Diplomarbeit

Evaluierung, Implementierung und Testen von Message Oriented Middleware

ausgeführt am Institut

Information & Software Engineering Group
Technische Universität Wien

unter der Anleitung von
Ao.Univ.Prof. Dipl.Ing. Dr.techn. Andreas Rauber
und
Dipl.Ing. Dr.techn. Alexander Schatten

als verantwortlich mitwirkendem Assistenten

durch

David Sternberger, Bakk. techn.
Molkereistrasse 3/7, 1020 Vienna
Matr.Nr. 9925040

Wien, 03.02.2006 ________________________

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Wien, 03.02.2006 ______________________________

Danksagung

An dieser Stelle möchte ich all jenen Personen Dank sagen, die während des Studiums hinter mir gestanden sind und mich moralisch, motivierend und finanziell unterstützt haben. Meinen Eltern Rudolf und Roswitha sowie meinem Bruder Thomas gilt besonderer Dank für die Unterstützung in allen Lebenslagen sowie den familiären Rückhalt. Sie haben nie an mir gezweifelt - Vielen Dank für dieses Vertrauen.
Eine Person möchte ich noch besonders hervorzuheben. Elke Fierlinger hat einen großen Anteil am Erfolg dieser Arbeit. Sie hat mich auf meinem Weg lange Zeit begleitet und mich dabei immer unterstützt.
Ohne meine Freunde Thomas und Robert wäre diese Arbeit wohl auch nicht zustande gekommen - vielen Dank für die gute Freundschaft und Zusammenarbeit.
Auch meine restlichen Freunde möchte ich hier erwähnen, sie sorgten oft für Motivation und auch für den notwendigen Ausgleich während des Studiums.
Großer Dank gilt auch Herrn Alexander Schatten für die Betreuung der Diplomarbeit.

Abstract

The rapid improvements regarding software and networks trends implicate envolvement of the classical periphery of isolated applications into a heteogenous network of distributed services. In order to ensure an economical and competitive operation in a company various software products have to work together as flawless as possible. The interaction of different software solutions and hardware platforms creates the need of middleware, which should provide a standardized way of data communication between the heterogenous systems. Over the years a number of different approaches for maintaining an interaction of the applications and paradigms have been developed. Furthermore there is a necessity of new methods which should help to ensure software quality. At first the following thesis explains the basic principles of middleware. Afterwards a detailed reflection of message-oriented systems will be given. The remainder of the work will focus on the implementation and the testing of message-oriented middleware.

Zusammenfassung

Durch die rasche Entwicklung im Bereich von Software und Netzwerken wandelt sich das klassische Umfeld von einzelnen Insellösungen hin zu einem vernetzten heterogenen System. Verschiedenste Softwareprodukte sollen möglichst reibungslos interagieren, um einen wirtschaftlichen und konkurrenzfähigen Betrieb aufrechtzuerhalten. Da viele unterschiedliche Softwarelösungen und Plattformen aufeinander treffen, ist die so genannte Middleware entstanden, welche eine einheitliche und standardisierte Datenkommunikation zwischen heterogenen Systemen ermöglichen soll. Im Laufe der Zeit entwickelte sich eine Reihe an verschiedenen Ansätzen und Paradigmen, die eine Interaktion verteilter Anwendungen unterstützen. Im Weiteren sind neue Methoden zur Sicherstellung und Quantifizierung der qualitativen Güte notwendig. In dieser Arbeit werden zunächst die Grundlagen von Middleware erleutert, anschließend erfolgt eine detailierte Betrachtung von nachrichtenorientierten Systemen. Abschließend wird die Implementierung und das Testen eines nachrichtenorientierten Middleware Systems beschrieben.

Inhaltsverzeichnis

Eidesstattliche Erklärung i
Acknowledgements i
Abstract ii
Zusammenfassung ii

Table of Contents **iii**

1 Middleware **1**
1.1 Definition von Middleware 1
1.2 Geschichte von Middleware 2
1.3 Middleware Services 4
1.3.1 Identifiers 5
1.3.2 Naming Services und Directories 5
1.3.3 Authentifikation 5
1.3.4 Autorisation und Zutrittsbeschränkung 5
1.3.5 Zertifikate und PKI 6
1.3.6 Transaktionsüberwachung 6
1.3.7 Prozess und Thread Kontrolle 6
1.3.8 User und Debuginterface 6
1.3.9 Load Balancing 6
1.3.10 Persistenz 7
1.3.11 Connectivity 7
1.3.12 Datentransformation und Austausch 7
1.3.13 Broker Service 7
1.3.14 Firewall 7
1.3.15 Location Transparency 7
1.3.16 Message Integrity 8
1.3.17 Dynamische Konfiguration 8
1.3.18 Quality of Service 8
1.3.19 Context-awareness 8
1.4 Kategorisierung von Middleware 8
1.4.1 Unterscheidung aufgrund der Komponenten 10
1.4.2 Unterscheidung aufgrund der Verbindung 10
1.4.3 Statischer oder dynamischer Kontext 10
1.4.4 Nicht funktionale Ansprüche 11
1.4.5 Prozedurbasierte Middleware (POM) 11

1.4.6 Datenbankorientierte Middleware (DOM) 12
1.4.7 Nachrichtenbasierte Middleware (MOM) 14
1.4.8 Komponentenbasierte Middleware 16
1.4.9 Objektorientierte Middleware (OOM) 17
1.4.10 Echtzeitorientierte Middleware 18
1.4.11 Web-Services 18
1.5 Kopplung 20
1.5.1 Kopplung in Bezug auf Raum 20
1.5.2 Kopplung in Bezug auf Zeit 22
1.5.3 Kopplung in Bezug auf Synchronisation 24
1.6 System Architektur 27
1.6.1 Zentralisierte Systeme 27
1.6.2 Hierarchisch zentralisierte Systems 27
1.6.3 Vermittelte Systeme 28
1.6.4 Dezentralisierte Systeme 28
1.6.5 Super Peer Systeme 28
1.7 Anwendungsintegration 28
1.7.1 Abhängigkeit 29
1.7.2 Einfachheit der Integration 29
1.7.3 Integrationstechnik 29
1.7.4 Dateneigenschaften 29
1.7.5 Funktionsaufrufe 30

2 Nachrichtenbasierte Middleware (MOM) 31
2.1 Definition 31
2.1.1 Bezeichnungen 32
2.2 Nachrichten 33
2.3 Typen von Message - Channels 34
2.3.1 Point-to-Point Channel 34
2.3.2 Publish Subscribe Channel 35
2.4 Publish/Subscribe Paradigma 36
2.4.1 Events und Notifizierungen 36
2.4.2 Publisher und Subscriber 37
2.4.3 Subscriptions 38
2.4.4 Advertisement 42
2.4.5 Notification Service 42
2.4.6 Security in Publish/Subscribe Systems 43
2.5 Beschreibungssprachen 46

2.5.1 MSC Message Sequence Chart . . . 46
2.5.2 Erweiterte Basic Message Sequence Charts . . . 50
2.5.3 LSC Live Sequence Chart . . . 52

3 Bestehende Produkte im Überblick 55
3.1 Kommerzielle Produkte . . . 55
3.1.1 Advanced Queuing . . . 55
3.1.2 Arjuna Message Service . . . 56
3.1.3 MessageQ . . . 56
3.1.4 MSMQ . . . 56
3.1.5 WebSphere MQ . . . 56
3.2 Open Source Produkte . . . 57
3.2.1 JORAM . . . 57
3.2.2 MQ4CPP . . . 57
3.2.3 MantaRay . . . 57
3.2.4 xmlBlaster . . . 57

4 Implementation eines Publish/Subscribe Message Server 58
4.1 FMS Ziele und Designentscheidungen . . . 58
4.1.1 Publish / Subscribe Paradigma . . . 58
4.1.2 Plugin basierter Ansatz . . . 58
4.1.3 Unterstützung verschiedener Kommunikationsinterfaces . . . 59
4.1.4 Leichte funktionelle Erweiterbarkeit . . . 59
4.1.5 Leichte Einbindung von serverseitigen Funktionen . . . 59
4.1.6 Grafische Oberfläche . . . 59
4.1.7 Leicht konfigurierbar . . . 60
4.1.8 Entwicklung in Java 1.5 . . . 60
4.1.9 Konfiguration über XML- Files . . . 60
4.1.10 Unterschiedliche Laufumgebungen . . . 61
4.1.11 Pluginerweiterungen zur Laufzeit . . . 61
4.1.12 Universell einsetzbarer Basisclient . . . 61
4.2 System Architektur . . . 61
4.2.1 Topics . . . 62
4.3 Server Architektur . . . 64
4.3.1 Message Broker . . . 64
4.3.2 Plugin Framework JPF . . . 65
4.3.3 Systemplugins . . . 68

5 Testen eines Publish/Subscribe Systems 75
5.1 Risiken komponentenbasierter Systeme 75
5.1.1 Komponenten Entwickler 76
5.1.2 System Entwickler 76
5.1.3 Kunde 76
5.2 Testen 77
5.3 Testziel 78
5.4 Test Bereich 79
5.5 Test Verteilung 79
5.6 Testablauf 79
5.7 Testkriterien 79
5.7.1 Workflow 80
5.7.2 Message Delivery 80
5.7.3 Messge Ordering 81
5.7.4 Performance/Throughput 81
5.7.5 Response Time 82
5.7.6 Reaction Time 82
5.7.7 Reliability - Availability 82
5.7.8 Realtime Properties 82
5.8 Funktionale Tests 83
5.8.1 Funktionsorientiertes Testen (Black-Box) 83
5.8.2 Strukturorientiertes Testen (White Box) 84
5.9 Nichtfunktionale Tests 84
5.9.1 Heavy Load Test 85
5.9.2 Stress Test /Peak Test 86
5.9.3 Performance Test 86
5.10 Konkrete Testdruchführung am FMS 87
5.10.1 Teststruktur 87
5.10.2 Testnachrichten 90
5.10.3 Logging / Measurements 91
5.10.4 Test Szenarios 92
5.10.5 Testergebnisse 96

6 Fazit 97

Literaturverzeichnis 99

Abbildungsverzeichnis 105

Tabellenverzeichnis **106**

1 Middleware

1.1 Definition von Middleware

Der Begriff Middleware wird für eine Vielzahl an unterschiedlichen Softwarelösungen verwendet. Es gibt viele verschiedene Definitionen von Middleware. Das Wort Middleware lässt einigen Spielraum für Interpretation, doch diese Vielfältigkeit entspricht auch den vielfältigen Einsatzgebieten und Anwendungen, bei denen Middlewarelösungen zum Einsatz kommen. In der Literatur werden meist Definitionen verwendet, die das Spektrum bereits auf ein Gebiet einschränken. Je allgemeiner die Definition desto weniger wird beschrieben, was Middleware bedeutet. Um diesem Dilemma zu entgehen, werden mehrere unterschiedliche Definitionen vorgestellt.

"What is middleware anyway? A standing joke is that if you ask five People, you will get six definitions." **[AB02]**

- "*Middleware is software glue*": Dies ist wohl die einfachste Definition, jedoch nicht sehr aussagekräftig.
- "*Middleware is the slash between Client/Server":* Auch diese Definition wird dem Term Middleware nicht umfassend gerecht, da sie sich auf ein Paradigma beschränkt und somit die Vielfalt an Middlewarelösungen und Ansätzen verbirgt.
- "*Software that mediates between an application program and a network. It manages the interaction between disparate applications across the heterogeneous computing platforms*" [Com05b]. Diese Definition trifft die Materie schon sehr gut, Vermittlung ist die Hauptaufgabe von Middleware.
- "*In computing, middleware consists of software agents acting as an intermediary between different application components. It is used most often to support complex, distributed applications. The software agents involved may be one or many*" [Wik05c]. Im Gegensatz zur vorherigen Definition wird hier auf eine Interaktion zwischen mehreren Komponenten hingewiesen. Ebenso wird auf ein komplexes Umfeld verwiesen.
- "Middleware is any software that allows other software to interact" [Mid05].

- Eine Software heißt genau dann Middleware, wenn sie die Entwicklung und den Betrieb eines verteilten Systems ermöglicht und Funktionen anbietet, die über einfache Netzwerkkommunikation hinausgehen.

- Unter "Middleware" versteht man Software, die heterogene, miteinander inkompatible Anwendungsprogramme und Datenbestände so verknüpft, dass ein Datenaustausch zwischen ihnen möglich wird [IB05].

Middleware vereinfacht die Interaktion zwischen Applikationen, Softwarekomponenten, Netzwerken, Hardware und/oder dem Betriebssystem. Middleware vereinfacht die Entwicklung von Applikationen, da sie Services für die Kommunikation bereitstellt und diese nicht neu entwickelt werden müssen. Weiters stellt Middleware Dienste zur Verbesserung von Dienstqualität, Ausfallssicherheit, Persistenz und Sicherheit auf eine transparente Art und Weise zur Verfügung.

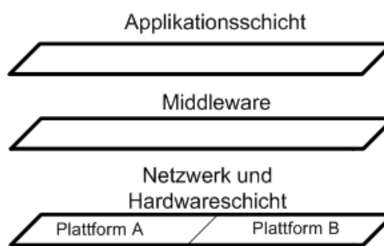

Abbildung 1: Schichtenbetrachtung von Middleware

Um dieses breite Spektrum der Definitionen nun genauer analysieren zu können, bedarf es einer Kategorisierung von Middlewaresysteme und deren Idee. Diese Kategorisierung ermöglicht die Bestimmung von Eigenschaften und eine Abgrenzung der verschiedenen Ansätze und Lösungen.

1.2 Geschichte von Middleware

Nach [TAB03] wurden die ersten middlewareähnlichen Ansätze noch nicht unter dem Begriff Middleware eingesetzt. Diese Systeme waren als Hilfsapplikationen für Kommunikationssysteme vorgesehen. Das erste Einsatzgebiet für solche Kommunikationssysteme war die Kommunikation zwischen Firmen, die im selben Geschäftsumfeld agierten, wie zum Beispiel Airlines. Bereits in

den 60er Jahren hätten solche Unternehmen ein verteiltes Netzwerk benötigt jedoch wurden diese Lösungen erst in den 70er Jahren von den großen Hardwareherstellern angeboten. Als die großen Firmen begannen, verteilte Netzwerke aufzubauen, wurde schnell ersichtlich, dass ein Produkt nötig war, welches Verbesserungen in der Performance, Kontrolle, Datenintegrität und der Usability brachte. Die Entwicklung eines solchen Produktes war sowohl zeitaufwändig als auch sehr kostspielig und war somit den großen Computerfirmen vorbehalten. Die erste richtige Middleware entstand in den frühen 80er Jahren von Sun Microsystems. Es basierte auf dem Remote-Procedur-Call (RPC) Protokoll und wurde im Open Network Computing (ONC) System eingesetzt. Dieses System ermöglichte den Aufruf eines Programms in einem entfernten Computer ohne detaillierte Kenntnisse über das Netzwerk besitzen zu müssen. Die Programmiersprache Ada trug in den Anfängen ebenfalls stark zur Entwicklung von Middleware bei, da diese die Portierung auf andere Hardwareplattformen und Betriebssysteme ermöglichte.
Die rasche Entwicklung von Netzwerken und der Hardware führte zur Implementierung von File und Directory Services. Dies trug dazu bei, dass sich der mehrschichtige Middlewareansatz immer mehr durchsetzte und die Client/Server Modelle zunehmend ablöste. Erste verteilte Systeme mit Middleware waren unter anderem Athena(C) am MIT, DACNOS an der Universität Karlsruhe und IBM Heidelberg. Im Jahre 1980 veröffentlichte die International Organisation for Standardization (ISO) den ersten Entwurf von Open System Interconnection (OSI). Dies begünstigte den Wettbewerb und führte so zu einer Preisreduktion von Middlewarelösungen, hatte jedoch keinen großen Einfluss auf die Evolution der Systeme.
Ein großer Evolutionssprung wurde durch die Einführung von Event Handling erreicht. Dies brachte die Möglichkeit der zeitgerechten Verarbeitung von Alarmen, Fehlern und der Kommunikation. Im Jahr 1989 wurde die Object Management Group (OMG) gegründet. Diese Organisation war für die Entwicklung der ersten Middleware-Spezifikation mit dem Namen Object Request Broker (ORB) verantwortlich. 1992 wurde von System Strategies (SSI) das nachrichtenorientierte System ezBridge für VMS, Unix und Tandem entwickelt. 1993 veröffentlichte IBM drei Standard API´s für Kommunikationssysteme (CPI-C, RPC, MQI) und der erste Artikel in der IEEE Explorer Datenbank zum Thema Middleware wurde veröffentlicht. Ebenfalls 1993 kaufte IBM die Lizenz für ezBridge und daraus entstand MQSeries Version 1 für die Plattformen VMS, Tandem, AS/400 und SCO Unix. Ab diesem Zeitpunkt nahmen die Publikationen rasch zu, so waren es im folgenden Jahr

bereits sieben und um die 170 Artikel in den darauf folgenden Jahren.
Die weitere Evolution bestand darin, zusätzliche Services in die Middleware wie Sicherheit und Transaktionshandling zu inkludieren. 1996 wurde die Spezifikation um Autorisierung und Zugriffskontrolle erweitert und 1998 wurde die Echtzeitspezifikation hinzugefügt. Mit der steigenden Notwendigkeit von Enterprise Ressource Planning (ERP) Anwendungen entwickelte sich Middleware weiter und ermöglichte somit die effiziente Integration neuer Systeme in eine bestehende Infrastruktur. Die Grenzen von Hardwareplattformen, Netzwerken, Betriebssystemen, Datenbanken und Anwendungen waren somit durch die Middleware überwunden.
Für Unternehmen ergeben sich somit nur zwei Optionen bei der Modernisierung und Optimierung vom IT-System. Zum einen die Homogenisierung von Systemen, Hardware, Software, Plattformen, Protokollen und Architekturen und zum anderen die Einführung von intelligenten Middlewarelösungen um die Administration zu vereinfachen. Der erste Ansatz ist sowohl zeit- und kostenintensiv und bringt für den Kunden kein wesentlich besseres Service. Die Einführung eines Middlewaresystems hingegen spart Zeit und Geld und kann zu verbesserten Services beitragen.

1.3 Middleware Services

Ein Service stellt seiner Umgebung einen Zugang zu einer Menge von Diensten zur Verfügung. Zum einen hält es die Deklaration der Dienstschnittstellen bereit, zum anderen kennt es die Implementierung oder kann diese zumindest lokalisieren. So ist das Service - wenn auch nur logischer - Knotenpunkt der Interaktionen zwischen Client und Server. Nach [Rau96] ist ein Client ein Prozess, der Dienste nutzt und die Anforderung eines Dienstes initiiert. Dazu übergibt er dem geforderten Service den Dienstidentifikator und die Eingabeparameter und erhält die Ausgabeparameter oder eine Fehlermeldung zurück. Da Clients meist dynamisch erzeugt werden, sind sie a priori nicht bekannt. Ein Server ist eine logische Softwareeinheit, die Dienste implementiert. Der Server stellt die Dienste über einen oder mehrere Services einer beliebigen Anzahl unbekannter Clients zur Verfügung.
Durch die starke Verbreitung von Middleware in den unterschiedlichsten Anwendungsgebieten haben sich bald die eigentlichen Kerndienste herauskristalisiert. Es gibt kein Produkt welches alle Dienste bereitstellt. Dies ist auch nicht nötig, da ein bestimmtes Einsatzgebiet selten alle Dienste benötigt. Die wichtigsten Dienste von Middleware sind:

1.3.1 Identifiers

Ein Identifier verbindet ein Objekt der realen Welt mit einem Datenobjekt. Dies ist üblicherweise eine Zeichenkette, die einer bestimmten Richtlinie folgt. Identifier könne für Menschen, Gruppen und beliebige andere Objekte stehen. Entscheidend dabei sind die Relationen zwischen verschiedenen Identifier Objekten.

1.3.2 Naming Services und Directories

Ein Dienst den jede Middleware anbietet, versieht die Entitäten mit eindeutigen Namen.
Ein Naming Service ähnlich dem DNS (Domain Naming Service) welcher semantische Namen in interne Bezeichnungen übersetzt oder vergleichbar mit einem Telefonbuch. Um die Entitäten auffinden zu können, werden Directory Services wie NDS (Network Directory Service) oder Microsoft Active Directory(tm) benötigt. Diese bieten eine allgemeinere Möglichkeit, beliebige Objekte aufzufinden.

1.3.3 Authentifikation

Die Authentifizierung (auch Authentifikation, engl. authentication) bezeichnet den Vorgang, die Identität einer Person oder eines Programms anhand eines bestimmten Merkmals zu überprüfen. Dies kann zum Beispiel mit einem Fingerabdruck, einem Passwort oder einem beliebigen anderen Berechtigungsnachweis geschehen.
Nah verwandt mit der Authentifizierung ist die Authentisierung. Die Authentisierung ist das Nachweisen einer Identität, die Authentifizierung deren Überprüfung. Im Englischen wird zwischen den beiden Begriffen nicht unterschieden, das Wort authentication steht für beides [Wik05a].

1.3.4 Autorisation und Zutrittsbeschränkung

Zutrittsbeschränkung (Access Control) bezeichnet die Regeln, nach denen entschieden wird, ob und wie Benutzer, Programme oder Programmteile, Operationen auf Objekten ausführen dürfen. Übliche Berechtigungen sind Read, Write, Update und Execute.

1.3.5 Zertifikate und PKI

Unter Private Key Infrastruktur (PKI) versteht man in der Kryptologie und Kryptografie ein System welches es ermöglicht, digitale Zertifikate auszustellen, zu verteilen und zu prüfen. Die innerhalb einer PKI ausgestellten Zertifikate sind meist auf Personen oder Maschinen festgelegt und werden zur Absicherung computergestützter Kommunikation verwendet [Wik05h].

1.3.6 Transaktionsüberwachung

Unter einer Transaktion versteht man eine feste Abfolge von Operationen, die entweder komplett oder gar nicht ausgeführt wird. Dieser Service dient zur Überwachung von Transaktionen um sicherzugehen, dass entweder die ganze Transaktion erfolgreich ist, oder gar keine Modifikation von Daten erfolgt. Dies ist entscheidend, wenn mehrere Datenbanken oder Systeme in eine Transaktion involviert sind.

1.3.7 Prozess und Thread Kontrolle

Bei Systemen, bei denen mehrere Threads beteiligt sind, wird oft ein Service zum Überwachen und Administrieren dieser Threads bereitgestellt. Vor allem bei synchroner Kommunikation, wie bei RPC, kann ein blockierender Thread die Funktionalität des gesamten Systems stark beeinflussen.

1.3.8 User und Debuginterface

Für die Wartung, Überwachung und Administration von Systemen werden meist Interfaces bereitgestellt, die Betriebsdaten und Systeminformationen in einem lesbaren Format bereitstellen.

1.3.9 Load Balancing

Unter Lastaufteilung versteht man das Aufteilen der verschiedenen Anfragen auf parallel arbeitende Systeme. Die Verteilung geschieht aufgrund bestimmter Vorschriften oder Regeln wie zum Beispiel eine Aufteilung aufgrund der Priorität oder des Anfragezeitpunktes. Lastaufteilung beinhaltet im weiteren Sinne auch Ausfallsicherheit.

1.3.10 Persistenz

Dieser Dienst ermöglicht es, beliebige Entitäten auf irgendeine Art und Weise abzuspeichern. Dies kann entweder über ein Dateisystem oder über eine Datenbank erfolgen.

1.3.11 Connectivity

Die Überbrückung unterschiedlicher Netzwerkprotokolle, Plattformen, Programmiersprachen, Betriebssystemen, Datenbanken und anderen Systemen ist ein wesentlicher Service von Middleware.

1.3.12 Datentransformation und Austausch

In verteilten Systemen ist oft ein Dienst erforderlich, der die Daten zwischen den verschiedenen Systemen übersetzt oder bei Bedarf manipuliert. Auch die Verifikation und Plausibilisierung, Formatierung, Aufteilung und Zusammenführung von Nachrichten und Daten fällt in den Aufgabenbereich dieses Dienstes.

1.3.13 Broker Service

Ein Broker empfängt Daten von verschiedenen Quellen und bestimmt die richtige Destination der Daten. Dieses Service agiert als intelligenter Router zwischen den beteiligten Komponenten, wobei die Funktionalität von den Teilnehmern verborgen bleibt.

1.3.14 Firewall

Untersucht die empfangenen Daten auf gewissen Sicherheitskriterien. Entsprechen die Daten nicht den Kriterien, werden diese ausgefiltert, bei Übereinstimmung mit den Sicherheitsrichtlinien wird die Erlaubnis zum Passieren dieses Dienstes erteilt.

1.3.15 Location Transparency

Die beteiligten Komponenten benötigen kein explizites Wissen über die Netzwerkadressen anderer Systeme. Dies ermöglicht zum Beispiel eine Applikation auf einem anderen Node zu starten, ohne dass das System in irgendeiner Weise rekonfiguriert werden muss.

1.3.16 Message Integrity

Überprüft die Integrität der Daten, um sicherzustellen, dass diese nicht verändert oder aufgrund von Übertragungsfehlern korrumpiert wurden. Ebenso wird überwacht, ob Daten verloren gehen oder mehrfach gesendet werden.

1.3.17 Dynamische Konfiguration

Um in einem Umfeld mit stark fluktuierenden Umgebungsbedingungen eine durchgehende Funktionalität bereitstellen zu können, ist es erforderlich, dass das System während des Betriebs auf die geänderten Attribute abgestimmt werden kann.

1.3.18 Quality of Service

Unter Quality of Service (QoS) versteht man die Dienstgüte in Kommunikationssystemen, welche aufgrund von verschiedenen Parametern, je nach eingesetztem Kommunikationsmedium, bestimmt wird. Übliche Parameter sind zeitliche Bestimmungen, Zuverlässigkeit, Genauigkeit usw.

1.3.19 Context-awareness

Durch eine globale Infrastruktur werden manche Dienste ortsunabhängig in Anspruch genommen. Diese entfernten Dienste können in einem anderen Kontext stehen, der von der Lokalität abhängt. Dadurch wird ein Dienst benötigt, der die Daten in einen globaleren Kontext übersetzt. Ein einfaches Beispiel sind Maßeinheiten, die je nach Land unterschiedlich sind.

1.4 Kategorisierung von Middleware

Es gibt viele Möglichkeiten Middleware in verschiedene Klassen zu unterteilen. Das Problem bei der Klassifizierung besteht darin, dass Middleware nicht ein einzelnes Service ist, sondern vielmehr eine Kombination aus mehreren Services. Es gibt weiters unzählige Middlewarelösungen mit verschiedenen Technologien als deren Grundlage. Um eine Klassifizierung vornehmen zu können, werden noch zusätzliche Unterscheidungskriterien benötigt. Eine anschauliche Unterteilung von Middleware ist in [TAB03] ersichtlich. Dabei werden die Middlewaresysteme zunächst in Implementierungen und Applikationen unterteilt, wie in der folgenden Grafik ersichtlich ist.

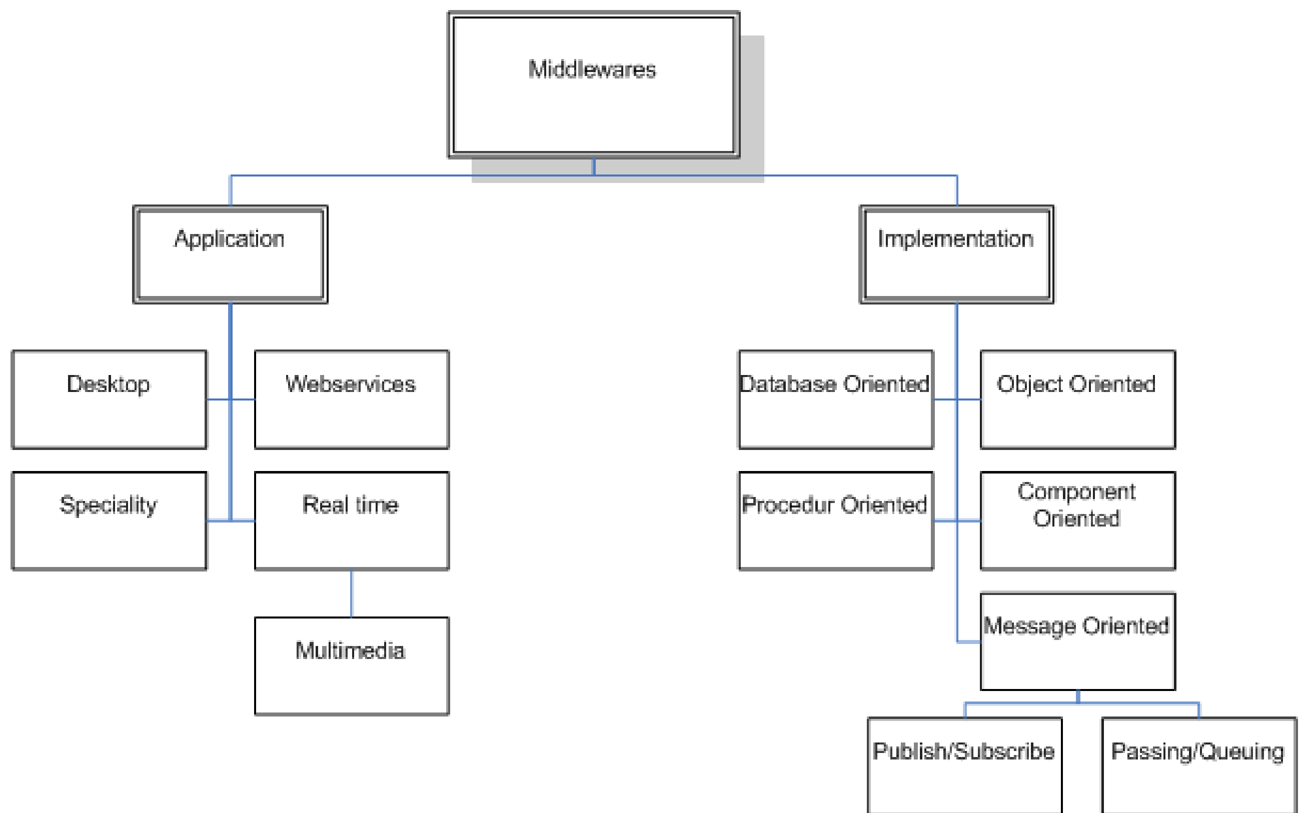

Abbildung 2: Kategorisierung von Middleware

Die Klassifizierung nach der Implementierungsart beinhaltet all jene Systeme, die einem grundlegenden Pattern oder einer grundlegenden Idee folgen. Sie unterscheiden sich untereinander durch verschiedene Kommunikationsprotokolle. Die Applikationskategorie beinhaltet Systeme, welche speziell mit einer Applikation arbeiten.
Für eine genauere Unterteilung von Middleware benötigt man noch weitere Unterscheidungskriterien.

1.4.1 Unterscheidung aufgrund der Komponenten

In einem fixen verteilten System sind die Komponenten örtlich fest angesiedelt, während im Gegensatz dazu in einem mobilen verteilten System die Geräte keinem bestimmten Ort zugeordnet werden können. Während fixe Geräte üblicherweise über mehr Ressourcen, wie Speicher und Prozessorkapazität, sowie einer schnellen Netzwerkanbindung verfügen, besitzen mobile Geräte wie PDA´s oder Handys im Allgemeinen nicht über dieses Ausmaß an Ressourcen. Abhängig davon, welche Zielsysteme eine Middleware bedienen soll, ergeben sich wesentliche Unterschiede in Funktion und Aufbau.

1.4.2 Unterscheidung aufgrund der Verbindung

Fixe Komponenten sind meist mit einer permanenten und breitbandigen Verbindung mit dem Netzwerk ausgestattet. Verbindungsunterbrechungen finden verhältnismäßig sehr selten statt und werden durch administrative Vorgänge oder unvorhersehbaren Fehlern verursacht. Bei mobilen Geräten werden Verbindungsabbrüche nicht als Ausnahme betrachtet, sondern gehören zum normalen Betrieb, ebenso wie starke Schwankungen bei der Bandbreite wenn Zonen mit einer schlechteren Infrastrukturabdeckung betreten werden. Die Middleware muss auf kurzlebige Verbindungen ausgelegt sein, um den Anforderungen gerecht zu werden.

1.4.3 Statischer oder dynamischer Kontext

Unter Kontext wird die Systemumgebung verstanden, welche die Funktionalität des Systems beinträchtigen kann. Dies beinhaltet interne Ressourcen der Clients sowohl externe Ressourcen wie Netzwerk und Bandbreite. In einem fixen System ist der Kontext mehr oder weniger statisch. Komponenten werden zwar hinzugefügt und entfernt, die Geräte ändern jedoch nur selten die Position und die Netzwerkanbindung ist nahezu konstant. Serviceanbieter

können einfach durch einen bekannten Service Provider aufgefunden werden. In mobilen Systemen unterliegen diese Umgebungsparameter jedoch starken Schwankungen. Das Auffinden von Services gestaltet sich als schwierig, vor allem wenn keine fixe Infrastruktur vorhanden ist.

1.4.4 Nicht funktionale Ansprüche

Die geforderte Performance eines Middlewaresysteme gehört zu den nicht-funktionalen Anforderungen. Die Hauptaufgabe von Middleware ist die Bereitstellung von Kommunikationsdiensten und Remoteservices. Dabei unterscheiden sich verschiedene Systeme anhand der Zuverlässigkeit, mit welcher diese Dienste angeboten werden. Es bedarf wesentlich höheren Anstrengungen ein Service mit sehr hoher Zuverlässigkeit anzubieten, als nach dem best-effort Prinzip, bei dem nicht sichergestellt ist ob die Anforderungen erfüllt werden oder nicht. Die best-effort Variante benötigt wesentlich weniger Ressourcen als ein hochgradig zuverlässiges System. Um ein fehlertolerantes Verhalten zu erreichen, setzen viele Middlewaresysteme auf Replikationsmechanismen. Die Synchronisierung der Replika benötigt üblicherweise eine Menge an Ressourcen (z.B. Netzwerktransfer). Aufgrund der benötigten Ressourcen einer Middleware lässt sich eine Kategorisierung durchführen. Es gibt Systeme, die einen hohen Bedarf an Ressourcen benötigen, um die Services bereitstellen zu können. Diese werden mit dem Term heavy-weight bezeichnet [MCE02]. Im Gegensatz dazu benötigen light-weight Systeme nur ein Minimum an Ressourcen um die geforderte Funktionalität bereitzustellen.

1.4.5 Prozedurbasierte Middleware (POM)

Bei einer prozedurbasierten Middleware konvertiert der Client die Parameter eines Prozeduraufrufes in ein Nachrichtenpaket. Dieser Vorgang wird als "*marschaling*" bezeichnet. Der Client sendet dieses Nachrichtenpaket an den Server, der die Parameter aus der Nachricht extrahiert, auch "*unmarshaling*" genannt. Nach dem *unmarshaling* der Nachricht führt der Server die angeforderte Prozedur mit den verfügbaren Parametern aus und verpackt die erhaltenen Ergebnisse, eventuelle Fehlermeldungen und Exceptions wiederum in eine Nachricht und schickt diese den Client zurück. Während der Server die Anfrage bearbeitet, bleibt der Client in einem blockierenden Zustand, bis dieser die Antwort auf die Anfrage empfängt. Die Vorteile dieser Vorgehensweise sind:

- Im Fehlerfall wird eine Nachricht mit dem Fehler oder der Exception gesendet.
- Verbirgt die Kommunikation durch einfache Methodenaufrufe
- Unterstützt unterschiedliche Datentypen
- Standardisiertes Naming Service

Es gibt jedoch auch eine Reihe von Nachteilen bei POM:

- Schlecht skalierbar, da RPC keinen Replikationsmechanismus unterstützt.
- Unflexibel durch die Kopplung an eine bestimmte Prozedur
- Benötigt eine zuverlässige Netzwerkanbindung
- Benötigt Multi-Threading

Bekannte Systeme, die prozedurbasiert sind, sind unter anderen Open Network Computing (ONC) von Sun Microsystems und Distributed Computing Environment (DCE) von der Open Software Foundation (OSF).

1.4.6 Datenbankorientierte Middleware (DOM)

Das wesentliche Erkennungsmerkmal von datenbankorientierten Systemen ist, dass die Interaktion der Anwendungen mit einer Lokalen- oder Remotedatenbank, einem Datawarehouse oder einer anderen Art von Datenquelle, wie z.B. Dateien, erfolgt. Die Datenbasis muss spezielle Anforderungen erfüllen, welche oft als ACID (Atomic, Consistent, Isolation, Durable) bezeichnet werden.

- Atomic: jede Transaktion wird als ganzes Ausgeführt
- Consistency: Die Konsistenz und die Relationen der Daten müssen korrekt sein
- Isolation: Eine Transaktion wird nie von einer anderen beeinflusst
- Durable: Die Auswirkungen einer Transaktion müssen im Datenbestand dauerhaft bestehen bleiben. Die Effekte von Transaktionen dürfen also nicht mit der Zeit verblassen oder aus Versehen verloren gehen [Wik06].

Die Anwendungen greifen mithilfe von speziellen Sprachen wie Application Programming Languages (APL) oder Structured Query Languages (SQL) auf den Datenbestand zu. Die Middleware sorgt für die Einhaltung der ACID Eigenschaften und leitet die Anfragen für die Datenoperationen, Modifikationen, Transaktionen und Datenlöschungen an die jeweils richtige Datenbank weiter. Da nicht sichergestellt werden kann, dass alle Datenbanken verfügbar sind, wird bei DOM ein Transaktionsmonitor benötigt. Dieser Dienst ist auch notwendig, wenn Transaktionen aufgrund eines Fehlers nicht ausführbar sind. Bekannte DOM Systeme sind Oracle Glue von Oracle Corporation, OLE-DB von Microsoft. Open Database Connectivity (ODBC) und Java Database Connectivity (JDBC) sind Standards für DOM jedoch nicht als DOM selbst zu sehen. Die Vorteile von DOM sind:

- Erlaubt die Kommunikation verschiedenster Datenquellen und Datenbanken
- Wandelt die userfreundliche APL in die jeweils von der Zieldatenbank geforderten Sprache um
- Stellt ein transparentes Interface unabhängig von der Plattform zur Verfügung
- Es ermöglicht die Verwendung von Stored Procedures
- Ermöglicht gleichzeitiges Absetzen mehrerer Anfragen

Die Schwachstellen von DOM sind:

- Sicherstellung das ACID Eigenschaften erfüllt werden ist teilweise schwierig zu realisieren
- Kommunikation erfolgt synchron: die Anwendung blockiert, bis die Antwort empfangen wird
- Transaktionsmanagement benötigt viel Ressourcen

1.4.7 Nachrichtenbasierte Middleware (MOM)

MOM (Message Oriented Middleware) charakterisiert sich durch die Eigenschaft, dass sämtlicher Datenaustausch zwischen den Applikationen auf Nachrichten basiert. Der Unterschied zu POM besteht darin, dass die Kommunikation asynchron erfolgt. Da die Nutzdaten für die Middleware nicht relevant sind ist kein marshaling und demarshaling wie bei POM notwendig. Es kommen dabei grundsätzlich zwei verschiedene Ansätze zum Einsatz: Message Queuing und Publish/Subscribe. Eine Message Queue ist eine Warteschleife für Nachrichten, welche unabhängig von den mit der Middleware verbundenen Anwendungen ist. Die Nachrichten warten in der Queue, bis das Netzwerk verfügbar ist, oder bis die Zielanwendung empfangsbereit ist. Message Queues können einen transaktionsorientierten Service bereitstellen, bei dem Nachrichten bei Fehlschlagen einer Transaktion nicht versendet werden. Bekannte MOM Produkte sind unter anderen TIB/Rendezvous von TIBCO, MessageQ ursprünglich von Digital Equipment Corp jetzt von BEA Systems, MSMQ von Microsoft, MQSeries von IBM, Java Message Queue von Sun Microsystems und Advanced Queuing von Oracle Corporation. Die Vorteile von MOM sind:

- Asynchrone Kommunikation mit store and forward Möglichkeiten, falls ein Zielclient nicht zum Zeitpunkt der Kommunikation verfügbar ist.
- Der zentrale Message Broker bietet meist eine Vielzahl an zusätzlichen Services wie Message Queuing, Message Warehouse, Message Transformation, Routing, Publish/Subscribe.
- Möglichkeit eines Persistenzmechanismus
- Kein Blockieren der Clients beim Warten auf eine Antwort
- Kein Multithreading notwendig

Die Nachteile von MOM sind:

- Asynchrone Kommunikation für zeitkritische Anwendungen nicht sehr gut geeignet
- Kein Security Konzept, vor allem bei Publish/Subscribe adressiert der Sender keinen Zielclient, sondern die Nachricht wird an eine Vielzahl an, für den Sender unbekannte, Clients versendet.

- Kein festgelegtes Nachrichtenformat, so kann es dazu führen, dass der Empfänger das Nachrichtenformat nicht verarbeiten kann. Es gibt jedoch Standards wie SOAP [Con05] die Richtlinien und Implementierungen bereitstellen.

Message Passing/Message Queuing Bei Message Passing Systemen gelangen die Anfragen von den Clients zunächst in eine Queue. Die Middleware wählt in einer bestimmten Reihenfolge die Anfragen aus und entfernt diese bei Beginn der Abarbeitung aus der Queue. Die Reihenfolge kann zum Beispiel nach einer first-in/first-out Strategie, nach einem Prioritätenschema oder aufgrund der Auslastung erfolgen (load balancing). Die MOM entscheidet, ob eine Nachricht verarbeitet, weitergeleitet oder verworfen wird. Bei Systemen mit einem Persistenzmechanismus werden die Nachrichten zusätzlich auf einem nichtflüchtigen Speicher festgehalten, bei nichtpersistenten Systemen befinden sich die Nachrichten lediglich im Arbeitsspeicher. Die MOM funktioniert wie ein Router und interagiert üblicherweise nicht mit den Nachrichten. Damit eine Anwendung andere Anwendungen auffinden kann, muss die Middleware ein Directory Service bereitstellen.
Bei Message Queuing Systemen fragen die Clients bei der MOM an, ob Nachrichten warten. Sind Nachrichten vorhanden, werden diese zum Client übertragen und serverseitig werden alle Kopien entfernt. Ein vergleichbares System ist das Mail System. Mails bleiben solange am Mailserver, bis der Anwender diese von Mailserver runterlädt. Nachrichten können auch gesendet werden, wenn der Empfänger zu der Zeit nicht verfügbar ist.

Publish/Subscribe Im Vergleich zu Message Queuing oder Message Parsing arbeitet ein Publish/Subscribe System etwas unterschiedlich. Publish/Subscribe Systeme sind eventbasierte Systeme. Möchte ein Client an der Kommunikation teilnehmen, so muss er sich beim System registrieren. Bei der Registrierung gibt der Client an, an welchen Daten er interessiert ist. Die Clients können entweder als Publisher oder als Subscriber mit dem System interagieren. Ein Publisher sendet Informationen zum System und ein Subscriber erhält die gewünschten Informationen automatisch, sofern diese verfügbar sind oder sich geändert haben.

1.4.8 Komponentenbasierte Middleware

Eine Komponente bezeichnet ein Programm, welches eine bestimmte Funktion erfüllt und auf eine Art und Weise realisiert ist, dass diese auf einfachen Weg mit anderen Komponenten und Anwendungen zusammenarbeiten kann. Die Middleware selbst ist mehr als eine spezielle Zusammenstellung einer Menge von Komponenten und deren Konfiguration zu sehen. Die Komponenten werden entweder beim Erstellen der Applikation oder erst zur Laufzeit in die Middleware aufgenommen. Bei der Erstellung solcher Systeme werden die Grundelemente von UML [Gro05] zur Hilfe genommen.

- Components: Programmeinheiten
- Connectors: die Verbindung zwischen den Components (Pipes, Client/Server Verbindungen)
- Configurations: Legt die Applikationsstruktur fest und kombiniert die Komponenten und Verbindungen zu einem System.

Bekannte Systeme sind unter anderen Open CORBA von OMG, .Net von Microsoft und J2EE von Sun Microsystems.
Die Vorteile sind:

- Sehr anpassungsfähig auf wechselnde Umwelteinflüsse, durch flexible Konfigurierbarkeit.
- Hoher Grad an Flexibilität, aufgrund der Komponenten
- Eingliederung neuer Funktionalität zur Laufzeit

Die Nachteile sind:

- Je größer das System, desto unübersichtlicher wird es. Es kann daher zu schwer nachvollziehbaren Abhängigkeiten zwischen den Komponenten kommen.
- Steigender Entwicklungsaufwand bei komplexen Datenstrukturen, da diese in die Interfaces mit aufgenommen werden müssen.

1.4.9 Objektorientierte Middleware (OOM)

Im Gegensatz zu Komponenten sind Objekte feinkörniger und alleine nicht funktionstüchtig. Objektorientierte Middleware kann als Weiterentwicklung von prozedurbasierter Middleware gesehen werden. Während der Client bei POM nur eine entfernte Methode aufrufen kann, ist es bei OOM möglich, ganze Objekte am Server zu kreieren. Das heißt, es stehen Daten und Methoden auf einem entfernten System zur Verfügung welche in einem Objekt gekapselt sind. Der Client erhält über einen Name oder Directoryservice von der Middleware lediglich eine Referenz auf die Objekte. Mithilfe dieser Referenzen können clientseitig die Methoden aufgerufen werden, unabhängig davon, wo sich die tatsächlichen Objekte befinden. Für die Übertragung selbst ist ein Object Request Broker Proxy (ORB - Proxy) verantwortlich. Die Vermittlung zwischen Clients und den Objekten übernimmt der Broker, der die Referenzen hält und auf die Objektquellen Zugriff hat. Bekannte OOM Systeme sind CORBA und COM/DCOM von Microsoft.
Die Stärken von OOM Systemen sind:

- Einfache Eingliederung von objektorientierten Datenbanken
- Unterstützung von Lastverteilung
- Ermöglicht die Speicherung unterschiedlichster Daten durch objektorientierten Ansatz
- Objekte können selbstorganisierend implementiert werden, sodass diese für die benötigten Ressourcen selbst verantwortlich sind. Objekte können andere Objekte von Ereignissen oder Aktivitäten informieren.
- Der Broker agiert als Vermittlungsstelle und sammelt Daten von verschiedenen Objekten.
- OOM Systeme können untereinander kommunizieren

Zu den Nachteilen zählen:

- Die Interfaces für die Objekte müssen vor der Laufzeit übersetzt werden.

1.4.10 Echtzeitorientierte Middleware

Echtzeitsysteme sind dadurch charakterisiert, dass die übertragenen Daten nur dann gültig sind, wenn diese bestimmten zeitlichen Kriterien folgen. Werden Daten zu spät empfangen, werden diese als ungültig betrachtet. Die Middleware stellt diese zeitkritischen Services zur Verfügung, die meist mittels strengen Scheduling Richtlinien realisiert werden. In Kombination mit echtzeitorientierter Middleware werden oft Echtzeitdatenbanken eingesetzt. Die Haupteinsatzgebiete sind Multimediaanwendungen wie Sprach- und Videoübertragung und in der Industrie, wo zeitkritische Anforderungen erfüllt werden müssen. Als Beispiel von echtzeitorientierter Middleware sind Realtime CORBA, TANGO, TMOSM zu nennen.

1.4.11 Web-Services

Laut W3C wird ein Web-Service folgend definiert:
"*A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards*" [Con04].
Ein Web Service ist eine Softwareanwendung, die mit einem Uniform Resource Identifier (URI) eindeutig identifizierbar ist und deren Schnittstellen als XML-Artefakte definiert, beschrieben und gefunden werden können. Ein Web Service unterstützt die direkte Interaktion mit anderen Softwareagenten unter Verwendung XML-basierter Nachrichten durch den Austausch über internetbasierte Protokolle. Web Services orientieren sich an der Service Oriented Architecture (SOA) [Wik05i] und vereinen daher verteilte und objektorientierte Programmierstandards und richten sich auf betriebswirtschaftliche Lösungen im Internet [Wik]. Es lassen sich die Instanzen Konsument, Anbieter und Verzeichnis identifizieren. Der Anbieter veröffentlicht in einem Verzeichnis die Beschreibung seiner Dienste. Der Konsument durchsucht das Verzeichnis und wählt den gewünschten Dienst aus. Nachdem eventuell weitere Protokolldetails ausgetauscht werden, findet die dynamische Anbindung des Konsumenten an den Anbieter statt. Der Konsument greift nun auf Methoden zurück. Die Grundlage hierbei bilden vier Standards, die jeweils auf XML basieren und in den zugehörigen Artikeln näher beschrieben werden:

- UDDI als Verzeichnisdienst zur Registrierung von Web Services. Es ermöglicht das dynamische Finden des Web Services durch den Konsumenten.
- WSDL zur Beschreibung der unterstützten Methoden und deren Parametern (z. B. Datum) für den Programmierer.
- SOAP (oder XML-RPC) zur Kommunikation. Hier wird der eigentliche Aufruf gestartet.

Erreichbar sind Web Services über einen eindeutigen URI. Die verwendeten plattformunabhängigen Standards sind in der Lage, entfernte Methodenaufrufe beliebiger Plattformen zu dekodieren und einer Anwendung weiterzuleiten. Auf diese Weise entsteht eine verteilte Architektur. Die Kommunikation mit Web Services erfolgt über Nachrichten, die über mehrere Protokolle transportiert werden können.
Die Vorteile von Webservices sind:

- Die verwendeten offenen Standards verringern die Lizenzkosten
- Web Services sind nicht an HTTP gebunden und lassen sich auch mit anderen Protokollen wie SMTP oder FTP übertragen und sind somit offen für verschiedene Anwendungsszenarien.
- Durch die Verwendung von bereits bestehenden und weit verbreiteten Internetstandards (HTTP, XML etc.) entsteht eine offene und flexible Architektur, die unabhängig von den verwendeten Plattformen, Programmiersprachen und Protokollen ist

Die Schwachpunkte sind:

- Durch die Verwendung von Internetinfrastruktur ist darauf zu achten, dass die Sicherheitsanforderungen eingehalten werden. So ist beim Transport zu beachten, dass wichtige Web Services verschlüsselt werden.
- Schlechte Performance durch Verwendung von XML wegen notwendigem Parsen und großem Datenoverhead.

1.5 Kopplung

Die verschiedenen Arten und Architekturen von verteilten Systemen unterscheiden sich voneinander durch unterschiedliche Levels der Kopplung der beteiligten Komponenten. Bei jeder Kommunikation zwischen zwei oder mehreren Komponenten lässt sich eine Kategorisierung aufgrund der Kopplung aufstellen, die wesentliche Eigenschaften über das System aufzeigt. Klassische Client/Server Systeme besitzen einen hohen Grad an Kopplung der Clients, durch die zunehmende Mobilisierung der Services besteht jedoch ein starker Trend hin zu Systemen die eine minimale Kopplung in allen Dimensionen besitzen. Schlagworte wie blocking, non blocking, directed, non directed, synchronous, asynchronous werden verwendet um den Grad der Kopplung anzugeben. In der Literatur wird zwischen drei Dimensionen der Kopplung unterschieden [LAH05].

- Kopplung in Bezug auf Raum: Die Adresse des Empfängers muss dem Sender nicht bekannt sein, es werden logische Namen verwendet.
- Kopplung in Bezug auf Zeit: Sender und Empfänger müssen bei der Übertragung einer Nachricht nicht gleichzeitig verfügbar sein.
- Kopplung in Bezug auf Synchronisation: Die Threads innerhalb der Beteiligten zeigen nicht blockierendes Verhalten bei der Übertragung.

1.5.1 Kopplung in Bezug auf Raum

Kopplung in Bezug auf Raum liegt dann vor, wenn ein Endpoint bei der Interaktion die direkte Adresse des empfangenden Endpoints verwendet. Somit benötigt der Sender ein Wissen über den Empfänger beziehungsweise wie und über welche Adresse dieser zu erreichen ist. Eine Interaktion zwischen zwei Endpoints findet also nur statt, wenn die Zieladresse (appID) senderseitig mit der Adresse des Empfängers übereinstimmt.

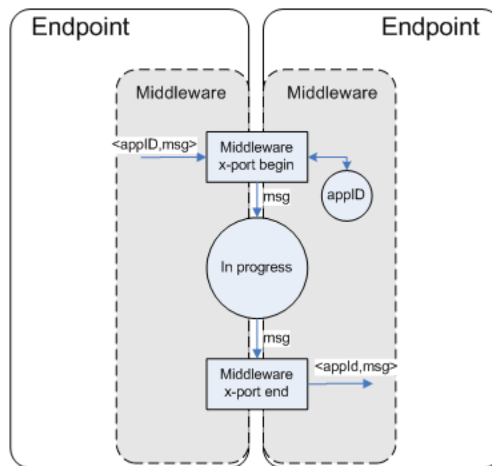

Abbildung 3: gekoppelte Übertragung im Bezug auf Raum; Die direkte Adresse (appID) ist notwendig für den Datenaustausch

Der sendende Teilnehmer benötigt wie in Abbildung 3 dargestellt die direkte Adresse des Empfängers (appID). Der eigentliche Datenaustausch findet zwischen x-port begin und x-port end statt.
Entkoppelt in Bezug auf Raum bedeutet, dass für eine Interaktion kein Wissen über den Empfänger notwendig ist. Diese Eigenschaft der räumlichen Entkoppelung macht es möglich ein verteiltes System zur Laufzeit zu erweitern. Um diese Entkoppelung zu erreichen, wird senderseitig eine abstrakte oder symbolische Adresse für den Ziel-Endpoint (cID) verwendet. Die Middleware dient als Vermittler oder agiert als Message Channel zwischen den beiden Endpoints. Wie in Abbildung 4 dargestellt, wird hier nur die symbolische Adresse (cID) zur Übertragung benötigt.

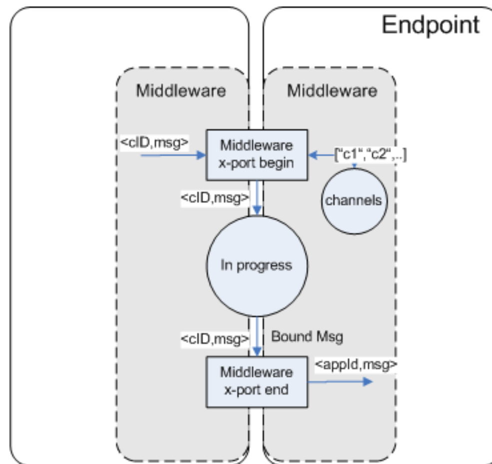

Abbildung 4: ungekoppelte Übertragung im Bezug auf Raum; Die Teilnehmer verwenden nur symbolische Adressen

Der Sender adressiert die Nachricht nur an einen bestimmten Channel, der mit chID identifiziert wird. Die Nachricht wird nur empfangen, wenn der Endpoint zu diesem Channel verbunden ist.

1.5.2 Kopplung in Bezug auf Zeit

Bei der Eigenschaft der zeitlichen Kopplung ergeben sich wesentliche Unterschiede zwischen den verschiedenen Systemen. Von zeitlicher Kopplung spricht man, wenn eine Interaktion zweier Endpoints erst stattfinden kann, wenn beide Endpoints zur gleichen Zeit aktiv sind. Somit unterliegen Peer-to-Peer Systeme immer einer zeitlichen Kopplung. Bei einer zeitlich gekoppelten Übertragung beginnt der Empfänger im selben Moment mit dem Empfangen der Nachricht wie der Sender mit dem Senden beginnt. Wie in Abbildung 5 ersichtlich müssen beide Endpoints zur selben Zeit an der Übertragung teilnehmen.

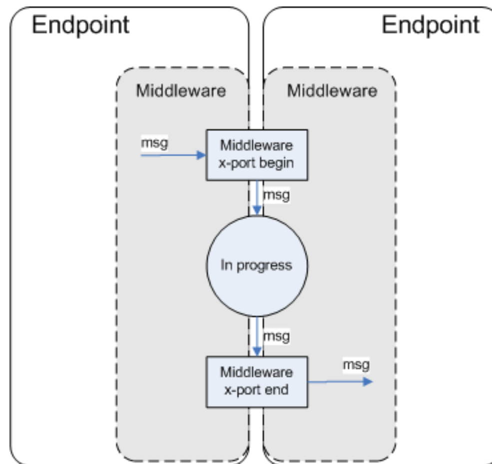

Abbildung 5: gekoppelte Übertragung im Bezug auf Zeit; Beide Endpoints müssen zur selben Zeit aktiv am Datenaustausch teilnehmen

Zeitliche Entkoppelung liegt vor, wenn die Interaktion, unabhängig davon ob ein Endpoint zur gleichen Zeit aktiv ist oder nicht, stattfinden kann. Bei Peer-to-Peer Systemen ist dies nicht der Fall, erst wenn ein dritter Teilnehmer mit der Fähigkeit Nachrichten zu speichern an der Interaktion teilnimmt. Der Sender deponiert die Nachricht bei diesem Teilnehmer und sobald der für den Empfang bestimmte Endpoint verfügbar ist, wird die Nachricht vom Teilnehmer, wo diese gespeichert ist, weitergeleitet. Dies ist der Grund, weshalb viele MOM Systeme eine zentralisierte Architektur verwenden. Wie in Abbildung 6 dargestellt speichert die Middleware die Daten solange, bis der Empfänger verfügbar ist es liegt somit zeitliche Entkoppelung vor.

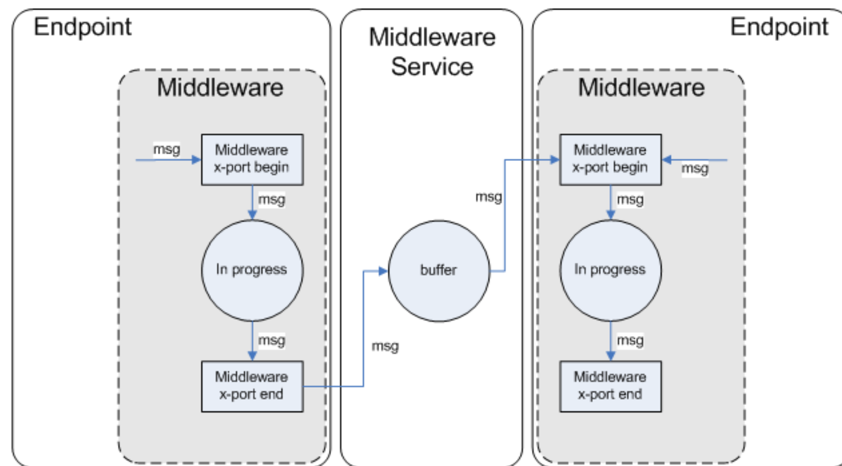

Abbildung 6: ungekoppelte Übertragung im Bezug auf Zeit; Die Daten werden von der Middleware zwischengespeichert bis der Empfänger empfangsbereit ist.

1.5.3 Kopplung in Bezug auf Synchronisation

Die wesentliche Eigenschaft von Kopplung in Bezug auf Synchronisation, beschreibt das Verhalten eines Endpoints beim Senden oder Empfangen einer Nachricht. Wird der Clientprozess durch die Kommunikation blockiert, so spricht man von einem blockierenden Verhalten.

Senden Die Operation des Sendens einer Nachricht kann entweder blockierend oder nicht blockierend erfolgen. Ein blockierendes Senden bedeutet, dass der sendende Client seine Verarbeitung stoppt, bis die Nachricht tatsächlich den Client verlassen hat. Ist eine Nachricht bereit zum Senden (Token msg-ready) und die Sendeapplikation bereit zum Senden (app control), so wird die Nachricht an die eingebettete Middleware übergeben. Der Endpoint übergibt beim blockierten Senden ebenfalls seinen Control Thread, welcher von der Middleware zurückgegeben wird (siehe Abbildung 7), sobald die Nachricht die clientseitige Middleware vollständig verlassen hat.
Die Übergänge x-port begin und x-port end symbolisieren die Grenze des Clients zum Empfänger oder zum Middleware Service. Hier erfolgt die eigentliche Übertragung der Nachricht in einer zeitlichen und synchronen gekoppelten Art und Weise, unabhängig vom Verhalten gegenüber dem Sender.

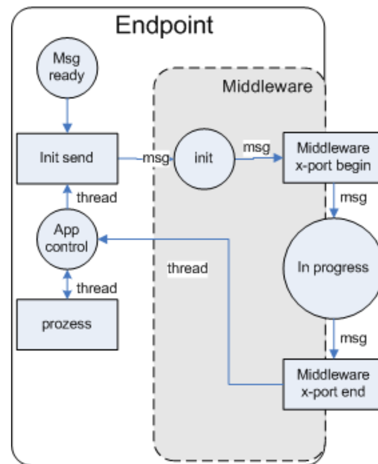

Abbildung 7: gekoppelte Übertragung im Bezug auf Synchronisation; Der Thread wird erst nach vollständiger Übertragung der Daten zurückgegeben.

Eine lose Kopplung wird aus der Sicht des Senders nur mit einem nicht blockierenden Senden erreicht. Ein nicht blockierendes Senden ist gegeben, wenn die eingebettete Middleware den Control Thread unmittelbar nach dem Übergeben der Nachricht wieder zurückgibt (siehe Abbildung 8), also bevor die Nachricht das System tatsächlich verlässt.
Um eine volle Entkopplung in Bezug auf Synchronisierung zu erhalten, ist ein nicht blockierendes Senden notwendig. Das ist jedoch noch nicht ausreichend. Die fehlende Eigenschaft ist ein nicht blockierendes Empfangen. Sobald Senden und Empfangen auf eine nicht blockierende Weise ablaufen, liegt eine Entkopplung in Bezug auf Synchronisation vor.
Eine teilweise Entkopplung liegt vor, wenn der Sender und Empfänger unterschiedliche Blockierungsmodi einsetzen.
Nicht blockierendes Senden ist in fast keiner Middlewarelösungen zu finden. Zum Beispiel sind jegliche RPC Aufrufe blockierend. Dies ist kein wesentlicher Nachteil bei Systemen die einem zuverlässigen Netzwerk zugrunde liegen. Ebenso ist die Kopplung meist nur zwischen den Clients und der Middleware und nicht zwischen den an der Interaktion beteiligten Clients. Bei mobilen Clients, welche auf unzuverlässigen Netzen kommunizieren, ist hingegen ein nicht blockierendes Senden notwendig. Der mobile Client muss die Möglichkeit besitzen, die Nachricht so lange lokal zu speichern, bis das Netz

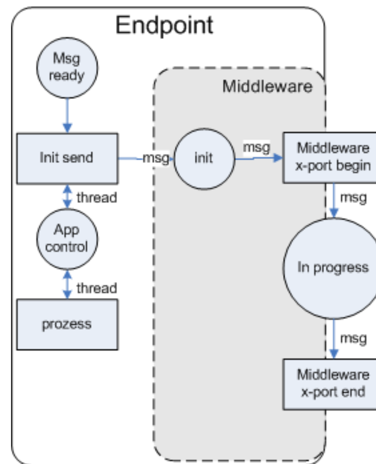

Abbildung 8: ungekoppelte Übertragung im Bezug auf Synchronistaion; Der Thread steht sofort wieder zur Verfügung, die Übertragung erfolgt nicht blockierend.

wieder verfügbar ist (Store and Forward) [KP03].

Empfangen Wie auch beim Senden kann das Empfangen von Daten blockierend oder nicht blockierend erfolgen. Beim blockierenden Empfangen wird ein Prozess in einen Wartezustand versetzt, bis die Nachricht vollständig empfangen ist. Das heißt, der Empfänger ist im Bezug auf Synchronisierung mit dem Sender oder dem Middleware Service gekoppelt.
Im Gegensatz zu blockierenden Empfang ist bei nicht blockierendem Empfangen keine Notwendigkeit einen Prozess in den Wartezustand zu versetzen. Ein Beispiel für nicht blockierenden Empfang ist das Konzept mit Event-Handlern. Ein Handler wird bei der Middleware registriert und wird nach dem Eintreffen einer Nachricht aufgerufen. Der Middleware muss kein Thread übergeben werden, um eine Nachricht zu empfangen.
Oft wird im Zusammenhang mit Message-Oriented-Middleware der Begriff asynchron genannt, welcher im Sinne der zeitlichen Entkopplung zu verstehen ist. Die Clients kommunizieren mit der Middleware in den meisten Fällen mit blockierendem Sende- und Empfangsmechanismen.

1.6 System Architektur

Ebenso wie Netzwerke die unterschiedlichsten Topologien aufweisen können, besitzen auch verteilte Systeme verschiedene Architekturen. Jede Architektur weißt bestimmte Vorteile oder Nachteile auf, und ist für bestimmte Anwendungsgebiete besser oder schlechter geeignet.
Jede Komponente agiert entweder als Server oder als Client. Ein Server empfängt Daten, speichert diese, bearbeitet diese und leitet sie wieder an die Clients weiter.

1.6.1 Zentralisierte Systeme

Zentralisierte Systeme sind die klassischen Server/Client Architekturen. Sie zählen jedoch ebenfalls zu den Peer-to-Peer Architekturen, da jeder Client eine eigene Verbindung zum Server herstellt. Bei zentralisierten Systemen gibt es einen zentralen Server, über den die gesamte Kommunikation abläuft. Die Clients können lediglich Dienste vom Server in Anspruch nehmen, jedoch nicht untereinander kommunizieren. Ein typisches Beispiel eines zentralisierten Systems ist ein einfacher Web-Server. Die Clients, in diesem Fall die Browser, verbinden sich zu einem zentralen Server und bedienen sich der bereitgestellten Dienste. Der Vorteil von zentralisierten Systemen ist, dass der Server bekannt ist, und eine Verbindung somit sehr einfach aufgebaut werden kann. Ein wesentlicher Nachteil besteht darin, dass der Server einen Flaschenhals darstellt und eine Skalierbarkeit des Systems nur schwer möglich ist.

1.6.2 Hierarchisch zentralisierte Systems

Bei einem hierarchischen System stehen mehrere Server in einer hierarchischen Relation zueinander. Jeder Server bedient eine Anzahl an Clients und ist weiters mit einem übergeordneten Server verbunden. Die Kommunikation zwischen Server und Clients läuft über das gleiche Protokoll wie die Server - Server Kommunikation. Ein Server unterscheidet also nicht zwischen Servern und Clients. Durch die hierarchische Struktur erreicht man ein gewisses Maß an Skalierbarkeit. Der Root Server empfängt alle Notifizierungen und Subscriptions von seinen Clients, leitet diese jedoch nur an den untergeordneten Teilbaum für den diese Nachricht bestimmt ist weiter. So wird allgemeiner Datentransfer von den untergeordneten Systemen abgehalten.

1.6.3 Vermittelte Systeme

Ein vermitteltes System (Brokered System) besteht ebenfalls aus einem zentralen Server, ermöglicht jedoch auch eine Kommunikation unter den Clients. Diese Kommunikation unter den Clients kann entweder über den Server laufen, oder auch direkt zwischen den Clients, wobei der Server die nötigen Daten für eine direkte Verbindung den Clients bereitstellt. Ein vermitteltes System verbindet die Vorteile eines zentralisierten Systems mit der Möglichkeit, dass Clients auch untereinander Daten austauschen können. Ein Beispiel für ein vermitteltes System ist Napster oder diverse Chats.

1.6.4 Dezentralisierte Systeme

Im Gegensatz zu den bisher genannten Topologien gibt es in dezentralisierten Systemen keine Server, die den Datenverkehr koordinieren. Der Datenverkehr wird über mehrere Clients aufgeteilt und es wird auf jegliche zentralisierte Infrastruktur verzichtet. Jeder verbundene Client ist gleichberechtigt und stellt Dienste für das Aufrechterhalten des Systems zur Verfügung. Zwei dezentralisierte Systeme können über einen Client verbunden werden, der zu beiden Systemen verbunden ist. Nachdem die beteiligten Komponenten von dem Verbindungsclient vorgestellt wurden, können mehrere Verbindungen zwischen den beiden Systemen aufgebaut werden. Ein Nachteil von solchen Systemen ist, dass eine Suche nach bestimmten Ressourcen viele Clients durchlaufen muss und somit lange dauert bzw. bei großen Systemen die Suche nicht alle Clients erfassen kann.

1.6.5 Super Peer Systeme

Super Peer Systeme sind eine Mischform aus vermittelten und dezentralen Systemen. Hat ein Client besonders günstige Umgebungsbedingungen (wie eine schnelle Internetanbindung) so agiert dieser Client als Supernode. Ein Supernode ist eine Art Broker innerhalb eines größeren dezentralen Systems. Jeder Client, auch Supernodes, verbinden sich mit anderen Supernodes.

1.7 Anwendungsintegration

Unter Integration versteht man das Vorhaben, eine Menge an Applikationen und Systemen so zu verbinden, dass diese zusammenarbeiten und eine vereinheitlichte Funktionalität zur Verfügung stellen. Die Schwierigkeit der In-

tegration liegt in der großen Vielfalt der verschiedenen Systeme. Meist laufen die Applikationen auf verschiedenen Rechnern, unterschiedlichen Plattformen und verschiedenen geografischen Orten. Manche Anwendungen sind spezielle Kundenlösungen, andere im Gegensatz standardisierte Produkte oder die Systeme sind nicht für eine Integration ausgelegt [HW03].
Daraus ergeben sich einige Kriterien, die bei der richtigen Auswahl einer Integrationslösung in Betracht gezogen werden sollen [Pep04].

1.7.1 Abhängigkeit

Vor allem bei integrierten Anwendungen sollen die Abhängigkeiten zu anderen Applikationen minimiert werden, damit jede Anwendung selbstständig ihre Funktionalität erfüllen kann, ohne Probleme bei anderen Anwendungen zu verursachen. Stark abhängige Applikationen machen viele Annahmen darüber, wie die anderen Applikationen arbeiten. Ändert sich nun eine dieser Anwendungen und verletzt die gemachten Annahmen, ist keine Integrität mehr gegeben. Die Schnittstellen zwischen zwei Applikationen sollen speziell genug sein, um die notwendige Funktionalität gewährleisten zu können und allgemein genug, damit sich die Implementierung einzelner Applikationen ändern kann.

1.7.2 Einfachheit der Integration

Beim Integrieren einer Applikation sollen möglichst wenige Änderungen der Applikation und möglichst wenig Integrationscode erforderlich sein. Jedoch bietet die Lösung mit den geringsten Auswirkungen auf die Anwendung nicht immer das beste Integrationsergebnis.

1.7.3 Integrationstechnik

Verschiedene Lösungen erfordern verschiedene Software bezeihungsweise verschiedene Infrastruktur. Diese Anforderungen können mit hohen Kosten verbunden sein und führen eventuell zu Abhängigkeiten an bestimmte Produkte oder Hersteller.

1.7.4 Dateneigenschaften

Da verschiedene integrierte Anwendungen möglicherweise unterschiedliche Datenformate benötigen, müssen Konverter eingesetzt werden, um die nö-

tige Datenkompatibilität gewährleisten zu können. Interessant ist dabei wie sich die Datenformate mit der Zeit ändern und welche Auswirkungen dies auf die einzelnen Applikationen hat.
Eine Integration soll auch die Dauer minimieren, die benötigt wird, um Daten von einem Teilsystem zu einem anderen zu senden. Je länger die Verzögerungen sind, desto wahrscheinlicher ist es, dass diese Daten bereits nicht mehr gültig sind. Die Komplexität und die benötigte Logik für die Validierung wachsen somit.

1.7.5 Funktionsaufrufe

Nicht immer werden nur Daten zwischen den einzelnen Anwendungen ausgetauscht, sondern es werden auch Funktionen in anderen Systemen angestoßen. Solche Aufrufe externer Funktionen erfordern komplexe Abläufe und können auf die Gesamtfunktionalität der Integration Einfluss nehmen.
Es gibt viele verschiedene Ansätze um Integration zu erreichen. Die meisten können in einen von vier grundlegenden Vorgehensweisen eingeordnet werden.

- Filetransfer: Jede Applikation erstellt Files mit den Daten, die für andere Anwendungen zugänglich sind, und importiert Files mit den Daten anderer Anwendungen.
- Gemeinsame Datenbank: Jede Applikation speichert die Daten in eine öffentliche Datenbank.
- Externe Funktionsaufrufe: Jede Anwendung stellt öffentliche Funktionen zur Verfügung, die von anderen Anwendungen ausgeführt werden können.
- Nachrichtenaustausch: Jede Anwendung ist mit einem Nachrichtensystem verbunden. Über Nachrichten werden Daten ausgetauscht und bestimmte Systemverhalten realisiert.

Filetransfer und gemeinsame Datenbank beschränken sich auf das Teilen von Daten, nicht jedoch auf das Teilen von Funktionalität. Filetransfer bietet eine geringe Abhängigkeit, birgt jedoch zeitliche Nachteile. Bei gemeinsamen Datenbanken ist jede Anwendung auf diese Datenbank angewiesen. Externe Funktionsaufrufe wiederum stellen nur Funktionalität zur Verfügung und

schaffen auch eine sehr enge Eingliederung in den Prozessablauf und somit eine hohe Abhängigkeit.
Benötigt wird ein System, dass vom Prinzip her wie ein Filetransfer arbeitet, jedoch viele Datenpakete in kurzer Zeit versenden kann und der Empfänger beim Empfang eines neuen Paketes verständigt wird. Das Filesystem oder die Datenbankstruktur soll von der Anwendung verborgen bleiben damit, im Gegensatz zu einer gemeinsamen Datenbank, eine Änderung der Datenschemas ohne Auswirkungen bleibt und jederzeit an die Anforderungen angepasst werden kann. Eine Anwendung soll über ein gesendetes Paket Funktionalität in einer anderen Anwendung wie bei einem externen Funktionsaufruf auslösen können, jedoch ohne die dabei entstehende enge Abhängigkeit. Der Datenaustausch soll asynchron erfolgen, somit braucht der Sender nicht auf den Empfänger zu warten.

2 Nachrichtenbasierte Middleware (MOM)

2.1 Definition

Eine Middleware heißt genau dann nachrichtenorientierte Middleware (MOM), wenn die Kommunikation zwischen den beteiligten Komponenten durch den Austausch von Nachrichten über eine Zwischeninstanz erfolgt [SH05].

- Endpoint: Ein Endpoint ist eine Komponente, die mit anderen Komponenten interagiert und die Fähigkeit besitzt, Nachrichten zu senden und/oder zu empfangen.
- Interaktion: Unter Interaktion verstehet man den Informationsaustausch zwischen zwei Endpoints
- Message Channel: Ein Endpoint versendet bei der Interaktion die Daten über einen Message Channel, einer virtuellen Verbindung, welche einen Sender mit einem Empfänger verbindet. Der Sender selbst muss nicht wissen, welcher Client sich am anderen Ende des Channels befindet. Es werden lediglich die Nachrichten an den entsprechenden Channel geleitet. Ein Channel ist vielmehr als eine Art Container für Daten zu sehen, in dem eine Anwendung Daten platzieren kann und andere Anwendungen diese wieder herausnehmen können. Die Unterscheidung,

ob ein Channel unidirektional oder bidirektional ist, ergibt sich bei näherer Betrachtung. Ein bidirektionaler Channel würde heißen, dass der Sender auch die selbst gesendeten Nachrichten konsumieren würde, die eigentlich für andere Anwendungen bestimmt sind. Da eine Nachricht von einer Anwendung zu einer anderen übertragen wird, folgt daraus, dass der Channel unidirektional ist.

- Multi-Step-Delivery: Im einfachsten Fall werden Nachrichten direkt vom Versender zum Empfänger übertragen. In manchen Anwendungen benötigt eine Nachricht zusätzliche Bearbeitung, nachdem sie vom versendenden Teilnehmer versendet wurde und bevor sie vom empfangenden Teilnehmer empfangen wird. Ein Beispiel für so einen Fall wäre eine notwendige Transformation der Daten in ein anderes Format.

- Routing: In großen Umgebungen mit unzähligen Anwendungen und Channels, welche diese miteinander verbinden, kann es sein, dass Nachrichten durch eine Vielzahl an Channels geleitet werden müssen, bevor diese ihre endgültige Zieladresse erreichen. Ist diese Route durch die Channels so komplex, dass der Sender nicht weiß, wie die Zielstation erreichbar ist, ist ein Router erforderlich. Der Sender sendet die Nachricht an den Router, der diese an den richtigen Channel weiterleitet.

- Transformation: Erwarten einige Anwendungen für dieselben Daten verschiedene Formate, so ist eine Transformation dieser Formate notwendig. Dazu ist ein Nachrichtentransformator oder Konverter nötig, der die Daten von einem Format in ein anderes überführt.

2.1.1 Bezeichnungen

Es gibt viele Ausdrücke, wie eine Anwendung genannt wird, wenn diese über einen Message Channel kommuniziert. Die allgemeinste Form ist wohl die Bezeichnung Sender und Empfänger. Eine Anwendung sendet eine Nachricht an einen Nachrichtenkanal, um von einem Empfänger einer anderen Anwendung empfangen werden zu können.
Eine weitere Bezeichnung ist Producer und Consumer oder auch Publisher und Subscriber (wird meist in Zusammenhang mit Publish/Subscribe Systemen verwendet, aber teilweise auch als allgemeine Bezeichnung). Oft findet man auch den Ausdruck eine Anwendung horcht (listener) auf einem Kanal oder spricht (talker) zu einem Kanal. In Zusammenhang mit Web-services

verwendet man Provider und Requester was impliziert, dass der Requester eine Nachricht an den Provider sendet und dann vom Provider eine Antwort bekommt. Manchmal wird die sendende Applikation auch als Consumer eines Services bezeichnet. Der Consumer sendet eine Nachricht zum Provider und konsumiert (consumes) dann die Antwort vom Provider. Auch die Terme Client und Server werden häufig verwendet.

2.2 Nachrichten

Eine Nachricht oder Message ist ein Paket von Daten, die über einen Channel übertragen werden können. Um die Daten zu übertragen, muss eine Anwendung die Daten in verschiedene Pakete unterteilen, diese in Nachrichten einpacken und über einen Nachrichtenkanal versenden. Der Empfänger muss die Daten aus der Nachricht extrahieren, bevor er diese verarbeiten kann. Eine Nachricht besteht aus zwei grundlegenden Teilen, dem Nachrichtenkopf oder Messageheader und dem Nachrichtenkörper oder Messagebody.

- Der Nachrichtenkopf enthält Daten für das Nachrichtensystem wie Sender, Empfänger, Art der Daten, Länge usw.
- Der Nachrichtenkörper enthält die eigentlichen Nutzdaten die übertragen werden sollen. Üblicherweise werden diese Daten vom Nachrichtensystem ignoriert und einfach so übertragen, wie sie sind.

Für das Nachrichtensystem sind alle Nachrichten gleich zu handhaben. Die Nachrichten werden aufgrund ihrer Daten im Header vom Nachrichtensystem verarbeitet. Anders ist es auf der Anwendungsseite, wo verschiedene Nachrichtentypen unterschieden werden können. Diese Unterscheidung erfolgt anwendungsspezifisch und kann zum Beispiel folgende Typen beinhalten:

- Command-Message: Nachricht, um eine Funktion in einer anderen Anwendung auszuführen oder um einen zusätzlichen Service des Systems in Anspruch zu nehmen.
- Document-Message: Nachricht, um Daten an eine andere Anwendung oder zu einem anderen Client zu senden.
- Event-Message: Nachricht, um eine Anwendung oder einen Client auf ein eingetroffenes Ereignis aufmerksam zu machen.

- Request-Reply: Nachricht, um eine Antwort von einer Anwendung zu beantragen
- Message-Sequence: Ist eine Nachricht zu klein für das Senden der gewünschten Datenmenge, so können die Daten aufgespaltet werden und mit einer Sequenz von Nachrichten versendet werden.

Es gibt noch unzählige andere Attribute für Nachrichten, die stark vom Kommunikationssystem abhängig sind. Siehe dazu die Definition von gängigen Nachrichten [Mic05d, Mic05b, Con05].

2.3 Typen von Message - Channels

Grundsätzlich unterscheidet man zwei Arten von Message Channels. Zum einen den One-to-One Channel und zum anderen den One-to-Many Channel. Werden die Daten zu einem einzelnen Client versendet, so spricht man von einem One-to-One oder Point-to-Point Channel. Dieser One-to-One Channel garantiert jedoch nicht, dass alle Datenpakete die an den Channel gesendet werden beim gleichen Empfänger ankommen, da ein Channel mehrere verschiedene Empfänger besitzen kann. Es wird jedoch garantiert, dass jedes Datenpaket genau zu einem dieser Empfänger gelangt. Sollen alle Empfänger diese Nachricht erhalten, so spricht man von einem One-to-Many oder einem Publish/Subscribe Channel. Wird eine Nachricht über einen Publish/Subscribe Channel gesendet, so erzeugt der Channel intern für jeden Empfänger eine Kopie der Nachricht und sendet diese an den Empfänger.

2.3.1 Point-to-Point Channel

Bei Nachrichtensystemen werden alle Daten in Nachrichtenform an einen Channel gesendet.
Möglicherweise gibt es eine Vielzahl an Clients, die auf diesen Channel horchen. Nehmen wir an, bei den Daten handelt es sich um Kommandonachrichten zum Starten einer externen Funktion, so kann es geschehen, dass mehrere Clients aufgrund der Nachricht entscheiden die Prozedur zu starten. Um dem entgegenzuwirken und um zu erreichen, dass nur einer dieser Clients die gewünschte Funktionalität ausführt, kann das Nachrichtensystem von vornherein verhindern, dass sich mehrere Clients an einen Channel anmelden. Dies jedoch würde die Möglichkeiten der Sender beim Versenden von Daten an mehrere Clients einschränken. Es kann durchaus erwünscht

sein, dass mehrere Clients gleichzeitig auf die im Message Channel verfügbaren Nachrichten zugreifen. Nur einzelne Nachrichten sollen nur an einen der Clients gelangen. Weiters wäre eine Koordination der Clients denkbar, dies würde jedoch einen großen Kommunikationsoverhead erzeugen und eine Art Abhängigkeit zwischen Clients, die normalerweise voneinander unabhängig sind.
Ein Point-to-Point Channel stellt genau diese Eigenschaft sicher, dass eine Nachricht an genau einen Empfänger gelangt. Versuchen mehrere Clients auf dieselbe Nachricht zuzugreifen, so ist der Zugriff nur für einen Client erfolgreich.
Bei nur einem Empfänger ist das Verhalten einfach, sind mehrere Empfänger mit einem Channel verbunden so werden diese zu konkurrierenden Empfängern.

Konkurrierende Empfänger Die Abarbeitung der Nachrichten erfolgt üblicherweise in der Reihenfolge des Eintreffens. Ist nur ein Client mit einem Channel verbunden und benötigt dieser mehr Zeit zum Abarbeiten als die Eingangsfrequenz der Nachrichten beträgt, so stauen sich Nachrichten im Channel auf. Das Nachrichtensystem wird sozusagen zu einem Flaschenhals, der den Durchsatz der Gesamtapplikation herabsetzt. Ein weiterer Channel bringt nicht notwendigerweise eine Lösung, höchstens eine geringfügige Reduktion des Problems, da auf einem Channel nach wie vor ein Engpass vorliegen kann, während ein anderer Channel nur sehr wenig Transferaufkommen hat. Konkurrierende Empfänger sind eine Gruppe von Empfänger, welche zu einem einzelnen Point-to-Point Channel verbunden sind. Das Nachrichtensystem entscheidet indirekt, welcher der Clients die Nachricht konsumiert, die Clients stehen jedoch in einer Art Konkurrenz zueinander. Empfängt ein Client eine Nachricht, so beginnt dieser die empfangenen Daten zu verarbeiten und delegiert den Empfang der nächsten Nachricht indirekt an die verbleibenden Clients. Die Clients benötigen keinerlei Koordination untereinander, dies geschieht rein über die Funktionalität des Point-to-Point Channels. Dies funktioniert nicht mit einem Publish-Subscribe Channel, da hier jede Nachricht als Kopie an alle verbundenen Clients weitergeleitet wird.

2.3.2 Publish Subscribe Channel

Jeder Subscriber oder Client, der zum Channel verbunden ist, erhält jede an den Channel gesendete Nachricht genau einmal. Eine Nachricht gilt erst dann

als konsumiert, wenn alle verbundenen Clients diese empfangen haben und wird zu diesem Zeitpunkt aus dem Channel entfernt. Ein Publish-Subscribe Channel besteht aus einem Eingangschannel und einer Menge an Ausgangschannels. Jeder Ausgangschannel ist mit genau einem Subscriber verbunden. Der Publish-Subscribe Channel sendet an jeden Ausgangschannel eine Kopie der Nachricht, so wird sichergestellt, dass jeder Subscriber jede Nachricht genau einmal zugestellt bekommt.
Diese Vorgehensweise bringt Sicherheitsrisiken mit sich, da das Mitlauschen am Channel nicht so leicht feststellbar ist wie bei Point-to-Point Verbindungen, da die Nachricht bei anderen Clients nicht abgeht, was beim Point-to-Point Channel der Fall wäre. Durch Richtlinien beim Anmelden am Channel kann dem jedoch entgegengewirkt werden.

2.4 Publish/Subscribe Paradigma

Ein Publish/Subscribe System besteht mindestens aus folgenden Elementen. Publisher oder Producer und Subscriber oder Consumer als die miteinander interagierenden Komponenten, Ereignisse und Notifizierungen als Daten, die zwischen den Publishern und Subscribern ausgetauscht werden sowie Subscriptions als Indikatoren, dass ein bestimmter Subscriber an bestimmten Notifizierungen interessiert ist. Als Koordinator zwischen Publishern und Subscribern steht das Nachrichtensystem selbst, welches anhand von den Subscriptions die richtigen Events an die richtigen Subscriber weiterleitet. Das Nachrichtensystem wird im Publish/Subscribe Kontext auch oft als Broker oder Event-Notification Service bezeichnet [Zei04].

2.4.1 Events und Notifizierungen

Unter einem Event versteht man im Allgemeinen ein beobachtbares Eintreten eines Ereignisses, welches von Interesse für den Systemkontext ist. Dieses Ereignis kann sowohl physikalischer als auch virtueller Natur sein. Das beobachtende Organ leitet in einem komplexeren Umfeld üblicherweise das Event an eine höhere Ebene weiter, wo systemabhängige Reaktionen auf das Eintreten des Ereignisses gesetzt werden. Ereignisse können von allen Ebenen eines Systems stammen, angefangen von low-level Hardware Events wie Interrupts bis zu high Level Business Events in E-Commerce Lösungen. Ein Event oder Ereignis wird in einem Publish/Subscribe System durch eine Notifizierung abgebildet. Eine Notifizierung beinhaltet die Daten, die das Eintreten eines

	Beschreibung
Ereignis(Event)	Beobachtung eines bestimmten Vorkommnises
Verständigung(Notification)	Repräsentation eines Events zur Weiterverarbeitung in einem IT-System. Ein Event kann mehrere Notifizierungen auslösen
Nachricht(Message)	Datencontainer, um eine Notifizierung im System zu transportieren

Tabelle 1: Elemente von Publish/Subscribe

Ereignisses beschreiben. Eine Notifizierung wird vom beobachtenden Organ erstellt und kann je nach System nur über das Eintreten des Ereignisses oder auch weitere Daten wie das Zustandekommen des Ereignisses beinhalten. Man kann nicht davon ausgehen, dass ein Ereignis mit genau einem Event in Verbindung steht, vielmehr kann ein Ereignis eine ganze Abfolge von Events auslösen. Die Notifizierungen werden mithilfe von Nachrichten weitergeleitet [Zei04].

2.4.2 Publisher und Subscriber

Die beteiligten Clients eines ereigbnisbasierten Systems agieren als Producer und/oder Consumer von Notifizierungen. Producer emittieren Notifizierungen bei bestimmten Ereignissen. Clients sind üblicherweise intelligente geschlossene Systeme, das heißt nicht jedes Ereignis wird notwendigerweise weitergeleitet. Je nach Client können durch auch higher-level Events generiert werden in die bereits anwendungsabhängige Logik einfließt.
Wenn ein (Publisher) eine Notifizierung absetzen möchte, so publiziert (published) er diese einfach an das Nachrichtensystem. Der Client weiß jedoch nicht Bescheid darüber, wie das Gesamtsystem auf diese Notifizierung reagiert. Ebenfalls weiß der Client nicht, welche anderen Clients diese Nachricht erhalten. Diese Entkopplung ist ein wesentlicher Vorteil von Publish/Subscribe Systemen. Nach dem Versenden der Notifizierung ist das Nachrichtensystem für die richtige und zuverlässige Verteilung an alle interessierten Clients der Nachricht zuständig. Auf der anderen Kommunikationsseite stehen die Empfänger oder im Kontext von Publish/Subscribe die

Subscriber. Als Grundlage fürs Empfangen von Notifizierungen muss ein Subscriber im Vorfeld beim Nachrichtensystem sein Interesse an spezifischen Notifizierungen in Form einer Subscription bekannt geben. Ab diesem Zeitpunkt erhält der Client die abonnierten Notifizierungen und reagiert auf diese. Ein Client kann sowohl als Publisher als auch als Subscriber mit dem System interagieren [Zei04].

2.4.3 Subscriptions

Eine Subscription repräsentiert das Interesse an einer Untermenge von Notifizierungen. Clients, die an einer Gruppe von Notifizierungen interessiert sind, registrieren sich beim Notifizierungssystem mit einer Subscription für den Empfang dieser Notifizierung. Das Nachrichtensystem filtert aufgrund der vorliegenden Subscriptions die gewünschten Notifizierungen, unter allen eingehenden Notifizierungen, für den jeweiligen Client aus. Die Entscheidung ob eine Nachricht einer Subscription genügt erfolgt boolesch, entweder eine Notifizierung erfüllt die Kriterien einer Subscription oder nicht. Allgemein kann eine Subscription mehrere Kriterien beinhalten, die nicht notwendigerweise nur auf die Nutzdaten beschränkt sind. So können ebenfalls Meta-Daten dazu benutzt werden den Nachrichtenfluss zu beeinflussen. Für die Art der Filterung gibt es je nach gewünschter Anwendung einige verschiedene Ansätze, die sich wesentlich voneinander unterscheiden.

Channel Die einfachste Form besteht darin, die Notifizierungen nicht auszufiltern, sondern aufgrund der Channels eine fixe Subscription abzuleiten. Dabei erhält jeder Client der zu einem Channel verbunden ist, alle Notifizierungen die auf diesem veröffentlicht werden, unabhängig vom eigentlichem Interesse des Clients. Der Client hat also nur die Wahl zu welchem Channel er sich verbindet, die damit verbundenen Subscriptions ergeben sich aufgrund des Channels. Eine Implementierung dieser Art ist zum Beispiel das CORBA Event Service [Gro]. Das Corba Notification Service [Gro04] baut ebenfalls auf Channels auf, bietet jedoch noch zusätzliche Filtermöglichkeiten.

Topic, Group, Subject Bei Topic basierten Subscriptions werden die Notifizierungen über ein Topic kategorisiert. Das Topic oder Subject dient in der Applikation als eindeutiger Identifier, welcher einer Notifizierung zugeordnet wird. Dieser Identifier ist üblicherweise als Zeichenkette mit einer für die Anwendung sprechender Bezeichnung realisiert und muss a priori konfiguriert

werden. Jede Notifizierung ist einem dieser Topics zugeordnet. Man kann sozusagen jedes Topic als eigenes Eventservice mit eindeutigem Namen betrachten. Die Clients subscriben sich zu den Topics, an denen sie Interesse haben. Ein Publisher muss die Notifizierung mit dem entsprechenden Topic markieren damit das Notifizierungssystem diese Nachricht mit den registrierten Subscriptions vergleichen kann, um die Nachricht an die richtigen Clients weiterzuleiten. Die Filterung selbst erfolgt über einen einfachen Vergleich des Topics der Notifizierung und der, von den Clients registrierten Subscriptions. In diesem Zusammenhang spricht man anstatt von einem Topic auch oft von einer Group. Diese Bezeichnung kommt von den ersten Publish/Subscribe Systemen, die auf group communication [Pow96] basieren. Das Subscriben zu einem Topic kann auch als Registrierung als Mitglied einer Gruppe aufgefasst werden und das publishen als Broadcast an alle Mitglieder der Gruppe. Eine nützliche Erweiterung zu einem Topic basierten System ist die Einführung verschiedener Hierarchien. Hiermit heben sich Topic basierte Systeme von Group-based Systeme ab, da bei Group based keine Hierarchien möglich sind. Die unterschiedlichen Hierarchien werden meist über eine URI oder Pfad ähnliche Notation angegeben. Zum Beispiel

“Schicht1/Fertigungsstrasse1/Maschine1/Modul2/Ventil3” oder

“Schicht1/Fertigungsstrasse1/Maschine3/Modul2/Ventil3”

Jede Unterhierarchie verkleinert die Anzahl, der ihr zugeordneten Notifizierungen. Je nach System kann der Client auch mit Wildcards Notifizierungen subscriben wie etwa

“Schicht1/Fertigungsstrasse1/*/Modul2/Ventil3”

oder alle Notifizierungen in einer Hierarchie inklusive aller Unterhierarchien wie etwa

“Schicht1/FertigungsstrasseA”

Type Oft besteht nur Interesse am Objekttyp oder Datentyp der Notifizierung. Type-based Subscription bezieht sich auf den Typ einer Notifizierung. Jede Notifizierung wird als Objekt eines bestimmten Typs betrachtet, ähnlich wie objektorientierte Klassen in Java. Multiple Vererbungen sind dabei wesentlicher Bestandteil. Die Filterung an sich geschieht über eine einfache Typ Prüfung [Com05a, EFGK03].

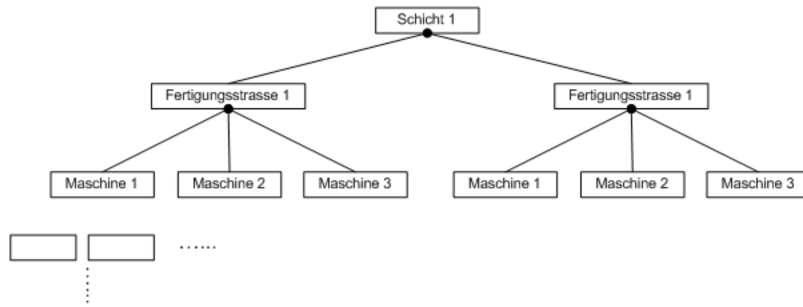

Abbildung 9: hierarchiesche Topicstruktur

Name	Type	Erforderlich	Anzahl der Werte
Eventname	String	Ja	1
Preis	Float	Ja	1
Volumen	Integer	Nein	1
Emission	String	Ja	1

Tabelle 2: Event Schema

Content Content-based Subscription ist ein sehr universeller und allgemeiner Ansatz um Notifizierungen zu filtern. Während alle vorherigen Methoden auf eine vorher definierte Eigenschaft einer Notifizierung zugreifen (Topic oder Type), verwendet Content-based Subscription den gesamten Inhalt der Nachricht als mögliches Filterkriterium [ASS+99]. Die Filterfunktionen reichen von template matching [CDF01], einfachen Vergleichen [CRW01], komplexen Filterausdrücken für Key/Value Paare [MF01] bis zu XPath für XML [AF00].

Value/Key Paare Das content-based System unterstützt eine Menge an Information Spaces. Jedem Information Space wird ein Event Schema zugeordnet. Ein Event Schema besteht aus mehreren Attributen. Nehmen wir als kleines Beispiel ein Wertpapier Trading System. Das vereinfachte Event Schema würde etwa folgenderweise aussehen.
Eine gültige Subscription ist ein Prädikat über die gegebenen Attribute wie (Eventname=changedStock;Preis<120;Volumen>1000;Emission=CA).
Dadurch wird eine gezielte Auswahl der Notifizierungen ermöglicht, ohne

dem administrativen Aufwand neue Topics zu deklarieren. Bei einem topic-basierten System hätte jeder Wertpapiertitel ein eigenes Topic, hier reicht ein Information Space mit dem dazugehörigen Event Schema. Der Client hat die zusätzliche Möglichkeit Notifizierungen aufgrund von Kriterien auszusortieren. Die Kriterien können in verschiedener Weise implementiert sein.

- Strings: Meistens wird das Filterkriterium als Zeichenkette dem Notifizierungssystem übergeben. Dabei müssen diese Kriterien einer Grammatik folgen und bilden somit eine Art eigene Sprache.

- Template Object: Der Subscriber verwendet ein Template Objekt, und meldet somit beim Notifizierungssystem das Interesse für alle Notifizierungen vom selben Typ wie das Template Objekt an. Zusätzlich müssen die Attribute der Notifizierungen mit denen des Template Objektes übereinstimmen, mit Ausnahme von Wildcards.

- Executable Code: Der Subscriber stellt einen ausführbaren Code zur Verfügung der zur Laufzeit die Notifizierungen filtert.

Content-based Subscription ist ein allgemeiner Ansatz, der es auch ermöglicht Topic-based Subscriptions zu implementieren, umgekehrt ist dies nicht möglich. Ein wesentliches Problem bei Content-based Systemen ist die Anwendung der Kriterien aus den Subscriptions auf die Notifizierungen welches in [ASS+99] behandelt wird.

Concept Concept-based Filtering ist ein weiterer allgemeiner Ansatz, der als Erweiterung zu Content based Filtering gesehen werden kann [CABB04]. Die bisher behandelten Systeme tauschen Daten aus, die in einem gemeinsamen Kontext stehen, da es sich um ein begrenztes System handelt. Diese Kontextinformationen werden implizit von den Entwicklern und Anwendern in das System integriert. Überschreitet ein System jedoch die Grenze eines Kontexts, so sind zu den Daten noch zusätzliche Metadaten erforderlich, damit die Bedeutung dieser Daten in den richtigen Kontext gebracht werden kann. Einfache Beispiele für Kontextunterschiede sind physikalische Einheiten, Währungen oder Datumsformate. Um dies zu verhindern, ist eine abstraktere Sichtweise notwendig um die Absichten der Publisher und Subscriber zu beschreiben. Dazu bedient man sich Ontologien, welche von Grund auf die Lücke zwischen Daten und deren Interpretation schließen.

Unter einer Ontologie versteht man in der Informatik im Bereich der Wissensrepräsentation ein formal definiertes System von Konzepten und Relationen. Die bekannteste Definition lautet "Spezifikation einer Konzeptualisierung". Zusätzlich enthalten Ontologien Inferenz- und Integritätsregeln.
Ontologien dienen in verschiedenen Bereichen als Mittel zur Strukturierung und zum Datenaustausch, um bereits bestehende Wissensbestände zusammenzufügen - beispielsweise genetische Daten in der Bioinformatik. Experten aus verschiedenen Gebieten müssen sich lediglich um die Modellierung ihres jeweiligen Spezialwissens und die dafür notwendigen Inferenzprozesse kümmern. Auf diese Weise können deklaratives Wissen, Problemlösungstechniken und Schlussfolgerungsmechanismen von mehreren Systemen geteilt werden [Wik05d].
Erreicht wird diese Funktionalität, indem zusätzlich zu den Daten noch Meta-Daten mitübertragen werden und Kontextkonvertierungsfunktionen bereitgestellt werden. Der Notifizierungsconsumer gibt den gewünschten Kontext an und die Notifizierungen werden mithilfe der Konvertierungsfunktionen in den entsprechenden Kontext konvertiert.

2.4.4 Advertisement

Das Gegenteil einer Subscription ist ein Advertisement. Steht eine Subcription für das Interesse an bestimmten Notifizierungen, so steht ein Advertisement dafür, dass ein Client nur bestimmte Notifizierungen erzeugen wird. Dies dient einerseits zu einer Restriktion der Producer auf bestimmte Notifizierungen und andererseits zum optimierten Routen der Nachrichten, da im Vorfeld bereits bekannt ist, welche Clients an den Notifizierungen des Producers Interesse haben.

2.4.5 Notification Service

Da ein Publish/Subscribe System eine lose Kopplung zwischen den verschiedenen Teilnehmern beabsichtigt, ist ein Medium zwischen den Teilnehmern notwendig. Dieses Medium wird als Broker, Notifizierungsservice oder Event Notifizierungsservice bezeichnet und ermöglicht die lose Kopplung der beteiligten Clients.
Das Notifizierungsservice stellt eine einfache jedoch ausreichende Publish/Subscribe Schnittstelle für die Clients zur Verfügung. Sowohl die Schnittstelle für Producer, als auch für Consumer müssen vom Service be-

reitgestellt werden. Grundsätzlich sind dafür nur folgende vier Dienste notwendig.

- Publish
- Subscribe
- Unsubscribe
- Notify

Eine Nachricht N gelangt mittels eines Publish Aufrufes, eines verbundenen Clients C, in das Notifizierungssystem. Das Notifizierungssystem überprüft die neue Nachricht, ob registrierte und passende Subscriptions zu dieser Nachricht bestehen. Die Subscriptions werden von den Clients, die ihr Interesse an Notifizierungen anmelden möchten, mittels eines Subscribe Aufrufes beim Notifizierungssystem registriert. Das Notifizierungsservice fügt diese Subscriptionen zu einer Liste aller aktiven Subscriptionen hinzu. Sobald eine Überprüfung einer Nachricht mit den aktiven Subscriptionen zu einem positivem Resultat führt, sendet das Notifizierungssystem diese Nachricht mit einem Notify Aufruf an alle Clients, welche vorher eine gültige Subscription abgesetzt haben.
Aus der Sicht der Anwendung ist das Verhalten des Notifizierungssystems nicht ersichtlich und wird als Blackbox betrachtet. Einzig die Schnittstelle ist für die Anwendung von Interesse, welche die Dienste für die Publish/Subscribe Funktionalität bereitstellt. Je nach System können auch zusätzliche Dienste im Notifikationsservice integriert sein, wie asynchrones Lesen oder Anforderung von Systemdaten wie Topiclisten, Clientlisten usw.
Neben dem Notification-Service ist auch ein Service für die Clientverwaltung notwendig. Dies ist verantwortlich für die korrekte Anmeldung und Abmeldung der Clients, sowie für einfache sicherheitsrelevante Vorkehrungen. Ein allgemeines Sicherheitskonzept ist durch die lose Kopplung der einzelnen Komponenten etwas schwieriger zu realisieren.

2.4.6 Security in Publish/Subscribe Systems

Da ein Publish/Subscribe System eine Vielzahl an unterschiedlichen Clients miteinander verbindet, muss auch die Sicherheit solcher Systeme betrachtet werden. Nachdem die Interaktion zwischen Consumer und Provider über das

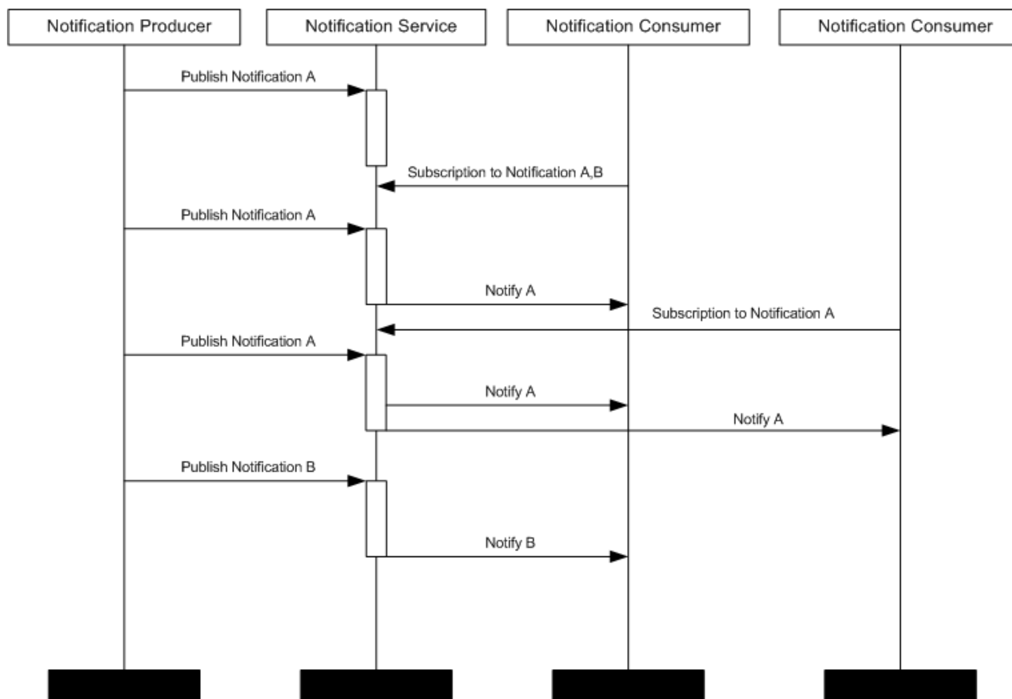

Abbildung 10: Ablauf Notifizierung

Publish/Subscribe Service abläuft reicht es nicht die beteiligten Clients als sichere Clients zu implementieren, sondern auch die Infrastruktur muss in diese Sicherheit mit einbezogen werden [FZP+04]. Somit ist eine Blackbox Betrachtung des Publish/Subscribe nicht mehr vollständig möglich. Das Publish/Subscribe Paradigma selbst beinhaltet keinerlei sicherheitsrelevanten Themen wie Vertrauen und Sicherheit. In erster Linie basiert die Sicherheit darauf, die Identität aller beteiligten Komponenten zu kennen. Dies ist in Publish/Subscribe Systemen grundsätzlich jedoch nicht möglich, da dies die lose Kopplung verletzen würde. Um eine vertrauenswürdige Kommunikation unter einer Menge von Clients durchführen zu können, führen wir zunächst den Begriff group of trust ein. Eine solche vertrauenswürdige Gruppe besteht aus Consumern, Publishern und dem Notifizierungssystem, welches die Consumer und Publisher miteinander verbindet. Weiters wird nun unterschieden, ob die Kommunikation innerhalb dieser Gruppe durchgeführt wird oder ob die Kommunikation mit Komponenten von außerhalb erfolgt. Eine solche Einführung von Gruppen erfordert neue Funktionalität beim Verteilen von Nachrichten und neue Interfaces für die beteiligten Clients. Jeder Kommunikationsweg der zwischen einem Producer und einem Consumer liegt muss dasselbe Maß an Sicherheit gewährleisten wie wenn die Kommunikation direkt zwischen den beiden stattfinden würde. Um diese Anforderungen technisch zu realisieren, werden so genannte Scopes eingeführt.

Scopes Scopes in Publish/Subscribe Systemen grenzen eine Gruppe von Consumern und Producer auf Applikationsebene und die Kontrolle der Ausbreitung von Nachrichten auf Infrastrukturebene ab. Scopes sind sozusagen die technische Realisierung der Grundfunktionalität von groups of trust. Ein Scope gruppiert Komponenten des Systems, welche sich üblicherweise nicht über eine solche Gruppierung bewusst sind. Die grundlegende Idee dahinter ist, die Sichtbarkeit von Notifizierungen und auch der Subscriptions außerhalb des Scopes zu beeinflussen. Die Sichtbarkeit von Notifizierungen ist anfänglich auf den Scope, in dem diese gepublished wurde beschränkt. Der Übergang einer Notifizierung von einem Scope zu einem anderen wird über ein eigenes Scope Interface gewährleistet. Dafür setzt der Scope selbst Subscriptions und Advertisments ab, um als normaler Consumer oder Producer in einem übergeordneten Scope (Superscope) agieren zu können. Um diese Funktionalität bereitstellen zu können, werden zusätzlich zu den Publish/Subscribe Diensten weiter vier Dienste benötigt. Jeweils ein Dienst zum

Anlegen und Auflösen eines Scopes sowie zum An- und Abmelden in einem bestehenden Scope. Es gibt zwei Ansätze dies zu implementieren, einerseits die Implementierung der zusätzlichen Dienste im Notifizierungsservice und andererseits ein eigener administrativer Service, der die Scopefunktionalität ermöglicht. Beim ersten Ansatz geht die lose Kopplung der Komponenten verloren, da eine Komponente wissen muss, in welchen Scopes diese angemeldet sein muss, um die Gesamtfunktionalität der Applikation gewährleisten zu können. Abhilfe schafft der zweite Ansatz, in dem die einzelne Komponente nicht über die Scopes Bescheid weiß. Ein Administrator stellt hierbei die nötige Information zur Verfügung, welche zur Deploymenttime die Scopezuordnung veranlasst.

Sicherheit Durch die Einführung der Scopes ist es nun möglich, die Kommunikation innerhalb der Scopes von der restlichen Kommunikation abzuschotten. Diese Aufspaltung ermöglicht es nun, verschiedene Sicherheitsvorkehrungen, je nach Erfordernissen der Anwendung zu implementieren. In üblichen Anwendungen wie E-Commerce oder mobilen Anwendungen muss der Zugang zum System beschränkt werden. Es muss sichergestellt sein, dass nur berechtigte Clients Zugang zu den Publish/Subscribe Diensten besitzen. Um dies zu erreichen stehen etliche Mechanismen zur Verfügung, welche in dieser Arbeit jedoch nicht näher behandelt werden.

2.5 Beschreibungssprachen

Um die Zusammenhänge in Nachrichtensystemen formal oder grafisch beschreiben zu können, bedient man sich speziellen Beschreibungssprachen, welche den zusätzlichen Anforderungen in nachrichtenbasierten Systemen gerecht werden. Klassische Spezifikationen sind meist schwer lesbar und lassen teilweise zu viel Spielraum für unterschiedliche Interpretationen.
Im Bereich von Kommunikationssystemen und Protokollen haben sich zwei Sprachen etabliert, die sich aus UML ableiten, auf die in dieser Arbeit näher eingegangen wird.

2.5.1 MSC Message Sequence Chart

MSC ist eine grafische und textuelle abstrakte Beschreibungssprache zur Darstellung, Beschreibung uns Spezifikation der Interaktion zwischen verschiedenen Systemkomponenten. Die Hauptanwendung von MSC hat ihren

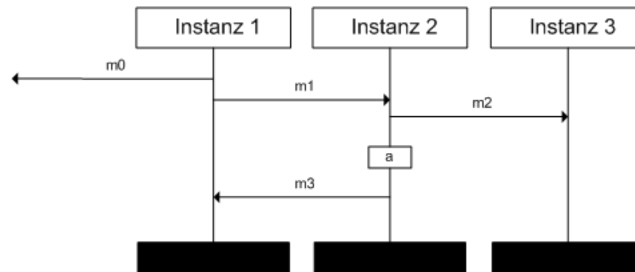

Abbildung 11: Einfaches MSC

Ursprung in der übersichtlichen Spezifikation des Systemverhaltens von Echtzeitsystemen, im Speziellen von Telekommunikationssystemen. Die erste Definition von MSC stammt von der ITU (International Telekomunication Union) [UNI98]. MSC werden vor allem zur Anforderungsspezifikation, Interfacespezifikation, Spezifikation von Testfällen und zur Dokumentation von Echtzeitsystemen verwendet. Für eine Einführung in MSC ist die grafische Darstellung am besten geeignet. Benötigt man jedoch eine formale semantische Repräsentation wird die textuelle Darstellung bevorzugt.

Basic Message Sequence Chart Die Grundelemente werden in der Basic Message Sequence Chart Language definiert [UNI98]. Die Basic Message Sequence Chart konzentriert sich ausschließlich auf die Kommunikation und lokalen Aktionen, welche in allen zu MSC vergleichbaren Beschreibungssprachen (Extended Sequence Chart, Informationsflussdiagramme, Nachrichtenflussdiagramme) ebenso vorkommen. Die Basiselemente in einem MSC sind eine endliche Anzahl von Instanzen. Eine Instanz ist eine abstrakte Einheit welche Nachrichten versendet, empfängt oder lokale Aktionen durchführen kann.

Jede Instanz beginnt mit einem Instanz-Beginn-Symbol und endet mit dem entsprechendem Instanz-Ende-Symbol, welche jedoch nicht für das Anlegen und Terminieren einer Instanz stehen sondern nur für den Start und dem Ende der Beschreibung. Der Instanzname steht dabei über oder im Instanz-Beginn-Symbol und muss innerhalb eines MSC eindeutig vergeben werden. Lokale Aktionen werden mit einem Aktionssymbol dargestellt, wobei eine, die Aktion beschreibende, Anmerkung im Symbol platziert wird. Eine Nachricht zwischen zwei Instanzen wird als Pfeil dargestellt, welcher bei der Send-

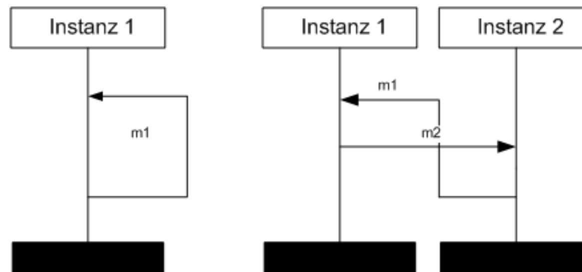

Abbildung 12: Inkonsistentes MSC

einstanz beginnt und bei der Empfangsinstanz endet. Wird eine Nachricht von einer Instanz zur Umgebung des Systems gesendet, so wird dies mit einem Pfeil startend bei der Sendeinstanz bis zum Rand des MSC dargestellt, ebenso in die umgekehrte Richtung. Jede Nachricht wird mit einem Nachrichtennamen verbunden, wobei dieser möglichst nahe am dazugehörigen Nachrichtenpfeil platziert werden sollte. Im Normalfall ist es nicht erlaubt auf der Instanzachse mehrere Events auf derselben Höhe zu platzieren, außer es handelt sich um eine eingehende und ausgehende Nachricht, wobei dann davon ausgegangen wird, dass die eingehende Nachricht zeitlich früher angeordnet ist. Ein MSC beschreibt nicht nur eine Reihe von Events wie Empfangen und Senden von Nachrichten sowie die Ausführung von Aktionen, sondern es enthält auch die Information der zeitlichen Abfolge dieser Ereignisse. Eine wesentliche Annahme bei MSC ist, dass ein Event keine Zeit benötigt und dass nie mehrere Events gleichzeitig auftreten. Weiters ist es nicht gestattet, dass ein Senden kausal vom Empfang der dazugehörigen Nachricht abhängig ist. Werden diese Eigenschaften verletzt, so spricht man von einem inkonsistenten MSC siehe folgende Abbildung.
Dieses Diagramm spezifiziert, dass der Empfang der Nachricht m1 kausal vor deren Versenden liegt und zeigt somit ein inkonsistentes MSC. Der Hauptaugenmerk bei MSC liegt bei der grafischen Repräsentation und Darstellung doch es gibt auch eine konkrete textuelle Syntax um die Inhalte zu beschreiben.

Textuelle Beschreibung von MSC Es gibt zwei Arten der textuellen Beschreibung, zum einen die isolierte Beschreibung der einzelnen Instanzen, welche als instance-oriented bezeichnet wird und zum anderen die ereignis-

orientierte Beschreibung, in der nur eine Liste aller Ereignisse, die entsteht, wenn man ein MSC von oben nach unten durchscannt, angeführt wird. Es ist ebenfalls möglich, eine Kombination aus beiden Beschreibungen zu verwenden. Die textuelle Beschreibung besteht aus den Keywörtern msc und endmsc zwischen denen sich der MSC- Name und der MSC-Body befindet. Ein Nachrichtenevent wird entweder als Nachrichteninput oder Nachrichtenoutput beschrieben. Wenn m eine Nachricht von Instanz i zur Instanz j ist, wird dies textuell mit i: out m to j und j: in m from i beschrieben. Bei der grafischen Darstellung wird der Zusammenhang zwischen Nachricht und Instanz über den Pfeil abgebildet. In der Textform müssen folgende Eigenschaften erfüllt sein, um den Zusammenhang zwischen Nachrichten Eingang und Ausgang darzustellen:

- Die Ereignisse haben denselben Nachrichtennamen
- Beim Ausgangsevent ist der Instanzname der gleiche wie beim Empfangsereignis der Sendeinstanzname
- Beim Eingangsevent ist der Instanzname der gleiche wie beim Ausgangsereignis der Zielinstanzname

Für jeden Nachrichteneingang muss ein korrespondierender Nachrichtenausgang existieren und umgekehrt. Diese Bedingung muss nicht erfüllt sein, wenn eine Nachricht an die Umgebung des Systems gesendet oder eine Nachricht von der Umgebung des Systems empfangen wird. Eine lokale Aktion wird mit dem Keywort action gekennzeichnet. Beispiel:

msc beispiel1;

i1 : out m0 to env;

i1 : out m1 to i2;

i2 : in m1 from i1;

i2 : out m2 to i3;

i3 : in m2 from i2;

i3 : out m3 to i4;

i4 : in m3 from i3;

i2 : action a;

i2 : out m4 to i1;

i1 : in m4 from i2;

endmsc;

2.5.2 Erweiterte Basic Message Sequence Charts

Um komplexere Systeme mithilfe von MSC beschreiben zu können, bedarf es weiterer Elemente wie Prozesserzeugung und Terminierung, zeitliche Elemente, unvollständige Nachrichten und Bedingungen.

Prozesserzeugung und Terminierung Erzeugt eine Instanz eine andere, so wird dies mit einem gestrichelten Pfeil dargestellt.Dieser Pfeil startet bei der erzeugenden Instanz und endet am Instanzkopf der neu erzeugten Instanz. Ebenso wie eine neue Instanz erzeugt werden kann, kann eine Instanz auch terminiert werden. Ein Instanzstopp darf nur als letztes Ereignis in einer Instanz vorkommen und wird durch ein Kreuz signalisiert.

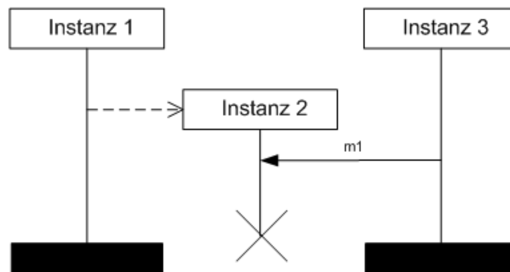

Abbildung 13: Instanzerzeugung und Instanzende

In der textuellen Darstellung wird eine Prozesserzeugung der Instanz j durch create j und deren Terminierung durch stopp dargestellt.

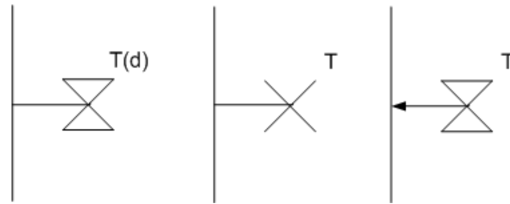

Abbildung 14: Zeitliche Elemente

Zeitliche Elemente Um zeitliche Zusammenhänge und Abläufe zu modellieren, werden drei zusätzliche Ereignisse eingeführt, das Setzen eines Timers, das Zurücksetzen eines Timers und der Ablauf eines Timers.
Da MSC keine quantitative Zeiteinheit besitzt, sind diese Elemente symbolisch zu verstehen. Das Setzen, Rücksetzen und der Timerablauf werden als Events interpretiert.
In der textuellen Darstellung verwendet man set T, reset T und timout T.

Unvollständige Nachrichten Neben der erfolgreichen Übertragung von Nachrichten wird MSC nun um eine lost Message und eine found Message erweitert. Eine lost Message ist eine Nachricht, die zwar gesendet wurde, jedoch nie beim Empfänger ankommt, also verloren gegangen ist. Äquivalent dazu ist eine found Message eine Nachricht, die zwar empfangen wird, jedoch nie von einer Instanz gesendet wurde.
Grafisch wird eine lost Message durch einen Pfeil, der in einem schwarzen Punkt endet und eine found Message durch einen Pfeil, welcher von einem weißen Punkt ausgeht, dargestellt. Die entsprechende textuelle Darstellung ist out m to lost j und in n from found i

Bedingungen Eine sehr wichtige Erweiterung zu den Basic Message Sequence Charts ist die Einführung von Bedingungen. Eine Bedingung kann eine beliebige Menge an Instanzen betreffen und aus einer Vielzahl an Bedingungen.
Wird eine Instanz nicht in die Bedingung mit einbezogen, so wird diese durch die Bedingung gezeichnet. In Abbildung 4 werden die Instanzen 1 und 3, nicht jedoch Instanz 2 mit der Bedingung verknüpft.

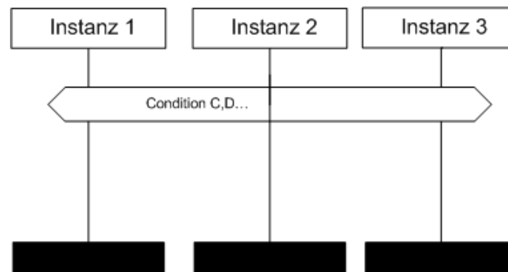

Abbildung 15: Bedingungen

2.5.3 LSC Live Sequence Chart

Eine abgeänderte und stark erweiterte Form der MSC wurde durch [DH01] eingeführt. Die wesentlichen Nachteile von MSC sind laut [JB02] folgende Punkte:

- Es wird nur eine existierende oder eine szenarioähnliche Ansicht unterstützt. Es wird also nur ein beispielhaftes Verhalten eines Systems oder eine mögliche Kommunikationsabfolge beschrieben.
- Es wird nicht spezifiziert, wann die im MSC spezifizierte Ereignisabfolge aktiviert wird oder eintritt.
- Es ist nicht möglich Ereignisse als unbedingt notwenig auszuzeichnen, wie Nachrichten die empfangen werden müssen.
- Keine Unterscheidung von möglichem Verhalten und notwendigem Verhalten

Die Grundidee der LSC ist die Unterscheidung in notwendiges und mögliches Verhalten. Nahezu jedes Element kann einem der beiden Eigenschaften zugeordnet werden und wird entweder durch eine durchgängige oder eine strichlierte Linie grafisch dargestellt.

Instanzen und Nachrichten Instanzen und Nachrichten bilden die Basiselemente wie auch in MSC. Die grafische Darstellung ist von der MSC-Spezifikation adaptiert. Ein Unterschied ist das Senden und Empfangen von und zur Systemumgebung. Wurde dies bei MSC durch eine Nachricht an

den Chartrand signalisiert, so gibt es bei LSC eine eigene Instanz welche die Systemumgebung repräsentiert. Die Umgebungsinstanzen werden mit einem schattierten Instanzkopf gekennzeichnet.
Bei den Nachrichten unterscheidet man grundsätzlich asynchrone und synchrone Nachrichten. Eine synchrone Nachricht blockiert den Sender, bis der Empfänger bereit ist, die Nachricht zu empfangen. Dadurch wird es überprüfbar, ob der Nachrichtenaustausch auch tatsächlich stattgefunden hat. Asynchrone Nachrichten werden durch schräge Pfeile dargestellt, um die mögliche zeitliche Differenz zwischen dem Sendeereignis und Empfangsereignis zu veranschaulichen.
Operative Aufrufe oder Methodenaufrufe werden durch zwei synchrone Nachrichten, der Call und der Return Nachricht, veranschaulicht. Der Methodenbody wird durch ein kleines Rechteck beim Empfänger der Call Nachricht symbolisiert.

Fortschrittsmodellierung Im Gegensatz zu MSC gibt es bei LSC die Möglichkeit einen Fortschritt zu modellieren. Bei MSC kann man keine zeitliche Begrenzung für ein Ereignis festlegen. Um diesem Nachteil Abhilfe zu schaffen, werden bei LSC den Positionen und den Nachrichten eine temperature zugeordnet. Positionen sind jene Punkte auf der vertikalen Zeitachse an denen ein Ereignis angehängt ist. Die Temperatur kann entweder den Wert heiß oder kalt annehmen. Die dahinter stehende Idee ist, dass man nicht lange an einer heißen Position verweilen kann, ohne sich dabei die Füße zu verbrennen. Dies führt dazu, dass eine heiße Position verlassen werden muss, um die folgende Position einzunehmen. An einer kalten Position kann man lange verweilen und die folgende Position muss nicht erreicht werden. Bezieht man dies auf eine Nachricht, so heißt dies, dass eine heiße Nachricht zugestellt werden muss. Im Gegensatz dazu steht eine kalte Nachricht, für eine Nachricht, deren Eintreffen beim Empfänger nicht garantiert wird.
Grafisch werden heiße Temperaturen mit durchgehenden Linien und kalte Temperaturen durch strichlierte Linien gekennzeichnet.

Bedingungen Bedingungen werden wie in den MSC dargestellt, jedoch gibt es ebenfalls als zusätzliche Erweiterung zu den MSC die Unterscheidung zwischen zwingenden Bedingungen (mandatory Conditions) und möglichen Bedingungen (possible Conditions). Diese Unterscheidung entspricht nicht den Temperaturen wie bei den Positionen und Nachrichten, vielmehr

	Heiß	Kalt
Position	Position muss verlassen werden, um die Folgende zu erreichen	Position kann für unbestimmte Zeit beibehalten werden
Nachricht	Muss beim Empfänger ankommen, wenn sie versendet wurde	Nachricht kann auf dem Übertragungsweg verloren gehen

Tabelle 3: LSC Temperaturen

ist es eine Indikation der Auswirkung bei einer nicht erfüllten Bedingung. Im Gegensatz dazu stellt die Temperatur eine Lebensdauer oder eine zeitliche Gültigkeit dar.

Gleichzeitigkeit Werden bei MSC keine gleichzeitigen Ereignisse erlaubt, gibt es bei LSC die Möglichkeit Ereignisse als zeitlich gleichzeitig auf der vertikalen Zeitachse zu kennzeichnen. Diese Kennzeichnung erfolgt mit einem gefüllten Kreis.

Lokale Invariante Bedingungen sind auf einen bestimmten Zeitpunkt beschränkt, oft ist es jedoch notwendig, die Gültigkeit einer Bedingung über einen ganzen Zeitraum zu beschreiben. Eine lokale Invariante ist eine Bedingung, mit zeitlichem Start und Endpunkt. Der Start und Endpunkt muss mit einem Ereignis wie Empfangs- und Sendeereignis oder einem Time-out zusammenfallen. Ist die lokale Invariante exklusive diesem Ereignis, so wird beim Ereignis kein Gleichzeitigkeitszeichen gesetzt, gilt die lokale Invariante bereits beim Zeitpunkt des Ereignisses muss das Ereignis mit dem gefüllten Kreis gekennzeichnet werden.

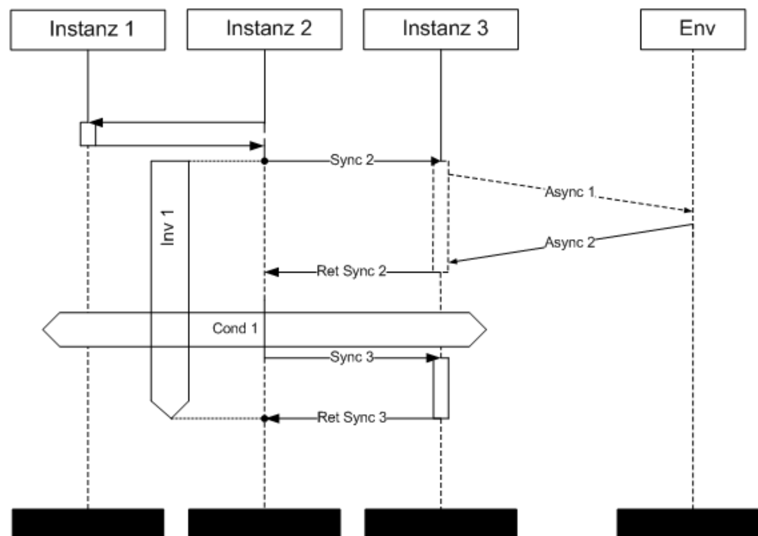

Abbildung 16: LSC - Lokale Invariante

3 Bestehende Produkte im Überblick

Im Laufe der Zeit haben sich viele Middleware-Produkte am Markt etabliert. Hier werden nur überblicksmäßig einige vorgestellt.

3.1 Kommerzielle Produkte

3.1.1 Advanced Queuing

Advanced Queuing [Ora06] von Oracle ist ein vollwertiges Message-Queuing System für Geschäftsapplikationen. Die enge Eingliederung in die Oracle Datenbank hat eine hohe Zuverlässigkeit und Sicherheit sowie Datenintegrität für Advanced Queuing zur Folge. Die Nachrichten, die zwischen den Applikationen ausgetauscht werden, werden in speziellen Tabellen gespeichert. Erst nach erfolgreicher Abarbeitung der Nachricht wird diese aus der Tabelle entfernt.

3.1.2 Arjuna Message Service

ArjunaMS [Arj06] ist ein Nachrichtensystem, welches auf der JMS-Architektur von Sun Microsystems aufsetzt. ArjunaMS entstand aus dem Hewlett-Packard Message Service (HP-MS) und ist in Java implementiert. Daraus resultiert die Uterstützung einer Vielzahl an Plattformen.

3.1.3 MessageQ

BEA MessageQ [Sys05] ist eine einfach zu bedienende, schnelle und zuverlässige Messaging-Software, die Anwendungen ermöglicht, über die marktführende Message-Bus-Technologie miteinander zu kommunizieren. Als bewährte und vielfach implementierte Middleware-Lösung für verteilte Enterprise-Anwendungen garantiert BEA MessageQ die Zustellung von Anwendungs-Messages über Plattformgrenzen hinweg. BEA MessageQ basiert auf einer robusten Integrations-Architektur für die Entwicklung leistungsstarker, messagebasierter Anwendungen, die Nachrichten in unterschiedlichen Formaten austauschen können.

3.1.4 MSMQ

Message Queuing [Mic05a] ist eine Kommunikationsinfrastruktur, die verteilte, lose gekoppelte Nachrichtenanwendungen eine zuverlässige und einfache Kommunikation ermöglichen. MSMQ ermöglicht die Kommunikation über heterogene Netzwerke sowie mit Computern, die zeitweise offline sind. Message Queuing gewährleistet die Zustellung von Nachrichten, effizientes Routing, Sicherheit, Transaktionsunterstützung und prioritätsbasierte Nachrichtenübermittlung.

3.1.5 WebSphere MQ

IBM WebSphere MQ [IBM05] ist ein weit verbreitetes Produkt und hält mit 75% Marktanteil die Führung am Markt. WebSphere MQ zeichnet sich durch hohe Zuverlässigkeit und einer Vielzahl an Schnittstellen zu Fremdsystemen aus. Ebenfalls hervorzuheben ist die Unterstützung von 38 verschiedenen Plattformen. Die Informationsübertragung erfolgt asynchron und die Übertragung der Informationen erfolgt unter Garantie, wenn erforderlich auch verschlüsselt.

3.2 Open Source Produkte

3.2.1 JORAM

JORAM (Java Open Reliable Asynchronous Messaging) [JOR06] ist eine Implementierung der JMS (Java Messaging Service) API von Sun Microsystems. Es wird sowohl ein Point-to-Point als auch ein Publish/Subscribe Kommunikationsmodell unterstützt. JORAM unterstützt den JMS-Standard 1.1 und bietet außerdem noch einige zusätzliche Dienste wie Load-Balancing, Monitoring und grafische Oberfläche.

3.2.2 MQ4CPP

MQ4CPP oder Message Queue for C++ [Pom] ist eine open Source Implementierung eines nachrichtenorientierten Nachrichtensystem. Dieses System unterstützt eine Reihe an Kommunikationsparadigmen wie Publish/Subscribe, Store and Forward, broadcast, uvm. Weiters werden eine Vielzahl an Diensten angeboten wie Verschlüsselung, Session Management, Lookup-Service und komplexe Nachrichtenformate.

3.2.3 MantaRay

MantaRay [Man06] ist eine Serverlose Peer-to-Peer Kommunikationslösung, welche JMS und RMI API´s bereitstellt. Eine Integration mit WebSphere, Jboss und WebLogic ist ebenfalls möglich. MantaRay verspricht hohe Performance, Scalability und Zuverlässigkeit. Eine Verbindung zu JMS und WebServices ist ebenfalls leicht möglich.

3.2.4 xmlBlaster

XmlBlaster [XML06] ist eine Publish/Subscribe und Point-to-Point Middleware. Die Nachrichten werden mithilfe von XML codiert und können beliebige Nutzdaten enthalten. Interessant ist die Unterstützung von mehreren Protokollen wie CORBA, RMI, XmlRPC, uvm. Weiters werden für eine Reihe an Programmiersprachen Beispielclients zur Verfügung gestellt.

4 Implementation eines Publish/Subscribe Message Server

Ein wesentlicher Teil dieser Arbeit bestand darin, ein Message-Oriented Middleware System zu entwickeln und dieses auch zu testen. Das grundlegende Ziel der Entwicklung war die Schaffung eines sehr flexiblen und erweiterbaren Frameworks welches zum Datenaustausch zwischen verschiedenen Applikationen und Systemen dient. Die Entwicklung des Flexible Message Server (FMS) teilt sich grob in drei unterschiedliche Bereiche. Die Entwicklung eines Standalone-Servers, die Schaffung eines Client Frameworks und die Implementierung funktioneller Plugins für sowohl Server als auch Client.

4.1 FMS Ziele und Designentscheidungen

Da bereits viele ausgereifte Produkte auf diesem Marktsegment etabliert sind und sich die Entwicklung sehr aufwendig gestaltet, wurde auf Einfachheit und Flexibilität großer Wert gelegt. Ziel war es ein leicht erweiterbares Framework zu schaffen, welches nur die elementare Grundfunktionalität bereitstellt. Die Anpassung an das gewünschte Einsatzgebiet wird über Plugins möglich, welche die anwendungsspezifische Funktionalitäten enthalten. Die Grundfunktionalität soll nur eine schlichte Basisfunktionalität bereitstellen, auf der komplexere Erweiterungen und Anpassungen aufsetzen.

4.1.1 Publish / Subscribe Paradigma

Es kommt ein üblicher Publish/Subscribe Mechanismus zum Einsatz. Die Subscriptionen erfolgen Topicbasierend, wobei jedes Topic über einen eindeutigen URI [(IE98] identifiziert wird.

4.1.2 Plugin basierter Ansatz

Um einen flexiblen und leicht erweiterbaren Systemaufbau zu erreichen, wird beim FMS ein Plugin-Framework als zentrale Architektur verwendet. Das System bietet nur elementare Funktionen, die jedoch leicht über Plugins erweitert werden können. Die Grundfunktionalität ist teilweise ebenfalls als Plugin implementiert.

4.1.3 Unterstützung verschiedener Kommunikationsinterfaces

Ein wesentliches Feature des FMS ist die Möglichkeit, unterschiedliche Kommunikationsinterfaces simultan zu nutzen. So besteht die Möglichkeit, verschiedenste Systeme und Applikationen ohne den Einsatz von System-Bridges untereinander zu verbinden. Dies öffnet auch die Möglichkeit, lokale Ressourcen mithilfe einer Art Dummy-Kommunikationsplugin in das System einzubinden.

4.1.4 Leichte funktionelle Erweiterbarkeit

Generelle Erweiterbarkeit wird über das Pluginframework möglich. Der FMS besteht aus 3 Basis- Plugins, einem Basis-Kommunikations-Plugin, einem Basis-Broker-Plugin und einem Basis-Topic-Plugin. Diese Plugins können über Extension Points mit Funktionalität leicht erweitert werden und an die anwendungsspezifischen Anforderungen angepasst werden.

4.1.5 Leichte Einbindung von serverseitigen Funktionen

Der FMS ist nicht nur als zentraler Kommunikationsdienst einsetzbar, sondern bietet über die Pluginfunktionalität das einfache Einbinden von serverseitigen Funktionen und serverseitiger Applikationslogik. Jedem Topic kann ein eigenes Plugin zugewiesen werden, welches über verschiedenste Methoden diverse Funktionalität bereitstellen kann. Angestoßen wird die Funktion beim Eintreten von einem Ereignis wie z.B. der Empfang einer Nachricht unter einem Topic.

4.1.6 Grafische Oberfläche

Um die Entwicklung zu erleichtern, wurde von Grund auf eine grafische Oberfläche entwickelt, die Systeminformationen über die Topics, Plugins und die verbundenen Clients bereitstellt. Es können einige Funktionen ebenfalls von dieser Oberfläche ausgehend bedient werden. Ebenfalls können Topic oder clientbezogene Logeinträge betrachtet werden, das das Debuggen der Gesamtapplikation erleichtert. Jedes Plugin kann eine grafische Oberfläche bereitstellen, welches sich in die Oberfläche einbindet.

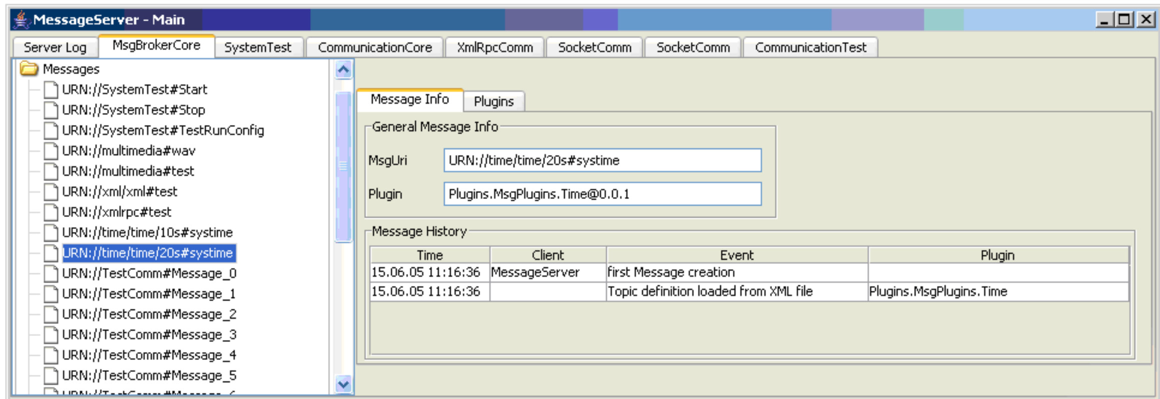

Abbildung 17: Grafische Oberfläche

4.1.7 Leicht konfigurierbar

Easy to Configure and Easy to Start ist ebenfalls ein grundlegendes Ziel von FMS, die Basis Konfiguration erfolgt über ein Propertiefile, in dem der Ort der Plugins angegeben wird. Der Rest der Konfiguration erfolgt pluginbezogen. Dies ermöglicht einen einfachen Start des Systems.

4.1.8 Entwicklung in Java 1.5

Der Einsatz der aktuellsten Java Version bietet einige Vorteile, jedoch auch Nachteile.
Zu den wichtigen Vorteilen zählen die Erweiterungen generische Datentypen und non-blocking IO (NIO) [Mic05c]. Als Nachteil ergibt sich, dass einige Hardware Plattformen noch kein aktuell portiertes Java 5 unterstützen, die vor allem in der Industrie noch häufig anzutreffen sind wie VMS, OpenVMS und ältere Unix Derivate.

4.1.9 Konfiguration über XML- Files

Die Konfiguration erfolgt -je nach Plugin- meist über XML Dateien. Lediglich die Basiskonfiguration erfolgt in einem Java konformen Propertyfile. Die Konfiguration über XML Files bietet komfortable Möglichkeiten, da viele Softwareprodukte für XML Dokumente vorhanden sind.

4.1.10 Unterschiedliche Laufumgebungen

Um den Entwicklungsprozess von Applikationen gut zu unterstützen, wurde ein grafisches Interface entwickelt. Für den laufenden Betrieb ist die leistungsstärkere Konsolenvariante vorzuziehen welche keine grafischen Elemente unterstützt. Für den Einsatz in Webapplikationen ist auch eine Integration in die Tomcat Engine möglich [Apa05]. Ein einfaches Web - Debug Interface steht ebenfalls zur Verfügung.

4.1.11 Pluginerweiterungen zur Laufzeit

Der FMS unterstützt das Laden von Java Plugins zur Laufzeit. Das bedeutet, dass Erweiterungen ohne neuen Start des Servers aktiviert werden können. Ebenso können Systemkomponenten und Plugins zur Laufzeit entfernt werden.

4.1.12 Universell einsetzbarer Basisclient

Ebenso wie die serverseitige Implementierung wurde auch ein Pluginbasierender Client geschaffen, der teilweise mit denselben Plugins arbeitet als der FMS. Der Client stellt alle systembezogenen Funktionen zur Verfügung, lediglich die applikationsspezifischen Anforderungen müssen über ein Plugin bereitgestellt werden.

4.2 System Architektur

Das Gesamtsystem besteht aus einem Server und einer Vielzahl an Clients, die jeweils verschiedene funktionelle applikationsspezifische Aufgaben übernehmen und somit ein funktionelles Gesamtsystem bilden. Die Topologie entspricht grundsätzlich einer klassischen Client/Server Topologie mit sternförmigem Aufbau. Aufgrund der leichten Erweiterbarkeit durch Plugins ist es auch möglich andere Topologien zu unterstützen. Vor allem ein Cluster mit Load-Balancing und Client-Handover oder synchronisierte Server bzw. Backupserver stellen sinnvolle Erweiterungen dar.Jeder Client verbindet sich zu einem ihm bekannten FMS Server. Die Kommunikation zwischen Clients erfolgt in der Basisversion ausschließlich über einen Server.

4.2.1 Topics

Jedes Topic stellt einen Container für kontextabhängige Daten dar. Der Kontext wird über einen eindeutigen URI identifiziert, wobei der URI den in [(IE98] spezifizierten Richtlinien folgen muss. Ein Topic stellt zwei wichtige Funktionen bereit, eine um topicspezifische Ereignisse in einer History zu protokollieren und um interne oder pluginspezifische Daten zu einem Topic in einem Attachment abzulegen. Die Daten, die unter dem Topic Kontext verwaltet werden, benötigen kein bestimmtes Format. Der FMS schränkt das Datenformat grundsätzlich nicht ein. Applikationsspezifisches Datenmanagement kann über ein Topic-Plugin bereitgestellt werden, falls dies nötig ist. Dies ermöglicht die Validierung, Verifizierung oder Authentifizierung der Daten bzw. beliebige Modifikation oder aus den Daten abgeleitete Funktionalität. Ebenso kann über ein Topic- Plugin das Standard Systemverhalten abgeändert werden. Jedes Topic verfügt über eine Reihe an ereignisabhängigen systeminternen Funktionen, welche ohne Plugin einer Standardfunktionalität folgen. Wird zu einem Topic ein Plugin geladen, so übernimmt dieses Plugin die Steuerung und kann somit ein spezielles Verhalten erzielen. Ein einfaches Beispiel für ein solches funktionsmodifiziertes Topic wäre ein Systemzeit-Topic. Das Plugin sorgt dafür, dass alle Clients in einem definierten Zeitintervall die Systemzeit des FMS-Servers erhalten. Diese Architektur ermöglicht es außerdem externe Ereignisse (wie Webcam, E-Mail, GPS, Zeit,) einfach und direkt ins System einzubinden.
Folgende Ereignisfunktionen können durch ein Topic Plugin von der Standardfunktionalität abweichen:

- onInit: Wird beim erstmaligen Anlegen des Topics aufgerufen und dient zur Ausführung von eventuell notwendigen Initialisierungen.
- onReceive: Ereignis beim Empfang einer Nachricht für dieses Topic. Die Standardfunktonalität speichert zum Beispiel die erhaltenen Daten ab, erstellt einen Eintrag in der Topic- History.
- onSend: Ereignis, wenn die Topic Daten zu einem Client versendet werden.
- onPublish: Ereignis, wenn die Topic Daten zu einer Gruppe von Subscribern versendet wurde.

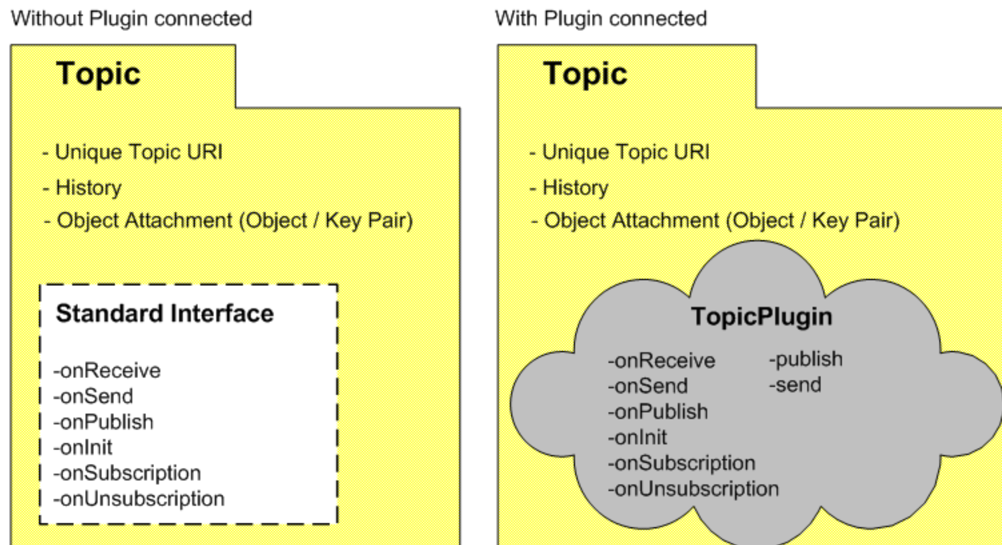

Abbildung 18: Topic

- onPluginAddedToTopic: Wird ein Plugin einem Topic zugewiesen, so wird jedes andere Topic Plugin verständigt. Dies dient dazu, um Abhängigkeiten zwischen den Topic Plugins leicht handhabbar zu machen.
- onTopicAdd: Ereignis, wenn ein neues Topic beim FMS registriert wird.
- onSubscription: Subscribed sich ein Client zu diesem Topic, so wird diese Methode aufgerufen. Bei manchen Daten macht es ev. Sinn diese gleich nach der Subscription dem Client zuzusenden.
- onUnSubscription: Ebenso wie beim Subscriben, kann man Funktionalität beim Unsubscribe Ereignis einfügen.
- setData: Beim Setzen der topicbezogenen Daten ist eine Vielzahl an Operationen vorstellbar, angefangen von der Frage ob Daten ersetzt oder angehängt werden, bis zur Validierung oder Verschlüsselung.
- getData: Weichen die tatsächlichen Daten von den versendeten Daten ab, so kann man diese Methode auf die Anforderungen der Anwendung anpassen.

- getDataAsGUI: Ist ein grafisches Debug Interface verfügbar und gibt es eine sinnvolle grafische Repräsentation der Daten, so kann die GUI mit dieser Methode die grafische Darstellung zur Verfügung gestellt werden.
- getDataAsString: Kann die Datendarstellung als ASCI String bereitstellen.
- createAndShowGUI: Besitzt das Topic Plugin selbst über eine grafische Oberfläche, wird diese mithilfe dieser Methode in die Debug Oberfläche eingebunden.
- Publish: Das Plugin kann mit dieser Methode die Entscheidung treffen ob dieses Topic überhaupt gepublished wird oder nicht.
- Send: Ebenso wie Publish gilt dies beim Senden zu einzelnen Clients.

4.3 Server Architektur

4.3.1 Message Broker

Das zentrale Element des FMS ist der Message Broker (MsBroker). Er bildet die Verbindung zwischen den serverseitigen Daten und den Clients und stellt die nötige Kernfunktionalität zur Verfügung. Der MsBroker ist für das richtige Routing der einzelnen Nachrichten zuständig. [HW03]. Der MsBroker hat eine Vielzahl an Aufgaben

- Kapselt systemspezifische Funktionalität
- Vermittelt zwischen Client und Broker
- Verteilt die Nachrichten an die richtigen Clients
- Registriert die Clients
- Stellt die Core-Plugins zur Verfügung
- Registriert die Topics
- Verwaltet die Subscriptionen

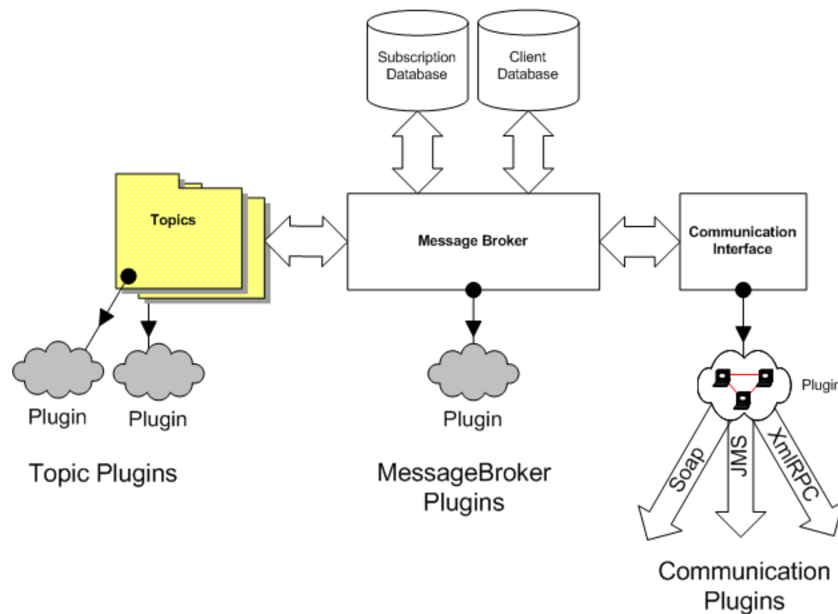

Abbildung 19: Topic

Um über einen Broker kommunizieren zu können, muss sich jeder Client beim Broker anmelden und registrieren. Der Broker hält alle verbundenen und registrierten Clients in einer Client Datenbank. Ab dem Zeitpunkt der Registrierung nimmt der Broker Anfragen vom Client entgegen. Die Kommunikation erfolgt über ein Kommunikationsplugin, welches auf die Dienste des Brokers zurückgreifen kann. Alle kommunikationsspezifischen Eigenschaften und Funktionen werden über dieses Kommunikationsplugin vom Gesamtsystem gekapselt. Stellt man das System dem OSI-Schichtenmodell gegenüber, so erhält man folgende Struktur:

4.3.2 Plugin Framework JPF

Als Basis für die Entwicklung wurde ein Java Pluginframework [Ols05] verwendet. Das Framework stellt eine Standard Plugin-Infrastruktur bereit, welche einfach in eine Java Applikation integriert werden kann.
Durch die Einführung einer Plugin-basierten Architektur ergeben sich eine Reihe an wesentlichen Vorteilen.

- Einfache Entwicklung und Implementierung spezieller Funktionalität

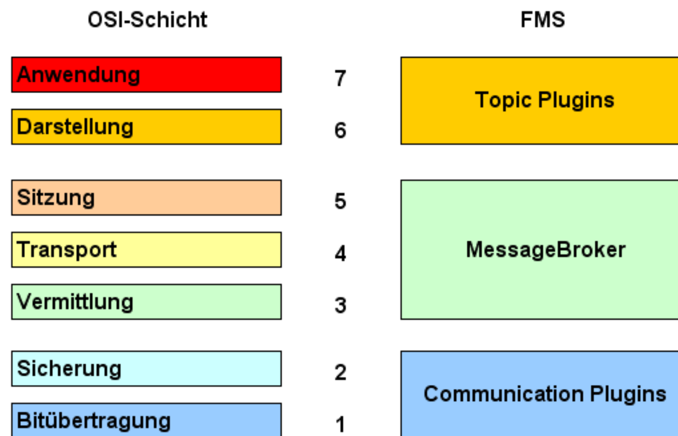

Abbildung 20: Schichtendarstellung

- Kein Wissen über den Sourcecode notwendig
- Testen der Komponenten wird einfacher
- Das Verwenden von definierten Interfaces ist notwendig
- Serverseitige als auch clientseitige Verwendung der Funktionalität
- Die Applikation wird extrem erweiterbar
- Wartung gestaltet sich wesentlich einfacher
- Widerverwertbarkeit von Code ist leichter zu realisieren
- Dynamisches Laden und Entfernen von Plugins zur Laufzeit
- Dependency Check

Die Laufzeitkosten, die durch den zusätzlichen Overhead entstehen, halten sich im Normalfall gering und wirken sich vor allem beim Systemstart und beim Laden der Plugins aus.

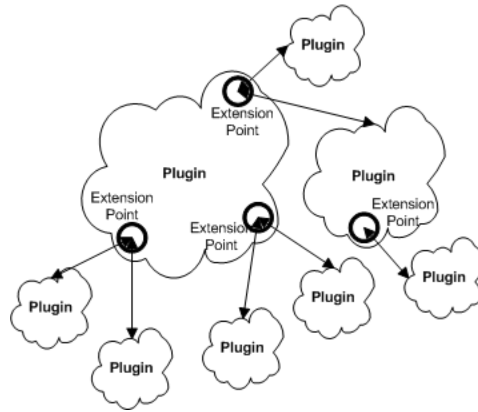

Abbildung 21: Plugin-Extensionpoints

Plugin Plugin (von engl. to plug in - einstöpseln, anschließen) oder Ergänzungs- oder Zusatzmodul ist eine gängige Bezeichnung für ein Softwareprogramm, welches in ein anderes Softwareprodukt "eingeklinkt" wird [Wik05g]. Ein Plugin ist eine strukturierte Komponente, welche Programmcode und Ressourcen für die Hauptapplikation bereitstellt und sich über eine Manifest-Datei beim Framework als solches beschreibt. Plugins sind in der Regel ohne die Anwendung für die sie geschrieben wurden nicht funktionsfähig oder ausführbar. Da üblicherweise ein Plugin genau für eine spezielle Funktionalität zuständig ist, bedarf es einer Möglichkeit, Plugins an definierten Punkten einzuhängen. Diese Punkte (Extension Points) akzeptieren nur Plugins, die einem speziellen Interface für eben diesen Extension Point folgen. Jedes Plugin kann ebenfalls Extension Points für weitere Plugins bereitstellen. So entsteht die Möglichkeit, Plugins untereinander zu verschachteln. Dies bringt den Nachteil von möglichen Abhängigkeiten, der über einen implementierten Dependency Check jedoch tragbar ist.
Die Extension Points, die ein Plugin bereitstellt, werden über das Plugin-Manifest File beschrieben. Dies ermöglicht dem Framework die Abhängigkeiten auch, ohne dem vorherigen Laden des Plugins zu überprüfen. Ein Plugin wird tatsächlich erst dann geladen, wenn zum ersten Mal auf die Funktionalität zugegriffen wird.

Plugin Manifest Jedes Plugin verfügt über eine eigene Plugin Manifest-Datei, welche in der Standardimplementierung im XML Format abgelegt ist.

Die wichtigsten Tags und Elemente der Plugin Konfiguration sind:

Plugin: Mit dem id Attribut wird ein eindeutiger Name für dieses Plugin vergeben. Zusammen mit der Version wird intern ein Pluginidentifier gebildet. Neben der id muss auch das tatsächliche Class-File mit der dazugehörigen Package-Struktur angegeben werden.

Attributes: Mit beliebigen Value/Key Attributen können Parameter und Attribute definiert werden, auf die man über die Framework API zur Laufzeit Zugriff erhält.

Requires: Dieser Tag spezifiziert die Abhängigkeiten. Benötigt ein Plugin ein anderes, so wird die Plugin Id des benötigten Plugins mit einem import Tag angegeben.

Runtime: Hier wird der Speicherort, an dem die Plugin Klassen zu finden sind, angegeben. Es besteht ebenfalls die Möglichkeit, JAR Files zu referenzieren.

Extension-Point: Stellt dieses Plugin Extension Points für andere Plugins zur Verfügung, müssen diese mit dem Extension-Point Tag festgelegt werden. Ebenfalls können eine Reihe an erwarteten Parametern definiert werden, die beim Verbinden zu diesem Extension-Point mitgeliefert werden müssen. Jedes Plugin, welches sich zu diesem Punkt verbinden möchte, muss ein spezielles Interface implementieren.

Extension: Soll sich ein Plugin an einem existierenden Extension Point anhängen, so wird dieser mit dem Extension Tag angegeben. Neben der Plugin-Id des Extension Point-Provider muss auch der Extension Point Name bereitgestellt werden, da ein Plugin mehrere solcher Extension Points implementieren kann.

4.3.3 Systemplugins

Die eigentliche Kernapplikation versucht, bei Systemstart die Core-Plugins zu laden. Diese Core Plugins sind im Gegensatz zu allen weiteren Plugins für das System unbedingt notwendig. Alle weiteren Plugins verbinden sich nicht zur eigentlichen Applikation, sondern werden je nach funktioneller Kategorie an einem in den jeweiligen Core-Plugins bereitgestellten Extension Point

```xml
<plugin id="Plugins.MsgBroker.XmlLoader"
          version="0.0.2"
          class="Plugins.MsgBroker.XmlLoaderPlugin.XmlLoader">

   <attributes>
      <attribute id="XMLFile"
                  value="file:///C:/Projekte/DiplArbeit/MsgServer/topics.xml"/>
   </attributes>

   <requires>
      <import plugin-id="Plugins.MsgBroker.Core"/>
   </requires>

   <runtime>
      <library id="DBLoader"
               path="file:///C:/Projekte/DiplArbeit/MsgServer/classes/"
               type="code">
      </library>
   </runtime>

   <extension-point id="PIFCommunication">
      <parameter-def id="class"/>
      <parameter-def id="name"/>
      <parameter-def id="description" multiplicity="none-or-one"/>
      <parameter-def id="icon" multiplicity="none-or-one"/>
   </extension-point>

   <extension  plugin-id="Plugins.MsgBroker.Core"
                  point-id="PIFMessageFactory"
                  id="XmlLoader">

      <parameter id="class" value="Plugins.MsgBroker.XmlLoaderPlugin.XmlLoader"/>
      <parameter id="name" value="XML Loader"/>
      <parameter id="description" value="Plugin to load Messages"/>

   </extension>

</plugin>
```

Abbildung 22: Plugin-Manifest

angehängt. Die Kernapplikation greift also nur auf diese System-Plugins und deren Funktionen zu. Für eine vollständige Funktionalität sind die folgenden 3 Core Plugins notwendig.

- Communication Core Plugin
- Message Broker Core Plugin
- Topic Core Plugin

Weiters benötigt das System mindestens ein Plugin, welches ein Kommunikationsprotokoll oder vergleichbares implementiert um eine Kommunikation mit Clients zu ermöglichen.

Communication Plugin Um eine flexible Basis für die Kommunikation zur Außenwelt zu schaffen, wurde die Kommunikationsfunktionalität in eine eigene Pluginkategorie gekapselt. Ein Communication-Plugin muss die Basisfunktionen die für eine Kommunikation zwischen Server und Client nötig sind bereitstellen. Das Plugin hat prinzipiell auf alle Services Zugriff und ist für deren korrekten Aufruf verantwortlich wobei die Aufrufe je nach Anwendungsfall (Verbinden eines Clients, Empfangen einer Nachricht, Abmelden eines Clients) einem vorgegebenen Schema folgen müssen, um einen problemlosen Betrieb sicherzustellen.
Wie die Verbindung zu einem Client aussieht oder welche Ressource einen Client repräsentieren, ist dem Plugin überlassen. So kann ein am Server angeschlossener GPS Empfänger ebenso einen Client repräsentieren wie eine Datei, ein PDA oder ein entfernter PC. Ein gültiges Communication Plugin muss ein eigenes Interface implementieren, damit es beim System als solches akzeptiert wird. Das Interface gibt vor, welche Methoden vom FMS aufgerufen werden können, jedoch nicht, welche Services von FMS für eine vollwertige Funktionalität in Anspruch genommen werden müssen. Die interessanteren Abläufe, die ein Kommunikations-Plugin abarbeiten muss, sind im Allgemeinen:

- Verbindungsaufbau (connect)
- Verbindungsabbau (disconnect)
- Anmeldung zu einem Topic (subscription)

- Abmeldung von einem Topic (unsubscription)
- Empfangen einer Nachricht (receiving)
- Senden einer Nachricht (sending)

Der FMS verwendet für die Kommunikation mit dem Client nur 3 Methoden im Plugin, welche über das Interface vorgegeben sind:

- Connect: Stellt die Verbindung zum Client her.
- Disconnect: Bricht die Verbindung zum Client ab und gibt eventuell benötigte Ressourcen wieder frei.
- SendMsgTo: Sendet eine Nachricht zum Client.

Die restlichen Abläufe sind Abläufe, die vom Client aus angestoßen werden. Um die nötige serverseitige Reaktion zu erhalten, muss das Plugin auf Services des FMS zugreifen. Die Services des FMS, welche für das Plugin interessant sind, werden über folgende Handles bereitgestellt:

- MessageFactory: Stellt den Zugriff auf die Topics und deren Daten bereit.
- ClientFactory: Stellt Zugriff auf das Client Management bereit.
- MsBroker: Stellt Methoden zum Subscriben/Unsubscriben, Senden, Publishen und einige andere bereit.

Der genauere Ablauf in den einzelnen interessanten Fällen ist wie folgt: (je nach Kommunikationsmedium und Client kann es zu kleinen Abweichungen kommen, der Grundablauf jedoch bleibt erhalten)

Connect: Das Plugin erkennt einen Client der sich zum FMS verbinden möchte, erstellt alle für die Kommunikation mit dem bestimmten Medium nötigen Ressourcen, erstellt über die ClientFactory ein neues Client Objekt mit einer eindeutigen Client ID in Form eines URI´s [(IE98], legt die Ressourcen als Attachment im Client Objekt ab und ruft im Client Object die Methode onConnect auf. Der Client ist ab diesem Zeitpunkt beim FMS registriert und kann somit die bereitgestellten Dienste nutzen.

Disconnect: Das Plugin erkennt einen Verbindungsabbruch, oder den Wunsch eines Clients die Verbindung zu beenden, leitet die nötigen Aktionen ein die für dieses Kommunikationsmedium nötig sind, gibt eventuelle Ressourcen frei und ruft die Methode disconnect im Client Objekt auf. Der Server löscht alle mit dem Client verbundenen Subscriptions und entfernt abschließend das Clientobjekt selbst.

Subscripton: Das Plugin empfängt vom Client den Wunsch ein bestimmtes Topic zu subscriben und ruft die Methode attachSubscriber im MsBroker mit dem eindeutigen Topic URI auf. Der FMS fügt die Subscription zur Subscription Datenbank hinzu. Sobald sich Topic spezifische Daten ändern erhält der Client ab nun Notifizierungen.

- UnSubscription: Das Plugin ruft äquivalent zur Subscription die Methode deleteSubscriber mit der eindeutigen URI des Topics auf. Der Server löscht den Eintrag aus der Subscription Datenbank. Der Client erhält keine Nachrichten bei Datenänderung mehr.

- Empfangen einer Nachricht: Das Plugin empfängt eine Nachricht vom Client, erstellt ein Nachrichten Objekt, und setzt den Topic URI und die entsprechenden Daten. Nach dem Erstellen des Nachrichtenobjektes wird es der Methode receiveMsg im Client Objekt übergeben. Der FMS überprüft ob der Client registriert (Connected) ist, ob der Topic URI existiert und ruft anschließend, wenn vorhanden, das Topic Plugin und dessen Funktionalität auf. Anschließend werden alle zu diesem Topic angemeldeten Clients aus der Subscription Datenbank mit den aktuellen Daten versorgt und verständigt.

- Senden einer Nachricht: Das Plugin erhält vom FMS ein Nachrichtenobjekt, codiert dieses in das vom Kommunikationsmedium unterstützte Datenformat und versendet dieses.

Topic Plugin Um serverseitige Funktionalität und spezielles Systemverhalten bei einzelnen Topics zu ermöglichen, kann jedem Topic ein spezifisches Topic Plugin zugewiesen werden. Betrachtet man das gesamte System im OSI-7-Schichten Referenzmodell, entspricht das Topic Plugin den Schichten 6 und 7 (Darstellung und Anwendung) [Wik05e]. Dieses Plugin ermöglicht es, vom üblichen Verhalten abzuweichen, indem die Standardfunktionalität

überschrieben wird. Das Plugin hat Zugriff auf alle Topic und Client relevanten Services und kann somit auch komplexere Abläufe und Kommunikationslogik, sowie Applikationslogik anstoßen. Weiters kann die Manipulation, Validierung und Authentifizierung der Daten nach Empfang einer Nachricht auf die Anforderungen abgestimmt werden. Ein selbstständiger Anstoß beim Eintreten eines bestimmten Ereignisses ist ebenfalls vorgesehen und ermöglicht das einfache Eingliedern externer ereignisorientierter Daten wie zum Beispiel E-Mails, Messenger, Datenbanken, Filesystem oder Daten von externen Devices. Ein heikler Punkt ist hierbei die Repräsentation der Daten, die nur im Zusammenspiel mit dem geeigneten Communication-Plugin sichergestellt werden kann. So muss dafür gesorgt sein, dass die Daten in einem für den Transport unterstütztem Format vorliegen oder ein geeigneter Wrapper bereitgestellt wird. Wie beim Communication-Plugin, muss auch das Topic Plugin einem spezifiziertem Interface folgen und bestimmte Methoden implementieren, die vom FMS aufgerufen werden.

- onInit: Wird aufgerufen, sobald der FMS das Topic registriert und die notwendigen Ressourcen anlegt.
- createAndShowGUI: Hier kann das Plugin für die grafische Oberfläche grafische Elemente für die Informationsdarstellung sowie für die Steuerung und Bedienung bereitstellen. Wird das Plugin vom FMS geladen, wird die zur Verfügung gestellte Oberfläche eingefügt.
- onReceive: Wird eine Topic-spezifische Nachricht von einem beliebigen Communication-Plugin empfangen, so wird diese Methode aufgerufen. Das Standardverhalten ersetzt die Topic-Daten mit den soeben empfangenen Daten.
- onSend: Werden Topic Daten an einen Client gesendet, so wird diese Methode im Plugin nach erfolgreichem Versenden ausgeführt.
- onPublish: Wie bei onSend wird nach einem kompletten Publish Zyklus, in dem die Topicdaten, an alle in der Subscription-Datenbank für dieses Topic registrierten Clients, versendet werden, das Plugin durch den Methodenaufruf in Kenntnis gesetzt.
- onPluginAddedToMsg: Wird einem Topic ein Plugin zugewiesen, werden alle anderen Plugins verständigt, um eventuelle Abhängigkeiten zwischen den Topics und Plugins auflösen zu können.

- onMsgAdd: Registriert der FMS ein neues Topic, so kann es ebenfalls sein, dass Abhängigkeiten innerhalb der Topics und Plugins existieren und die entsprechenden Module vom Neuzugang eines Topics informiert werden müssen.

- onSubscription: Subscribt ein Client zu einem Topic, wird jedes Plugin verständigt.

- onUnSubscription: Wie bei der Subscription wird eine Unsubscription den Plugins mitgeteilt.

- setData: Werden vom FMS Topicdaten gesetzt, so wird diese Methode aufgerufen. Dies ermöglicht topicspezifische Behandlung der Daten, sowie eine mögliche Validierung oder Konvertierung.

- getData: Benötigt der FMS Topicdaten um diese an Clients zu senden, ruft er diese Methode auf. Hier sollten die Daten in ein vom Communication Plugin unterstütztem Format bereitgestellt werden. (eine korrekte Anwendung dieser Anforderung muss erst mit der notwendigen Architektur bereitgestellt werden und bedarf noch erheblicher Überarbeitung).

- publish: Bevor der FMS ein Topic published, kann das Plugin entscheiden, ob das Topic tatsächlich versandt wird oder nicht.

- send: ebenfalls wie beim Publishing kann das Plugin entscheiden, ob die Topicdaten zu einem bestimmten Client gesendet werden oder nicht.

Broker Plugin Alle Erweiterungen, die weder direkt mit der Kommunikation oder einem spezifischen Topic zu tun haben, können über das Broker-Plugin Interface in den FMS eingebunden werden. Typische Funktionen für diese Gruppe an Plugins ist das Laden der Topic Definitionen beim Systemstart, dynamische Pluginverwaltung, Logging, Cluster oder Backupserver, Subscriptionverwaltung, und alle weiteren Erweiterungen, die Zugriff auf das gesamte System benötigen.Die Methoden, die durch das Interface vorgegeben werden, decken sich stark mit den Topic Plugins, da alle Nachrichten über den Message Broker laufen.

- onInit: Wird der FMS gestartet, im Speziellen das Message-Broker Core Plugin, werden in allen Broker Plugins die onInit Methode aufgerufen. Hier können zum Beispiel Topicdefinitionen geladen werden und Backups oder Verbindungen zu anderen Servern aufgebaut werden.
- createAndShowGui: Wie bei den anderen Plugins kann auch ein Broker Plugin eine grafische Oberfläche bereitstellen.
- onMsgAdd: Ein Plugin kann auf neue Topics reagieren.
- onConnect: Stellt ein Client eine Verbindung zum FMS her, so kann im Plugin eine serverseitige Funktion ausgeführt werden, wie zum Beispiel eine Authentifizierung.
- onDisconnect: Aktionen, die nach einem Disconnect eines Clients notwendig sind, können in der onDisconnect Methode realisiert werden.
- onSubscription: Wird aufgerufen wenn ein Client zu einem Topic subscribed.
- onUnsubscription: Wie onSubscription, jedoch beim Abmelden eines Topics.
- onReceive: Empfängt der FMS eine Nachricht, kann hier vor Aufruf der Topic Plugins Funktionalität eingebracht werden.
- onSend: Wurden Topic Daten zu einem Client versendet, wird die onSend Methode aufgerufen.
- onPublish: gleich wie onSend beim erfolgreichen Publishing von Topicdaten

5 Testen eines Publish/Subscribe Systems

5.1 Risiken komponentenbasierter Systeme

Der steigende Druck auf die Softwareentwicklung im Bereich Time-to-Market und Qualität führen zu dem Trend, Software nicht mehr von Grund auf zu entwickeln, sondern auf teilweise vorhandene Komponenten und Module zurückzugreifen. Speziell bei Plugin-basierten Applikationen erfolgt eine starke Unterteilung des Systems in einzelne Komponenten [Vit03]. Unter einer

Komponente versteht man ein ausführbares Codesegment, dem gewisse Interfaces zugrunde liegen. Durch den Einsatz von Component Based Software Development (CBSD) ergeben sich einige wesentliche Vorteile. Geringere Entwicklungszeit durch bereits vorhandene Komponenten Geringer Kosten durch Wiederverwendbarkeit von Komponenten Höhere Qualität, da Komponenten in verschiedenen Applikationen getestet werden Bessere Wartbarkeit, da Komponenten leicht durch verbesserte ausgetauscht werden können. Bei CBSD sind grundsätzlich 3 Parteien beteiligt und es ist notwendig, die Risiken und Anforderungen der einzelnen Stateholder zu ermitteln.

5.1.1 Komponenten Entwickler

Die Anforderungen und Risiken unterscheiden sich je nach Zielgruppe (Massenmarkt oder Individualkunden). Beim Massenmarkt muss ein Geschäftsbereich ausfindig gemacht werden, der genug Absatz verspricht. Als Risiko gilt, dass durch schlechte Planung und schlechtes Design die Komponente nicht den industriellen Anforderungen gerecht werden. Weitere Risiken bringen die unterschiedlichen Versionen, die von einer Komponente in verschiedensten Anwendungen existieren. Das Testen ist meist schwierig, da der Entwickler das schlussendliche Einsatzgebiet der Komponenten nicht kennt. Böswillige Manipulation der Komponenten durch Viren und Hacker stellen eine weitere Gefahr dar, die berücksichtigt werden muss.

5.1.2 System Entwickler

Die Risiken des Systementwicklers betreffen die Zusammenführung der einzelnen Komponenten zu einer Anwendung. Wenn der Assembler benötigte Komponenten findet, heißt das noch nicht, dass diese genau die Funktionen bieten, bzw. dass sie mit den anderen Komponenten kompatibel sind und reibungslos zusammenarbeiten. Hier gilt es abzuschätzen, ob die Entwicklung einer individuellen Komponente nicht Kosten und Ärger erspart oder ob es preiswerter ist, die Komponente vom Softwaremarkt zu erwerben.

5.1.3 Kunde

Der Kunde hat einerseits das Risiko zu tragen, dass die Applikation alle Anforderungen bewerkstelligt, als auch die Verwaltung und Wartung. Die Qualität der Applikation birgt weitere Risiken, welche sich durch die Qualität der einzelnen Komponenten ergibt. Die Schuldzuweisung bei Fehlern oder

Problemen kann sich als schwierig herausstellen, da eventuell der Assembler die Schuld auf den Entwickler schiebt und umgekehrt. Weiters besteht die Gefahr, dass es keine Komponenten gibt, die genau den Anforderungen des Kunden gerecht werden, und so der laufende Betrieb nicht reibungslos unterstützt wird. Das CBSD bringt viele Vorteile gegenüber traditioneller Softwareentwicklung, birgt aber auch neue Risiken für die beteiligten Parteien.

5.2 Testen

"*Testen ist eine der wichtigsten Möglichkeiten, um die Qualität von Software zu quantifizieren und in weiterer Folge auch zu verbessern.*" G. Myers [Mye79] definiert Testen wiefolgt: *"the process of executing a program with the intent of finding errors."*
Doch Testen hat sich mittlerweile vom klassischen Fehlerfinden auch zu einer Anzahl an Entwicklungsmethoden entwickelt wie TDD (Test Driven Development) [Mug03] und XP (Extreme Programing) [Abr03].
Softwarefehler können natürlich auch ohne Ausführung des zu testenden Programms lokalisiert werden. Man spricht hierbei von statischen Tests im Gegensatz zu dynamischen Tests die auf Ausführung basieren. Das zu testende Objekt wird oft als Item under Test (IUT) bezeichnet. Ein IUT kann ein Programm, eine Methode, ein Modul oder Softwarekomponente oder ein ganzes System darstellen. Die Komponente, die einen Test an einem IUT durchführt, wird in diesem Zusammenhang Testkomponente genannt.
Da ein Test im Allgemeinen von den Eigenschaften eines IUT stark abhängt, benötigt man eine Klassifizierung. In dieser Arbeit wird nur eine, in diesem Kontext benötigte Klassifizierung eingeführt.

- Zeitunkritische Systeme: Systeme, die keinen zeitlichen Vorgaben folgen müssen, bezeichnen wir als zeitunkritische Systeme. Es ist nur das funktional korrekte Verhalten des Systems wichtig.

- Zeitkritische Systeme: Für zeitkritische Systeme, wie Echtzeitsysteme, ist nicht nur das funktional korrekte Systemverhalten entscheidend, sondern es müssen auch zeitliche Anforderungen erfüllt werden.

- Monolithische Systeme: Monolithische Systeme bestehen nur aus Komponenten, die auf einer lokalen physikalischen Einheit (Nodes) ausgeführt werden.

- Verteilte Systeme: Die Komponenten eines verteilten Systems sind auf verschiedene Nodes aufgeteilt, wobei die einzelnen Nodes miteinander kommunizieren.

5.3 Testziel

Ein grundlegendes Ziel ist, die Tests reproduzierbar zu machen. Ein Testergebnis, welches nicht mit einem Test reproduzierbar ist, hat nur eine sehr geringe Bedeutung. Je nach gewünschtem Testziel kommen unterschiedliche Tests zum Einsatz. Die Art der Tests resultiert aus den verschiedenen Eigenschaften der IUT's.

Statische Tests: ohne Ausführung des IUT. Dies ermöglicht das Auffinden von Fehlern zu einem zeitlich früheren Entwicklungsstadion und kann so eventuell die Kosten im Vergleich zu einem Fehler der zu einem späteren Zeitpunkt gefunden wird verringern.

Die folgenden Tests sind dynamische Tests und können je nach Test Ziel noch weiter untergliedert werden.

Struktur Tests: Mit Strukturtests wird versucht, die Struktur bei der Ausführung abzudecken (Datenfluss oder Kontrollstruktur). Um dies zu erreichen, muss die interne Struktur des IUT bekannt sein. Man bezeichnet diese Tests auch als White-Box Tests.

Funktionale Tests: Ein funktionaler Test verifiziert die korrekte Funktionalität eines IUT. Der funktionale Test basiert auf der Spezifikation des IUT. Im Gegensatz zu den Strukturtests wird kein Wissen über den internen Aufbau benötigt und man spricht hier von Black-Box Tests.

Nichtfunktionale Tests: Wie bei den funktionalen Tests basieren die nicht funktionalen Tests auf der Spezifikation des IUT, jedoch werden die nicht funktionalen Anforderungen geprüft. Es gibt eine Vielzahl an nicht funktionellen Anforderungen wie Echtzeitverhalten, Zuverlässigkeit, Ökonomie. Nichtfunktionelle Tests zählen grundsätzlich zu den Black-Box Tests, wird jedoch zum Testen interne Informationen benötigt (Werte, Zeiten ...), so spricht man von Gray-Box Tests.

5.4 Test Bereich

Abhängig von der Komplexität und Größe der einzelnen IUT´s ergeben sich unterschiedliche Bereiche die getestet werden.

- Unit: Die kleinste testbare Einheit ist eine Unit. Typischerweise eine Methode oder eine Klasse in einer objektorientierten Implementierung.
- Integration: Eine untereinander fest verbundene Menge an Units, die jedoch noch kein eigenständiges System bilden.
- System: Das gesamte System wird als Testbereich angesehen.

5.5 Test Verteilung

- Lokale Verteilung: Der Test besteht aus einer Testkomponente auf einem Node, der auf das IUT über ein oder mehrere Interfaces zugreift.
- Verteilter Test: Ein verteilter Test besteht aus mehreren Testkomponenten, die über mehrere Nodes verteilt sind, welche gleichzeitig ausgeführt werden. Um einen planmäßigen Testablauf zu erhalten, müssen die Testkomponenten untereinander koordiniert sein.

5.6 Testablauf

Das Ergebnis eines Tests ist im Allgemeinen ein Testergebnis, welches Auskunft darüber gibt, ob ein System, oder genauer ein IUT, die spezifizierte Funktion erfüllt oder nicht. Die Spezifikation enthält eine Vielzahl an verschiedenen Anforderungen und bildet somit die Basis der einzelnen Tests. Aus der Spezifikation werden test purposes abgeleitet, welche auf einzelne Anforderungen der Spezifikation ausgerichtet sind. Die tatsächlich notwendigen Aktionen um die Anforderung zu testen werden in einem test case realisiert. Alle test cases bilden eine test suite. Eine test suite wird auf das zu testende System angewendet und liefert als Ergebnis ein Testprotokoll, welches die Resultate der einzelnen Tests beinhaltet.

5.7 Testkriterien

Aufgrund der umfangreichen Möglichkeiten, die ein verteiltes System bietet, ergeben sich verschiedene Kriterien und Anforderungen, die getestet werden

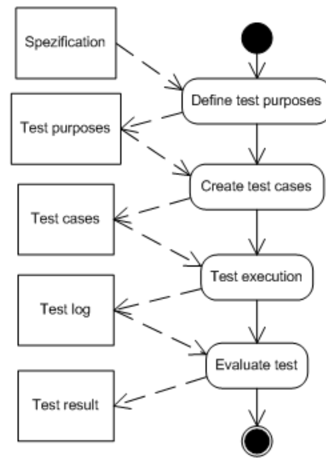

Abbildung 23: Testablauf

können.

5.7.1 Workflow

Ein Workflow ist eine Aktivität, die eine koordinierte Ausführung von mehreren Aufgaben (Tasks) einbezieht, die von verschiedenen Verarbeitungsentitäten bearbeitet werden. Solche Verarbeitungsentitäten (processing entities), auch Ausführungsinstanzen (execution instance) genannt, umfassen z.B. Menschen, Computersystemen und Applikationen [KS95] In verteilten Systemen sind oft mehrere Nodes an der koordinierten Ausführung einer Aufgabe beteiligt. Diese Workflows sind für die Gesamtfunktion von wesentlicher Bedeutung und benötigen deshalb besonderes Augenmerk bei den Tests.

5.7.2 Message Delivery

Viele nachrichtenbasierte Systeme garantieren eine zuverlässige Nachrichtenzustellung. Auch im Falle dieser garantierten Zustellung empfiehlt es sich, zu testen, ob tatsächlich alle Nachrichten zugestellt werden. Ist keine garantierte Nachrichtenzustellung gewährleistet, gewinnt dieser Test zusätzlich an Bedeutung [SD04]. Ein Protokoll, welches zuverlässige Nachrichtenzustellung garantiert, wird in [IBM04] spezifiziert.

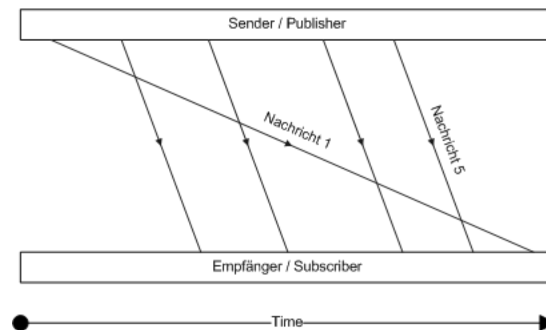

Abbildung 24: Nachrichtenreihenfolge

5.7.3 Messge Ordering

Nachrichten mit gleicher Priorität die mit einem gleichen Übertragungsmechanismus übertragen werden, müssen in derselben Reihenfolge beim Empfänger eintreffen wie diese vom Sender versandt wurden [SD04].

5.7.4 Performance/Throughput

Performance bezeichnet in der Informatik die Effizienz von Programmen und Hardware bezüglich des Resourcenverbrauchs und der Qualität der Ausgabe. Meistens ist mit Performanz die Datenrate gemeint, also die Menge von Daten, die innerhalb einer bestimmten Zeitspanne verarbeitet werden kann. Im Gegensatz zur Zeitkomplexität bezieht sich die Performanz nicht auf einen abstrakten Algorithmus, sondern auf ein konkretes Programm auf einem konkreten Computer [Wik05f]. Im Zusammenhang mit nachrichtenorientierten Systemen sind unter anderem folgende Faktoren in Bezug auf Performance von Interesse:

- Versendete Nachrichten pro Zeiteinheit: wesentliche Leistungseigenschaft aus Applikationssicht, die angibt, wie viele Nachrichten in einer gewissen Zeitspanne versendet werden können. Die Nachrichtengröße muss für einen repräsentablen Wert mitspezifiziert werden.

- Empfangene Nachrichten pro Zeiteinheit: gibt an wie viele Nachrichten in einer Zeitspanne empfangen werden können.

- Versendete Bytes pro Zeiteinheit: gibt mehr Auskunft über die Kapazität des Kommunikationsmediums im Gegensatz zu dem abstrakteren Werte Nachrichten pro Zeiteinheit
- Empfangene Bytes pro Zeiteinheit
- Bearbeitete Requests pro Zeiteinheit
- Durchschnittliche Zeit für einen Request

5.7.5 Response Time

Es gibt verschiedene Definitionen, die stark vom Kontext abhängig sind. Im Allgemeinen handelt es sich um eine obere Schranke für das Zeitintervall vom Ende eines Ereignisses bis zum Beginn der Reaktion auf dieses Ereignis.

5.7.6 Reaction Time

Bei nachrichtenorientierten Systemen versteht man unter reaction time, oft auch als Latency bezeichnet, die Zeit, die zwischen dem Eintreffen eines Ereignisses e und der Terminierung der aus e resultierenden Aktionen vergeht. Die reaction time ist somit die Summe aus Response time und der Ausführungszeit [Kho02][GDW04].

5.7.7 Reliability - Availability

Unter Zuverlässigkeit versteht man die Fähigkeit eines Systems, für eine gegebene Zeit korrekt zu arbeiten. Dabei wird vorausgesetzt, dass das System zu Anwendungsbeginn korrekt ist und nur Ausfälle zur Unkorrektheit führen können. Reliability ist eine Kennzahlm, die Auskunft über die Zuverlässigkeit des Systems gibt. Die Availability entspricht der Wahrscheinlichkeit für korrekte Funktion im Sinne der Zuverlässigkeit.

5.7.8 Realtime Properties

Wird von einem System ein definiertes zeitliches Verhalten gefordert, kann man laut [Kop97] die Echtzeiteigenschaften in soft- und hard Realtime Properties unterteilen.

- Hard Realtime Properties: sind zeitliche Vorgaben, die unter allen Umständen eingehalten werden müssen. Eine Nachricht, die diese Vorgaben nicht erfüllt, ist eine falsche Nachricht da sie aufgrund ihrer Verzögerung womöglich bereits nicht mehr relevante Information besitzt.

- Soft Realtime Properties: sind zeitliche Vorgaben, die üblicherweise oder zu einem gewissen prozentuellen Anteil erfüllt werden sollen. Eine verzögerte Nachricht ist auch aufgrund ihrer Verspätung noch gültig. Soft Realtime Systeme sind wesentlich leichter zu implementieren, jedoch ist die mathematische Beschreibung dieser komplexer, da statistische Aspekte mitspielen.

5.8 Funktionale Tests

Gegenstand von funktionalen Tests ist die Überprüfung des Systemverhaltens auf Basis der Spezifikation. Bei funktionalen Tests werden zeitliche Aspekte üblicherweise komplett außer acht gelassen, rein die richtige und korrekte Erfüllung der Kriterien wird bewertet. Es gibt verschiedene Kategorisierungen von funktionalen Tests, hier wird die in der Literatur [Rät04] oft verwendete Unterscheidung in Black und White-Box Tests getroffen.

5.8.1 Funktionsorientiertes Testen (Black-Box)

Bei dieser Testmethode betrachtet man das zu testende System als Black Box, das heißt, es wird keinerlei Information über den Aufbau des IUT für den Test verwendet. Bei funktionalen Tests werden verschiedene Eingangsmuster angelegt und die resultierenden Ausgangsmuster mit der Spezifikation verglichen. Stimmt die Antwort auf die Eingangsmuster eines Testfalls mit der Spezifikation überein, entspricht das System in diesem Fall der Spezifikation. Im anderen Fall liegt offensichtlich ein Fehler vor, der behoben werden muss, um den definierten Anforderungen zu entsprechen. In [Mye79] findet sich eine gute Definition für die Black-Box-Methode: "*One way to examine this issue is to explore a testing strategy called black-box, data-driven, functional or input/output-driven testing. In using this strategy, the tester views the program as a black box. That is, the tester is completely unconcerned about the internal behaviour and structure of the program. Rather, the tester is only interested in finding circumstances in which the program does not behave according to its specifications. Test data are derived solely from the specification*

(i.e., without taking advantage of knowledge of the internal structure of the program)."
Die internen Programmstrukturen sind nicht von Bedeutung und werden außer Acht gelassen. Der größte Vorteil dieses Ansatzes ist die Konzentration auf die Spezifikation und damit den Benutzeranforderungen. Die totale Vernachlässigung der internen Programmstruktur kann jedoch auch einen Nachteil darstellen. Diese Problematik greift das White-Box-Testen auf.

5.8.2 Strukturorientiertes Testen (White Box)

Diese Methode orientiert sich besonders an der Struktur, also der konkreten Implementierung, des IUT beim Entwurf der Testfälle. White-Box-Testen ist das Gegenstück zum im letzten Kapitel vorgestellten Black-Box-Testen. Die Überdeckung ist eine der wichtigsten Qualitätsmerkmale. Darunter versteht man den Prozentsatz, der vom Testfall abgedeckten Testbasis, also Anweisungen bzw. Anforderungen. Es gibt bestimmte Überdeckungsmaße, die sich auf den gesamten Satz von Testfällen beziehen und in erster Linie für strukturelle Tests (White-Box) angewandt werden. Das Maß ergibt sich aus der Überdeckung aller möglichen Kombinationen von Aktionen, Bedingungen, Pfaden und Anforderungen. Die Anzahl dieser Kombinationen bestimmt dadurch bei einer bestimmten, angestrebten Abdeckung die Anzahl der dafür nötigen Testfälle. Von Vorteil ist die Möglichkeit, die Testgenauigkeit zu variieren, da die Überdeckung sich auf den vorliegenden Code und nicht auf Anforderungen bezieht. Außerdem erlaubt der White-Box-Ansatz das Testen von Anforderungen, die nicht explizit genannt, aber im Programm vorhanden sind [Wie03]. Es oft sinnvoll, die Black-Box- und White-Box-Methode miteinander zu kombinieren. Der strukturelle Ansatz kann dabei helfen, Testfälle, die sich auf denselben Codeblock beschränken, zusammenzulegen und nicht explizit geforderte Funktionen miteinzubeziehen. Dadurch erhält man meist nicht nur kleinere Testsuiten, sondern auch eine bessere Anpassung der Tests an das gegebene Programm.

5.9 Nichtfunktionale Tests

Bei nichtfunktionalen Tests werden die nichtfunktionalen Anforderungen des Systems getestet. Nichtfunktionale Eigenschaften stehen unmittelbar im Zusammenhang mit der positiven Erfüllung der funktionalen Anforderungen. Um nichtfunktionale Eigenschaften testen zu können, wird also ein funktio-

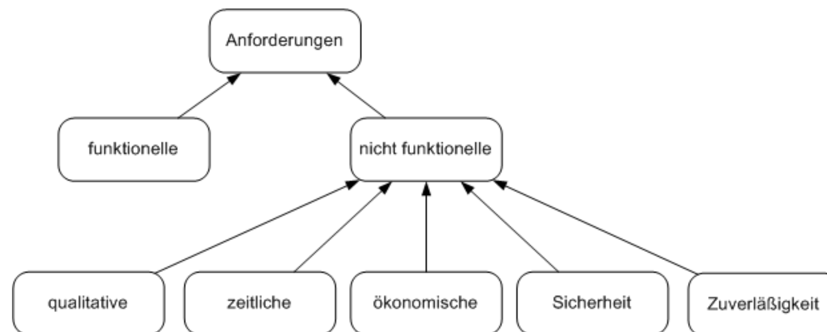

Abbildung 25: Anforderungen

nal korrektes Verhalten gefordert. Es gibt eine Menge an nichtfunktionalen Anforderungen, wobei die Abgrenzung zu funktionalen Anforderungen nicht scharf ist.
Generell spielt Zeit bei Last- bzw. Stresstest eine wichtige Rolle. Daraus folgt jedoch, dass diese Art von Tests nicht auf alle Programme anwendbar ist. Diese Testform wird vor allem auf Programme angewendet, die mit wechselnder Belastung konfrontiert werden. Da meist die zu testenden Systeme sehr leistungsstark sind, benötigt man oft erhebliche Ressourcen, um diese an die Leistungsgrenzen zu bringen. Um eine entsprechende Last aufbringen zu können, kann man verteilte Tests nach dem Schema von Distributed Denial of Service Attacken (DDoS) einsetzen [Wik05b].

5.9.1 Heavy Load Test

Heavy Load Tests werden zu der Klasse nichtfunktionale Testmethoden gezählt. Bei diesem Test wird das IUT unter Last gesetzt, wobei unter Last eine Vielzahl an Anfragen an das System zu verstehen ist. Die Belastung an das System sollte so gewählt sein, dass die Leistungsreserven des Systems beinahe voll ausgeschöpft werden. Daraus ergibt sich die Möglichkeit, den IUT auf Laufzeiteigenschaften wie Speicherplatzbedarf, Hardwareauslastung und das dynamische Verhalten zu analysieren.
In der Literatur wird oft nicht zwischen Heavy Load Test und Stress Test unterschieden.

5.9.2 Stress Test /Peak Test

Unter Stresstest versteht man das temporäre Überschreiten der Leistungsfähigkeit des Systems. Dadurch wird ein heikler Systemstatus erreicht. Das Ziel von Stresstests ist es, Fehler in der gleichzeitigen Ausführung bzw. in der parallelen Ausführung von Systemfunktionen zu finden. Weiters ist es von Interesse ob sich das System bei Überlastung stabil verhält und ob es nach diesem Peak-Load wieder in einen stabilen Zustand übergeht. Die Funktionalität während der Überlastung ist ebenfalls von entscheidender Bedeutung und es stellen sich einige Fragen, die durch diesen Test beantwortet werden können.

- Gehen Requests bei Überlastung des Systems verloren?
- Wie lange benötigt das System um die ausstehenden Requests abzubauen?
- Wie wirkt sich diese Überlastung auf den Throughput aus?

5.9.3 Performance Test

Der Performancetest ist im Grunde ein Stresstest, der dazu dient, festzustellen, ab welchem Zeitpunkt das System weniger Requests abarbeiten kann, als eintreffen. Wie ein System auf das Überschreiten der Leistungsgrenze reagiert ist sehr unterschiedlich und hängt natürlich von der Systemarchitektur ab. Die Resultate eines Performance Tests sind als grundlegende Systemeigenschaft Ausgangsparameter für weitere Tests. Mit dieser Größe können auch Systemverbesserungen quantifiziert werden. Es gibt eine Menge an Parametern, welche direkten Einfluss auf die Ergebnisse eines Performance Tests haben. In folgender Tabelle werden exemplarisch einige angeführt [GDW04].

Systembelastung	Anzahl der Clients; Datenmengen; Frequenz der Client Requests
Physikalische Ressourcen	Anzahl der CPU´s; Speichergröße; Netzwerkbandbreite
Middleware	Thread Pool Size; JVM Heap Size; Message Queue Buffer Size; Cache Size
Applikationsspezifisch	RPC; Asynchrone Nachrichtenübertragung; Synchrone Nachrichtenübertragung;Backupmanagement

Tabelle 4: Einflussparameter

5.10 Konkrete Testdruchführung am FMS

Bei der praktischen Testdurchführung am FMS wurde hauptsächlich auf die nichtfunktionalen Anforderungen eingegangen. Um die Tests möglichst realitätsnahe zu gestalten wurde eine anwendungsnahe Struktur gewählt.

5.10.1 Teststruktur

Die gewählte Struktur entspricht einem verteilten Test. Das bedeutet, dass eine Reihe an spezieller Test Clients mit dem zu testenden System verbunden werden, wobei jeder dieser Clients über die allgemeinen Dienste des Servers mit einer Koordinationseinheit kommuniziert. Die Koordination erfolgt mittels eines serverseitigen Plugins. Für die Testkoordination wird eine Reihe an speziellen Topics verwendet. Näheres wird weiter unten erläutert. Um den Testablauf gezielt auf bestimmte Testkriterien abzustimmen, werden unterschiedlich parametrisierte Tests benötigt.

Test Runs / configuration Da beim Testen des Gesamtsystems verschiedene Testkriterien auf ihre Richtigkeit getestet werden müssen, erfolgt eine Unterteilung des Gesamttests in verschiedene Test Runs. Jeder dieser Test Runs besitzt eine unterschiedliche Konfiguration der Testparameter. Damit kann der Testablauf gezielt gesteuert werden und somit verschiedene Testkriterien auf mögliche Schwachstellen überprüft werden. Wesentlich beim Test-

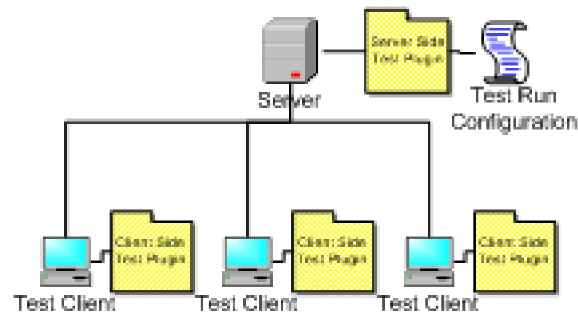

Abbildung 26: Teststruktur

design ist es, unterschiedliche Systemteile mit unterschiedlichen Leistungsforderungen beaufschlagen zu können. Dadurch erreicht man, mithilfe einer Gruppe von Test Runs ein Test Szenario abbilden und realisieren zu können. Die dazu benötigten Testparameter sind speziell auf dieses System abgestimmt. Die Parametrisierung erfolgt in einem externen XML File. Eine Beispielkonfiguration eines Testruns sieht wie folgt aus:

```
<?xml version="1.0" encoding="UTF-8"?>
<testrun nr="7">
   <topics random="false">10</topics>
   <datasize random="true">1024</datasize>
   <subscriptions random="false">10</subscriptions>
   <messages random="true">1000</messages>
   <messagedelay random="true">5</messagedelay>
   <duration>0</duration>
   <clients>0</clients>
   <clienturl>uri://systemtest/clients#testclient</clienturl>
   <topicurl>uri://systemtest/topics#testtopic</topicurl>
   <logdir>log.txt</logdir>
   <testtype all2one="false" one2one="true" all2all="false" all2random="true"/>
</testrun>
```

Abbildung 27: Konfiguration Test-Run

Jeder Test Run besitzt eine Reihe an Attributen, mit denen ein bestimmtes Test- und Systemverhalten erreicht werden kann. Die meisten Parameter können zusätzlich mit einer stochastischen Funktion belegt werden, wobei der eingestellte Wert dann den Maximalwert widerspiegelt. Folgende Testparameter können konfiguriert werden:

- Anzahl der Topics, die ein Client published: legt die Anzahl der verschiedenen Topic fest, unter deren ein Client Nachrichten versendet.
- Größe der Testnachrichten: Datengröße der Testnachrichten in Byte. Mit dieser Einstellung ermöglicht man, dass die Bandbreite des Übertragungsmediums unterschiedlich stark belastet wird. Auch eventuelle Bufferüberläufe können so provoziert werden.
- Anzahl der Topics die ein Client subscribed: Je mehr Topics ein Client subscribed, desto mehr Nachrichten muss der Server zu den einzelnen Clients versenden.
- Anzahl der Messages die in Client versendet: Es kann einerseits die Anzahl der Nachrichten bzw. andererseits die Zeitdauer, die ein Test dauert festgelegt werden.
- Testdauer: Solange die Testdauer noch nicht erreicht ist, werden Nachrichten mit der für den aktuellen Test Run geltenden Konfiguration versendet.
- Verzögerung zwischen den jeweiligen Messages: Mit diesem Wert kann eine Zeitverzögerung zwischen dem Versenden zweier Nachrichten aktiviert werden.
- Anzahl der notwendigen Clients: Ist ein Test Run für eine bestimmte Anzahl an Testclients vorgesehen so startet der Test erst, wenn die nötige Anzahl an Test Clients mit dem Server verbunden ist.
- Testtyp: Mit dem Testtyp kann eines von mehreren Testszenarien ausgewählt werden. Es stehen folgende Szenarien zur Verfügung: one2one, one2many, many2many, exponential. Nähere Details zu den Testszenarien werden weiter unten angeführt.
- Client Identifikations-URI: Konfiguration der Clientnamen, um eventuell 2 Test Plugins mit unterschiedlichen Clients laufen lassen zu können.
- Testnachrichten URI: Namen der Test Topics für diese Instanz vom Test Plugin.
- Logfile: Name und Pfad des Logfiles

Server Side Aufgrund der plugin-basierten Architektur kommt auf der Serverseite ein Test-Plugin zum Einsatz, welches über das vorher behandelte externe XML-File konfiguriert wird. Dieses Plugin hat die Aufgabe, die Test Run Konfigurationen an die jeweiligen Test Clients zu schicken. Ebenfalls stellt das Testplugin sicher, dass die notwendigen Ressourcen am Server, genauer die richtige Anzahl an Topics, bereitgestellt wird. Sind die Parameter des aktuellen Test Runs zu allen Clients übertragen und sind genügend Test Clients mit dem Server verbunden, so wird über ein eigenes Start Topic der Testlauf bei den einzelnen Clients angestoßen. Das Testplugin überprüft ab diesem Zeitpunkt alle am Server eintreffenden Nachrichten. Enthält eine Nachricht als Topic URI den in der Konfiguration spezifizierten Testnachrichten URI, so erfolgt eine Protokollierung ins Logfile. Ebenfalls werden aus der Testnachricht Metadaten ausgelesen und ebenfalls mitprotokolliert.

Client Side Wie auf der Serverseite ist auch clientseitig ebenfalls ein Plugin zur Testdurchführung zuständig. Dieses unterscheidet sich jedoch erheblich vom serverseitigen Plugin. Die Aufgabe des clientseitigen Plugins ist es, die von Server versandte Konfiguration zu übernehmen und auf das Startkommando zu reagieren. Wird die Konfiguration des Testlaufes empfangen, so werden clientseitig die nötigen Topics angelegt und falls erforderlich die Subscriptions zu den einzelnen Topics an den Server gesandt. Wird ein Start Event empfangen, so wird von jedem Client ein Thread erstellt, der mit der zu diesem Test Run passenden Konfiguration den Testlauf abarbeitet. Gleichzeitig werden die vom Client empfangenen Nachrichten ausgewertet und in einem Logfile protokolliert. Die Testnachrichten der Clients werden mit einer laufenden Nummer und mit der Gesamtanzahl der Nachrichten, die dieser Client sendet, versehen. Dies ermöglicht dem serverseitigen Plugin, die Vollständigkeit (Message Delivery) und die Reihenfolge (Message Ordering) zu verifizieren.

5.10.2 Testnachrichten

Die Testnachrichten werden aufgrund der Konfiguration entweder auf eine fixe oder zufallsverteilte Größe gesetzt. Über die Nachrichtengröße kann man Bandbreitenengpässe des Übertragungsmediums nachstellen und somit eventuelle Engpässe lokalisieren. Dabei spielen unter anderem die physikalischen und parameterabhängigen Eigenschaften des Übertragungsprotokolls und des

Mediums eine wesentliche Rolle (wie z.B. der MTU Wert bei TCP/IP - Maximum Transfer Unit).

5.10.3 Logging / Measurements

Um aus den Tests aussagekräftige Resultate zu erhalten, ist ein umfangreiches Logging und Monitoring notwendig. Das Logfile selbst enthält nur elementare Informationen, die Auswertung dieser Daten erfolgt gesondert. Das für dieses System gewählte Logfile enthält die folgenden Daten

Name	Beschreibung	Mögliche Werte
Ereignistyp	Gibt an, um welches Ereignis es sich handelt	R - Receive P - Publish S - Send T - Timestamp
Timestamp	Fortlaufende Zeit in [ms] seit Teststart	[ms]
Messagename	Name des Topics, der dieser Message zugeordnet ist	M: #uri
Clientname	Clientname, der für dieses Event zuständig ist	C: #uri
Metadaten	Daten der Testnachricht	D: #daten

Tabelle 5: Logeinträge

Ein Auszug aus einem Logfile sieht zum Beispiel folgendermaßen aus:

```
R 7125  C:URN://xmlrpc/clients#client_1 M:uri://systemtest/topics#testtopic0  D:18 of 100
S 7141  M:uri://systemtest/topics#testtopic0  C:URN://xmlrpc/clients#client_3
S 7156  M:uri://systemtest/topics#testtopic0  C:URN://xmlrpc/clients#client
S 7172  M:uri://systemtest/topics#testtopic0  C:URN://xmlrpc/clients#client_2
S 7188  M:uri://systemtest/topics#testtopic0  C:URN://xmlrpc/clients#client_1
P 7188  M:uri://systemtest/topics#testtopic0
R 7188  C:URN://xmlrpc/clients#client_2 M:uri://systemtest/topics#testtopic0  D:20 of 100
S 7188  M:uri://systemtest/topics#testtopic0  C:URN://xmlrpc/clients#client_3
S 7203  M:uri://systemtest/topics#testtopic0  C:URN://xmlrpc/clients#client
S 7219  M:uri://systemtest/topics#testtopic0  C:URN://xmlrpc/clients#client_2
S 7219  M:uri://systemtest/topics#testtopic0  C:URN://xmlrpc/clients#client_1
P 7219  M:uri://systemtest/topics#testtopic0
R 7219  C:URN://xmlrpc/clients#client_3 M:uri://systemtest/topics#testtopic0  D:19 of 100
S 7234  M:uri://systemtest/topics#testtopic0  C:URN://xmlrpc/clients#client_3
S 7250  M:uri://systemtest/topics#testtopic0  C:URN://xmlrpc/clients#client
S 7250  M:uri://systemtest/topics#testtopic0  C:URN://xmlrpc/clients#client_2
S 7266  M:uri://systemtest/topics#testtopic0  C:URN://xmlrpc/clients#client_1
P 7266  M:uri://systemtest/topics#testtopic0
R 7266  C:URN://xmlrpc/clients#client M:uri://systemtest/topics#testtopic0  D:17 of 100
S 7281  M:uri://systemtest/topics#testtopic0  C:URN://xmlrpc/clients#client_3
S 7359  M:uri://systemtest/topics#testtopic0  C:URN://xmlrpc/clients#client
S 7359  M:uri://systemtest/topics#testtopic0  C:URN://xmlrpc/clients#client_2
T 7375  R 6 P 24
```

Abbildung 28: Konfiguration Test-Run

5.10.4 Test Szenarios

One-to-One Szenario Beim One2One Test subscribed jeder Client genau ein Topic. Dieses Topic wird ausschließlich von diesem einen Client subscribed. Ebenso published jeder Client genau auf dem ihm zugeordneten Topic. Dies hat zur Folge, dass der Server bei den Publish-Transaktionen nur zu dem Client sendet, von dem die Nachricht ausgegangen ist.

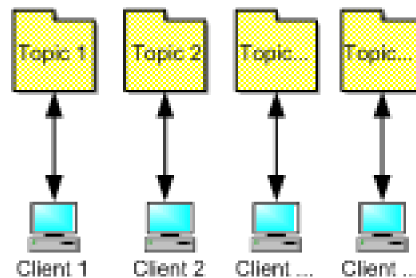

Abbildung 29: One-to-One Szenario

Für einen Testlauf ergibt sich eine Gesamtanzahl der Nachrichten M, die gesendet werden durch folgenden Zusammenhang:

$$M = 2 * \sum_{c=0}^{AnzahlClients} P_c^t$$

wobei P_c^tdie Anzahl der von Client c versendeten Nachrichten auf Topic t liefert

One to Many Im Unterschied zum One-to-One Szenario published beim One-to-Many jeder Client auf unterschiedliche Topics, jedoch sind die Clients zu keinem Topic subscribed. Dies bedeutet, dass der Server nur Messages empfängt und verarbeitet jedoch keine Messages an die Clients zurückschickt.

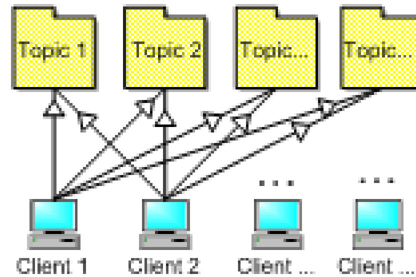

Abbildung 30: One-to-Many Szenario

In diesem Fall ergibt sich für die Gesamtanzahl Mder Nachrichten der Zusammenhang

$$M = \sum_{t=0}^{AnzahlTopics} \sum_{c=0}^{AnzahlClients} P_c^t$$

Many to Many In diesem Test subscribed jeder Client zu mehreren Topics und published auch auf unterschiedlichen Topics. Die Anzahl der versendeten Nachrichten steigt mit zunehmender Client Anzahl und Topic Anzahl stark an. Dies ist ein applikationsorientiertes Szenario und entspricht dem eigentlichen Zweck eines Publish/Subscibe Systems.

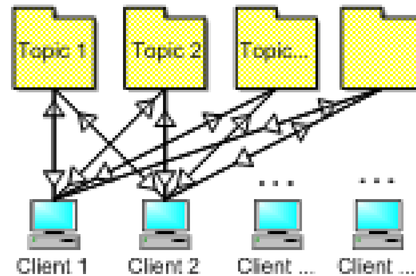

Abbildung 31: One-to-One Szenario

Für die Anzahl M der versendeten Nachrichten ergibt sich beim Many-to-Many Szenario folgender Zusammenhang:

$$M = \sum_{t=0}^{AnzahlTopics} \sum_{c=0}^{AnzahlClients} \left(S_t * P_c^t + 1\right)$$

wobei S_t die Anzahl der zum Topic t subscribten Clients liefert

Exponentielles Szenario Dieser Test entspricht einem Stress Test, bei dem die Nachrichtenanzahl in Abhängigkeit des Testfortschrittes exponentiell anwächst. Im Gegensatz zu den bisher vorgestellten Szenarien ist dieses nicht deterministisch. Alle Clients subscriben zu einem gemeinsamen Topic. Wie bei einem Schneeballprinzip beginnt ein Client eine Nachricht unter diesem Topic zu publishen. Erhält ein Client eine Notifizierung, so published dieser ebenfalls eine Nachricht unter diesem Topic. Somit resultierten aus jeder gesendeten Nachricht weitere m Nachrichten, wobei m genau der Anzahl der Clients entspricht.
Es ergibt sich also folgender Zusammenhang.

$$M(n) = C^n + 1$$

wobei n dem Testfortschritt entspricht.

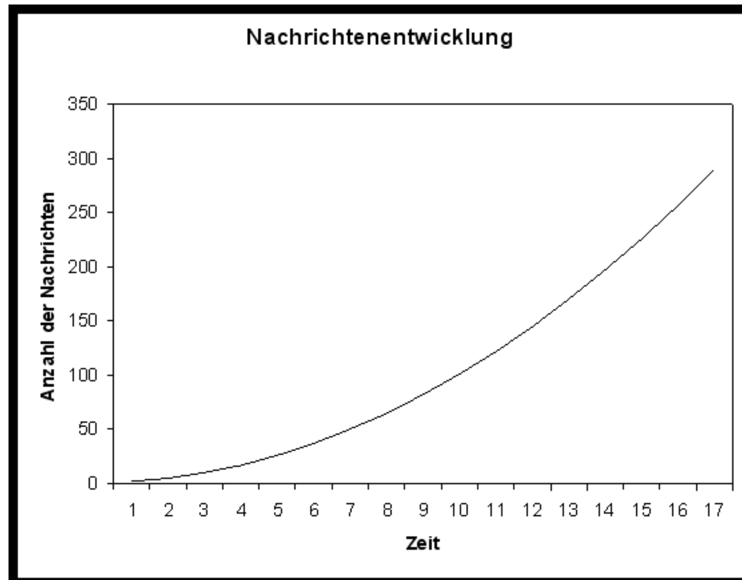

Abbildung 32: Theoretische Nachrichtenentwicklung

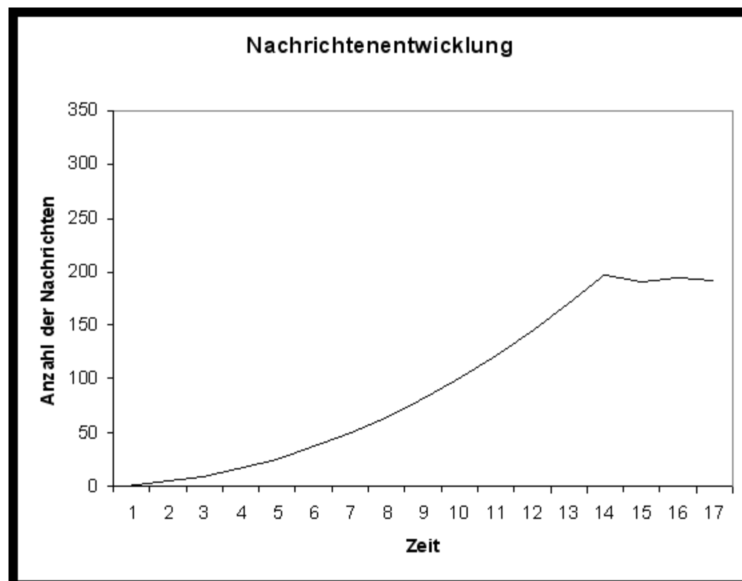

Abbildung 33: Tatsächliche Nachrichtenentwicklung

Datengröße[byte]	Nachrichten / Sekunde	Übertragungsrate [kb/s]
128	40 - 48	5 - 6
1024	40 - 48	50 - 60
4096	24 - 30	96 - 120

Tabelle 6: Performance Ergebnisse

Dieses Szenario eignet sich, um die Leistungsgrenzen des Systems auszuloten. Die Anzahl der pro Zeiteinheit versendeten Nachrichten wird nach einiger Zeit die Leistungsgrenze erreichen und sich auf einem annähernd konstanten Wert einpendeln, vorausgesetzt die Nachrichtenlänge bleibt konstant.

5.10.5 Testergebnisse

Nach dem Ausführen der ersten Tests wurde sofort das Übertragungsprotokoll als Flaschenhals identifiziert. Ein Kommunikations-Plugin stellt den Datenaustausch über XML-RPC zur Verfügung. Die Übertragung der Daten erfolgt mittels XML über das HTTP Protokoll. Dabei kommt sowohl auf Client- als auch auf der Serverseite ein integrierter Webserver zum Einsatz, der die XML-Daten auf einen RPC (Remote Procedure Call) umleitet. Die Übertragung erfolgt synchron, das heißt, dass eine Sendetransaktion wartet, bis die entfernte Methode erfolgreich beendet wird und den entsprechenden Rückgabewert zurückgibt.
Ein RPC-Aufruf im lokalen LAN benötigt laut Logfile zwischen 15 und 20ms bei einer Datennutzgröße von 1024byte. Daraus ergibt sich eine theoretische Übertragungsleistung von 50 bis maximal 75 Nachrichten pro Sekunde und gleichzeitig eine Bandbreitenbelastung von 50 bis 75Kb/Sekunde. Leistungsdaten in Abhängigkeit der Datengröße ergeben sich wie folgt.
Abgesehen von den Leistungsdaten, die unter den Erwartungen blieben, wurde ein noch größeres reproduzierbares Problem beim Ausführen des exponentiellen Szenarios entdeckt. Nach ca. 70 Sekunden liefert der XML-RPC Prozeduraufruf eine Exception. Es scheint, als ob nach einer gewissen Zeit das Schließen der Verbindung zum Webserver länger dauert, als der durch den erneuten Aufruf geforderte Verbindungsaufbau. Eventuell ist dies ein Bufferüberlauf im Connectionpool, jedoch sind dies nur Mutmaßungen und sind nicht Gegenstand dieser Arbeit.
Aufgrund der Nachrichtenreihenfolge und laufender Nummerierung wurde

```
java.lang.Exception: Error sending Message
        at Plugins.Communication.XmlRpcPlugin.XmlRpc.sendMsgTo(XmlRpc.java:216)
        at Client.ClientMain.sendMessage(ClientMain.java:132)
        at ClientPlugins.SystemTest.ClientSystemTestPlugin$2$1$1.run(ClientSystemTestPlugin.java:88)
Caused by: java.net.BindException: Address already in use: connect
        at java.net.PlainSocketImpl.socketConnect(Native Method)
        at java.net.PlainSocketImpl.doConnect(PlainSocketImpl.java:333)
        at java.net.PlainSocketImpl.connectToAddress(PlainSocketImpl.java:195)
        at java.net.PlainSocketImpl.connect(PlainSocketImpl.java:182)
        at java.net.SocksSocketImpl.connect(SocksSocketImpl.java:364)
        at java.net.Socket.connect(Socket.java:507)
        at java.net.Socket.connect(Socket.java:457)
        at java.net.Socket.<init>(Socket.java:365)
        at java.net.Socket.<init>(Socket.java:178)
        at org.apache.xmlrpc.LiteXmlRpcTransport.initConnection(LiteXmlRpcTransport.java:149)
        at org.apache.xmlrpc.LiteXmlRpcTransport.sendXmlRpc(LiteXmlRpcTransport.java:81)
        at org.apache.xmlrpc.XmlRpcClientWorker.execute(XmlRpcClientWorker.java:71)
        at org.apache.xmlrpc.XmlRpcClient.execute(XmlRpcClient.java:193)
        at org.apache.xmlrpc.XmlRpcClient.execute(XmlRpcClient.java:184)
        at org.apache.xmlrpc.XmlRpcClient.execute(XmlRpcClient.java:177)
        at Plugins.Communication.XmlRpcPlugin.XmlRpc.sendMsgTo(XmlRpc.java:211)
```

Abbildung 34: RPC-Exception

ein fehlender Lock im Programmcode entdeckt, der beim Senden und gleichzeitigem Empfangen einer Nachricht die Nachrichtendaten überschrieb. Dies resultierte, dass eine Nachrichtennummer des Öfteren gesendet wurde, die Gesamtanzahl jedoch korrekt war.

Testing can only show the presence of errors, not their absence.

6 Fazit

Die Entwicklung gerade im Bereich der verteilten Systeme schreitet unaufhaltsam voran. Die Nachfrage an hochflexiblen, skalierbaren und integrationsfähigen Lösungen gibt dabei den Kurs der zukünftigen Entwicklung vor. Einzellösungen und Spezialsoftware werden zunehmend von zentralen Plattformen abgelöst welche die geforderten Dienste bereitstellen.

Think Big - Start Small Middleware wird immer mehr zu einer zentralen IT-Basis für Unternehmen. Die hervorragende Skalierbarkeit ermöglicht schnelles Wachstum des Unternehmens, sowie den einfachen Ausbau um neue Services. Die Zeiten wo der Einsatz von Middleware lediglich großen Firmen

vorbehalten ist, sind vorbei. Immer mehr kleine und mittlere Unternehmen nutzen die Technologie zur Schaffung einer zukunftsorientierten IT Plattform.

In der Ordnung liegt die Kraft Die Strukturierung von IT Systemen in einzelne und überschaubare Services, die durch Middleware verbunden sind, schafft einen besseren Überblick. Die Wartbarkeit und Weiterentwicklung wird wesentlich vereinfacht. Auch die administrative Verwaltung wird durch die Unterteilung in einzelne Komponenten und Services positiv beeinflusst.

Eine Lösung - Viele Möglichkeiten Viele Unternehmen wandeln oder erweitern ihre Marktpräsenz bzw. bieten mit der Zeit zusätzliche Leistungen an. Die einfache Erweiterung von Middleware unterstützt ein Unternehmen um neue Marktbereiche zu erschließen. Auch die unternehmensübergreifende Einbindung von Produktion, Planung, Logistik, Rechnungswesen oder Personalmanagement ist für ein zukunftsorientiertes Unternehmen ein wichtiges Argument für den Einsatz einer Middleware.

Kommunikation ohne Grenzen Die uneingeschränkte Kommunikation über die Grenzen von Raum und Technologie hinweg unterstützt jedes Unternehmen mit mehreren Standorten. Auch die direkte Einbindung von Kunden und Lieferanten kann Vorteile in der Planung und Kundenzufriedenheit zur Folge haben.

Ausblick Eine Prognose ist durch die ohnehin schnelle Entwicklung schwer abzugeben, doch im Bereich der Softwaretests in Bezug auf verteilte Systeme gibt es noch jede Menge an Potential. Dienste wie Echtzeitüberwachung und dynamische Anpassung an die Umgebungsbedingungen sowie sicherheitsverbessernde Services werden die nächste Generation von Middleware prägen. Gerade im Bereich der mobilen Services besteht enormer Bedarf an large-scaled Middlewarelösungen, wobei die Leistungsfähigkeit keiner Unterscheidung zwischen fixen und mobilen Clients mehr unterliegen wird. Besondere Beachtung gilt auch der Entwicklung von Open-Source Lösungen die vor allem bei kleineren und mittleren Unternehmen steigende Verbreitung verzeichnen werden.
Im Bereich von Standardisierung und Kompatibilität der verschiedenen Middlewareprodukte wird für den Konsumenten eine positive Entwicklung in naher Zukunft stattfinden.

Literatur

[AB02] Alan Blatecky, Ann West, Mary Spada: Middleware the new Frontier. In: *Educause* (2002), 24–. http://www.educause.edu/ir/library/pdf/erm0241.pdf

[Abr03] Pekka Abrahamsson: Extreme Programming: First Results from a Controlled Case Study. In: *29th Euromicro Conference (EUROMICRO'03)*, IEEE, 2003, p. 259

[Apa05] Apache: *Apache Webserver*. www.apache.org. Version: 2005

[Arj06] Arjuna: *Arjuna Messaging Service*. http://www.arjuna.com/products/arjunams/index.html. Version: 2006

[ASS+99] Marcos Kawazoe Aguilera and Robert E. Strom and Daniel C. Sturman and Mark Astley and Tushar Deepak: Matching Events in a Content-Based Subscription System. In: *Symposium on Principles of Distributed Computing*, 53-61

[CABB04] M. Cilia and M. Antollini and C. Bornhövd and A. Buchmann: Dealing with Heterogeneous Data in Pub/Sub Systems: The Concept-Based Approach. In: *International Workshop on Distributed Event-Based Systems (DEBS'04)*. Edinburgh, Scotland, mai 2004

[Com05a] Zope Community: *Instance and Type based subscriptions*. http://www.zope.org/Wikis/DevSite/Projects/ComponentArchitecture/InstanceAndTypeBasedSubscriptions. Version: 2005

[Com05b] FOLDOC - Free Online Dictionary Of Computing: *Middleware*. http://foldoc.org/. Version: 2005

[Con04] W3C - World Wide Web Consortium: *Web Services Architecture*. http://www.w3.org/TR/ws-arch/. Version: feb 2004

[Con05] W3C - World Wide Web Consortium: *SOAP*. http://www.w3.org/TR/soap12-part1/. Version: 2005

[DH01] Werner Damm and David Harel: LSCs: Breathing Life into Message Sequence Charts. In: *Formal Methods in System Design* 19 (2001), Nr. 1, p. 45–80

[EFGK03] P.Th. Eugster and P.A. Felber and R. Guerraoui and A.-M. Kermarrec: The many faces of publish/subscribe. In: *ACM Comput. Surv.* 35 (2003), Nr. 2, p. 114–131

[FZP+04] Ludger Fiege and Andreas Zeidler and Alejandro P.Buchmann and Roger Kilian-Kehr and Gero Mühl: Security Aspects in Publish/Subscribe Systems. In: *Third Intl. Workshop on Distributed Event-based Systems (DEBS'04)*. EE The Institution of Electrical Engineers

[GDW04] Giovanni Denaro, Andrea Polini and Emmerich Wolfgang: Early Performance Testing of Distributed Software Applications. In: *Proceedings of the 4th international workshop on Software and performance*, ACM Press, 2004, p. 94 – 103

[Gro] OMG - Object Management Group: *CORBA - Specification*. http://www.omg.org/technology/documents/spec_catalog.htm

[Gro04] OMG - Object Management Group: *CORBA Notification Service*. http://www.omg.org/technology/documents/formal/notification_service.htm. Version: 2004

[Gro05] OMG - Object Management Group: *UML Specification*. http://www.uml.org/. Version: 2005

[HW03] Gregor Hohpe and Bobby Woolf: *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2003

[IB05] Lexitron - Fachlexikon der IT-Begriffe: *Middleware*. http://www.lexitron.de. Version: 2005

[IBM04] IBM: *Web Services Reliable Messaging*. mar 2004

[IBM05] IBM: *WebSphereMQ*. http://www-306.ibm.com/software/de/websphere/webserver/mqseries.html. Version: 2005

[(IE98] The Internet Engineering Task Force (IETF): *Uniform Resource Identifiers (URI) - RFC2396*. http://www.ietf.org/rfc/rfc2396.txt?number=2396. Version: aug 1998

[JB02] J. Bohn, W. Damm, J. Klose, A. Moik, H. Wittke: Modeling and Validating Train System Applications Using Statemate and Live Sequence Charts. In: ERTAS, H. Ehrig; B. J. Krämer; A. (Hrsg.): *Proceedings of the Conference on Integrated Design and Process Technology (IDPT2002)*, Society for Design and Process Science

[JOR06] JORAM: *JORAM: Java (TM) Open Reliable Asynchronous Messaging*. http://joram.objectweb.org/. Version: 2006

[Kho02] Ahmed Khoumsi: A Temporal Approach for Testing Distributed Systems. In: *IEEE Trans. Softw. Eng.* 28 (2002), Nr. 11, p. 1085–1103

[Kop97] Hermann Kopetz: *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Norwell, MA, USA : Kluwer Academic Publishers, 1997

[KP03] M. Kaddour and L. Pautet: Towards an Adaptable Message Oriented Middleware for Mobile Environments. In: *Proceedings of the IEEE 3rd workshop on Applications and Services in Wireless Networks*. Berne, Switzerland, jul 2003

[KS95] Narayanan Krishnakumar and Amit P. Sheth: Managing Heterogeneous Multi-system Tasks to Support Enterprise-Wide Operations. In: *Distributed and Parallel Databases* 3 (1995), Nr. 2, p. 155–186

[LAH05] Lachlan Aldred, Wil M.P. van der Aalst, Marlon Dumas, and Arthur H.M. ter Hofstede: On the Notion of Coupling in Communication Middleware. In: MEERSMAN, Robert (Hrsg.) ; TARI, Zahir (Hrsg.): *Proceedings International Conference on Distributed Objects and Applications (DOA) 2005*, 2005, p. 1015–1033

[Man06] MantaRay: *MantaRay*. http://www.mantamq.org/. Version: 2006

[MCE02] Cecilia Mascolo and Licia Capra and Wolfgang Emmerich: Mobile computing middleware. In: *Advanced lectures on networking*. New York, NY, USA : Springer-Verlag New York, Inc., 2002, p. 20–58

[Mic05a] Microsoft: *MSMQ*. http://www.microsoft.com/windows2000/de/server/help/default.asp?url=/windows2000/de/server/help/msmq. Version: 2005

[Mic05b] Microsoft: *.NET Framework*. http://msdn.microsoft.com/netframework/. Version: 2005

[Mic05c] Sun Microsystems: *Java 1.5*. http://java.sun.com/j2se/1.5/. Version: 2005

[Mic05d] Sun Microsystems: *JMS - Java Message Service*. http://java.sun.com/products/jms/. Version: 2005

[Mid05] Middleware.org: *Middleware Definition*. http://www.middleware.org/. Version: 2005

[Mug03] Rick Mugridge: Test Driven Development and the Scientific Method. In: *Agile Development Conference (ADC '03)*, IEEE, 2003, p. 43

[Mye79] Glenford J. Myers: *The Art of Software Testing*. 1. John Wiley and Sons, 1979

[Ols05] Dmitry Olshansky: *JPF - Java Plugin Framework*. http://jpf.sourceforge.net/. Version: 2005

[Ora06] Oracle: *Oracle Advanced Queuing*. http://www.oracle.com. Version: 2006

[Pep04] Ingo Peper: Enterprise Application Integration. (2004). http://www.ibr.cs.tu-bs.de/courses/ss04/svs/ESB.pdf

[Pom] Riccardo Pompeo: *MQ4CPP*. http://www.sixtyfourbit.org/mq4cpp.htm

[Pow96] David Powell: Group communication. In: *Communications of the ACM* 39 (1996), Nr. 4, p. 50–53

[Rau96] Andreas Rausch: *Verteilte Anwendungen:Grundlagen, Stand der Technologie und Architekturvarianten*, TECHNISCHE UNIVERSITÄT MÜNCHEN, Diplomarbeit, 1996

[Rät04] Manfred Rätzmann: *Software-Testing und Internationalisierung*. 2. Galileo Computing, 2004

[SD04] Schahram Dustdar, Stephan Haslinger: Testing of Service Oriented Architectures - A practical approach / Distributed Systems Group, Vienna University of Technology. Version: jul 2004. `http://www.infosys.tuwien.ac.at/Staff/sd/papers/TestingOfServiceOrientedArchitectures%96APracticalAp`. – Forschungsbericht. – Online–Ressource

[SH05] Steffen Heinzl, Markus Mathes: *Middleware in Java*. 1. vieweg, 2005

[Sys05] BEA Systems: *MessageQ*. `http://de.bea.com/produkte/messageq.jsp`. Version: 2005

[TAB03] Toni A. Bishop, Ramesh K. Karne: A Survey of Middleware. In: *Computers and Their Applications*, 2003, p. 254–258

[UNI98] ITU - INTERNATIONAL TELECOMMUNICATION UNION: Annex B: Formal semantics of Message Sequence Charts. (1998), apr, Nr. Z.120

[Vit03] Padmal Vitharana: Risks and challenges of component-based software development. In: *Communications of the ACM (CACM)* 46 (2003), Nr. 8, p. 67–72

[Wie03] Quality Software Engineering - TU Wien: *Fortgeschrittene Aspekte des Qualitätsmanagements*. `http://qse.ifs.tuwien.ac.at/courses/F_A_QM/188151_VO.htm`. Version: 2003

[Wik] Wikipedia: *Web Services*. `http://de.wikipedia.org/wiki/Web_Services`

[Wik05a] Wikipedia: *Authentifizierung*. `http://de.wikipedia.org/wiki/Authentifizierung`. Version: 2005

[Wik05b] Wikipedia: *DDoS*. http://de.wikipedia.org/wiki/Ddos. Version: 2005

[Wik05c] Wikipedia: *Middleware*. http://en.wikipedia.org/wiki/Middleware. Version: 2005

[Wik05d] Wikipedia: *Ontologie - Informatik*. http://de.wikipedia.org/wiki/Ontologie_%28Informatik%29. Version: 2005

[Wik05e] Wikipedia: *OSI-Modell*. http://de.wikipedia.org/wiki/OSI-Modell. Version: 2005

[Wik05f] Wikipedia: *Performance*. http://de.wikipedia.org/wiki/Performance_%28Informatik%29. Version: 2005

[Wik05g] Wikipedia: *Plugin*. http://de.wikipedia.org/wiki/Plugin. Version: 2005

[Wik05h] Wikipedia: *Public-Key-Infrastruktur*. http://de.wikipedia.org/wiki/Pki. Version: 2005

[Wik05i] Wikipedia: *SOA - Service Oriented Architecture*. http://de.wikipedia.org/wiki/Service_Oriented_Architecture. Version: 2005

[Wik06] Wikipedia: *Transaktion*. http://de.wikipedia.org/wiki/Transaktion_%28Informatik%29. Version: 2006

[XML06] XMLBlaster: *XMLBlaster*. http://www.xmlblaster.org/. Version: 2006

[Zei04] Andreas Zeidler: *A Distributed Publish/Subscribe Notification Service for Pervasive Environments*. Mülheim an der Ruhr, Technischen Universität Darmstadt, Diplomarbeit, sep 2004

Abbildungsverzeichnis

1	Schichtenbetrachtung von Middleware	2
2	Kategorisierung von Middleware	9
3	gekoppelte Übertragung im Bezug auf Raum	21
4	ungekoppelte Übertragung im Bezug auf Raum	22
5	gekoppelte Übertragung im Bezug auf Zeit	23
6	ungekoppelte Übertragung im Bezug auf Zeit	24
7	gekoppelte Übertragung im Bezug auf Synchronisation	25
8	ungekoppelte Übertragung im Bezug auf Synchronistaion	26
9	hierarchiesche Topicstruktur	40
10	Ablauf Notifizierung	44
11	Einfaches MSC	47
12	Inkonsistentes MSC	48
13	Instanzerzeugung und Instanzende	50
14	Zeitliche Elemente	51
15	Bedingungen	52
16	LSC - Lokale Invariante	55
17	Grafische Oberfläche	60
18	Topic	63
19	Topic	65
20	Schichtendarstellung	66
21	Plugin-Extensionpoints	67
22	Plugin-Manifest	69
23	Testablauf	80
24	Nachrichtenreihenfolge	81
25	Anforderungen	85
26	Teststruktur	88
27	Konfiguration Test-Run	88
28	Konfiguration Test-Run	92
29	One-to-One Szenario	92
30	One-to-Many Szenario	93
31	One-to-One Szenario	94
32	Theoretische Nachrichtenentwicklung	95
33	Tatsächliche Nachrichtenentwicklung	95
34	RPC-Exception	97

Tabellenverzeichnis

1 Elemente von Publish/Subscribe 37
2 Event Schema 40
3 LSC Temperaturen 54
4 Einflussparameter 87
5 Logeinträge 91
6 Performance Ergebnisse 96