Universität Koblenz-Landau,
Fachbereich 4: Informatik

Anwendungsintegration für den Social Semantic Desktop mittels Publish/Subscribe

Michael-Tobias Mantsch

Diplomarbeit

Betreuer:
Prof. Dr. Steffen Staab
Dipl. Inform. Thomas Franz

Koblenz, Januar 2009

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst und keine außer den angegebenen Quellen und Hilfsmitteln verwendet habe.

Michael-Tobias Mantsch

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Übersicht

Die vorliegende Arbeit dokumentiert die Evaluation und Erarbeitung eines Moduls zur Integration von Anwendungen dahingehend, dass die Anwendungen RDF-basierte Informationen untereinander austauschen können. Dabei muss mindestens eine Anwendungen Informationen zur Verfügung stellen, und beliebige andere Anwendungen können diese Informationen abonnieren. Die Arbeit ist im Kontext der „Semantischen Desktops“ angesiedelt, allerdings soll das letztendlich angestrebte Ergebnis nicht auf den Einsatz auf einem einzelnen Desktop beschränkt sein, sondern soll auch auf mehreren Desktops in einer Art Kollaborationsnetz eingesetzt werden können.

1.1 Motivation

Im täglichen Umgang mit dem Computer und seiner Arbeitsoberfläche, dem Desktop, kommen, in der Regel, verschiedene Anwendungen zum Einsatz. Diese Anwendungen haben meist unterschiedliche Datenmodelle oder unterschiedliche Repräsentationen ihrer Daten. Dies macht eine Integration der verschiedenen Anwendungen schwierig, obwohl möglicherweise die verschiedenen Datenmodelle die gleichen Entitäten der realen Welt abbilden. Ein kleines Beispiel soll dieses Dilemma etwas verdeutlichen:

Max Mustermann sei Mitarbeiter einer fiktiven Musterfirma, die Software

entwickelt. In den Emailprogrammen seiner Kollegen und Kunden sind Max Mustermanns Kontaktdaten gespeichert. Zusätzlich kommuniziert Max mit einigen Kollegen auch über ein Instant Messaging Programm, wo er lediglich über eine bestimmte Benutzerkennung identifiziert wird. Des Weiteren ist Herr Mustermann auch in den Codeerstellungs- und Verwaltungswerkzeugen, die er täglich verwendet, als Autor von Code vorhanden. Sämtliche Repräsentationen von Max Mustermann in den verschiedenen Kontexten beschreiben die selbe reale Person aus der realen Welt. Die Anwendungen können jedoch an dieser Stelle keine Verknüpfung herstellen, da sie alle unterschiedliche Sichten auf Max Mustermann haben.

Die hier beispielhaft aufgeführten Schwierigkeiten, die bei der Integration der Informationen verschiedener Anwendungen existieren, lassen sich auf folgende Ursachen zurückführen (vgl. hierzu auch [**?**]):

1. Implizite Datenmodelle in den einzelnen Anwendungen
2. Proprietäre und möglicherweise nicht austauschbare Datenformate
3. Schmale Konzepte, die nur sehr eingeschränkte Sichten auf die Informationen zulassen und damit nur wenige oder gar keine Gemeinsamkeiten bei der Betrachtung der gleichen Information zulassen.

An dieser Stelle versucht das Konzept des „Semantischen Desktops“ anzuknüpfen und eine kontextübergreifende Sicht auf Informationen zu erzeugen.

Die Wurzeln des semantischen Desktop liegen in der Erforschung und Entwicklung semantischer Webtechnologien. Tim Berners Lee machte bereits 2001 einen entsprechenden Vorschlag zur Weiterentwicklung des WWW mit dem Ziel, Inhalten eine maschinenlesbare Bedeutung zu geben [**?**]. Damit soll es ermöglicht werden, dass Maschinen automatisch Bezüge zwischen Inhalten herstellen können, die bislang von den Benutzern des WWW selbst geknüpft werden mussten. So könnte einem beispielsweise die Suchen nach einem bestimmten Fachbuch auch gleich Informationen über den Autor, Informationen über einen thematisch verwandten Kongress oder aktuelle wissenschaftliche Veröffentlichungen auf einem ähnlichen Fachgebiet anbieten.

Das Schaffen dieser Zusammenhänge ist allerdings mit einer herkömmlichen Suchmaschine praktisch kaum möglich. Es lassen sich lediglich sehr schwache Zusammenhänge bilden, die meist auf das Vorkommen von Ausdrücken in der Nähe anderer Ausdrücke in Dokumenten basieren. Dadurch entstehen allerdings auch leicht Verwechslungen, man denke beispielsweise an die Suche nach dem Begriff "Bank". Ein Benutzer hat möglicherweise ein Geldinstitut im Sinn, ein anderen such lediglich eine neue Sitzgelegenheit für seinen Garten. Genau an dieser Stelle können semantische Systeme möglicherweise Abhilfe schaffen, die die Bedeutung des Wortes "Bank" durchaus erkennen könnten. Allerdings tun sie das nicht von sich aus, sondern der Autor des Inhalts beschreibt diesen mit Metadaten, die maschinell ausgewertet werden können und damit einen maschinenlesbaren Kontext für den gesuchten Begriff. Das heißt, während sich herkömmliche Suchmaschinen durch Unmengen unstrukturierter Daten wühlen müssen, um an Informationen zu gelangen, können semantische Systeme Inhalte deutlich leichter und vor allem präziser finden, sofern die Inhalte ausreichend beschrieben sind. Diese strukturierten Beschreibungen Die strukturierte Annotation der Inhalte erfolgt über entsprechende Sprachen und unter Anwendung festgelegter Wortschätze und Grammatiken (Ontologien). RDF und OWL sind Beispiele für formale Sprachen, die zur Erstellung von Ontologien verwendet werden können (vgl. [**?**], Kap.3, 5).

Aufbauend auf den Erkenntnissen und Erfahrungen aus der Entwicklung der Idee des Semantic Web entstand die Idee des Semantic Desktop. Prinzipiell geht es hier um die gleichen Konzepte mit dem Unterschied, dass sie auf die Applikationen und Daten eines durchschnittlichen Desktops angewendet werden. Insbesondere geht es darum, die unterschiedlichen Sichten auf Daten, die durch unterschiedliche Applikationen erzeugt werden, über eine kontextübergreifende Beschreibung zu integrieren. Beispielsweise ist eine Emailadresse möglicherweise bei der Anzeige in einem Programm als aktiver Link dargestellt, über den sich eine Email an die Adresse verfassen lässt, in einem anderen Programm wird sie aber lediglich als Zeichenkette dargestellt und behandelt. Obwohl es sich für den Betrachter um die gleiche

Emailadresse handelt, sieht der Desktop im zweiten Fall lediglich eine Zeichenkette, die keinen weitere Bedeutung hat. In der angestrebten idealen Welt des Semantischen Desktop würde diese Emailadresse, unabhängig von ihrem Anzeigekontext, auch als solche erkannt werden, wobei auch hier gilt, sofern ein Anwender sie mit den entsprechenden Annotationen versehen hat.

Die Motivation dieser Arbeit liegt nun darin, eine Integrationsmöglichkeit zu entwickeln, so dass Applikationen Informationen über ihre Inhalte austauschen können. Zudem sollte die Integration über die Desktopgrenzen hinausgehen und so zu einer Vernetzung semantischer Desktops führen, um die Zusammenarbeit in kollaborativen Netzen zu unterstützen. Prinzipiell ist die Größe der Netze nicht begrenzt allerdings liegt hier der Fokus auf "fachlichen" Netzen, in denen Menschen mit ähnlichen Aufgaben, Interessen oder Zielvorgaben miteinander arbeiten und sich untereinander austauschen. .

1.2 Thematische Einordnung

Wie oben beschrieben, ist die Arbeit im Bereich "Semantic Web / Semantic Desktop" einzuordnen. Entwickelt und betreut wird sie im Rahmen der Arbeitsgruppe "ISWEB - Informationssystem und Semantic Web" der Universität Koblenz-Landau. Forschungsschwerpunkt der Arbeitsgruppe sind Grundlagen und Anwendungsmöglichkeiten semantikbasierter Technologien. Dabei geht es insbesondere um die Integration dieser Technologien in komplexe Informationssysteme. Zur Grundlagenforschung gehören die Entwicklung von geeigneten Ontologien sowie die Beschreibung ("semantische Annotation", vlg. ISWEB) von Inhalten. Im Bereich der Anwendungsmöglichkeiten ist insbesondere die semantische Suche nach entsprechend beschriebenen Inhalten in komplexen heterogenen Systemen hervorzuheben.

1.3 Aufbau der Arbeit

Die Arbeit wird zunächst die Anforderungen an die zu erarbeitende Kommunikationskomponente aufführen und daraus das technische Umfeld abgeleitet. Dabei wird vor allem auf die zu verwendenden Technologien eingegangen und angewendete bzw benötigte Konzepte erläutert. Danach wird die oben beschriebene Integrationskomponente erarbeitet. Die Entwicklung und Beschreibung einer Testanwendung schließen den technischen Teil ab. Den Abschluss der Arbeit bilden eine kritische Beleuchtung der gesamten Entwicklung und ein Ausblick auf mögliche fortführende Entwicklungen.

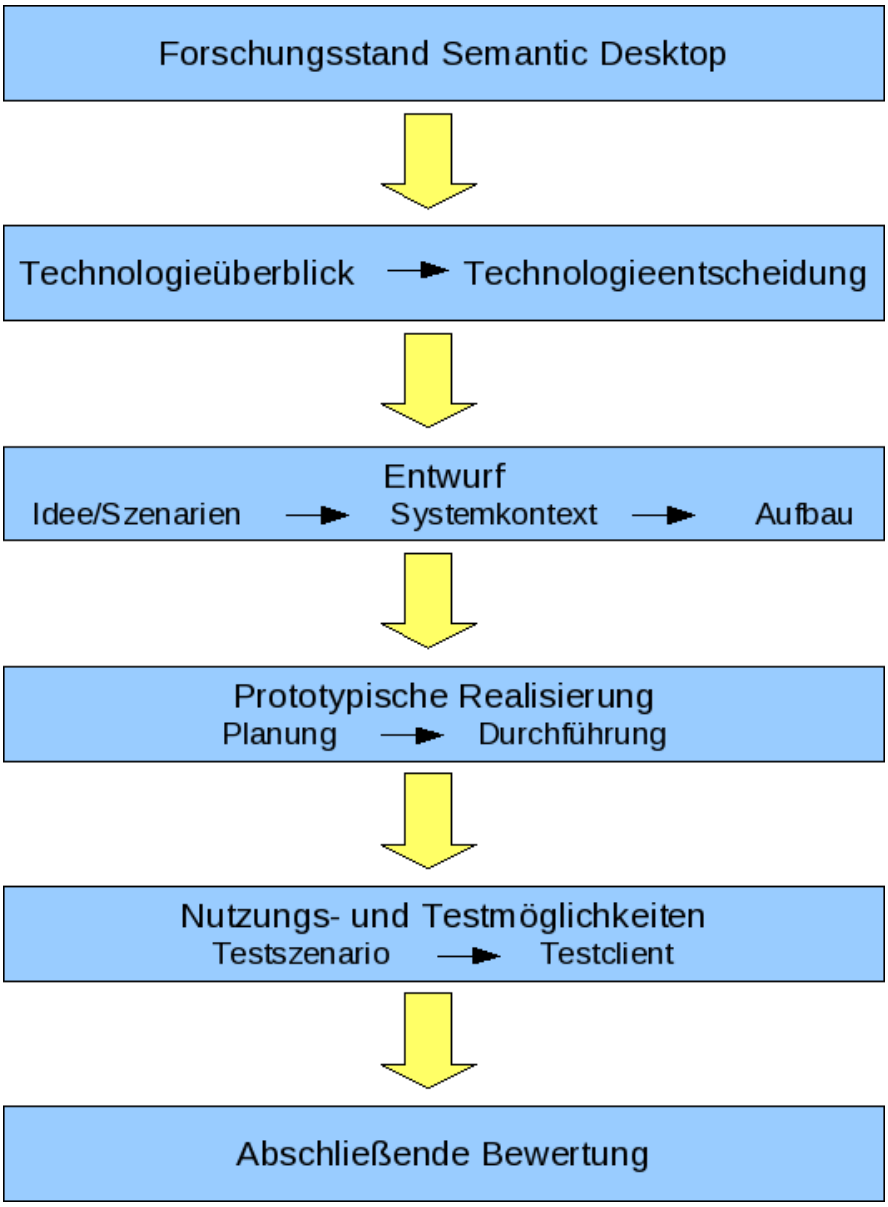

Abbildung 1.1: Grafische Übersicht des Aufbaus der Arbeit

Kapitel 2

Anforderungen

In diesem Teil werden die Anforderungen an die Kommunikationskomponente anhand eines Beispielszenarios definiert und diskutiert. Im Anschluss wird aus den Anforderung das technische Umfeld abgeleitet, das den Rahmen für diese Arbeit bieten wird.

2.1 Beispielszenario

Peter schickt Lisa über das lokale Firmennetz eine Email, die einen Dateianhang enthält. Lisa speichert diesen Anhang. Später manipuliert Lisa die Datei z.B. durch Umbenennen oder Ändern des Inhalts. Zu diesem Zeitpunkt wäre es für sie durchaus interessant, wenn der Emailclient von den Änderungen, die an dem ursprünglichen Anhang gemacht wurden, informiert werden könnte um einen Bezug zwischen der Email und dem auf der Platte gespeicherten Anhang herstellen zu können. Auf diese Art und Weise könnte bei einem späteren Weiterleiten der ursprünglichen Email der Client darauf hinweisen, dass der Anhang gespeichert und verändert wurde und anbieten, auf die geänderte Datei als Anhang zurückzugreifen. Des Weiteren wäre es in einem kollaborativen Szenarion auch für Peter von Interesse, wenn er von Änderungen an dem von ihm verschickten Anhang benachrichtigt würde, beispielsweise bei der gemeinsamen Bearbeitung eines Dokumentes.

Aus diesem Szenario und der in 1 aufgeführten Problematik ergeben sich gewisse Anforderungen an das zu entwickelnde Kommunikationskonzept, die im Folgenden aufgeführt sind. Zu jeder Anforderung wird auch die daraus resultierte Technologieentscheidung aufgeführt und begründet.

2.2 Allgemeine Beschreibung der Daten

Bevor man beginnen kann, Anwendungen auf der Basis ihrer Daten zu integrieren, muss erst mal eine gemeinsame und allgemeine Sprache zur Beschreibung der Daten und möglicherweise sogar der Operationen auf diesen Daten. Ohne ein solches Vokabular und eine solche Grammatik gibt es wenige Möglichkeit, Daten zu klassifizieren und bestimmte Ereignisse aufgrund von Klassifizierungen eintreten zu lassen. Eine Möglichkeit, Inhalte unabhängig von ihrem Kontext zu beschreiben, stellt das ***R****esource* ***D****escription* ***F****ramework*, kurz *RDF*, dar. *RDF* wurde ursprünglich als Metadaten-Schema für Webinhalte entwickelt, ist inzwischen aber zu ein Quasistandard für die Beschreibung beliebiger Inhalte geworden. Zusätzlich gibt es mit *SPARQL* eine mächtige Abfragesprache für *RDF*. Daher wird innerhalb dieser Arbeit die Möglichkeit untersucht, auf *RDF* als Beschreibungssprache zurückzugreifen. Tiefergehende Information zu *RDF* und *SPARQL* liefert das Kapitel “Technische Grundlagen”.

2.3 Plattformunabhänigkeit

Da beliebige Anwendungen und deren Daten auf möglichst beliebigen Systemen integriert werden sollen, darf das zu entwickelnde Modul nicht auf nur wenige Plattformen beschränkt sein. Die Grundidee ist schließlich, dass zwischen Peter und Lisa eine Art Zusammenarbeit stattfinden kann und soll und die beiden verwenden

- nicht notwendigerweise den selben elektronischen Arbeitsplatz

- u. U. technisch sehr unterschiedliche Systeme.

Daher würde eine starke Einschränkung der Systeme und Anwendungen die gesamte Integrationsidee ad absurdum führen. Aus diesem Grund sollte die Integrationsanwendung in Java entwickelt werden, da damit eine weitestgehende Plattformunabhängigkeit meist ohne zusätzlichen Implementierungsaufwand erreicht wird. ("Write Once, Run Anywhere" [**?**]). Zudem ist mit *XCOSIM* eine von der Arbeitsgruppe ISWEB entwickelte Architektur basierend auf Java und *RDF* verfügbar, deren Einsatz im Zusammenhang mit dieser Arbeit an späterer Stelle geprüft wird.

2.4 Lose Kopplung

Die zu integrierenden Anwendungen sollten möglichst wenig aneinander gekoppelt werden, zeitlich wie auch räumlich.

Räumliche Entkopplung ist in diesem Fall so zu verstehen, dass es nicht nötig sein sollte, sämtliche zu integrierenden Anwendungen auf dem gleichen Rechner zu betreiben. Ebenso sollte die Integration weitestgehend unabhängig von der Betriebsumgebung der Anwendungen sein.

Bei der zeitlichen Entkopplung kommt es vor allem darauf an, dass Anwendungen zeitlich nicht voneinander abhängen und dass keine Informationen verloren gehen, selbst wenn Anwendungen nicht zeitgleich relevante Informationen zur Verfügung stellen oder abrufen. Das bedeutet im Einzelnen, dass Anwendungen Daten anbieten können, diese aber nicht sofort abgerufen werden müssen. Genauso können Anwendungen Daten anfordern, die unter Umständen noch nicht zur Verfügung stehen.

Aus dieser Anforderung, speziell aus der zeitlichen Entkopplung, ergibt sich, dass in diesem Szenario eine Methode zur asynchronen Kommunikation zum Einsatz kommen muss. Unter asynchroner Kommunikation versteht man eine Kommunikation, bei der das Senden und Empfangen von Nachrichten zeitlich versetzt erfolgen kann und kein Blockieren durch das Warten auf einen der Kommunikationsteilnehmer erfolgt.

In Zusammenhang dieser Arbeit ist die Bedeutung der losen Kopplung auch dahingehend zu verstehen, dass der Austausch von Daten nicht nur zwischen genau einem Sender und genau einem Empfänger stattfinden soll. Vielmehr sollen alle Interessenten für einen Typ von Daten bedient werden, sobald die Daten zur Verfügung stehen.

Das Kommunikationskonzept *"Publish-Subscribe"* verfolgt genau diesen Ansatz. Dabei gibt es Anbieter (*Publisher*), die Information oder Nachrichten veröffentlichen und Konsumenten, die sich für den Erhalt bestimmter Informationen anmelden (*Subscriber*). Wenn ein Anbieter nun Daten sendet (*publish*), so müssen sämtliche Konsumenten, die sich für den Erhalt dieser Art von Nachrichten angemeldet haben (*subscribe*), die Nachricht erhalten.

Um zu vermeiden, dass der Datenanbieter seine Daten jedem potentiellen Empfänger direkt schicken muss und sich damit selbst um die Verteilung seiner Daten kümmern muss, bietet es sich hier an, ein Kommunikationsmodell zu wählen, das es erlaubt, eine Nachricht einmal zu senden, damit aber möglicherweise viele Empfänger zu erreichen.

2.5 Integration innerhalb eines kollaborativen Netzes

Um überhaupt die Verwendung in einem Netz von zusammenarbeitenden Menschen zu gewährleisten, muss es gegeben sein, dass die einzelnen Kollaborationspartner sich untereinander austauschen können. Dieser Anforderung kann durch die Verwendung unterschiedlicher Topologien Rechnung getragen werden:

2.5.1 Exkurs: P2P vs Middleware

Peer-to-Peer Kommunikation (P2P)

Eine Möglichkeit der Kommunikation der Teilnehmer eines Netzes untereinander ist der *Peer-to-Peer*-Ansatz. Dabei ist prinzipiell jeder der Kommunikationspartner mit jedem anderen verbunden. Damit entfällt theoretisch die Notwendigkeit eines zentralen Servers, über den zentral kommuniziert wird. Weitere Pluspunkte für *P2P* sind das Teilen von Resourcen unter den Teilnehmern des Netzes, und es besteht nicht die Gefahr, dass durch das Ausfallen eines Knotens das gesamte Netz arbeitsunfähig wird . Diese Art der Kommunikation hat im Kontext dieser Arbeit allerdings auch entscheidende Nachteile:

- Mit jedem Teilnehmer der zu dem Netz hinzugefügt wird verdoppelt sich die Anzahl der Verbindungen innerhalb des Netzes
- Jeder Teilnehmer ist gleichzeitig Server und Client, entsprechend muss sich die Kommunikationskomponente verhalten
- Ohne die Verwendung eines netzweit gültigen Archivs kann ein Teilnehmer nur die Nachrichten empfangen, die gesendet werden während er ein Teil des Netzes ist.

Middleware/Client-Server

Bei dem Einsatz einer Middleware, also eines oder mehrerer zentraler Kommunikationsserver, entfällt die Notwendigkeit, dass jeder Teilnehmer des Netzes gleichzeitig Server und Client ist. Da sämtliche Kommunikation über den/die zentralen Server geleitet wird, reduziert sich auch die Anzahl der Verbindungen drastisch, da pro Teilnehmer nur eine Verbindung zu dem Server hergestellt werden muss. Des Weiteren würde diese Architektur es deutlich erleichtern, Nachrichten für Clients vorzuhalten, die zwischenzeitlich

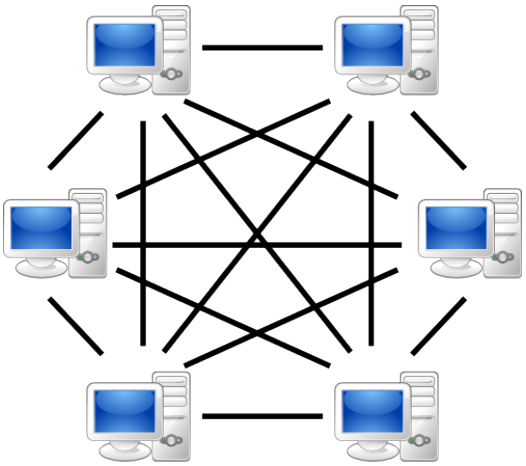

Abbildung 2.1: Schematische Darstellung eines P2P-Netzes

aus dem Kommunikationsverbund aussteigen. Jedoch birgt auch die Client-Server-Architektur Risiken und Nachteile, die gegen diejenigen von Peer-to-Peer-Netzen aufzuwiegen sind:

- Der Ausfall des Servers hat i. d. R. den sofortigen Ausfall des Betriebs des gesamten Netzes zur Folge (Single Point of Failure, [**?**], [**?**]). Dieser Nachteil kann in diesem Zusammenhang nur durch den Einsatz mehrerer redundanter Server abgemildert werden.
- Die Last auf dem Server erhöht sich mit jedem Client, der in das Netz aufgenommen wird. Das kann zu Beeinträchtigung des Betriebs bis hin zum Totalausfall führen. Auch hier kann lediglich durch den Einsatz redundanter Server und entsprechendem Load-Balancing nachgeholfen werden.

Auswertung

Aufgrund der oben angeführten Anforderungen fiel die Wahl auf die Middleware-Architektur. Insbesondere die Anforderung der losen Kopplung sprach gegen eine Lösung die P2P-Technologien einsetzt. Des Weiteren gibt es mit *JMS* (Java Message Service) ein mächtiges Framework, das Pu-

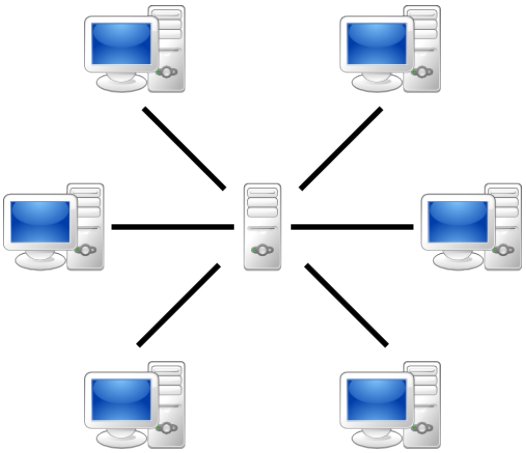

Abbildung 2.2: Serverbasiertes Netzwerk

blish/Subscribe im Java-Umfeld realisiert [?]. Grundsätzlich ließe sich Publish/Subscribe auch im P2P-Umfeld einsetzen, wie bei dem späteren folgenden Blick über den Tellerrand zu erkennen sein wird, allerdings würde das den Fokus dieser Arbeit komplett verschieben und den Rahmen sprengen, da wesentlicher Aufwand gebracht werden müsste, um das Routing, die Synchronisation und insbesondere auch die Ausfallsicherheit eines solchen Systems zu entwickeln

2.6 Zusammenfassung

Nach der Definition und Auswertung der Anforderungen an die zu entwickelnde Integrationskomponente lässt sich nun ein recht eindeutiger technischer Rahmen abstecken:

- Evaluation einer geeigneten Sprache zur Beschreibung und Abfrage der Daten
- Java als Implementierungssprache, um eine höchstmögliche Plattformunabhängigkeit zu erreichen
- Aufbau einer Client-Server-Architektur mit JMS

- Asynchrone Kommunikation nach dem Prinzip *Publish-Subscribe*

Bevor nun die konkrete Entwicklung vorgestellt wird, wird noch ein Blick auf einige Grundlagen und auf vergleichbare Arbeiten anderer Forschungsgruppen gewährt.

Kapitel 3

Technische Grundlagen und vergleichbare Forschung

Bereits weiter oben wurden einige grundlegende Informationen zum Semantischen Desktop gegeben, die eine grobe Einordnung des Themas zulassen. In diesem Abschnitt wird nun ein kurzer Blick über den Tellerrand geworfen und einige grundsätzliche Ideen und andere Forschungsansätze vorzustellen, die sich ebenfalls mit dem Thema befassen. Das Ziel ist dabei auch, die vorliegende Arbeit gegen andere Entwicklungen abzugrenzen. Grundsätzlich lässt sich sagen, dass "Semantic Desktop" ein sehr junges Thema ist, das allerdings sehr aktiv vorangetrieben wird. Stefan Decker prägte den Begriff und stellte in seiner Arbeit "The Networked Semantic Desktop" auch gleich seine Idee von der Entwicklung vom Semantischen Web hin zum Semantischen Desktop vor [**?**].

Die jährlich stattfindende "International Semantic Web Conference ISWC", sowie die zweijährig stattfindende "International Conference on Knowledge Capture K-CAP" geben Interessierten regelmäßig einen Einblick in aktuell laufende Forschungsprojekte auf diesem Gebiet, sowie auf erste produktive Umsetzungen der Konzepte von Semantic Web und Semantic Desktop.

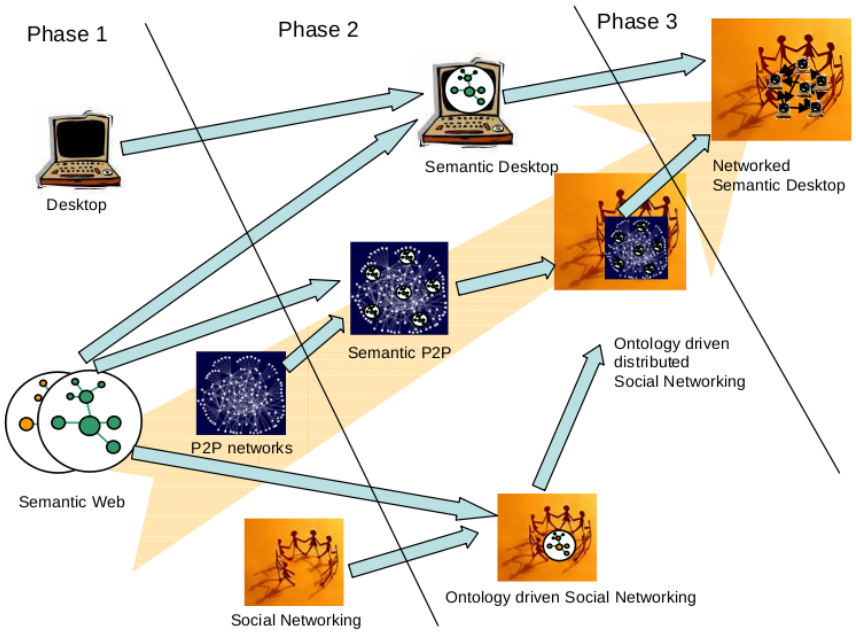

Abbildung 3.1: Entwicklung des Networked Semantic Desktop nach [?]

3.1 RDF

Das ***R**esource **D**escription **F**ramework* ist eine formale Sprache zur Beschreibung von Ressourcen im Internet. Unter Ressourcen versteht man in diesem Kontext Metainformationen zu Webinhalten, wie, zum Beispiel, Titel, Autor oder Kurzbeschreibung [?]. *RDF* wurde bereits 1997 beim W3C vorgeschlagen als Möglichkeit zur Klassifizierung und Beschreibung von Webinhalten mit dem Ziel, Katalogisierung oder Durchsuchen von Inhalten zu verbessern. *RDF* wurde 2004 vom World Wide Web Consortium (W3C) als Empfehlung ausgegeben und wurde damit zum Quasistandard für die Beschreibung von Daten im Web. Entsprechend befinden sich auf den Seiten des W3C Unmengen von Informationen zu *RDF*, die das Thema umfassend erötern. Daher wird im Folgenden lediglich ein kurzer Überblick über *RDF* gegeben.

3.1.1 Grundlagen

Grundsätzlich beschreibt *RDF* Daten in so genannten Tripeln bestehend aus:

- *Ressourcen* oder auch Subjekten. Ressourcen repräsentieren die beschriebenen Daten. Das können Webseiten oder sonstige im Web angebotenen Inhalte sein aber auch beliebige andere Objekte, wie z. B. Bücher. Wichtig ist aber, dass die Ressource durch eine eindeutige Bezeichnung identifiziert wird.

- *Eigenschaften* oder auch Prädikaten. Die Eigenschaften bilden den beschreibenden Kontext für die Ressourcen und stellen gleichzeitig den Bezug zu den *Objekten* her.

- *Objekten*. Die Objekte stellen die tatsächlichen Werte dar, die die Eigenschaften für die Ressourcen annehmen.

Ein *RDF*-Modell besteht aus einem oder mehreren Tripeln bestehend aus jeweils einer Ressource, einer Eigenschaft und einem Objekt. Alternativ lässt sich auch sagen, dass es aus Tripeln von jeweils einem Subjekt, einem Prädikat und einem zugeordneten Objekt besteht.

Zur Darstellung von *RDF*-Modellen wurden mehrere Ansätze entwickelt, die im Folgenden kurz beleuchtet werden.

3.1.2 RDF-Graph

Als eingängigste Darstellungsform wurde ein gerichteter Graph gewählt. Ressourcen werden dabei als Ellipsen dargestellt. Eigenschaften sind benannte Kanten, die von einer Ressource zu einem Objekt, dargestellt als Rechteck, führen. Abbildung 3.2 zeigt einen beispielhaften *RDF*-Graphen.

Aus der Grafik lassen folgende Informationen ableiten:

- Das Subjekt “http://www.example.org/index.html” hat für die Eigenschaft “http://purl.org/dc/elements/1.1/creator” den Wert “http://www.example.org/staffid/85740”

- “http://www.example.org/staffid/85740” ist selbst ebenfalls ein Subjekt in weiteren beschreibenden Tupeln, für die Eigenschaft

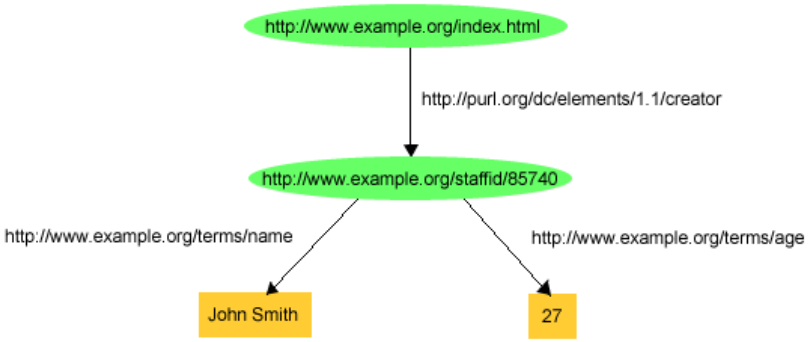

Abbildung 3.2: Einfacher RDF Graph [**?**]

> "http://www.example.org/terms/name" hat es den Literalwert "John Smith" und für die Eigenschaft "http://www.example.org/terms/age" hat es den Literalwert "27".

Umgangssprachlich ausgedrückt würde man formulieren, dass der Erzeuger von "http://www.example.org/index.html" bei "http://www.example.org/" die Mitarbeiternummer "85740" hat, "John Smith" heißt und 27 Jahre alt ist. mit *RDF* hat man diese umgangssprachliche Aussage nun formalisiert und in eine maschinenlesbare Form gebracht, sofern die Maschine die verwendeten Eigenschaften versteht, d. h. die angewandte Ontologie versteht.

Zwei wichtige Eigenschaften von *RDF* lassen sich aus dem RDF-Graphen noch ableiten: Zum Einen, lassen sich die als Prädikat verwendeten Ressourcen ebenfalls mit *RDF* beschreiben. Zum anderen bildet ein komplettes Tripel an sich ebenfalls eine Ressource, die wiederum näher beschrieben werden kann. Diese als **Reifikation** bekannte Eigenschaft von *RDF* ist essentiell, da sie erlaubt, nicht nur Aussagen über Ressourcen zu machen, sondern auch über die Aussagen selbst Aussagen zu treffen.

3.1.3 RDF/XML

Der oben beschriebene *RDF*-Graph braucht auch eine syntaktische Repräsentation, um ihn maschinenlesbar zu machen. Eine Möglichkeit ist

RDF/XML [**?**]. Abbildung 3.3 zeigt den Graphen aus Abbildung 3.2 als vereinfachtes RDF/XML.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
             xmlns:dc="http://purl.org/dc/elements/1.1/"
            xmlns:exterms="http://www.example.org/terms/">

  <rdf:Description rdf:about="http://www.example.org/index.html">
      <dc:creator><http://www.example.org/staffid/85740></dc:creator>
  </rdf:Description>
  <rdf:Description rdf:about="http://www.example.org/staffid/85740">
      <exterms:name>John Smith</exterms:name>
      <exterms:age>27</exterms:age>
  </rdf:Description>
</rdf:RDF>
```

Abbildung 3.3: Beispiel für RDF/XML Notation

Für einen kurzen, aber intensiven Einblick in *RDF/XML* sei an dieser Stelle "An Idiot's Guide to the Resource Description Framework" von Renato Iannella empfohlen [**?**].

3.2 SPARQL

SPARQL ist eine Möglichkeit, in RDF notierte semantische Tripel abzufragen [**?**]. Die Sprache ist dabei entfernt an SQL angelehnt. Ein kurzes Beispiel soll die Verwendung von SPARQL erläutern, wobei wiederum auf den oben vorgestellten Graphen zurückgegriffen wird. Der Anschaulichkeit halber, hier die Informationen aus dem Graphen in Tripelnotation:

```
<http://www.w3.org/1999/02/22-rdf-syntax-ns#> dc:creator <http://www.example.org/staffid/85740>
<http://www.example.org/staffid/85740> exterms:name "John Smith"
<http://www.example.org/staffid/85740> exterms:age "27"
```

Um jetzt beispielsweise den Namen des "http://www.example.org/staffid/85740" abzufragen würde man die SPARQL-Abfrage aus Abbildung 3.4 starten.

```
SELECT ?name
WHERE {<http://www.example.org/staffid/85740> exterms:name ?name}
```

Abbildung 3.4: Beispiel für SPARQL Abfrage

3.3 X-COSIM

X-COSIM ist ein von der Arbeitsgruppe ISWEB and der Universität Koblenz-Landau entwickeltes Framework zur kontextübergreifenden Informationsverwaltung. Es besteht aus einer Ontologie X-COSIMO und einem API, X-COSIMA [**?**]. Dabei liegt das Hauptaugenmerk auf der Integration von Personal Information Management Systemen (PIM)

X-COSIM ist dreischichtig aufgebaut. Die unterste Schicht bildet X-COSIMO, das Referenzmodell und ein RDF-Datastore, der entsprechende Daten sinnvoll verwalten kann. Darüber liegt X-COSIMA, ein API, über das Applikationen auf den RDF-Datastore zugreifen können. Die oberste Schicht bilden die Clients, die RDF-annotierte Daten erzeugen und über X-COSIM verwalten.

X-COSIM wäre ein durchaus geeigneter Ansatz zur Verwaltung und Abfrage der semantischen Daten, die im Zusammenhang mit der hier zu entwickelnden Semantischen Middleware anfallen.

3.4 Mit dieser Arbeit vergleichbare Ansätze

Bereits 2004 stellten Chirita, Idreos, Koubarakis und Nejdl mit "Publish/Subscribe for RDF-based P2P Networks" [**?**] ein vergleichbares Konzept vor. In einem P2P Netz sollten sich Peers für bereitgestellte Informationen anmelden können. Die Anmeldung werden als RDF-Queries formuliert. Information, die diesen Anfragen entspricht muss dann durch das Netz zu den entsprechenden Abonnenten geroutet werden. Um das zu erreichen, mussten allerdings im Netz sogenannte Super-Peers existieren, die die Anfragen und Angebote ihrer zugeordneten Peers verwalten und durch das Netz routen

([**?**], S.5ff). Die vorgestellten Konzepte, insbesondere das Routing und die RDF-basierten Subscriptions sollten in Edutella, einem entsprechenden Prototypen verwirklicht werden. Allerdings lässt sich heute (Oktober 2008) kein Hinweis darauf finden, dass diese Arbeit tatsächlich fortgeführt wurde.

Li und Jiang stellten 2004 einen dieser Arbeit sehr ähnlichen Ansatz für eine Semantische Middleware für Publish/Subscribe Netze vor ([**?**]). Ihre Implementierung basierte ebenfalls auf einem JMS-Backend. Darauf wurde ein semantisches System aufgesetzt, das auf der Basis von DAML+OIL (eine Ontologie basierend auf RDF, s. http://www.daml.org) die Topics in JMS auf DAML-Klassenbeschreibung abbildet. Subscriber abonnieren mit einer, DAML-Klassenbeschreibung, die auf eines der vorhandenen Topics passt. Publisher veröffentlichen Nachrichten als DAML-Instanzbeschreibungen. Eine Logik prüft, auf welche Klassenbeschreibung die von einem Publisher veröffentlichte Nachricht passt. Dieses System ist der in dieser Arbeit entwickelten Lösung sehr ähnlich, jedoch unterscheidet es sich in einigen grundlegenden Punkten, auf die bei der Vorstellung des hier erarbeiteten Konzeptes explizit hingewiesen wird.

Kapitel 4

SeMOM - Idee und Konzept

Ausgehend von den in Kapitel 2 erarbeiteten Ergebnissen wurde ein Konzept aufgebaut, nach dem das beschriebene Ziel realisiert werden sollte. Als Name für das Projekt wurde von mir willkürlich "SeMOM" gewählt, als Abkürzung für "Semantic Message Oriented Middleware".

4.1 Idee

Am Anfang steht meist eine grobe Idee und so war das auch im Fall von SeMOM. Angeregt durch die Arbeit von Li und Jiang [**?**] entstand das grobe gedankliche Gerüst:

- Ein JMS-Backend bietet die Publish/Subscribe Funktionalität und stellt Topics zur Verfügung.
- Clients kommunizieren nicht direkt mit dem JMS-Backend, sondern mit einer darüber gelegten Applikation, die zwischen den Clients und dem JMS-Backend vermittelt.
- Clients melden sich an SeMOM mit einer Art semantischem Filter an, mit dem sie ausdrücken, für was für Nachrichten sie sich anmelden. Sobald ein Client sich anmeldet, wird für ihn ein dedizierter Subscriber erzeugt, der für ihn die Nachcrichten vom JMS-Backend empfängt.

Soweit liest sich das ganze nach genau dem, was Li und Jiang [?] in ihrer Arbeit beschrieben haben. Nun kommen aber die entscheidenden Punkte, an denen sich SeMOM unterscheidet:

- Es gibt nur ein Topic, auf das alle Nachrichten geschrieben werden. Damit entfällt ein Abgleich zwischen Topicbeschreibungen und Nachrichteninhalten.
- Jeder Teilnehmer darf Nachrichten an das System schicken. Diese Annahme darf getroffen werden, da das anvisierte Einsatzgebiet von SeMOM kleinere kollaborative Netze von sich vertrauenden Partnern oder gar nur Applikationen auf ein und demselben Desktop sind.
- Aus den beiden obigen Punkten ergibt sich, dass die Notwendigkeit für Publisher-Agenten, wie sie bei Li und Jiang besteht, entfällt. Damit reicht dem System ein Publisher, der für alle Clients Nachrichten aufnimmt und diese an das Topic schickt.
- Jeder Client gibt über einen oder mehrere semantische Filter an, für welche Art von Nachrichten er sich interessiert. Diese Filter werden von dem für den Client bereitgestellten Subscriber-Agenten auf ankommende Nachrichten angewandt und es werden nur diejenigen weitergeleitet, die durch den Filter durchgelassen werden.
- Solange ein Client sich nicht explizit vom System trennt, bleibt sein Subscriber-Agent am Leben und sammelt weiter Nachrichten für seinen Client. Auf diese Art wird sichergestellt, dass Clients bei Störungen oder freiwilligem kurzzeitigen Ausscheiden aus dem Verbund beim Wiedereintritt die Nachrichten noch erhalten, die während ihrer Abwesenheit für sie bestimmt waren.
- Das System soll für alle Arten von Clients offen sein, also nicht ausschließlich Java Clients bedienen können. Li und Jiang sprechen in ihrer Arbeit nicht explizit an, dass nur Java Clients unterstützt werden, allerdings lassen die Beschreibungen der Abläufe und die Grafiken darauf

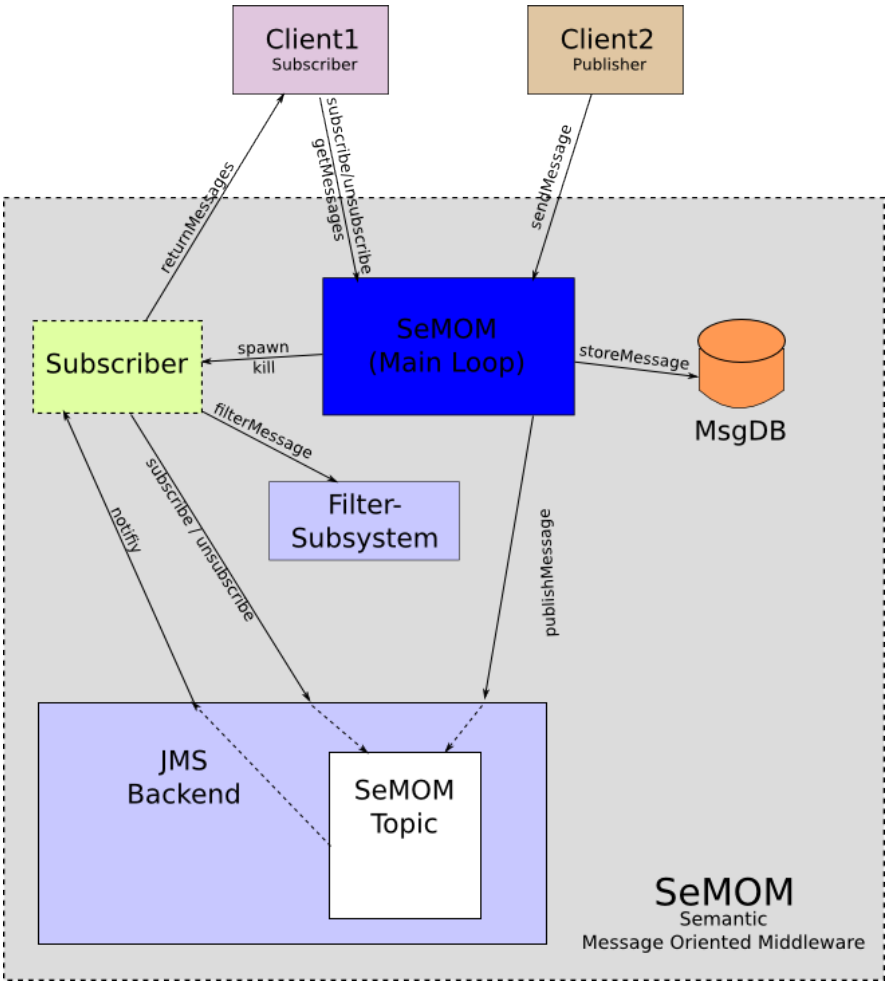

Abbildung 4.1: Übersicht über die komplette Architektur

schließen. Um die Offenheit des System auch zu demonstrieren, soll eine Testanwendung entwickelt werden, die nicht auf Java basiert. Dieser Punkt birgt einen Nachteil: Die Clients müssen explizit nach den Nachrichten fragen, da ein Notification-System für beliebige Anwendungen den Rahmen der Arbeit deutlich gesprengt hätte.

4.2 Systemübersicht

Abbildung 4.1 bietet einen groben Architekturüberblick, der aus den obigen Ideen entwickelt wurde. Die einzelnen Komponenten und ihre Aufgaben werden in den folgenden Kapiteln beschrieben.

4.2.1 JMS Backend

Das JMS Backend ist genau das, was der Name aussagt, das Werkzeug, mit dessen Hilfe das Messaging und Publish/Subscribe behandelt wird.

4.2.2 MsgDB

Diese Datenbank macht derzeit nichts anderes als die empfangenen Nachrichten aufzunehmen. Die "SeMOM (Main Loop)" legt darin empfangene Messages ab. Dies geschieht zur Zeit ausschließlich zu Archivierungszwecken. Tatsächlich wird mit den in der Datenbank gespeicherten Nachrichten nichts weiter gemacht.

4.2.3 SeMOM (Main Loop)

Das ist das Kernmodul, das die direkte Kommunikation mit den Clients regelt:

- Annehmen von Subscribe-Requests und ggf. erzeugen von Subscriber-Agenten
- Annehmen von Unsubscribe-Requests und entfernen des entsprechenden Subscriber-Agenten
- Annehmen von Publish-Requests, speichern der Nachrichten in die MsgDB und senden der Nachrichten an das JMS-Topic
- Annehmen von GetMessages-Requests

Dieses Modul ist zweigeteilt, der Nachrichten produzierende Teil und der Nachrichten konsumierende Teil sind voneinander getrennt. Das hat vor allem den Grund, dass damit weniger Thread-Synchronisation vonnöten ist und dass sich das "Hängen" einer der beiden Teile nicht negativ auf den anderen Teil auswirkt.

4.2.4 Subscriber

Hierbei handelt es sich um die Agenten, die pro Client einmal erzeugt werden und die Kommunikation mit dem JMS-Backend für die angeschlossenen Subscriber-Clients übernehmen. Die Subscriber sind selbst dafür verantwortlich Nachrichten zu filtern. Dazu werden sie beim Erzeugen mit dem vom Client mitgegebenen Filter initialisiert. Clients haben die Möglichkeit nochmal mit einem zusätzlichen Filter zu abonnieren. In diesem Fall wird kein neuer Subscriber-Agent erzeugt, sondern lediglich dem bereits bestehenden Subscriber ein weiterer Filter hinzugefügt, den er bei den ankommenden Nachrichten beachten muss. Nachrichten, die keinem der Filter eines Subscribers entsprechen, werden verworfen. Alle anderen Nachrichten werden so lange vorgehalten, bis der Client, der zu dem Subscriber gehört, die Nachrichten abholt. Sobald ein Client einen Unsubscribe-Request an das System verschickt, wird der zugehörige Subscriber von dem JMS-Topic als Abonnent entfernt und schließlich zerstört. Solange aber kein Unsubscribe-Request kommt, bleibt der Subcriber am Leben und sammelt Nachrichten für seinen Client, auch wenn dieser nicht da ist. Damit ist gewährleistet, dass der Client keine Nachrichten verpasst.

4.2.5 Filter-Subsystem

Über dieses System filtern die Subscriber-Agenten die Nachrichten, die an sie geschickt werden. Da keine Filterung auf Topic-Basis stattfindet, bekommen Subscriber erst mal grundsätzlich alle Nachrichten und müssen diese dann gegen ihre Filter prüfen. Nach der ursprünglichen Idee sollte das Filtersystem

ein semantisches System sein, über das Nachrichten entsprechend geprüft werden. Was schließlich daraus geworden ist, dazu an späterer Stelle mehr.

4.2.6 Clients

Clients können prinzipiell alle Applikationen sein, die sich mit dem System verbinden können und das vereinbarte Protokoll sprechen. Konkret heißt das, es muss ihnen möglich sein, sich über einen TCP-Socket zu verbinden und darüber einen XML-Stream zu senden und zu empfangen. In diesem speziellen Fall wurde zu Demonstrationszwecken entschieden, eine kleine KDE-Applikation zu entwickeln, die sich mit SeMOM verbindet, Nachrichten schickt und Nachrichten erhält. Dabei sollten die Nachrichten kontextuell mit einer Datei verbunden sein, die der Benutzer in seinem Dateimanager anklickt.

KDE ist eine Desktopumgebung für den Linux X-Server. Im Zeitraum der Entwicklung der vorliegenden Arbeit lag KDE in Version 4.1 vor Diese galt allgemein als Entwicklerversion, die noch nicht ausreichend stabil für einen irgendwie gearteten produktiven Einsatz war. Daher entschied ich mich, den Testclient auf der alten ausgereiften KDE Version 3.5 zu entwickeln.

4.3 Eingesetzte Technologien und Werkzeuge

Als Basissystem für die Entwicklung habe ich OpenSuse 10.3 gewählt. Als Linuxbasiertes System mit KDE als Desktopumgebung hielt ich es für sehr gut geeignet, die in dem Projekt vorgesehene KDE-Client-Entwicklung zu unterstützen. Zudem ist das System, meiner Meinung nach leicht zu installieren und administrieren. Das System wurde allerdings nicht nativ auf einem Rechner betrieben, sondern lief als virtueller Gast auf einem VirtualBox Hostsystem.

Als JMS-Provider wurde der Application Server von Sun “Glassfish” in der Version “v2ur2” gewählt. Dies lag zum Einen daran, dass er die aktu-

ellste Spezifikation für JEE und das aktuellste Java Runtime Environment unterstützt. Zum Anderen bestand meinerseits bereits etwas Erfahrung im Umgang mit dem Applicationsserver, Wissen, das ich bei der Verwendung einer anderen Umgebung erst hätte aufbauen müssen. Des Weiteren wird bei Glassfish gleich eine kleines DBMS namens Derby mitgeliefert, das für die Archivierung der Nachrichten verwendet wurde.

Aufgrund der hervorragenden Integration der Entwicklungsumgebung NetBeans mit dem Application Server wurde die Java-seitige Entwicklung komplett mit Netbeans gemacht.

Für die Entwicklung des Testclients wurde KDevelop 3.4.1 auf KDE 3.5.7 eingesetzt.

Kapitel 5

Implementierung

Die Implementierung von SeMOM und des Testclients gliederte sich hauptsächlich in drei Phasen:

1. Schaffung der technischen Grundlagen
2. Implementierung gemäß weiter oben vorgestelltem Konzept
3. Anpassungen und Verfeinerung

Grundsätzlich kann ich sagen, dass ich mit beidem nahezu zeitgleich begonnen habe, um beide Komponenten auch relativ gleichzeitig fertig zu haben und zu vermeiden, dass ein zu langes Befassen mit einer der Komponenten es verhindert, dass die andere Komponente fertig gestellt werden kann.

5.1 Schaffung der technischen Grundlagen

Auf die Installation der einzelnen Komponenten wird an dieser Stelle nicht eingegangen, da das nicht dem Fokus dieses Dokumentes entspricht. Lediglich so viel sei gesagt:

1. Nach der Installation von Glassfish müssen darin ConnectionFactories und mindestens ein Topic angelegt werden. Das geschieht recht kom-

fortabel über die mitgelieferte Weboberfläche. Genauere Informationen dazu finden sich im Anhang.

2. Die Einrichtung der Derby Datenbank ist ebenfalls kaum der Rede wert, auch hierzu gibt es im Anhang eine kleine Anleitung.

Für den Testclient mussten weniger technische Grundlagen geschaffen als viel mehr Grundlagen beschafft werden. Leider war die Dokumentationslage nicht sonderlich ergiebig, insbesondere, weil sich sehr viel Entwicklerdokumentation und Anleitungen auf KDE 2 bezogen. Dadurch stellte sich die Implementierung des Clients auch als deutlich komplizierter heraus, als ursprünglich angenommen. Der Testclient wird im Detail in Kapitel 6 beschrieben. Nichtsdestotrotz, wurde relativ zeitgleich mit der Implementierung von SeMOM und dem Client begonnen. Um möglichst rasch eine brauchbare Kommunikation aufzubauen und nicht erst lange Zeit mit der Entwicklung von Ontologien zu verbringen, war der erste Ansatz, das bereits fertige RDF-basierte Ontologiesystem X-COSIM einzusetzen. Leider stellte hier während der Evaluation das System große Hürden auf, da weder die Version 1.0 noch die Version 1.1 zur Mitarbeit zu bewegen waren. Der mitgelieferte Sesame-Store lies sich nicht erfolgreich anbinden und starten und der Versuch einen externen Sesame-Store anzubinden scheiterte mit ClassCastExceptions. Daher habe ich, zu dem Zeitpunkt nur übergangsweise, ein kleines XML-Format entwickelt, über das die Nachrichten ausgetauscht werden sollten. Zu einem späteren Zeitpunkt sollte dann zu dem RDF/X-COSIM-Ansatz zurückgekehrt werden.

5.2 Das Übergangsprotokoll

Da sämtliche Kommunikation zu SeMOM durch die Clients zu initiieren ist, wurde ein kurzes XML-basiertes Requestprotokoll aufgebaut.

Die folgende Übersicht schlüsselt die Elemente des Protokolls auf und zeigt mögliche Werte:

request: Wurzelelement

```
<request>
  <action></action>
  <client></client>
  <filter></filter>
  <totopic></totopic>
  <message>
    <localflag></localflag>
    <urlorpath></urlorpath>
    <dir></dir>
    <file></file>
    <created></created>
    <modified></modified>
    <lastread></lastread>
    <properties>
      <property>
        <name></name>
        <value></value>
      </property>
      <property>
      ...
      </property>
    </properties>
  </message>
</request>
```

Abbildung 5.1: Protokoll für SeMOM-Requests

action: Art des Requests - *subscribeWithFilter, unsubscribeAll, getAllMsg4topics, getFileMsg4topics*

client: ID des Clients - *beliebiger String, vergibt der Clients selbst*

filter: Nachrichtenfilter - *Eigenschaft (is—has) Wert* (lediglich bei *subscribeWithFilter*)

totopic: derzeit ohne Bedeutung

message: der Beginn der eigentlichen Message

localflag: derzeit ohne Bedeutung

urlorpath: URL oder Pfad zur Datei - *URL oder Pfad zur Datei*

dir: Verzeichnispfad - *¡Verzeichnispfad¿*

file: Dateiname - *Dateiname*

created, modified, lastread: Erstellungs-, Modifizierungs- und Zugangsdatum der Datei

properties: beginnt einen Bereich in dem freie Eigenschaften drinstehen

property: frei zu vergebende Eigenschaft, dürfen nur innerhalb von *properties* vorkommen. Es können beliebig viele *property* mitgegeben werden

name: Name einer mitzuschickenden *property*

value; Wert einer mitzuschickenden *property*

5.3 SeMOM

Die Hauptkomponente besteht aus zwei getrennten Modulen, die als Java Enterprise Application Client in den Applicationserver deployt werden müssen.

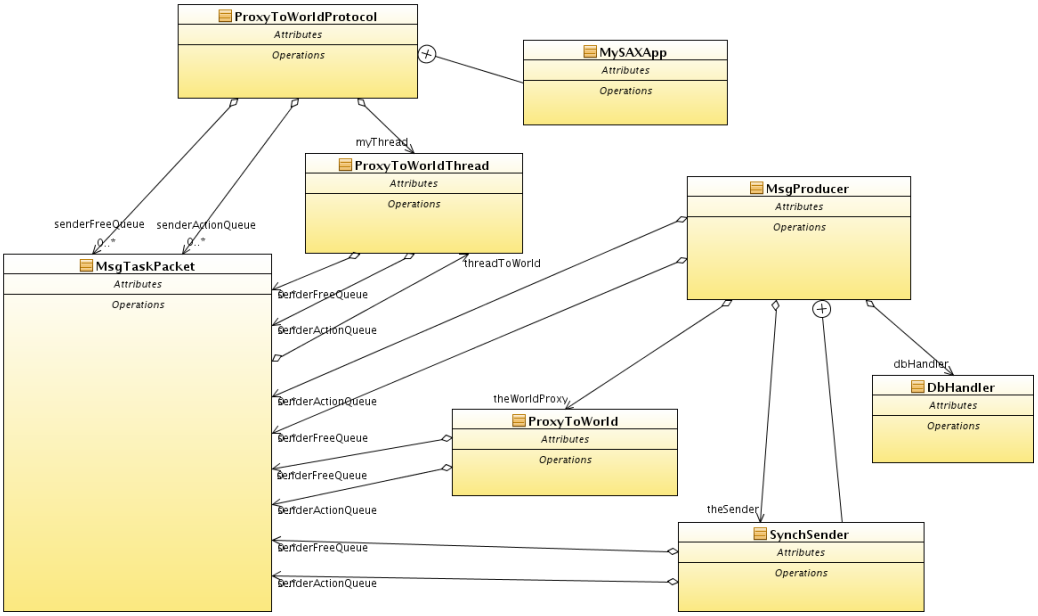

Abbildung 5.2: Klassendiagramm des Moduls MsgProducer (vereinfacht)

Das ist, zum einen, der MsgProducer. Dieser sorgt dafür, dass Nachrichten von den Clients an das Topic geschickt werden. Zum anderen ist das der MsgSubscriptionHandler, die die Abonnements der Clients überwacht und der von den Clients nach neuen Nachrichten gefragt werden kann. Hierin ist auch das Filtermodul implementiert, das auf die von den Clients abonnierten Filter anwendet.

5.3.1 MsgProducer (Modul)

Abbildung 5.2 bietet eine Übersicht über die Klassen, die das Modul ausmachen, Attribute und Methoden wurden aus Gründen der Übersicht entfernt, die folgenden Unterkapitel stellen die einzelnen Klassen etwas näher vor.

MsgProducer (Klasse)

Hierbei handelt es sich um das Hauptprogramm, das gestartet wird. Von hier aus wird alles weitere veranlasst. Insbesondere wird ein `ProxyToWorld`

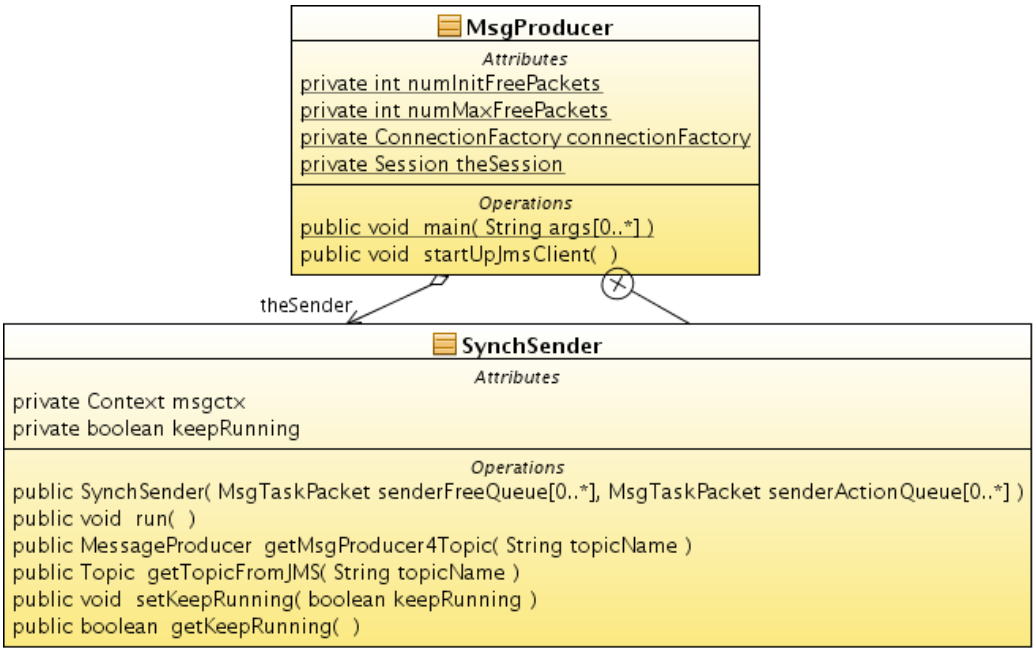

Abbildung 5.3: Die Klasse semom.msgproducer.MsgProducer

gestartet, der notwendig ist, damit Clients sich überhaupt verbinden können. Des weiteren wird ein Thread der inneren Klasse `SynchSender` gestartet, der die Kommunikation mit dem JMS-Backend übernimmt.

ProxyToWorld

Dieser Proxy stellt den ServerSocket bereit, auf dem Clients sich verbinden können. Sobald ein Client sich verbindet wird ein ProxyToWorldThread erzeugt, der den Client bedient. `ProxyToWorld` kann dann sofort neue Verbindungen aufbauen.

ProxyToWorldThread

Dienstthread, der die Kommunikation mit dem Client durchführt.

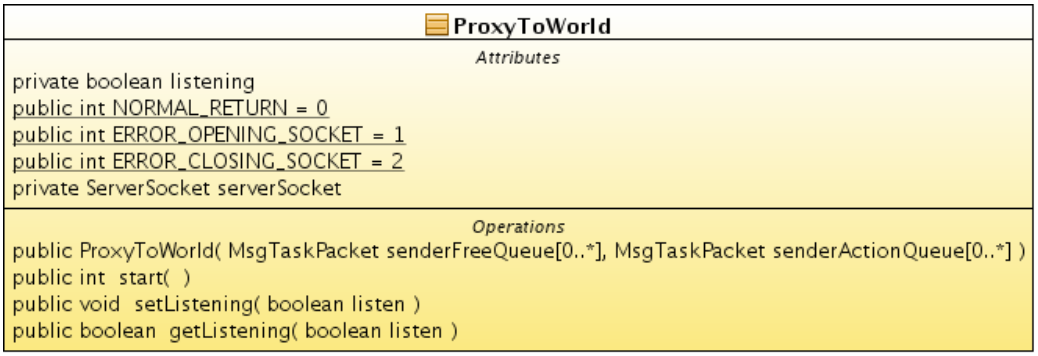

Abbildung 5.4: Die Klasse semom.msgproducer.ProxyToWorld

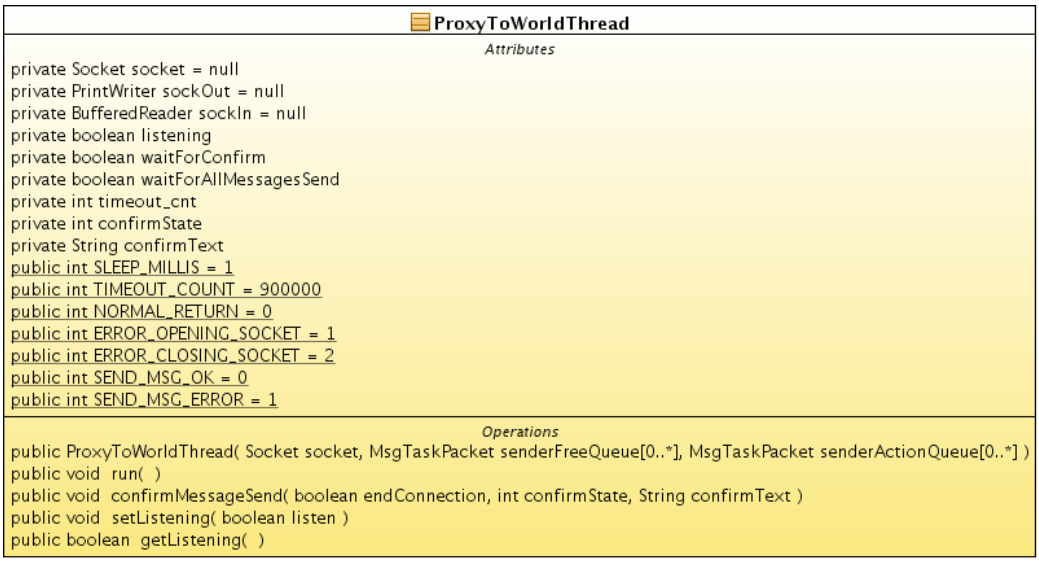

Abbildung 5.5: Die Klasse semom.msgproducer.ProxyToWorldThread

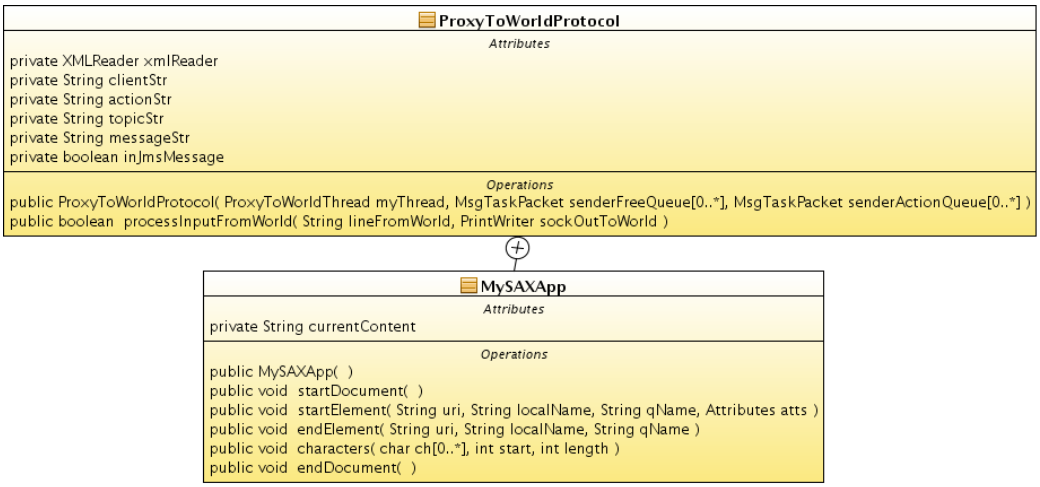

Abbildung 5.6: Die Klasse semom.msgproducer.ProxyToWorldProtocol

ProxyToWorldProtocol

Hierüber wird der vom Client ankommende Input angenommen und geprüft. Wenn es sich um einen gültigen Request handelt, wird ein entsprechendes MsgTaskPacket erstellt und in die ActionQueue des `MsgProducer` gehängt. Die innere Klasse `MySAXApp` verarbeitet den Inhalt der Nachricht und ordnet ihn neu, so dass nur die relevanten Teile an das Topic gesandt werden.

MsgTaskPacket

Representation eines Arbeitspakets, das durch den MsgProducer an das Topic geschickt werden muss. Da die Anwendung multithreaded ist, nutzen die einzelnen Komponenten untereinander LinkedBlockingDeques um Nachrichten bzw MsgTaskPackets auszutauschen. Das `MsgTaskPacket` enthält unter anderem auch die zu veröffentlichende Nachricht.

DBHandler

Diese Hilfsklasse wird genutzt, um ankommende Nachrichten in die MsgDB zu schreiben. Sie wird direkt innerhalb `ProxyToWorldProtocol` aufgerufen, sobald die eingegangene Nachricht erfolgreich verarbeitet wurde.

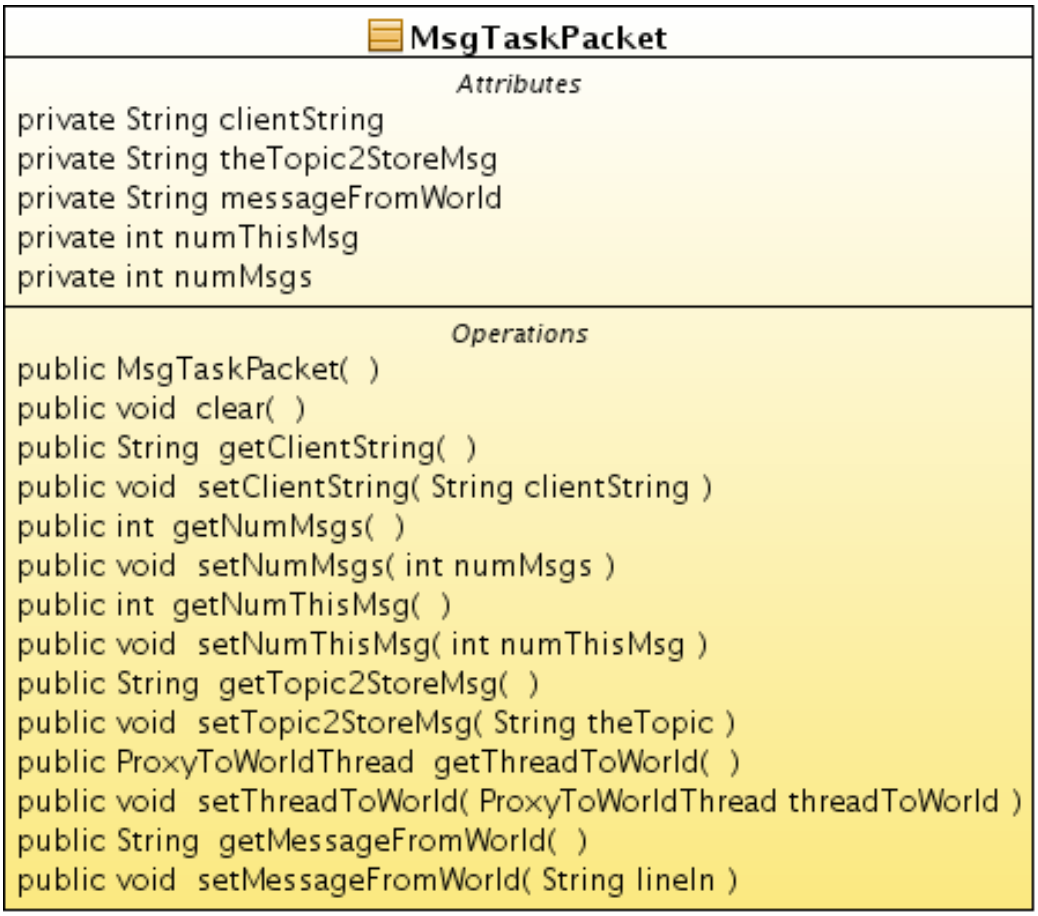

Abbildung 5.7: Die Klasse semom.msgproducer.MsgTaskPacket

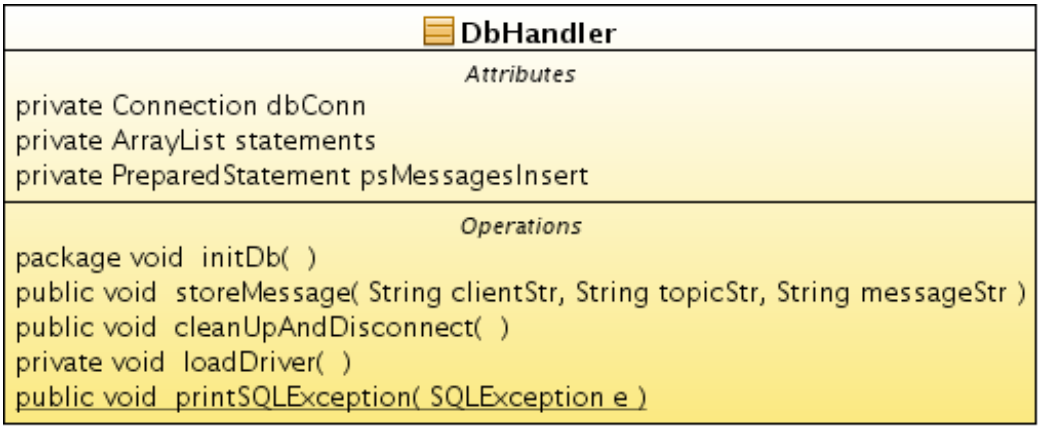

Abbildung 5.8: Die Klasse semom.msgproducer.DBHandler

Constants

Hier sind etliche Konstanten definiert, über die das Modul konfiguriert werden kann. Es gibt derzeit keine andere Konfigurationsmöglichkeit. Die zu konfigurierenden Werte sind selbsterklärend.

Ablauf eines publishing-Requests

Zunächst mal, bevor ein Request eingehen kann, startet der `MsgProducer` den `ProxyToWorld`. Dieser erzeugt einen ServerSocket auf dem in `Constants` angegebenen Port. Sobald nun ein Request auf dem Port angekommt, akzeptiert `ProxyToWorld` den request und erzeugt einen neuen `ProxyToWorldThread`, der die weitere Kommunikation mit dem Client übernimmt. `ProxyToWorld` ist damit wieder frei für weitere Requests.
`ProxyToWorldThread` erzeugt nun eine Instanz von `ProxytoWorldProtocol`, die die vom Client eingegangene Nachricht überprüft, verarbeitet und eine entsprechende Rückmeldung gibt. Eine positive Verarbeitung schreibt die Nachricht einmal in die `MsgDB` und hängt sie anschließend verpackt in ein `MsgTaskPacket` in die Verarbeitungsqueue des `MsgProducer`. Dieser hat eine innere Threadinstanz, `SynchSender`, der gleichzeitig ein Publisher auf dem Topic im JMS ist. Der `SynchSender` arbeitet die Queue ab und sendet die Messages an das JMS-Topic. Das Ganze ist auch nochmal grafisch in Abbildung 5.9 nachvollziehbar.

5.3.2 MsgSubscriptionHandler (Modul)

Das Modul `MsgSubscriptionHandler` behandelt alles, was mit dem abonnieren von Nachrichten durch die Clients und dem Versand von Nachrichten an die Clients zu tun hat. Grob gesagt heißt das, Clients abonnieren über diesen Dienst Nachrichten und fragen hier an, wenn sie ihre Nachrichten abholen wollen. Abbildung 5.10 zeigt einen Überblick über das Modul insgesamt, auf die einzelnen Klassen wird in den folgenden Kapiteln eingegangen.

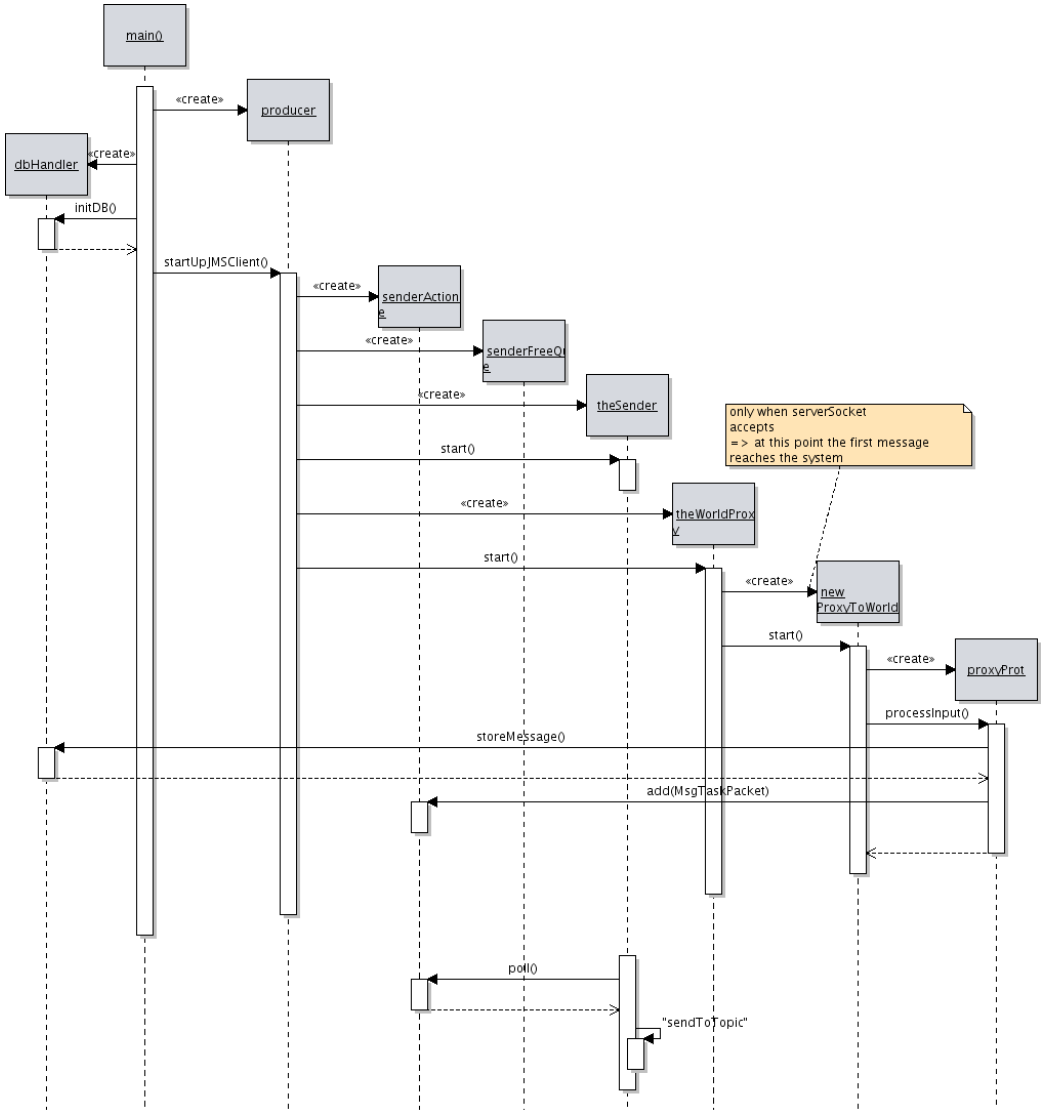

Abbildung 5.9: Darstellung des Ablaufs vom Start des MsgProducer bis zum Publishen der ersten Message

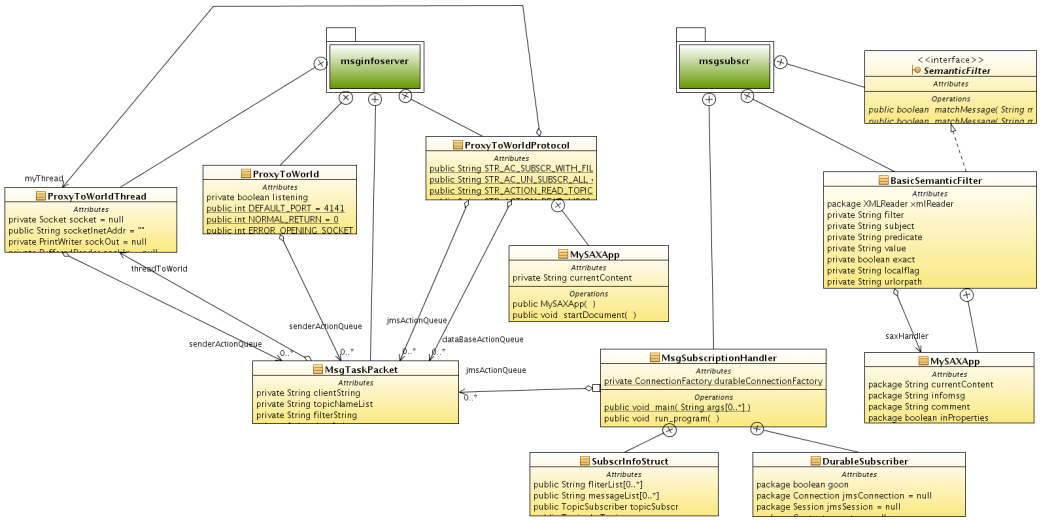

Abbildung 5.10: Übersicht über die Pakete und Klassen des Moduls MsgSubscriptionHandler

MsgSubscriptionHandler (Klasse)

`MsgSubscriptionHandler` ist die Main-Klasse der Applikation. Allerdings übernimmt sie, im Gegensatz zum `MsgProducer`, lediglich einige Konfigurationsaufgaben und übergibt die Kontrolle dann an die eigentliche Hauptapplikation, den `DurableSubscriber`. Diese Nomenklatur ist auf den ersten Blick durchaus verwirrend und das Vorgehen möglicherweise schwer nachvollziehbar. Daher erinnere ich an dieser Stelle nochmal an die drei Implementierungsphasen am Anfang des Kapitels. Ursprünglich waren die Anwendungsfälle “Subsription/Unsubscription”, “Message Retrieval” und “Publishing” auf drei einzelne Applikationen verteilt, was auch mit einer frühen Implementierungsidee zusammenhing, die allerdings wieder fallengelassen wurde. Im Zuge eines späteren Refactorings wurde damit begonnen, die Applikationen zusammenzuführen. Begonnen wurde dabei mit den Applikationen, die sich mit dem Abonnieren und Abholen von Nachrichten befassten.

Aufgrund dieses nicht hundertprozentig abgeschlossenen Refactorings sind derzeit noch einige Ungereimtheiten oder nicht mehr verwendete Abschnitte im Code zu finden, die noch nicht aufgeräumt wurden. Das ändert allerdings

nichts an der grundsätzlichen Funktionsfähigkeit des Systems.

`DurableSubscriber` ist somit die Klasse, die die Hauptarbeit verrichtet und mit der Methode `doWorkOrWaitLoop` stellt sie auch die Main-Loop des Programms. Es sei an dieser Stelle nochmal betont, die Klasse hat nichts weiter mit einem Subscriber gemeinsam, außer dem Namen! Die Aufgabe dieser Klasse ist es, die Anwendung zu koordinieren und die Requests der Clients an die richtigen Stellen weiterleiten. Wie das genau geschieht, wird weiter unten beschrieben.

`MySAXApp` ist ein nicht mehr verwendetes Überbleibsel aus der Zeit vor dem Refactoring. Mit der Einführung der Filterfunktionalität wurde diese Klasse überflüssig.

In Objekten vom Typ `SubscrInfoStruct` werden Informationen über die Abonnements der Clients gespeichert. Zu jedem Client wird ein `SubscrInfoStruct` angelegt und in einer Map verwaltet. Im einzelnen sind das folgenden Informationen:

- `filterList`:
  Eine Liste von Strings, die Filter repräsentieren. Filter werden in der Form <Eigenschaft >[is |has] <Wert >. Filter prüfen also, ob eine Eigenschaft einen bestimmten Wert hat, oder einen Wert enthält, je nachdem ob man `is` oder `has` verwendet. Da Clients lediglich einen Subscriber zugewiesen bekommen, allerdings beliebig viele Filter verwenden können sollen, werden diese in `filterList` aufgenommen.

- `messageList`:
  Hierin werden die Nachrichten gesammelt, für die sich ein Client zwar angemeldet hat, die er aber noch nicht abgeholt hat.

- `topicSubscriber`:
  Eine Referenz auf den Subscriber, der für einen Client Nachrichten aus dem Topic empfängt.

- `thetopic`:
  Eine Referenz auf das zur Kommunikation verwendete Topic

- `topicName`;
  Der Name des verwendeten Topics. Dieser sollte für alle gleich sein, da das System lediglich ein Topic verwendet.

- `clientStr`:
  Eine Zeichenkette, über die der Client möglichst genau identifiziert werden kann. Einen Teil des Strings muss der Client selbst bei der Anmeldung angeben, das System fügt dem aber noch die IP-Adresse des Clients hinzu.

- `subscrName`:
  Eine Kombination aus `clientStr` und `topicName`. Darüber werden die zu einer Zeit aktiven Subscriber in einer entsprechenden Map identifiziert.

`TextListener` ist eine Implementierung von `javax.jms.MessageListener`. Subscriber brauchen einen solchen Listener, da die Callback-Methoden des Listeners aufgerufen werden, sobald das Topic neu Nachrichten hat. Die Implementierung hier benötigt ein Objekt vom Typ `SubscrInfoStruct` zur Initialisierung. Nur mit Hilfe der Informationen in diesem Objekt kann der Listener bestimmen, ob die ankommende Nachricht tatsächlich für ihn bestimmt ist oder nicht.

Das Filtermodul

`SemanticFilter`
Dieses Interface stellt lediglich zwei Methoden zur Verfügung:
`boolean matchMessage(String msg, String filter)` und
`boolean matchMessage(String msg, Collection<String> filters)`.
Implementierung dieser Methoden müssen dafür sorgen, dass Nachrichten gegen die gebotenen Filter geprüft werden und ein entsprechendes Ergebnis zurückliefern.

`BasicSemanticFilter`

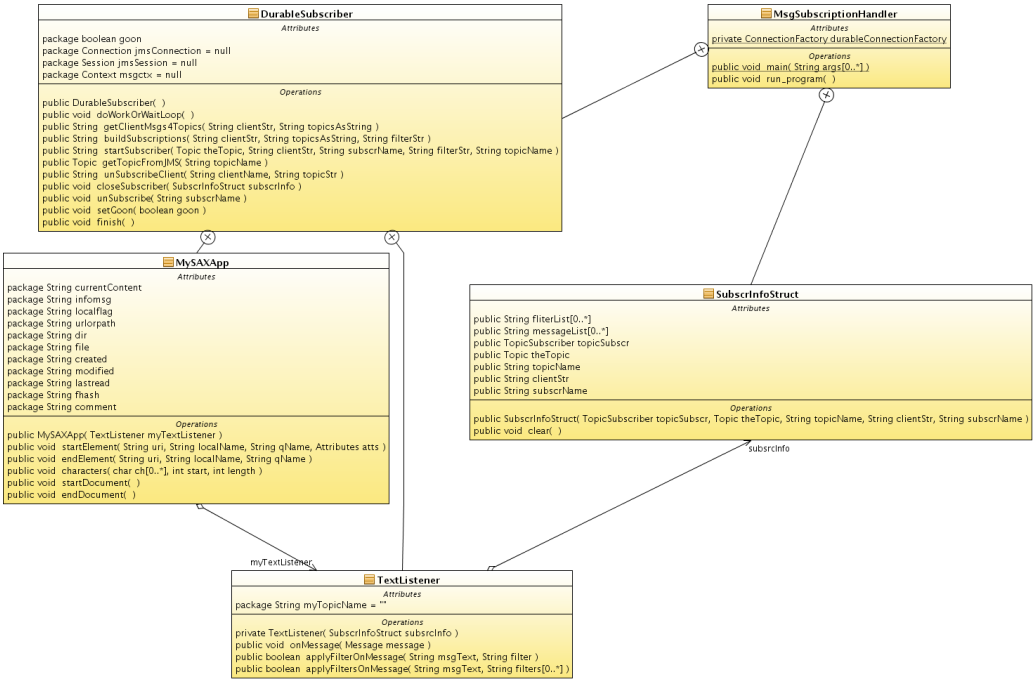

Abbildung 5.11: Die Klasse MsgSubscriptionHandler und eingebettete Klassen

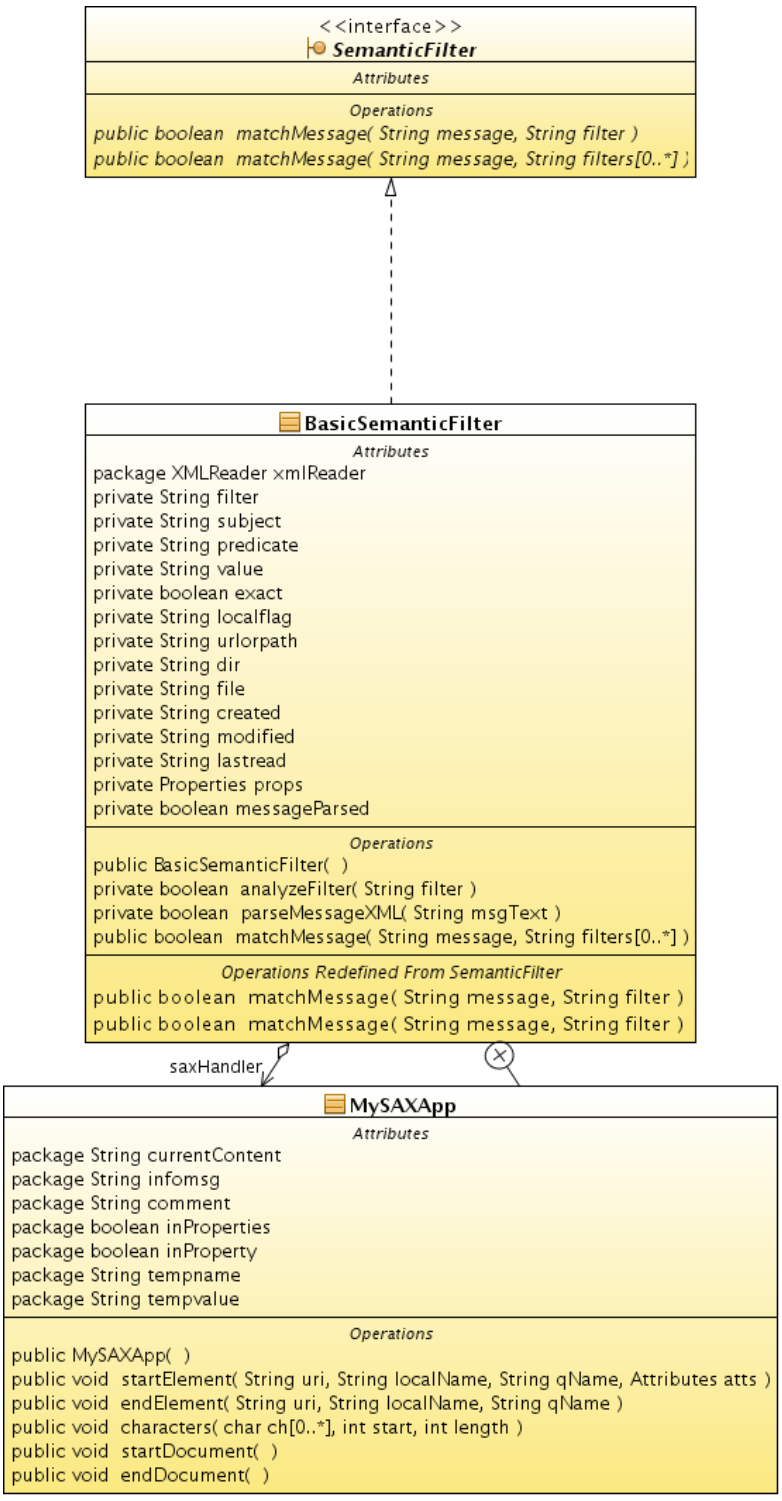

Abbildung 5.12: Die für das Filtern der Nachrichten relevanten Klassen

Dies ist eine Implementierung von `SemanticFilter`, die die bereits weiter oben aufgeführten Filter auf Nachrichten im hier verwendeten Format anwendet (s. Kapitel 5.2). Dazu bedient sich der Filter zunächst seiner eingebetteten Implementierung eines SAX-Handlers. Damit wird einen einkommende Nachricht erst einmal in ihre Elemente zerlegt. Anschließend wird der oder werden die Filter auf die Elemente bzw. Eigenschaften der Nachricht angewendet. Sobald einer der Filter passt, liefert die `matchMessage` Methode `true` zurück. Der `BasicSemanticFilter` wird derzeit von den weiter oben beschriebenen TextListenern verwendet, um eingehende Nachricht aus dem Topic auf ihre Eignung für die eigenen Abonnements zu prüfen. Dazu ein kleines Beispiel: Subscriber A wartet auf Nachrichten, die ihn über seine Subscription auf dem Topic erreichen sollen. A ist natürlich nicht irgendein Subscriber, sondern ein Subscriber in unserem Kontext. Das bedeutet er dient einem Klienten und wurde von ihm mit zwei Filter versorgt, auf die eingehende Nachrichten passen müssen:

1. `modifier is John Smith`
2. `file has blume`

Nun geht bei A folgende Nachricht ein:

```
<message>
  <localflag>y</localflag>
  <urlorpath>/home/peter/blaue_blumen.jpg</urlorpath>
  <dir>/home/peter</dir>
  <file>blaue_blumen.jpg</file>
  <created>20070202</created>
  <modified>20070202</modified>
  <lastread>20080722</lastread>
  <properties>
    <property>
      <name>creator</name>
      <value>Peter Schmitt</value>
    </property>
    <property>
      <name>modifier</name>
      <value>Peter Schmitt</value>
    </property>
    <property>
      <name>about</name>
      <value>Blaue Blumen auf einer Wiese</value>
```

```
    </property>
  </properties>
</message>
```

A's TextListener reagiert auf die eingehende Nachricht und jagt sie durch einen `BasicSemanticFilter` f, dem er auch seine beiden Filterstrings mitgegeben hat. f zerstückelt zunächst die Nachricht und versucht danach die beiden Filter auf die einzelnen Felder anzuwenden. Mit dem ersten Filter hat er keinen Erfolg. Zwar gibt es ein Name-Wert-Paar im Bereich `properties`, das den Namen "modifier" hat, allerdings passt der Wert "Peter Schmitt" nicht zu den Filterangaben "John Smith". Bei der Überprüfung mit dem zweiten Filter wird f jedoch fündig. Das Element `file` enthält tatsächlich den Wert "blumen", wenn auch nur als Teilstring. Da der zweite Filter aber mit `has` formuliert wurde, ist dieses Ergebnis für f korrekt und damit bestätigt er dem Aufrufer, dass die Nachricht auf die abnonnierten Filter passt. Damit bekommt A nun die Bestätigung, dass sein Klient Interesse an der Nachricht hat und nimmt sie in seine Liste der zuzustellenden Nachrichten auf.

Andere Implementierungen von `SemanticFilter` könnten durchaus kompliziertere Algorithmen zur Prüfung der Nachrichten anwenden, insbesondere, wenn aufwändigeren Filter zum Einsatz kämen.

Das Proxymodul, die Kommunikation mit der Außenwelt

Wie beim MsgProducer-Modul, gibt es auch hier die Klassen `ProxyToWorld`, `ProxyToWorldThread` und `ProxyToWorldProtocol`. Die Aufgaben der Klassen sind identisch zu den oben verwendeten, sie unterscheiden sich lediglich etwas in der Implementierung, das das Protokoll leicht variiert und damit andere Methoden angestoßen werden müssen. Daher werden die drei Klassen hier lediglich grafisch dargestellt, mit ihren Attributen und Methoden. Im gleichen Zusammenhang sei noch ein Überbleibsel erwähnt, das noch nicht durch Refactoring entfernt wurde. Die Klasse `MsgInfoServer` dient inzwischen lediglich noch zum Start des `ProxyToWorld`. Da diese Logik aber noch nicht umgezogen wurde, wurde der `MsgInfoServer` noch beibehalten und kann in der jetzigen Form auch noch nicht umgezogen werden.

Zudem gibt es noch die Klasse `MsgTaskPacket`, die, genau wie ihr Gegenstück im `MsgProducer` die Arbeitsanweisungen durch die Queues transportiert. Unterscheiden tun sich die beiden Arten lediglich durch die transportierten Inhalte. Während die Variante im `MsgProducer` die zu veröffentlichenden Nachrichten transportiert, werden die Aufgabenpakete

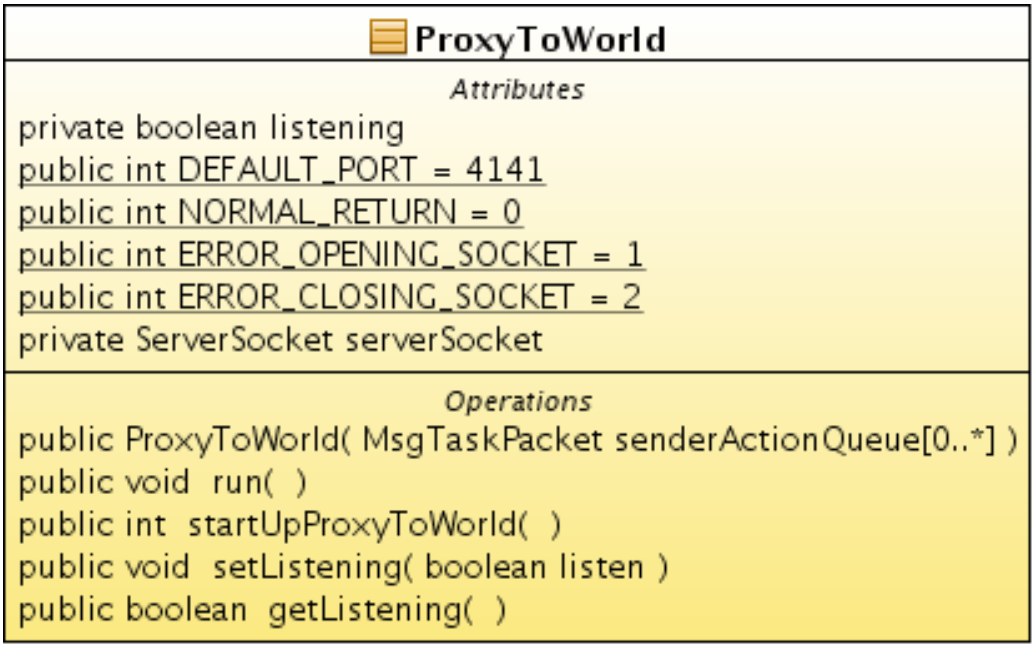

Abbildung 5.13: semom.msgsubscr.ProxyToWorld

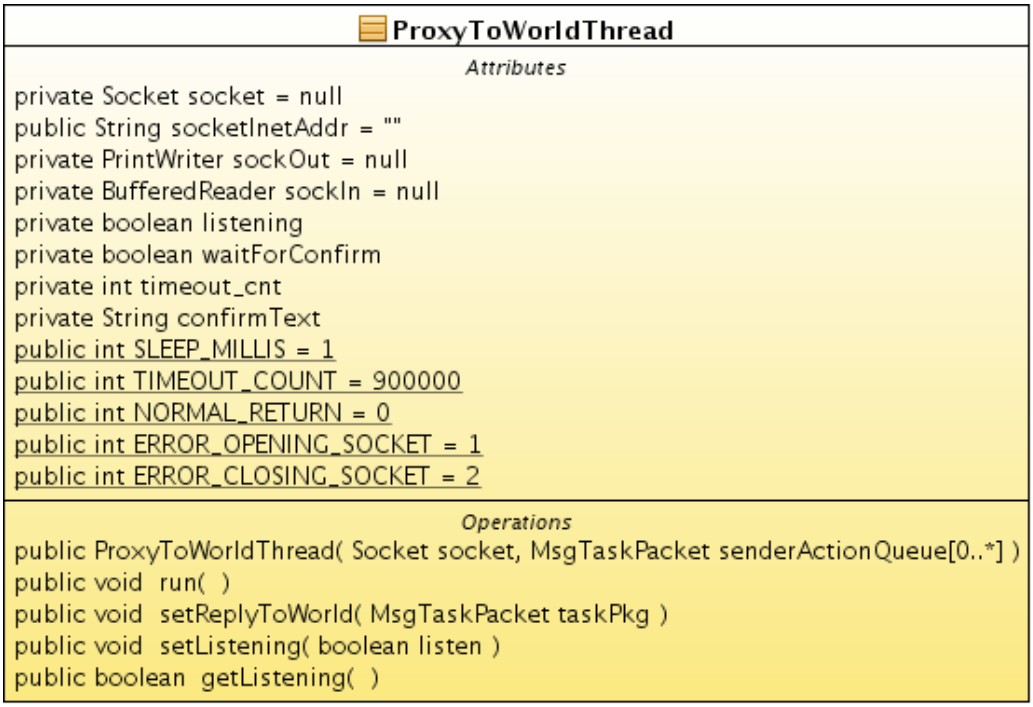

Abbildung 5.14: semom.msgsubscr.ProxyToWorldThread

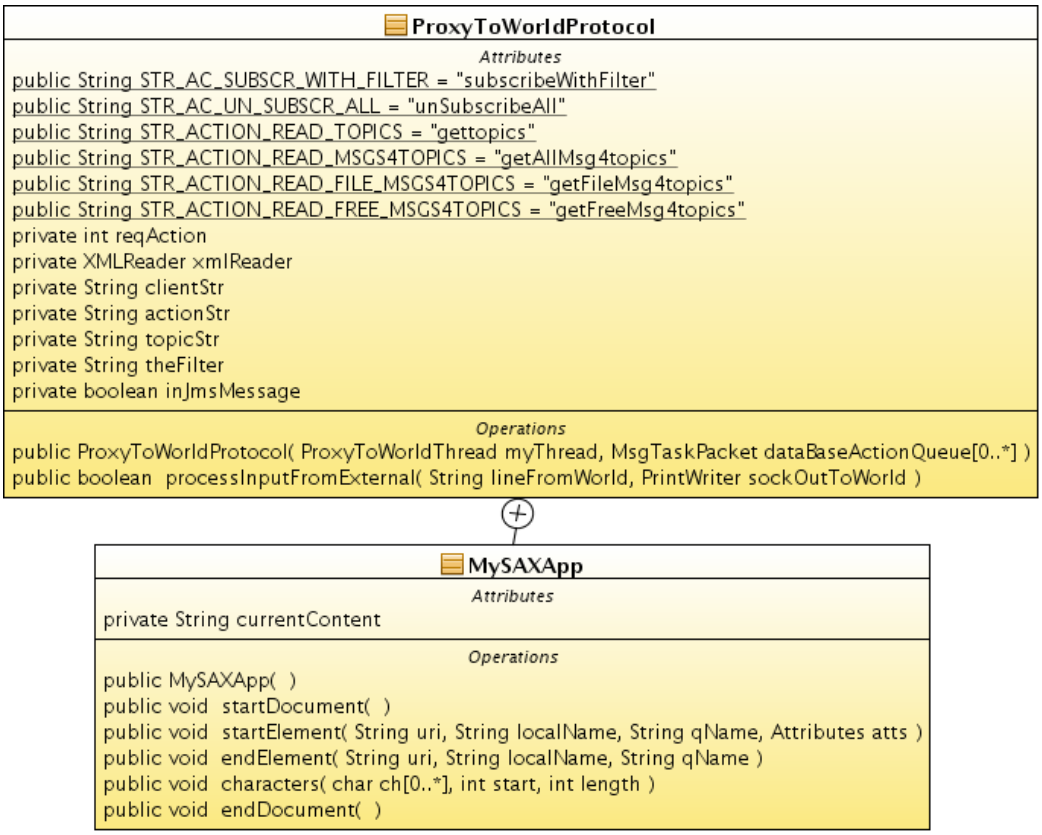

Abbildung 5.15: semom.msgsubscr.ProxyToWorldProtocol

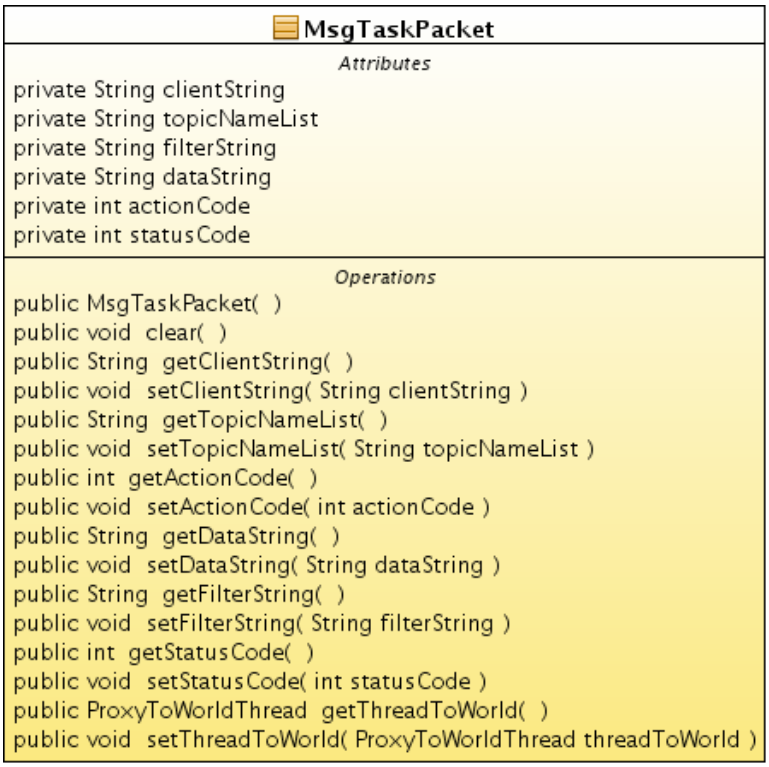

Abbildung 5.16: semom.msgsubscr.MsgTaskPacket

hier verwendet, um die unterschiedlichen Requests mit den zugehörigen Metadaten darzustellen, die der `MsgSubscriptionHandler` bedienen muss.

Ablauf eines Requests im MsgSubscriptionHandler

Da sich der Ablauf vom Start der Anwendung bis zum Bedienen des ersten Requests nicht sehr von dem Ablauf bei `MsgProducer` unterscheidet, soll an dieser Stelle ein Sequenzdiagramm zur Dokumentation ausreichen (Abbildung 5.17). Zudem wird noch ein Aktivitätsdiagramm, Abbildung 5.18 den Ablauf einer Subscription darstellen.

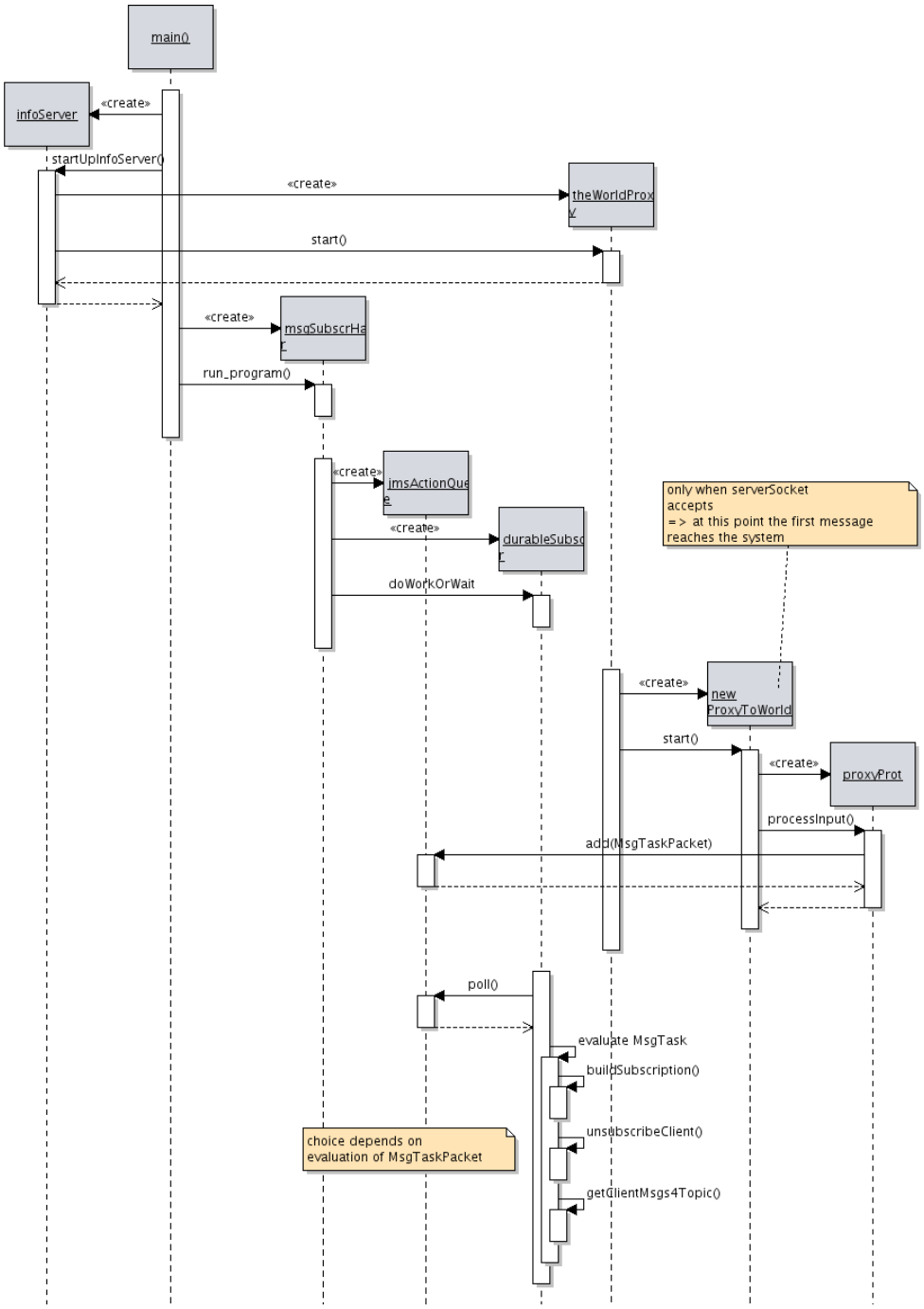

Abbildung 5.17: MsgSubscriptionHandler - vom Start der Anwendung, bis zur Bearbeitung der Requests

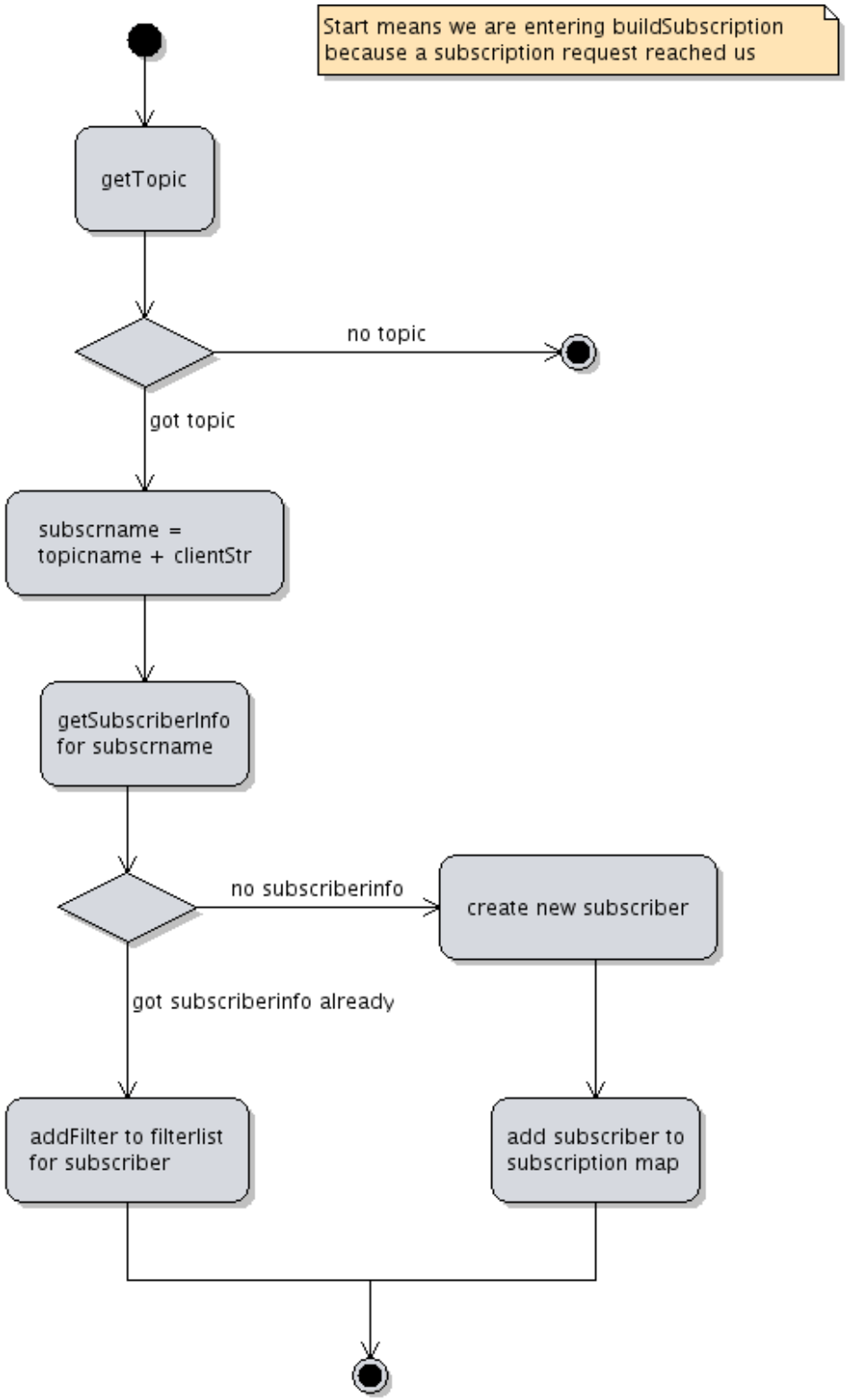

Abbildung 5.18: Verarbeitung eines Subscription-Requests

Kapitel 6

Clientanwendung und Tests

Damit lassen wir die Entwicklung von SeMOM nun hinter uns schauen, welchem konkreten Verwendungszweck man das jetzt zuführen kann.
"HALT!", schreit da der aufmerksame Leser.
"Wo sind den RDF und Konsorten geblieben?"

Das ist an dieser Stelle eine berechtigte Frage, die sich allerdings recht schnell beantworten lässt: "Draußen."
Zur Begründung sei an dieser Stelle erst mal nur gesagt, dass RDF-basierte Systeme und Ontologien nicht aufgenommen wurden, um den Rahmen der Diplomarbeit nicht zu sprengen. Im Schlussteil wird nochmal ausführlicher auf diese Thematik eingegangen werden, gefolgt von einem Ausblick auf das, was aus SeMOM noch möglicherweise machbar wäre.

6.1 Implementierung eines Clients

Bereits weiter oben wurde angeführt, dass SeMOM mit dem Gedanken der Plattform- und Sprachunabhängigkeit im Sinn entwickelt wurde. Es sollte eben kein proprietäres System werden, dass nur mit seinesgleichen oder gar überhaupt nicht kommunizieren kann. Um das zu unterstreichen, sollte eine Testapplikation entwickelt werden, die technologisch möglichst fremd ist. Der erste Gedanke fiel auf eine native Anwendung, beispielsweise in Linux. Da es bei SeMOM hauptsächlich darum geht, bestehende Anwendungen möglichst einfach zu integrieren und nicht ständig neue zu entwickeln, hatte ich mir überlegt, ein Plugin beispielsweise für den Konqueror, einem Dateimanager

und Webbrowser der KDE-Umgebung auf Linux zu entwickeln. Dieser Abschnitt soll einen Eindruck vermitteln, auf was für Schwierigkeiten man stoßen kann, wenn man so etwas zu realisieren versucht. Es wird an dieser Stelle keine genaue technische Beschreibung der erfolgten Entwicklung geben, da das nicht zu dem Fokus dieser Arbeit passen würde. Auf der dieser Arbeit beigefügten CD-ROM befindet sich sämtlicher Sourcecode, aber auch kompilierte Binärdateien, die lediglich entsprechend den Anleitungen im Anhang oder auf der CD-ROM installiert werden müssen.

6.1.1 Aller Anfang...

Die erste Hürde, auf die man stoßen kann und oft auch wird ist, dass KDE nicht gleich KDE ist. Viele der Linux-Distributionen, die sich auch an Endanwender richten, belassen die verwendeten Pakete nicht in ihrer ursprünglichen Form, sondern fügen eigene Inhalte oder Patches dazu, oder tauschen möglicherweise Bibiotheken aus. So musste ich leider feststellen, dass Tool-gestützte KDE-Entwicklung unter K/Ubuntu, einer sehr populären Linux-Distribution, praktisch nicht möglich ist. KDevelop, die Entwicklungsumgebung für KDE und C++ verweigerte oftmals ihren Dienst. Nach einiger Recherche in entsprechenden Foren und Communities und der Vermutung, dass K/Ubuntu Pfade anders setzt, als KDevelop sie erwartet. Nach einem Schwenk auf OpenSUSE 10.3 ließ sich KDevelop aber installieren und weitgehend problemlos verwenden.
Damit stand ich aber schon vor der nächsten Hürde. Ich hatte mich aufgrund der angenommenen Instabilität von KDE4 dazu entschlossen KDE 3.5 einzusetzen, auch in der Hoffnung, dass das deutlich reifere System, KDE 3.5, besser dokumnetiert sei. Dies stellte sich jedoch im Laufe der Arbeit als nicht zutreffend heraus. Während für KDE4 sehr ausführliche Anleitungen existieren, sieht das im Bereich von KDE 3 sehr schlecht aus. Viel Information ist veraltet und teilweise noch für KDE 2 gedacht. Erst der Schwenk auf die Entwicklung einer eigenen kleinen Applikation ("KPart") und die Integration dieses Programms in das Kontextmenü des Konqueror brachten nennenswerte Fortschritte. An dieser Stelle möchte ich die drei Referenzen benennen, die mir bei dieser Entwicklung am meisten weitergeholfen haben. Zum einen ist das ein Artikel von David Faure, "Creating and Using Components (KParts)" [**?**], zum anderen ist das eine Anleitung von Aaron J. Saigo zur Erstellung von Menüeinträgen in KDE [**?**]. Zu guter letzt möchte ich auch die frendlichen Menschen im KDE IRC-Channel benennen, die meist hilfsbereit und immer freundlich auskunft gegeben haben. Als Fazit aus diesem kleinen Ausflug in

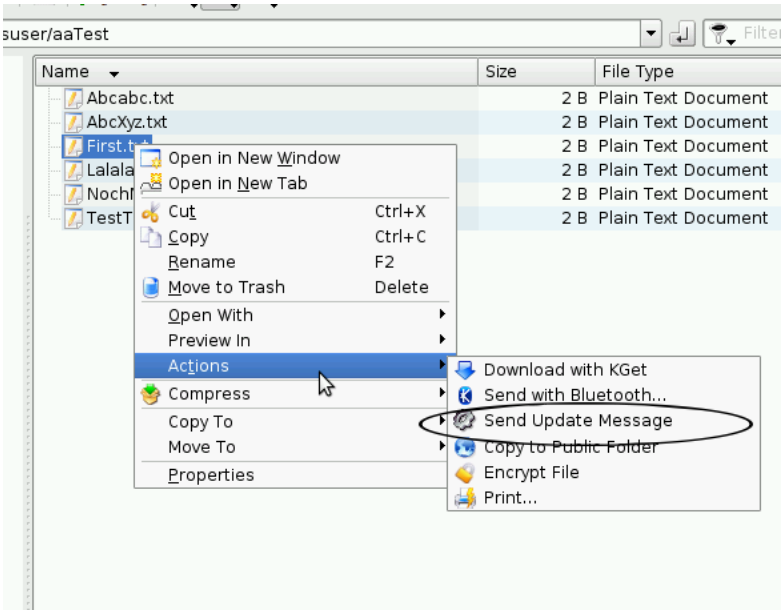

Abbildung 6.1: Aufruf des Testclients über das Kontextmenü einer Datei

die Welt der KDE-Softwareentwicklung kann man eigentlich nur empfehlen Entwicklungen auf KDE 4 zu machen, da hier der Umfang an Dokumentation deutlich überwiegt (vgl. “http://techbase.kde.org/Development/Tutorials”).

6.1.2 Grundlegende Beschreibung und Konfiguration

Zur Verwendung des Clients ist zunächst mal festzuhalten, dass er normalerweise nur aus dem Kontextmenü einer Datei aufgerufen werden kann.

Beim Aufruf präsentiert siche die in Abbildung 6.2 dargestellte Oberfläche. Das große Textfeld oben ist für Ausgaben gedacht, die von SeMOM and den Client geschickt werden. Darunter, linksseitig mit “Filter” beschrieben, ist das Eingabefeld für den Filter, mit dem der Client möglicherweise subscriben möchte. Die linke Schaltfläche darunter, “Subscribe with Filter” würde genau das auslösen und versuchen einen entsprechenden Request an SeMOM zu senden. Daneben die Schaltfläche “Unsubscribe alle” würde veranlassen, dass sämtliche Subscriptions dieses Clients in SeMOM gelöscht würden.
“Nachricht senden” baut aus einigen Metainformationen der Datei (Name, Pfad, Erstellungsdatum etc, s. dazu Kap. 5.2) sowie aus den derzeit drei freien Eingabefeldern, “ifnfo01”, “info02”, “info03” eine Nachricht zusammen und versucht dieses über den MsgProducer an das Topic zu senden. Dabei

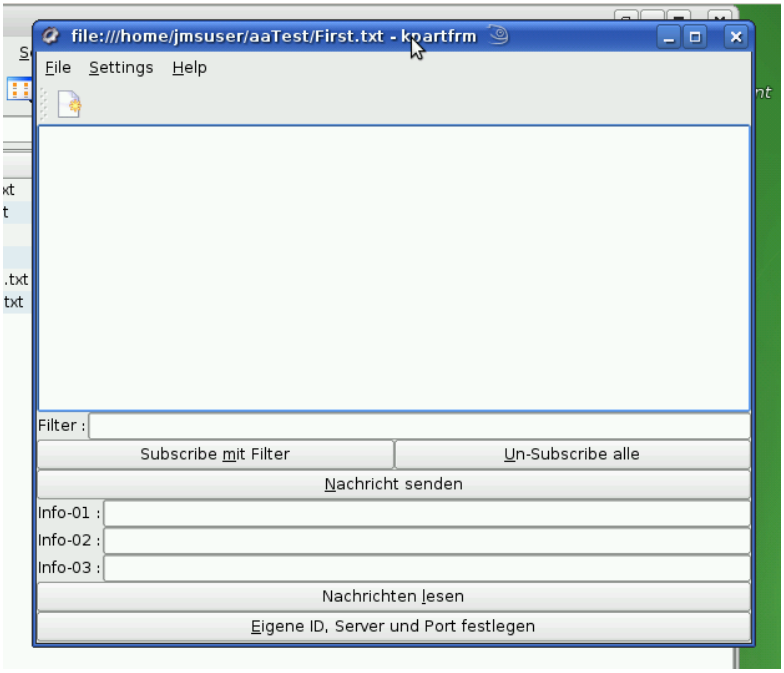

Abbildung 6.2: Hauptansicht des Testclients

werden die “info”-Felder in den properties-Bereich der XML-Nachricht eingebunden. Bevor das alles aber passieren kann, muss erst mit der untersten Schaltfläche “Eigene ID, Server und Port festlegen” der Einstellungsdialog aufgerufen werden.

Zunächst sollte hier ein “Eigener Client-ID String” eingegeben werden. Über diesen String kombiniert mit der IP-Adresse des Aufrufers unterscheidet SeMOM die Subscriptions. Es ist möglich mehrere Instanzen des Clients auf der selben Maschine unter dem gleichen angemeldeten Benutzer laufen zu lassen, allerdings teilen sich die Clients alle die gleiche Konfiguration, so dass sie für SeMOM alle dieselbe Entität representieren Daher ist es für einen Mehrbenutzer-Test ratsam, entweder diesen Test tatsächlich auf mehrere Maschinen, physicalisch oder virtuell, laufen zu lannen. Alternativ man kann auch das Feature “Schneller Benutzerwechsel” nutzen, welches inzwischen auf vielen Desktopumgebungen verfügbar ist. Damit lassen sich mehrere Benutzer quasi gleichzeitig mit einer Maschine testen, ohne dass dieses sich gegenseitig Stören.
Dann müssen noch hostname oder ip und der richtige Port eingetragen werden. InfoServer bezieht sich dabei auf den MsgSubscriptionHandler,

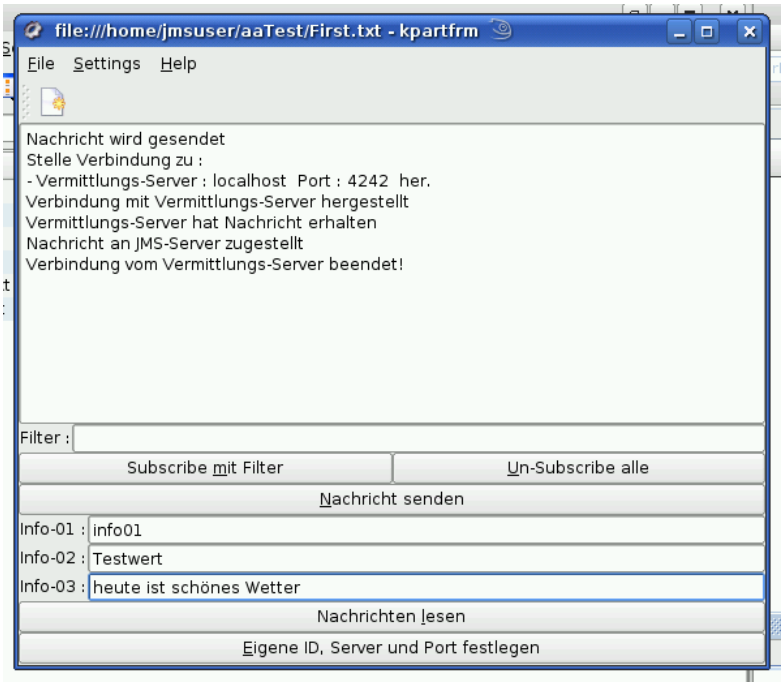

Abbildung 6.3: Hauptansicht des Testclients nach Verschicken einer Nachricht

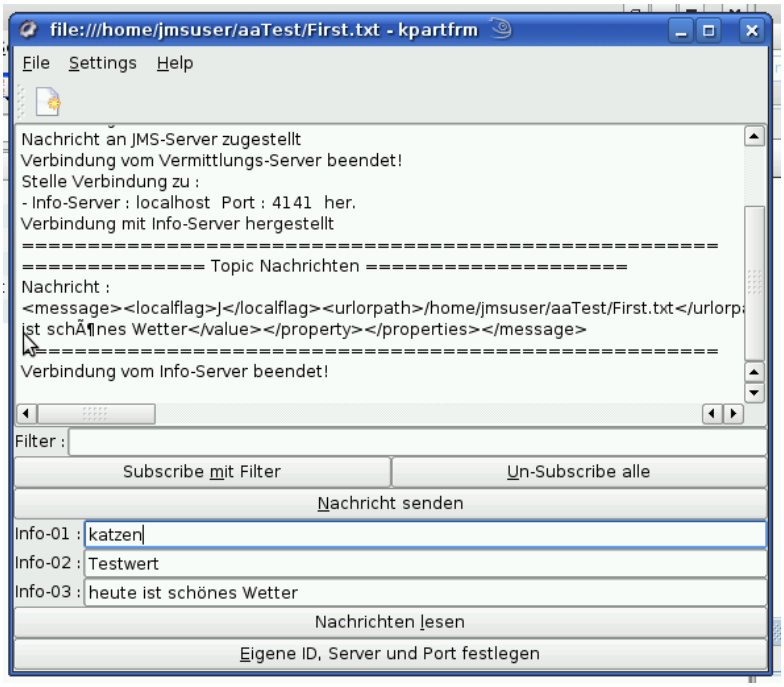

Abbildung 6.4: Hauptansicht des Testclients bei Erhalt einer Nachricht

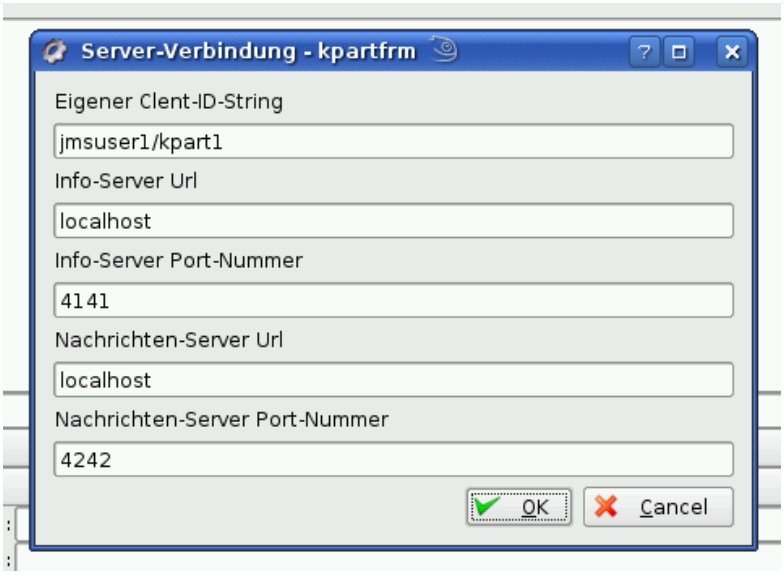

Abbildung 6.5: Der Einstellungsdialog des Testclients

Nachrichtenserver ist der MsgProducer.

Im Auslieferungszustand lässt der Client sich nur über einigen wenigen Textformaten öffnen, dies lässt sich aber leicht in einer Konfigurationsdatei ändern. Der genaue Weg ist im Anhang beschrieben.

6.1.3 Testdurchführung

1. In einem ersten Test der Clientanwendung und von SeMOM wurden mit dem Client verschiedene Subscriptions erstellt, die auf unterschiedliche Filter reagierten. Dann schickte man einfach Messages an das Backend, um zu prüfen, ob die passenden Messages behalten und die unbrauchbaren verworfen werden.

2. In einem weiteren Test wurde in einer Art "Hot-Seat"-Modus der schnelle Benutzerwechsel dazu genutzt, mehrere gleichzeitige Usersessions zu erzeugen. Die einzelnen Benutzer abonnierten unterschiedliche Filter und schickten unterschiedliche Nachrichten an das Backend. Ziel war es, nach dem bereits erfolgreichen ersten Test mit einem Benutzer, das Verhalten des Systems bei der Beanspruchung durch mehrere Benutzer zu beobachten. Auch dieser Test verlief wie vorgesehen, es konnte keine Fehlfunktion festgestellt werden.

3. In einem letzten Test wurde der Client auf zwei verschiedene virtuelle Maschinen verteilt, die sich beide auf dem gleichen SeMOM-Backend anmelden und es nutzen sollten. Auch hier traten keine problematischen Situationen auf.

Kapitel 7

Bewertung und Ausblick

Ziel dieser Diplomarbeit war die Entwicklung einer Middleware, die es Anwendungen ermöglicht, Informationen über ihre Daten über einen Publish/Subscribe Mechanismus auszutauschen. Für die Bewertung der erreichten Ziele werden jetzt die eingangs gestellten Anforderungen herangezogen.

7.1 Allgemeine Beschreibung der Daten

Gleich die erste gestellte Anforderung muss zumindest als nur teilweise erfüllt angesehen werden. Es werden zwar keine proprietären Anwendungsdaten ausgetauscht, allerdings steckte in der Formulierung der Anforderung der Wunsch, nach der Verwendung von semantischen Beschreibungen, die einer Ontologie entsprechen, auf die sich alle Teilnehmer des Netzes geeinigt haben. Das verwendete XML-Protokoll kann dieser Anforderung nur teilweise genügen. Es ist eben genau das, ein relativ starres Protokoll, das auf den Austausch von Metadaten über Dateien spezialisiert ist.
Geschuldet ist dieser Umstand dem Zeitrahmen der Arbeit und der Priorisierung anderer Ziele. Allerdings ließe sich die Middleware in einer möglicherweise zukünftigen Arbeit durchaus um entsprechende Funktionalitäten erweitern. Die Clients müssten ihre Daten entsprechend beschreiben. Eine entsprechende Implementierung des `SemanticFilter` Interfaces würde dann die Filterung anhand eines zum Beispiel in einem RDF-Schema angegebenen Filters auf den entsprechenden Daten vornehmen.

7.2 Plattformunabhängigkeit

Diese Anforderung kann als voll erfüllt gelten. Die Middleware selbst ist zwar in Java geschrieben und bedarf eines JMS-Containers, um publish/subscribe anzubieten, aber die integrierbaren Anwendungen sind nicht auf eine bestimmte Zielplattform oder Technologie beschränkt. Die einzige Einschränkung ist, dass sie sich mit einem TCP-Socket verbinden können.

7.3 Lose Kopplung

Auch diese Anforderung kann als sehr gut erfüllt bewertet werden. Zum einen sind die teilnehmennde Clients lediglich schwach über die Middleware miteinander verbunden. Andererseits ist auch die Verbindung zu Middleware sehr lose, da Clients nicht dauernd mit ihr verbunden sein müssen, um Daten zu empfangen, an denen sie Interesse haben. Das einzige, was einen etwas stärkere Bindung abverlangt, ist das relativ starre Protokoll, aber das wurde bereits weiter oben in Augenschein genommen.

7.4 Integration innerhalb eines kollaborativen Netzes

Wie die Testszenarien gezeigt haben, lässt sich die Middleware ohne weitere Schwierigkeiten innerhalb eines Netzes betreiben. Allerdings sollte die Verwendung derzeit tatsächlich nur innerhalb eines kleinen vertrauenswürdigen Netzes stattfinden. Zum einen darf jeder Teilnehmer ohne weitere Überprüfung seiner Identität beliebige Nachrichten an das Topic schicken. Genauso kann auch jeder Teilnehmer ohne weiteres Nachrichten empfangen, die er möglicherweise nicht empfangen dürfte. Daher wäre der Einsatz von SeMOM in seiner jetzigen Form nur in abgeschlossenen Netzen sinnvoll, in der jeder den anderen kennt und ein hohes Vertrauen untereinander herrscht.

7.5 Ausblick

Mit SeMOM wurde die Basis geschaffen, um darauf aufbauend den Ansatz einer kollaborativen Middleware in Publish/Subscribe Netzen weiter-

zuführen. Die konsequente Einführung semantischer Ansätze wäre hier eine durchaus sinnvolle und wichtige Weiterentwicklung.

Die Entwicklung von Sicherheitsmechanismen wären ebenfalls eine Maßnahme, die zum Wert von SeMOM beitragen würden, da SeMOM derzeit sehr freizügig ist und keinerlei Authentisierung und Authentifizierung stattfindet. Auch das Einbringen einer Verschlüsselungsschicht, wie hier [**?**] könnte das Einsatzspektrum für SeMOM erweitern, da zusammenn mit entsprechenden Authentisierungsmethoden durchaus auch Daten über potentiell unsichere Netze geschickt werden könnten.

Schließlich wäre es noch möglich, das derzeit durch die Clients durchzuführende Abrufen der neuesten Nachrichten durch eine Notifikationsmethode zu ersetzen. Mit DBUS ließe sich möglicherweise eine Notifikationsmethode erschaffen, die die Kopplung zwischen den beteiligten Anwendung nicht zu sehr verstärken würde, da DBUS für mehrere Systeme verfügbar ist und zudem APIs für verschiedene Plattformen zur Verfügung stehen.

Anhang A

Installation und Einrichtung eines Glassfish Application Servers für SeMOM

Beim Applikationsserver müssen einige Einstellungen vorgenommen werden welche nicht zur Laufzeit gemacht werden können, aber zum Funktionieren der Anwendung unbedingt notwendig sind.

1. Erweiterung des Suchpfades um das Verzeichnis `bin`-Verzeichnis des Glassfish.
   Unter Linux müsste dazu beispielsweise im Home-Verzeichnis des Benutzers die Datei `.profile` um folgende Zeile ergänzt werden:
   `export PATH=$PATH:/home/xxuser/glassfish-v2ur2/bin`
   Dabei wäre `/home/xxuser/glassfish-v2ur2/bin` der Pfad zu dem oben angesprochenen Verzeichnis.
   In diesem Verzeichnis befinden sich Skripte zur Verwaltung des Applikationsservers

2. Einrichten der benötigten JMS-Ressourcen
   Bei laufendem glassfish (in einer Konsole
   `asadmin start-domain domain1`
   eingeben) muss auf der Konfigurationshomepage des glassfish (http://localhost:4848) Unter
   Resources / JMS-Resources / JMS-Resources / Connection Factories
   müssen die von den JMS-Client-Programmen verwendeten "Connecti-

on Factoriesßchon vorhanden sein oder erzeugt werden:
`jms/ConnectionFactory` und `jms/DurableConnectionFactory`
!! Wichtig !! falls schon solche "Factories"bestehen ist es unbedingt notwendig das sie (evtl. durch kopieren) unter den oben angegebenen "JNDINamen "jms/....änsprechbar sind, da die Clients diese über JNDI suchen und referenzieren! Falls solche "Factories"noch nicht vorhanden sind, ist es am einfachsten diese beiden Resourcen mit Hilfe der im JavaEE-Tutorial [**?**] beschriebenen, und bei den zugehörigen Examples mitgelieferten ANT-Scripten zu erstelln, Hirzu ist es notwendig ein einfaches producer-Beispiel und ein Beispiel mit "durablesubscriberßu installiern. (Achtung! je nach GlassFish-Subrelease kenn es sein, dass schon eine Äpplication-Client-Resoure! "producerïm AppServer vorhanden ist ich hab die einfach umbenannt/gelöscht um das JMS-producer-Beispiel zu installiern)
Beim Installiern der JMS-Beispiele [**?**] wird schon ein Topic mit dem Namen "jms/Topicängelegt. Das Anlegen von Topics ist sehr AppServer speziefisch und wird von entsprechenden Scripten oder per Hand gemacht. Beim GlassFish ist das Anlegen per Hand sehr einfach.
!! Wichtig ist !! : egal welche Topics man anlegt (auch neue), das diese einen JNDI-Namen haben welcher mit "jms/änfängt! Denn: Die SeMOM-Programme. tauschen Topic-Namen immer ohne diesen "jms/-Prefix aus und stellen diese auch ohne diesen Prefix in dem Client. dar. Bei Kommunikation mit dem JMS(App)-Server wird immer diese Prefix vorangestellt!

Anhang B

Einrichten der JavaEE Applikationen

Client-Programme bzgl. des JMS/App-Servers sind die beiden Java-Progs:
`MsgProducer`
`MsgSubscriptionHandler`,
wobei:
`MsgProducer` Nachrichten-Auftrüge vom KDE-Prog. entgegen nimmt und daraus echte JMS-Nachrichtem im JMS-Server "produziert".

`MsgSubscriptionHandler` Erstellt dauerhafte (durable) SSubscriptions"für die JMS-Topics, nimmt deren Nachrichten entgegen und schreibt in die DB (msgdb).

Wichtig ist das beide Client-Programme zwei Jar-Files in ihrem Distributions-Verzeichnis haben von denen jeweils einer zum AppServer "deployed"werden muss! Die Datei zum deployen ist immmer die Datei mit Namen
`apppnameClient.jar`
also konkret:
`MsgProducerClient.jar` und `MsgSubscriptionHandlerClient.jar`

Die Dateien
`MsgProducer.jar` und `MsgSubscriptionHandler.jar`
sind dann die eigentlichen Client-Programme welche gestartet werden müssen.

!! Achtung !!

Netbeans 6.x erstellt diese `xxxClient.jar` - Files im `dist`-Verzeichnis erst, wenn man das etsprechende Prog. mit "runäufruft! "Build"gefolgt von Ündeploy and Deploy"reicht noch nicht!

Anhang C

Installation des KDE-Parts

Da die KDevelop-IDE kein korrektes "Distributions-Paketërzeugt, muss die Installation der einzelnen KDE-Programm-Komponenten von Hand vorgenommen werden. Unter dem Verzeichnis ÄppDistribäuf der mitgelieferten CD sind Unterverzeichnisse für die einzelnen Programm-Komponenten zur "Distribution"bzw. zum Aufrufen vorhanden. Für die KDE-Komponente ist hier der Verz.-Baum mit der Wurzel "kderootßu finden. Mit "kderootïst das KDE-Home-Verzeichnis gemeint, das bei einer Suse-Linux 10.3 `/opt/kde3` ist.
In diesem Verzeichnisbaum sind die jeweiligen Dateien so verteilt, dass sie eins zu eins in die Verzeichnisse des echten KDE-Baumes kopiert werden können. Die Entwicklungs Projekt-Dateien für die KDE-Komponente liegen unter: `jmsproj/kdepart/kpartfrm`
wobei "kpartfrm"das eigentliche Projekt-Verzeichnis ist.

!! Wichtiger Hinweis !!
Unter dem Verz. `jmsproj/kdepart/kpartfrm/src` sind zwei versteckte Verzeichnisse zu finden (`.libs` und `.deps`) von denen das `.libs` Verzeichnis die Dateien der Linked-Library enthält. Dies ist eventuell wichtig, falls eine Neuübersetzung nötig wird, und die Dateien unter dem Distributions-Verzeichnis "kderootäktualisiert werden müssen.

Die Verzeichnis / Datei - Struktur ist die im Folgenden beschriebene. Dabei wird unter jeder Datei in Klammern ihr Herkunfts-Pfad bzgl. des Entwicklungs-Projekts "kpartfrmängegeben.

```
kderoot/
```

```
bin/
 kpartfrm    -    Executable
   ( /kpartfrm/src/kpartfrm )

lib/kde3/
 libkpartfrmpart.la
   ( kpartfrm/src/libkpartfrmpart.la )
   !!! der Eintrag /kpartfrm/src/.libs/libkpartfrmpart.la ist ein Link
   !!! auf die Datei im "/kpartfrm/src" - /kpartfrm/src - Verzeichnis

 libkpartfrmpart.so
   ( kpartfrm/src/.libs/libkpartfrmpart.so )


share/

/share/applnk/Utilities
 kpartfrm.desktop
   ( kpartfrm/src/kpartfrm.desktop )
   !!! Achtung es gibt eine Datei gleichen Namens aber anderem
   !!! Inhalts und Funktion, welche weiter unten beschrieben wird.
   !!! Diese beiden Dateien müssen immmer "auseinander-gehalten" werden
   !!! Die andere Datei liegt im Prjekt-Verz.: "/kpartfrm/aaextconfig

share/apps/

share/apps/konqueror/servicemenus/
 kpartfrm.desktop
   ( kpartfrm/aaextconfig/kpartfrm.desktop )
   !!! Achtung dies ist die oben beschriebene Datei mit gleichem Namen
   !!! aber anderem Inhalt und Funktion.
 ===> hier können unter
        ServiceTypes=text/plain,text/x-c++hdr,text/x-c++src ... weitere
        Datei-Typen angegeben werden.
 ===> und weiter unten in kpartfrm_part.desktop MimeType=....

share/apps/kpartfrm/
 kpartfrm_shell.rc
   ( kpartfrm/src/kpartfrm_shell.rc )

share/apps/kpartfrmpart/
```

```
 kpartfrm_part.rc
   ( kpartfrm/src/kpartfrm_part.rc )

share/icons/hicolor/16x16/apps/
 kpartfrm.png
   ( /kpartfrm/src/hi16-app-kpartfrm.png )

share/icons/hicolor/32x32/apps/
 kpartfrm.png
   ( /kpartfrm/src/hi32-app-kpartfrm.png )

share/services/
 kpartfrm_part.desktop
   ( kpartfrm/src/kpartfrm_part.desktop )
 ===> hier können unter
       MimeType=text/english;text/plain;text/x-makefile;text/x-c++hdr;.
       Datei-Typen angegeben werden.
 ===>  s.o. kpartfrm.desktop ServiceTypes=...
```

C.1 Erweitern der erlaubten Dateitypen für den KPart

Zur Änderung der Dateitypen, für die das KDE-Part eingesetzt werden kann, müssen also in
`$kderoot/share/apps/konqueror/servicemenus/kpartfrm.desktop`
die entsprechenden Typen bei dem Eintrag `ServiceTypes` ergänzt werden (s.o.). Außerdem können unter
`$kderoot/share/services/kpartfrm_part.desktop`
bei dem Eintrag `Mimetypes` die entsprechenden Typen ergänzt werden.

Anhang D

Einrichten der DerbyDB

Die von den Programmen verwendeten daten sind:
DB-name : msgdb
User : APP
Password : msgdb

Bei der Derby-DB (Apache-Variante der früheren "Java DB") welche u.a. beim GlassFish als auch beim Java-SE-6 mitgeliefert sind folgende Punkte zu beachten:

1. Es kann sein das auf einem PC mehrer (unterschiedliche) Versionen installiert sind.

2. Die verwendete Datenbank sollte nicht von einer Derby-Version erstellt worden sein welche neuer ist als der Derby-DB-Server welcher zur Verarbeitung gestartet wird. Die erstellte DB "msgdb"läuft ab Derby-Server-Vers: 10.2.2.1 (Es gibt mittlerweile aktuellere Derby-Versionen)

3. Sollte aus versehen mit einer älteren Derby-Version auf die DB zugegriffen werden, so sind die Fehlermeldungen nicht sehr aufschlussreich. Bei irgendwelchen Fehlermeldungen sollte man immer in der `derby.log` Datei nachsehen welche in dem DB-Home-Pfad des Derby-Servers liegt. Im oben beschriebenen Fall steht dann so etwas wie: "Datenbank wurde mit neuerem Release erstellt. Formate nicht kompatibelSSollte in dem derby.log - File nichts stehen was auf die gerade laufende Derby-Serve-Instanz hindeutet dann ist Punkt 4) zu beachten.

4. Das Wichtigste bei dem Derby-DB-Server ist, dass er die angeforderten Datenbanken findet! Hier darf man sich NICHT darauf verlassen das z.B. die Derby-Installation im AppServer ihre Datenbanken in dem angegebenen "default-Verzeichnisßucht! Dies kann abhängig durch andere installierte Derby-Installation / IDE's (Netbeans) u.s.w geändert sein. Daher ist es besonders wichtig, egal ob man die Derby-Instanz mittels AppServer-Kommandos oder mit den (von Derby) migelieferten Scripten startet:
   Es sollte immer der `--dbhome` - Parameter angegeben werden!
   also beim Start der AppServer-Instanz:
   `asadmin start-database --dbhome $PfAD_ZUM_VERZEICHNIS_MIT_DEN_DATENBANKEN`
   Der `--dbhome` - Parameter gibt das Verzeichnis an unter welchem der Derby-Server die Datenbanken sucht.

5. Die Java-Programme welche auf einen Derby-Server zugreifen wollen müssen die Derby-Driver-Jar's in ihrem classpath haben. Dies ist für die AppClient-Progs. der Fall da, sie auf die Derby-Installation des AppServers zugreifen. Ansonsten muss man die entsprechenden Driver-Jar-Files in der jeweiligen IDE mit zum Projekt hinzu nehmen. Um Inkompatibilitäten wegen unterschiedlicher Derby-Versionen zu vermeiden wählt man für solche Java-Projekte die Derby-Jar-Files des AppServers, da andere zur Anwendung gehörende ClientProgramme ja gleichzeitig auf die Derby- Instanz des AppServers zugreifen. Diese findet man für GlassFish unter :
   `$GLASSFISH_HOME/javadb/lib`

Anhang E

Start der Anwendungen

1. Start des AppServers:
   In einer Konsole folgendes eingeben:
   `asadmin start-domain --verbose domain1` Danach ist die Admin-Konsole über
   `http://localhost:4848/asadmin/` erreichbar.
2. Stoppen des AppServers
   In einer anderer Konsole eingeben
   `asadmin stop-domain domain1`
3. Start der AppServer Derby-Server-Instanz
   In einer Konsole eingeben:
   `asadmin start-database --dbhome $PfAD_ZUM_VERZEICHNIS_MIT_DEN_DATENBANKEN`
   DerbyDB wieder stoppen mit
   `asadmin stop-database`
4. Start von "MsgProducer"
   In einer Konsole eingeben:
   `cd $PFAD_ZU/MsgProducer/dist`
   `appclient -client MsgProducer.jar`
5. Start von "MsgSubscriptionHandler"
   In einer Konsole eingeben:
   `cd $PFAD_ZU/MsgSubscriptionHandler/dist`
   `appclient -client MsgSubscriptionHandler.jar`
6. Danach kann wie in Kapitel 6 beschrieben, mit dem KDE-Client auf SeMOM zugegriffen werden.

Anhang F

Verzeichnisstruktur auf der CD-ROM

```
AppDistrib/
  Die binär / jar und config - Dateien der Anwendung,

AppDistrib/derbydbs/
  Hier liegt die Nachrichten-Datenbank "msgdb"

AppDistrib/kderoot/
  Die KDE-Komponente, siehe auch Kapitel \ref{ch:kdepart}.

MsgSubscriptionHandler/dist
  Die Subscription-Handler jars
  Holt Nachrichtem beim AppServer ab und schreibt sie in die Datenbank.

AppDistrib/MsgProducer/dist
  Die Message-Producer jars
  Nimmte Nachrichten-Aufträge vom KDE-Prog. entgegen und erzeugt
  JMS-Nachrichten in den geforderten Topics.
```

```
docu4jmsproj/
  Dokumentations-Verzeichnis (enthält diese Datei)

jmsproj/
  Das "Hauptverzeichnis" für alle Entwicklungs-Projekte mit den
  Unterverzeichnissen "javapart" und "kdepart". In diesen UnterVerz.
  befinden sich dann die einzelnen Projekt-Verzeichnisse wie sie
  in der Netbeans 6.5 bzw. Kdevelop-IDE zu finden sind.
  Genauer:

jmsproj/kdepart/kpartfrm
    Das Projekt-Verzeichnis für die KDE-Komponente der Anwendung.

jmsproj/javapart
  Hier liegen die zwei Java-Projekte

jmsproj/javapart/MsgSubscriptionHandler
  Der Subscription-Handler holt Nachrichtem beim AppServer ab und schreibt s

jmsproj/javapart/MsgProducer
  Der Nachrichten-Produzent von Nachrichten im JMS-Topics.
  Nimmte Nachrichten-Aufträge vom KDE-Prog. entgegen und erzeugt JMS-Nachric
```

Literaturverzeichnis

[BL01] Tim Berners-Lee. Conceptual graphs and the semantic web - reflections on web architecture. 2001.

[CFT08] Kendall Grant Clark, Lee Feigenbaum, and Elias Torres. SPARQL Protocol for RDF. W3c recommendation, W3C, January 2008.

[CIKN04] Paul-Alexandru Chirita, Stratos Idreos, Manolis Koubarakis, and Wolfgang Nejdl. Publish/subscribe for rdf-based p2p networks. In Christoph Bussler, John Davies, Dieter Fensel, and Rudi Studer, editors, *ESWS*, volume 3053 of *Lecture Notes in Computer Science*, pages 182–197. Springer, 2004.

[DF04] Stefan Decker and Martin R. Frank. The networked semantic desktop. In *Proc. WWW Workshop on Application Design, Development and Implementation Issues in the Semantic Web*, Net York City, NY, May 2004.

[Div] Diverse. Reliability engineering.

[Div08] Diverse. The java ee tutorial, 2008. http://java.sun.com/javaee/5/docs/tutorial/doc/ besucht Nov. 2008.

[Fau] David Faure. Creating and using components (kparts). http://developer.kde.org/documentation/tutorials/kparts/ besucht Oktober 08.

[Fla02] David Flanagan. *Java in a Nutshell. Deutsche Ausgabe der 4. A.* O'Reilly, 4. a. edition, 2002.

[FSA07] Thomas Franz, Steffen Staab, and Richard Arndt. The x-cosim integration framework for a seamless semantic desktop. In Derek H. Sleeman and Ken Barker, editors, *K-CAP*, pages 143–150. ACM, 2007.

[HBSF01] Mark Hapner, Rich Burridge, Rahul Sharma, and Joseph Fialli. Java message service — version 1.0.2b. Technical report, Sun Microsystems, August 2001.

[HKRS08] Pascal Hitzler, Markus Krötzsch, Sebastian Rudolph, and York Sure. *Semantic Web — Grundlagen*. eXamen.press. Springer, Berlin–Heidelberg, Germany, 2008. In German.

[Ian98] Renato Iannella. A complete idiot's guide to the resource description framework. *The New Review of Information Networking*, 4, 1998. besucht Juni 2008.

[LJ04] Han Li and Guofei Jiang. Semantic message oriented middleware for publish/subscribe networks. In *proc of SPIE*, volume 5403, pages 124–133, 2004.

[MM04] Frank Manola and Eric Miller. Rdf primer. World Wide Web Consortium, Recommendation REC-rdf-primer-20040210, February 2004.

[Sai] Aaron J. Saigo. Creating konqueror service menus. http://developer.kde.org/document
besucht Oktober 2008.

[Wik] Single point of failure.