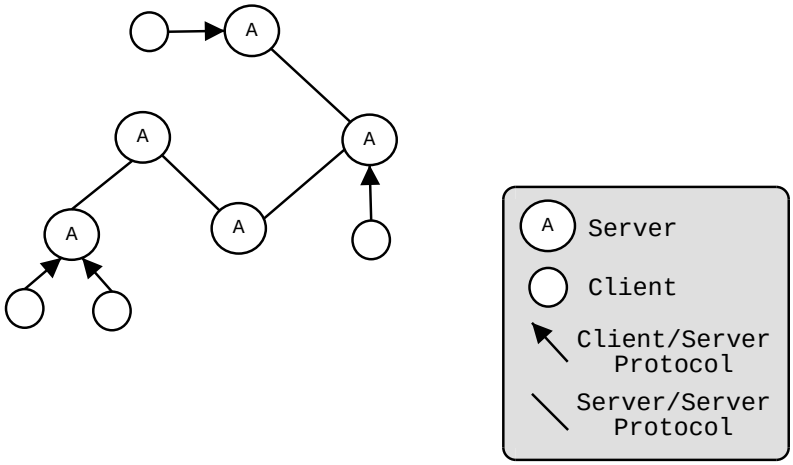

Abbildung 2.12.: Azyklisches Peer-to-Peer Netz

Azyklisches Peer-to-Peer Netz In einem azyklischen *Peer-to-Peer* Netz kommunizieren Server mittels eines bidirektionalen Server/Server Protokolls. Die Server bilden dabei einen azyklischen ungerichteten Graphen (vgl. Abb. 2.12). Besonderes Augenmerk muss bei dieser Topologie auf die Konfiguration des Netzes gelegt werden. Beim Einbinden neuer Server in ein bestehendes Netzwerk dürfen keine Zyklen entstehen, da die Algorithmen, die im System verwendet werden, sonst scheitern können. In einem zentral administrierten System kann dies durch Sorgfalt bei der Konfiguration erreicht werden. Spätestens in einem dezentral administrierten System müssen jedoch weitere Mechanismen wie automatische Konfiguration existieren, um die Konsistenz des Netzwerkes zu gewährleisten. Ähnlich der hierarchischen Topologie existieren keine redundanten Verbindungen. Damit können durch Ausfall eines Servers mehrere Subnetze nicht mehr miteinander kommunizieren.

Allgemeines Peer-to-Peer Netz Die Server in einem allgemeinen Peer-to-Peer Netz bilden einen ungerichteten Graphen, in dem auch Zyklen erlaubt sind (siehe Abbildung 2.13 auf der nächsten Seite). Diese Topologie ist leicht zu konfigurieren und damit sehr flexibel. Durch die redundanten Verbindungen

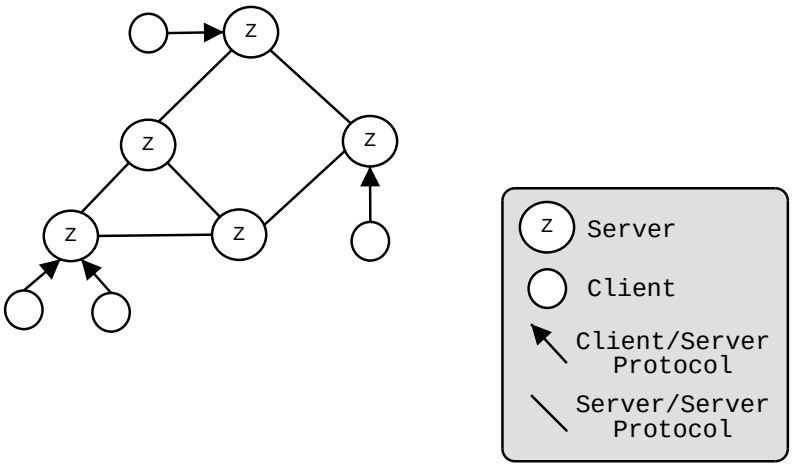

Abbildung 2.13.: Zyklisches Peer-to-Peer Netz

ist das System außerdem fehlertolerant gegenüber dem Ausfall einzelner Server. Erhöhter Aufwand muss jedoch bei der Realisierung der verwendeten Algorithmen getrieben werden. Für Verbindungen zwischen zwei Knoten müssen die kürzesten Pfade berechnet werden. Dabei müssen Zyklen im Graph erkannt und vermieden werden.

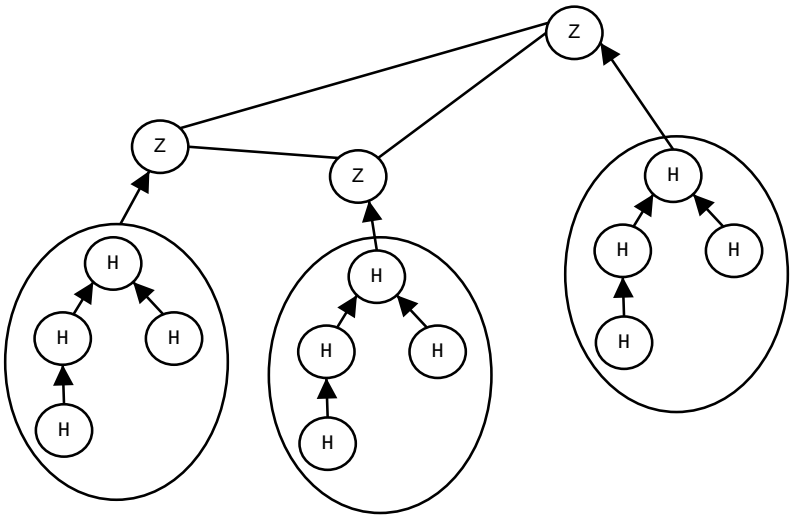

Abbildung 2.14.: Hybride Topologie: hierarchisch/zyklisch

Hybride Topologie Hybride Topologien erlauben es, Subnetze unterschiedlicher Topologien miteinander zu verbinden. In Abbildung 2.14 sind mehrere Subnetze, die eine hierarchische Topologie betreiben, miteinander über ein allgemeines Peer-to-Peer Netz verbunden.

Ein anderes Szenario, wie in Abbildung 2.15 auf der nächsten Seite dargestellt, wären mehrere Subnetze, die ein allgemeines Peer-to-Peer Netz betreiben.

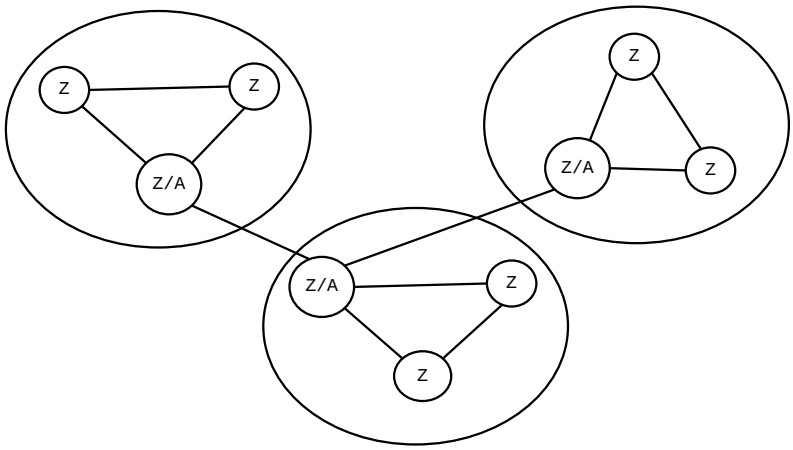

Abbildung 2.15.: Hybride Topologie: zyklisch/azyklisch

In jedem Subnetz wird dabei ein Server als Gateway definiert. Die Gateways aller Subnetze sind mit den anderen Gateways zu einem azyklischen Peer-to-Peer Netz verbunden.

Routingstrategien

In einem zentralisierten System werden die Nachrichten von den Clients an eine zentrale Serverinstanz gesendet. Dieser Server entscheidet dann aufgrund der Subskriptionsinformationen, an welche Clients die Nachricht weitergeleitet werden soll. Es kommt daher nicht zu unnötigen Netzwerkverbindungen.

Sobald in einer dezentralisierten oder hybriden Architektur mehrere Server aktiv sind, kommt es ohne entsprechende Vorkehrungen zu unnötigem Netzwerkverkehr. Sind beispielsweise die Server in einer Baumstruktur organisiert (vgl. 2.3.2), sendet ein Server *B* alle von seinen Clients empfangenen Nachrichten an den ihm übergeordneten Server *A*. Im Gegenzug leitet er Nachrichten, die von *A* kommen, an seine Clients weiter. Hat Server *B* keine Clients, denen er Nachrichten senden kann, so erhält er trotzdem weiterhin Nachrichten von *A*. Dies stellt eine Vergeudung von Serverressourcen und Netzwerkbandbreite dar.

In einem Benachrichtigungssystem, das auf einer dezentralisierten bzw. hybriden Architektur basiert, bieten sich daher folgende Strategien an:

Späte Vervielfachung der Nachrichten Soll eine Nachricht an mehrere Empfänger zugestellt werden, so wird stets nur eine Kopie der Nachricht versandt. Die Nachricht wird erst so kurz wie möglich vor dem Ziel dupliziert und dann den Empfängern zugestellt (vgl. Abb. 2.16 auf der nächsten Seite). Dieses Prinzip ist dem Multicast-Routing eng verwandt. Durch diese Technik wird gewährleistet, dass kein unnötiger Verkehr den gleichen Netzwerkabschnitt durchzieht.

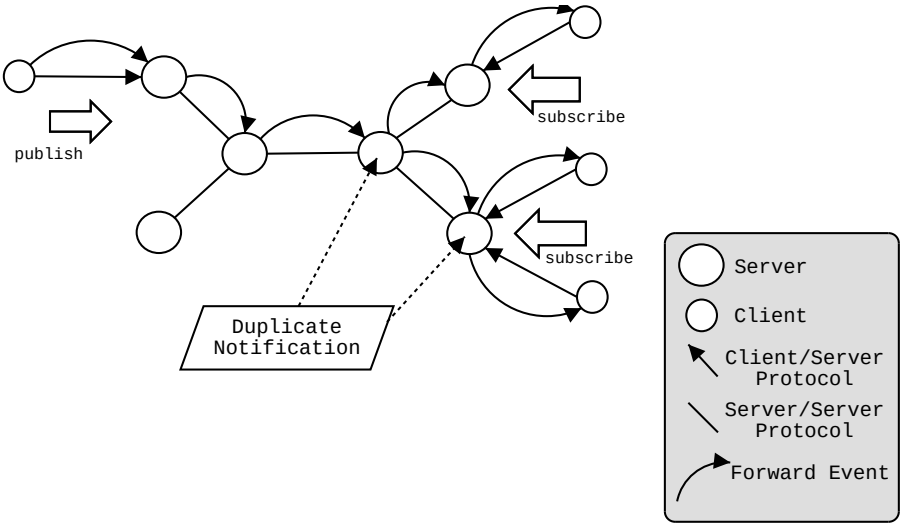

Abbildung 2.16.: Späte Vervielfachung der Nachrichten

Frühe Evaluierung der Filter Eine weitere Möglichkeit, unnötigen Netzwerkverkehr zu vermeiden, wird durch eine möglichst frühe Evaluierung der Filter erreicht. Falls also ein Client einen Filter verwendet, wird der Filter so nah wie möglich an die Ereignisquelle platziert, um unnötigen Nachrichtenaustausch zu minimieren (vgl. Abbildung 2.17). Um diese Anforderungen zu realisieren,

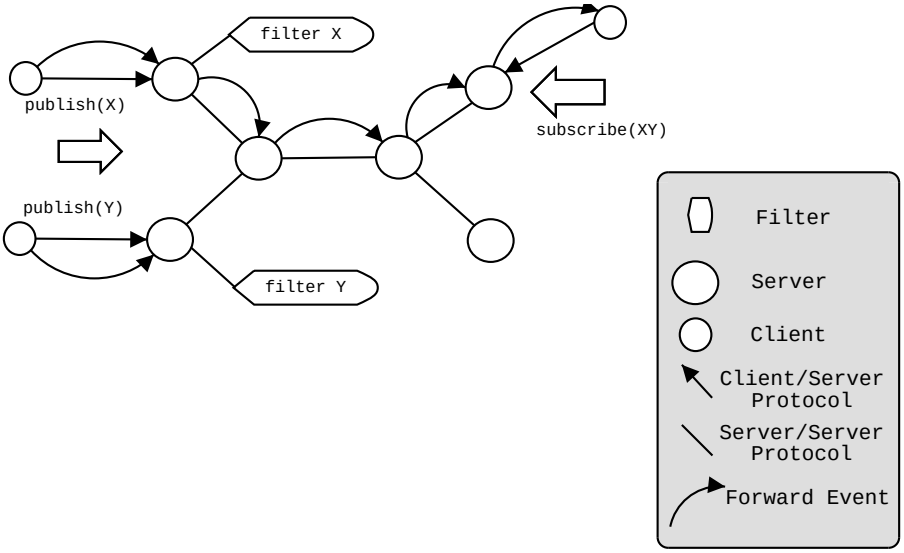

Abbildung 2.17.: Frühe Evaluierung der Filter

kommen innerhalb verteilter Architekturen Algorithmen zum Einsatz, um [9]:

1. Subskriptionen im Servernetz zu verbreiten, und
2. Ankündigungen im Servernetz zu verbreiten.

Verbreitung von Subskriptionen Durch die Verbreitung von Subskriptionsinformationen im Servernetz wird ein Routingpfad definiert, der später bei Auslieferung von Nachrichten verwendet wird. Ein Server, der eine Subskription von einem seiner Clients erhält, speichert diese und leitet sie an alle ihm bekannten Server weiter. Die Server speichern und leiten die Subskription ebenfalls weiter. Damit kennt das gesamte Servernetz eine bestimmte Subskription. Empfängt nun einer der Server eine Nachricht von einem seiner Clients, die zu der Subskription passt, muss er die Nachricht lediglich an den Server weiterleiten, von dem er die Subskription ursprünglich erhalten hat.

Verbreitung von Ankündigungen Systeme, die aufgrund der Subskriptionen Routingentscheidungen treffen, können weiter von der Ankündigung (advertisement) neuer Nachrichtentypen profitieren. Dabei veröffentlicht ein Client zunächst Informationen über die Struktur der Nachrichten, die er zu versenden beabsichtigt. Diese Information wird auf die gleiche Weise wie die Subskriptionen im Servernetz verbreitet. Ein Server, der nun eine Subskription im Netz verbreiten möchte, muss die Subskription lediglich an Server weiterleiten, die vorher eine passende Ankündigung versendet haben.

Überlappende Subskriptionen Wie weiter oben beschrieben, werden Subskriptionen durch das Netzwerk verbreitet. Werden viele Subskriptionen eingetragen, kann dies einen erheblichen Netzwerkverkehr verursachen. Aus diesem Grund definiert Carzaniga den Begriff der sich überlappenden Subskriptionen. Dies sei an einem Beispiel verdeutlicht:

Angenommen, ein Server hat einen Client, der die Subskription `value < 20` aktiviert hat. Der Server wird diese Subskription an die ihm benachbarten Server weiterleiten. Existiert an diesem Server ein zweiter Client, der später die Subskription `value = 10` aktiviert, so muss der Server diese Subskription nicht weiterleiten, da die Subskription des ersten Client die des zweiten Client überlappt. Erst wenn der erste Client seine Subskription deaktiviert, muss die zweite Subskription verbreitet werden.

Abhängig von der Komplexität der Filtersprache kann es schwierig bis unmöglich sein festzustellen, ob zwei Filterausdrücke sich überlappen.

Dienstgüte

Die variierenden Einsatzbereiche eines Benachrichtigungsdienstes definieren unterschiedliche Anforderungen an dessen Dienstgüte (Quality of Service).

Bei einem synchronen Methodenaufruf hat der Aufrufer, nachdem der Methodenaufruf ohne das Auslösen einer Ausnahme beendet wurde, die implizite Kenntnis, dass die Nachricht beim Empfänger entgegengenommen wurde. Bei asynchroner Benachrichtigung entfällt dies.

In manchen Szenarien kann es tolerierbar sein, wenn einzelne Nachrichten durch Ausfall des Benachrichtigungsdienstes, der Kommunikationsverbindungen oder zeitweiliger Nicht-Verfügbarkeit des Empfängers verloren gehen kön-

nen. Dies ist beispielsweise bei Newstickern der Fall. Diese Form der Zuverlässigkeit wird als *best-effort* oder *at-most-once* bezeichnet. In Szenarien, wo dies nicht tolerabel ist, kommt die garantierte Auslieferung von Nachrichten zum Einsatz. Dies wird als *persistent* oder *at-least-once* bezeichnet. Dabei wird die Nachricht zuerst vom Benachrichtigungssystem gespeichert und dann den Empfängern zugestellt (*store-and-forward*). Fällt der Benachrichtigungsdienst aus oder schlägt eine Auslieferung fehl, so kann versucht werden, die Nachricht zu einem späteren Zeitpunkt erneut zuzustellen. Dabei kann es zur mehrfachen Auslieferung der gleichen Nachricht an einen Empfänger kommen. Ist dies nicht erwünscht, existiert die *exactly-once* Semantik.

Bestimmte Nachrichten können wichtiger als andere sein. Um dies auszudrücken, können einzelnen Nachrichten oder Gruppen von Nachrichten Prioritäten zugeordnet werden. Höher priorisierte Nachrichten werden vor niedriger priorisierten Nachrichten ausgeliefert.

Des weiteren existieren noch zahlreiche Optionen, die ein Benachrichtigungsdienst im Zusammenhang mit der Dienstgüte beachten kann. Dazu gehören die maximale Anzahl der zwischenzuspeichernden Nachrichten und die Auswahl der Strategie, die angewandt wird, wenn interne Puffer oder Warteschlangen überzulaufen drohen.

Weiteres

Es bleibt, einige wichtige, offen gebliebene Punkte zu nennen, die bei Realisierung und Einsatz eines Benachrichtigungsdienstes relevant sind, hier aber nicht vertieft werden.

- Portierbarkeit
- Security
- Ortstransparenz

3. Ein Vergleich nachrichtenbasierter Middleware

Im folgenden werden die Messaging-Plattformen Websphere MQ, JMS, SIENA und CORBA Event/Notification Service einander gegenübergestellt. Das vielfach eingesetzte Websphere MQ Eventbroker stammt von IBM. JMS ist eine standardisierte Programmierschnittstelle zu einem Benachrichtigungssystem. Der Einsatz legt nicht auf eine spezielle Implementierung fest. SIENA ist ein Forschungsprototyp der Universität von Colorado. Diese Systeme werden ausführlich vorgestellt. Als Grundlage dienen die im letzten Kapitel erarbeiteten Kriterien.

3.1. Message oriented Middleware

Das Konzept eines Benachrichtigungsdienstes ist nicht neu. Es existieren Systeme wie IBM MQSeries, Microsoft Message Queue Server (MSMQ), TIBCO Rendezvous, Open Horizon Ambrosia und Modulus InterAgent.[1]

MOM Systeme waren so populär, dass eine Organisation mit dem Ziel gegründet wurde, MOM zu standardisieren. *Message oriented Middleware Association* (MOMA) ist ein Konsortium von Herstellern, Benutzern und sonstigen Interessenten, die das Ziel haben, die Interoperabilität zwischen verschiedenen MOM-Produkten zu erhöhen. Die Homepage der MOMA ist jedoch nicht mehr registriert.

Die einzelnen MOM-Systeme sind proprietär und daher untereinander nicht interoperabel. Die Arbeit der MOMA bestand daher vor allem darin, eine Interoperabilität zwischen unterschiedlichen Betriebssystemen, Hardwareplattformen, Programmiersprachen und den einzelnen MOM-Systemen zu erreichen. Für einzelne Kombinationen von MOM-Systemen existieren Bridges und Gateways zur Konvertierung der Nachrichtenformate.

Stellvertretend für verschiedene Systeme wird hier Websphere MQ Event Broker von IBM vorgestellt. Als Quellen wurden frei erhältliche Produkthandbücher von IBM verwendet [34, 35].

Websphere MQ ermöglicht es Komponenten, miteinander über ein Netzwerk zu kommunizieren. Es existieren Sprachanbindungen für C, C++, COBOL und Java. Zusätzlich zu zwei proprietären Schnittstellen kann Websphere MQ über eine JMS Schnittstelle angesprochen werden. Darüber hinaus wird Anwendungen die Möglichkeit geboten, über Gateways und Adapter auf Systeme und

[1] Für einen weiteren Überblick verschiedener MOM-Systeme siehe [40]

Schnittstellen von Produkten wie Lotus Domino, Microsoft Exchange/Outlook und SAP/R3 zuzugreifen. Mit Websphere ist sowohl point-to-point als auch publish-subscribe Kommunikation möglich.

Nachrichten sind jeweils einem Thema (Subject) zugeordnet. Ein Thema wird durch eine beliebige Zeichenfolge beschrieben. Das Trennzeichen / kann verwendet werden, um die hierarchische Struktur eines Themenbereichs darzustellen:

- `Europa/Deutschland/Berlin`
- `Europa/Deutschland/Hamburg`
- `Europa/Niederlande/Amsterdam`

Interessierte Empfänger können eine Subskription zu einem Thema einrichten. Dabei können in der Subskription Platzhalter verwendet werden:

- Platzhalter für mehrere Ebenen.
  Die Subskription `Europa/#` beschreibt bspw. Nachrichten zu den Themen `Europa`, `Europa/Deutschland` und `Europa/Deutschland/Berlin`.
- Platzhalter für eine Ebene.
  Die Subskription `Europa/+` umfasst lediglich die Nachrichten, die das Thema `Europa/Deutschland` haben, jedoch nicht Nachrichten mit den Themen `Europa` oder `Europa/Deutschland/Berlin`.

Mit Hilfe von Zugriffssteuerlisten (ACL) kann für jedes Thema und jeden Client festgelegt werden, ob ein Client das Recht zum Veröffentlichen von Nachrichten bzw. das Recht zum Einrichten einer Subskription hat.

Beim Einrichten einer Subskription kann zusätzlich zur Angabe des Themas ein inhaltsspezifischer Filter angegeben werden. Dieser erlaubt es, Nachrichten aufgrund ihres Inhalts auszuwählen. Nachrichten können auf folgende Art und Weise strukturiert sein:

- Die Nachricht ist eine selbstdefinierende XML Nachricht.
- Die Nachrichtenschablone ist in einem speziellen Headerfeld definiert.

Zur Formulierung der Filterausdrücke wird ESQL verwendet. ESQL ist eine Abfragesprache, die auf SQL-3 basiert [36]. Ein Filterausdruck kann beispielsweise wie folgt aussehen:

```
Body.Name LIKE 'Schu%'
```

Das bedeutet, dass der Inhalt des Feldes `Name` im Hauptteil einer Nachricht extrahiert und mit der angegebenen Zeichenfolge verglichen wird.

Die Felder, die innerhalb eines Filters evaluiert werden können, werden über ihren Namen identifiziert. Folgende Namen können verwendet werden:

Root Dieser Name bezeichnet die Wurzel der Nachricht.

Properties Dieser Anteil der Nachricht enthält die Headerfelder.

Body Dieses Feld enthält die Nutzlast der Nachricht.

Weiterer Name Anderen Namen, die nicht in eine der oberen Kategorien fallen, wird implizit das Präfix *Body* vorangestellt.

Ein Feld kann strukturierte Werte enthalten. Diese Werte können namensbasiert oder positionsbasiert dereferenziert werden:

- `Person.salary`.
- `Body.person.Address[0]`.
- `Properties.Topic`.
- `Root.MQMD.UserIdentifier`.

Die eigentliche Evaluierung der Filterausdrücke findet nur auf den primitiven Basistypen *String*, *Integer*, *Boolean*, *Float* statt. Die Nachrichten können zwar weitere Datentypen, wie z. B. *Time*, *Byte* oder *Character* enthalten, diese werden jedoch in die Basistypen konvertiert. Dabei sind verschiedene automatische Umwandlungen definiert. Die Zeichenfolge TRUE wird beispielsweise in den Booleschen Wahrheitswert konvertiert.

Mit dem Mengenoperator `IN` lässt sich prüfen, ob ein Element in einer Menge enthalten ist.

```
value IN set1, set2, ...
```

Um Zeichenketten miteinander zu vergleichen, existiert der Operator `LIKE`:

```
string1 LIKE string2
```

Bei Verwendung dieses Operators können zwei Platzhalter verwendet werden. Das Zeichen % steht für beliebig viele unbekannte Zeichen, das Zeichen _ steht für ein unbekanntes Zeichen. Soll explizit nach % oder _ gefragt werden, so müssen diese durch Verwendung von \ maskiert werden.

Der Operator `BETWEEN` erlaubt zu prüfen, ob ein Wert in einem bestimmten Wertebereich liegt:

```
expr BETWEEN low AND high
```

Wenn ein Filterausdruck, der für eine Nachricht ausgewertet wird, Feldnamen enthält, die nicht in der Nachricht enthalten sind, ist das Ergebnis bei manchen Operatoren undefiniert. Um zu prüfen, ob ein Feldname definiert ist bzw. einen gültigen Wert enthält, existiert der Operator `IS NULL`:

```
expr IS [NOT] NULL
```

Die algebraischen Operatoren `<>`, `<`, `>`, `<=`, `>=`, `=` erlauben die dynamische Berechnung von Werten, während ein Filterausdruck evaluiert wird. Zusätzlich existieren die arithmetischen Operatoren `+`, `-`, `*`, `/`.

Komplexe Ausdrücke lassen sich mit Hilfe von `AND`, `NOT`, `OR` kombinieren:

```
Person.Address[1] NOT LIKE Blen% AND Person.Salary > 15000
```

Mehrere Broker können zu einem Brokerverbund zusammengefasst werden. Zur Konfiguration eines Brokerverbundes existiert eine spezielle Applikation, das Control Center. Das Control Center erlaubt es, mehrere Broker zu einem azyklischen Netz zu verbinden. Das Control Center prüft die Konsistenz des Brokerverbundes und erlaubt beispielsweise keine Erzeugung von Zyklen. Darüber hinaus können mit dem Control Center die Nachrichtenflüsse kontrolliert und Zugriffsrechte konfiguriert werden.

Um Nachrichten miteinander auszutauschen, speichert jeder Broker Subskriptionen seiner lokalen Clients und Informationen zu Subskriptionen auf anderen Brokern. Wenn ein Client eine Subskription einrichtet, richtet der Broker eine entsprechende Subskription bei seinen benachbarten Brokern ein. Dies wird als Proxy-Subskription bezeichnet. Wenn bereits eine identische Subskription eingerichtet wurde, legt der Broker keine erneute Subskription an, d. h. es ist immer nur eine gültige Proxy-Subskription vorhanden.

Löscht ein Client eine Subskription bei einem Broker und ist der Client der letzte (oder einzige) Client, für den der Broker die Proxy-Subskription eingerichtet hat, so löscht der Broker auch die entsprechende Proxy-Subskription bei seinen benachbarten Brokern.

Inhaltsspezifische Filter sind in Proxy-Subskriptionen nicht enthalten. Daher empfängt der Broker, bei dem ein Client eingetragen ist, der einen Inhaltsfilter angegeben hat, eine Obermenge von Nachrichten. Die Filterung erfolgt erst im Broker, der die Nachricht letztendlich an den Empfänger weitergibt.

Die einzelnen Broker bilden zwar ein azyklisches Netz, aber, wie ausgeführt, werden keine speziellen Routingmechanismen, wie in Abschnitt 2.3.2 beschrieben, angewendet. Zum Nachrichtenaustausch richtet ein Broker eine Proxy-Subskription bei einem anderen Broker ein. Die beiden Broker stehen also in einer Client/Server Beziehung. Aus diesem Grund ist MQSeries der hierarchischen Architektur zuzuordnen.

MQSeries bietet unterschiedliche Dienstgüteeinstellungen pro Nachricht: *at-most-once*, *at-least-once*, *exactly-once*. Die persistente Speicherung der Daten erfolgt dabei in einer Datenbank (DB2, Oracle, MS SQL, Sybase). Es existieren weitere dienstgütebezogene Eigenschaften, beispielsweise die Lebensdauer einer Nachricht.

3.2. SIENA

Scalable Internet Event Notification Architectures (SIENA) ist ein Forschungsprojekt, das vom Software Engineering Research Laboratory der University of Colorado betrieben wird.

Ziel des Projektes ist, einen generischen, skalierbaren Benachrichtigungsdienst zu realisieren. Dabei wird eine hybride Architektur eingesetzt. Verteilte Server bilden miteinander ein Netzwerk, an dem sich Clients anmelden können.

Der Kompromiss zwischen Skalierbarkeit des Dienstes und Mächtigkeit der Filtermöglichkeiten wird bei SIENA besonders sorgfältig erforscht. Prinzipiell

gilt, dass der Einsatz einfacherer Filtermöglichkeiten die einfachere Realisierung eines skalierbaren Dienstes ermöglicht. Im einfachsten Fall, ohne Filtermöglichkeiten, kann die Kommunikation zwischen den Servern mit Multicast abgewickelt werden.

Mächtigere Filtermöglichkeiten erleichtern die Entwicklung von Applikationen, die den Benachrichtigungsdienst einsetzen. Mit steigender Komplexität der Filter wird es jedoch schwieriger, Optimierungen aufgrund statischer Analyse der Filter durchzuführen. Dies kann die Skalierbarkeit eines Dienstes einschränken.

SIENA trifft daher in allen relevanten Bereichen Entscheidungen, um einen optimalen Kompromiss zwischen Skalierbarkeit und Flexibilität zu erreichen.

Bei SIENA besteht eine Nachricht aus mehreren Attributen. Jedes Attribut ist ein Tripel der Form $attribute = (name, type, value)$. Der Name eines Attributes muss dabei stets eindeutig sein. SIENA unterstützt die einfachen Datentypen: *char*, *integer*, *boolean*, *float*, *string*, *byte-array* und *date*.

Zusätzlich existieren folgende Operatoren, die für die unterschiedlichen Datentypen überladen sind: `any`, `=`, `>`, `<`, `>*`, `*<`. Der Operator `any` ist ein *Dont't Care*-Operator, der für jedes Argument wahr ist. `>*` ist ein String Präfix Operator:

```
www.inf.fu-berlin.de >* www
```

Korrespondierend ist `*<` ein String Postfix Operator

```
www.inf.fu-berlin.de *< berlin.de
```

Name, Typ und Wert eines Attributes werden durch namensbasierte Dereferenzierung mit Hilfe des Dereferenzierungsoperators . addressiert. Ist beispielsweise α ein Attribut, dann bezeichnet $\alpha.name$ den Namen, $\alpha.type$ den Typ und $\alpha.value$ den Wert des Attributes.

Ein Filter beschreibt eine Menge von Nachrichten, indem Attributnamen, Typen und Einschränkungen auf deren Wertebereich definiert werden. Ein Filter besteht dabei aus einer Menge von Attributfiltern. Jeder Attributfilter hat dabei die Form

```
attribute_filter=(name, type, operator, value).
```

Mehrere Filter können mit der Hilfe von Patterns miteinander kombiniert werden. In SIENA existieren die Patterns Sequenz (.) logisches oder (|) und der Kleene Operator ($*$). Mit dem Sequenzpattern kann ein Ereignis ausgelöst werden, wenn zwei oder mehrere andere Ereignisse in einem bestimmten Zeitfenster auftreten. Dies entspricht einem passiven Ereignis (vgl. Abschnitt 2.3.1).

Die Server in einem SIENA-Netzwerk lassen sich sehr flexibel miteinander kombinieren. Insbesondere werden alle in Abschnitt 2.3.2 beschriebenen Topologien unterstützt.

Innerhalb des Netzes kommt ein intelligentes Routingverfahren zum Einsatz, das die Subskriptionen (subscription) und Ankündigungen (advertisement) der angemeldeten Clients berücksichtigt, um den Austausch von Nachrichten zwischen den Servern zu minimieren. Dabei werden die in Abschnitt 2.3.2 vorgestellten Routingverfahren eingesetzt.

3.3. Java Message Service

Java Message Service (JMS) ist ein Java API, das standardisiert wie Java Applikationen mit einem darunterliegenden Benachrichtigungsdienst, dem sog. Service-Provider, kommunizieren kann. JMS standardisiert darüber hinaus, wie der Service-Provider mit der Java Applikation kommunizieren kann. JMS definiert ähnlich wie JDBC oder JNDI ein API für Applikationsentwickler und ein API für die Entwickler des Service-Provider.

Das JMS API ist Teil der *Java 2 Enterprise Edition* (J2EE). Eine Implementierung des J2EE muss jedoch keine JMS Implementierung zur Verfügung stellen, um konform zu sein.

Teile der JMS Funktionalität verwenden die *Java Transaction Architecture* (JTA) und benötigen daher auch einen *Java Transaction Service* (JTS).

JMS bietet die Benachrichtigungsmodelle point-to-point und publish-subscribe an.

Ein Thema wird bei JMS als *Topic* bezeichnet. Es definiert einen virtuellen Kanal. Interessenten, die Nachrichten empfangen möchten, können sich für ein Thema subskribieren (engl. to subscribe). Jede vom Sender abgeschickte Nachricht wird an alle angemeldeteten Empfänger zugestellt. JMS unterstützt dabei lediglich push/push Kommunikation.

Alle Daten und Ereignisse in einer JMS Anwendung werden mit Hilfe von JMS Nachrichten versandt. In Abbildung 3.1 ist ein *Message*-Objekt, bestehend aus drei Teilen, dargestellt: Message Header, Message Properties und die eigentliche Nutzlast (Payload) oder Message Body. Es existieren unterschiedliche

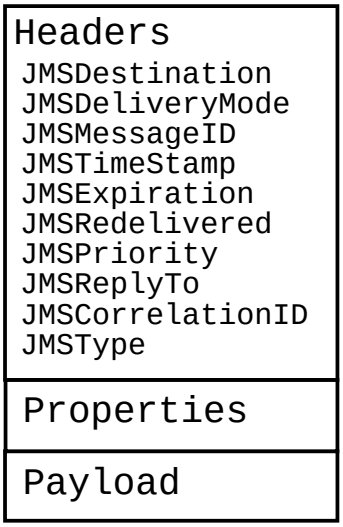

Abbildung 3.1.: Aufbau einer JMS Nachricht

Nachrichtentypen. Diese unterscheiden sich durch ihre Nutzlast. Es existieren Nachrichten mit strukturierter Nutzlast, wie *StreamMessage* oder *MapMessage*, und Nachrichten mit unstrukturierter Nutzlast wie *TextMessage*, *ObjectMessage* und *BytesMessage*.

Die Message Header enthalten standardisierte Metadaten, z. B. die Quelle der Nachricht, Gültigkeitsdauer, Routinginformationen und den Zeitpunkt der Erzeugung der Nachricht. Die meisten dieser Attribute werden automatisch vom JMS Service-Provider initialisiert, sobald die Nachricht dem System übergeben werden. Einige Attribute können vom Absender initialisiert werden.

Properties ermöglichen es, Nachrichten weiter an einen bestimmten Anwendungsbereich anzupassen. Einer Nachricht können beliebige Namen/Wertpaare zugeordnet werden, wobei als Werte die Java Typen *int*, *String*, *double*, *float*, *byte*, *long*, *short* und *Object* verwendet werden können. Die Properties können weitere Informationen über die Nachricht enthalten und kommen beim Filtern der Nachricht zum Einsatz.

Um Empfängern eine Auswahl der Nachrichten zu erlauben, existieren sog. Message-Selectors. Ein Message-Selector wird mit Hilfe eines Filterausdrucks initialisiert. Innerhalb des Filterausdrucks kann Bezug auf die Header und Propertyfelder einer JMS-Nachricht genommen werden.

Die Syntax der Filtersprache ist an SQL-92 angelehnt und hat große Ähnlichkeit mit ESQL (vgl. 3.1). Es können Prädikate definiert werden, die ähnlich den Prädikaten in einer `WHERE` Klausel sind.

Im folgenden Beispiel werden die Bezeichner `Age`, `Weight` und `Name` verwendet. Sie beziehen sich auf einen hypothetischen Nachrichtentyp, der die Properties Age vom Typ *int*, Weight vom Typ *double* und Name vom Typ *String* enthält. Diese Properties sind nutzerdefiniert und existieren zusätzlich zu in JMS vordefinierten Headerwerten.

```
Age < 30 AND Weight >= 100.00 AND Name = 'Smith'
```

Manche der vordefinierten JMS Header können ebenfalls innerhalb eines Filterausdrucks verwendet werden. Andere JMS Header können nicht innerhalb eines Filterausdrucks verwendet werden, da deren Wert keine String-Repräsentation besitzt.

Als Operatoren stehen die algebraischen Vergleichsoperatoren `=,>,>=,<,<=` und `<>` zur Verfügung. Die Vergleichsoperatoren können mit allen primitiven Typen verwendet werden. Eine Ausnahme bildet dabei der Typ *boolean*, der lediglich die Operatoren `=` und `<>` unterstützt.

Der LIKE Operator ermöglicht es, Werte vom Typ String miteinander zu vergleichen. Dabei können `%` und _ als Platzhalter für unbekannte Teile verwendet werden. _ steht für ein einzelnes und `%` für beliebig viele unbekannte Zeichen.

Der `BETWEEN` Operator wird verwendet, um Wertebereiche anzugeben:

```
Age BETWEEN 20 and 30
```

Der `IN` Operator testet auf Mengenmitgliedschaft. Die Existenz von Properties in Nachrichten kann mit dem `IS NULL` Operator geprüft werden. Die Operatoren `LIKE`, `BETWEEN`, `IN` und `IS NULL` können mit Hilfe von `NOT` negiert werden und mit `AND` und `OR` logisch verknüpft werden. Die Verwendung nicht existierender Properties in Filterausdrücken kann zu Problemen führen. Für diesen Fall ist der Wert *unknown* vorgesehen.

Zusätzlich zu den logischen Operatoren existieren die arithmetischen Operatoren +,-,/ und *, um Werte zum Zeitpunkt der Evaluierung berechnen zu können:

```
Weight NOT BETWEEN (Age * 5) AND (Height/Age * 2.23)
```

Die JMS Spezifikation definiert lediglich die Schnittstelle zwischen JMS Service Provider und Anwender. Ob die darunterliegende JMS Implementierung auf einer zentralisierten oder dezentralisierten Architektur beruht, ist nicht Teil der Spezifikation. Tatsächlich existieren daher verschiedene Varianten von JMS Implementierungen: SonicMQ setzt beispielsweise eine zentralisierte Architektur ein. Fiorano bietet unter anderem zwei JMS Implementierungen *Fiorano InfoBus* und *FioranoMQ Message Server* an. Beide Produkte haben einen unterschiedlichen Leistungsumfang. Das auf IP Multicast basierende InfoBus verwendet keinen zentralen Server und bietet z. B. keine Persistenzmechanismen. FioranoMQ hingegen setzt eine zentralisierte Architektur ein und bietet Persistenz.

Die Kommunikation zwischen Anwendung und JMS Provider findet abhängig vom gewählten Provider über RMI oder lokale Methodenaufrufe statt. Setzt der JMS Provider mehrere Server ein, kommt zwischen den Servern ein proprietäres Protokoll zum Einsatz. Eine Interoperabilität zwischen JMS Providern ist nicht Teil der JMS Spezifikation. Viele JMS Provider bieten Adapter zu anderen JMS Providern und MOM Systemen wie z. B. MSMQ an. Insbesondere existieren Bridges zu MQSeries, da MQSeries ein weit verbreitetes Messaging System ist.

3.4. CORBA

Die *Common Object Request Broker Architecture* (CORBA) ist eine von der *Object Management Group* (OMG) entwickelte Referenzarchitektur, die die Entwicklung portabler, verteilter objektorientierter Anwendungen in heterogenen Umgebungen fördern soll. Im Rahmen der Referenzarchitektur *Object Management Architecture* (OMA) wird ein Objekt- und ein Referenzmodell definiert, das eine orts- und plattformunabhängige sowie von Programmiersprachen unabhängige Kommunikation zwischen Applikationen erlaubt.

Neben dem ORB Kern definiert die OMG anwendungsunabhängige Dienste, die sog. CORBAservices, die die Erstellung komplexer Anwendungen erleichtern sollen. Dazu gehören ein Namingservice, ein Securityservice, ein Transaktionsservice und ein Notificationservice, der asynchrone nachrichtenbasierte Kommunikation erlaubt.

Die Schnittstellenspezifikation der CORBA Services ist in *Interface Definition Language* (IDL), einer rein deklarativen Sprache definiert. Die Implementierung eines CORBA Notification Service kann damit in verschiedenen Programmiersprachen erfolgen.

3.4.1. CORBA Event Service

Ereignisquelle und Beobachter werden in der CORBA Spezifikation *Supplier* und *Consumer* genannt. Supplier erzeugen Nachrichten, die von einem Event Channel an mehrere Consumer ausgeliefert werden. Ein Supplier kommuniziert jeweils mit einem *ProxyConsumer*. Korrespondierend wird ein Consumer von einem *ProxySupplier* beliefert. ProxyConsumer und ProxySupplier werden von einen *ConsumerAdmin* bzw. *SupplierAdmin* erzeugt. Diese hierarchische Anordnung erleichtert die Administration des Systems (vgl. Abbildung 3.2).

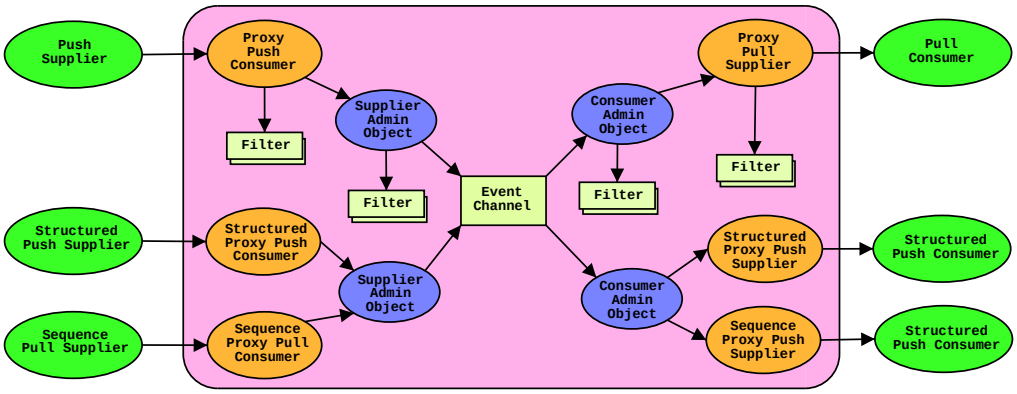

Abbildung 3.2.: Architektur des CORBA Notification Service

Der Event Service unterstützt alle Kombinationen der *Push*- und *Pull*-Kommunikation. Als Nachricht kommt der CORBA Datentyp *Any* zum Einsatz. In einem Any können beliebig komplexe Datentypen enthalten sein. Der CORBA Event Service ist eine Ausprägung von Channel-based Subscription. Der Event Channel hat demnach keine Kenntnis über Struktur der übertragenen Nachrichten. Insbesondere besteht keine Möglichkeit für den Empfänger, eine Auswahl der Nachrichten zu treffen.

Es existieren zahlreiche Implementierungen des CORBA Event Service. Mico, ORBit, TAO, JacORB und Prismtechnologies liefern Implementierungen.

Die Kommunikation zwischen Anwendung und dem Event Service wird über einen ORB abgewickelt. Eine direkte Kommunikation zwischen mehreren Instanzen des Event Service ist nicht vorgesehen.

Die interne Architektur hängt von den einzelnen Implementierungen ab. Die Nützlichkeit des CORBA Event Service ist durch eine mangelnde Spezifikation eingeschränkt. Insbesondere die fehlenden Filter- und Dienstgütemöglichkeiten schränken die Nützlichkeit des Dienstes ein.

3.4.2. CORBA Notification Service

Die Nachteile von CORBA Event Service motivierten die Entwicklung einer neuen Spezifikation, die die Schwachstellen der früheren Spezifikation beheben sollte.

Die CORBA Notification Service Spezifikation [23] erweitert die Spezifikation des CORBA Event Service. Insbesondere sind die neuen IDL Schnittstellen von den alten IDL Schnittstellen abgeleitet, wodurch das System abwärtskompatibel bleibt. Applikationen, die den Event Service nutzen, können unverändert den Notification Service verwenden.

Zusätzlich zum einfachen ungetypten Event Channel, der den Datentyp *Any* unterstützt, bietet der Notification Service drei weitere Schnittstellen: den *Structured Event Channel*, den *Sequenced Event Channel* und den *Typed Event Channel*.

Der Structured Event Channel und der Sequenced Event Channel verwenden vordefinierte Nachrichten vom Typ *StructuredEvent* (vgl. Abbildung 3.3).

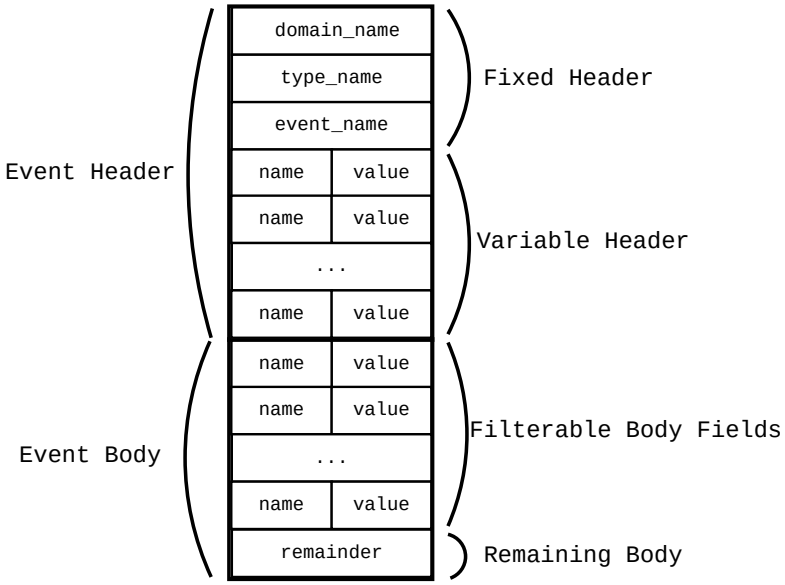

Abbildung 3.3.: Format eines StructuredEvent

Ein StructuredEvent besteht ähnlich einer JMS Nachricht aus Header und Body. Der Header setzt sich aus einem festgelegtem und einem variablen Anteil zusammen und enthält Angaben, die die Nachricht beschreiben sollen. Der festgelegte Anteil enthält drei Werte. Eine Nachricht minimaler Größe enthält lediglich den festgelegten Header. Damit sind sehr kleine Nachrichten möglich. Im variablen Teil des Header und im filterbaren Teil des Body können weitere Attribute eingefügt sein, die die Nachricht beschreiben und zur Filterung verwendet werden können. Die Nutzlast der Nachricht ist im Feld *remainder_of_body* enthalten. Prinzipiell ist es auch möglich, Filterausdrücke zu formulieren, die diesen Teil der Nachricht evaluieren. Die komplette Beschreibung dieses Datentyps findet sich in der Spezifikation des Notification Service [23, Kap. 2.2].

Der Sequenced Event Channel verwendet als Nachrichtentyp Sequenzen vom Typ *StructuredEvent*. Die vierte und letzte Variante, der Typed Event Channel, unterscheidet sich von den vorangegangenen Varianten. Ein Typed Event Channel verwendet keine explizit definierten Datentypen. Daher existieren auch

keine in IDL vordefinierten Operationen wie *push_structured_event* oder *push*. Stattdessen kommunizieren die Nutzer des Typed Event Channel über nutzerdefinierte Schnittstellen. In der Spezifikation des Notification Service werden diese Schnittstellen mit <*I*> bezeichnet. Operationen in einem <*I*>-Interface dürfen keine Rückgabewerte haben. Es sind also nur *void*-Operationen ohne *out* oder *inout* Parameter zulässig. Um den Typed Event Channel zu verwenden, erfragt ein Nutzer beim Event Channel eine Referenz, die dem Interface <*I*> genügt, er kann dann auf dem referenzierten Objekt Methoden aufrufen, um Nachrichten zu versenden. Der Aufruf wird auf eine Nachricht vom Typ *TypedEvent*, bestehend aus Operationsname und Parametern, abgebildet, die vom Event Channel bearbeitet wird. Ein Typed Consumer muss ebenfalls die Schnittstelle <*I*> implementieren. Auf diese Weise kann der Event Channel eine Nachricht auf einen Methodenaufruf beim Consumer abbilden.

Dieser Ansatz erlaubt es den Nutzern des Typed Event Channel, mittels stark getypter, nutzerdefinierter Schnittstellen zu kommunizieren, ohne dass die Notwendigkeit besteht, Applikationsdaten in vorgegebene Datentypen zu konvertieren. Im Gegensatz dazu muss bei Verwendung des Structured, Sequenced oder Untyped Event Channel eine generische low-level Schnittstelle verwendet werden, die es erfordert, die Daten vor Versand in ein spezielles Format zu konvertieren und nach Empfang wieder zu extrahieren. Ungetypte Event Channels bieten daher eine semantisch schwächere, weniger typsichere Schnittstelle, deren Verwendung eine Verletzung des objektorientierten Programmierparadigmas darstellt.

An verschiedenen Stellen ist es möglich, Filter einzubinden. Die Filter werden mit einer Filtersprache namens *Extended Trader Constraint Language* (ETCL) konfiguriert. ETCL ist eine Erweiterung der *Trader Constraint Language* (TCL), die beim CORBA Trader Service zum Einsatz kommt. Darüber hinaus sieht die Spezifikation vor, Filter einzusetzen, die mit anderen Sprachen oder Mechanismen konfiguriert werden. Die Implementierung von Prismtechnologies bietet beispielsweise die Möglichkeit, SQL-92 basierte Filter einzusetzen.

Ein ETCL Filterausdruck ist ein Prädikat auf einer Nachricht. Durch Evaluierung kann festgestellt werden, welchen Wahrheitswert der Filter hat. Abhängig von diesem Ergebnis kann die Nachricht weitergeleitet oder verworfen werden. Der Notification Service sieht auch sog. *MappingFilter* vor, die bei erfolgreicher Auswertung Attribute einzelner Nachrichten ändern können.

Bezeichner in Filterausdrücken können sich auf die gesamte Nachricht beziehen. Dabei bezeichnet `$` die aktuelle Nachricht:

```
$.header.fixed_header.event_name == 'Alarm'
```

Die Filtersprache unterstützt die Vergleichsoperatoren `>`, `>=`, `==`, `<=`, `<` und `!=`. Arithmetische Operationen können mit `+`, `-`, `*`, `/` durchgeführt werden. Der Operator `~` ermöglicht es zu prüfen, ob ein String in einem anderen String enthalten ist. Mit Hilfe von `and`, `or` und `not` können komplexere Ausdrücke gebildet werden. Mit `exist` lässt sich die Existenz einzelner Attribute prüfen, `in` prüft, ob ein Element sich in einer Sequenz befindet, und mit `type` kann schließlich die Repository ID eines Wertes abgefragt werden.

Die restlichen Neuerungen des Notification Service gegenüber dem Event Service betreffen die Möglichkeiten, Dienstgüte auf unterschiedlichen Ebenen zu konfigurieren. Damit ist es möglich, beipielsweise zwischen *best-effort* und *at-least-once* Zuverlässigkeit zu wählen und Timeoutwerte für einzelne Nachrichten zu konfigurieren.

Grundsätzlich realisiert der CORBA Notification Service eine zentralisierte Architektur. Es ist zwar möglich, mehrere EventChannel-Instanzen miteinander zu verbinden. Dabei ist jedoch kein Server-Server Protokoll definiert, das Optimierungen des Nachrichtenaustausches wie bei SIENA zulässt. Die einzelnen EventChannel-Instanzen kommunizieren lediglich anhand des Client-Server Protokolls. In der Programmierschnittstelle sind jedoch Operationen vorhanden, die es erlauben, eine entsprechende Funktionalität auf Nutzerebene zu realisieren.

3.5. Zusammenfassung

Bestehende Benachrichtigungssysteme können anhand ihrer Serverarchitektur miteinander verglichen werden. Die Architektur hat einen direkten Einfluss auf die Skalierbarkeit des Systems [11]. In Tabelle 3.1 sind die bisher genannten Technologien in Beziehung zueinander gesetzt. Es existiert kein optimaler Be-

		Architektur		
		zentralisiert	hierarchisch	peer-to-peer
Subskriptionsstil	Channel-based	CORBA Event Service	CORBA Event Service	IP Multicast
	Subject-based	ToolTalk	NNTP	—
	Content-based	CORBA Notification Service, JMS, Elvin	SIENA, MQSeries, Keryx	SIENA, JMS, Gryphon [60]

Tabelle 3.1.: Klassifikation der Technologien

nachrichtigungsdienst. Abhängig vom Einsatzgebiet und aus den sich daraus ergebenden Anforderungen muss eine Auswahl für eine Architektur getroffen werden. Viele der betrachteten Systeme sind zwar konfigurierbar und können daher in gewissem Rahmen an unterschiedliche Anforderungen angepasst werden, aber letztendlich existieren architekturbedingte Einschränkungen, die den Einsatz erschweren können.

Systeme wie MQSeries haben eine breite Nutzerbasis. Modernere Systeme wie JMS benötigen daher meist Adapter, um in bestehende Systemumgebungen integriert zu werden. Die Tatsache, dass die meisten Hersteller von JMS-

Systemen solche Adapter anbieten, spiegelt den Bedarf der Anwender, die offenbar heterogene Systemumgebungen betreiben.

Ab Version 1.3 des J2EE Standards ist auch das JMS API integriert. JMS ist damit das Standard API, um von Java Applikationen auf Benachrichtigungssysteme zuzugreifen. In heterogenen Umgebungen wird der CORBA Notification Service verwendet. Besonders interessant ist die Integration von J2EE, JMS und dem CORBA Notification Service. Bisher gibt es zwar herstellerspezifische Lösungen [3], es existieren jedoch Bestrebungen, die Interoperabilität zu standardisieren [24].

Schließlich bietet SIENA ein konzeptionell überlegenes System, wenn die Skalierbarkeit eine besondere Rolle spielt. Die Einschränkungen auf Seite der Filtermöglichkeiten werden mit einer ausgezeichneten Skalierbarkeit ausgeglichen. Es werden Testszenarien mit bis zu 10000 Clients beschrieben [9]. Es bleibt zu beobachten, inwieweit entsprechende Konzepte von der Industrie adaptiert werden.

4. Entwurf des JacORB Notification Service

In diesem Kapitel wird der Entwurf des JacORB Notification Service beschrieben. Die Funktionalität des Notification Service ist in dem Dokument *Notification Service Specification* festgelegt, das auf dem Webserver der OMG erhältlich ist [23]. Das Dokument enthält eine allgemeine Beschreibung des Dienstes, die in IDL beschriebenen Nutzerschnittstellen und umfangreiche, natürlichsprachliche Spezifikationen der Nutzerschnittstellen.

Die Spezifikation beschreibt eine funktionale Sicht auf den Dienst. Es gibt keine besonderen Vorgaben bezüglich der Architektur einer Implementierung. Um eine performante und skalierbare Implementierung zu erzielen, ist daher eine guter Entwurf notwendig. Die Spezifikation des Dienstes ist sehr umfangreich, was es schwierig macht, die gesamte Funktionalität im Rahmen dieser Arbeit zu realisieren. Auch wenn in dieser Implementierung nicht die gesamte Funktionalität realisiert wird, sollte der Entwurf erweiterbar sein, um fehlende Funktionalität in späteren Versionen nachliefern zu können.

Ein guter Ausgangspunkt ist die Orientierung an bestehenden Systemen. Es existieren einige Implementierungen des Notification Service. Eine Internetrecherche ergab folgende Ergebnisse:

- Prismtechnologies OpenFusion
- TAO
- ORBacus
- omniorb
- DSTC dCon
- e*orb
- The Community OpenORB

OpenFusion, ORBacus, dCon und e*orb sind kommerzielle Produkte. OpenORB und TAO stehen unter einer Open Source Lizenz zur Verfügung. Es sind jedoch nur wenige Informationen über die interne Architektur der Implementierungen erhältlich. Die Informationen beschränken sich auf Nutzerhandbücher und White Papers. Lediglich aus dem TAO und dem omniorb Projekt sind zwei Dokumente erhältlich, die die Architektur darstellen [28, 20].

Ein detaillierter Überblick über Funktionsumfang verschiedener ORBs findet sich auf der Website von Arno Puder.

4.1. Problembereiche und Lösungsmuster

Die Skalierbarkeit einer Implementierung des CORBA Notification Service ist von unterschiedlichen Faktoren abhängig. Als wichtigste Faktoren sind zu nennen [28]:

- Die zugrundeliegende Hardware,
- das Betriebssystem,
- die Leistungsfähigkeit des darunterliegenden CORBA ORB,
- die Performanz der Filterevaluierung,
- die Effizienz beim Filtern und beim Ausliefern der Nachrichten.

Auf die Auswirkungen der Auswahl von Betriebssystem und Hardware wird hier nicht weiter eingegangen.

In den anderen genannten Bereichen werden im Folgenden einige Problembereiche identifiziert und mögliche Lösungen aufgezeigt, um eine performante, skalierbare Implementierung zu erreichen.

Die Filterevaluierung ist eine rechenzeitintensive Aufgabe. Das Versenden der Nachrichten an die einzelnen Empfänger ist eine I/O intensive Aufgabe. Wenn diese Aufgaben nebenläufig ausgeführt werden können, wird die Parallelität der Applikation erhöht. Daher ist ein nebenläufiges Design sinnvoll.

4.1.1. Filterevaluierung

In einem EventChannel können Proxy- und Adminobjekten Filter zugeordnet sein (Abb. 3.2 auf Seite 45). Es existieren zwei Varianten, um diese effizient nebenläufig zu evaluieren [28]:

1. die parallele Bearbeitung der an den Event Channel gesendeten Nachrichten, und
2. die parallele Bearbeitung der einzelnen Filter.

Bei paralleler Bearbeitung der Nachrichten evaluiert ein einzelner Thread alle Filter, die den Proxy- und Admin-Objekten zugeordnet sind. Da stets nur ein Thread eine Nachricht bearbeitet, sind keine Synchronisationsmechanismen beim Zugriff auf die Nachricht notwendig.[1] Andererseits kann ein Filter von mehreren Threads nebenläufig verwendet werden, um unterschiedliche Nachrichten zu evaluieren. Daher muss in diesem Fall die Verwendung des Filters reentrant realisiert sein. Sind Synchronisationsmechanismen erforderlich, um die Konsistenz eines Filters zu gewährleisten, kann dies bei hohen Nachrichtenraten einen Engpass darstellen.

In Abhängigkeit von der Komplexität der einzelnen Filterausdrücke kann die Auswertungsdauer aller Filter für eine Nachricht stark differieren. Dadurch

[1] In den meisten Fällen wird beim Filtern lediglich lesend auf eine Nachricht zugegriffen. Mappingfilter greifen jedoch auch schreibend auf Nachrichten zu.

kann eine Nachricht unter Umständen schneller ausgeliefert werden als eine andere Nachricht, die früher abgesendet wurde, deren Filterauswertung jedoch länger dauert. Ist die Reihenfolge der Auslieferung relevant,[2] müssen in diesem Fall weitere Schritte für die Sortierung der Nachrichten getroffen werden.

Bei paralleler Auswertung der Filter werden für jede Nachricht diskunkte Gruppen von Filtern von je einem Thread evaluiert. Die Gruppierung der Filter kann nach unterschiedlichen Kriterien erfolgen, z. B. alle Filter, die einem ProxyObjekt zugeordnet sind. Da jeder Filter von nur einem Thread verwendet wird, sind keine Synchronisationsmechanismen innerhalb der Filter notwendig. Vorteilhaft ist außerdem, dass die Reihenfolge der Nachrichten nicht verändert werden kann. Da eine Nachricht von mehreren Threads zur gleichen Zeit bearbeitet werden kann, sind Synchronisationsmechanismen beim Zugriff auf die Nachricht notwendig. Darüber hinaus kann sich, bei stark unterschiedlichen Filterkomplexitäten, auch der Aufwand für die Evaluierung der einzelnen Filtergruppen erheblich unterscheiden. Dies stellt eine schlechte Verteilung der Aufgaben auf die Threads dar.

4.1.2. Auslieferung der Nachrichten

Nachdem alle Filter eines Consumers evaluiert wurden, ist entschieden, ob die Nachricht zugestellt wird. Die Übermittlung einer einzelnen Nachricht vom ProxySupplier zum Consumer erfolgt mit dem CORBA two-way Protokoll über TCP/IP. Der Event Channel ist daher beim Versand von Daten über das Netzwerk den größten Teil der Zeit mit Warten beschäftigt. Die Parallelisierung an diesem Punkt ist daher am erfolgversprechendsten. Während auf die Vollendung der Zustellung einer Nachricht an einen Consumer gewartet wird, kann eine andere Nachricht bereits an einen anderen Consumer gesendet werden.

Es existieren mehrere Strategien, um eine Nachricht zuzustellen:

1. single-thread,
2. thread-per-consumer, und
3. thread-pool.

Die Verwendung eines einzelnen Threads kommt nur für Systeme mit stark beschränkten Ressourcen oder wenigen verbundenen Empfängern in Frage, da ansonsten durch die Serialisierung der Auslieferungen hohe Latenzzeiten entstehen können.

Die Verwendung eines Threads pro Consumer ist einfach zu realisieren und bietet eine einfache Möglichkeit, die Zustellung zu parallelisieren. Dieser Ansatz skaliert jedoch nicht mit einer wachsenden Anzahl von Empfängern. Wenn zahlreiche Consumer mit dem Event Channel verbunden sind, werden viele Ressourcen des zugrundeliegenden Betriebssystems in Anspruch genommen. Außerdem ist dieser Ansatz sehr ineffizient, wenn Clients sich nur kurz mit dem Event Channel verbinden. Der Ansatz bietet sich daher bei Systemen an, die mit einer geringen bis mittleren Anzahl von Consumern verbunden sind.

[2] Dies hängt von den Dienstgüteeinstellungen ab.

Bei Verwendung eines Thread Pool wird eine feste Anzahl von Threads verwendet, um Nachrichten den Empfängern zuzustellen. Dieser Ansatz ist ressourcenschonender als das thread-per-connection Modell, benötigt jedoch zusätzlichen Synchronisationsaufwand.

4.1.3. Uniforme Behandlung unterschiedlicher Event-, Supplier- und Consumer-Typen

Innerhalb des CORBA Notification Service kommen Nachrichten in unterschiedlichen Formaten zum Einsatz. Es existieren *Any*, *StructuredEvent*, *Sequenzen von StructuredEvent* und *TypedEvent*. Supplier können Nachrichten in diesen Formaten an den Event Channel übergeben. Entsprechend existieren Consumer für die verschiedenen Typen. In der Spezifikation des Notification Service ist beschrieben, wie die Nachrichtentypen ineinander umgewandelt werden können. Damit kann beispielsweise ein Supplier Nachrichten vom Typ *Any* versenden und ein Consumer diese als StructuredEvent empfangen.

Um nicht für jeden Nachrichtentyp spezifischen Code schreiben zu müssen, sollte in der Implementierung vom konkreten Typ der Nachricht abstrahiert werden.

Eine Möglichkeit dazu wäre, die Nachrichten stets in ein kanonisches Format zu konvertieren. Wenn Sender und Empfänger jedoch die gleichen Nachrichtentypen unterstützen, würden in diesem Fall unnötige, kostspielige Konvertierungen stattfinden. Ebenso muss darauf geachtet werden, Konvertierungen nicht mehrfach durchzuführen.

Ein ähnliches Problem besteht bei den unterschiedlichen Proxy- und Admintypen. Proxy- und Adminobjekte bieten gleiche Teilfunktionalitäten, beispielsweise für die Verwaltung der Filter. Nur wenige Methoden sind spezifisch. Es bietet sich daher für den Entwurf an, interne Schnittstellen für Teilsysteme mit gleicher Funktionalität zu definieren.

4.1.4. Minimierung der Abhängigkeiten der Aufgaben

Ein Event Channel wird im Allgemeinen von einer Menge parallel agierender Supplier und Consumer verwendet. Für einen Supplier soll der Zeitaufwand, um eine Nachricht an den Event Channel zu übergeben, minimal sein. Insbesondere soll der Supplier nicht blockiert werden, bis alle Consumer beliefert wurden.

Die Auslieferung der Nachrichten an einzelne Consumer kann sehr lange dauern. Da die Auslieferung mittels CORBA two-way Kommunikation erfolgt, ist der mit der Auslieferung beauftragte Thread so lange blockiert, bis die Nachricht zugestellt ist.

Durch Einsatz eines Threadpool kann die Gesamtzahl der Threads im System beschränkt sein. Daher können alle Threads beschäftigt sein, wenn ein Supplier mittels *push* eine neue Nachricht an den Event Channel übergibt. In einer einfachen Realisierung müsste der Supplier daher so lange blockiert werden, bis

ein Thread die Bearbeitung der Nachricht übernehmen kann. Dies widerspricht der Forderung, den Supplier nur minimale Zeit zu blockieren.

Um den Aufruf einer Operation von ihrer Ausführung zu entkoppeln, kann das Entwurfsmuster *Befehl* zum Einsatz kommen [19]. Dabei wird ein Befehl als Objekt gekapselt und in eine Warteschlange gestellt. Die Befehle werden zu einem späteren Zeitpunkt aus der Warteschlange entnommen und von einem aktiven Thread ausgeführt. Besonders wichtig ist, dass die einzelnen Befehle unabhängig ausführbar sein müssen. Ansonsten kann es zu Verklemmungssituationen kommen.

Durch Einsatz dieses Musters werden die einzelnen Aufgaben unabhängiger voneinander. Auch wenn alle Threads ausgelastet sind, kann der Event Channel Nachrichten, ohne zu blockieren, entgegennehmen. Die Aufgaben können flexibel auf die Threads verteilt werden

4.1.5. Fairness beim Abarbeiten der Teilaufgaben

Es ist möglich, einzelne Nachrichten unterschiedlich zu priorisieren. Ein Notification Service bearbeitet die Nachrichten gemäß ihrer Prioritäten.

Falls die Bearbeitung einer höher priorisierten Nachricht relativ lange dauert und stets neue, hoch priorisierte Nachrichten eintreffen, können niedriger priorisierte Nachrichten unter Umständen vernachlässigt werden.

Um diesem Problem zu begegnen, kann die Ausführung der Aufgaben in mehrere Teilschritte zerlegt werden [20]:

1. Evaluierung der ProxyConsumer Filter.
2. Evaluierung der SupplierAdmin Filter.
3. Evaluierung der ConsumerAdmin Filter.
4. Evaluierung der ProxySupplier Filter.
5. Auslieferung der Nachrichten.

Nach jedem Schritt kann eine neue Teilaufgabe zur Ausführung ausgewählt werden. Wird dieser Ansatz durch die Verwendung eines geeigneten Scheduling-Algorithmus ergänzt, kann eine Fairness zwischen den einzelnen Aufgaben gewährleistet werden.

4.1.6. Optimierung der Any-Performanz

Ein CORBA-konformer Notification Service muss in der Lage sein, Nachrichten zu verarbeiten, die Daten vom Typ *Any* enthalten. *Any* ist ein Datentyp, dessen Verwendung teuer sein kann, da *Any* Datenstrukturen beliebig tief geschachtelt sein können. Abhängig von der Implementierung des verwendeten ORB werden beim Kopieren oder beim Extrahieren eines Wertes aus einem *Any* interne Puffer kopiert bzw. neu angelegt.

Um damit verbundene Performanzkosten zu vermeiden, ist es möglich, die Any-Implementierung des zugrundeliegenden ORB zu optimieren [20]. Eine andere, vom ORB unabhängige Möglichkeit ist, die aus einem *Any* extrahierten Werte zu cachen. Wird erneut auf den gleichen Wert zugegriffen, kann die kostspielige Extraktion umgangen werden. Der positive Effekt dieser Optimierung kommt jedoch lediglich bei Filterausdrücken, die mehrfach den gleichen Wert referenzieren, zum Tragen.

4.1.7. Optimierung des statischen Ressourcenbedarfs

Die Spezifikation des CORBA Notification Service definiert ein flexibel an viele Anwendungsgebiete anpassbares System. Dementsprechend ist die Spezifikation sehr umfangreich. Es existieren Einsatzgebiete, die nur einen Teil der verfügbaren Funktionalität benötigen. Manche der dort eingesetzten Plattformen, z. B. Embedded Systems, verfügen nur über eingeschränkte Ressourcen. Für solche Systeme kann es notwendig sein, eine Applikation so konfigurieren zu können, dass sie einen geringeren statischen Platzbedarf hat. Insbesondere sollen das ausführbare Programm bzw. die Bibliotheken nur die wirklich verwendete Funktionalität enthalten.

Um dieses Ziel zu erreichen, müssen die einzelnen Komponenten des Event Channel lose miteinander gekoppelt sein. Das macht die Herauslösung einzelner, nicht benötigter Komponenten möglich. So könnte es beispielsweise möglich sein, eine Konfiguration des Notification Service zu erstellen, die keine Filterfunktionalität anbietet.

Die Architektur des JacORB Notification Service ist modular gestaltet. Es wird jedoch kein besonderes Augenmerk auf diese alternative Anforderung gerichtet.

4.1.8. Optimierung des dynamischen Ressourcenbedarfs

Eine Optimierung des dynamischen Ressourcenbedarfs ist notwendig, um eine skalierbare Performanz der Implementierung zu erreichen. Ist beispielsweise der Ressourcenbedarf bei Einsatz eines thread-per-connection Modell (vgl. Abschnitt 4.1.2) bei einer geringen Anzahl von aktiven Clients tolerierbar, werden bei wachsender Anzahl von Verbindungen die Ressourcen des Betriebssystems übermäßig in Anspruch genommen. Die Kosten für die Verwaltung der zahlreichen Threads lassen die Gesamtperformanz der Implementierung sinken.

Das gleiche gilt für das Anlegen und Zerstören temporärer Objekte. Werden beim Filtern und Zustellen von Nachrichten viele Objekte angelegt, kurz verwendet und dann wieder freigegeben, können bei hohen Benachrichtigungsfrequenzen Speicherengpässe auftreten. Es darf außerdem nicht vernachlässigt werden, dass die Zerstörung eines Objekts durch den Java Garbage Collector ebenfalls Rechenzeit in Anspruch nimmt. Daher sollen Objekte, die beim Bearbeiten einer Nachricht erzeugt werden, möglichst wiederverwendet werden. Die Verwendung des Entwurfsmusters *Factory* [19] kann dabei helfen. Insbesondere in den Codeabschnitten, die bei der Bearbeitung einer Nachricht durchlaufen

werden, sollten keine Objekte explizit durch den Aufruf eines Konstruktors erzeugt werden. Stattdessen wird die Erzeugung an ein Factoryobjekt delegiert. Das Factoryobjekt kann geeignete Mechanismen realisieren, um nicht mehr verwendete Objekte wiederzuverwenden.

4.1.9. Einsatz von Timern

Manche Aufgaben innerhalb des Notification Service werden nur zu bestimmten Zeitpunkten oder in bestimmten Intervallen ausgeführt. So kann beispielsweise ein PullSupplier seine Nachrichten regelmäßig von einem ProxyPullConsumer pollen lassen (vgl. Pull-Modell in Abschnitt 2.3.1). Korrespondierend kann sich ein SequencePushConsumer, ebenfalls regelmäßig, von einem SequenceProxyPushSupplier mit allen aufgelaufenen Nachrichten beliefern lassen.

Auch bei der Bearbeitung der Nachrichten spielt die Zeit eine Rolle. So lassen sich Timeouts definieren, nach denen nicht zugestellte Nachrichten verworfen werden. Für andere Nachrichten kann man eine Aktivierungszeit definieren, nach der eine Nachricht bearbeitet wird.

Prinzipiell handelt es sich hier um eine Optimierung des dynamischen Ressourcenbedarfs. Eine geeignete Lösung ist der Einsatz eines Schedulingmechanismus, der es erlaubt, eine Aufgabe mit einem (wiederkehrenden) Zeitpunkt zu verknüpfen.

Eine einfache Variante, einen Timer zu realisieren, ist, einen Thread zu starten, der durch Aufruf der Methode *Object.wait* entsprechend lange wartet und danach die Aufgabe bearbeitet. Dieser Ansatz ist im Prinzip dem thread-per-connection Ansatz (vgl. 4.1.2) vergleichbar. Damit gelten die gleichen Nachteile: Werden für zahlreiche Aktionen Timer verwendet, müssen viele Threads aktiviert werden, die schlafend zwar keine Rechenzeit kosten, aber dennoch einen Verwaltungsaufwand für das Betriebssystem darstellen.

Ein effizienter Scheduler wird hingegen nur einen aktiven Thread verwenden, die Aufgaben gemäß ihres Ausführungszeitpunkts ordnen und zum richtigen Zeitpunkt ausführen.

4.1.10. Effiziente Auswahl der Filter

Für jede Nachricht, die an den Event Channel gesendet wird, muss eine Menge von Filtern ausgewertet werden. Da diese Menge unter Umständen nur ein sehr geringer Teil aller Filter ist, ist eine effiziente Auswahl der Filter notwendig [28]. Üblicherweise enthalten die Constraints in einem Filter eine Liste von Domain- und Typnamen. Anhand dieser Liste kann entschieden werden, ob ein Filter zu einer konkreten Nachricht passt. Sowohl im Domain- als auch im Typnamen kann das Platzhalterzeichen * verwendet werden, um für Null oder beliebig viele Zeichen zu stehen. Der Platzhalter darf an jeder Stelle stehen. Erwartungsgemäß bezeichnet {‘*’,‘*’} einen Filter, der auf jeden Nachrichtentyp passt.

Um effizient Werte einzelnen Schlüsseln zuzuordnen, wird üblicherweise Hashing verwendet, da mit Hashing die sog. Dictionary Operationen in der Laufzeit

$O(1)$ realisierbar sind. Durch die Existenz des Platzhalters in Namen verbietet sich jedoch leider der Einsatz der existierenden Klasse *Map* aus dem Standard Java API, da diese keine Platzhalter in den Schlüsseln unterstützen.

Ein einfach zu realisierender Ansatz ist, die Domain/Typnamen als Liste zu verwalten. Eine Nachricht muss dann jeweils mit jedem Element der gesamten Filterliste verglichen werden. Die Laufzeit ist damit linear zur Anzahl der Filter.

Diese Datenstruktur kann intuitiv als Hashtable, mit der Möglichkeit, reguläre Ausdrücke als Schlüssel zu verwenden, aufgefasst werden. Nachfolgend wird die Signatur dieser Datenstruktur beschrieben:

put einem Schlüssel wird ein Wert zugeordnet werden. Der Schlüssel ist vom Typ String und kann an beliebigen Positionen den Platzhalter * enthalten. Bei dieser Operation wird der Platzhalter nicht besonders berücksichtigt.

remove ein Schlüssel/Wert-Paar wird entfernt. Auch bei dieser Operation wird der Platzhalter nicht besonders berücksichtigt.

lookup zu einem gegebenem Namen werden alle passenden Werte bestimmt werden. Entweder der Schlüssel des Wertes enthält keinen Platzhalter und passt exakt zum Namen, oder der Schlüssel enthält Platzhalter und diese lassen sich zum Namen expandieren. Diese Operation kann daher mehrere Werte zurückliefern.

4.2. Architektur des JacORB Notification Service

In diesem Abschnitt wird der Entwurf des JacORB Notification Service beschrieben und an Beispielen verdeutlicht. Folgende Entwurfsziele stehen für den Entwurf im Mittelpunkt:

- Vollständige Realisierung der Filterfunktionalität.
- Keine MappingFilter.
- Standardkonforme Implementierung. Bei Aufruf nicht realisierter Funktionalität sollen entprechende Fehlermeldungen geliefert werden.
- Die TypedEvent Schnittstelle wird nicht unterstützt. Eine Implementierung des CORBA Notification Service gilt auch als standardkonform, wenn diese Schnittstelle nicht realisiert wird.
- Gute Laufzeiteffizienz. Die Performanz soll anderen Java Implementierungen des Notification Service mindestens vergleichbar sein.
- Keine Unterstützung von Dienstgüteeinstellungen. Es werden insbesondere keine Persistenzmechanismen realisiert.
- Erweiterbare Architektur.

Grundsätzlich ist die Entscheidung zu treffen, auf welcher Ebene eine Implementierung des Notification Service auf den darunterliegenden ORB zugreift. Eine Implementierung auf Nutzerebene verwendet lediglich die von der OMG definierten Schnittstellen, um mit dem ORB zu kommunizieren. Die andere Möglichkeit, ist den ORB auf Applikationsebene anzusprechen und damit dessen interne Schnittstellen zu verwenden.

Eine Implementierung auf Nutzerebene kann einfach auf einen anderen ORB portiert werden und ist unabhängig von möglichen Änderungen der internen Schnittstellen des ORB. Andererseits verbietet dieser Ansatz bestimmte Einsatzgebiete [8]. Ein Notification Service, der mit EJB verwendet werden soll, muss in der Lage sein, den Datentyp *ValueType* zu übertragen. ValueType ist ein CORBA IDL Datentyp, der komplexe Datenstrukturen enthalten kann. Der Datentyp wird an vielen Stellen vom RMI Protokoll verwendet, um serialisierbare und andere Java-spezifische Datenstrukturen über IIOP zu übertragen. Daher ist beim Einsatz eines Notification Service mit EJB damit zu rechnen, dass Nachrichten auch den Typ ValueType enthalten können. Wird eine Nachricht vom Supplier abgesendet (Schritt 1 in Abbildung 4.1), wird diese an den ORB des Notification Service übertragen (Schritt 2). Der ORB muss die Nachricht demarshallen und diese der Applikation (dem Event Channel) übergeben (Schritt 3). Der Event Channel übergibt die Nachricht später wieder dem ORB (Schritt 4), um sie den Empfängern zuzustellen (Schritte 5 und 6).

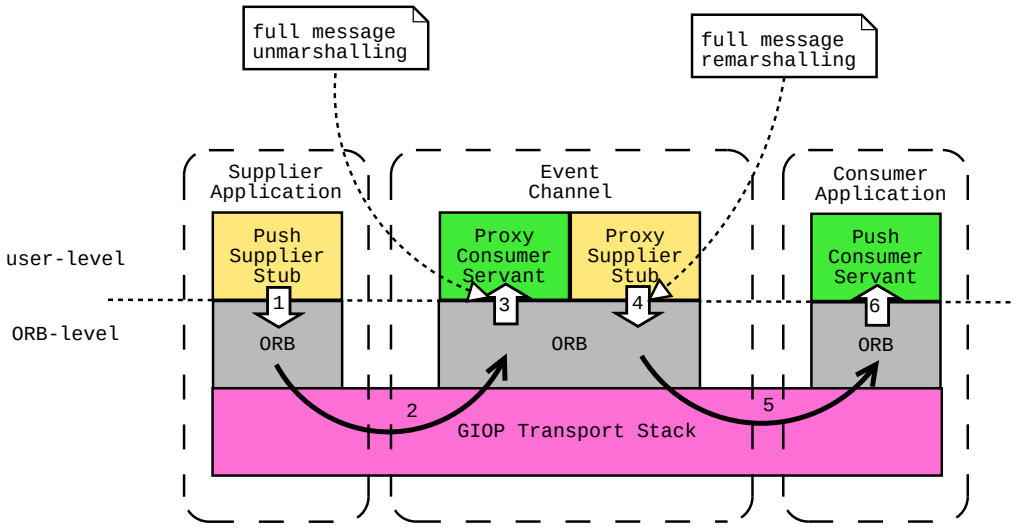

Abbildung 4.1.: Implementierung auf Nutzerebene

Wenn eine Nachricht ein ValueType enthält, benötigt der ORB des Event Channel eine *ValueFactory* für das (De-)Marshalling der Daten [vgl. 7, Kap. 5.13]. ValueFactories werden vom Anwendungsprogrammierer realisiert oder vom IDL Compiler generiert und beim ORB registriert. Daher steht dem ORB des Event Channel eine ValueFactory für applikationsspezifische ValueTypes im Allgemeinen nicht zur Verfügung. Eine Implementierung des Notification Service auf Nutzerebene kann daher nicht uneingeschränkt mit ValueTypes und

damit EJB verwendet werden.

Wenn die internen Schnittstellen eines ORB verfügbar sind, kann eine Implementierung des Notification Service enger in den ORB integriert werden (Abbildung 4.2). Bei diesem Ansatz werden nur die Teile der Nachricht demarshallt, die für die Evaluierung der Filter benötigt werden. Damit kann die oben genannte Problematik der ValueType Factories vermieden werden.

Durch diesen Ansatz kann eine allgemein höhere Performanz erreicht werden [8].

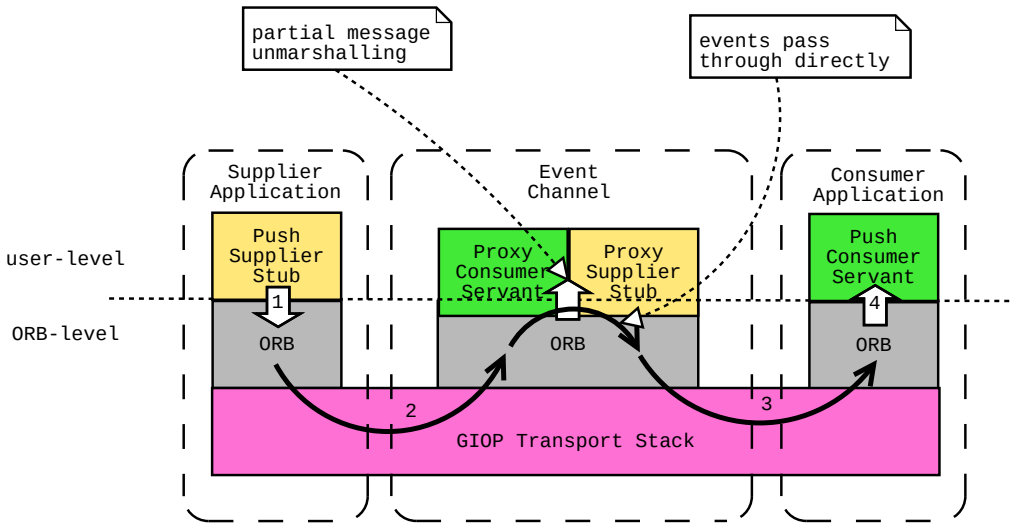

Abbildung 4.2.: Implementierung auf ORB Ebene

Die Implementierung auf ORB Ebene setzt detaillierte Kenntnis des verwendeten ORB und unter Umständen auch Änderungen an diesem voraus. Aus diesem Grund, und da die Interoperabilität mit EJB nicht im Vordergrund steht, wird die einfachere Implementierung auf Nutzerebene realisiert.

4.2.1. Verwendung eines einheitlichen Nachrichtentyps

Der Notification Service unterstützt die Nachrichtentypen *Any*, *StructuredEvent* und *Sequence of StructuredEvent*. Um Verständnis und Entwicklung der Implementierung zu erleichtern, soll nur ein Datentyp zum Einsatz kommen (vgl. Abschnitt 4.1.3). Erst beim Ausliefern der Nachrichten an die Empfänger wird in das nötige Zielformat konvertiert. Dabei ist darauf zu achten, keine unnötigen oder mehrfachen Konvertierungen durchzuführen.

Eine Lösung dieser Anforderung lässt sich durch Anwendung des Entwurfsmusters *Adapter* erreichen [19]. Das Adaptermuster erlaubt es, Klassen zusammenzuarbeiten, die wegen inkompatibler Schnittstellen dazu sonst nicht in der Lage wären.

Abbildung 4.3 auf der nächsten Seite zeigt die Anwendung des Adapterpatterns in dieser Implementierung. Die Adapterklassen *NotificationAnyEvent* und *NotificationStructuredEvent* passen die Schnittstelle der IDL-Typen *CORBA::Any*

und *CosNotification::StructuredEvent* an die Schnittstelle *NotificationEvent* an. Die Adapterklassen enthalten die notwendige Logik, um beispielsweise ein Objekt vom Typ *Any* in ein Objekt vom Typ *StructuredEvent* zu konvertieren. Innerhalb der Implementierung des JacORB Notification Service ist damit der Zugriff auf einzelne Nachrichten uniform. Erst im SupplierProxy muss in das vom Empfänger benötigte Format konvertiert werden.

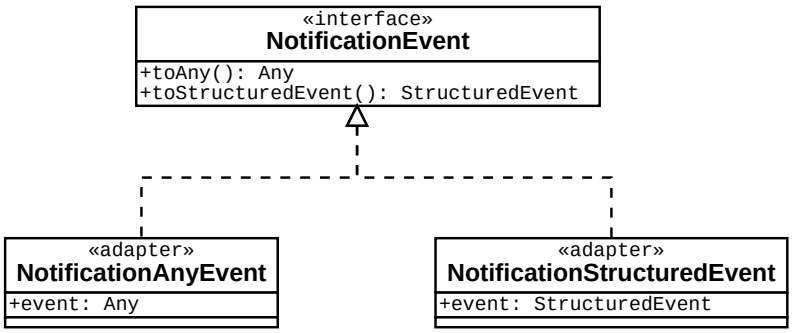

Abbildung 4.3.: Der NotificationEvent Adapter

4.2.2. Vereinheitlichung der Schnittstellen

Um gleiche Funktionalität innerhalb der Implementierung einheitlich zu bezeichnen, wird eine Menge von Java Interfaces definiert. Alle Klassen, die eine gemeinsame Funktionalität aufweisen, realisieren damit eine einheitliche Schnittstelle (vgl. Abschnitt 4.1.3).

Es existieren beispielsweise sechs verschiedene ProxySupplier Klassen (vgl. Abbildung 4.4 auf der nächsten Seite). Aus Sicht des Event Channel wird lediglich eine Nachricht an einen ProxySupplier übergeben, um diese dem verbundenen Consumer zu übermitteln. Daher realisieren die einzelnen ProxySupplier alle das Interface *EventConsumer*. Dieses Interface bietet die Methode *deliverEvent*. Die Implementierung des ProxySupplier realisiert die Logik, die Nachricht in das richtige Format zu konvertieren und dem Consumer per *push* oder *pull* zuzustellen.

Ähnlich verhält es sich mit Klassen, die Ressourcen verwalten. Diese Klassen realisieren das Interface *Disposable*, das die Methode *dispose* definiert. So können alle von einem Objekt verwendeten Ressourcen freigegeben werden.

4.2.3. Verwendung abstrakter Basisklassen

Im vorangegangenem Abschnitt wurde beschrieben, wie die Schnittstellen der Klassen vereinheitlicht werden. Darüber hinaus bietet es sich an, auch gemeinsame Funktionalität zusammenzufassen, was durch Verwendung von abstrakten Basisklassen geschieht. Ergänzend wird, wo möglich, gemeinsam verwendete Funktionalität in Hilfsklassen zusammengefasst.

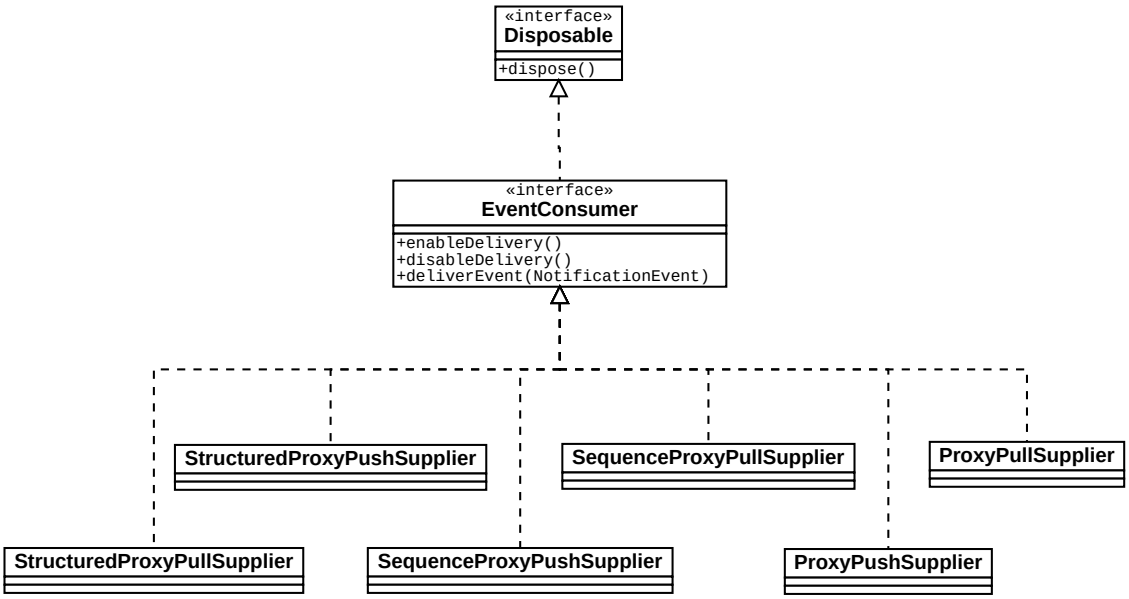

Abbildung 4.4.: Vereinheitlichung der verschiedenen ProxySupplier

Diese Vorgehensweise wird am Beispiel der Filterverwaltung demonstriert. Einem Proxy- oder Adminobjekt können ein oder mehrere Filter zugeordnet werden. Daher realisieren die entsprechenden Klassen die IDL-Schnittstelle *CosNotification::FilterAdminOperations*, die Operationen zur Verwaltung der Filter bietet.

Die Klassen *ConsumerAdminImpl* und *SupplierAdminImpl* (vgl. Abb. 4.5 auf der nächsten Seite) sind von der abstrakten Basisklasse *AdminBase* abgeleitet. Diese Basisklasse realisiert die Verwaltung der Filter. Die spezifischen Operationen sind in den abgeleiteten Klassen realisiert. Da die Filterverwaltung auch für die Proxyobjekte wichtig ist, ist die entsprechende Funktionalität in die Klasse *FilterManager* ausgelagert. Die abstrakte Basisklasse *ProxyBase* dient als Grundlage für die Implementierungen der einzelnen Proxy-Klassen. Sowohl *AdminBase* als auch *ProxyBase* delegieren die Operationen aus dem IDL-Interface *CosNotifyFilter::FilterAdminOperations* an ein assoziiertes Exemplar der Klasse *FilterManager*.

4.2.4. Ressourcenkontrolle

Java bietet ein komfortables Speichermanagement. Speicher wird beim Erstellen eines Objekts automatisch zugewiesen. Ein Garbage Collector gibt den zugewiesenen Speicher wieder frei, wenn das Objekt nicht mehr verwendet wird. Low-Level Funktionen wie *malloc* oder *free*, um auf rohen Speicher zuzugreifen, existieren in Java nicht. Daher besteht die weit verbreitete Annahme, man müsse sich nicht um Speicherverwaltung kümmern. Diese Annahme ist zugleich wahr und falsch.

Zutreffend ist, dass viele kritische Probleme, die mit der C/C++ Program-

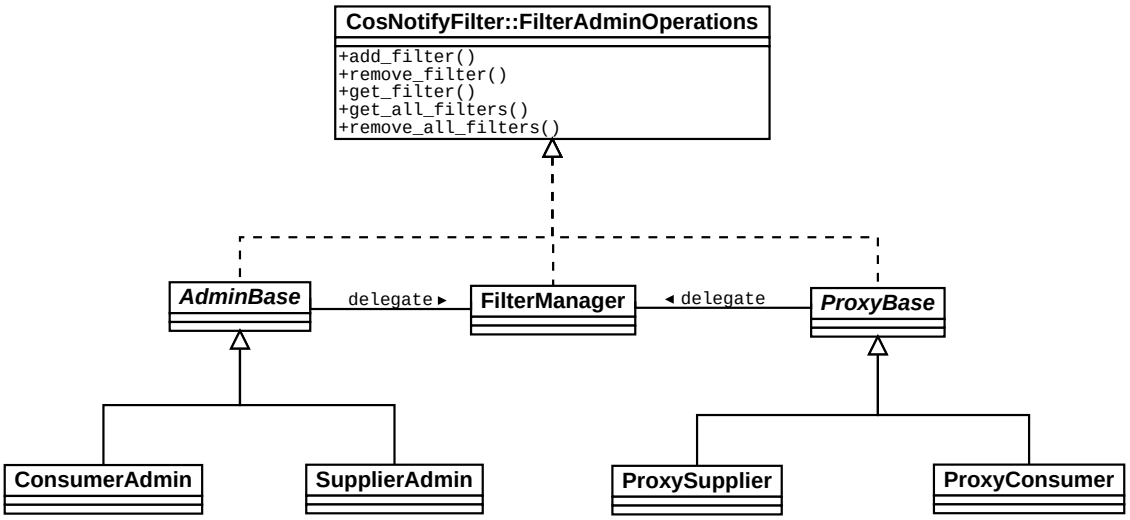

Abbildung 4.5.: Gemeinsame Nutzung von Funktionalität durch abstrakte Basisklassen

mierung verbunden sind (falsche casts, dangling pointer), vermieden werden können. Im schlimmsten Fall wird in einer Java Anwendung eine Exception ausgelöst.

Java verhindert jedoch nicht, übermäßig viel Speicher zu verwenden (beispielsweise durch Anlegen und kurzzeitige Verwendung zahlreicher temporärer Objekte). Wider Erwarten ist es daher möglich, Speicherlecks zu verursachen, indem Referenzen auf Objekte nicht freigegeben werden. Dadurch kann der Garbage Collector nicht feststellen, dass ein Objekt freigegeben werden darf.

Um in einer Implementierung Probleme zu vermeiden, die im Zusammenhang mit dem Ressourcenbedarf stehen, bieten Richtlinien einen Weg, typische Fehler zu vermeiden und eine performante Implementierung zu erzielen [56].

Nachfolgend werden zwei in der Implementierung verwendete Konzepte beschrieben, die helfen sollen, den Ressourcenbedarf des Systems kontrollieren zu können, um damit eine gute Performanz zu erreichen.

Threadmodell

Unter Beachtung der Erkenntnisse aus den Abschnitten 4.1.2, 4.1.4, 4.1.5 wird eine nebenläufige Architektur realisiert. Ein System, das einen Thread pro Nachricht verwendet, verbietet sich jedoch aus den dort angegebenen Gründen. Eine geeignete Alternative ist der Einsatz eines sog. *lightweight executable framework* [42]. Dort sind einige Threads dafür verantwortlich, mehrere unabhängige Teilaufgaben zu bearbeiten. Diese Threads werden üblicherweise als *Worker-Thread*, *Hintergrund-Thread* oder *Thread Pool* bezeichnet.

In Abbildung 4.6 auf der nächsten Seite ist das Threadmodell des JacORB Notification Service skizziert. Ein PushConsumerProxy, dessen *push* Operation aufgerufen wurde, stellt einen Bearbeitungsbefehl für eine neue Nachricht in ei-

ne Warteschlange. Danach kann der Methodenaufruf terminieren. Der Supplier, der die *push*-Operation aufgerufen hat, ist also lediglich so lange blockiert, bis die Nachricht in die Warteschlange gestellt wurde. Damit ist bereits eine wichtige Anforderung erfüllt: Der aufrufende Supplier ist nur kurz blockiert.

Im weiteren Verlauf entnehmen mehrere Worker-Threads die Befehle aus der Warteschlange und bearbeiten sie. Zuerst werden alle relevanten Filter bearbeitet und die Nachricht schließlich an die Consumer ausgeliefert.

Um das System optimal konfigurieren zu können, existieren zwei Gruppen von Worker-Threads. Die eine Gruppe ist für die Evaluierung der Filter verantwortlich, die andere Gruppe übernimmt die Auslieferung der Nachrichten. Durch eine Erhöhung der Parallelität kann eine bessere Skalierbarkeit des Systems erreicht werden. Die Gesamtanzahl der Threads und die Aufteilung auf die beiden Threadgruppen ist hierbei ein sehr wichtiger Aspekt. Die optimale Anzahl der Worker-Threads und das beste Verhältnis von Filter- und Zustellthreads kann experimentell ermittelt werden.

Die optimale Anzahl der aktiven Threads hängt von der Hardware und dem Threadmodell des Betriebssystems ab.[3] Es ist zu erwarten, dass insbesondere die Anzahl der Prozessoren einen positiven Effekt auf die Performanz und Skalierbarkeit der Applikation hat.

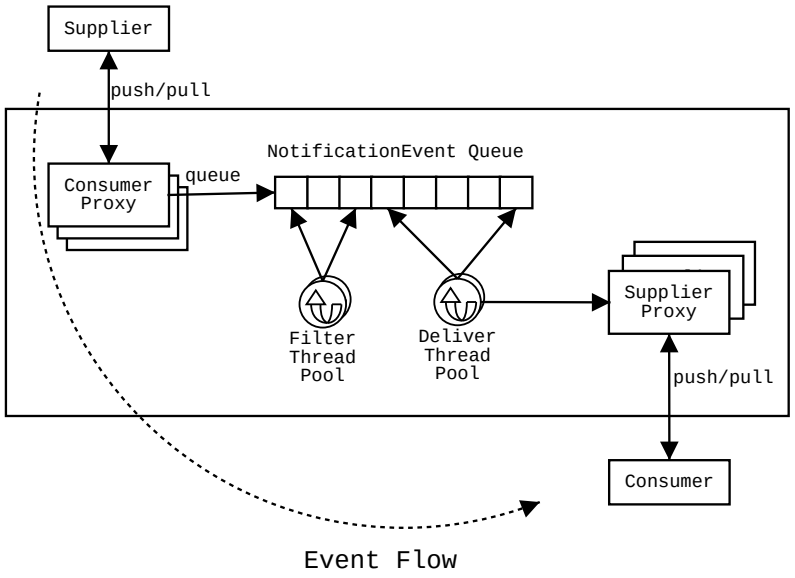

Abbildung 4.6.: Thread Design des JacORB Notification Service

Um den Aufruf einer Operation von deren Ausführung zu trennen, werden die elementaren Operationen – Evaluierung der Filter und Zustellen der Nachrichten – als sog. *Task*-Objekte gekapselt. Dies macht es möglich, sie in eine Warteschlange zu stellen und von den Worker-Threads bearbeiten zu lassen. In der ersten Version des JacORB Notification Service werden noch keine Dienstgüteeinstellungen unterstützt. Das Design erlaubt jedoch, zu einem späteren

[3] http://java.sun.com/docs/hotspot/threads/threads.html

Zeitpunkt eine unterschiedliche Priorisierung der Nachrichten auf einfache Weise zu realisieren, indem die einzelnen Task Objekte in der Warteschlange nach ihrer Priorität geordnet werden.

Zur Realisierung des Threadpools stehen im Java API lediglich primitive Konstrukte (*synchronized*, *Object.wait*, *Object.notify*) zur Verfügung. Die Eigenentwicklung mächtigerer Konstrukte ist langwierig und fehleranfällig. Aus diesem Grund wird in der Implementierung eine externe Bibliothek eingesetzt, die Lösungen für Standardprobleme nebenläufiger Programmierung bietet. Es existieren verschiedene frei erhältliche Bibliotheken. Ein Vorteil der gewählten Bibliothek *util.concurrent* ist die Tatsache, dass sie voraussichtlich Teil des nächsten JDK Version 1.5 *Tiger* wird [62].

ProxyPullConsumer und SequenceProxyPushSupplier Die Proxytypen *ProxyPullConsumer* und *SequenceProxyPushSupplier* stellen ein weiteres Problem dar. ProxyPullConsumer müssen den mit ihnen verbundenen PullSupplier regelmäßig abfragen (Polling). Korrespondierend dazu muss ein SequenceProxyPushSupplier in regelmäßigen Abständen die inzwischen aufgelaufenen Nachrichten an ihren PushConsumer ausliefern.

Diesem Zweck dient es, pro ProxyPullConsumer bzw. SequenceProxyPushSupplier einen Thread zu starten, der in regelmäßigen Abständen die notwendigen Aktionen ausführt. Aus bereits beschriebenen Gründen (vgl. Abschnitt 4.1.8), verbietet sich dieser Ansatz, stattdessen wird, wie in Abschnitt 4.1.9 genannt, ein Schedulingmechanismus realisiert. Dieser Mechanismus erlaubt es, Objekten, die in regelmäßigen Abständen Aufgaben ausführen müssen, ein Callback-Interface bei einem Timer zu registrieren. Sobald der Zeitraum abgelaufen ist, wird das Callback-Interface aufgerufen und die Aufgabe kann bearbeitet werden.

Da nur ein Timer existiert, ist es besonders wichtig, dass sich die Aufgabe schnell ausführen lässt. Ansonsten ist der Timer zu lange blockiert und kann ggfs. andere Aufgaben nicht ausführen. Erfolgt eine länger dauernde Aufgabe wie beispielsweise die Auslieferung von Nachrichten an einen PushConsumer, muss ein entsprechendes Task Objekt einem Worker-Thread-Pool übergeben werden.

In der verwendeten Bibliothek *util.concurrent* steht eine Timerklasse zur Verfügung. Diese wird in der Implementierung eingesetzt.

Verwaltung temporärer Objekte

Wie in Abschnitt 4.1.8 beschrieben, ist eine strenge Ressourcenkontrolle während des Zustellens einer Nachricht notwendig. Idealerweise werden zwischen der ersten *push*-Operation, die eine Nachricht an den Event Channel liefert, und der letzten Auslieferung an einen PushConsumer keine temporären Objekte angelegt.

Es ist schwierig, temporäre Objekte ganz zu vermeiden. Die Anzahl der verwendeten temporären Objekte lässt sich jedoch stark reduzieren, insbesondere

sollen temporäre Objekte wiederverwendet werden. Für eine Implementierung wird daher ein einfacher Objekt Pool Mechanismus entworfen und realisiert.

Jeder Objekt Pool verwaltet Objekte eines Typs. Um ein Objekt zu verwenden, muss es durch Aufruf der Methode *lendObject* ausgeliehen werden und kann dann verwendet werden. Dieser Methodenaufruf ersetzt den Aufruf eines Konstruktors.

Wenn ein Objekt nicht mehr benötigt wird, muss es durch Aufruf der Methode *returnObject(Object)* wieder an den Pool zurückgegeben werden. Es ist wichtig, darauf hinzuweisen, dass das Objekt nach der Rückgabe nicht mehr verwendet werden darf. Das Objekt kann nach der Rückgabe vom Pool einem neuen Benutzer zugeteilt werden. In diesem Fall würde das Objekt von mehreren Nutzern gleichzeitig verwendet werden, was ein undefiniertes Verhalten zur Folge haben kann und daher dringend vermieden werden muss. Leider lässt sich dies nur durch Programmierdisziplin erreichen, da keine entsprechenden Sprachmittel existieren.

Um an verschiedenen Stellen im Notification Service flexibel Objectpooling einsetzen zu können, wird eine abstrakte Basisklasse *ObjectPoolBase* realisiert. Diese Klasse definiert die generischen Verwaltungsoperationen. Spezifische Operationen wie das Anlegen neuer Objektexemplare werden an Subklassen delegiert. In Abbildung 4.7 auf der nächsten Seite ist die Vorgehensweise am Beispiel der Klasse *NotificationEvent* demonstriert.

Zur Vereinheitlichung sind in dieser Implementierung alle Klassen wie *NotificationEvent*, deren Exemplare wiederverwendet werden können, von der abstrakten Basisklasse *Poolable* abgeleitet. Ein Poolable Objekt hält eine Referenz auf den Objektpool, der es erzeugt hat. Daher kann es durch Aufruf der Methode *release* an den Pool zurückgegeben werden. Die abstrakte Methode *reset* wird in Subklassen überschrieben und reinitialisiert das Exemplar.

Die Klasse *NotificationEventFactory* dient als Factory, um Exemplare der Klasse *NotificationEvent* zu erzeugen. Um ein Exemplar zur Verfügung zu stellen, wird die Methode *lendObject* aufgerufen. Die Implementierung von *ObjectPoolBase* prüft, ob ein Objekt im internen Cache zur Verfügung steht. Ist dies nicht der Fall, wird durch Aufruf der Methode *newInstance* ein neues Exemplar erzeugt. Diese Methode wird in der Klasse *NotificationEventFactory* implementiert.

Nachdem ein Objekt nicht mehr benötigt wird, wird es durch Aufruf von *release* freigegeben und an den Pool zurückgegeben. Der Pool ruft die Methode *reset* auf, um das Objekt zu reinitialisieren und fügt es in eine interne Liste ein.

4.2.5. Filter

Parsen eines Filterausdrucks

Jedem Filter können beliebig viele Filterausdrücke zugeordnet werden. Ein Filterausdruck wird in Klartext formuliert. Diese Form erleichtert es Benutzern, Filterausdrücke zu formulieren und zu lesen, kann aber vom Rechner schlecht verarbeitet werden. Üblicherweise wird zur Erkennung von Sprachen ein Parser verwendet. Da für die Filtersprache ETCL eine Grammatik vorliegt, bietet es

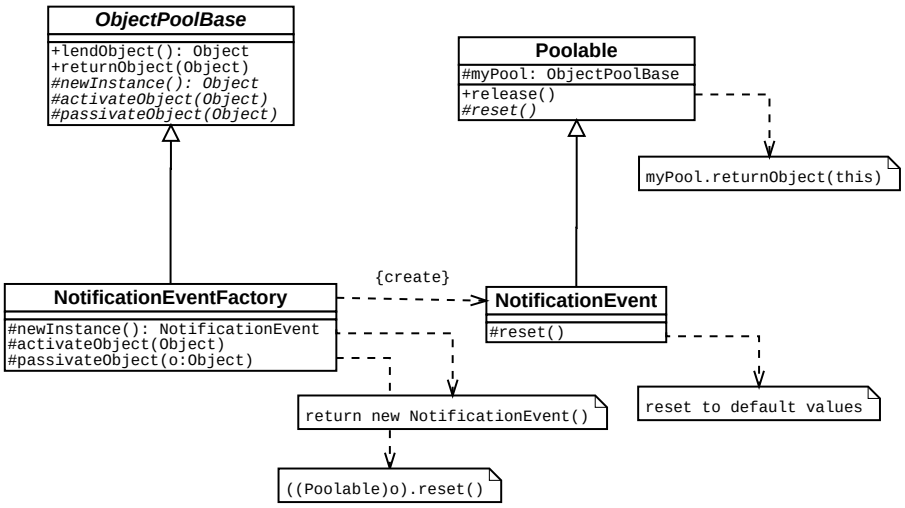

Abbildung 4.7.: Anwendung des Objektpooling

sich an, einen Generator einzusetzen, um einen Parser zu erzeugen.

Ein Parsergenerator erlaubt es, Lexer und Parser aus einer Grammatikdefinition zu generieren. Es ist darauf hinzuweisen, dass handgeschriebene Lexer/Parser eine höhere Verarbeitungsgeschwindigkeit erreichen können. Die Eigenentwicklung eines Lexer/Parser ist allerdings ein langwieriger und fehleranfälliger Prozess. Da das Parsen der Filterausdrücke kein performanzkritischer Teil der Funktionalität des Notification Service ist, wird ein Parsergenerator verwendet, um den Parser für ETCL zu erzeugen.

Das bekannteste Gespann für einen Parsergenerator sind *Lex* (ein Lexergenerator) und *Yacc* (der zugehörige Parsergenerator). Bei der Realisierung des JacORB Notification Service wurde der Parsergenerator *Another Tool for Language Recognition* (ANTLR) verwendet [5]. Der größte Unterschied zwischen Lex/Yacc und ANTLR besteht darin, dass ANTLR ein LL(k) Parsergenerator und Lex/Yacc nur LL(1) Parsergeneratoren sind, was bei Lex/Yacc zu relativ komplexen Grammatiken führt.

Der Parser liefert einen erkannten Ausdruck in Form eines *abstrakten Syntaxbaumes* (AST) zurück. ANTLR bietet dabei zwei Möglichkeiten, den Syntaxbaum aufzubauen. Die Knoten können alle vom gleichen Typ sein (*homogener Syntaxbaum*). Alternativ ist es möglich, einen *heterogenen Syntaxbaum* mit nutzerdefinierten Knotentypen aufzubauen.

Um mit Syntaxbäumen weiterzuarbeiten, bietet ANTLR die Möglichkeit, sog. *TreeParser* zu definieren. Dieser erlaubt es, einen in einer vorherigen Phase generierten abstrakten Syntaxbaum zu transformieren oder Code zu generieren.

Für die Zustellung einer Nachricht müssen die Filterausdrücke evaluiert werden. Der AST muss daher in einem geeigneten Format vorliegen, das eine effiziente Evaluierung zulässt. Eine Möglichkeit ist, aus dem AST Bytecode zu generieren. Um eine Nachricht zu filtern, wird der Bytecode dann interpretiert. Da Java eine interpretierte Sprache ist, bietet es sich an, direkt Java Bytecode aus

dem AST zu generieren, der dann von der Java Virtual Machine (JVM) interpretiert werden kann. Die Generierung von Java Bytecode könnte beispielsweise mit Hilfe der *Byte Code Engineering Library* (BCEL) erfolgen.

Ein einfacherer Ansatz ist, nutzerdefinierte Knotentypen zu verwenden. In den einzelnen Klassen können Operationen zur Evaluierung einer Nachricht implementiert werden. Um eine Nachricht zu evaluieren, wird der AST dann zur Laufzeit traversiert und die entsprechenden Methoden werden aufgerufen.

Das Generieren von Bytecode lässt generell eine höhere Performanz erwarten. Dieser Ansatz ist jedoch wesentlich komplexer zu realisieren. Aus diesem Gründen wird die einfachere zweite Variante realisiert.

Aufbau des abstrakten Syntaxbaumes

Damit die selbstdefinierten Knotentypen von ANTLR verwendet werden können, müssen diese vom vordefinierten Basistyp *antlr.BaseAST* abgeleitet sein. Dieser Basistyp enthält jedoch auch Operationen, die in der Implementierung nicht benötigt werden. Daher sind alle nutzerdefinierten Knotentypen von einer abstrakten Basisklasse *TCLNode* abgeleitet, die wiederum von der Klasse *antlr.BaseAST* abgeleitet ist. Die Klasse *TCLNode* definiert alle Methoden, die zur Verwendung im JacORB Notification Service notwendig sind. In Abbildung 4.8 ist ein Ausschnitt aus der Klassenhierarchie für die Darstellung des abstrakten Syntaxbaums dargestellt. Insgesamt existieren 30 Klassen, die in der Abbildung aus Platzgründen jedoch lediglich angedeutet sind.

Jede AST Klasse enthält die Operationen *left* und *right* zur Traversierung des Syntaxbaumes. Die Operation *evaluate* dient zur Evaluierung einer konkreten Nachricht. Argument dieser Operation ist der Typ *EvaluationContext*. Dieser Typ kapselt eine Nachricht und den Zustand bezüglich der Nachricht.

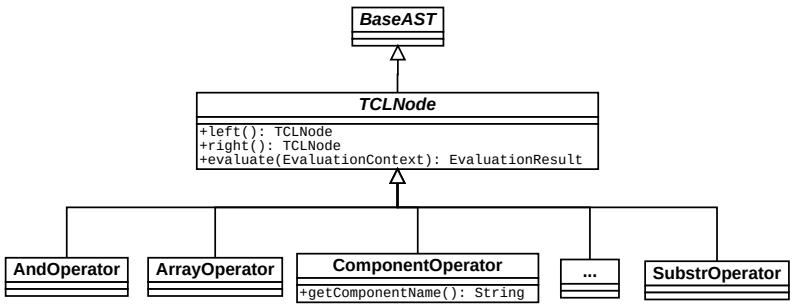

Abbildung 4.8.: Klassenhierarchie für den abstrakten Syntaxbaum

Zuordnung der Nachrichten zu den Filtern

Wie in Abschnitt 4.1.10 dargelegt, ist eine Datenstruktur notwendig, die die Dictionary Operationen unterstützt und bei der Suche Platzhalter im Schlüssel expandieren kann. Diese Datenstruktur wird benötigt, um effizient die Filter

auszuwählen, die zur Bearbeitung einer gegebenen Nachricht evaluiert werden müssen.

Insbesondere die *lookup*-Operation soll möglichst effizient sein. Im folgenden wird eine Datenstruktur namens *WildcardMap* beschrieben, die für diesen Zweck entworfen wurde.

Die Datenstruktur ist an einen *Trie* angelehnt. Die Grundstruktur ist ein n-ärer Suchbaum, in dem Strings als Schlüssel verwendet werden können. Als Werte können beliebige Java Objekte benutzt werden. Gemeinsame Präfixe von Schlüsseln werden jeweils nur einmal gespeichert, um den Platzbedarf zu minimieren.

In Abbildung 4.9 auf der nächsten Seite ist der Aufbau der Datenstruktur skizziert. Im Wurzelknoten existiert ein Index, der nach dem ersten Zeichen des Schlüssels indiziert ist und auf mehrere Kinderknoten zeigt. Jeder Kindknoten verwaltet einen Teil des eingefügten Schlüssels. Im einfachsten Fall verwaltet ein Kindknoten den ganzen eingefügten Schlüssel. In Abbildung 4.9 auf der nächsten Seite ist dies beispielsweise für den Schlüssel *notification* der Fall.

Wird ein weiterer Schlüssel eingefügt, der mit dem gleichen Zeichen beginnt, findet eine Kollision im Kindknoten statt. Die Datenstruktur wird so rekonfiguriert, dass der Kindknoten nur noch den gemeinsamen Präfix des alten und des neuen Schlüssels verwaltet. Unterhalb dieses Knotens werden zwei neue Kindknoten angelegt. Diese werden im Knoten, in dem die Kollision stattgefunden hat, im Index eingetragen. Die beiden neuen Knoten werden anhand des ersten unterschiedlichen Zeichens im Schlüssel identifiziert. Die neuen Kindknoten verwalten den restlichen, nicht gemeinsamen Teil des Schlüssels. In Abbildung 4.9 auf der nächsten Seite ist diese Situation an den Schlüsseln *acme* und *atom* deutlich.

Eine weitere Situation liegt vor, wenn ein Schlüssel echter Präfix eines anderen Schlüssels ist. In diesem Fall wird der Wert bereits an einen inneren Knoten angehängt. In Abbildung 4.9 auf der nächsten Seite sind die Schlüssel *corba*, *corbaevent* und *corbanotification* eingetragen.

Beim Einfügen und Entfernen von Schlüssel/Wertpaaren wird der Platzhalter * nicht besonders behandelt. Um die Operation *lookup* auszuführen, wird im Wurzelknoten anhand des ersten Zeichens des gesuchten Schlüssels entschieden, in welchem Kindknoten weitergesucht wird. Der Schlüssel des Kindknotens wird mit dem Suchschlüssel verglichen. Dabei existieren folgende Fälle:

1. Der Schlüssel des Knotens stimmt mit dem Suchschlüssel überein.
   In diesem Fall ist die Suche abgeschlossen. Der dem Knoten assoziierte Wert kann der Ergebnismenge hinzugefügt werden.

2. Der Schlüssel des Knotens stimmt nicht mit dem Suchschlüssel überein.
   Die Suche in diesem Knoten wird abgebrochen.

3. Der Schlüssel des Knotens stimmt mit dem Suchschlüssel überein. Der Suchschlüssel ist jedoch noch länger.
   Der Knoten verwaltet ggfs. nur einen Präfix des gesamten Schlüssels. Daher muss unterhalb des Knotens weitergesucht werden. Anhand des ers-

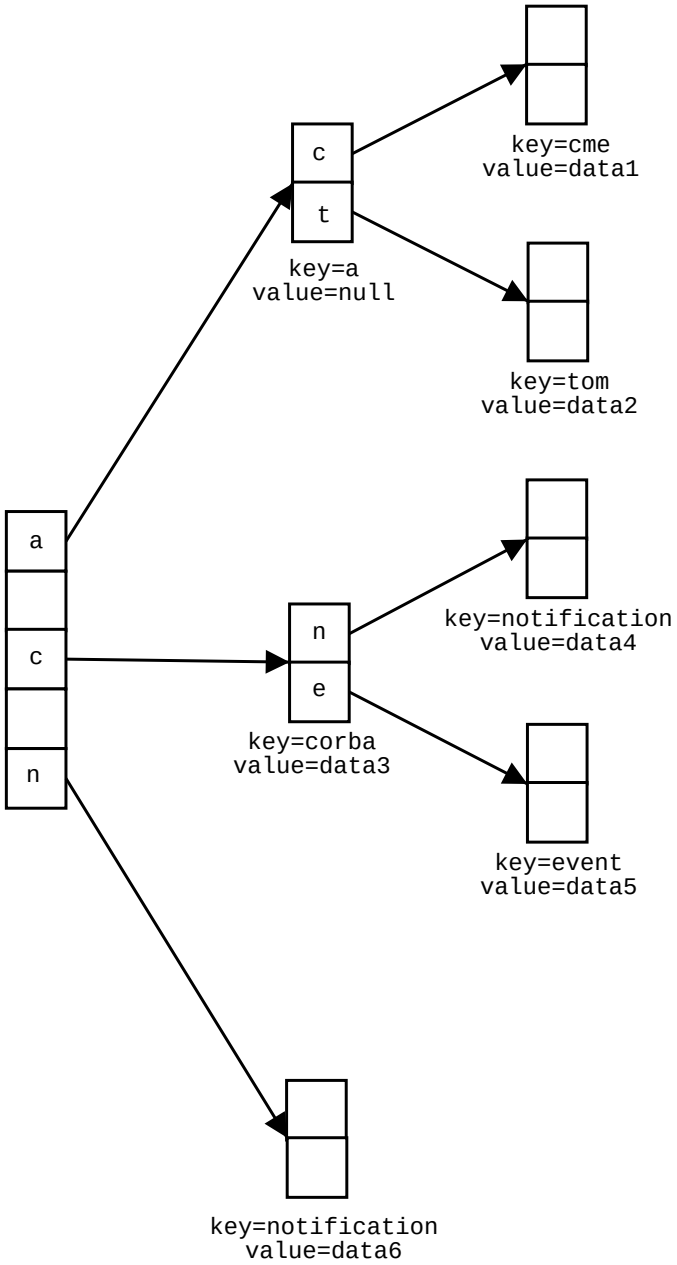

Abbildung 4.9.: Aufbau der Datenstruktur WildcardMap

ten überzähligen Zeichens im Suchschlüssel kann der nachfolgende Knoten bestimmt werden, in dem weitergesucht werden muss. Existiert solch ein Knoten nicht, wird die Suche abgebrochen.

Eine besondere Situation liegt bei der Verarbeitung des Platzhalters * vor. Ausgehend von einem Knoten wird der Kindknoten, in dem weitergesucht wird, anhand des Zeichens an der aktuellen Position im Suchschlüssel bestimmt. Existiert im Knoten auch ein Eintrag für das Zeichen *, so muss der dadurch bestimmte Kindknoten ebenfalls durchsucht werden.

In Abbildung 4.10 auf der nächsten Seite sind beispielsweise die Schlüssel *, *corba*, *corba** und *corbanotification* eingetragen. Angenommen, es wird nun anhand des Suchschlüssels *corbaloc* gesucht und das erste Zeichen des Suchschlüssels ist *c*. Dies identifiziert den unteren Kindknoten des Wurzelknotens. Da im Wurzelknoten der Schlüssel * eingetragen ist, muss auch der dadurch bezeichnete obere Knoten durchsucht werden.

Der obere Knoten enthält lediglich den Schlüssel *. Durch Patternmatching kann festgestellt werden, dass der Schlüssel zum Suchschlüssel passt. Damit wird der Wert *data1* der Ergebnismenge hinzugefügt.

Der untere Knoten enthält den Schlüssel *corba*. Durch Vergleich mit dem Suchschlüssel wird festgestellt, dass der Suchschlüssel zwar passt, aber länger ist. Also wird versucht, einen Kindknoten für das Zeichen *l* zu finden. Dieser existiert nicht. Es existiert jedoch ein Eintrag für *. Der entsprechende Knoten enthält den Schlüssel *corba**. Durch Vergleich des Suchschlüssels *corbaloc* mit dem Schlüssel *corba** wird festgestellt, dass der Schlüssel passt. Damit kann auch der Wert *data4* der Ergebnismenge hinzugefügt werden.

Die weitere Suche schlägt fehl. Daher werden *data1, data4* als Ergebnismenge zurückgeliefert.

Durch Verwendung dieser Datenstruktur ist die Suchzeit nicht mehr linear zur Anzahl der darin enthaltenen Elemente ($O(n)$). Die Suchzeit hängt nun von der Länge des Schlüssels ab ($O(length(key))$).

Effiziente Auswertung der Filter

Wie in Abschnitt 4.1.6 dargestellt, stellt das Extrahieren von Werten aus einem *Any* eine aufwendige Operation dar. Aus diesem Grund soll ein einmal extrahierter Wert zwischengespeichert werden, um zu einem späteren Zeitpunkt wiederverwendet werden zu können.

Die Teilausdrücke, die mit dem Zeichen $ beginnen, bezeichnen ein Element einer Nachricht. Diese Teilausdrücke werden in der Implementierung mit der Klasse *ComponentName* repräsentiert. Diese Klasse bietet die Methode *getComponentName*, um auf den kompletten Teilausdruck (als String) zuzugreifen. Nach Auswertung kann der extrahierte Wert mit dem Teilausdruck als Schlüssel in einer Dictionary Datenstruktur abgelegt werden.

Ein Vorteil dieser Herangehensweise ist, dass Teilausdrücke eines Filters, die den gleichen Wert bezeichnen, nur einmal evaluiert werden müssen. Wenn beispielsweise der Filterausdruck `exist $.nested.value and $.nested.value > 2`

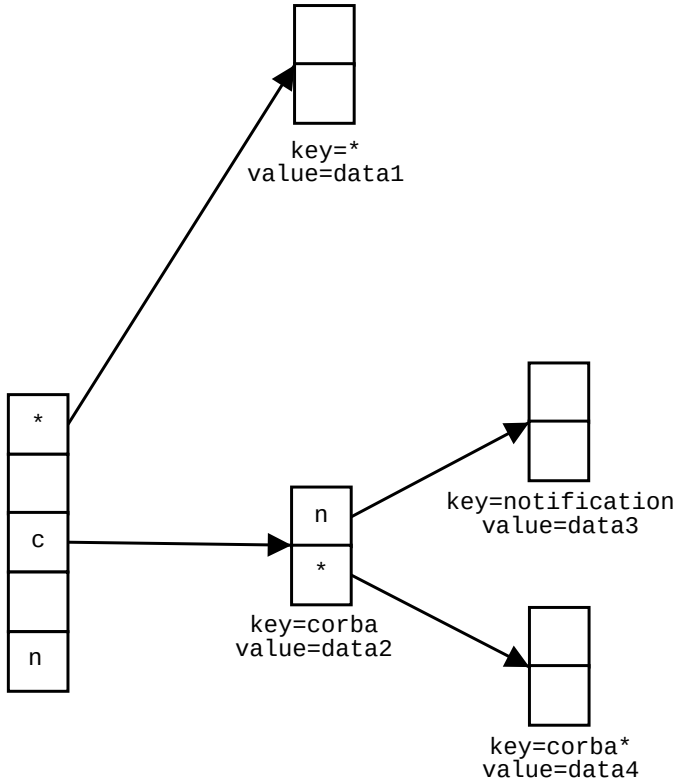

Abbildung 4.10.: Datenstruktur WildcardMap und der Platzhalter *

evaluiert wird, kann der Teilausdruck `$.nested.value` nach dem erstmaligen Evaluieren zwischengespeichert werden. Wird der Teilausdruck das zweite Mal verwendet, muss der Wert nicht erneut extrahiert werden. Die Werte der Teilausdrücke werden in einem Kontext gespeichert, der von der Klasse *EvaluationContext* realisiert wird. Die Gültigkeit dieses Kontextobjekts erstreckt sich auf die Lebensdauer einer *match* bzw. *match_structured*-Operation und damit über die Auswertung aller Filterausdrücke innerhalb eines Filterobjektes.

Teilausdrücke bezeichnen dann den gleichen Wert, wenn sie im Sinne eines Zeichenkettenvergleichs gleich sind. ETCL erlaubt es, Elemente einer Nachricht positionsbasiert und namensbasiert zu dereferenzieren. Daher können auch Teilausdrücke, die nicht gleich sind, den gleichen Wert innerhalb einer Nachricht referenzieren. Entsprechend kann vorkommen, dass ein Wert mehrere Male extrahiert wird, falls Filterausdrücke existieren, die den Wert auf unterschiedliche Weisen referenzieren. Dies zu vermeiden, ist nicht leicht.

Um einen Filterausdruck auf einer Nachricht zu evaluieren, wird die Methode *evaluate* auf der Wurzel des AST aufgerufen. Durch den Methodenaufruf wird der Baum traversiert und auf den einzelnen Knotentypen wird jeweils die Methode *evaluate* aufgerufen. Die Implementierungen der einzelnen Klassen überschreiben die Methode mit der entsprechenden Funktionalität. Rückgabewert der Methode ist der Datentyp *EvaluationResult*. Dieser Typ kapselt das Ergebnis und bietet Konvertierungsfunktionen, um die in ETCL gültigen Typkonvertierungen durchzuführen.

4.2.6. Nachrichtenfluss innerhalb des Systems

Eine Nachricht wird von einem Supplier über einen ProxyConsumer an den Event Channel übergeben. Auf Ebene eines ProxyConsumer, SupplierAdmin, ConsumerAdmin oder ProxySupplier können jeweils Filter existieren, die für die Nachricht evaluiert werden müssen, bis die Nachricht dann schließlich an einen oder mehrere Consumer übergeben wird (vgl. Abb. 3.2 auf Seite 45).

Die verschiedenen Proxytypen bieten gegenüber dem Client unterschiedliche Schnittstellen. Aus Sicht des Nachrichtenflusses können die Proxy- und Consumerklassen jedoch einheitlich behandelt werden.

Der Weg einer Nachricht durch das System lässt sich als Baumstruktur der daran beteiligten Proxy- und Adminobjekte verstehen. Der ProxyConsumer, der die Nachricht erstmalig entgegen nimmt, bildet die Wurzel des Baums. Die Wurzel hat als einzigen Nachfolger den SupplierAdmin, der den ProxyConsumer erzeugt hat. Nachfolger des SupplierAdmin sind alle im Event Channel aktiven ConsumerAdmin-Objekte. ProxySupplier-Objekte bilden die vorletzte Ebene des Baums. Jeder innere Knoten des Baumes ist dadurch ausgezeichnet, dass er einen oder mehrere Nachfolger hat und mehrere Filter assoziiert haben kann (vgl. Abbildung 4.11 auf Seite 75). Die letzte Ebene des Baumes bilden Knoten vom Typ *EventConsumer*. Das Interface EventConsumer abstrahiert von der Zustellmethode eines konkreten ProxySuppliers (vgl. Abschnitt 4.2.2).

Um dies zu repräsentieren, existiert das Interface *FilterStage*, welches die Operation *getFilters* enthält, um auf alle assoziierten Filter zuzugreifen. Mit der

Operation *getSubsequentFilterStages* kann auf die nachfolgenden Knoten zugegriffen werden (vgl. Abbildung. 4.12 auf der nächsten Seite). Die Operation *getEventConsumer* erlaubt es, falls vorhanden, auf den EventConsumer zuzugreifen.

Wie in Abschnitt 4.1.5 beschrieben, wird die Auswertung der Filterausdrücke in Teilschritte zerlegt. Die Klasse *FilterTask* kapselt den aktuellen Zustand der Filterauswertung. Dadurch ist es möglich, nach Ausführung eines Teilschrittes eine andere Aufgabe auszuwählen. Nach dem letzten Filterschritt wird die Nachricht an einen *EventConsumer* zugestellt.

Damit ergibt sich die in Abbildung 4.13 auf Seite 76 dargestellte Gesamtstruktur mit folgenden Klassen:

NotificationEvent Diese Klasse abstrahiert von den konkreten Nachrichtentypen und bietet eine einheitliche Schnittstelle (vgl. Abschnitt 4.2.1).

EventConsumer Abstraktion von einem ProxySupplier. Ein EventConsumer erlaubt es, das generische *NotificationEvent* an einen Consumer auszuliefern.

TaskBase Abstraktion von einer Aufgabe. Eine Aufgabe bezieht sich jeweils auf eine Nachricht.

PushToConsumerTask Diese von *TaskBase* abgeleitete Klasse kapselt die Aufgabe, ein *NotificationEvent* an ein *EventConsumer*-Exemplar zuzustellen.

FilterTask Diese von *TaskBase* abgeleitete Klasse erlaubt es, schrittweise alle Filter für eine Nachricht auszuwerten.

ProxyBase Abstraktion von den Proxyklassen.

AdminBase Abstraktion von den Adminklassen.

ChannelContext Diese Klasse bietet einen Kontext pro *EventChannel*-Instanz. Alle Ressourcen, die einmal pro *EventChannel* verfügbar sein sollen, können hier abgelegt werden.

FilterStage Abstraktion von Proxy- und Adminobjekten. Ein *FilterStage*-Objekt hat mehrere assoziierte Filterobjekte und einen oder mehrere Nachfolger.

TaskProcessor Diese Klasse koordiniert den Nachrichtenfluss innerhalb der Implementierung. Die Aufgaben werden in eine Warteschlange eingestellt und von Worker-Threads bearbeitet (vgl. Abschnitt 4.2.4).

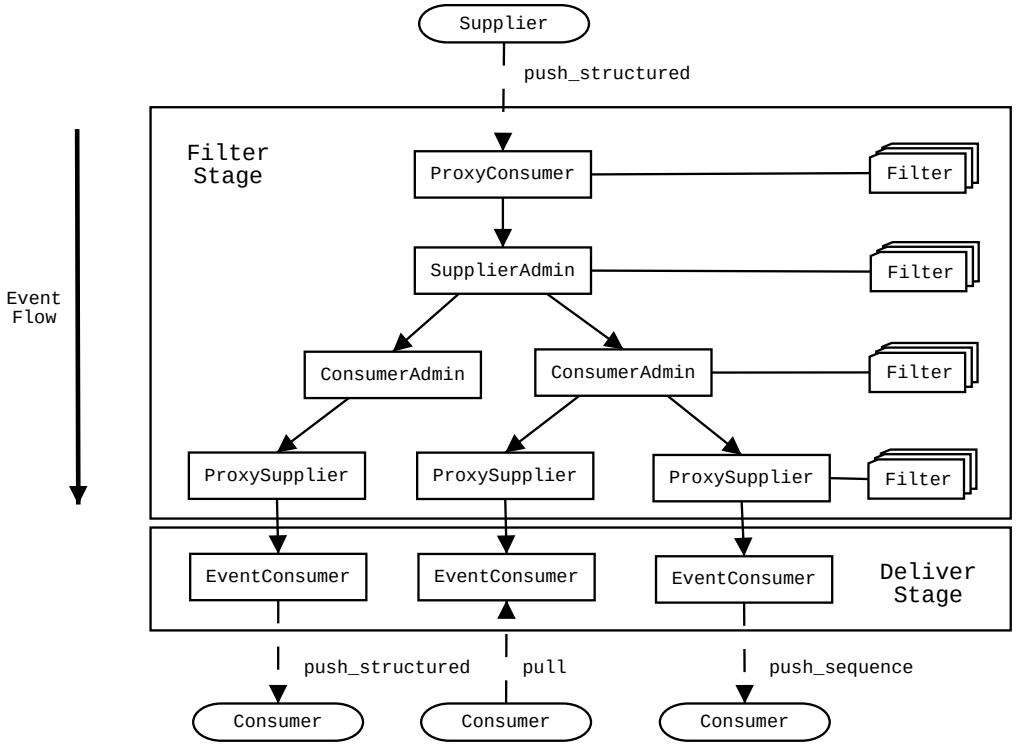

Abbildung 4.11.: Interne Repräsentation der Proxy- und Admin-Objekte

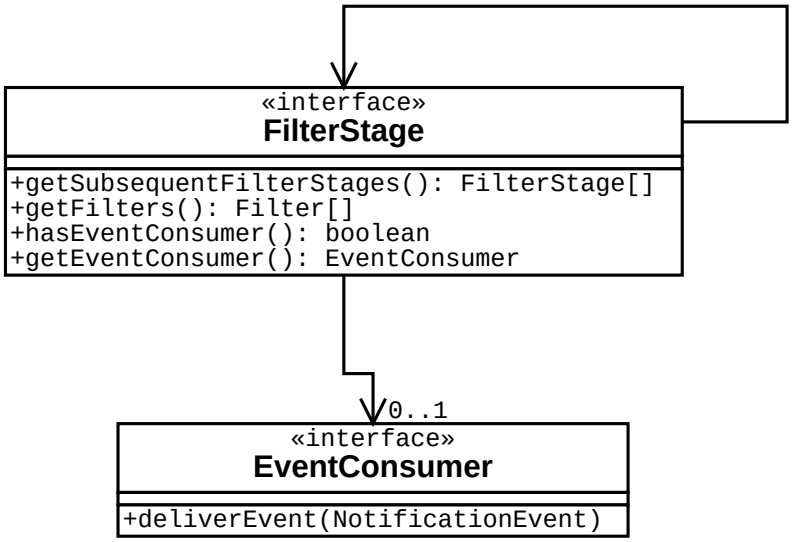

Abbildung 4.12.: Die Interfaces FilterStage und EventConsumer

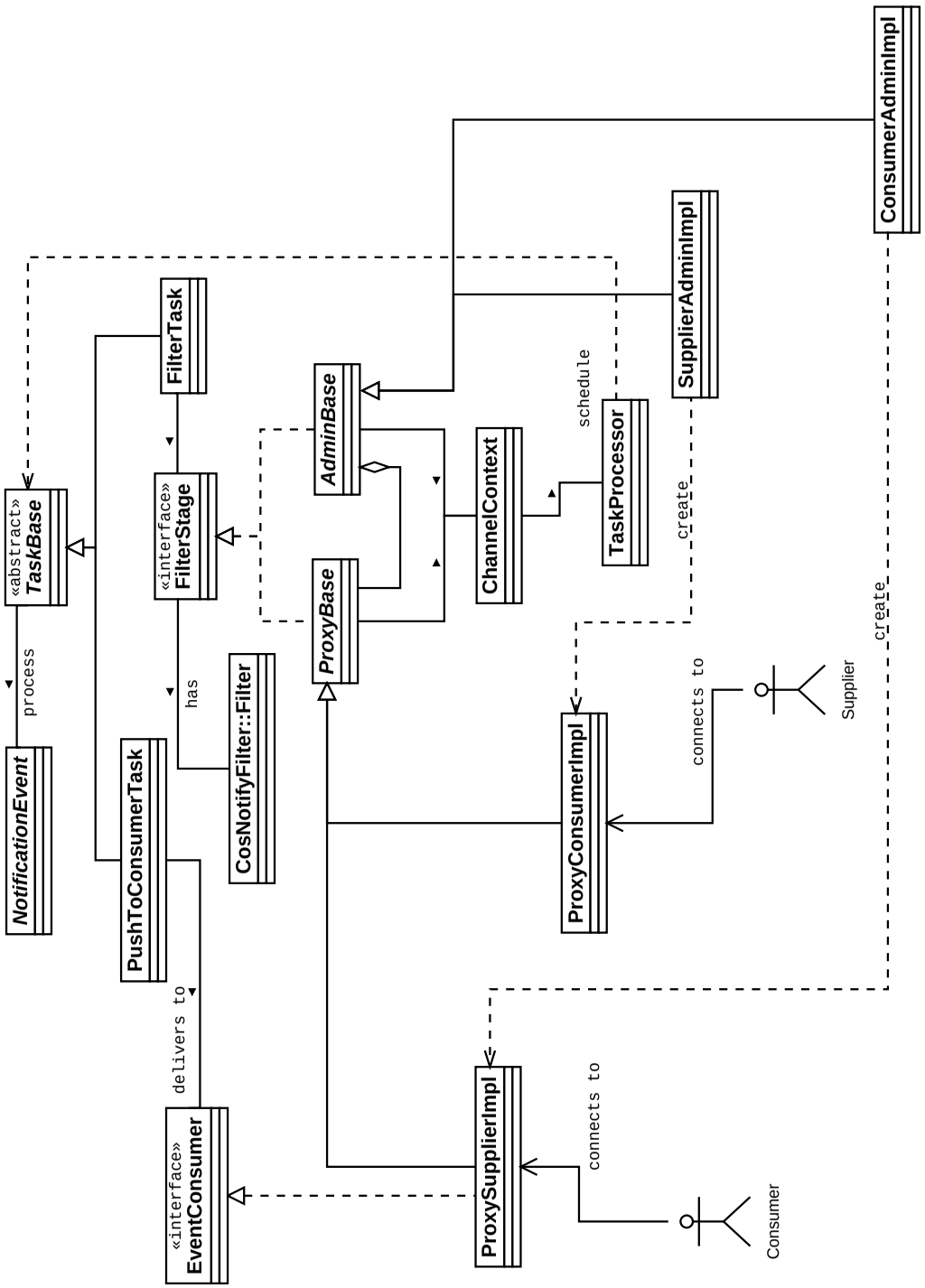

Abbildung 4.13.: Architektur – Klassenmodell

5. Implementierung des JacORB Notification Service

Im vorangegangenen Kapitel wurde die Architektur des JacORB Notification Service beschrieben, die der Implementierung zugrundeliegt. In diesem Kapitel wird vertiefend auf Aspekte der Implementierung eingegangen.

Dabei soll keine erschöpfende, umfassende Beschreibung der Implementierung erfolgen. Vielmehr wurden einige interessante Bereiche ausgewählt und einige nicht unmittelbar offensichtliche Konzepte beschrieben. Die Lektüre dieses Kapitels soll daher insbesondere dem interessierten Leser bei der Einarbeitung in den Implementierungscode dienen.

5.1. Die Packagestruktur

Die erstellten Implementierungsklassen befinden sich im und unterhalb des Java Package *org.jacorb.notification*. Eine Übersicht der Packages findet sich in Abbildung 5.1 auf Seite 79.

org.jacorb.notification Dieses Package enthält die Servantklassen, die die IDL Schnittstellen realisieren. Hier befindet sich ein Großteil der Funktionalität.

org.jacorb.notification.grammar Dieses Package existiert im Sourcebaum, enthält aber keinen Java Quellcode. Stattdessen befinden sich in diesem Verzeichnis die Grammatikbeschreibungen für ANTLR. Aus den Dateien werden Lexer und Parser generiert.

org.jacorb.notification.parser Die von ANTLR generierten Lexer- und Parserklassen befinden sich in diesem Package. Das Verzeichnis existiert daher lediglich nach Generierung im Verzeichnis *src/generated* des JacORB Quellcodebaums.

org.jacorb.notification.node Die Klassen dieses Packages bilden den abstrakten Syntaxbaum, die Repräsentation eines Filterausdrucks. Ein konkreter AST wird vom Parser erzeugt.

org.jacorb.notification.evaluate In diesem Package befinden sich an der Evaluierung von Filterausdrücken beteiligte Klassen.

org.jacorb.notification.interfaces In diesem Package sind verschiedene Java-Interfaces definiert, um eine einheitliche Verwendung der Implementierungsklassen zu erlauben.

org.jacorb.notification.util Hier befinden sich die Hilfsklassen *ObjectPoolBase* und *WildcardMap*.

org.jacorb.notification.engine Dieses Package enthält die Klassen, die den dynamischen Nachrichtenfluss innerhalb des Systems realisieren. Hier befinden sich der Schedulingmechanismus, die Verteilung der Aufgaben auf Worker-Threads, etc.

util.concurrent Diese Bibliothek enthält Lösungen für Standardprobleme nebenläufiger Programmierung. Die Realisierung der Worker-Threads und des Schedulingmechanismus greift beispielsweise auf sie zurück.

antlr Diese Bibliothek besteht aus zwei Anteilen: Eine Entwicklungsbibliothek, die benötigt wird, um Parser und Lexer zu generieren, und eine Laufzeitbibliothek, die benötigt wird, den generierten Parser und Lexer zu verwenden.

Insgesamt wurden 110 Java Quelldateien und 3 Grammatikdateien für den Parsergenerator ANTLR erstellt.[1] Die Anzahl der Codezeilen wurde mit Hilfe des Programms *sloccount* bestimmt, wonach die Implementierung aus über 10000 Codezeilen besteht. In Anhang B ist ein Teil der Ausgabe des Programms mit einer detaillierten Übersicht über die einzelnen Packages und weiteren interessanten Informationen dargestellt.

[1] Stand: April 2003

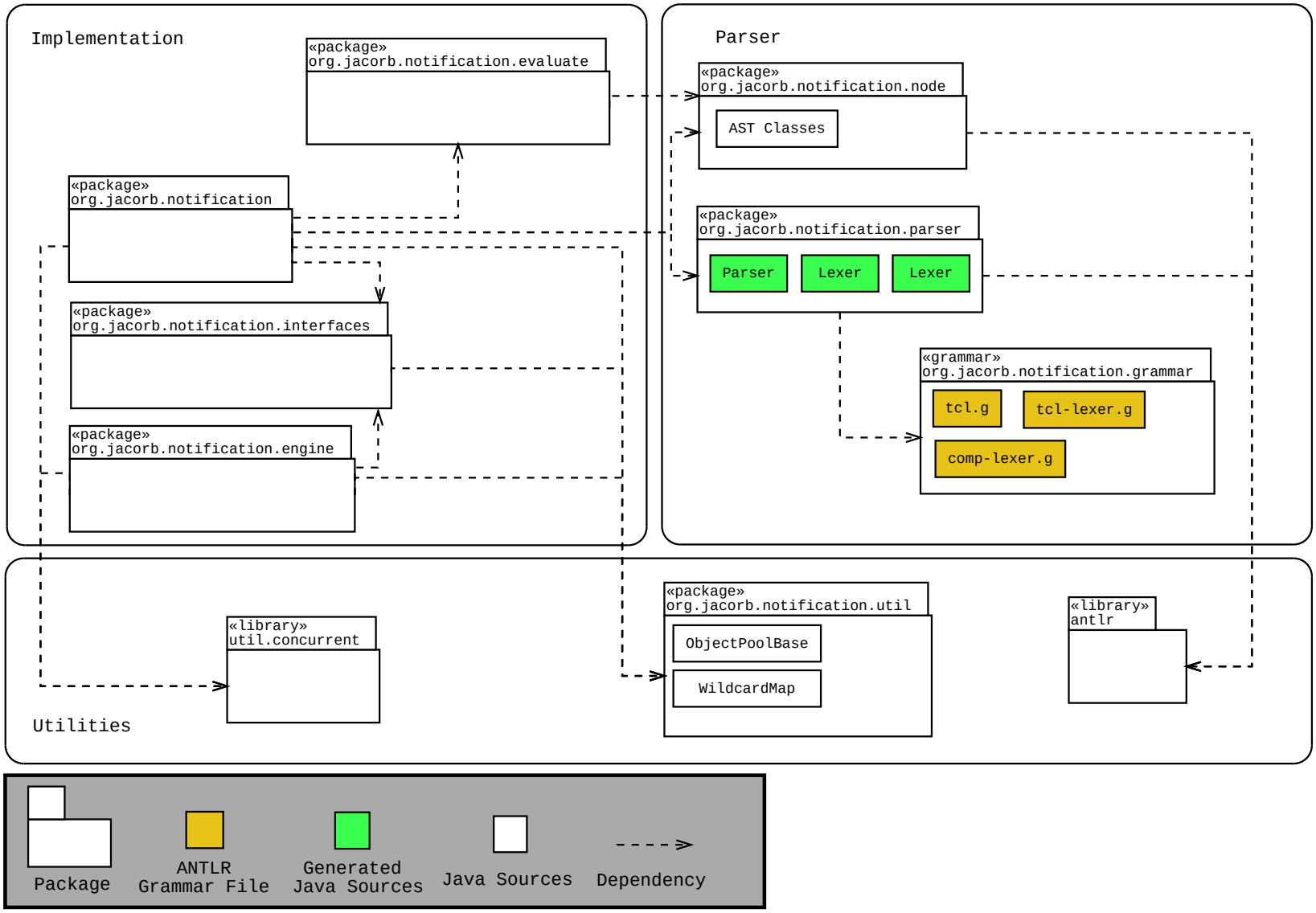

Abbildung 5.1.: Einteilung in Pakete

5.2. Implementierung des Parser

Die Implementierung des Parsers erfolgt mit ANTLR [5].

Der Lexer für ETCL wird in den Dateien `tcl-lexer.g` und `comp-lexer.g` definiert. Für den Lexer sind zwei Dateien vorhanden, da aus folgendem Grund tatsächlich zwei Lexer verwendet werden:

In ETCL sind zum einen Gleitkommaliterale wie beispielsweise `.1` möglich. Andererseits existiert die positionsbasierte Dereferenzierung innerhalb zusammengesetzter Namen (z. B. `$.1`). Auf Ebene des Lexers existiert daher eine Mehrdeutigkeit. Ein Lexer hat keine Kenntnis von vorher gelesenen Tokens. Wird der Text `$.1` gelesen, erkennt der Lexer das Token `DOLLAR`, gefolgt vom Token `FLOAT`. Solche Mehrdeutigkeiten innerhalb eines einzelnen Lexers zu lösen, ist schwierig und hat komplexe Grammatikdateien zur Folge.

ANTLR erlaubt es, einen Parser mit mehreren Lexern zu verwenden. Der erste in der Datei `tcl-lexer.g` definierte Lexer ist für alle Ausdrücke außerhalb eines zusammengesetzten Namens zuständig.

Nach Erkennung des Token `DOLLAR` wird zum in der Datei `comp-lexer.g` definierten Lexer gewechselt, der für die zusammengesetzten Namen verwendet wird. Dieser Lexer ist so lange aktiv, wie Tokens gelesen werden, die noch zum zusammengesetzten Namen gehören. Es wird wieder zum ursprünglichen Lexer gewechselt, sobald ein Operator (`==`, `!=`, `<`, `>`, ...) gelesen wurde.

In der Grammatikdatei `tcl.g` ist der Parser spezifiziert. In dieser Datei ist angegeben, welche Java Klassen von ANTLR zum Aufbau des abstrakten Syntaxbaumes instantiiert werden sollen.

Aus den Grammatikdateien werden Lexer und Parser als Java Sourcecode generiert. Der erzeugte Code wird in den Buildprozess eingebunden und steht damit der Implementierung zur Verfügung.

5.3. Evaluierung eines Filterausdrucks

Ein Filterausdruck liegt in geparster Form als abstrakter Syntaxbaum vor. Die Wurzel dieses Syntaxbaum wird von einem Exemplar der Klasse *FilterConstraint* verwaltet (vgl. Abbildung 5.2 auf der nächsten Seite).

Einem Filter können mehrere Filterausdrücke zugeordnet werden. Dementsprechend verwaltet die Klasse *FilterImpl* mehrere *FilterConstraint*-Exemplare. Die Verwaltung der *FilterConstraint*-Exemplare erfolgt durch ein Objekt der Klasse *WildcardMap*. Als Schlüssel wird *domain_name* und *domain_type* verwendet. Dadurch kann für eine gegebene Nachricht effizient die Menge der auszuwertenden *FilterConstraint*-Exemplare bestimmt werden.

Um einen Filterausdruck in Zusammenhang mit einer Nachricht zu evaluieren, wird die Methode *evaluate* auf der Wurzel des Syntaxbaum aufgerufen. Als Parameter wird ein *EvaluationContext*-Exemplar übergeben. Diese Klasse bildet einen Kontext, dessen Gültigkeit sich über die Lebensdauer einer *match*-Operation erstreckt.

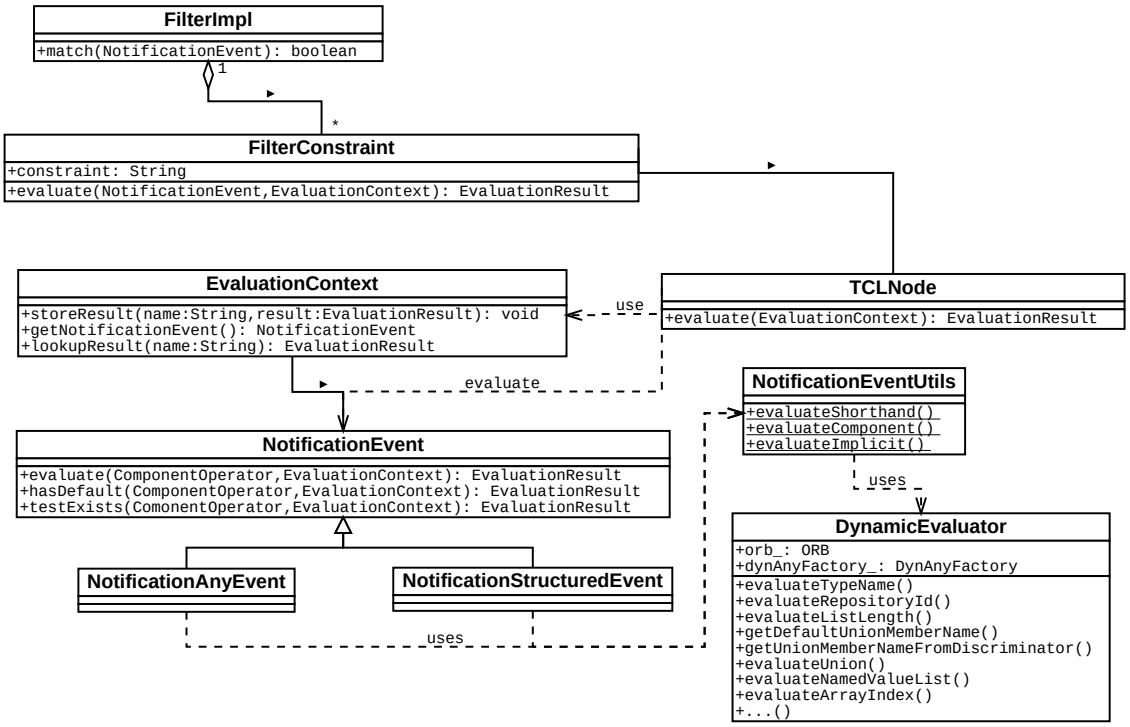

Abbildung 5.2.: An der Filterevaluierung beteiligte Klassen

Die *evaluate*-Operation traversiert rekursiv den Syntaxbaum und führt abhängig von den im Baum enthaltenen Knotentypen die spezifischen Operationen zur Evaluierung aus. Existiert im Baum ein *ComponentName*-Knoten, muss ein Wert dynamisch aus der Nachricht extrahiert werden. Der Pfad zum Wert wird durch die unterhalb des ComponentName nachfolgenden Knoten bestimmt.

Als Beispiel ist in Abbildung 5.3 der Filterausdruck `$.age > 20` als Syntaxbaum dargestellt. Um den Teilausdruck `$.age` auf einer Nachricht zu evaluieren, wird die Methode *evaluate(ComponentName c, EvaluationContext c)* auf dem *NotificationEvent*-Exemplar aufgerufen.

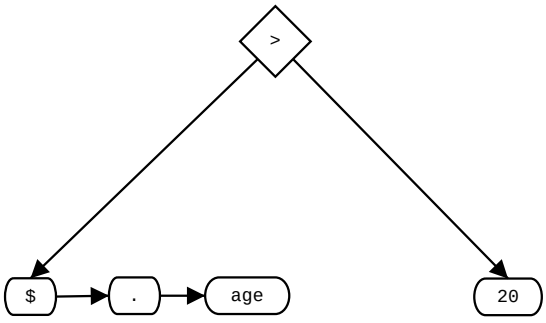

Abbildung 5.3.: Repräsentation eines Filterausdruckes

Die Implementierung dieser Methode extrahiert unter Verwendung der in der Klasse *NotificationEventUtils* definierten Operationen den Wert aus der Nachricht. Der extrahierte Wert wird unter seinem Pfadnamen (in diesem Fall `$.age`) im *EvaluationContext* vermerkt. Existiert im Syntaxbaum ein weiterer *ComponentName*-Knoten, der den gleichen Pfadnamen bezeichnet, muss der Wert nicht erneut extrahiert werden. Stattdessen kann der zwischengespeicherte Wert aus dem *EvaluationContext* verwendet werden. Der Pfadname steht durch Aufruf der Methode *getComponentName* auf dem *ComponentName*-Knoten zur Verfügung.

Ein *EvaluationContext*-Exemplar wird erzeugt, sobald eine Nachricht durch Aufruf einer der Methoden *match* oder *match_structured* einem *FilterImpl*-Exemplar übergeben wird. Ein *EvaluationContext*-Exemplar existiert dann für die Lebensdauer der *match*-Operation. Die Erzeugung der *EvaluationContext*-Exemplare wird an eine Factorymethode delegiert, die die Klasse *ObjectPoolBase* verwendet, um die Anzahl der Objekte zu kontrollieren.

Die Klasse *DynamicEvaluator* verwendet das CORBA DynAny API, um dynamisch Werte aus *StructuredEvent*- und *Any*-Typen zu extrahieren.

5.4. Admin- und Proxy-Objekte

5.4.1. Admin-Objekte

Die Implementierungen der Admin-Objekte SupplierAdmin und ConsumerAdmin sind von der abstrakten Basisklasse *AdminBase* abgeleitet. Diese Basisklasse realisiert die gemeinsame Funktionalität beider Klassen.

Die Klasse *AdminBase* realisiert das Filtermanagement und die Implementierung der Methoden aus dem IDL-Interface *QoSOperations*. Die Methoden aus dem Interface *QoSOperations* enthalten jedoch noch keine Logik. Das Filtermanagement wird an die Klasse *FilterManager* delegiert, die auch von den Proxy-Objekten verwendet wird (vgl. Abb. 4.5 auf Seite 63).

Jedes Admin-Objekt verwaltet eine Menge von Proxy-Objekten. Beim Erzeugen eines neuen Proxy-Objektes wird dieses mit den Einstellungen des Admin-Objekts, das es erzeugt hat, initialisiert. Das Admin-Objekt hält Referenzen auf die von ihm erzeugten Proxy-Objekte, die die Navigation von einem Admin-Objekt zu seinen Proxy-Objekten ermöglichen.

Die mit einem Schlüssel erzeugten Proxy-Objekte (bspw. durch Aufruf der Methode *obtain_notification_push_consumer*), werden – nach ihren Schlüsseln indexiert – in einer Dictionary-Datenstruktur abgelegt.

CosEventProxy-Objekte, die z. B. mit *obtain_push_consumer* erzeugt wurden, haben keinen Schlüssel. Es ist daher nicht möglich, vom Admin-Objekt aus auf sie zuzugreifen. Sie werden daher lediglich in einer Liste gespeichert.

5.4.2. Proxy-Objekte

Alle Proxy-Objekte sind von der abstrakten Basisklasse *ProxyBase* abgeleitet. Wie bei den Admin-Objekten realisiert diese Basisklasse das Filtermanagement.

Jedes Proxy-Objekt ist mit einem Supplier bzw. Proxy assoziiert. Der Client verbindet sich durch Aufruf einer der *connect*-Methoden mit dem Proxy-Objekt. Das Proxy-Objekt hält lediglich eine Referenz auf den Client. Eine Implementierung, die persistente Verbindungen unterstützt, müsste die Referenz zusätzlich persistent speichern, um diese auch nach einem Neustart des Notification Service weiterverwenden zu können.

Die ProxySupplier-Objekte realisieren zusätzlich das Interface *EventConsumer*, das von den Eigenheiten der einzelnen ProxySupplier abstrahiert. Zentrale Methode ist *deliverEvent*. Diese Methode akzeptiert ein *NotificationEvent* und wird von einem *DeliverTask* aufgerufen. Die dahinterliegende Implementierung ist dafür verantwortlich, die Nachricht in das richtige Format zu konvertieren und dem tatsächlichen Consumer zuzustellen. Die Logik und der sich daraus ergebende Aufwand sind abhängig vom konkreten ProxySupplier-Typ:

- Ein ProxyPushSupplier stellt die Nachricht unverzüglich durch Aufruf der Methode *push* dem Consumer zu.
- Ein ProxyPullSupplier stellt die Nachricht in eine interne Warteschlange. Aus dieser kann der Consumer die Nachricht durch Aufruf der Methode *pull* auf dem ProxyPullSupplier entnehmen.
- Ein SequencePushSupplier stellt eine Mischform dar. Dieser Suppliertyp sammelt zunächst die Nachrichten in einer internen Warteschlange, bis entweder die Warteschlange eine bestimmte Größe überschreitet oder ein bestimmtes Zeitintervall abläuft (vgl. Abschnitt 5.5.3). Danach wird die Sequenz der zwischengespeicherten Nachrichten durch Aufruf von *push_structured_events* an den Consumer übertragen.

5.4.3. FilterStage

Die IDL-Schnittstellen definieren die Möglichkeit, vom EventChannel aus über die Admin-Objekte zu den Proxy-Objekten zu navigieren. Diese Sichtweise bildet die hierarchische Organisation des EventChannel ab und ist aus Nutzersicht sinnvoll. Innerhalb der Implementierung wird eine zusätzliche Sichtweise verwendet. Das interne verwendete Interface *FilterStage* definiert eine Baumsicht auf die Proxy- und Admin-Objekte innerhalb eines EventChannel (vgl. Abbildung 4.11 auf Seite 75).

Das Interface erlaubt es, über die Proxy- und Admin-Objekte zu traversieren und sie in der Reihenfolge zu besuchen, wie es für die Abarbeitung der Filter notwendig ist.

Die Implementierung der Methode *getSubsequentFilterStages* innerhalb der ProxyConsumer-Klassen liefert eine konstante, einelementige Liste zurück, die das SupplierAdmin-Objekt enthält, das vom ProxyConsumer-Objekt erzeugt wurde.

Die SupplierAdmin-Implementierung der Methode liefert eine Liste der aktiven ConsumerAdmin-Objekte zurück. Dementsprechend liefert die ConsumerAdmin-Implementierung eine Liste aller ProxySupplier-Objekte. Diese Lis-

te bestimmt für jeden Teilschritt der Filterauswertung, welche Filter welcher Admin- bzw. Proxy-Objekte ausgewertet werden müssen. Sobald beispielsweise der Teilschritt „Evaluierung der SupplierAdmin-Filter“ erfolgt ist, wird ein Task-Objekt für die Evaluierung aller ConsumerAdmin-Filter erzeugt und in die Ausführungsqueue gestellt.

An dieser Stelle existiert ein Problem: Zwischen Erzeugung eines Task und dessen Ausführung kann prinzipbedingt einige Zeit vergehen. In dieser Zeit können einige der in der Liste eingetragenen Objekte ungültig werden, wenn z. B. ein ProxySupplier freigegeben wird. Das freigegebene Objekt darf nicht sofort aus der Liste gelöscht werden, da ein FilterThread diese Liste womöglich gerade verwendet. Aus diesem Grund ist im Interface *FilterStage* die Methode *isDisposed* definiert, die es möglich macht zu prüfen, ob ein Proxy- bzw. Admin-Objekt noch gültig ist. Beim Iterieren der Liste muss also vor Bearbeitung auf jedem Element die Methode *isDisposed* aufgerufen werden, um festzustellen, ob das Element noch verwendet werden darf.

Die Liste wird bei Aufruf von *getSubsequentFilterStages* bei Bedarf neu generiert. FilterTasks erhalten dann die neue aktualisierte Liste.

5.5. Engine

Die Klassen im Package *org.jacorb.notification.engine* realisieren den Nachrichtenfluss und die damit verbundenen Scheduling-Algorithmen innerhalb des Systems.

5.5.1. Die Task-Objekte

Die Operationen des Notification Service werden in Teilaufgaben aufgeteilt. Aufgaben werden durch den Typ *Task* repräsentiert. Alle Tasks realisieren das Interface *org.jacorb.notification.engine.Task*, das wiederum vom Standard-Interface *java.lang.Runnable* abgeleitet ist. Ein Task verfügt über zwei Methoden:

run Diese Methode wird in Subklassen überschrieben und realisiert die Funktionalität des Task-Objekts. Die Abhängigkeit vom Interface *Runnable* ergibt sich, da ein Task-Objekt von einem Thread ausgeführt wird und diese Methode den Einstiegspunkt für den Thread darstellt.

getStatus Diese Methode erlaubt es, den Zustand eines Task-Objektes abzufragen. Die möglichen Zustände eines Task sind in Abbildung 5.4 auf der nächsten Seite dargestellt. Ein Task ist zunächst im initialen Zustand *NEW*. Nach Aufruf der Methode *run* geht der Task in den impliziten Zustand *RUNNING* über. Wird der Task erfolgreich beendet, befindet er sich im Zustand *DONE*. Falls eine Exception ausgelöst wird, geht der Task in den Zustand *ERROR* über. Falls der Task erfolgreich war, jedoch erneut aufgerufen werden muss, um seine Aufgabe zu vollenden, befindet sich der Task im Zustand *RESCHEDULE*.

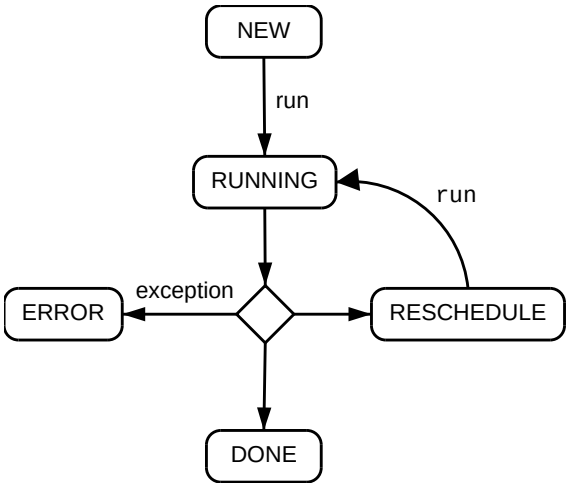

Abbildung 5.4.: Zustandsdiagramm eines Task

Die gemeinsame Teilfunktionalität aller Task-Objekte ist in der abstrakten Basisklasse *TaskBase* zusammengefasst. Diese Klasse definiert ein einfaches Algorithmenskelett.

Für verschiedene Aufgaben existieren unterschiedliche Task-Klassen (Abbildung 5.5 auf der nächsten Seite). Diese überschreiben die Methode *doWork* mit der jeweiligen Funktionalität. Die Methode wird in der Klasse *TaskBase* von der Schablonenmethode *run* aufgerufen.

Die *run*-Methode ruft nach erfolgreicher Ausführung von *doWork* die Methode *handleTaskFinished* am assoziierten *TaskFinishHandler* auf. Tritt bei der Ausführung von *doWork* eine Exception auf, wird die Methode *handleTaskError* am assoziierten *TaskErrorHandler* aufgerufen (vgl. Listing 5.1 auf der nächsten Seite).

Durch Verwendung der TaskHandler kann die Implementierung an dieser Stelle einfach erweitert werden. Wenn beispielsweise eine neue Fehlerbehandlungsstrategie für DeliverTasks realisiert wird, muss die Realisierung vom Interface *TaskErrorHandler* durchgeführt werden. Ein neuer Task kann dann leicht mit dem neuen ErrorHandler konfiguriert werden und ruft diesen auf, falls bei seiner Ausführung eine Exception auftritt. Es müssen keine Änderungen am Task selbst durchgeführt werden.

Wie in den Abschnitten 4.1.5 und 4.2.6 beschrieben, wird die Bearbeitung einer Nachricht in fünf Teilschritte aufgeteilt: Die Evaluierung der den ProxyConsumer- und SupplierAdmin-Objekten zugeordneten Filter-Objekten (Schritte 1 und 2) wird von der Klasse *FilterIncomingTask* realisiert. Die Klasse *FilterOutgoingTask* evaluiert die den ConsumerAdmin- und ProxySupplier-Objekten zugeordneten Filter (Schritte 3 und 4). Schließlich wird die Zustellung der Nachricht (Schritt 5) von der Klasse *PushToConsumerTask* realisiert.

Die Task-Klassen interagieren mit FilterStage- und EventConsumer-Interfaces. Dadurch wird davon abstrahiert, welche tatsächliche Klasse angesprochen wird. Für die Implementierung von FilterIncomingTask ist es beispielsweise ir-

Listing 5.1: Implementierung der Schablonenmethode *run*

```
abstract class TaskBase implements Task {
  TaskFinishHandler taskFinishHandler_;
  TaskErrorHandler taskErrorHandler_;

  // override in subclasses
  abstract protected void doWork();

  // template method
  public void run() {
    try {
       doWork()
       taskFinishHandler_.handleTaskFinished(this);
    } catch (Throwable t) {
       taskErrorHandler_.handleTaskError(this, t);
    }
  }
}
```

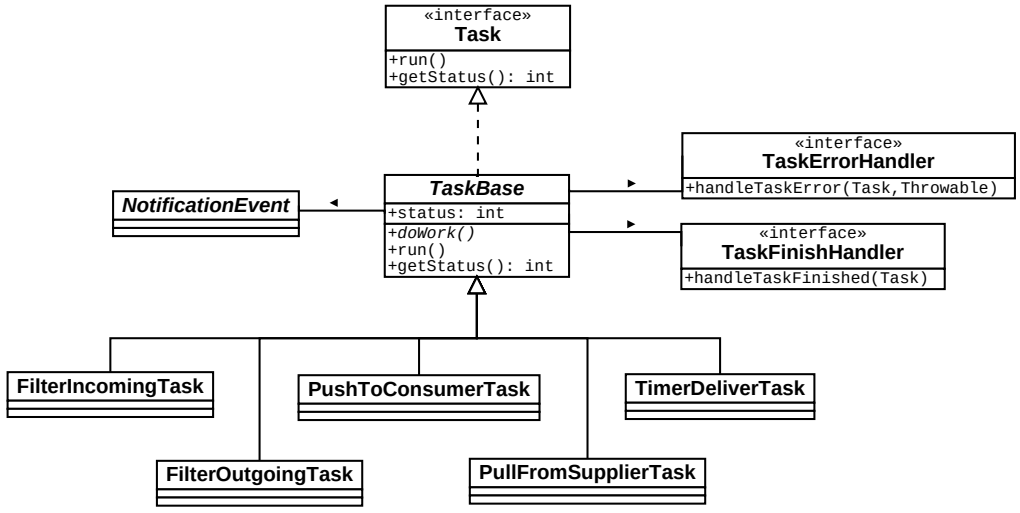

Abbildung 5.5.: Klassenhierarchie der Task-Klassen

relevant, ob gerade ein ProxyPushConsumer oder ein SequenceProxyPullConsumer bearbeitet wird.

Die Implementierung von *PushToConsumerTask* kann durch dieses Konzept ebenfalls sehr einfach gehalten werden. Die *doWork* Methode ruft lediglich die *deliverEvent* Methode des assoziierten EventConsumer auf, die auch die tatsächliche Zustellung realisiert.

Der Laufzeitaufwand für diese Methode hängt davon ab, welchen konkreten ProxySupplier-Typ der Task bearbeitet. Beliefert der Task beispielsweise einen ProxyPushSupplier, so wird die Nachricht synchron an den Consumer zugestellt. Beliefert der Task hingegen einen ProxyPullSupplier, so wird die Nachricht in eine Warteschlange eingereiht. Erst der Consumer bestimmt durch Aufruf der Methode *pull* den Zeitpunkt der Zustellung.

5.5.2. Konfiguration der Tasks

Die Klasse *TaskProcessor* bietet die Operationen, die für den Nachrichtenfluss innerhalb des Systems notwendig sind (vgl. Abbildung 5.6).

Die Klasse kooperiert eng mit der Klasse *TaskConfigurator*. Sie enthält die Logik, um Task-Objekte zu instantiieren und zu konfigurieren, und in ihr sind auch die verwendeten TaskError- und TaskFinishHandler definiert.

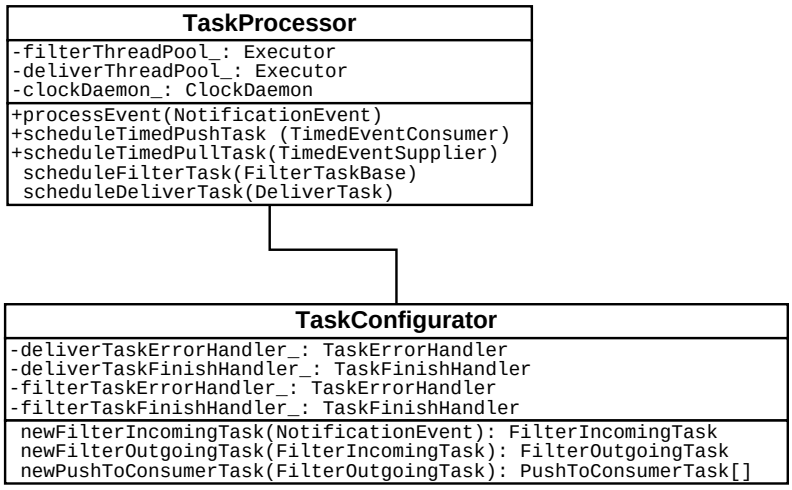

Abbildung 5.6.: Die Klassen Task Processor und Task Configurator

Die Interaktion von *TaskProcessor* und *TaskConfigurator* wird durch die Darstellung der Abbildung 5.7 auf der nächsten Seite näher erläutert. Eine Nachricht (*Any*, *StructuredEvent* oder *Sequenz von StructuredEvent*) wird zunächst von einem ProxyConsumer angenommen. Dieser erzeugt zuerst ein Exemplar der Klasse *NotificationEvent*, das mit der Nachricht (*Any* oder *StructuredEvent*) konfiguriert wird (nicht in der Abbildung dargestellt), und danach übergibt er das *NotificationEvent* durch Aufruf von *processEvent* an den *TaskProcessor* (Schritt 1).

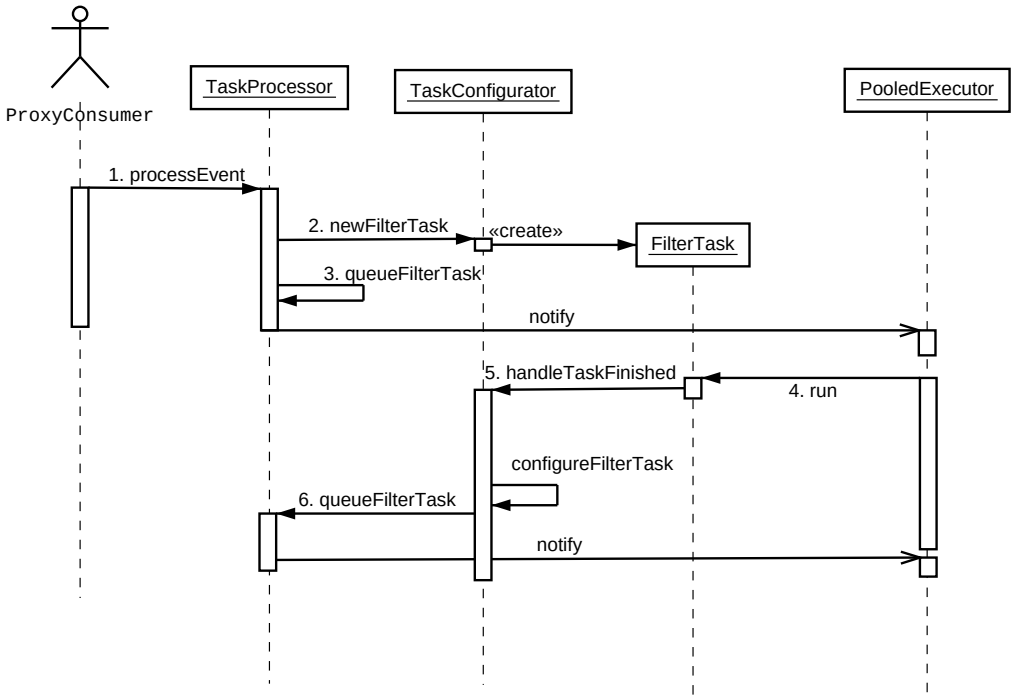

Abbildung 5.7.: Sequenzdiagramm: Scheduling eines FilterTask

Der *TaskProcessor* erzeugt ein *FilterTask*-Objekt (Schritt 2). Dieses wird mit dem zu bearbeitenden *NotificationEvent* konfiguriert. Die Erzeugung und Initialisierung des Objekts wird an die Klasse *TaskConfigurator* delegiert. Das neue *FilterTask*-Objekt wird dann vom *TaskProcessor* dem *PooledExecutor* übergeben (Schritt 3). *PooledExecutor* ist eine Hilfsklasse aus der Bibliothek *util.concurrent*. Diese Klasse realisiert einen Threadpool mit interner Warteschlange, in die neue Aufgaben eingestellt werden können [62]. Der aufrufende Thread ist an diesem Punkt beendet. Im Falle einer *push*-Operation kann der Kontrollfluss zum Supplier zurückkehren.

Die Worker-Threads des *PooledExecutor* entnehmen ständig die Task-Objekte aus der Warteschlange und führen sie durch Aufruf von *run* aus (Schritt 4). Wie beschrieben, ist diese Methode als Schablonenmethode realisiert. Nach erfolgreicher Ausführung wird der *TaskFinishHandler* aufgerufen (Schritt 5). Der *TaskFinishHandler* für FilterTask-Objekte ist als Instanzvariable der Klasse *TaskConfigurator* realisiert, der *TaskFinishHandler* rekonfiguriert den Task und übergibt den neuen bzw. rekonfigurierten Task erneut dem *TaskProcessor*, der ihn dann ausführt (Schritt 6).

Nachdem die vier Filterstufen durchlaufen wurden, kann die Nachricht dem Consumer zugestellt werden (vgl. Abbildung 5.8 auf der nächsten Seite). Nach Beendigung des letzten Filterdurchlaufs wird erneut der *TaskFinishHandler* für FilterTasks aufgerufen (Schritt 1). Der *TaskFinishHandler* erzeugt für jeden Consumer, dem eine Nachricht zugestellt werden soll, ein *DeliverTask*-

Exemplar und übergibt dieses dem *TaskProcessor* zur Ausführung (Schritte 2 und 3). Der TaskProcessor führt die Tasks mit Hilfe eines weiteren PooledExecutor aus (Schritt 4).

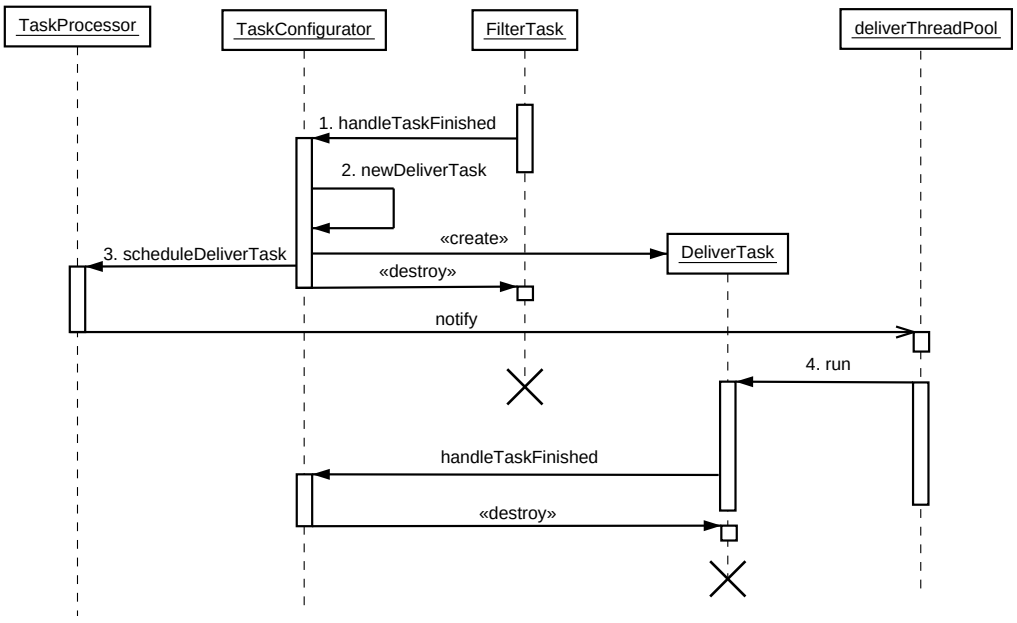

Abbildung 5.8.: Sequenzdiagramm: Scheduling eines DeliverTask

Um die Anzahl der Task-Objekte zu kontrollieren, ist es nötig, diese nicht ständig neu zu erzeugen. Geschähe dies, würden pro Nachricht zwei *FilterTask*-Objekte und, abhängig von der Anzahl der Consumer, mehrere *DeliverTask*-Objekte erzeugt werden. Die Folge wäre, wie in Abschnitt 4.1.8 ausgeführt, eine schlechte Performanz. Stattdessen werden im *TaskConfigurator* die Factory-Methoden *newFilterTask* und *newDeliverTask* aufgerufen. Diese Methoden liefern ein entsprechendes Task-Objekt aus einem Objekt-Pool zurück. Nachdem das Task-Objekt nicht mehr verwendet wird, wird es in den Objekt-Pool zurückgegeben.

Die Spezifikation des Notification Service sieht die Möglichkeit vor, Nachrichten unterschiedlich zu priorisieren. Diese Funktionalität ist noch nicht implementiert. Das realisierte System lässt sich auf einfache Weise erweitern, um die Priorisierung unterschiedlicher Nachrichten zu beachten. Die Klasse *PooledExecutor* verwaltet eine einstellbare Anzahl von Threads. Bei Erzeugung eines *PooledExecutor*-Exemplars lässt sich eine *ThreadFactory* angeben, die konsultiert werden soll, wenn der *PooledExecutor* neue Threads erzeugen muss. An dieser Stelle lässt sich die Priorität der Threads konfigurieren. Damit könnten beispielsweise die Filter-Threads höher priorisiert werden als die Deliver-Threads. In der jetzigen Implementierung existiert lediglich ein globaler *TaskProcessor*. Daher ist es nicht möglich, die Threads, die einen einzelnen Supplier beliefern, höher zu priorisieren.

Wenn ein Task-Objekt dem *PooledExecutor* übergeben wird und ein Thread

verfügbar ist, wird das Task-Objekt unverzüglich ausgeführt. Andernfalls wird das Task-Objekt in eine interne Warteschlange gestellt, aus der es dann später zur Bearbeitung von einem Thread entnommen wird. Die Möglichkeit, die verwendete Warteschlange zu konfigurieren, erlaubt an dieser Stelle die Anwendung einer Warteschlange, die die Priorisierung der in sie eingefügten Elemente beachtet.

Zunächst müsste die Schnittstelle der Task-Objekte um eine Methode erweitert werden, die eine Ordnung auf den Task-Objekten definiert. Dazu könnte bspw. das Interface *java.lang.Comparable* verwendet werden. Die Implementierung der Methode *compareTo* muss ermöglichen, unterschiedliche Task-Objekte hinsichtlich der Priorität der assoziierten *NotificationEvent*-Exemplare zu vergleichen. Die Priorität einer Nachricht lässt sich auf Ebene des EventChannel, des Admin-Objekts, des Proxy-Objekts oder auf Ebene der einzelnen Nachricht konfigurieren.

Im nächsten Schritt müsste eine Datenstruktur realisiert werden, um diese als Warteschlange im *PooledExecutor* einzusetzen. Standardmäßig wird dort eine einfach verkettete Liste verwendet. Eine alternative Implementierung muss vom Interface *EDU.oswego.cs.dl.util.concurrent.Channel* abgeleitet sein. Diese Implementierung muss die einfügten Task-Objekte hinsichtlich ihrer Priorität ordnen. Hierfür kann beispielsweise eine eine Standardimplementierung unter Verwendung eines *Heap* verwendet werden.

5.5.3. Der Schedulingmechanismus

Eine besondere Situation liegt bei ProxyPullConsumer- und SequenceProxyPushSupplier-Objekten vor (vgl. Abschnitte 4.1.9 und 4.2.4). Diese Proxy-Typen erfordern es regelmäßig, *push*- bzw. *pull*-Operationen auszuführen. Wie beschrieben, wird dafür ein Schedulingmechanismus realisiert.

In der verwendeten Bibliothek *util.concurrent* existiert die Klasse *ClockDaemon*, die zu diesem Zweck verwendet wird. Die Klasse *Engine* verwendet ein *ClockDaemon*-Exemplar, um das Scheduling zu realisieren. Die Klasse *ClockDaemon* bietet die Methode *executePeriodically (long period, java.lang.Runnable command, boolean startNow)*.

Sobald sich ein PullSupplier mit einem ProxyPullConsumer verbindet (vgl. Schritt 1 in Abbildung 5.9 auf der nächsten Seite), registriert der ProxyPullConsumer eine Callback-Methode bei dem *TaskProcessor* (Schritt 2). Der *TaskProcessor* verwendet den *ClockDaemon* aus der Bibliothek *util.concurrent*, um die Callback-Methode zu registrieren. Der *ClockDaemon* ruft die Callback-Methode dann regelmäßig auf (Schritt 3). Der *ClockDaemon* darf nicht zu lange blockiert werden, weil sonst eine langdauernde Operation die Ausführung anderer Aufgaben verhindern könnte. Aus diesem Grund müssen die Operationen, die vom *ClockDaemon* ausgeführt werden, schnell terminieren. Insbesondere dürfen Operationen wie *Object.wait* oder *Thread.sleep* nicht aufgerufen werden. Das trifft auf die Methode *pull* zu, die man deshalb nicht direkt in der Callback-Methode ausgeführt. Stattdessen wird ein *PullFromSupplierTask*-Exemplar instantiiert und in die Ausführungsqueue gestellt (Schritt 4).

Dieses Task-Objekt wird von einem Worker-Thread ausgeführt und führt die tatsächliche *pull*-Operation durch (Schritt 5). Die vom PullSupplier abgeholte Nachricht wird dann vom Event Channel weiterverarbeitet.

Der Vorgang wiederholt sich gemäß des angegebenen Zeitintervalls regelmäßig, bis der PullSupplier die Verbindung mit dem ProxyPullConsumer trennt (Schritt 6). Dies veranlasst den ProxyPullConsumer, die Registrierung beim *ClockDaemon* zu löschen.

Eine ähnliche Vorgehensweise existiert beim SequenceProxyPushSupplier. Er liefert *Sequenzen von StructuredEvent* an seinen Consumer aus. Der ProxySupplier wartet dabei jeweils so lange, bis eine bestimmte Anzahl von Nachrichten aufgelaufen ist, bevor er die nächste *push*-Operation aufruft. Zusätzlich existiert ein Zeitintervall, nachdem eine Auslieferung erzwungen wird.

Die Realisierung dieses Mechanismus ist in Abbildung 5.10 auf der nächsten Seite dargestellt. Nachdem der Consumer sich mit Schritt 1 bei dem ProxySupplier registriert hat, registriert der Supplier eine regelmäßige Aufgabe beim Scheduler (Schritt 2). Im weiteren Verlauf werden Nachrichten vom *TaskProcessor* an den ProxySupplier geliefert (Schritt 3). Der ProxySupplier fügt die Nachrichten in eine interne Warteschlange ein. In regelmäßigen Abständen wird der ProxySupplier vom *ClockDaemon* aufgerufen, um die aufgelaufenen Nachrichten zuzustellen (Schritt 4). Um den *ClockDaemon* nicht zu lange zu blockieren, registriert der ProxySupplier eine Aufgabe beim *TaskProcessor*, und während der Bearbeitungsdauer werden die Nachrichten dem Consumer zugestellt (Schritt 5). Zusätzlich kann der Aufruf von *deliverEvent* direkt eine Zustellung an den Consumer veranlassen, falls die Größe der Warteschlange einen bestimmten Wert überschreitet (Schritt 6).

Wenn sich der Consumer vom ProxySupplier abmeldet, wird die regelmäßige Aufgabe beim Scheduler gelöscht (Schritt 7).

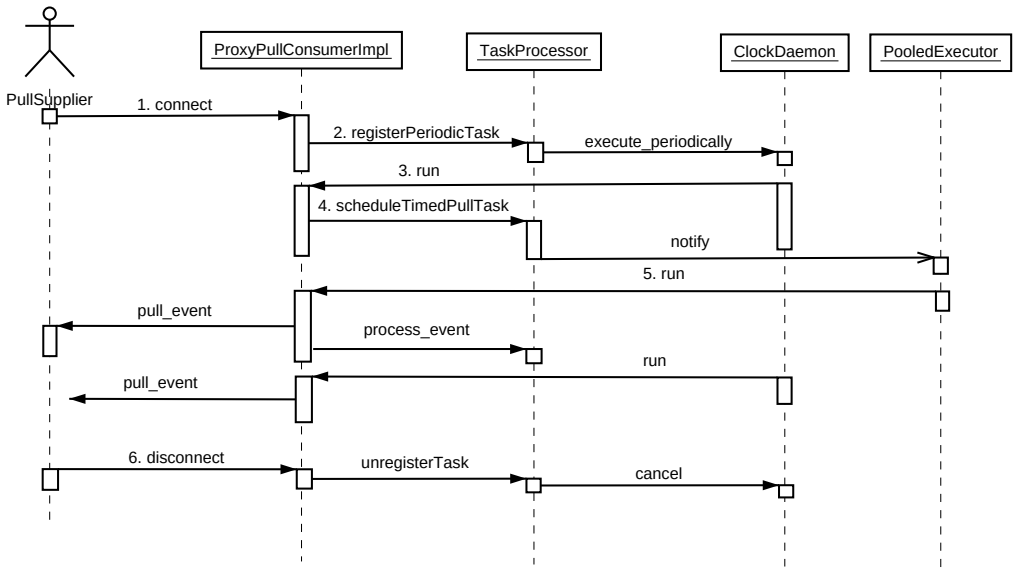

Abbildung 5.9.: Scheduling einer Pull-Operation

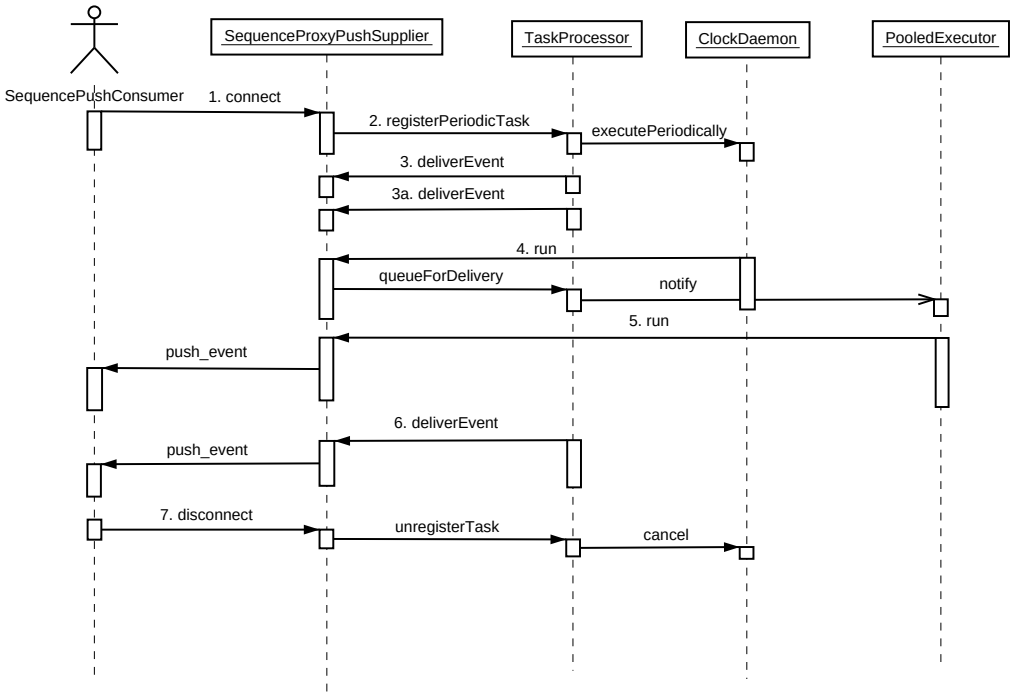

Abbildung 5.10.: Scheduling einer Push-Operation

6. Bewertung

Um eine Bewertung der Implementierung abgeben zu können, wird der JacORB Notification Service in diesem Kapitel zwei anderen Implementierungen gegenübergestellt.

Zunächst folgt ein kurzer Abschnitt über die Konformität des JacORB Notification Service in Bezug auf die OMG Spezifikation.

6.1. Konformität

Unter Konformität versteht man, dass eine Implementierung alle von der Spezifikation geforderten Leistungsmerkmale erfüllt. Für den ORB-Kern existiert die CORVAL2 Testsuite, mit der ein ORB auf seine Standardkonformität überprüft werden kann.

Für die CORBA Services existiert Vergleichbares nicht. Ein Anwender, der an der Standardkonformität eines CORBA Service interessiert ist, muss sich daher auf die Angaben des Herstellers verlassen. Insbesondere kommerzielle Hersteller sind jedoch kaum daran interessiert, eine Aussage wie „Unser Produkt XYZ ist nicht standardkonform“ zu veröffentlichen. Daher ist der Anwender im Zweifelsfall darauf angewiesen, eigene Tests zu realisieren, um Aussagen über die Konformität treffen zu können.

Um einen groben Überblick darüber zu bieten, inwieweit der JacORB Notification Service standardkonform ist, sind in Tabelle 6.1 die in der aktuellen Version realisierten Features dargestellt. Die im Rahmen der Implementierung realisierten Tests können zum Teil dazu herangezogen werden, genauere Aussagen über die Konformität einer Implementierung zu machen. So stellte sich beispielsweise bei den weiter unten beschriebenen Performanzmessungen heraus, dass die OpenORB-Filterobjekte die Methode *destroy* nicht implementieren. Ein Aufruf der Methode löst eine `NO_IMPLEMENT`-Exception aus.

Die Tests wurden nicht mit dem speziellen Ziel entwickelt, die Konformität der Implementierung zu prüfen. Eine Erweiterung der bestehenden Tests um diese Funktionalität könnte jedoch Teil der Weiterentwicklung des JacORB Notification Service werden.

Insgesamt gesehen, muss an einigen Stellen der JacORB Implementierung noch Aufwand investiert werden, um eine vollständig standardkonforme Implementierung zu erreichen. Insbesondere werden die Admin- und QoS-Properties noch nicht korrekt behandelt. Eine konforme Implementierung muss prinzipiell alle spezifizierten Properties kennen, kann aber eine entsprechende Fehlermeldung ausgeben, falls die Implementierung das Property nicht unterstützt.

Die in der Aufgabenstellung geforderte vollständige Unterstützung der Filterfunktionalität konnte realisiert werden.

Feature	JacORB Notification Service
CORBA Any Event	Ja
Structured Event	Ja
Sequenced Event	Ja
Typed Event	Nein
Admin-Properties	Teilweise: • MaxConsumers • MaxSuppliers
QoS-Properties	Teilweise: • EventReliability (BestEffort) • ConnectionReliability (BestEffort) • MaximumBatchSize • PacingInterval
Filter	Ja
Mapping Filter	Nein
Filter Conststraint Language	ETCL
Push- und Pull-Modell	Ja
offer_change und *subscription_change*	Nein
Proxy, Admin und EventChannel	Ja
Event Type Repository	Nein
Timer Events	Nein

Tabelle 6.1.: Übersicht der im JacORB Notifcation Service realisierten Features

6.2. Performanzbetrachtung

Für viele Anwendungsgebiete existieren Programme zum Leistungsvergleich von DV Systemen (Benchmark). So kann mit ECPerf beispielsweise die Performanz eines J2EE-Server gemessen werden. Manche Benchmarks sind standardisiert. Andere werden von einzelnen Herstellern in Zusammenhang mit ihren Produkten vertrieben. Gemeinsam haben alle Benchmarks, dass sie das zu testende System auf definierbare, reproduzierbare Art und Weise belasten und dabei

Messdaten erfassen. Aus diesen Messdaten wird dann versucht, die tatsächliche Performanz in einem realen Anwendungsfall zu antizipieren.

Die Realisierung einer Benchmarksuite ist eine komplexe Aufgabe. Entsprechende Programme zur Messung der CPU-Performanz werden beständig weiterentwickelt. Im Kontext von Benachrichtigungssystemen wäre eine Benchmarksuite wünschenswert, mit der sich Systeme unterschiedlicher Standards miteinander vergleichen lassen, so z. B. JMS mit CORBA Notification Service [14].

Der Vorteil standardisierter Benchmarks ist, dass sie meist eine breite Zustimmung seitens Entwicklern und Anwendern erfahren. Im Gegensatz dazu lassen sich die Messdaten eines auf ein bestimmtes System optimierten Benchmarks nur schwierig mit anders ermittelten Daten vergleichen.

Standardisierte Benchmarks kommunizieren mit dem zu testenden System lediglich über eine standardisierte Schnittstelle. Ein Benchmark kann in diesem Zusammenhang die Validität des Tests garantieren.

Für den CORBA Notification Service selbst existiert bisher kein standardisierter Benchmark. Prismtechnologies bieten im Zusammenhang mit ihrer kommerziellen Implementierung des Notification Service ein „Evaluation Toolkit“ an, das Administratoren, Entwicklern und Anwendern eine Handhabe bietet, die Performanz des OpenFusion Notification Service zu evaluieren.

Typische Messdaten, die von einem Benchmark für die Applikationsklasse der Benachrichtigungssysteme erfasst werden, sind die für die Zustellung einer Nachricht benötigte Latenzzeit, der maximale Durchsatz in Abhängigkeit der Anzahl der Clients und der Laufzeitspeicherbedarf. Im Rahmen dieser Arbeit wurden Testszenarien realisiert, die die Latenzzeit und den maximalen Durchsatz erfassten. Da die ersten Tests bereits frühzeitig realisiert wurden, konnten bereits während der Implementierung performanzkritische Codeabschnitte identifiziert und optimiert werden.

6.2.1. Andere Implementierungen

In Kapitel 5 wurde bereits eine Übersicht anderer verfügbarer Implementierungen des CORBA Notification Service gegeben. Aus diesen wurden zwei Implementierungen für den direkten Vergleich mit dem JacORB Notification Service ausgewählt. Als Vertreter einer kommerziellen Implementierung wurde der OpenFusion Notification Service von Prismtechnologies ausgewählt. Der frei verfügbare OpenORB bildete den Kandidaten aus der OpenSource-Fraktion.

OpenFusion

Prismtechnologies bietet eine kommerzielle Implementierung des CORBA Notification Service, den OpenFusion Notification Service, und darüber hinaus auch Implementierungen des Naming Service, des Trading Service und weiterer CORBA Services.

Die OpenFusion CORBA Services unterstützen nach eigener Aussage alle marktführenden ORBs, darunter auch den JacORB ORB.

Der OpenFusion Notification Service soll als Beispiel einer kommerziellen Implementierung mit dem JacORB Notification Service verglichen werden. Prismtechnologies bietet zu Evaluationszwecken eine 30-tägige Testlizenz, die hierfür verwendet wurde.

OpenFusion offeriert einen umfangreichen Lieferumfang. Besonders interessant ist ein graphisches Konfigurationswerkzeug, das entwickelt wurde, um Services zu konfigurieren, zu starten und zu stoppen. Das Werkzeug schafft einen eindrucksvollen Überblick über die vielfältigen Einstellmöglichkeiten des OpenFusion Notification Service.

Der OpenFusion Notification Service bietet nach eigener Aussage eine vollständige Implementierung der in der OMG Spezifikation beschriebenen Einstellmöglichkeiten. Zusätzlich existieren zahlreiche herstellerspezifische Erweiterungen, um den Notification Service ideal an ein bestimmtes Einsatzgebiet anpassen zu können.

Der OpenFusion Notification Service bietet bspw. zusätzliche Dienstgüteeinstellungen und die Möglichkeit, Filterausdrücke in SQL-92 zu formulieren. Zusätzlich können benutzerdefinierte, in Java geschriebene Filter eingebunden werden.

OpenORB

OpenORB ist ein frei verfügbarer ORB, der unter anderem eine Implementierung des Notification Service unter einer OpenSource-Lizenz zur Verfügung stellt.

Die OpenORB-Implementierung realisiert nicht die komplette OMG Spezifikation. Im Vergleich mit der JacORB Implementierung steht jedoch wesentlich mehr Funktionalität zur Verfügung, es werden insbesondere persistente EventChannels unterstützt. TypedEventChannels werden nicht unterstützt.

Erwartungsgemäß ist wesentlich weniger Dokumentation erhältlich als für OpenFusion. Auch die Konfiguration beschränkt sich auf eine XML Datei, in der einige Einstellungen getätigt werden können.

6.2.2. Testaufbau

Der Notification Service wird auf einem Rechner, im Folgenden *Server* genannt, gestartet (vgl. Abbildung 6.1 auf der nächsten Seite). Ein zweiter Rechner, im Folgenden *Tester* genannt, ist mit dem Server über ein 100MBit Netzwerk verbunden. Auf dem Tester werden Consumer und Supplier gestartet, die mit dem Event Channel auf dem Server kommunizieren.

Auf dem Server wurden der OpenORB Notification Service v1.3.0, der OpenFusion Notification Service v3.0 und der JacORB Notification Service installiert. OpenORB und OpenFusion wurden jeweils in der Standardkonfiguration betrieben.

6.2.3. Vorüberlegungen zur Performanz des Notification Service

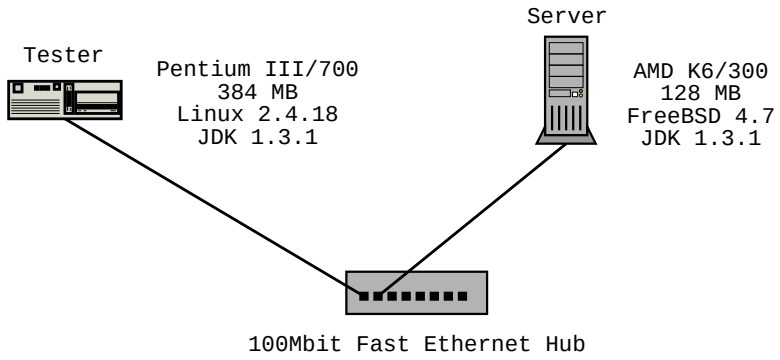

Abbildung 6.1.: Rechnerkonfiguration des Testszenarios

Ein direkter, synchroner Methodenaufruf zwischen einem Supplier und Consumer verursacht einen RPC-Aufruf mit den damit verbundenen Kosten. Der zusätzliche Aufwand, der durch Einsatz des Notification Service ins Spiel kommt, ist ein zusätzlicher RPC-Aufruf. Im Extremfall, wenn nur ein Supplier und Consumer aktiv sind, werden für jede Nachricht zwei RPC-Aufrufe ausgeführt werden, doppelt so viele wie bei Verwendung von direkten, synchronen Methodenaufrufen.

Bei einer größeren Anzahl von Consumern ändert sich dieses Verhältnis. Sind beispielsweise 50 Consumer aktiv, werden für jede Nachricht 51 RPC-Aufrufe durchgeführt, nämlich ein Aufruf, um die Nachricht dem EventChannel zuzustellen, und 50 Aufrufe, um die Nachricht an die Consumer weiterzuleiten. Bei synchronem RPC fallen dagegen 50 Aufrufe an. Der durch den Einsatz des Notification Service verursachte zusätzliche Aufwand kann also mit steigender Zahl der Consumer vernachlässigt werden.

Im praktischen Betrieb ergeben sich zusätzliche Faktoren, die die Performanz beeinflussen:

- Der Versand großer, komplexer Nachrichten hat einen hohen Aufwand für das (De-)Marshalling zur Folge.
- Normalerweise kann ein ORB Optimierungen treffen, falls Consumer und Supplier sich auf dem gleichen Rechner befinden. In diesem Fall muss der Aufruf nicht durch das Marshalling bearbeitet werden. Beim Einsatz eines Notification Service, der sich auf einem anderen Rechner befindet, sind solche Optimierungen nicht möglich.
- Die Bearbeitung persistenter Nachrichten erzeugt zusätzlichen Aufwand, da die Nachrichten zeitaufwendig in eine Datenbank oder das Filesystem geschrieben werden müssen.
- Der Einsatz von Filtern erzeugt zusätzlichen Aufwand. Jede Nachricht muss zusätzlich gemarshalled und vom Filter bearbeitet werden.
- Wenn sich der Notification Service auf einem entfernten Rechner befindet,

kann die Performanz auch durch die Leistungsfähigkeit des Netzwerkes beeinflusst werden.

Bei der Realisierung einer Applikation, die den Notification Service verwendet, können daher verschiedene Punkte beachtet werden, um eine gute Performanz zu erreichen:

- Wenn möglich, sollten sich Clients mit dem Notification Service auf einem Rechner befinden, um Aufrufe über das Netzwerk zu vermeiden.
- Prinzipiell können sich die Filter, die von einem EventChannel verwendet werden, auf einem anderen Rechner befinden. Dies ist zu vermeiden.
- Verwendung von Sequence Consumern und Suppliern. Diese Clients verwenden *Sequenzen von StructuredEvent*. Bei Verwendung dieser Clients werden größere, aber weitaus weniger Netzwerkpakete erzeugt.
- Persistente Nachrichten sollten nur eingesetzt werden, wenn dies absolut notwendig ist. Persistenzmechanismen sind aufwendig und stellen prinzipiell eine erhebliche Zusatzbelastung für jedes System dar.

6.2.4. Testszenarien

Latenzzeiten ohne Filter

Im ersten Szenario wird die durch den Einsatz des EventChannel verursachte Latenzzeit gemessen. Der Tester startet einen Supplier, der pro Testdurchlauf 100 Nachrichten an den EventChannel sendet. Zwischen zwei Nachrichten wartet der Supplier jeweils 250ms. Als Nutzlast enthält die versandte Nachricht lediglich die aktuelle Systemzeit (ermittelt durch Aufruf der Methode *System.currentTimeMillis*). Zugleich wird eine bestimmte Anzahl von Consumerinstanzen erzeugt. Die Aufgabe der Consumer besteht darin, die Nachricht entgegenzunehmen und die aktuelle Systemzeit mit der Absendezeit der Nachricht zu vergleichen. Die Differenz der beiden Zeiten bildet die Latenzzeit. Als Messergebnisse werden festgehalten:

- die minimale Latenzzeit,
- die maximale Latenzzeit,
- die mittlere Latenzzeit.

Die Consumer und Supplier werden innerhalb eines Prozesses gestartet. Dieser Prozess ist mit den Consumerinstanzen synchronisiert und blockiert, bis alle Consumer die Nachricht empfangen haben bzw. in einen einstellbaren Timeout laufen.

Zusätzlich wird die gesamte Bearbeitungszeit gemessen. Das Zeitintervall beginnt mit dem Start des ersten Supplier und endet, wenn alle Consumer alle Nachrichten empfangen haben.

Der Test zeigt den Aufwand, der durch den Einsatz des EventChannel verursacht wird. Idealerweise ist die Latenzzeit minimal und wird von der Anzahl der Supplier, der Consumer und der versandten Events nicht beeinflusst. Abhängig von den Ressourcen des Servers und der vom Notification Service implementierten Strategie kann nur eine bestimmte Anzahl von Consumern nebenläufig beliefert werden. Daher ist zu erwarten, dass die Latenzzeiten im Mittel ansteigen. Ab einem gewissen Punkt ist ein rapider Performanzeinbruch zu erwarten. Dieser Punkt tritt ein, wenn die Nachrichten schneller im EventChannel eintreffen als sie ausgeliefert werden können.

Von jedem Supplier werden jeweils 100 Events versandt. In unterschiedlichen Testläufen sind 1, 10 und 50 Consumer aktiv. Zusätzlich wird die Zahl der Supplier in den Schritten 1, 5 und 10 variiert.

Im Vergleich mit OpenORB zeigte sich, dass JacORB stets geringere Latenzzeiten erzielte (vgl. Tabellen A.2 und A.3). Auch die mittleren Bearbeitungszeiten lagen stets unter denen von OpenORB. Bei einer großen Anzahl von Consumern und Suppliern brachen die Messwerte jedoch bei beiden Testlingen ein. Mit 50 verbundenen Consumern zeigten OpenORB und JacORB Werte, die sich von denen bei OpenFusion gemessenen Werten um Größenordnungen unterschieden (OpenORB ca. Faktor 500, JacORB ca. Faktor 80). Im Gegensatz zu OpenORB zeigte JacORB mit 10 verbundenen Consumern noch gute Werte. Diese Werte sind insbesondere den Werten von OpenFusion vergleichbar (vgl. Tabelle A.1 in Anhang A).

Latenzzeiten mit Filtern

Im nächsten Szenario wird pro ProxySupplier ein Filter hinzugefügt. Die Filter werden mit einem einfachen Filterausdruck auf eine Weise konfiguriert, dass jeder Consumer von jedem Supplier exakt eine Nachricht erhält. Folglich werden die restlichen Nachrichten vom Event Channel verworfen.

Dieser Test zeigte, inwieweit der Einsatz von Filtern die Latenzzeit des Event Channel erhöht. Im Test versendete jeder Supplier 100 Events. In den unterschiedlichen Testreihen waren 1, 10 und 50 Consumer aktiv.

Im direkten Vergleich mit OpenORB und OpenFusion zeigte dieses Testverfahren schlechtere Werte (Tabellen A.4, A.5 und A.6). Dabei war auffallend, dass sich OpenORB in diesem Test besser präsentieren konnte, die Messwerte übertrafen sogar die Werte von OpenFusion.

Zeitaufwand für das Filtern

Auf der Suche für die Ursache des schlechteren Testergebnisses aus dem vorangegangenem Abschnitt wurde die Filterperformanz in einem weiteren Testszenario gesondert betrachtet.

Testling in diesem Testszenario ist lediglich ein einzelner Filter. Das Testobjekt wird angelegt und mit einem Filterausdruck konfiguriert. Danach wird 100 mal die *match*-Operation aufgerufen. Parameter der Operation sind *Any*- und

StructuredEvent-Werte. Filter und Nachrichten sind so gewählt, dass die Filter stets erfolgreich ausgewertet werden.

In mehreren Durchläufen werden komplexere Filterausdrücke verwendet. Die Filterausdrücke enthalten dabei teils redundante Teilausdrücke, um zu prüfen, ob eine Optimierung auf Seiten des Filters stattfindet.

Es werden folgende Filterausdrücke verwendet:

I. `$ == 10` (Any)

II. `$event_name == 'TESTING'` (StructuredEvent)

III. `$type_name == 'TESTING'` (StructuredEvent)

IV. `$.phone_numbers[0] == '12345678'` (Any)

V. `exist $.phone_nubers[0] and $.phone_numbers[0] == '12345678'` (Any)

VI.
```
exist $.phone_numbers[0] and exist $.phone_nubers[0]
and $.phone_numbers[0] == '12345678'
```
(Any)

VII.
```
exist $.phone_numbers[0] and exist $.phone_numbers[0]
and exist $.phone_numbers[0]
and $.phone_numbers[0] == '12345678'
```
(Any)

Zugegebenermaßen werden manche der angegebenen Filterausdrücke in der Praxis kaum eingesetzt werden. Sie dienen jedoch beispielhaft dafür, Optimierungen bei der Filterevaluierung aufzuzeigen.

Der Test bestätigte, dass die Filterevaluierung der OpenORB-Implementierung prinzipiell schneller ist (vgl. IV in Abb. 6.2 auf der nächsten Seite). Er zeigte jedoch auch, dass bei OpenORB offensichtlich keine Optimierungen innerhalb des Filters stattfinden. Die Filterausdrücke IV, V, VI und VII enthalten den redundanten Teilausdruck `$.phone_numbers[0]`. Wie in Abschnitt 4.2.5 beschrieben, werden gleiche Teilausdrücke in der JacORB Implementierung nur einmal evaluiert. Dieser Effekt spiegelt sich in den Messdaten von JacORB (vgl. Tabelle A.9). Die Messwerte für die Filterausdrücke V, VI und VII sind gleich. Der Hauptaufwand besteht darin, den Ausdruck das erste Mal auszuwerten.

Die Messdaten für OpenORB zeigen ein anderes Bild (Tabelle A.8). Der initiale Aufwand für Filterausdruck IV ist geringer als beim gleichen Filter für JacORB. Es findet jedoch keine Optimierung statt, so dass der Aufwand für die Filter V bis VII weiter ansteigt.

Überraschenderweise zeigte sich dieses Verhalten in verstärkter Form bei den Messwerten für OpenFusion (Tabelle A.7). Die Evaluierung der komplexen Filterausdrücke dauerte dort zum Teil doppelt so lange wie bei OpenORB.

Bei einfachen Filterausdrücken ist die Performanz von OpenORB besser. Bei komplexen Filterausdrücken mit gleichen Teilausdrücken werden die Optimierungen der JacORB Implementierung relevant. Daher werden komplexere Filterausdrücke von JacORB schneller als von OpenORB bearbeitet.

Eine Laufzeitanalyse der Filterperformanz führte zur Implementierung einer neuen Datenstruktur *CachingWildcardMap*. Es fiel auf, dass der Code für das

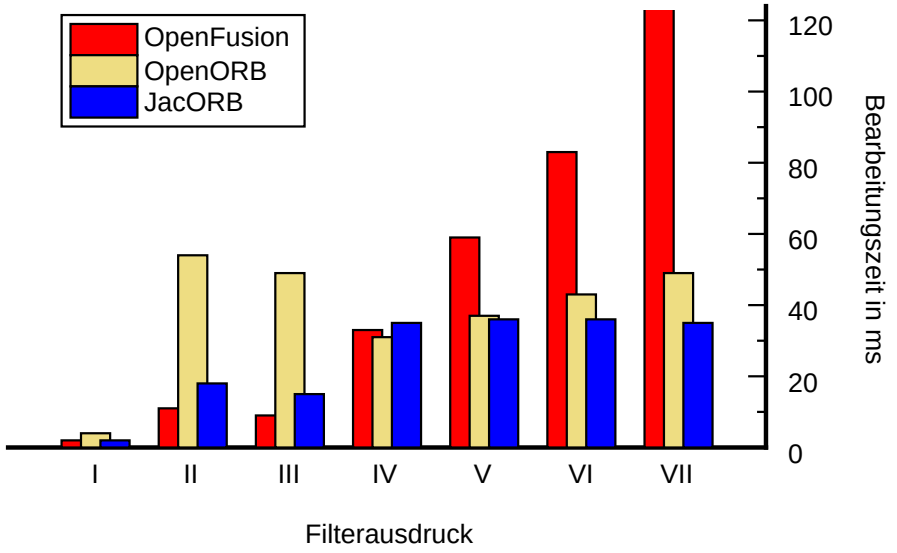

Abbildung 6.2.: Laufzeiten für unterschiedliche Filterausdrücke

Patternmatching, der bei der Nutzung der Klasse *WildcardMap* durchlaufen wird, relativ teuer ist, aber teils vermieden werden kann.

Es wurde die Klasse *CachingWildcardMap* realisiert. Diese Klasse erweitert die Funktionalität der Klasse *WildcardMap* um einen einfachen Cachingmechanismus. Der Einsatz bietet sich insbesondere an, wenn nur wenige unterschiedliche Schlüssel verwendet werden bzw. Einfüge- und Löschoperationen nur selten stattfinden. Die Performanz der Datenstruktur wird dadurch in diesem Szenario wesentlich erhöht.

Die Verwendung der Datenstruktur stellt jedoch nur einen Teil der Filterevaluierung dar, der Hauptaufwand fällt an anderen Stellen an. Daher konnte die Gesamtfilterperformanz nur um ca. 10% erhöht werden.

Eine Nachricht vom Typ *StructuredEvent* enthält die Attribute *domain_name*, *event_type* und *event_name*. Die Attribute sind relativ tief innerhalb der Nachricht verschachtelt. Dem Attribut domain_name entspricht beispielsweise der voll qualifizierte Name *$.header.fixed_header.event_type.domain_name*. Die Attribute bilden die minimale Attributmenge für ein StructuredEvent. Es ist daher anzunehmen, dass ein Großteil der Filterausdrücke sich lediglich auf diese drei Attribute beziehen wird.

Die Filter-Schnittstelle ist entsprechend ausgelegt und erlaubt es, Paare von *event_type*/*domain_name* den zu evaluierenden ETCL Ausdrücken zuzuordnen. Der einfachste FilterConstraint ist aus einem *event_type*/*domain_name*-Paar und einem leerem ETCL Ausdruck zusammengesetzt. Der leere Ausdruck wird dabei implizit als *true* behandelt.

Innerhalb von ETCL Ausdrücken existiert die sog. *Shorthand*-Notation, die erlaubt anstelle voll qualifizierter Namen die Laufzeitvariablen *$domain_name*,

$event_type und *$event_name* zu verwenden.

Die erste Implementierung des Filters expandierte die Laufzeitvariable zu dem entsprechenden voll qualifizierten Pfad und verwendete dann das DynAny API, um die Nachricht zu traversieren und den entsprechenden Wert zu extrahieren.

Wie bereits erwähnt, ist die Verwendung des DynAny API teuer. Dementsprechend ergaben sich für die Filterausdrücke II und III im Vergleich zu OpenORB und OpenFusion schlechtere Werte (OpenFusion: ca. 10ms, OpenORB: ca. 50ms, JacORB: ca. 80ms).

Die sehr guten Werte von OpenFusion in diesem speziellen Szenario legten nahe, dass es sich um einen typischen Anwendungsfall handelt, der entsprechend optimiert werden sollte. Durch eine Optimierung konnte die mittlere Bearbeitungszeit von ca. 80ms auf ca. 16ms verringert werden (In Abb. 6.2 auf der vorherigen Seite dargestellt). Dies übertrifft die Werte von OpenORB bei weitem und nähert sich wesentlich den Werten von OpenFusion an.

Maximaler Durchsatz

Im nächsten Testszenario wurde der maximale Durchsatz gemessen. Hierzu wird ein Consumer und ein Supplier verwendet. Der Supplier sendet mit maximaler Geschwindigkeit einen Schub von Events an den EventChannel. Der Supplier misst dabei die Zeit, die dafür benötigt wird. Daraus lässt sich ermitteln, wie viele Nachrichten der EventChannel pro Sekunde entgegennehmen kann.

Parallel wird erfasst, wie viele Nachrichten der EventChannel pro Sekunde ausliefern kann. Der Consumer misst die Zeit, die zwischen Eintreffen der ersten und der letzten Nachricht verstreicht. Daraus wird der Durchsatz berechnet.

Um einen Referenzwert zur Verfügung zu haben, wurde zunächst der RPC-Durchsatz gemessen. Dazu wurde ein einfacher CORBA-Server realisiert, der eine Methode mit einem *Any*-Parameter zur Verfügung stellt. Der übergebene Parameter wird vom Servercode ignoriert.

Vom Tester aus wurde diese Methode aufgerufen. Die gemessenen Werte sollen Rückschluss auf den maximalen Durchsatz geben, der durch ORB, JDK, Betriebssystem CPU und Netzwerk beschränkt wird. Es ist zu beachten, dass der Durchsatz in jeder praktischen Anwendung niedriger liegen würde, da der Testserver im Gegensatz zu einer konkreten Anwendung keinerlei Programmlogik enthält.

In den nachfolgenden Tests wurden Nachrichten an den EventChannel gesendet. Idealerweise liegt der Durchsatz möglichst nahe der Hälfte des Referenzwertes und wird nicht durch die Anzahl der in einem Schub versendeten Events beeinflusst. Die Hälfte des Referenzwertes wird angenommen, da bei der Übertragung einer Nachricht vom Supplier zum Consumer zwei RPC-Aufrufe stattfinden. Im Gegensatz dazu findet beim Testserver nur ein RPC-Aufruf statt.

In diesem Szenario wurden je Schub 10, 100, 1000, 5000 10000 und 20000 Events versandt. Aus der Gesamtzeit lässt sich der Durchsatz in Events pro Sekunde berechnen. Die Ergebnisse finden sich in den Tabellen A.10, A.11

und A.12 in Anhang A. Die Messwerte sind zusätzlich in Abbildung 6.3 graphisch aufgetragen.

Es zeigte sich, dass JacORB im unteren Bereich durchaus mit OpenFusion zu vergleichen ist. Im oberen Bereich >10K nimmt jedoch der Durchsatz von OpenORB und JacORB rapide ab. OpenFusion zeigt hier immer noch konstante Messwerte.

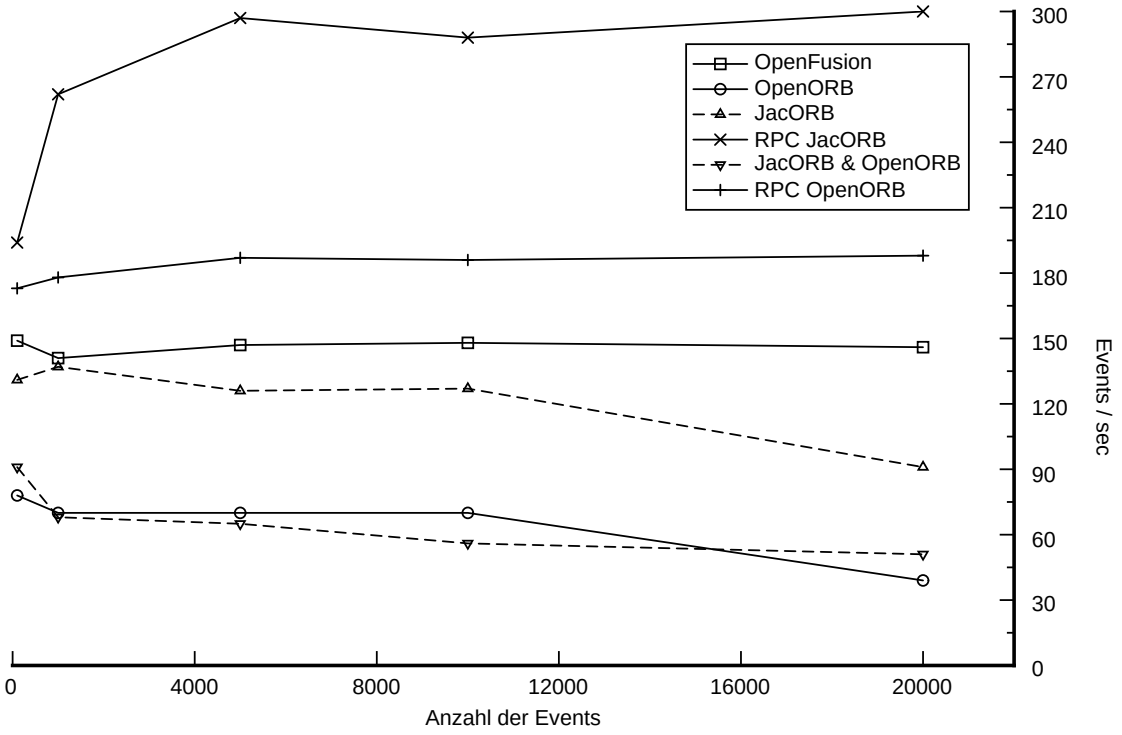

Abbildung 6.3.: Durchsatz

Unterhalb des OpenFusion und des JacORB Notification Service kommt der JacORB ORB zum Einsatz. Der ORB hat sicherlich einen nicht unerheblichen Einfluss auf die Gesamtperformanz eines Notification Service. Folglich wurde versucht, den OpenORB Notification Service ebenfalls auf dem JacORB ORB zu betreiben, um den Einfluss des ORB auf die Messergebnisse auszuschalten. Dies scheiterte leider, da die Implementierung des OpenORB Notification Service offenbar fest an den OpenORB ORB gekoppelt ist. Daher wurde der JacORB Notification Service in Zusammenhang mit dem OpenORB ORB betrieben (vgl. Tabelle A.13). Die gemessene Performanz ist weitgehend den Werten des OpenORB Notification Service vergleichbar. Die ORB Implementierung hat folglich einen erheblichen Einfluss auf die Performanz. Daher wurde der RPC-Durchsatz des OpenORB ebenfalls mit Hilfe des *EchoServer* gemessen. Dabei wurden nur ca. 60-70% der JacORB Werte erreicht (vgl. Abb. 6.3).

In einer weiteren Messung des Durchsatzes wurde die Größe der Nutzlast der versendeten Nachrichten variiert. Es wurden jeweils 100 Nachrichten ver-

sandt. Die versandten Nachrichten enthielten *int*-Arrays der Größen 1, 32, 64, 128, 256, 512 und 1024. Entsprechend ergibt sich die Größe der Nutzlast als 32, 1024, 2048, 4096, 8192, 16384 und 32768 byte. Um einen Vergleichswert zur Verfügung zu haben, wurden wiederum Messwerte mit dem beschriebenen *EchoServer* erstellt.

Die Ergebnisse finden sich in den Tabellen A.14, A.15 und A.16 in Anhang A. Eine graphische Darstellung befindet sich in Abbildung 6.4. Wie zu erkennen, sinkt der RPC-Durchsatz mit steigender Nachrichtengröße. Der Durchsatz der drei verglichenen Notification Service=Ïmplementierungen sinkt in vergleichbarer Weise. Die Übertragung über das Netzwerk stellt hier also einen wichtigen Faktor dar. Entscheidend ist jedoch, dass die Implementierung eines Notification Service keine zusätzlichen Performanzeinbussen durch große Nachrichten erfährt. Dies trifft offenbar auf alle Implementierungen zu.

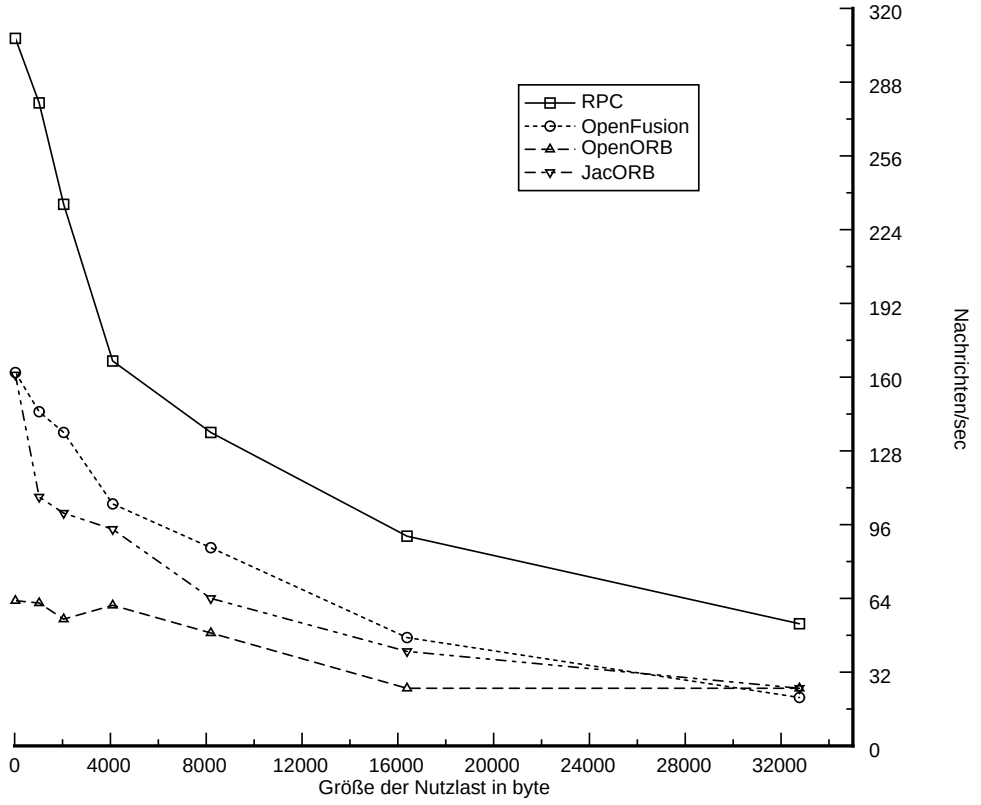

Abbildung 6.4.: Durchsatz in Abhängigkeit von der Größe der Nutzlast

6.3. Zusammenfassung

Bei einer kleinen und mittleren Anzahl von Clients und Nachrichten zeigt der JacORB Notification Service eine gutes Laufzeitverhalten, das auch dem Ver-

gleich mit der kommerziellen Implementierung von Prismtechnologies standhalten kann.

Unter hoher Last zeigt sich jedoch ein Performanzeinbruch. OpenFusion bietet noch in diesem Bereich ein ausgezeichnetes Laufzeitverhalten. Auffallend war der sehr hohe Speicherbedarf des OpenFusion Notification Service. Außerdem scheinen beim OpenFusion Notification Service ständig Persistenzmechanismen aktiv zu sein, auch wenn ein nicht-persistenter EventChannel verwendet wird. Der hohe Speicherbedarf führte gelegentlich zu OutOfMemoryExceptions. Es ist davon auszugehen, dass die Performanz von OpenFusion bei geigneten Optimierungen noch höher liegen kann als in den Tests gemessen, da prinzipiell zahlreiche Einstellmöglichkeiten existieren, die in keinster Weise ausgeschöpft wurden. Auch bei OpenORB existieren einige Möglichkeiten, den Notification Service zu konfigurieren.

Der OpenORB Notification Service wurde in allen Disziplinen abgeschlagen. Einzig die Performanz der Filter kann als sehr gut angesehen werden.

Der eingesetzte ORB hat einen wesentlichen Anteil an der Gesamtperformanz des Notification Service. Aus Zeitgründen wurde davon abgesehen den OpenORB Notication Service auf den JacORB ORB zu portieren. Daher können streng genommen nur OpenFusion/JacORB und OpenORB/JacORB miteinander verglichen werden.

Im Laufe der Messungen konnten einige Optimierungen an der Implementierung des JacORB Notification Service durchgeführt werden, die die Performanz wesentlich verbessert haben.

Die Vergleichsergebnisse mit OpenFusion waren nicht anders zu erwarten. OpenFusion ist bereits seit einiger Zeit auf dem Markt. Der Hersteller bezeichnet sein Produkt als Marktführer. Vor diesem Hintergrund können die erzielten Ergebnisse als sehr erfreulich eingestuft werden.

Vom Funktionsumfang ist der JacORB Notification Service beiden Kontrahenten unterlegen. Dies ist durch die Aufgabenstellung auch nicht anders beabsichtigt. Die performante Basisfunktionalität stellt eine gute Ausgangsbasis für eine künftige Weiterentwicklung dar.

7. Fazit

In der vorliegenden Arbeit wurde der Entwurf und die Implementierung des CORBA Notification Service beschrieben. Einleitend wurden Einsatzgebiete und Anforderungen an Benachrichtigungssysteme dargestellt. Die Beschreibung relevanter Technologien zeigte die Tragfähigkeit, aber auch die Grenzen bereits bestehender Lösungen.

Nach der Erläuterung zentraler Begriffe in Abschnitt 2.3.1, wurden die Benachrichtigungssystemen zugrundeliegenden Konzepte und Technologien erläutert. Anhand dieses Modells erfolgte in Abschnitt 3 ein Vergleich von JMS, MQSeries, SIENA und dem CORBA Event/Notification Service.

Nach diesem theoretischen Teil wurde in Kapitel 4 der Entwurf des Notification Service beschrieben. Die Problembereiche wurden identifiziert und mögliche Lösungen wurden diskutiert. Davon ausgehend, wurde die Architektur des JacORB Notification Service vorgestellt.

Die Ausführungen dieses Kapitels dienten als Grundlage für die tatsächliche Implementierung. In Kapitel 5 wurde näher auf ausgewählte Aspekte der Implementierung eingegangen.

Eine wichtige Anforderung an die Implementierung war deren Performanz. Aus diesem Grund wurde die Implementierung dem kommerziellen OpenFusion und einer OpenSource-Variante des Notification Service gegenübergestellt. In diesem Rahmen wurden einige Testszenarien entwickelt und Messdaten erfasst.

Im Laufe der Messungen zeigten sich beim Betrieb mit einer kleinen bis mittleren Anzahl von Clients gute Leistungen, die auch dem Vergleich mit dem kommerziellen OpenFusion standhalten können. Beim Betrieb mit einer größeren Anzahl von Clients bleibt OpenFusion jedoch konkurrenzlos.

Um die Arbeit abzuschließen, wird im Folgenden ein Blick in die Zukunft getan. Zum einen existieren neue, den Notification Service betreffende OMG Spezifikationen. Zum anderen soll das Protokoll SOAP betrachtet werden, das dem Anschein nach verteilte Systeme revolutionieren soll. Zusätzlich wird auf die mögliche Weiterentwicklung des JacORB Notification Service eingegangen. Auf einige aktuelle Entwicklungen wird näher eingegangen, in deren Zusammenhang der Autor den Notification Service im Speziellen und Benachrichtigungssysteme im Allgemeinen sieht.

7.1. Ausblick

In verteilten, nachrichtenbasierten Systemen spielen passive oder zusammengesetzte Ereignisse eine wichtige Rolle [43]. Dementsprechend bieten Systeme

wie SIENA die Möglichkeit, die zeitliche Abfolge bestimmter Ereignisse als neues Ereignis zu definieren. Die Existenz aktueller Arbeiten in diesem Bereich weist darauf hin, dass die Verwendung zusammengesetzter Ereignisse (insbesondere deren effiziente Evaluierung) ein Bereich ist, dessen Bedeutung in Zukunft zunimmt [32, 31].

Beim Entwurf und der Implementierung einer verteilten Applikation erfolgt recht früh eine Festlegung auf einen bestimmten Kommunikationsstil. Es wird festgelegt, ob Subsysteme der Applikation nachrichten- oder aufrufbasiert miteinander kommunizieren. Bei der Eigenentwicklung einer Applikation mag dies akzeptabel sein, aber in komponentenbasierten Systemen kann dies ein Hindernis darstellen. Wenn beispielsweise in einem aufrufbasierten System eine nachrichtenbasierte Komponente eingesetzt werden soll, liegt eine Inkompabilität vor. Dies trifft ebenfalls zu, wenn zwei Komponenten nachrichtenbasiert miteinander kommunizieren sollen, die eine Komponente jedoch nur den Push-Stil und die andere den Pull-Stil unterstützt.

Ein Weg zur Vermeidung dieser Inkompabilität ist, die von einer Komponente angebotenen und verwendeten Schnittstellen abstrakt zu deklarieren. Dadurch muss der tatsächlich verwendete Kommunikationsstil idealerweise erst beim Deployment der Komponente ausgewählt werden.

Innerhalb der Komponenten-Architektur *Corba Components* wird der Notification Service eingesetzt, um Nachrichten zwischen einzelnen Komponenten auszutauschen [26]. Im *Component Assembly Descriptor* der einzelnen Komponente wird definiert, wie *Event Sink* und *Event Source* der Komponenten miteinander zu verbinden sind [z. B. 7, S. 624]. Der Entwickler der CORBA-Komponente muss jedoch immer noch gesonderten Code für die aufruf- und die nachrichtenbasierte Kommunikation schreiben. Auch der Entwickler einer Enterprise Java Bean muss die Entscheidung treffen, ob er eine nachrichtenbasierte Message Driven Bean oder eine aufrufbasierte Session Bean realisiert.

Der Typed Event Channel kann verwendet werden, um zwischen aufruf- und nachrichtenbasierten Komponenten zu vermitteln. Um dieses Ziel zu erreichen, sind jedoch noch Anpassungen notwendig. Diese Anpassungen werden üblicherweise durch Adapter- oder Wrappercode realisiert.

Denkbar ist, die Funktionalität völlig unabhängig vom Kommunikationsstil zu realisieren. Die Kommunikationsbedürfnisse der Komponente werden in einer abstrakten Schnittstellenbeschreibung deklariert. Mit Hilfe dieser Beschreibung kann die Vermittlung anschließend automatisiert werden [44].

Benachrichtigungssysteme können auch im Zusammenhang mit *mobilen Objekten* wertvolle Dienste liefern. Immer mehr mobile Geräte sind programmierbar und können miteinander kommunizieren.

Bereits heute verfügen viele Menschen über ein Mobiltelefon, ein Laptop oder ein PDA. Die Geräte kommunizieren miteinander, um beispielsweise Termin- oder Adressdaten gemeinsam zu nutzen. Im Kontakt untereinander sollen die Geräte zu einer dynamischen Verbindung in der Lage sein, beispielsweise zum Austauschen einer Visitenkarte.

In Zukunft werden immer mehr Objekte der realen Welt mit Prozessoren und Kommunikationsmöglichkeit ausgestattet. Solche Objekte können heute vom

Nutzer mit einem geeigneten Endgerät abgefragt bzw. konfiguriert werden, in Zukunft aber sollen die Objekte vermehrt direkt miteinander interagieren, um höherwertige Dienste zu erbringen [22, 25].

Charakteristisch für dieses Szenario ist, dass keine zentrale Instanz für die Kontrolle und Koordinierung der Objekte existiert. Die mobilen Objekte sind lose miteinander gekoppelt, die verbinden sich selbsttätig mit dem Netzwerk, informieren sich über verfügbare Dienste, bieten alleine oder in Kooperation mit vorhandenen Diensten einen Service an und verlassen das Netzwerk wieder. Die Grenzen des Systems sind „unscharf", da häufig Objekte das Netzwerk betreten oder verlassen. Man spricht auch von einem *ad-hoc*-Netzwerk oder *Spontaneous Computing*.

In einem verteilten System stellt die Nicht-Verfügbarkeit eines Objekts normalerweise einen Fehler dar. Im Zusammenhang mit mobilen Objekten ist dies jedoch eine Standardsituation.

Eine wichtige Rolle spielt daher die Bekanntmachung von neuen Diensten und das Abmelden von bestehenden Diensten im Netzwerk. In diesem Zusammenhang kann ein Benachrichtigungssytems eingesetzt werden, um derartige Informationen netzwerkweit verfügbar zu machen [22].

7.1.1. SOAP

SOAP ist ein Schlagwort, das zur Zeit oft verwendet wird, wenn über die Kopplung verteilter Systeme gesprochen wird. SOAP ist ein offenes, XML-basiertes Protokoll, um plattformunahängig auf Dienste, Objekte und Server zuzugreifen. SOAP hat in vielen Bereichen der Softwareindustrie große Beachtung erlangt. Wie es scheint, beabsichtigen die meisten Unternehmen, in irgendeiner Art und Weise SOAP zu unterstützen.

Die SOAP-Funktionalität konkurriert in zahlreichen Bereichen mit etablierten Technologien wie CORBA, RMI und DCOM. Viele der enthusiastischen Berichte, die sich bei einer Internetrecherche finden lassen, scheinen aber eher vom Marketing getrieben zu sein. Daher fällt die Unterscheidung zwischen echtem technischem Mehrwert und Marketingaussagen nicht immer leicht.

Im folgenden wird SOAP kurz vorgestellt und etablierten Middlewaretechniken gegenübergestellt. Abschließend wird eine Einschätzung gegeben, inwieweit der Einsatz in verteilten (nachrichtenbasierten) Systemen sinnvoll erscheint.

SOAP Überblick

SOAP wurde ursprünglich von Microsoft entwickelt und ist inzwischen vom W3C standardisiert. Die SOAP-Spezifikation definiert, wie aus strukturierten und typisierten Informationen Nachrichten aufgebaut werden können. SOAP verwendet als Datenformat XML. Eine-SOAP Nachricht besteht aus einem *Envelope* (Umschlag), der einen optionalen *Header* und den *Body* enthält. Der Body hat die eigentliche Nutzlast.

Mit SOAP-Nachrichten werden one-way Übermittlungen von einem Sender an einen Empfänger realisiert. Für two-way Kommunikation kann entsprechend

eine zusätzliche Antwort dem Sender zugestellt werden.

SOAP kann mit verschiedenen Transportmechanismen kombiniert werden. Meist wird HTTP eingesetzt. Es ist aber beispielsweise auch das e-Mail Protokoll SMTP vorgesehen. SOAP über HTTP bietet sich insbesondere für two-way Kommunikation an, da HTTP dem gleichen Kommunikationsmodell folgt. Daher kann die SOAP-Anwort mit dem HTTP-Response übertragen werden. Wenn das Transportprotokoll keine Unterstützung dafür bietet (SMTP), müssen SOAP-Request und -Response in zwei voneinander unabhängigen Übertragungen transportiert werden. Im SOAP-Header werden dabei eindeutige Identifikationsnummern mitgesandt, die eine Zuordnung der Nachrichten erlauben.

Die Schnittstelle eines Dienstes, der über SOAP aufrufbar ist (ein Webservice!), kann in der Schnittstellenbeschreibungssprache *Webservice Description Language* (WSDL) erfolgen. Ein WSDL-Deskriptor enthält nutzerdefinierte Datentypen, die Definition der Nachrichtentypen und die Beschreibung der verfügbaren Operationen. Auch WSDL basiert auf XML.

Ergänzend existiert ein Dienst, der von Anbietern von Webservices für die Veröffentlichung von Informationen genutzt werden kann. Nutzer können diese Informationen verwenden, um einen passenden Webservice zu suchen. Dieser Verzeichnisdienst wird *Universal Description, Discovery and Integration* (UDDI) genannt.

Damit sind prinzipiell mit SOAP sowohl entfernte synchrone und asynchrone Methodenaufrufe (RPC) als auch nachrichtenbasierte Kommunikation möglich. Im Vergleich mit Middlewaresystemen wie CORBA oder RMI entspricht SOAP dem Transportprotokoll. Ein Überblick wird in Tabelle 7.1.1 auf der nächsten Seite gegeben.

Der größte konzeptuelle Unterschied zwischen SOAP, CORBA und RMI ist die Abwesenheit eines Objektmodells. Demnach können SOAP-Aufrufe nur Parameter enthalten, die per Wert kopiert werden (call-by-value). Die Verwendung von Referenzparametern (call-by-reference) ist nicht möglich.

Üblicherweise ist das Wireprotokoll nur ein Implementierungsdetail eines verteilten Systems. SOAP stellt das Wireprotokoll in den Mittelpunkt der Spezifikation. Der Fokus auf die Protokollebene hat bei SOAP im Gegensatz zu CORBA und RMI ein relativ niedriges Abstraktionsniveau zur Folge.

Durch den Einsatz von XML ist das Wireprotokoll auch für Menschen relativ leicht lesbar. Andererseits ergeben sich dadurch erhebliche Nachteile für das Laufzeitverhalten, da XML zeitaufwendig geparset werden muss.

Auch WSDL als Schnittstellenbeschreibungssprache hat einige Schwächen. WSDL ist erheblich komplexer als beispielsweise CORBA-IDL. Die WSDL-Beschreibung eines Dienstes enthält wesentlich mehr, die Kommunikation betreffende Details, als ein entsprechendes IDL Pendant. Die meisten der Informationen sind dabei für den Entwickler irrelevant.

Darüber hinaus existiert keine standardisierte Anbindung von SOAP an die typischen Programmiersprachen. Dies schränkt die Portabilität von Applikationen, die SOAP verwenden, erheblich ein.

Zusammengefasst scheinen SOAP bzw. Webservices für die Anwendung RPC zur Zeit nur bedingt geeignet zu sein. Mangels entsprechender Spezifikatio-

	Java RMI	CORBA	SOAP
Objektmodell	Ja	Ja	Nein
Wireprotokoll	JRMP, IIOP	IIOP	SOAP Envelope
Datenkodierung	Java Serialisierung	Common Data Representation (CDR)	SOAP Encoding
Transportprotokoll	TCP/IP	TCP/IP	i.d.R: HTTP
Schnittstellenbeschreibung	Java	CORBA IDL	WSDL
Programmiermodell	Java	standardisierte Languagemappings	—
standardisierte Dienste	z. B. JMS	CORBA Services: z. B. Notification Service	UDDI

Tabelle 7.1.: SOAP im Vergleich mit CORBA und RMI

nen existieren herstellerspezifische Erweiterungen, die die Interoperabilität und Portierbarkeit von auf SOAP basierenden Applikationen erschweren [vgl. 15].

SOAP in verteilten Systemen

Die Beschränkung auf die Definition eines Übertragungsprotokolls führt bei SOAP zu einem netzwerkzentrierten Programmiermodell. Alles, bis auf die Kommunikationsdetails, bleibt dem Anwendungskontext überlassen. Im Gegensatz dazu bieten CORBA und RMI ein bibliothekszentriertes Programmiermodell. Ein derartiges Modell bietet eine Programmier-Schnittstelle, die von möglichst vielen Details der Kommunikation abstrahiert.

Der netzwerkzentrierte Ansatz kann Vorteile haben [16]. Ein Protokoll wie SOAP macht keine Annahmen über seine Nutzung und kann daher flexibel eingesetzt werden. Ein Protokoll wie IIOP ist hingegen eng an einen bestimmten Einsatzbereich gekoppelt.

Das WWW zeigt, dass ein einfaches, netzwerkzentriertes Programmiermodell in heterogenen Umgebungen wie dem Internet Vorteile haben kann. Die Beschränkung auf ein einfaches Protokoll lässt beim Einsatz von HTTP die Freiheit, auf Basis eines netzwerkzentrierten Programmiermodells eigene bibliothekszentrierte Programmierschnittstellen zu definieren.

Vor diesem Hintergrund kann SOAP als einfaches Protokoll zum Austausch strukturierter Informationen betrachtet werden. Es wird eine Art „maschinen-

lesbares Internet“ ermöglicht, indem eine einheitliche und einfache Sprache verwendet wird. Das bekannteste Beispiel dafür ist die Suchmaschine Google, die es ermöglicht, Suchanfragen über SOAP zu realisieren. Die Suchfunktionalität kann so leicht aus einer Applikation heraus angesprochen werden. Für solche wenig komplexen, lose gekoppelten Systeme würde eine entsprechende Funktionalität, mit CORBA oder RMI realisiert, eine zu hohe Einstiegshürde bedeuten.

Um eine echte Interoperabilität zwischen verschiedenen Systemen zu erreichen, ist die Standardisierung der Nachrichtenformate notwendig. Dies kann am Beispiel von Google veranschaulicht werden. Angenommen, weitere Suchmaschinen böten an, ihre Suchfunktionalität über SOAP anzusprechen. Es wäre dann entscheidend, welche Schnittstelle sie anbieten. Sie könnten die von Google definierte Variante realisieren oder eine eigene Schnittstelle definieren, wobei die zweite Variante wahrscheinlicher ist. Erst die Standardisierung einer entsprechenden Schnittstelle könnte diesem Wildwuchs Einhalt gebieten.

Ein großer Vorteil von SOAP ist die breite Unterstützung in der Softwareindustrie. Es bleibt jedoch abzuwarten, inwieweit die notwendige Standardisierung von Webservices stattfindet, was eine zentrale Voraussetzung ist, um tatsächlich Service-Angebote miteinander kombinieren zu können.

SOAP wird daher Technologien wie CORBA, RMI und DCOM nicht ersetzen. Stattdessen ist zu erwarten, das SOAP als Nachrichtenformat für die Kommunikation zwischen etwa einem CORBA basierten System und einem DCOM oder .Net basierten System eingesetzt wird.

Dementsprechend bieten einige Hersteller von ORB-Produkten eine SOAP-Unterstützung an [52]. Andere Hersteller haben entsprechende Funktionalität geplant. Es existiert auch eine OpenSource-Implementierung [59].

7.1.2. Relevante Spezifikationen der OMG

Im engen Zusammenhang mit SOAP steht das Dokument *CORBA to WSDL/-SOAP Interworking Specification*. Dies ist eine von der OMG erstellte Spezifikation, die die Interoperabilität zwischen CORBA und WSDL/SOAP standardisieren soll [27]. Hauptbestandteil der Spezifikation ist eine Abbildung zwischen WSDL und IDL.

Weiterhin kann auch das Dokument *XMLDOM: DOM/Value Mapping Specification* in engem Zusammenhang mit SOAP gesehen werden. In Anerkennung der weiten Verbreitung von XML zum Austausch von strukturierten Informationen wird dort eine Abbildung zwischen IDL und XML definiert. Dadurch wird es möglich, XML basierte Dokumente durch IDL-Datenstrukturen zu repräsentieren. Ohne vorheriges Parsen kann auf die einzelnen Elemente des XML-Dokuments zugegriffen werden.

Für die weitere Entwicklung der CORBA Notification Service-Spezifikation ist der Fortgang der Integration mit J2EE interessant. Es bestehen Bestrebungen, die Interoperabilität zwischen JMS und dem Notification Service zu standardisieren [24]. Dabei wird eine Abbildung zwischen JMS-Events und StructuredE-

vents definiert, die JMS und Notification Service-Clients erlaubt, auf standardisierte Weise miteinander zu kommunizieren.

7.2. Weiterentwicklung des JacORB Notification Service

Im Laufe der Implementierung und der Performanzmessungen entstanden einige Ideen für Verbesserungsmöglichkeiten. Viele dieser Ideen wurden unmittelbar in die Implementierung eingearbeitet. Andere Ideen wurden aus Zeitgründen nicht umgesetzt. Diese Ideen sind nachfolgend beschrieben.

Es handelt sich dabei nicht um Funktionalität, die notwendig ist, um eine Konformität mit der OMG-Spezifikation zu erreichen. Vielmehr werden implementierungsspezifische Verbesserungsmöglichkeiten beschrieben.

7.2.1. Implementierung der Filter

Eine Laufzeitanalyse der Auswertung der Filter ergab, dass ein Großteil der Zeit in der DynAny Implementierung von JacORB verbracht wird. Die DynAny Implementierung von JacORB wäre daher ein Ansatzpunkt, um eine höhere Performanz zu erreichen [vgl. 20]. Die derzeitige Variante des Evaluierungs-Algorithmus traversiert rekursiv durch den abstrakten Syntaxbaum und extrahiert die relevanten Daten aus der bearbeiteten Nachricht. Die Umwandlung des rekursiven Algorithmus in eine iterative Variante könnte die Performanz weiter verbessern.

Die geparsten Filterausdrücke werden als Baum repräsentiert. Alternativ entstand die Idee, aus dem Baum direkt ausführbaren Java-Bytecode zu generieren. Diese Generierung könnte mit einem Werkzeug wie BCEL durchgeführt werden, und dieser Bytecode wäre dann direkt vom Java Interpreter aus interpretierbar. Im Ergebnis würde man eine höhere Verarbeitungsgeschwindigkeit und einen geringeren Platzbedarf erzielen. Vorab wäre die Komplexität dieses Vorhabens und des dadurch erzielten Gewinns an einem Beispiel zu evaluieren.

Bei der Auswertung von *StructuredEvent*- oder *Any*-Nachrichten wird in der jetzigen Implementierung größtenteils der gleiche Code durchlaufen. Dies ist beabsichtigt (Abschnitt 4.1.3), bietet aber Raum für zahlreiche Optimierungen bei der Auswertung von StructuredEvents. Bei der Auswertung der sog. Shorthand-Notation konnten entsprechende Optimierungen die Performanz erheblich verbessern (vgl. Abschnitt 6.2.4). Es bleibt, weitere Bereiche zu identifizieren, in denen Ähnliches möglich ist.

In eine andere Richtung geht die Idee, ein zusätzliches Filterkonzept zu realisieren. Filtersprachen wie ETCL können auf Nutzerebene nicht erweitert oder angepasst werden. Die Filtersprachen sind ein zur Programmiersprache redundantes und orthogonales Konzept [17]. Die Formulierung von Filterausdrücken ist fehleranfällig, da die Gültigkeit der Ausdrücke erst zur Laufzeit geprüft werden kann. Außerdem stellt die direkte Bezugnahme auf Attributwerte innerhalb der Filtersprache eine Verletzung der Datenkapselung dar.

Die Autoren Eugster und Guerraoui schlagen daher ein alternatives Filterkonzept vor. Filterausdrücke können mit Hilfe von Objekten formuliert werden. Das Konzept ist *Query-by-Example* vergleichbar, das in objektorientierten Datenbanken eingesetzt wird. Dadurch wird die Kapselung von Objekten nicht verletzt und statische Typprüfung ermöglicht [17].

7.2.2. Genaue Evaluierung des Ressourcenbedarfs

Das realisierte Threadmodell (Abschnitt 4.2.4) und die Verwaltung temporärer Objekte (Abschnitt 4.2.4) stellt einen flexiblen, skalierbaren Ansatz dar. Andererseits kann der dadurch verursachte Mehraufwand bei kleinen Installationen (wenige Consumer, Supplier und Events) unnötig sein.

Im ersten Schritt sind die Einstellmöglichkeiten dem Nutzer auf komfortable Weise verfügbar zu machen. Dafür bietet sich die im JacORB verwendete Propertydatei *jacorb_properties* an. Alternativ können die Einstellmöglichkeiten durch entsprechend herstellerspezifisch erweiterte IDL-Schnittstellen auf Applikationsebene verfügbar gemacht werden.

Im nächsten Schritt können dann Testreihen mit unterschiedlicher Zielsetzung stattfinden. Eine Zielsetzung könnte beispielsweise sein, den Speicherbedarf, unter Vernachlässigung von Latenzzeiten und Durchsatz, drastisch zu verringern. Ein anderes, sehr interessantes Szenario wäre der Einsatz auf einem Multiprozessorsystem. Dort könnte versucht werden, die Latenzzeit zu minimieren. Die Testreihen sollten auf unterschiedlichen Konfigurationen von Hardware und Betriebssystem zum Einsatz kommen.

Werden im Laufe dieser Testreihen Optimierungen am Notification Service durchgeführt, wird das neue Verhalten für den Nutzer optional konfigurierbar gemacht. Damit entstehen zahlreiche „Stellschrauben“, die es erlauben, das System optimal an bestimmte Einsatzgebiete anpassen zu können.

7.2.3. Realisierung eines Administrationswerkzeugs

Im engen Zusammenhang mit den Einstellmöglichkeiten steht die Realisierung eines Konfigurationswerkzeugs. Das Programm `manager`, das zum Lieferumfang von OpenFusion gehört, demonstriert, wie ein Adminstrationswerkzeug aufgebaut sein kann. Ein entsprechendes Werkzeug bietet die Möglichkeit, einen EventChannel statisch zu konfigurieren. Ein weiterer wichtiger Bereich ist die Überwachung zur Laufzeit (Monitoring). Es ist ein Programm zur Überwachung eines EventChannel denkbar, das die Struktur eines aktiven EventChannel hierarchisch darstellt und verschiedene verfügbare Informationen zu den einzelnen Elementen darstellt. Damit könnte beispielsweise ermittelt werden wieviele Nachrichten ein bestimmter Supplier versendet hat oder wie lange die Auslieferung von Nachrichten an einen Consumer dauert.

Diese Informationen können beim Betrieb des Notification Service helfen Probleme und Engpässe frühzeitig zu erkennen und zu vermeiden.

7.2.4. Erstellung und Umsetzung einer Logging-Strategie

Eine gute Logging-Strategie kann einem Entwickler, der den JacORB Notification Service einsetzt, wertvolle Dienste leisten. Die derzeitige Logging-Strategie erfasst vor allem sog. Trace-Informationen. Diese Informationen dokumentieren den Verlauf der einzelnen Operationen und sind insbesondere für die Weiterentwicklung bzw. Fehlersuche in der Implementierung hilfreich.

Der Entwickler, der den Notification Service lediglich einsetzen will, ist an den meisten dieser Informationen nicht interessiert. Aus diesem Grund sollte eine Logging-Strategie entwickelt und in der Implementierung umgesetzt werden, die es erlaubt Informationen aus unterschiedlichen Subsystemen des Notification Service mit variablem Detaillierungsgrad zu erfassen. Das verwendete Avalon Logging API bietet dazu bereits gute technische Voraussetzungen.

A. Messergebnisse

Supplier	Consumer	Nachrichten	Mittlere Bearbeitungszeit in ms	Latenzzeit in ms		
				min	max	avg
1	1	100	285	5	103	8
1	10	100	415	22	217	38
1	50	100	1021	84	385	162
5	1	100	353	4	163	6
5	10	100	487	8	582	49
5	50	100	1655	113	4075	498
10	1	100	420	4	252	8
10	10	100	638	5	659	61
10	50	100	2534	113	3253	411

Tabelle A.1.: Latenzzeiten für OpenFusion. Keine Filter. Intervall 250ms

Supplier	Consumer	Nachrichten	Mittlere Bearbeitungszeit in ms	Latenzzeit in ms		
				min	max	avg
1	1	100	275	9	167	16
1	10	100	272	53	245	75
1	50	100	367	685	4752	2975
5	1	100	278	13	154	18
5	10	100	413	130	11709	7576
5	50	100	1720	655	137742	72221
10	1	100	294	17	298	39
10	10	100	844	160	52456	29879
10	50	100	4696	677	431192	237776

Tabelle A.2.: Latenzzeiten für OpenORB. Keine Filter. Intervall 250ms

Supplier	Consumer	Nachrichten	Mittlere Bearbeitungszeit in ms	Latenzzeit in ms		
				min	max	avg
1	1	100	264	6	36	7
1	10	100	267	16	173	37
1	50	100	296	138	391	197
5	1	100	265	7	146	10
5	10	100	303	13	352	61
5	50	100	949	314	21762	12546
10	1	100	288	10	424	19
10	10	100	431	33	789	113
10	50	100	1931	192	53812	34179

Tabelle A.3.: Latenzzeiten für JacORB. Keine Filter. Intervall 250ms

Supplier	Consumer	Nachrichten	Mittlere Bearbeitungszeit in ms	Latenzzeit in ms		
				min	max	avg
1	1	1	294	9	—	9
1	10	10	465	28	30	28
1	50	50	1394	45	556	195

Tabelle A.4.: Latenzzeiten für OpenFusion. 1 Filter pro Supplier. Intervall 250ms

Supplier	Consumer	Nachrichten	Mittlere Bearbeitungszeit in ms	Latenzzeit in ms		
				min	max	avg
1	1	1	269	99	—	99
1	10	10	301	19	141	38
1	50	50	406	18	592	91

Tabelle A.5.: Latenzzeiten für OpenORB. 1 Filter pro Supplier. Intervall 250ms

Supplier	Consumer	Nachrichten	Mittlere Bearbeitungszeit in ms	Latenzzeit in ms		
				min	max	avg
1	1	1	265	15	—	15
1	10	10	269	31	100	47
1	50	50	317	131	445	211

Tabelle A.6.: Latenzzeiten für JacORB. 1 Filter pro Supplier. Intervall 250ms

Filterausdruck	Bearbeitungszeit, ms
I	2
II	11
III	9
IV	33
V	59
VI	83
VII	124

Tabelle A.7.: Laufzeiten für die Filterevaluierung bei OpenFusion (siehe Seite 100 für die vollständigen Filterausdrücke)

Filterausdruck	Bearbeitungszeit, ms
I	4
II	54
III	49
IV	31
V	37
VI	43
VII	49

Tabelle A.8.: Laufzeiten für die Filterevaluierung bei OpenORB

Filterausdruck	Bearbeitungszeit, ms
I	2
II	18
III	15
IV	35
V	36
VI	36
VII	35

Tabelle A.9.: Laufzeiten für die Filterevaluierung bei JacORB

Nachrichten	gesendet	empfangen
10	196	204
100	151	149
1000	148	141
5000	152	147
10000	151	148
20000	150	146

Tabelle A.10.: Maximaler Durchsatz OpenFusion, Nachrichten pro Sekunde

Nachrichten	gesendet	empfangen
10	85	99
100	78	78
1000	71	70
5000	71	70
10000	71	70
20000	40	39

Tabelle A.11.: Maximaler Durchsatz OpenORB, Nachrichten pro Sekunde

Nachrichten	gesendet	empfangen
10	200	208
100	132	131
1000	141	137
5000	129	126
10000	132	127
20000	94	91

Tabelle A.12.: Maximaler Durchsatz JacORB, Nachrichten pro Sekunde

Nachrichten	gesendet	empfangen
10	81	123
100	90	91
1000	69	68
5000	66	65
10000	57	56
20000	53	51

Tabelle A.13.: Maximaler Durchsatz JacORB in Verbindung mit dem OpenORB ORB, Nachrichten pro Sekunde

Größe in byte	gesendet	empfangen
32	163	162
1024	150	145
2048	136	136
4096	105	105
8192	85	86
16384	47	47
32768	21	21

Tabelle A.14.: Durchsatz in Abhängigkeit von der Nachrichtengröße OpenFusion, Nachrichten pro Sekunde

Größe in byte	gesendet	empfangen
32	63	63
1024	62	62
2048	58	55
4096	61	61
8192	49	49
16384	25	25
32768	26	26

Tabelle A.15.: Durchsatz in Abhängigkeit von Nachrichtengröße, Nachrichten pro Sekunde: OpenORB

Nachrichten	gesendet	empfangen
32	159	161
1024	113	108
2048	101	101
4096	94	94
8192	65	64
16384	41	41
32768	25	25

Tabelle A.16.: Durchsatz in Abhängigkeit von Nachrichtengröße, Nachrichten pro Sekunde: JacORB

B. Ausgabe des Programms sloccount

```
> cd src/org/jacorb/notification
> sloccount .
[...]

SLOC    Directory       SLOC-by-Language (Sorted)
5892    top_dir         java=5892
2366    node            java=2366
1026    util            java=1026
816     engine          java=816
627     evaluate        java=627
617     grammar         antlr=617
78      interfaces      java=78

Totals grouped by language (dominant language first):
java:          10805 (100.00%)

Total Physical Source Lines of Code (SLOC)                = 10,805
Development Effort Estimate, Person-Years (Person-Months) = 2.43 (29.21)
 (Basic COCOMO model, Person-Months = 2.4 * (KSLOC**1.05))
Schedule Estimate, Years (Months)                         = 0.75 (9.01)
 (Basic COCOMO model, Months = 2.5 * (person-months**0.38))
Estimated Average Number of Developers (Effort/Schedule)  = 3.24
Total Estimated Cost to Develop                           = $ 328,812
 (average salary = $56,286/year, overhead = 2.40).
Please credit this data as "generated using 'SLOCCount' by David A. Wheeler."
```

Die Werte für das Verzeichnis *grammar* wurden mit Hilfe des Aufrufs `cat *.txt *.g | wc -l` bestimmt und von Hand eingetragen, da *sloccount* das ANTLR-Dateiformat nicht kennt und die Dateien daher ignoriert.

Literaturverzeichnis

[1] Yeturu Aahlad, Bruce E. Martin, Mod Marathe und Chung Lee. Asynchronous Notifications Among Distributed Objects. In *Proceedings of the Second USENIX Conference on Object-Oriented Technologies and Systems (COOTS), June 17–21, 1996, Toronto, Canada*, Seiten 83–95. Berkeley, USA, 1996. URL: http://www.usenix.org/publications/library/proceedings/coots96/aahlad.html.

[2] Marcos Kawazoe Aguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley und Tushar Deepak Chandra. Matching Events in a Content-Based Subscription System. In *18th ACM Symposium on Principles of Distributed Computing (PODC '99)*, Seiten 53–61. Atlanta, USA, 1999. URL: http://citeseer.nj.nec.com/aguilera99matching.html.

[3] Markus Aleksy, Martin Schrader und Alexander Schell. Design and Implementation of a Bridge Between CORBA's Notification Service and the Java Message Service. In *Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS'03)*. IEEE Computer Society, Hawaii, USA, 2003. URL: http://www.computer.org/proceedings/hicss/1874/track9/187490318babs.htm. 49

[4] Ant. Apache Ant. URL: http://jakarta.apache.org/ant/index.html.

[5] ANTLR. Complete Language Translation Solutions. URL: http://www.antlr.org. 67, 80

[6] Gerald Brose, Klaus-Peter Löhr und André Spiegel. Java does not distribute. In Christine Mingins, Roger Duke und Bertrand Meyer (Herausgeber), *Proceedings of the Twenty-Fifth Conference on the Technology of Object-Oriented Languages and Systems (TOOLS Pacific'97)*, Seiten 24–27. Melbourne, Australien, 1997. URL: http://citeseer.nj.nec.com/brose97java.html. 9

[7] Gerald Brose, Andreas Vogel und Keith Duddy. *Java programming with CORBA: Advanced Techniques for Building Distributed Applications*. John Wiley and Sons, New York, USA; London, GB; Sydney, Australia, dritte Auflage, 2001. 15, 23, 24, 59, 108

[8] Brenton Camac. Using EJBs with the OMG Notification Service. White Paper, Borland Software Corporation, 2002. URL: http://bdn.borland.com/article/0,1410,29321,00.html. 59, 60

[9] Antonio Carzaniga. *Architectures for an Event Notification Service Scalable to Wide-area Networks*. Doktorarbeit, Politecnico di Milano, Mailand, Italien, 1998. URL: http://www.cs.colorado.edu/~carzanig/papers/index.html. 15, 16, 18, 33, 34, 49

[10] Antonio Carzaniga, David R. Rosenblum und Alexander L. Wolf. Challenges for Distributed Event Services: Scalability vs. Expressiveness. In *Engineering Distributed Objects '99*. Los Angeles, USA, 1999. URL: http://www.cs.colorado.edu/~carzanig/papers/index.html.

[11] Antonio Carzaniga, David S. Rosenblum und Alexander L. Wolf. Achieving Scalability and Expressiveness in an Internet-Scale Event Notification Service. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, Seiten 219–227. Portland, USA, 2000. URL: http://www.cs.colorado.edu/~carzanig/papers/index.html. 48

[12] Antonio Carzaniga und Alexander L. Wolf. Content-based Networking: A New Communication Infrastructure. In *NSF Workshop on an Infrastructure for Mobile and Wireless Systems*. Scottsdale, USA, 2001.

[13] Antonio Carzaniga und Alexander L. Wolf. Fast Forwarding for Content-Based Networking. Technical Report CU-CS-922-01, Department of Computer Science, University of Colorado, 2001. URL: http://www.cs.colorado.edu/~carzanig/papers/index.html.

[14] Antonio Carzaniga und Alexander L. Wolf. A Benchmark Suite for Distributed Publish/Subscribe Systems. Technical Report CU-CS-927-02, Department of Computer Science, University of Colorado, 2002. URL: http://www.cs.colorado.edu/~carzanig/papers/index.html. 25, 95

[15] Irmen de Jong. Web Services/SOAP and CORBA. 2002. URL: http://www.xs4all.nl/~irmen/comp/CORBA_vs_SOAP.html. 111

[16] Arne M. Degenring. *Konzeption und Implementierung zur Integration eines SAS Servers in SOAP-basierte Web Service Umgebungen*. Diplomarbeit, Fernuniversität Hagen, 2002. 111

[17] Patrick T. Eugster und Rachid Guerraoui. Content-Based Publish/-Subscribe with Structural Reflection. In *6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'01)*. San Antonio, USA, 2001. URL: http://www.usenix.org/publications/library/proceedings/coots01/eugster.html. 113, 114

[18] Daniel Faensen, Lukas Faulstich, Heinz Schweppe, Annika Hinze und Alexander Steidinger. Hermes: A Notification Service for Digital Libraries. In *JCDL'01: Proceedings of the 1st ACM/IEEE-CS Joint Conference on Digital Libraries*, Scholarly Communication and Digital Libraries, Seiten 373–380. Roanoke, USA, 2001. URL: http://www.acm.org/pubs/articles/proceedings/dl/379437/p373-faensen/p373-faensen.pdf.

[19] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley, Bonn, 1996. 15, 22, 23, 55, 56, 60

[20] Pradeep Gore, Ron Cytron, Douglas C. Schmidt und Carlos O'Ryan. Designing and Optimizing a Scalable CORBA Notification Service. In *Proceedings of The Workshop on Languages, Compilers, and Tools for Embedded Systems LCTES 2001/ The Workshop on Optimization of Middleware and Distributed Systems OM 2001*, Seiten 196–204. Snowbird, USA, 2001. URL: http://citeseer.nj.nec.com/gore01designing.html. 9, 51, 55, 56, 113

[21] John Gough und Glenn Smith. Efficient recognition of events in distributed systems. In *Proceedings of 18th Australasian Computer Science Conference (ACSC)*. Adelaide, Australien, 1995. URL: http://citeseer.nj.nec.com/gough95efficient.html.

[22] Christophe Gransart und David Simplot. Communicating Mobile Objects - An Overview. In *Proceedings of the Gemplus Developer Conference (GDC'2000)*. Montpellier, Frankreich, 2000. URL: citeseer.nj.nec.com/gransart00communicating.html. 109

[23] The Object Management Group. *Notification Service Specification*, 2000. URL: http://www.omg.org/cgi-bin/doc?formal/2002-08-04. 46, 51

[24] The Object Management Group. *Notification / Java Message Service Interworking*, 2001. URL: http://www.omg.org/cgi-bin/doc?ab/2001-07-01. 49, 112

[25] The Object Management Group. *Super Distributed Objects*, 2001. URL: http://www.omg.org/cgi-bin/doc?sdo/01-07-05. 109

[26] The Object Management Group. *CORBA Components*, 2002. URL: http://www.omg.org/cgi-bin/doc?formal/02-06-65. 108

[27] The Object Management Group. *CORBA to WSDL/SOAP Interworking Specification*, 2003. URL: http://www.omg.org/cgi-bin/doc?ptc/2003-01-14. 112

[28] Robert Gruber, Balachander Krishnamurthy und Euthimios Panagos. CORBA Notification Service: Design Challenges and Scalable Solutions. In *17th International Conference on Data Engineering (ICDE'01)*, Seiten 13–20. IEEE, Washington - Brüssel - Tokio, 2001. 51, 52, 57

[29] Timothy H. Harrison, David L. Levine und Douglas C. Schmidt. The Design and Performance of a Real-time CORBA Event Service. In *Proceedings of the OOPSLA*, Seiten 184–200. Atlanta, USA, 1997. URL: http://citeseer.nj.nec.com/article/harrison97design.html. 12

[30] Annika Hinze. A Model of Alerting Services in Wide Area Networks. In *Grundlagen von Datenbanken*, Seiten 42–46. Jena, Deutschland, 1999. URL: http://citeseer.nj.nec.com/hinze99model.html. 21

[31] Annika Hinze. Efficient filtering of composite events. In *Proceedings of the 20th British National Conference on Databases (BNCD'03)*. Coventry, GB, 2003. 108

[32] Annika Hinze und Sven Bittner. Efficient Distribution-Based Event Filtering. In *Proceedings of the Workshop on Distributed Event-based Systems (DEBS)*, Seiten 2–5. Wien, Österreich, 2002. 27, 108

[33] Annika Hinze und Daniel Faensen. A Unified Model of Internet Scale Alerting Services. In *Proceedings of the 5th International Computer Science Conference ICSC'99*, Seiten 284–293. Hong Kong, China, 1999. URL: http://citeseer.nj.nec.com/hinze99unified.html.

[34] IBM. *WebSphere MQ Event Broker – Einführung und Planung*, Erste Auflage, 2002. URL: http://www.elink.ibmlink.ibm.com/public/applications/publications/cgibin/pbi.cgi?CTY=DE. 37

[35] IBM. *WebSphere MQ Event Broker – Programming Guide*, Erste Auflage, 2002. URL: http://www.elink.ibmlink.ibm.com/public/applications/publications/cgibin/pbi.cgi?CTY=DE. 37

[36] IBM. *WebSphere MQ Integrator – ESQL Reference*, dritte Auflage, 2002. URL: http://www.elink.ibmlink.ibm.com/public/applications/publications/cgibin/pbi.cgi?CTY=DE. 38

[37] JacORB. The free Java implementation of the OMG's CORBA standard. URL: http://www.jacorb.org. 6

[38] JMS. Java Message Service. URL: http://java.sun.com/products/jms. 7

[39] JUnit. Testing Resources for Extreme Programming. URL: http://www.junit.org.

[40] Markku Korhonen. Message Oriented Middleware (MOM). 1997. URL: http://www.tml.hut.fi/Opinnot/Tik-110.551/1997/mqs.htm. 37

[41] Stefan Langerman, Sachin Lodha und Rahul Shah. Algorithms for Efficient Filtering in Content-Based Multicast. In F. Meyer auf der Heide (Herausgeber), *9th Annual European Symposium on Algorithms (ESA'01)*, Band 2161 *LNCS*, Seiten 428–439. Springer, Aarhus, Dänemark, 2001. URL: http://link.springer.de/link/service/series/0558/bibs/2161/21610428.htm.

[42] Doug Lea. *Concurrent Programming in Java[tm], Second Edition: Design principles and Patterns*. The Java Series. Addison Wesley, zweite Auflage, 1999. 63

[43] Christoph Liebig, Mariano Cila und Alejandro Buchmann. Event Composition in Time-dependent Distributed Systems. In *Proceedings of the 4th IFCIS International Conference on Cooperative Information Systems (CoopIS'99)*, Seiten 70–78. IEEE Computer Society, Edinburgh, Schottland, 1999. URL: http://citeseer.nj.nec.com/liebig99event.html. 107

[44] Klaus-Peter Löhr. Automatic Mediation between Incompatible Component Interaction Styles. In *Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS'03)*. IEEE Computer Society, Hawaii, USA, 2003. URL: http://www.inf.fu-berlin.de/~lohr/papers/hicss-36.pdf. 108

[45] Richard Monson-Haefel und David Chappell. *Java Message Service*. O'Reilly & Associates, Inc., Newton, USA, 2001.

[46] Gero Mühl. Generic Constraints for Content-Based Publish/Subscribe Systems. In *Proceedings of the 6th International Conference on Cooperative Information Systems (CoopIS'01)*, Band 2172 *LNCS*, Seiten 211–225. Springer, Trento, Italien, 2001. URL: http://www.informatik.tu-darmstadt.de/GK/participants/muehl/muehl_f.html#CoopIS2001.

[47] OpenORB. The Community OpenORB. URL: http://openorb.sourceforge.net. 6

[48] Joao Pereira, Francoise Fabret, Francois Llirbat und Dennis Shasha. Efficient Matching for Web-Based Publish/Subscribe Systems. In *Conference on Cooperative Information Systems (CoopIS'2000)*, Seiten 162–173. Eilat, Israel, 2000. URL: http://citeseer.nj.nec.com/article/llirbat00efficient.html.

[49] Peter R. Pietzuch. Event-Based Middleware: A New Paradigm for Wide-Area Distributed Systems? URL: http://citeseer.nj.nec.com/527965.html.

[50] PrismTech. *OpenFusion Version 1.1 Developer's Guide Volume I*, 1999.

[51] PrismTech. *OpenFusion Notification Service: Performance Evaluation*, 2001. URL: http://www.prismtechnologies.com/English/Downloads/CORBA/index.html.

[52] Arno Puder. Objekte ohne Grenzen. *iX, Magazin für Professionelle Informationstechnik*, 12:86–95, 2002. URL: http://www.heise.de/ix/. 112

[53] Douglas C. Schmidt und Adam Porter. Leveraging Open-Source Communities To Improve the Quality Performance of Open-Source Software. URL: http://citeseer.nj.nec.com/453083.html.

[54] Heinz Schweppe, Annika Hinze und Daniel Faensen. Database Systems as Middleware – Events, Notifications, Messages. In *Advances in Databases and Information Systems (ADBIS-DASFAA)*, Seiten 21–22. Prag, Tschechische Republik, 2000. 19

[55] Heinz Schweppe, Annika Hinze und Daniel Faensen. Event-based Notification in the WWW. Tutorial at the 9th International World Wide Web Conference in Amsterdam, 2000.

[56] Jack Shirazi. *Java Performance Tuning*. Java series. O'Reilly & Associates, Inc., Newton, USA, 2000. 63

[57] SIENA. A Wide-Area Event Notification Service. URL: http://www.cs.colorado.edu/~carzanig/siena. 7

[58] Alex C. Snoeren, Kenneth Conley und David K. Gifford. Mesh-Based Content Routing using XML. In *18th ACM Symposium on Operating System Principles*. Banff, Kanada, 2001. URL: http://www.psrg.lcs.mit.edu/publications/Papers/xml-sosp01.html.

[59] soap2corba. Soap2Corba and Corba2Soap. URL: http://soap2corba.sourceforge.net/. 112

[60] Robert Strom, Guruduth Banavar, Tushar Chandra, Marc Kaplan, Kevan Miller, Bodhi Mukherjee, Daniel Sturman und Michael Ward. Gryphon: An information flow based approach to message brokering. 1998. URL: http://citeseer.nj.nec.com/strom98gryphon.html. 48

[61] Karsten Thurow. *Ein generisches Notifikations-Framework*. Diplomarbeit, Universität Hamburg, 2000.

[62] util.concurrent. A Java package containing locks, queues, thread pools. URL: http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html. 65, 88

[63] Steve Vinoski und Douglas C. Schmidt. Programming Asynchronous Method Invocations with CORBA Messaging. *SIGS C++ Report*, 11(2), 1999. URL: http://www.iona.com/hyplan/vinoski/col16.pdf. 15

[64] Michael Weber. *Verteilte Systeme*. Spektrum, Heidelberg – Berlin, 1998.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit über das Thema „Entwurf und Implementierung des CORBA Notification Service“ in der gesetzten Frist selbstständig verfasst und keine anderen Quellen und Hilfsmittel als die angegebenen verwendet habe.

Berlin, den 26. Mai 2003

Alphonse Bendt