Fachbereich Angewandte Informatik
an der Fachhochschule Bonn Rhein Sieg
Seminar: Verteilte und Paralle Systeme I

Seminarhausarbeit

Java Message Service

10. Juni 2002

Themensteller:
Prof. Dr. Rudolf Berrendorf

Author:
Matthias Lübken
matthias@luebken.com

Inhaltsverzeichnis

1 Einleitung **3**

2 J2EE **4**

2.1 Services 4

2.2 Architektur 5

2.3 Enterprise JavaBeans 6

3 JMS Modell **8**

3.1 Elemente und Begriffe 8

3.2 Kommunikationmodelle 9

4 JMS Nachricht **10**

4.1 Header / Kopf 10

4.2 Properties / Eigenschaften 11

4.3 Body / Nutzdaten 12

4.4 Nachrichten-Selektoren 13

5 Publish-and-Subscribe Messaging **14**

6 Point-to-Point Messaging **15**

7 Zuverlässigkeit **16**

7.1 Bestätigungen 16

7.2 Transaktionen 17

8 Message Driven Bean **20**

9 Beispiel: MP3s ankündigen **21**

9.1 MP3Server.java 21

9.2 MP3Client.java 25

A Definitionen: **27**

1 Einleitung

Der Java Message Service (JMS) definiert eine Nachrichten-orientierte Kommunikation zwischen verteilten Systemen. Man kann das Java Messaging auch als das Pendant zum Email System sehen, wobei beim JMS Applikationen miteinander kommunizieren und bei EMail in der Regel Personen.

Eine Java Komponente die den JMS nutzt, wird als ein Message Client bezeichnet. Dieser kann mittels JMS, Nachrichten zu anderen Clients senden und von diesen wiederum Nachrichten empfangen. Die wohl wichtigste Eigenschaft dabei ist, dass die Kommunikation zwischen den verschiedenen Applikationen lose ist. Das heißt die beiden Kommunikationspartner müssen nicht gleichzeitig erreichbar sein, noch müssen sie präzise Informationen von einander haben, im Gegensatz zu zum Beispiel Remote Method Invocation (RMI). Ein Client sendet einfach eine Nachricht von einem vorher vereinbarten Typ und JMS übernimmt den Rest. Dabei kann die Kommunikation (wie auch bei EMail), völlig asynchron ablaufen. Zusätzlich dazu biete JMS die Möglichkeit die Nachrichten dennoch zuverlässig zu senden. Mehr dazu später.

Folgende Vorteile bei dem Einsatz von JMS ergeben sich für einen Software Anbieter und dessen Kunden:

- Applikationen können unabhängig von einander laufen, das heißt wenn die eine Seite aus irgendeinem gewollten oder nicht gewollten Grund nicht erreichbar ist, muss dies nicht zum Ausfall des ganzen Systems führen.
- Damit einher geht der Vorteil das Applikationen nach einer Nachricht einfach weiter arbeiten können und sich darauf verlassen können, dass die Nachricht ankommt.
- Da Sun das JMS Interface vorgegeben hat, welches von verschiedenen Anbieter implementiert wird, muss man sich nicht auf eine konkrete Implementation verlassen. Sondern man hält sich die Möglichkeit offen, Komponenten in einer einfachen Art und Weise auszutauschen.

Sun beschreibt JMS als Teil der Java 2 Enterprise Edition (J2EE). Daher möchte ich zunächst eine kurze Einführung in den Gedanken hinter J2EE und den einzelnen Komponenten geben insbesondere Enterprise JavaBeans, da der Message Driven Bean eng mit JMS verknüpft ist. Anschließend werde ich mich dann näher mit dem eigentlichen Thema dem Java Message Service beschäftigen.

2 J2EE

Bei der Java Enterprise Edition handelt es sich um eine Zusammenfassung verschiedener Java Technologien. Mit dieser Zusammenfassung, der eingehenden Standardisierung und den verschiedenen Komponenten-Technologien wird die Herstellung so genannter Enterprise Applications deutlich vereinfacht. So stellt jede J2EE Plattform verschiedene Services, meist über so genannte Connectors / Adapters den einzelnen Komponenten einer Applikation zur Verfügung und reduziert somit viele Details, um die sich sonst der Anwendungsentwickler hätte kümmern müssen. Sicherlich hätte man dies durch einfache Komponenten erreicht, aber so stellt Sun zusammen mit anderen Firmen einen umfassen Standard zur Verfügung von denen die meisten Beteiligten profitieren. So können zum Beispiel die Anwendungsentwickler auf einen bestehendes Angebot von Services aufbauen und vertrauen. Und die Käufer von J2EE Applikationen können verschiedene Komponenten der Anwendung, als auch Teile der Plattform austauschen. So könnte eine Firma den Typ der Datenbank ändern oder die komplette J2EE-Platform, ohne dass die eigentliche Anwendung verändert werden müsste. Das verspricht zumindest Sun. Ob dies in einem realen Projekt so umgesetzt werden kann, mag ich nicht beurteilen.

Der Name Enterprise in J2EE ist ein wenig irreführend und gibt keinen präzisen Zusammenhang zu den vorhandenen Technologien wieder. Zwar ist der Business Bereich der wichtigste für die J2EE Plattform und sind einige Ausrichtungen klar für die Anforderungen im Enterprise Bereich ausgelegt, dennoch sind die verschiedenen Technologien auch für andere Bereiche vielseitig einsetzbar. Auch wenn der Begriff die Sachlage nicht völlig befriedigend beschreibt, könnte man auch von einer “distributed“ Edition sprechen, da sich die meisten Komponenten mit verteilten Systemen beschäftigen.

J2EE hat unter anderem einen so großen Erfolg, da für die verschiedensten bestehenden Legacy-Systeme Java Schnittstellen erstellt wurden. So können die verschiedenen Datenbanken weiter benutzt werden oder um im Bereich JMS zu bleiben, bewährte Message Oriented Middleware (MOM) Systeme können mit einer JMS Schnittstelle versehen in J2EE Plattformen eingebunden werden.

2.1 Services

Den Begriff J2EE gibt es erst seit Java 2, sprich seit der Version 1.2 . Dabei wurden verschiedene APIs und Technologien wie oben erwähnt zusammengefasst, die ich hier kurz vorstellen möchte.

- Einer dieser Systeme ist das Java Database Connectivity (JDBC), dass als erstes Popularität erlangte und mit dessen Hilfe man per Java auf vorhandene Datenbank Systeme zugreifen kann.

- Ein weiteres System ist das Java Naming and Directory Interface (JNDI) mit dessen Hilfe man verschiedene Lookup-Dienste nutzen kann, um notwendige Objekte aufzufinden.
- Die Java Transaktion API, ein Standard für verteilte Transaktionen.
- JavaMail zum Versenden von EMails aus Java Anwendungen.
- Java API for XML Processing (JAXP) zum Lesen und Schreiben von XML-Dokumenten.
- Enterprise JavaBeans (EJB) eine Komponenten Architektur.
- JSP / Servlets Anwendungen die auf einem Webserver laufen und durch verschiedene Hilfsmittel Java als Ersatz für andere Webserver-Sprachen anbieten.
- Remote Method Invocation (RMI) zum Ausführen von Methoden auf fremden Objekten.
- Corba compliance. Mit den zwei Technologien JavaIDL und RMI-IIOP, kann man Java Anwendungen mit Corba Systemen integrieren.
- Und natürlich der Java Message Service (JMS). J2EE Version 1.2 setzt voraus das kompatible Applikationsserver lediglich das Interface von Sun zur Verfgung stellen. Ab der J2EE Version 1.3 ist es Voraussetzung einen JMS Provider zu implementieren.

2.2 Architektur

Die J2EE Architektur besteht laut Sun aus vier Ebenen / tiers:

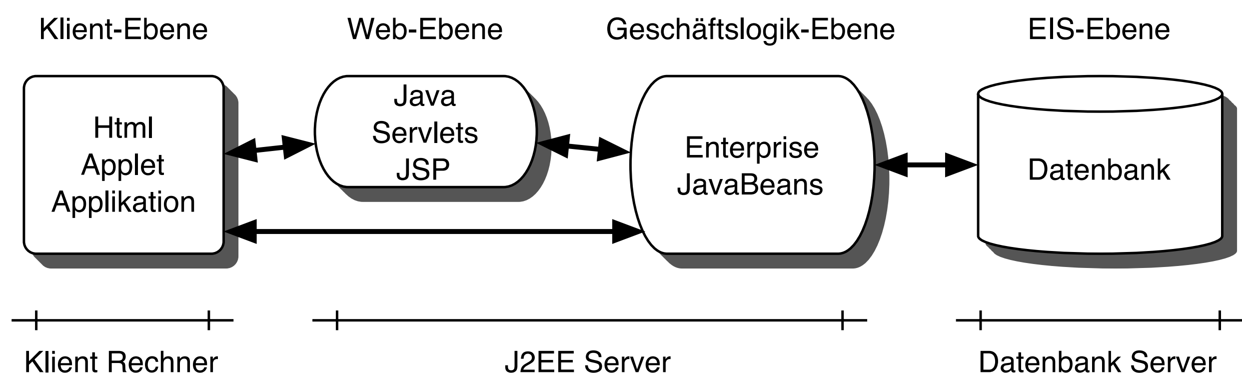

Abbildung 1: J2EE überblick

- Klienten Ebene / client tier:
  Auf dieser Ebene gibt es verschiedene Arten von Clients die mit dem Nutzer interagieren. Das reicht von Html-basierten Browser, über Applets bis hin zu vollwertigen Java Applikationen. Hierbei sei erwähnt, dass durch die verwendeten Clients die verschiedenartigsten Devices in Frage kommen. Von einfachen PCs (natürlich Betriebssystem übergreifend) bis hin zu kleinen Devices wie Handys oder PDAs.

- Web Ebene / web tier:
  Diese Schicht beliefert die Klienten Ebene mit verarbeiteten Daten aus der Geschäftslogic. Dabei wird das Komponenten Modell der Servlets und Java Server Pages verwendet.

- Geschäfts-Logik Ebene / business tier:
  Hier liegt die eigentliche Logik einer Anwendung und verbindet die Präsentation mit den Daten. Hier kommt nun ein weiteres Komponenten Modell zum Zuge, die Enterprise Java Beans welche unten erläutert werden.

- Enterprise Information System Ebene:
  Nach Sun liegen hier die Datenbanken und andere fremde Systeme.

Dabei kann man diese Architektur auch als eine Modell-View-Controller Architektur betrachten, da die View (Klienten- und Web-Ebene), der Controller (Geschäftslogik-Ebene) und das Modell (die Enterprise Information System-Ebene) von einander getrennt sind.

2.3 Enterprise JavaBeans

Die wohl wichtigste Java Komponenten Technologie sind die Java Beans oder die Enterprise Java Beans (EJB). Und da Sun einen neuen Typ, den Message Driven Bean eingeführt hat, auf den ich später eingehe, möchte ich hier kurz allgemein die Java Enterprise Beans vorstellen.

EJBs sind server-seitige Komponenten, welche die Geschäfts-Logik implementieren. Also der Teil der Applikation, der die eigentliche Aufgabe erfüllt.

Enterprise Beans laufen in einem so genannten EJB Container. Von diesem Container werden verschiedene Services zur Verfügung gestellt, die ein Bean direkt oder indirekt nutzen kann. So werden zum Beispiel peristente Beans automatisch vom Container in einer Datenbank gespeichert, ohne dass der Anwendungsentwickler einen SQL Befehl ausführen muss, so kann sich dieser um die eigentliche Anwendung kümmern.

Es existieren im Moment drei Typen von Enterprise Beans:

- *SessionBeans:*
  Die einen bestimmten Task für einen Klienten erfüllen.
- *EntityBeans:*
  Abbildung von Geschäftsobjekten die dauerhaft gespeichert werden
- *MessageDriven Beans:*
  Ein asynchroner Nachrichtenempfänger für die JMS API

Klienten greifen auf einen SessionBean oder EntityBean über ein Interface zu. Bei den Message Driven Beans existiert ein anderer Mechanismus (siehe Kapitel 8 auf Seite 20). Mit dem Interface ist also die Sicht eines Klienten (andere einer anderen Komponente) auf den Bean definiert und bietet somit eine Kapselung von den Implementierungsdetails, die für den Klienten nicht einsehbar sind.

Ein EJB kann lokalen oder entfernten Zugriff erlauben und je nach dem müssen unterschiedliche Interfaces definiert werden. Bei einem Bean mit remote Zugriff gibt es zum einem das *home interface* welches den Lebenszyklus eines Beans definiert, also Methoden wie create oder remove. Und zum anderen das *remote interface* welches die zur Verfügung gestellten Geschäftsmethoden definiert, also zum Beispiel getUserInfo() bei einem Informationssystem. Dabei ist der Ort des EJB für den entfernten Client transparent. Bei einem Bean mit lokalem Zugriff gibt es zum einem das *local home*, welches den Lebenszyklus eines Beans definiert. Und zum anderen das *local interface*, welches die zur Verfügung gestellten Geschäftsmethoden definiert. Bei einem Bean der beide Zugriffswege akzeptiert müssen auch die remote und local interfaces implementiert werden, obwohl dies unüblich ist.

Die folgende Tabelle soll einen Überblick über verschiedenen Teile eines EJB und deren Namenskonvention geben:

	Syntax	Beispiel
Enterprise bean name	`<name>EJB`	KontoEJB
EJB JAR Name	`<name>JAR`	KontoJAR
Enterprise bean class	`<name>Bean`	KontoBean
Home interface	`<name>Home`	KontoHome
Remote interface	`<name>`	Konto
Local home interface	`Local<name>Home`	LocalKontoHome
Local interface	`Local<name>`	LocalKonto
Abstract Shema	`<name>`	Konto

Abbildung 2: Überblick über verschiedene EJB Teile

Für eine tiefer gehende Einführung verweise ich auf das J2EE Tutorial [1] und die einschlägige Literatur.

3 JMS Modell

3.1 Elemente und Begriffe

Eine JMS Applikation ist aus folgenden Elementen zusammen gesetzt:
JMS clients, Non-JMS clients, Messages, JMS providers und *Administered objects*,
die ich im folgenden vorstellen werde.

Das von Sun ausgegebene Java Message Service API definiert zunächst einmal eine reine Programmierschnittstelle. Das heißt es gibt verschiedene Hersteller die eine Implementation zu den vorhanden Interfaces anbieten. Derjenige der eine Implementation zur Verfügung stellt nennt man einen *JMS Provider*. Oft handelt es sich hierbei um so genannte Message Oriented Middleware (MOM), die schon seit einiger Zeit auf dem Markt existieren und nur eine Schnittstelle über JMS hergestellt haben. Bei MOM und damit auch bei JMS werden Daten in Form von Nachrichten (bzw. Messages) in verteilten Systemen ausgetauscht.

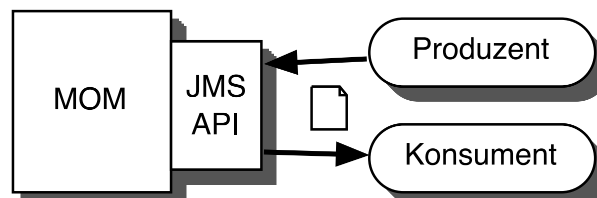

Abbildung 3: Clients in verschiedenen Rollen kommunizieren mit dem JMS Provider (hier eine MOM) über die JMS API

Der JMS Provider ist aus Sicht eines Benutzer eine Blackbox die mittels der JMS API Nachrichten versendet. Er stellt den eigentlichen Dienst des Sendens von Nachrichten zur Verfügung. Die Benutzer dieser Dienste werden als *JMS Clients* bezeichnet. In einer Applikation nimmt ein JMS Client eine von verschiedenen Rollen ein. Ein JMS Client nimmt im allgemeinen zunächst einmal die Rolle des Produzenten (producer) und / oder des Konsumenten (consumer) ein. Ein Produzent erzeugt eine Nachricht und gibt diese weiter an den JMS Provider zum Versenden. Der Konsument empfängt die Nachricht vom Provider und verarbeitet sie weiter. Hier sieht man ganz deutlich dass die verschiedenen Clients von einander, sowie von dem MOM System getrennt sind und nicht direkt mit einander kommunizieren. Dadurch können auch andere, nicht JMS Clients eingebunden werden oder die MOM Systeme ausgetauscht werden.

Beim Aufbau einer Kommunikation sind noch weitere wichtige Komponenten zu nennen. Das sind als erstes die *Ziele (destinations)* an die sich eine bestimmte Kommunikation richtet. Je nach Modell kann dies eine Topic oder Queue sein. Des weiteren sind dies die *Verbindungen (connections)*, die eine logische Verbindung zwischen Client und Provider aufbauen. Vor jeder Kommunikation muss eine Connection aufgebaut werden, die nach der Kommunikation wieder geschlossen werden muss. Auf diesen Connections setzen dann die Sessions

auf. Bei diesen *Sessions* handelt es sich um einen single-threaded Kontext um Nachrichten zu senden oder zu empfangen. Man kann eine bis mehrere Sessions pro Connection erzeugen. Eine Session kann als atomare Einheit bei Transaktionen wird zurück gesetzt werden. Zwar kann man auch mehrere Connections erzeugen, doch diese bedeuten einen hheren Ressourcen Verbrauch. Zum Beispiel wird ein neuer Port benutzt.
So genannte *Administered Objects* werden vom Provider verwaltet und von einem Administrator erstellt und konfiguriert. über JNDI greifen Clients auf diese Objekte zu. So gibt es die *Connection Factory* Über die ein Client sich die eigentliche *Connection* holt und die oben genannten *Destinations*.

Asynchron / Synchron

Generell kann der Empfänger kann entscheiden, wie er eine Nachricht empfangen möchte. Er kann dies synchron tun, in dem er die receive() Methode aufruft. Diese blockiert dann den Programmablauf bis eine Nachricht empfangen wurde. Oder er implementiert das Message-Listener Interface und damit die onMessage() Methode, die asynchron aufgerufen wird falls eine Nachricht eintrift.

3.2 Kommunikationmodelle

JMS bietet zwei Kommunikationsmodelle, die schon länger in alten Message Orietented Middleware Systemen existierten. Diese sind *publish-and-subscribe* und *point-to-point*, die allgemein auch als *messaging domains* bezeichnet werden.

Innerhalb dieser Domains gibt es für die oben erwähnten allgemeinen Rollen spezielle Rollen. Bei *point-to-point* wird aus dem Produzent der Sender und dem Konsument der Receiver. Analog wird bei *publish-and-subscribe* aus dem Produzent der Publisher und aus dem Konsument ein Subscriber.

Bei dem point-to-point Modell wird eine Nachricht an eine Queue geschickt. Jede Nachricht hat genau einen Empfänger (receiver). Der Empfänger bestätigt den Erhalt der Nachricht. Es gibt keine Zeit-Abhängigkeiten zwischen Sender und Empfänger. Die Queue speichert persistente Nachrichten ab, bis sie vom Empfänger gelesen werden oder sie je nach Typ der Nachricht nach einem Zeit-Limit ablaufen.
Mehr zu Point-to-point in Kapitel 6 auf Seite 15.

Bei dem publish-and-subscribe Modell schickt der Publisher die Nachricht an eine so genanntes Topic (Thema). Zu diesem Topic können sich dann mehrere Subscriber anmelden. Auch gibt es keine Zeit-Abhängigkeiten zwischen Sender und Empfänger und der Server kann persistente Nachrichten abspeichern.
Mehr zu Publish-and-subsribe in Kapitel 5 auf Seite 14.

4 JMS Nachricht

Eine JMS Nachricht hat ein recht simplen, aber dennoch sehr flexiblen Aufbau, welches auch einen Aufbau in heterogenen Message Netzwerken erlaubt. Eine JMS Nachricht besteht aus drei Teilen:

Headers	Properties	Data
JMSDestination, JMSDeliveryMode, JMSMessageID, JMSTimestamp, JMSExpiration, JMSRedelivered, JMSPriority, JMSReplyTo, JMSCorrelationID, JMSType	*(Application & Provider defined)*	*Message, TextMessage, StreamMessage, MapMessage, ObjectMessage, BytesMessage*

Abbildung 4: Eine JMS Nachricht

4.1 Header / Kopf

Der Header beinhaltet die Standardeigenschaften die von Clients und Provider genutzt werden, um die Nachricht zu identifizieren und zu Routen.

- *JMSMessageID:(String)*
  Eine ID die eine Nachricht eindeutig identifiziert und welche vom Provider gesetzt wird.
- *JMSDestination:(Destination)*
  Das Ziel der Nachricht. Vom Provider gesetzt.
- *JMSDeliveryMode:(int)*
  Der DeliveryMode kann entweder persistent oder nicht persistent sein. Persistent bedeutet die Nachricht wird extra zwischen gespeichert, bevor sie weiter gesendet werden. Damit sind die Nachrichten vor vielen Fehlern wie auch einem System-Absturz gesichert. Bei non-persistent werden die Nachricht nicht gespeichert und sind bei einem Ausfall des Providers verloren.
- *JMSTimeStamp:(long)*
  Hier wird die Zeit eingetragen, zu der Provider die Nachricht zum versenden bekam.
- *JMSExpiration:(long)*
  Die Zeit wann eine Nachricht auslaufen soll. Dabei zeigt ein 0 an, dass die Nachricht nie auslaufen soll.

- *JMSPriority:(int)*
  Die Priorität einer Nachricht gesetzt vom Provider und Client beträgt zwischen 0 (niedrigste) und 9 (höchste Priorität). Der Provider versucht Nachrichten mit höherer Priorität zu bevorzugen, garantiert dies aber nicht.

- *JMSCorrelationID:(String)*
  Wird dazu genutzt um zwei Nachrichten in Verbindung zu bringen. Typischerweise setzt der antwortende Client die Correlation ID mit der Message ID, auf die er antwortet. So kann der Sender der ersten Nachricht die beiden Sendungen in Verbindung bringen.

- *JMSReplyTo:(Destination)*
  Der Ort zu dem eine Antwort auf die aktuelle Nachricht geschickt werden soll.

- *JMSType:(String)*
  Hier wird der Typ der Nachricht festgelegt. Dies kann vom Client oder vom Provider gesetzt werden. Mehr dazu in Kapitel 4.3

- *JMSRedelivered:(boolean)*
  Zeigt an dass die Nachricht bereits gesendet wurde.

4.2 Properties / Eigenschaften

Hier werden Applikations und Provider spezifische Eigenschaften gespeichert. Falls ein Provider einen zusätzliche Information zu denen in dem Header einfügen will kann er dies hier tun. Dabei gibt JMS einige optionale Eigenschaften vor:

- JMSXUserID – type string
  ID des Nutzers, der die Nachricht sendet

- JMSXApplID – type string
  ID der Applikation, die die Nachricht sendet

- JMSXDeliveryCount – type int
  Zhlt die Sendeversuche mit

- JMSXGroupID – type string
  Gruppen ID zu der die Nachricht gehört

- JMSXGroupSeq – type int
  Sequenznummer der Nachricht innerhalb einer Gruppe

- JMSXProducerTXID – type string
  Identifiziert die Transaktion innerhalb der die Nachricht produziert wurde

- JMSXConsumerTXID – type string
  Identifiziert die Transaktion innerhalb der die Nachricht konsumiert wurde
- JMSXRcvTimestamp – type long
  Timestamp beim Empfnger
- JMSXState – type int
  Status fr spezielle Provider
- JMSX_vendor_name
  Hier knnen Hersteller spezifische Eigenschaften gespeichert werden

4.3 Body / Nutzdaten

In dem Body liegen die eigentlichen Daten die gesendet werden. Dabei gibt es unterschiedliche Typen von Nutzdaten die in den sechs verschiedenen Interfaces von JMS vorgegeben werden, und von den Providern implementiert werden müssen.

- `TextMessage`
  Ein einfacher `java.lang.String`, zum Beispiel ein XML String
- `MapMessage`
  Eine Map mit Schlüssel/Wert Paaren aus dem Collection Framework. Dabei sind die Schlüssel Strings und die Werte primitive Datentypen. Die Werte können dann über den Name ausgelesen werden, oder über einen `Enumerator`.
- `BytesMessage`
  Hier wird ein Feld primitiver Bytes gesendet. Die Interpretation dieser Daten müssen der Sender und der Empfänger übernehmen.
- `StreamMessage`
  Wie eine BytesMessage nur wird hier zusätzlich die Information des primitiven Datentyps gespeichert und forciert. So kann man bei einer BytesMessage zunächst ein Long Datum speichern und dann dann ein Short lesen, doch dies würde Exception bei StreamMessage auslösen.
- `ObjectMessage`
  Ein serialisierbares Java Objekt.
- `Message`
  Das Message Interface hat keine Nutzdaten. Hier werden nur die Headerdaten und keine Nutzdaten gesendet.

Weitere Informationen zu den verschiedenen Teilen einer Nachricht befinden sich in dem Quellcode der verschiedenen Interfaces.

4.4 Nachrichten-Selektoren

Mittels Nachrichten-Selektoren kann ein Empfänger für ihn relevante Informationen raus filtern. Dabei werden boolsche Filter über die Header und Properties Informationen gelegt. Die Syntax lehnt sich an den WHERE-Teil eines SQL Statements an. Zum Beispiel können so komplexe Selektoren wie Name IN ('Susi', 'Linda', 'Alex') AND (Alter * IQ) > 100 angegeben werden. Die Filter werden beim Erzeugen eines Empfängers eingeschaltet.
Bsp:

```
TopicSubscriber subs = session.(topic, " username == 'Linda' ", false);
// Unser Subscriber erhaelt nur Nachrichten,
// in der eine Eigenschaft username auf Linda gesetzt ist
```

5 Publish-and-Subscribe Messaging

Das Publish-and-subribe erlaubt das Senden von Nachrichten an mehrere Konsumenten in diesem Fall Subscriber. Dabei werden diese Nachrichten in einem push-basiertem Modell verschickt. Das heißt der Konsument muss sich nicht um das Ankommen der Nachrichten kümmern, sondern sie werden ihm automatisch zugestellt.

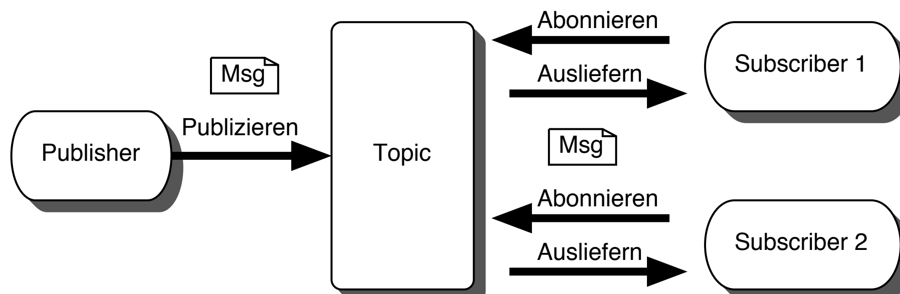

Abbildung 5: Publish and subscribe message domain

Vor Beginn der Kommunikation muss sich jeder Konsument zunächst an einem Thema (topic) anmelden. Der Produzent schickt die Nachrichten an das topic und der JMS Provider leitet dann alle Nachrichten an die angemeldeten Konsumenten weiter. Dabei werden die Nachrichten kopiert und einzeln weiter geschickt und mit Sendebestätigungen überprüft.

Es existiert keine enge Verknüpfung zwischen Publisher und Subscriber. Letztere können (und werden häufig) dynamisch hinzugefügt und entfernt werden.

Neben den einfachen Subscriptions können Empfänger auch Durable Subscriptions anmelden. Damit ist die Verteilung der Nachrichten sicher gestellt. Meldet sich ein Subscriber als Durable an und verpasst durch einen System Ausfall eine Reihe von Nachrichten, so bekommt er diese zugestellt so bald er sich wieder angemeldet hat.

6 Point-to-Point Messaging

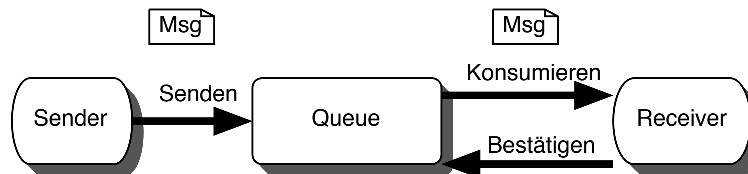

Abbildung 6: Point-to-point message domain

Das Point-to-point messaging wird über die so gennante Queue abgewickelt. Eine Nachricht eines Senders wird nur an einen Empfänger weitergeleitet. Sender und Empfänger sind auch hier nicht gekoppelt und so können diese wechseln und es können auch mehrere entstehen. Aber auch bei mehreren Empfängern wird die Nachricht nur einmal gesendet. Ein prominentes Beispiel für diesen Fall ist das Load-Balancing. Ein weiterer Aspekt bei Point-to-point ist, dass die Nachrichten sortiert sind. Denn Nachdem eine Nachricht gesendet wurde, wird sie zunächst aus der Queue gelöscht und erst anschließend wird die nächste gesendet.

Die beiden message domains sind mit gewissen Aufwand austauschbar, was vornehmlich an der historischen Entwicklung von JMS und den beteiligten MOM System liegt. Dabei sind die jeweiligen Modelle für das gedachte Einsatzgebiet einfacher zu nutzten.

7 Zuverlässigkeit

Um das Zustellen einer Nachricht sicher zustellen und die Sendung den Klienten zu garantieren, gibt es zwei grundsätzliche Möglichkeiten. Zum einem die Bestätigungen / Acknowledgements und zum anderen die darauf aufbauenden Transaktionen. Auch hier beschreibt JMS nur die logische Sicht, die eigentliche Implementation bleibt den Providern überlassen.

7.1 Bestätigungen

Allgemein werden Nachrichten wie folgt bestätigt:

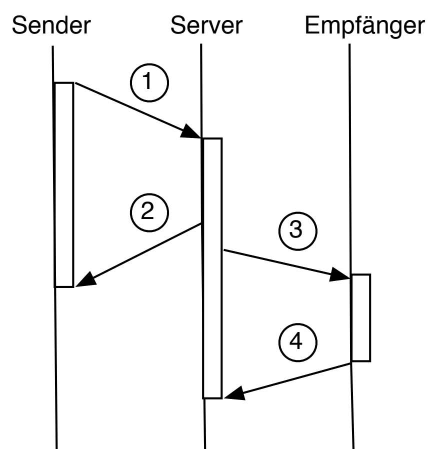

Abbildung 7: Message Acknowledgements

1. Der Sender schickt eine Nachricht an den JMS Server
2. Der Server bestätigt den Empfang der Nachricht
3. Der Server schickt die Nachricht an den Empfänger
4. Der Empfänger bestätigt den Empfang

Nachdem der Sender die Nachricht an den Server geschickt hat, bestätigt dieser den Empfang und akzeptiert damit die Verantwortlichkeit für das weitere Verschicken der Nachricht. Falls es sich um eine persistente Nachricht handelt, speichert der Server die Nachricht. Bei nicht persistetenten Nachrichten, versucht der Server einen Abnehmer zu finden und falls dies nicht erfolgreich ist verfällt die Nachricht. So knnte in dem Fall, dass sich bei einem Publish-and Subscribe, keiner für das Thema eingeschrieben hat die Nachricht entfallen.

Ein besonderer Vorteil von dem asynchronen Senden in JMS sieht man in der Abb. 7. Nachdem der Sender die Bestätigung des Servers erhalten hat, kann er sicher gehen dass die Nachricht ankommt und muss sich nicht weiter um diese Nachricht kümmern muss.

Es gibt drei Möglichkeiten der Bestätigungen. Diese werden bei der Erstellung einer neuen Session angegeben.

AUTO_ACKNOWLEDGE

Bei dem AUTO_ACKNOWLEDGE übernimmt der Provider das Versenden der Bestätigungen. Dabei wird die Session automatisch veranlassen eine Bestätigung zu verschicken, nachdem die Methode onMessage() erfolgreich abgearbeitet wurde. (Bei der onMessage() handelt es sich um die verantwortliche Methode auf der Empfänger Seite, die bei dem Eingang einer Nachricht aufgerufen wird.)

CLIENT_ACKNOWLEDGE

Möchte man den Zeitpunkt dieser Bestätigung anders wählen, zum Beispiel früher in der onMessage() Methode, um den Provider von seinen Pflichten zu entbinden, kann man dies mit CLIENT_ACKNOWLEDGE tun. Dazu muss man einfach die Methode message.acknowledge() aurufen. In allen anderen Fällen wird die Methode acknowledge() nicht genutzt.

DUPS_OK_ACKNOWLEDGE

Wenn man diese Option wählt, zeigt man an das auch Duplikate von Nachrichten gesendet werden dürfen. Damit erspart man dem Provider sicher zustellen, dass eine Nachricht genau einmal gesendet wurde und durch den geringeren Overhead kann man an Performance gewinnen. Auf der Empfänger Seite muss dann das Redelivered-Feld überprüft werden und gegebenenfalls die Nachricht an Hand der MessageID mit den bereits empfangenen Nachrichten verglichen werden.

7.2 Transaktionen

Bei Transaktionen kann man mehrere Operationen zu einer atomaren Operation zusammen führen. Dafür wird das Session Objekt benutzt. An diesem Objekt werden die folgenden Methoden aufgerufen: commit() um eine Transaktion erfolgreich abzuschließen oder rollBack() um alle Operationen rückgängig zumachen. Beim Erzeugen einer Session (createTopicSes-

sion(boolean, int)) gibt der erste Parameter an ob man Transaktionen nutzen möchte. Der zweite Parameter Acknowledge-Mode hat hier keine Bedeutung.

Auch bei Transaktionen werden im Allgemeinen Sende- und Empfang-Operationen von einander getrennt. So kann ein Sender mehrere Sende Operationen zu einer Transaktion zusammen führen.

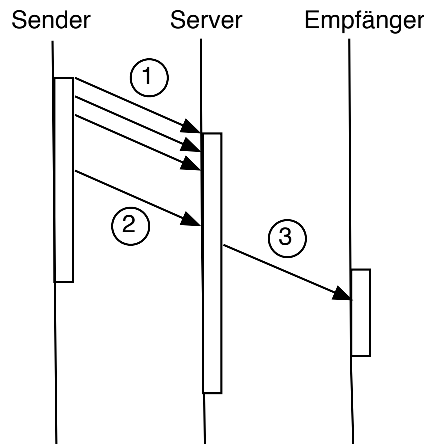

Abbildung 8: Unter (1) werden mehrere Nachrichten gesendet, die dann mit einem commit() unter (2) freigegeben werden. Erst dann wird der Server aktiv und verschickt die Nachrichten (3). Falls bei (2) ein rollBack() eingeleitet wurde, löscht der Server alle empfangenen Nachrichten.

Ausserdem können auch mehrere Empfänge zu einer Transaktion zusammen geführt werden.

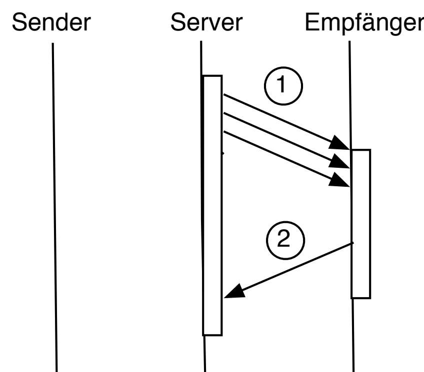

Abbildung 9: Unter (1) werden mehrere Nachrichten an den Empfänger gesendet oder aus Sicht des Empfängers empfangen. Dieser werden dann komplett mit einem commit() unter (2) bestätigt. Falls bei (2) ein rollBack() eingeleitet wurde, sendet der Server alle Nachrichten erneut.

Zu dem können Sende- und Empfang-Operationen zu einer Transaktion zusammen geführt werden. Damit kann ein Client Nachrichten empfangen und senden und diese zu einer Trans-

aktion zusammen führen. Falls dann ein Client einen Rollback ausführt, werden die gesendeten Nachrichten an den Server verworfen und die empfangenen Nachrichten erneut gesendet. Diese Transaktionen sollten aber genau untersucht werden, da diese zu einer Verzögerung oder ganz zu einem DeadLock führen können.

8 Message Driven Bean

Neben den alten Beans (SessionBeans und EntityBeans) gibt es einen neuen MessageDriven Bean, welcher erstmals den asynchronen Empfang von Nachrichten ermöglicht. Dazu muss dieser wie auch andere Anwendungen das MessageListenerInterface mit der Methode onMessage() implementieren.

Dennoch sind einige Dinge hier anders:

- Der EJB Container erzeugt automatisch einen Konsumenten für die Nachrichten, registriert automatisch den Bean als MessageListener und setzt den Acknowledgement Mode
- Neben dem MessageListenerInterface muss der Bean noch das javax.ejb.MessageDrivenBean Interface und die dazu gehörigen Methoden ejbCreate(), ejbRemove(), setMessageDrivenContext(...) implementieren.

Der größte Unterschied zu den anderen Beans ist, das kein `home` oder `remote interface` existiert, sondern nur die Bean Klasse selber.

Der Lebenszyklus eines Message Driven Beans ist recht simple. Es gibt zwei Zustände: Does not exist und Ready. Falls der Bean nicht existiert wird dieser instanziert und dann die Methode setMessageDrivenContext() und ejbCreate() aufgerufen. Danach wechselt der Bean und den Ready Zustand. Nun kann er Nachrichten empfangen und an ihm die onMessage() Methode aufgerufen werden. Am Ende wird dann die Methode ejbRemove() zum Entfernen des Beans aufgerufen.

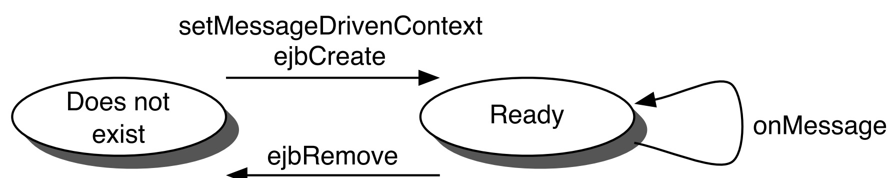

Abbildung 10: Der Lebenszyklus eines Message-Driven Beans.

9 Beispiel: MP3s ankündigen

Ich möchte in diesem Abschnitt ein kleines (mehr oder weniger) fiktives JMS Beispiel zeigen.

Als JMS Provider benutze ich OpenJMS der Firma ExoLab, da es sich hier um einen puren Java basierten Provider handelt, der einer BSD-like OpenSource Lizenz unterliegt. In dem Buch von Haefel und Chappel [3] und dem Java Magazin [4] werden weitere Provider besprochen.

Bei diesem Beispiel gibt es zunächst einmal einen MP3 Server. Dieser hat verschiedene MP3s vorrätig und bekommt laufend neue hinzu. Der Server bietet bestimmte Channels an, an denen sich interessierte Clients anmelden können um über die neuesten MP3s eines oder mehrerer Genres informiert zu werden. Hierbei könnte es sich in einer echten Applikation um einen kompletten J2EE Server mit Datenbank etc. handeln.
Auf der anderen Seite gibt es verschiedene Clients die über die neusten MP3s informiert werden möchten. Man könnte sich hier zum Beispiel ein MIDP Handy vorstellen.
Wie zu erwarten war können die Anforderungen dieser Anwendung mit einem MOM System elegant lösen. Die Clients sind lose mit dem Server verbunden und es können sich neue an- und abmelden.

9.1 MP3Server.java

```
import javax.jms.*;
import javax.naming.*;
import java.util.*;
import java.io.*;
import org.exolab.jms.jndi.rmi.RmiJndiInitialContextFactory;

public class MP3Server{

    private TopicSession tSession;
    private TopicPublisher tPublisher;
    private TopicConnection tConnection;
    private TextMessage hitListMsg;
    private Vector mp3s = new Vector();

    // ===== Konstruktor =====
    // Initialisierung f"ur den Publisher
    public MP3Server() throws Exception{

        // lokale Variablen
        Hashtable props = new Hashtable();
```

```
    String topic_name = "topic1";
    String host = "localhost";
    String port = "1099";
    String jndiname = "JndiServer";
    String mode = "rmi";
    String modeType = RmiJndiInitialContextFactory.class.getName();
    props.put(Context.PROVIDER_URL, "rmi://"+host+":"+port+"/"+jndiname);
    System.err.println("Using provider url "+props.get(Context.PROVIDER_URL));
    props.put(Context.INITIAL_CONTEXT_FACTORY, modeType);
    Context jndiContext = new InitialContext(props);

    // Lookup der connection factory
    TopicConnectionFactory tFactory =
        (TopicConnectionFactory)jndiContext.lookup("JmsTopicConnectionFactory");
    msg("Got Connection Factory. | ");

    // Erzeugen einer Connection
    tConnection = tFactory.createTopicConnection();
    msg("Established connection. | ");
    tConnection.setClientID("matt");

    // Erzeugen einer Verbindung
    tSession = tConnection.createTopicSession(false, AUTO_ACKNOWLEDGE);
    msg("Established session. | ");

    // Lookup des Themas
    Topic topic = (Topic)jndiContext.lookup(topic_name);
    msg("Got topic. | ");

    // === Bis hier wie in MP3Client.java ==

    // Erzeugen eines Publishers der Nachrichten zu dem topic schickt
    tPublisher = tSession.createPublisher(topic);
    msg("Established publisher ");

    //Initialisierung der Message die sp"ater gesendet wird
    hitListMsg = tSession.createTextMessage();

}

// Private Methode um aus den vorhandenen MP3s eine Hitliste zu erzeugen
// und diese in eine Text Message zu schreiben
private void makeHitList(String genre){
    Enumeration enum = mp3s.elements();
    MP3 mp3;
```

```
        StringBuffer sBuf = new StringBuffer();
        while(enum.hasMoreElements()){
            mp3 = (MP3)enum.nextElement();
            if(genre.equals(mp3.genre)) sBuf.append( mp3.toString()+"\n" );
        }
        try{
            hitListMsg.setText(sBuf.toString() );
            hitListMsg.setStringProperty("genre",genre);
        }catch(JMSException je){ msg(""+je); }
    }

    // Private Methode die ein MP3 hinzuf"ugt
    private void addMP3(MP3 newMp3){ mp3s.add(newMp3); }

    // Macht aus den vorhandenen MP3s eine Hitlist und versendet diese
    private void publishHitList(String s){
        msg("Publishing "+s+"\n");
        this.makeHitList(s);
        try{
        tPublisher.publish(hitListMsg);
        }catch(JMSException je){
            msg(""+je);
        }
    }

    // Private Methode die den String parst und die MP3 dann hinzuf"ugt
    private void parseMP3(String s){
        s = s.substring(2);
        StringTokenizer st = new StringTokenizer(s,"-");
        MP3 mp3 = new MP3(st.nextToken(),st.nextToken(),st.nextToken());
        this.addMP3(mp3);
        msg("Added: "+mp3+"\n");
    }

    public static void main(String args[]){
        MP3Server mp3server = null;
        try{
            mp3server = new MP3Server();
        }catch(Exception e){ msg ("Matt: Error in MP3Server.main()"+e); }

        mp3server.addMP3(new MP3("Sarah Connor","Liebeslied","Pop"));
        mp3server.addMP3(new MP3("Super DJ","I Rock","Dance"));
        mp3server.addMP3(new MP3("Heino","Ich bin blau","Volksmusik"));

        msg("\nWillkommen zu MP3 Server! Added some mp3s.\nUsage:\n");
```

```
        msg("a <Interpret - Titel - Genre>: add mp3\np <Genre>: Publish ");
        msg("genre hitlist\nq: Quit\n>");

        // === MAIN PROGRAMM LOOP ===
        String input ="";
        String input2 = "";
        while( true ){
            try{
                BufferedReader in =
                new BufferedReader( new InputStreamReader(System.in) );

                input2 = in.readLine();
                input = input2.substring(0,1);

                if(input.equals("a")){
                    //msg("parsing Mp3");
                    mp3server.parseMP3(input2);
                }
                if(input.equals("p")){
                    mp3server.publishHitList(input2.substring(2));
                    msg("Publishing hitlist\n");
                }
                if(input.equals("q")){
                    msg("Quitting.\n");
                    System.exit(0);
                }
                msg(">");
            }catch(Exception e){
                System.out.println(e);
            }
        }
    }
    private static void msg(String s){ System.out.print(s); }  //help method
}
```

9.2 MP3Client.java

```
import javax.jms.*;
import javax.naming.*;
import java.util.*;
import java.io.*;
import org.exolab.jms.jndi.rmi.RmiJndiInitialContextFactory;

public class MP3Client implements javax.jms.MessageListener{

    private TopicSession tSession;
    private TopicSubscriber tSubscriber;
    private TopicConnection tConnection;

    // ===== Konstruktor =====
    // Initialisierung f"ur den Subsriber
    // Kommentare siehe MP3Server.java
    public MP3Client(String genre) throws Exception{

        Hashtable props = new Hashtable();
        String topic_name = "topic1";
        String host = "localhost";
        String port = "1099";
        String jndiname = "JndiServer";
        String mode = "rmi";
        String modeType = RmiJndiInitialContextFactory.class.getName();
        props.put(Context.PROVIDER_URL, "rmi://"+host+":"+port+"/"+jndiname);
        System.err.println("Using provider url " + props.get(Context.PROVIDER_URL));
        props.put(Context.INITIAL_CONTEXT_FACTORY, modeType);
        Context context = new InitialContext(props);

        TopicConnectionFactory tFactory =
(TopicConnectionFactory)context.lookup("JmsTopicConnectionFactory");
        msg("Got Connection Factory. | ");

        tConnection = tFactory.createTopicConnection();
        msg("Established connection. | ");
        tConnection.setClientID("matt2");

        tSession = tConnection.createTopicSession(false, AUTO_ACKNOWLEDGE);
        msg("Established session. | ");

        Topic topic = (Topic)context.lookup(topic_name);
        msg("Got topic. | ");
```

```
        // === Bis hier wie in MP3Server.java ==

        // Erzeuge einen TopicSubcriber mit einem Selektor (z.B. genre='Pop')
        // und einem topic
        // Damit erh"allt der Subscriber nur Nachrichten von dem topic
        // in denen die String property genre meinem genre "ubereinstimmt
        String selector = "genre='"+genre+"'";
        tSubscriber = tSession.createSubscriber(topic, selector, false);
        tSubscriber.setMessageListener(this);
        msg("Subscribed to topic and listening. With selector "+selector+"\n");
        tConnection.start();
    }

    // Diese Methode wird vom JMSProvider aufgerufen falls eine
    // relevante Nachricht eintrifft
    public void onMessage(Message m){ msg("Got Hitlist: \n"+m+"\n"); }

    public static void main(String args[]){
        if(args.length == 0){
            msg("Usage: MP3Client <genre>\n");
            System.exit(0);
        }
        try{
            new MP3Client(args[0]);
        }catch(Exception e){
            System.out.println(">>> Error in MP3Server.main()");
            System.out.println(e);
        }
    }

    private static void msg(String s){ System.out.print(s); }// help method

}
```

A Definitionen:

Diese Definitionen erheben keinen Anspruch auf Vollständigkeit und Korrektheit. Im Zweifel bitte eigene Definitionen suchen.

- Middelware Definition:
  Infrastrukturen werden als Middleware-Plattformen bezeichnet, wenn sie zwischen der Anwendung und dem Betriebssystem liegen und dem Programmierer eine einfache, sichere und flexible rechnerübergreifende Verteilungsplattform bieten.

  Middleware Merkmale:

  - vereinfacht die SW-Verteilung
  - macht SW-Verteilung sicherer und flexibler.
  - unterstützt die Dezentralisierung von Anwendungen bei gleichzeitiger Erleichterung und Vereinfachung der Internet/Intranet-Integration.
  - stellt Interoperabilität zwischen Softwarekomponenten in homogenen oder heterogen Netzen bereit.
- Lagecy System: Altsystem

Literatur

[1] Bodoff, Green, Haase, Jendrock, Pawlan, Stearns *The J2EE Tutorial*
http://java.sun.com/j2ee/ 2002

[2] Kim Haase *JMS API*
http://java.sun.com/products/jms/

[3] Richard Monson-Haefel, David A. Chappell *Java Message Service*
O'Reilly

[4] Marian Kuffner *Java Message Service*
Java Magazin Ausgabe 05 2002
Enthällt auch eine Auflistung einiger JMS Provider