Master Thesis

Extending OSGi by Means of Asynchronous Messaging

Marc Schaaf

September 2009

Fachhochschule Hannover
University of Applied Sciences and Arts
Faculty IV – Department of Computer Science
Participants

Author
B.Sc. Marc Schaaf
Haupstraße 123
30974 Wennigsen Deister
Mat.Nr.: 1060767
E-Mail: marc@marc-schaaf.de

First Examiner
Prof. Dr.-Ing. Arne Koschel
Fachhochschule Hannover
Ricklinger Stadtweg 120
30459 Hannover
E-Mail: arne.koschel@fh-hannover.de

Second Examiner
Prof. Dr. Volker Ahlers
Fachhochschule Hannover
Ricklinger Stadtweg 120
30459 Hannover
E-Mail: volker.ahlers@fh-hannover.de

In Cooperation With
Progress Software Corporation
Dublin, Ireland

ProSyst Software GmbH
Cologne, Germany
Declaration of Authorship

I hereby declare that the whole of this master thesis is my own work, except where explicitly stated otherwise in the text or in the bibliography.

Hannover the 16. September 2009

__________________________
Marc Schaaf
Acknowledgments

At this point I would like to thank Progress Software and ProSyst Software and their employees for their support and the opportunity to realize this thesis in cooperation with them. Especially I would like to thank David Bosschaert, Roman Roelofsen and Roland Tritsch for the very good cooperation, their support, the countless constructive discussions and their commitment.

Furthermore I would like to thank Volker Ahlers and Arne Koschel for their very good supervision and in particular Arne Koschel for his commitment during the establishment of the cooperation with the two companies.

Finally I would like to thank my family and my girlfriend for their patience and their support through the last six months.
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5.2 Configuration</td>
<td>28</td>
</tr>
<tr>
<td>3.5.3 Discovery</td>
<td>28</td>
</tr>
<tr>
<td>3.6 OSGi Specific Concepts and Limitations</td>
<td>29</td>
</tr>
<tr>
<td>3.7 Definition and Analysis of Possible Integration Approaches</td>
<td>29</td>
</tr>
<tr>
<td>3.7.1 Messaging Service Approach</td>
<td>30</td>
</tr>
<tr>
<td>3.7.2 Distributed OSGi with the Apache CXF - JMS Transport</td>
<td>31</td>
</tr>
<tr>
<td>3.7.3 Asynchronous Callbacks</td>
<td>32</td>
</tr>
<tr>
<td>3.7.4 Event Admin Service Based Approach</td>
<td>33</td>
</tr>
<tr>
<td>3.7.4.1 The Mediation Component as a Client to the Event Admin</td>
<td>34</td>
</tr>
<tr>
<td>3.7.4.2 The Mediation Component as Event Admin Replacement</td>
<td>34</td>
</tr>
<tr>
<td>3.7.4.3 Conclusion of the Event Admin Approaches</td>
<td>35</td>
</tr>
<tr>
<td>3.7.4.4.1 The Mediation Component as a Client to the Event Admin</td>
<td></td>
</tr>
<tr>
<td>3.7.4.4.2 The Mediation Component as Event Admin Replacement</td>
<td></td>
</tr>
<tr>
<td>3.8 Selection of an Integration Approach</td>
<td>36</td>
</tr>
<tr>
<td>4 Key Concepts</td>
<td>37</td>
</tr>
<tr>
<td>4.1 Realization Options</td>
<td>37</td>
</tr>
<tr>
<td>4.1.1 Basic Functionality</td>
<td>38</td>
</tr>
<tr>
<td>4.1.1.1 Event Producing</td>
<td>38</td>
</tr>
<tr>
<td>4.1.1.2 Event Consuming</td>
<td>38</td>
</tr>
<tr>
<td>4.1.2 Reliability Aspects</td>
<td>39</td>
</tr>
<tr>
<td>4.1.2.1 Guaranteed Delivery</td>
<td>39</td>
</tr>
<tr>
<td>4.1.2.2 Durable Subscriptions</td>
<td>42</td>
</tr>
<tr>
<td>4.1.2.3 Handling Messages for Multiple Receivers</td>
<td>43</td>
</tr>
<tr>
<td>4.1.3 Configuration</td>
<td>45</td>
</tr>
<tr>
<td>4.1.3.1 Configuration Files</td>
<td>45</td>
</tr>
<tr>
<td>4.1.3.2 EDS Configuration Service</td>
<td>45</td>
</tr>
<tr>
<td>4.1.3.3 Configuration Data from the Event Handler and the Event</td>
<td>46</td>
</tr>
<tr>
<td>4.1.3.4 The OSGi Configuration Admin</td>
<td>47</td>
</tr>
<tr>
<td>4.1.3.5 Conclusion on Configuration Mechanisms</td>
<td>49</td>
</tr>
<tr>
<td>4.1.3.6 Configuration Parameters</td>
<td>49</td>
</tr>
<tr>
<td>4.1.3.7 Alternative Configuration Possibilities</td>
<td>51</td>
</tr>
<tr>
<td>4.1.3.8 Discovery</td>
<td>51</td>
</tr>
<tr>
<td>4.1.4 Event Message Conversion</td>
<td>53</td>
</tr>
<tr>
<td>4.1.4.1 A Suitable Message Type</td>
<td>53</td>
</tr>
<tr>
<td>4.1.4.2 Messaging Related Information</td>
<td>53</td>
</tr>
<tr>
<td>4.1.4.3 Serialization and OSGi</td>
<td>54</td>
</tr>
<tr>
<td>4.2 Resulting Concept</td>
<td>55</td>
</tr>
<tr>
<td>4.2.1 Evaluation</td>
<td>56</td>
</tr>
<tr>
<td>4.2.1.1 Basic Functionality - R1</td>
<td>56</td>
</tr>
<tr>
<td>4.2.1.2 Quality of Service - R2</td>
<td>56</td>
</tr>
<tr>
<td>4.2.1.3 Fit into the OSGi Framework - R3</td>
<td>56</td>
</tr>
<tr>
<td>4.2.1.4 Portability and Interoperability - R4</td>
<td>58</td>
</tr>
<tr>
<td>4.2.1.5 Transparent Usage - R5</td>
<td>58</td>
</tr>
<tr>
<td>4.2.1.6 Discovery - R6</td>
<td>58</td>
</tr>
<tr>
<td>4.2.2 Conclusion</td>
<td>58</td>
</tr>
<tr>
<td>5 Realization</td>
<td>59</td>
</tr>
<tr>
<td>5.1 Technology and Design Decisions</td>
<td>60</td>
</tr>
<tr>
<td>5.1.1 Additional Requirements</td>
<td>60</td>
</tr>
<tr>
<td>5.1.2 Bundles</td>
<td>60</td>
</tr>
<tr>
<td>5.1.3 Event Message Conversion</td>
<td>61</td>
</tr>
<tr>
<td>5.1.4 Message Format</td>
<td>62</td>
</tr>
<tr>
<td>5.2 Implementation</td>
<td>62</td>
</tr>
</tbody>
</table>
## Contents

5.2.1 EDS .................................................. 62  
5.2.2 STOMP Wrapper ..................................... 65  
  5.2.2.1 STOMP Client .................................... 65  
5.2.3 ActiveMQ Wrapper .................................... 65  
5.2.4 Configuration File Scanner .............................. 66  
5.3 Conclusion ............................................... 66  

6 Demo Scenario ............................................. 67  
6.1 The Flight Information System ............................. 67  
6.2 Realization .............................................. 68  
  6.2.1 Flight Information Manager ........................... 69  
  6.2.2 Airline Web Interface ................................ 70  
  6.2.3 Status Display ....................................... 71  
  6.2.4 The Mobile Client .................................... 71  
    6.2.4.1 Retroactive Consumer ............................ 73  
6.3 Reliability ............................................... 74  
6.4 Conclusion ............................................... 75  

7 Related Work ............................................. 77  
7.1 Eclipse Communication Framework .......................... 77  
  7.1.1 ECF’s Distributed Event Admin Service ................ 77  
7.2 Spring JMS ............................................... 78  
7.3 R-OSGi .................................................. 79  
7.4 Apache Project Proposal “Aries” ........................... 79  
7.5 Proxy Based Mobile Messaging .............................. 79  

8 Evaluation .................................................. 81  

9 Conclusion .................................................. 83  
9.1 Open Points and Outlook .................................. 84  

A Commonly used OSGi Patterns ............................... 85  
A.1 The Whiteboard Pattern .................................. 85  
A.2 Extender Pattern ........................................ 86  

B A Messaging Protocol for JavaME ............................ 87  
B.1 Protocols ............................................... 87  
  B.1.1 Selection of a Protocol ............................... 88  
B.2 The STOMP Implementation ............................... 88  

C Build and Deployment ...................................... 91  
C.1 Exemplary Implementation ................................ 91  
  C.1.1 Build ............................................... 91  
  C.1.2 Deployment ......................................... 91  
C.2 Demo Scenario ............................................ 92  
  C.2.1 Build ............................................... 92  
    C.2.1.1 Flight Information System, Airline Interface and Flight Status Display ................ 92  
    C.2.1.2 Mobile Client .................................... 92  
  C.2.2 Deployment ......................................... 93  
    C.2.2.1 Flight Information System ....................... 93  
    C.2.2.2 Airline Interface ................................ 93  
    C.2.2.3 Flight Status Display ............................ 94
List of Figures

1.1 The basic components of the demo scenario and their relations 3
2.1 The life cycle of an OSGi bundle. 8
2.2 Diagram notation for OSGi Services 10
2.3 The message oriented middleware 12
2.4 Messaging systems support two general channel types. 13
3.1 The general components and their interaction 18
3.2 Message Acknowledgments 22
3.3 The basic functionality of the OSGi Event Admin service 25
3.4 Distributed OSGi: Basic steps to export and import a service 28
3.5 The messaging service approach 30
3.6 The mediation component as a client to the Event Admin 34
3.7 The mediation component as an Event Admin replacement 36
4.1 Asynchronous event delivery 40
4.2 Asynchronous event reception 41
4.3 The basic functionality of the OSGi Configuration Admin 47
4.4 A possible integration of discovery 53
4.5 Abstract Architecture of the EDS 57
5.1 The general design of the EDS exemplary implementation 61
5.2 Simplified class diagram of the implementation of the EDS 63
6.1 The basic subsystems in the demo scenario and their relations 68
6.2 Parts of the Demo with their components. 69
6.3 The web interface of the airline’s management application 71
6.4 The Flight status display. 72
6.5 The deployed mobile client 73
7.1 The distributed Event Admin provided by ECF 78
A.1 Components of the Whiteboard Pattern 86

List of Tables

3.1 General component definitions 17
3.2 Evaluation of the messaging service approach. 31
3.3 Evaluation of the distributed OSGi approach 32
3.4 Evaluation of the asynchronous callback approach 33
3.5 Evaluation of the Event Admin approach 35
3.6 Results from the evaluations of the different integration approaches 36
4.1 The different message types supported by JMS 54
C.1 Automatically created configuration files for the deployment of the EDS. 92
List of Listings

2.1 A simple bundle manifest ........................................ 9
2.2 Registration and look up of an OSGi Service ..................... 10
2.3 Basic functions of the JMS API .................................... 14
2.4 The JMS MessageListener Interface ................................. 15
3.1 Registration of an EventHandler service ............................ 26
3.2 The OSGi EventHandler interface .................................. 26
3.3 The OSGi EventAdmin interface .................................... 26
4.1 Definition of a blocking fault aware postEvent method ............ 41
4.2 Register an event handler and sign up for a durable subscription ... 42
4.3 Revoke a durable subscription by changing the service registration ... 43
4.4 The ManagedServiceFactory interface for the Configuration Admin .... 48
4.5 An exemplary configuration block for the EDS ..................... 50
4.6 Exemplary configuration properties to trigger a discovery .......... 52
5.1 The syntax of the text based message format in EBNF ............... 62
5.2 The MessagingService interface defined by the EDS implementation ... 64
5.3 The Connection interface defined by the EDS implementation ........ 64
6.1 Log output of the EDS when the event delivery fails ................. 75
CHAPTER 1

Introduction

1.1 Motivation and Aim

The OSGi framework is a dynamic Java component framework which is currently emerging in the enterprise market. It is already used as the foundation for many application servers like IBM Websphere or Oracle WebLogic (former BEA WebLogic) [24]. However OSGi has its roots in the area of embedded and limited devices [4]. Consequently it is missing much of the functionality needed in enterprise environments like for example transactions, remote services or asynchronous messaging. To fill those gaps the OSGi Alliance formed a special Enterprise Expert Group (EEG) which aims to extend OSGi with the required functionality to successfully fulfill its new duties in the enterprise environment [32].

The EEG recently succeeded in the integration of service based remoting into the OSGi platform [33]. Starting with the 4.2 specification, OSGi provides a standardized way to call external services like for example Web-Services. In addition it defines how internal OSGi services can be exposed to other applications. Besides those new remoting capabilities other enterprise features like transactions or database access are introduced into the standard\(^1\). One enterprise concept that hasn’t found its way into the OSGi specification yet is the asynchronous message based remote communication via enterprise messaging systems.

Enterprise messaging is a well known concept which is directly usable from within Java in a standardized manner due to the Java Messaging System (JMS). However many enterprise grade systems like for example Enterprise Java Beans with their Message Driven Beans provide an integration based on their own concepts [47, Page 636]. Therefore an integration of messaging into OSGi which reuses OSGi’s concepts and thereby eases the usage of a messaging system for the OSGi developer would be desirable. Consequently the aim of this thesis is to find and analyze possible approaches for the integration of asynchronous messaging into OSGi.

The found approaches are evaluated and a promising approach is selected for further examination. Based on the results a concept for a realization of the approach is defined

---

\(^{1}\) Other new features can be found in [37].
and evaluated to further investigate its capabilities. Based on this concept an exemplary implementation is created and evaluated in the scope of a demo scenario. The demo scenario is shortly outlined in section 1.2.

This thesis was done in cooperation with Progress Software and ProSyst Software. Due to this cooperation some constraints regarding the selection and conception of an integration approach are given:

- The messaging integration needs to be suitable for mobile environments. Therefore the extension has to be usable in a Java ME CDC/FP environment.

- The messaging integration should be usable with Apache ActiveMQ as it is the basis for the FUSE Message Broker from Progress Software.

Those constraints especially influence the exemplary realization and the demo scenario. Both will cover the usage on mobile and limited devices and the communication with Apache ActiveMQ.

1.1.1 Exclusions

The analysis and conception of the integration approach is focused on the core functionality needed as a complete solution is not suitable for the scope of such a thesis. Therefore this thesis leaves out special quality of service aspects like for example security or performance aspects or a detailed coverage of discovery mechanisms.

Furthermore the handling of domain specific problems like for example partial connectivity of mobile devices will not be part of this work. However as the exemplary implementation is intended to be deployable on embedded devices, limitations like older Java versions resulting from this environment will be taken into account for the exemplary implementation.

Further information on the covered and excluded functionality can be found in the analysis chapter in section 3.2 where the requirements for the integration approach are defined.

1.2 Scenario “Mobile Flight Information”

The aim of the demo scenario is to inform flight passengers about the status of their flights at an airport (Figure 1.1). A central management application will handle flight information which can be changed by airlines. Therefore airlines can for example change the scheduling of flights.

The flight information is distributed to the passengers by the central management component via the usual flight information displays which are spread all over the airport. As an addition to this, the changes are also distributed to client applications, that the customers can install on their mobile phones. These applications than provide the passenger with the current status of his flight wherever he is on the airport. Therefore the customer is directly notified about changes of his flight.
1.3 Related Work

A more detailed description of the scenario, the components and their purpose is given in chapter 6. However the communication between the components will be based on messaging. The components around the central management system will be based on OSGi and therefore show how the newly created OSGi messaging integration can be used in conjunction with a foreign system which is posed by the central component.

![The basic components of the demo scenario and their relations](image)

**Figure 1.1:** The basic components of the demo scenario and their relations

1.3 Related Work

This thesis is related to the recent integration of synchronous service based remote communication into the OSGi framework by the Enterprise Expert Group. A description of this *Distributed OSGi* standard can be found as part of the analysis in chapter 3 section 3.5.

Besides this part of the new OSGi standard, some other related projects and publications are discussed in chapter 7 - *Related Work*. One of the in this chapter presented projects is the *Eclipse Communication Framework* (ECF). ECF features an implementation of some parts of the concept developed in this thesis. However the ECF implementation has a different aim than the concept and implementation developed in this thesis. Further discussion based on the results of this thesis can be found in chapter 7.

1.4 Structure of this Document

This thesis is structured as follows:

- Chapter 1 *Introduction* outlines the motivation and the aim of this thesis. It also gives a first glimpse on the demo scenario and some related work.

- Chapter 2 *Fundamental Technologies* will give a short introduction into the main technologies used in this thesis as a preparation for the following chapters.

- Chapter 3 *Analysis* discusses the requirements for the integration of messaging into OSGi and analyzes existing possibilities for OSGi internal and external
communication. Furthermore possible integration approaches are presented and evaluated. One of the approaches will be selected for further research.

- Chapter 4 *Key Concepts* discusses possible concepts for the realization of the selected approach from the previous chapter. Concrete realization options are deducted and an abstract architecture for a messaging extension will be defined.

- Chapter 5 *Realization* describes an exemplary realization of the concept produced in the previous chapter. The chapter defines what parts of the concept are realized and which are omitted. It will conclude with a short evaluation of the implementation to point out its capabilities and limits.

- Chapter 6 *Demo Scenario* describes how the component which was created in chapter 5 can be used in the outlined demo scenario. The scenario will show how messaging is possible with the new extension and how it integrates with foreign messaging components.

- Chapter 7 *Related Work* will describe some related projects with equal or similar goals as this thesis and compare them to the approaches discussed in this work.

- Chapter 8 *Evaluation* shortly summarizes the evaluations of the achievements in the different chapters.

- Chapter 9 *Conclusion* reflects the cognition’s of the whole thesis and summarizes what was achieved, what was left open and gives a quick outlook on possible future developments.

1.5 Notation

This document uses a common OSGi notation for diagrams which describe the usage and relationship of OSGi services and bundles. The notation is explained in figure 2.2 during the OSGi introduction in section 2.2 in the following chapter. An extended explanation can be found in [23]. In addition to the OSGi notation some diagrams will use UML notation like for example sequence diagrams.
CHAPTER 2

Fundamental Technologies

To provide the foundation for the discussions in the next chapters, this chapter gives a brief introduction to the central technologies used in this thesis. The explanations will be extended in the following chapters whenever more detailed information is required.

The following section will give a short introduction into Java and JavaME. This is followed by a more detailed description of OSGi, its basic concepts and components. The chapter concludes with an introduction into the messaging paradigm and a short description of the Java Messaging Service.

2.1 Java

Java is an object oriented programming language initially designed by SUN Microsystems. Java source code is compiled into Java bytecode and executed in a Java virtual machine (JVM or Java VM). The JVM provides a uniform runtime environment on all supported platforms. Therefore a pure Java program which was developed under Linux is usable on other platforms like for example Microsoft Windows. The Java VM is normally shipped together with a Java class library which provides a huge foundation of predefined classes.

One of the two major technologies used in this thesis is the OSGi platform. OSGi extends Java by a module concept and thereby changes some parts of Java’s default behavior. Therefore the remainder of this paragraph gives a quick introduction into the default Java class loading which builds the foundation for the description of OSGi’s concepts in the next section.

Java uses class loaders to load the bytecode of Java classes into the Java VM. The class loader loads the class definitions from class files which are located in its class path\(^1\). Java supports two different types of class loaders: The bootstrap class loader and the user defined class loaders. The bootstrap loader is created by the Java VM to load the initial Java classes when the runtime starts. User defined class loaders can be created at runtime. They are for example used to load classes from locations which are not covered by the class path of the bootstrap loader. These custom class loaders only try to load a class from their own class path if their parent class loader fails to load it

\(^{1}\) The class path is a simple list of directories and Java libraries.
first. Therefore all classes provided by the parent class loader override possible other
definitions of the same class in this class loader. In most cases the parent class loader is
the bootstrap class loader but it can also be a custom class loader.

This default structure of class loaders has some limitations which will be explained
in the following OSGi section together with OSGi’s class loading approach. Further
information on the Java VM and Java class loading can be found in the specification of
the Java virtual machine [25].

2.1.1 JavaME

The Java Mobile Edition (JavaME) is a special version of a Java runtime environment
with a limited class library and minimal changes to the JVM to make it more suitable
for environments with limited resources. Therefore the main purpose of JavaME is to
provide a standardized Java runtime on for example mobile devices.

The mobile runtime environment can be adapted to special requirements by several
so called profiles and configurations. A configuration specifies which language features,
Java virtual machine features, Java libraries and API’s are supported. A profile extends
the class library from the configuration to better suite concrete scenarios. This way a
combination of a configuration and a profile can be chosen to suite the requirements of
a given platform more precisely.

JavaME supports several different profiles and configurations. A definition of them is
omitted here but can be found in [42, Chapter 2.2]. However during the introduction a
JavaME with the Connected Device Configuration and the Foundation Profile (JavaME
CDC/FP) was specified as a required runtime environment for the exemplary imple-
mentation. Therefore the limits and features of this combination are roughly outlined
in the following paragraph.

The Java 2 ME Connected Device Configuration uses a complete Java virtual machine
of the version 1.4. The class library of the foundation profile supports most of the basic
Java classes\(^1\). In addition to that it supports I/O operations which include TCP based
networking. However it excludes all classes related to graphical user interfaces like AWT
or Swing.

Additional information on the CDC can be found in [46]. Further information on
JavaME can be found in [42].

2.2 The OSGi Framework

OSGi is a standardized non proprietary Java component framework which has its
roots in the area of mobile and embedded devices but has become quite popular in
the area of enterprise applications. OSGi is developed and standardized by the OSGi
Alliance. The basic OSGi functionality is specified in the OSGi Core Specification[35].

\(^1\) Packages like java.util, java.lang and java.text
In addition to this the alliance specifies several commonly used services and utilities in the OSGi Compendium Specification[36] to provide a standardized environment of common functionalities to the developers.

OSGi provides a component model that is capable of deploying multiple applications within one Java runtime without letting them interfere with each other. OSGi applications thereby consist of multiple components. Each component has its own well defined life cycle and context. Due to a strict separation of the components context, OSGi supports to load multiple versions of the same class in the same runtime which is normally not possible in Java. Besides this OSGi supports dependency resolution which includes the explicit handling of different versions of the dependencies. Therefore a component defines its own version and another component can explicitly depend on it and limit the dependency to exactly this version or version range. However the dependency can also be specified for concrete Java packages which can than be provided by any component.

In addition to the concept of components the framework provides mechanisms to define, publish, look up and use services inside the runtime in a standardized manner. The OSGi service concept is described in section 2.2.2 in more detail.

There are many commercial and non-commercial implementations of OSGi available. Four well known implementations are Eclipse Equinox[19], Apache Felix[15], Makewave’s Knopflerfish[38] and ProSyst’s mBedded Server[39]. The mBedded Server is also the foundation of the Sprint Titan Stack which is used in the demo scenario. Equinox and Felix are also used in the demo and the realization. All mentioned implementations support the at the time of this writing current OSGi release 4.1.

The following two sections will now describe the concepts of components and services in OSGi in more detail.

2.2.1 Components

In OSGi each component is a bundle. A bundle is a Java archive file (JAR) which is enriched by additional meta information. The information further describes the bundle by for example specifying the bundle’s name, it’s dependencies or the bundles version. The additional meta data is located in the JAR manifest file and is used by the framework to produce additional functionality like dependency resolution or versioning. OSGi defines a concrete life cycle for bundles which includes the possibility to unload bundles during runtime. A bundle can implement the BundleActivator interface to be notified on the starting and stopping of the component and therefore about its life cycle states.

The OSGi life cycle defines the following states (Figure 2.1):

- **Installed**
  Once a bundle is installed into the framework it enters the state Installed. It stays in this state while the framework tries to resolve all dependencies required by the bundle.
• **Resolved**
 When the framework was able to resolve all dependencies, the bundle enters the *Resolved* state from which it can be started or uninstalled.

• **Starting, Active, Stopping**
 When the bundle is started it enters the *Starting* state, the bundle activator is instantiated and the start(...) method it called by the framework. After the start(...) method invocation returns, the bundle enters the state *Active*. When the bundle is stopped the stop(...) method of the activator is called and it enters the *Stopping* state as long as the method hasn’t returned. Afterwords the bundle returns to the *Resolved* state.

The start and stop methods also provide the bundle with a reference to the BundleContext which provides access to the OSGi framework and its abilities.

• **Uninstalled**
 The bundle was successfully uninstalled it can’t change into another state anymore.

In addition to the well defined life cycle, each bundle possesses its own class loader\(^2\). The used class loaders feature a different behavior than the default Java class loaders which delegate all requests to the parent class loader first. A bundle class loader only delegates requests for packages beginning with java. to the parent class loader\(^3\). Therefore the class space of a bundle is much more separated.

A bundle class loader also has per default no link to other bundle class loaders with the exception for the implicit link to the parent class loader\(^4\). Consequently the class

![Diagram of OSGi bundle life cycle](Image: [35, Chapter 4, Page 82])

---

1 The call of the start method can be deferred if the bundle uses a lazy activation policy (cp. [35, Section 4.3.7]).
2 With the exception of so called fragment bundles which extend the class path of their corresponding bundle.
3 They also delegate requests for packages which have explicitly been configured to be delegated.
4 The parent class loader is provided by the OSGi system bundle.
spaces of the bundles are isolated from each other. If a bundle $A$ wants to use a set of packages from a bundle $B$, $A$ needs access to the class space of $B$. Therefore the bundle depends on the availability of those packages. In OSGi such a dependency needs to be explicitly specified via the bundle meta data in the bundle manifest file. However OSGi not only requires the specification of the import of packages from other bundles. It also requires the explicit definition of packages that are exported by a bundle and thus allowed to be imported by others. Therefore bundle $A$ needs to specify that it wants to import a set of packages and bundle $B$ needs to specify that it allows access to this set of packages.

An example for a bundle manifest which imports the package $\text{org.osgi.service.event}$ in any available version and exports the package $\text{com.fusesource.lightsabre.eds.configuration}$ in version $0.5.0.SNAPSHOT$ is given in listing 2.1. The manifest also defines a bundle activator which will be used by the framework to propagate the bundles life cycle states.

**Listing 2.1:** A simple bundle manifest which imports and exports packages and defines a bundle activator

```java
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-SymbolicName: EventDistributionSystem
Bundle-Name: Lightsabre EventDistributionSystem
Bundle-Version: 0.5.0.SNAPSHOT
Bundle-Activator: com.fusesource.lightsabre.eds.internal.Activator
Export-Package: com.fusesource.lightsabre.eds.configuration; version="0.5.0.SNAPSHOT"
Import-Package: org.osgi.service.event
```

### 2.2.2 Services

Besides the concept of bundles, OSGi supports mechanisms to publish, find and use services to communicate across the borders of bundles\(^1\). Due to those capabilities OSGi is often referred to as a Service Oriented Architecture (SOA) for one Java VM. OSGi also defines a set of common services in it’s compendium specification like for example a log service, a configuration service and a HTTP server service.

OSGi provides a service registry where bundles can register services they provide and look up services they need. The service registry can be accessed directly via the bundle context and provides a simple API that uses the Java interface name of the service to identify it. In addition to the interface name, services can be registered with service properties which can than be used to select a concrete service. Listing 2.2 shows the registration of a service which implements the interface $\text{MyServiceInterface}$.

---

\(^1\) With the new OSGi 4.2 specification it also supports to communicate across the borders of the OSGi container.
Furthermore it shows the look up of this service based on its interface and the obtaining of the concrete service object.

**Listing 2.2: Registration and look up of an OSGi Service**

```java
// Register an OSGi service without any service poroperties
bundleContext.registerService(MyServiceInterface.class.getName(), new MyServiceImplementation, null);

// Lookup an OSGi service reference based on its interface and obtain the service
ServiceReference ref = bundleContext.getServiceReference(MyServiceInterface.class.getName());
MyServiceInterface service = (MyServiceInterface) bundleContext.getService(ref);

// Invoke some method on the service
service.sayHello('Marc');
```

OSGi services can be registered and unregistered during runtime. Therefore a bundle which uses a service needs to cope with this dynamic behavior. OSGi provides special mechanisms like the Service Tracker or Declarative Services to ease the handling of services. However a bundle can also handle the dynamic behavior on its own. OSGi provides the possibility to register service listeners which are informed by the framework of changes regarding a particular service or a set of services. Figure 2.2 shows the relationship between a bundle that publishes a service, a bundle that consumes a service and a bundle which listens for a particular service. The notation shown and explained in this diagram is a common OSGi notation. It covers all in OSGi possible service interactions and will therefore be used in this document for diagrams which describe the relationship of OSGi services. Additional information on the notation can be found in [23].

OSGi services are extended by distributed OSGi in the upcoming OSGi 4.2 specification. Distributed OSGi adds the possibility to use remote services and to expose OSGi services to the outside world in a standardized way. The concepts of distributed OSGi are described in section 3.5 as part of the analysis.

![Figure 2.2: Diagram notation for OSGi Services](image-url)
2.3 Asynchronous Messaging

The aim of this thesis is to integrate asynchronous messaging into OSGi. Therefore it is essential to define what asynchronous messaging is. The book *Enterprise Integration Patterns* by Gregor Hohpe defines messaging as follows:

“Messaging is a technology that enables high-speed, asynchronous, program-to-program communication with reliable delivery. Programs communicate by sending packets of data called messages to each other. Channels, also known as queues, are logical pathways that connect the programs and convey messages.” [20, Page 13]

Therefore the book defines messaging as a communication paradigm together with reliability as an essential *Quality of Service* (QoS) aspect which characterizes enterprise messaging systems. The following section will describe this paradigm in further detail. With regard to this definition the description of the general paradigm is extended with a description of the concepts behind the reliability aspect. Besides those descriptions, more detailed information can be found in the already mentioned book *Enterprise Integration Patterns* by Gregor Hohpe[20].

The exemplary implementation in this thesis will be based on the Apache ActiveMQ¹ message broker which is freely available and supports all of the mechanisms described in the following sections. Besides this non commercial broker some well known commercial message brokers are IBM WebSphere MQ² and Progress SonicMQ³.

2.3.1 The Messaging Paradigm

Messaging tries to decouple distributed systems by providing a generalized message based communication mechanism. When using a messaging system the sender of a message doesn’t communicate directly with the receiver of the message. Instead it sends the message to an appropriate *messaging channel* without knowing the final recipient. The messaging channel is provided by the *Message oriented Middleware* (MoM) also called the *Messaging System* or *Message Broker* in this document.

After a message was transferred to the messaging channel on the MoM, the middleware is responsible for the message delivery and the sender can continue with other tasks. Components which are interested in receiving messages from a given channel need to register them self as a receiver for this channel at the MoM.

The placement of the MoM between two applications is shown in figure 2.3. Each application acts as a *messaging client* of the MoM. The client functionality is mostly provided by vendor specific libraries for the specific messaging system. Therefore if Apache ActiveMQ is used as MoM, than the client will normally use ActiveMQ’s client

---

libraries to communicate with the messaging system\(^1\).

**Figure 2.3:** The message oriented middleware for the communication between applications. (Image based on [30, Figure 1-1])

Due to the decoupling, the sender and receiver don’t directly depend on each other. The sender just creates and sends the message and can immediately continue with other operations even if the final recipient of the message is currently unavailable. To achieve this decoupling the messages transferred by the middleware need to be self contained so that the receiver can process them independently from the sender. Messages are simple data structures containing a header and a body. The header contains messaging related meta information and the body the payload.

Messaging systems normally support two kinds of message channels. They differ in the way they handle multiple possible recipients for a message and the situation where no recipient is available when a message should be delivered (Figure 2.4).

- **Point to Point Channels**
  A point to point messaging channel delivers each message only to one receiver. Even if multiple receivers are registered, the message will only be delivered to one of them. If a receiver disconnects than all following messages are distributed between the remaining recipients. If no recipient is registered for the channel when a message is about to be delivered than the message is kept in the channel until a recipient is available. Those messaging channels are often called *queue* which is also the designation used for them in this document.

- **Publish and Subscribe Channels**
  A publish and subscribe channel delivers a message to all currently subscribed receivers. Therefore one message is broadcasted to multiple recipients. If no subscription of a receiver is available when a message is about to be delivered, the message is discarded. If a messaging client is disconnected for a period of time, it will miss all messages transferred during this period\(^2\). Such channels are often called *topic* which is also the designation used in this document.

---

\(^1\) In the case of ActiveMQ generic communication protocols such as STOMP or XMPP are supported by the message broker. Therefore the client can also use other libraries which allow him to communicate with one of the other supported protocols. More information on the supported protocols can be found in appendix B and in [8].

\(^2\) Durable subscriptions can be used to avoid message loss when a message client is disconnected. They are explained in section 3.3 of the analysis chapter.
2.3 Asynchronous Messaging

Figure 2.4: Messaging systems support two general channel types. (Image based on [30, Figure 1-4])

2.3.2 Reliability

Messaging systems provide methods to guarantee the delivery of messages even in the case of failures of the network, the messaging clients or the messaging system itself. They realize mechanisms which ensure the successful reception of messages from messaging clients and the successful transport of messages to the clients. In addition to this they can support some variation of a store and forward mechanism where the message broker stores the messages in a persistent storage area as soon as it received them. The messages are kept there until they could be successfully delivered to all designated recipients. This behavior guarantees that the messages will be delivered even if the broker crashes or is restarted while it is in possession of them.

Guaranteed delivery is discussed in further detail in the following analysis chapter in section 3.3.

As mentioned in the given definition of messaging, reliability is an important QoS aspect for a messaging system. Besides this aspect many other aspects are relevant like for example transactions, availability or security. However this thesis is focused on the basic functionality of a messaging system and will therefore only cover reliability oriented QoS aspects.

2.3.3 The Java Messaging System

The *Java Messaging System* (JMS) is a standardized API to access Message oriented Middleware. The API was specified by SUN Microsystems, the current version is 1.1 which was released in 2002 [48]. Nearly all of today’s messaging systems provide a JMS API as it is the standard for accessing messaging systems from within Java.

JMS specifies only the API for the interaction with the messaging system, not the wire level format. Therefore it is not possible to use for example the ActiveMQ JMS messaging client to communicate with an IBM Websphere MQ message broker. However if the application uses only the JMS API it can be used with all messaging clients which provide a JMS API.
2.3.3.1 The Java Messaging System API

As this document includes an implementation that uses JMS, the remainder of this section will give a brief overview on the API and its basic usage. More information on JMS and the API can be found in the specification [48] or in the book *Java Message Service* [30].

With the JMS API the developer creates a connection to the messaging system via a `Connection` object which can be created by an connection factory (Listing 2.3, Line 3-4). The factory can be explicitly instantiated or looked up via JNDI\(^1\).

Once a connection is created, the developer can create a `Session` object (Line 7). Within one session he can define several parameters like transactional behavior or the acknowledgment mode\(^2\) to be used.

To select the messaging channel he wants to communicate with, the developer creates a `Destination` object which can either be a topic or a queue destination (Line 10). In both cases the destination is created for a concrete messaging destination like for example a topic with the name `flight_K4711`.

The developer can now create a `MessageProducer` for the destination to send messages. Therefore he creates a `Message` object, assigns the message contents and sends the

```java
// Create a connection to ActiveMQ explicitly using its connection factory
ConnectionFactory f = new ActiveMQConnectionFactory(user, password, url);
Connection c = f.createConnection();
c.start();

// Create a non transacted session in auto acknowledge mode
Session s = c.createSession(false, Session.AUTO_ACKNOWLEDGE);

// Create a topic destination
Destination d = s.createTopic("flight_K4711");

// Create a producer and a TextMessage and send it via the producer
MessageProducer mp = s.createProducer(d);
Message m = s.createTextMessage("Hello World");
mp.send(m);

// Create a consumer and register a MessageListener to obtain messages from the MoM
MessageConsumer mc = s.createConsumer(d);
MessageListener l = new MyMessageListener();
mc.setMessageListener(l);
```

\(^1\) The Java Naming and Directory Interface (JNDI) is a standardized API to access naming and directory services (cp. [28]).

\(^2\) Acknowledgment modes are explained in section 3.3
message via the message producer to the destination and thereby to the messaging system (Line 13-15).

When the developer wants to receive messages from a topic or queue he needs to create an appropriate `MessageConsumer` and register a `MessageListener` with this consumer (Line 18-20). The message listener interface defines the callback method `onMessage(Message)` which is called every time a message is available to allow the application to process it. (Listing 2.4).

**Listing 2.4: The JMS MessageListener Interface**

```java
public interface MessageListener {
    void onMessage(Message message);
}
```
CHAPTER 3

Analysis

In chapter 1 a quick introduction into the aim of this thesis was given. In this chapter the aim will be refined by a definition of concrete requirements which a solution needs to fulfill. Before those requirements are defined, this section introduces some general notations. Afterwards a detailed view on reliability aspects in the Java Messaging System is given followed by an analysis of existing possibilities for OSGi internal and external communication. Based on this and the requirements, four integration approaches of messaging into OSGi are discussed and evaluated. The chapter concludes with the selection of the most promising approach to provide the foundation for the conception which follows in the next chapter.

3.1 General Components

The general components for an integration of asynchronous messaging into OSGi are defined in table 3.1. Figure 3.1 shows these components and their interactions. The Message oriented Middleware forms the central communication component which relays messages between the two OSGi containers and the legacy system. Each OSGi container has its own mediation component which mediates between the OSGi internal communication and the external messaging system to send and receive messages. This component will also be the messaging client of the messaging system.

<table>
<thead>
<tr>
<th>Component Name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Consumer Bundle</td>
<td>A bundle which wants to use messaging by receiving messages from the mediation component.</td>
</tr>
<tr>
<td>Message Producer Bundle</td>
<td>A bundle which wants to use messaging by sending messages via the mediation component.</td>
</tr>
<tr>
<td>Mediation Component</td>
<td>A component which mediates between the used OSGi internal communication and the external messaging system to send and receive messages. This component will also be the messaging client of the messaging system.</td>
</tr>
<tr>
<td>Foreign System or Legacy System</td>
<td>A foreign system which is reachable via the messaging system but which is not using OSGi with the mediation component. Such a component might not even be based on Java at all.</td>
</tr>
</tbody>
</table>

Table 3.1: General component definitions
communication mechanism and the middleware. Message producer or consumer bundles can use this component to indirectly communicate with the middleware. The messaging system relays messages from all messaging clients. Therefore messages received by the mediation component don’t necessarily originate from another OSGi container. They can also originate from a foreign system which might not even be Java based. The same situation is given in the opposite direction, the mediation component is not necessarily sending messages to other OSGi based mediation components but may also send messages to a foreign system.

Figure 3.1: The general components and their interaction
3.2 Definition of the Requirements for an Integration

In this section the requirements for an integration of asynchronous messaging functionality into OSGi are specified. Most of the requirements are derived from the concept of OSGi and the asynchronous messaging paradigm as introduced in chapter 2. The requirements are numbered to reference them later for comparison with the considered integration approaches.

3.2.1 Basic Functionality - R1

A mediation component needs to provide the functionality to access one or more messaging systems from within an OSGi container. It also needs to enable multiple bundles to access messaging systems in parallel. Furthermore the communication needs to be possible in both directions. Therefore the mediation component needs to enable a bundle to send and receive messages. Besides this topic and queue semantic should be available.

Summarized the following should be supported:

- Send and receive messages
- Topic and queue semantic
- Parallel access from multiple bundles to multiple messaging systems

3.2.2 Quality of Service - R2

As pointed out during the introduction to asynchronous messaging, reliable message delivery is of great importance. Therefore the integration approach needs to support mechanisms that guarantee the delivery of messages. The concepts of those mechanisms should be oriented on the concepts of JMS to ease the usage of the extension for developers which are familiar with JMS.

Besides this QoS aspect, other aspects like transactions, security or performance are important for messaging systems. However the aim of this thesis is to provide basic messaging functionality within OSGi. Therefore the thesis will only cover aspects regarding the reliable delivery of messages.

Summarized the following should be supported:

- Reliable message delivery for the sending and the receiving direction
- Reuse known concepts from JMS

3.2.3 Fit into the OSGi Framework - R3

The integration should provide the user of the OSGi framework with a solution to use messaging in a similar way as he uses the other framework functionalities. Therefore the mediation component should comply with the programming model and the concepts
of the OSGi framework as much as it is possible in a sense-full manner. In addition the
extension should try to reuse existing functionalities of the framework.
Furthermore it is important that the mediation component is capable of handling the
dynamic behavior of the framework and its components.
Moreover the integration approach should not change the framework itself but should
instead be installable as an extension by simply loading some bundles. Once those
bundles are deployed in a standard OSGi container, the messaging functionality should
be available.
Summarized the following should be supported:

- Follow the OSGi programming model and reuse existing functionalities and con-
  cepts
- Handle the dynamic behavior of OSGi
- Provide the integration as a pluggable extension to OSGi which doesn’t require
  changes of the framework or its implementation

3.2.4 Portability and Interoperability - R4
As already outlined in the description of Figure 3.1 the extension will not only send
or receive messages to or from other OSGi containers. It will also send and receive
messages to or from foreign systems. Due to this, the extension needs to be able to use a
message format that is not limited to OSGi\(^1\). Therefore it needs to allow interoperability
with foreign non OSGi systems.
Furthermore the mediation component should hide the underlying technologies like
for example the transport protocol or the type of messaging system. Therefore the
mediation component should provide a generalized messaging interface similar in concept
to JMS.
As OSGi has its roots in the area of embedded devices and as one of the goals of this
thesis is to provide an exemplary implementation for a Java ME CDC environment, the
mediation component should also provide at least the basic messaging functionalities in
such limited environments.
Summarized the following should be supported:

- Interoperability with foreign non OSGi systems
- Provide an abstraction from the underlying messaging implementation
- Provide a solution which can also be used in limited environments

\(^1\) Messaging systems can also implement a message translator to translate between otherwise incom-
patible formats. However this still requires that the semantic of the message has a meaning outside
of OSGi.
3.2.5 Transparent Usage - R5

The following descriptions on distributed OSGi will show that it enables the usage of remote services in a nearly transparent manner. It is therefore desirable that the integration of asynchronous messaging is also done in such a transparent way. Therefore the optimal case would be that the integration makes it possible to use a normal bundle together with the mediation component without changing the bundle itself. To achieve this the mediation component should, if possible, support an already existing OSGi API to communicate with the messaging system. Such an API would of course need to be compatible to the semantic of native messaging API’s.

If the used API isn’t completely supporting all messaging functionalities the mediation component should provide other means to access the full functionality.

Summarized the following should be supported:

- Try to provide the messaging functionality with an exiting OSGi API to enable the transparent usage of messaging

3.2.6 Discovery - R6

The mediation component should support discovery to for example find the address of a suitable broker or other configuration properties. The service discovery approach defined for the mediation component should be flexible enough to support different discovery mechanisms. Therefore it should not define a concrete communication protocol but should instead only define an API with a general discovery semantic. The concrete protocol and mechanism could than be implementation specific.

- Provide a concept to integrate discovery mechanisms in the mediation component

3.3 Reliable Message Delivery in the Java Messaging System

Requirement 2 defined that the integration approach should provide mechanisms for reliable message delivery. Therefore this section analyzes the basic mechanisms behind reliable message delivery as they are defined in the Java Messaging System to provide a basis for further discussions in this area. The reliability of the message transfer, which will also be called guaranteed delivery, is based on several mechanisms which are explained in the next sections. The following descriptions are mainly based on [30, Chapter 6].

3.3.1 Message Acknowledgments

The key concept behind guaranteed delivery is the concept of message acknowledgments. Acknowledgments are used to prove the successful transport of the message between the messaging system and the messaging client. In addition to this they are also used to acknowledge the successful reception and processing of a message by a messaging client. JMS defines three acknowledgments modes:
3.3 Analysis

- AUTO_ACKNOWLEDGMENT
- CLIENT_ACKNOWLEDGMENT
- DUPS_OK_ACKNOWLEDGMENT

The mechanism behind this concept will now be explained for the auto acknowledgment mode. The other modes are based on the same principles. Information on the other modes can be found in [30, Chapter 6]. The way how acknowledgments are used differ between the message producer and the message consumer. Therefore both cases will be treated separately by the following two subsections.

3.3.1.1 Acknowledgments on the Message Producer Side

When the message producer sends a message to the messaging system, the server sends an acknowledgment to confirm the successful reception of the message. The client can only consider the message to be transmitted successfully if he receives this acknowledgment. If the client for some reason doesn’t receive the confirmation he has to assume that the message was lost.

JMS defines the methods that are used for sending messages in its API as synchronous. If one of the methods is called it will block until the acknowledgment is received from the server. This behavior is shown in figure 3.2 part A. If the acknowledgment isn’t received in a given time frame the method throws an exception to notify the caller of the failure.

![Figure 3.2: Sending and receiving of messages with message acknowledgments](#)
3.3 Reliable Message Delivery in the Java Messaging System

3.3.1.2 Acknowledgments on the Message Consumer Side

When the messaging client is the message producer, it's the job of the messaging server to acknowledge the successful reception of the message. However, if the messaging client is the consumer of the message, the client needs to send the acknowledgment. Therefore, the messaging client acknowledges the successful reception and processing of the message to the messaging server. The messaging server is only allowed to consider the message as delivered after it receives this confirmation. If the delivery of a message fails for some reason, the messaging server needs to take care of the redelivery.

Due to this mechanism, the message producer only delivers the message to the messaging system. From this point in time only the messaging system is responsible for the successful delivery of the message to its destination.

On the consumer side, the acknowledgment is also handled completely by the messaging client library. The client library acknowledges a received message as soon as the `onMessage` callback returns without an exception. Therefore, the message is sent to the messaging client, is processed and afterwards the client acknowledges the successful reception and processing to the server (Figure 3.2 part B).

3.3.2 Store and Forward Messaging

Message acknowledgments only provide mechanisms to prove the successful handover of the messages between the messaging client and the server. Store and forward messaging defines a mechanism for the reliable handling of the messages once they are in the possession of the messaging server.

JMS defines two operation modes for the message handling by the messaging server, **persistent operation** and **non persistent operation**. In persistent operation mode, the server has to store messages to a persistent storage before it acknowledges the successful reception of the message. The intend of this behavior is to guarantee the successful delivery of the message even if the messaging server crashes while it handles the message. If the server uses the non-persistent mode, messages might be lost if the server crashes or is restarted at this moment. That is caused by the fact that in the non-persistent mode the server is allowed to keep the messages in a volatile storage area which is normally much faster than persistent storage.

3.3.3 Durable Topic Subscriptions

Besides those general mechanisms which apply for topic and queue semantic, durable subscriptions are used in the context of topics.

When a messaging client loses its subscription for a topic due to for example network problems, it would miss all messages that are delivered by the topic in this period of time. JMS provides the concept of a durable subscription to avoid such message loss.

When the client uses a durable subscription, the messaging system will store all messages for the client if it is not connected when the messages should be delivered. When the client connects again, the messaging system will deliver the stored messages.
To identify the subscription, the client needs to provide a unique identifier. When the client reconnects he has to present his old identifier again so that the server can identify him as the old subscriber. When the client wants to revoke the subscription he has to explicitly unsubscribe with his identifier.

3.3.4 Summary of Reliable Message Handling in JMS

Message acknowledgments provide the basis for the reliable transportation of the messages between the messaging client and the server. Durable subscriptions provide additional reliability when obtaining messages from a topic. Therefore those two mechanisms should be provided by the mediation component.

The store and forward messaging mechanism is limited to the messaging server. Therefore the mediation component does not need to realize this mechanism. However it should provide support for the message producer to define that the messaging server should use this mechanism.

3.4 Existing Possibilities for Internal OSGi Communication

This section together with the next section will analyze existing OSGi internal and external communication mechanisms. This will provide the foundation to define possible integration approaches in the following section 3.7. This section will start with an analysis of OSGi internal communication mechanisms.

OSGi provides two basic possibilities for “inter bundle” communication. The most basic possibility is to communicate through a shared library. The more advanced and proposed way is to communicate through OSGi services.

A shared library is simply a set of objects provided by a single bundle and used by several other bundles. The library can therefore relay arbitrary information or method calls between different bundles which use this shared space.

OSGi services can be registered and found via a service registry by any bundle. Besides the general concept of services, OSGi defines a set of common services. One of those services is the OSGi Event Admin service which is analyzed in further detail in the following subsection.

3.4.1 The OSGi Event Admin Service

The OSGi Event Admin service is one of the services specified in the OSGi Compendium Specification ([36, Section 113]). The basic functionality of the Event Admin is to provide asynchronous and synchronous event based communication within the OSGi container.

---

1 This is only true as long as the mediation component only relays the messages and acts transparently in the acknowledgment process. Therefore the mediation component may only acknowledge messages from the messaging server once they are successfully delivered to the OSGi internal receiver and vice versa.

2 Security restrictions may apply and thereby limit the access to services.
The Event Admin service provides the functionality to receive events from bundles and to deliver them to appropriate event handler services. An event handler service can be registered by any bundle when it is interested in receiving events. Such a handling service needs to be registered with meta data describing which events the handler is interested in.

The Event Admin provides a publish and subscribe mechanism with the well known topic semantic of messaging systems. Event producers can publish events to a topic and consumers can “subscribe”\(^1\) to this topic if they want to receive the events. The Event Admin discards events when there is no subscription for the topic at the given time.

The Event Admin identifies a topic by a hierarchical name where a topic /flight/4711 would result in the topic flight with the “subtopic” 4711. As a result an event handler can subscribe to the topic flight to receive all events from this topic including its subtopics. It’s even possible to subscribe to the topic “*” to receive all events from all topics.

OSGi events are simple Java objects which contain a string that describes the topic and a set of key-value pairs as properties. OSGi events are defined as read-only so they don’t provide methods to change the topic or the properties once an event was created.

Figure 3.3 shows the participating components and their interaction when using the Event Admin. The Event Admin implementation bundle realizes the OSGi Event Admin interface and registers its implementation as a service. An event producing bundle searches for the Event Admin service in the service registry and uses the service to publish events. The Event Admin implementation than determines which registered event handler services are suitable for delivering this event. The implementation than delivers the event to all suitable handlers.

When a bundle wants to receive events it needs to register an event handler service with the appropriate service properties. The properties specify the topics the handler is interested in and an optional filter expression. The filter expression applies to the event properties and allows a more precise selection of the events received. The registration

---

\(^1\) The subscription is realized via the Whiteboard Pattern which is explained in appendix A.
of such an event handler service is shown in listing 3.1.

Listing 3.1: Registration of an EventHandler service

```java
// Create a filter which selects events from the topic "demotopic" with the property "myProperty" set to "myValue"
Dictionary<String, String> eventFilter = new Hashtable<String, String>();
    eventFilter.put(EventConstants.EVENT_TOPIC, "/demotopic");
    eventFilter.put(EventConstants.EVENT_FILTER, "(myProperty=myValue"));

// Register MyEventHandlerImpl as an EventHandler service with the defined filter
bundleContext.registerService(EventHandler.class.getName(),
    new MyEventHandlerImpl(), eventFilter);
```

The EventHandler interface defines the `handleEvent(...)` method as shown in the following listing 3.2. The method is called by the Event Admin implementation whenever it needs to deliver an event. The Event Admin specification defines that no time consuming tasks should happen in this method. The specification explicitly allows the implementation to blacklist stale handlers and thereby to exclude them from further event deliveries (cp. [36, Section 113.8.2])

Listing 3.2: The OSGi EventHandler interface

```java
public interface EventHandler{
    void handleEvent(Event event);
}
```

The Event Admin service provides two methods to publish events. As shown in listing 3.3, both methods have the same signature but a different semantic.

Listing 3.3: The OSGi EventAdmin interface

```java
public interface EventAdmin{
    public void postEvent(Event event);
    public void sendEvent(Event event);
}
```

The method `postEvent(...)` publishes the event asynchronously to the event handler services. Therefore it returns immediately and the Event Admin implementation can deliver the event in another thread. The other method `sendEvent(...)` publishes the event synchronously to the event handlers. Therefore it will block until the event was handled by all currently registered handler services that are interested in this event.
3.4.1.1 Reliability Mechanisms

The Event Admin guarantees only for the delivery of events to all currently available event handler services and it provides no concept like a durable subscription or acknowledgments. The specification also doesn’t include a concept for persistent storage of not yet delivered events in the case of a framework or Event Admin shutdown.

3.5 External OSGi Communication: Distributed OSGi

This section will now give an introduction to a standardized mechanism for service based remote communication in OSGi. Besides this new mechanism OSGi supports several other “smaller” mechanisms for external communication which are mostly limited to Java ME environments or specialized for a concrete task. As the integration of asynchronous messaging is especially not limited to Java ME, those mechanisms are not covered here.

The at the time of this writing upcoming OSGi 4.2 specification introduces the possibility for service based remoting. This so called distributed OSGi allows to invoke remote services as if they where OSGi internal services and to publish OSGi internal services as external services in a standardized way. The following descriptions are based on the on the time of this writing most recent third draft of the OSGi 4.2 specification [37].

Distributed OSGi defines the functionality of a distribution software component and a set of service properties and their semantic. The distribution software is the implementation of the distributed OSGi specification. It is therefore the component which exposes a local OSGi service for example as a Web-Service or imports a remote Web-Service. The configuration of what shall be imported or exported is realized by service properties or optional configuration files which contain the service properties.

3.5.1 Realization

Distributed OSGi defines that services which are tagged with a special property are automatically exposed as for example Web-Services when an appropriate distribution software is running in the OSGi container. Therefore the service provider has nothing more to do than to register his service with a special service property at the OSGi service registry. Everything else is done automatically by the distribution software (Figure 3.4 left side).

On the service consumer side the usage is also quite transparent. A bundle which is interested in a service looks for the OSGi service at the local service registry. The distribution software is notified of this interest for a particular service. If the distribution software knows of an appropriate remote service it creates a proxy service object for this remote service. This proxy is than registered with the local service registry and the consumer bundle finds this service and can obtain a reference to the service to use it like any other OSGi service (Figure 3.4 right side).
3.5.2 Configuration

As mentioned the configuration of distributed OSGi is based on service properties on the service producer side. These service properties specify at least that the service is intended to be exported but they may also add additional information on how the service should be exported.

On the service consumer side information is needed to reach the remote service. This information can be provided by a discovery mechanism or by storing it in a configuration file. The configuration file needs to be located in a special location of a bundle. The distribution software scans for this file and loads the configuration values from it. This behavior is based on the Extender Pattern which is described in appendix A.

Besides the information needed to publish or consume a service, additional information may be provided. This information may specify Quality of Service aspects like for example encryption or integrity. Distributed OSGi specifies such additional requirements as intents. Those intents are defined by the Service Component Architecture (SCA) Policy Framework\(^1\) and provide a distribution software independent syntax.

More information on the configuration possibilities can be found in the distributed OSGi specification [37].

3.5.3 Discovery

Distributed OSGi supports the discovery of remote services. Therefore the distribution software can be used together with a discovery service to find remote services and to publish own services. The discovery mechanism is independent of the distribution software and could therefore also be used to discover other information like connection information for a messaging system.

---

\(^1\) Information on the SCA Policy Framework can be found in [31].
3.6 OSGi Specific Concepts and Limitations

This section covers some general OSGi specialties which need to be considered when creating a concept for the integration.

As already mentioned in section 2.2, OSGi introduces a clear separation of the contexts of the loaded bundles. This separation is realized by creating a separate class loader for each bundle. Those class loaders are not structured in a hierarchical manner but instead as a graph where each bundle can have access to each of the other bundles.

Bundles only have access to a specific class loader if they request access to it with for example an **Import-Package** statement in their manifest. The bundle on the other side needs to explicitly export packages which are intended to be usable by others.

This situation has great influence on the mediation component as it is intended to serialize and deserialize Java objects. For the deserialization of a Java object, the mediation component needs access to the class definition of this object. A normal Java class loader structure is much simpler and structured in a hierarchical fashion. Therefore the mediation component would have no problem to find the needed class definition. In OSGi the mediation component would need to know in advance which package it needs for the deserialization so that it can define an appropriate import statement. In addition to this OSGi supports to load the same class multiple times within different bundles and to load different versions of the same class. Those aspects make the resolving of the right class definition for the mediation component a complex task. Possible solutions for the selected approach will be discussed in section 4.1.4.3 in the next chapter.

3.7 Definition and Analysis of Possible Integration Approaches

After the definition of the requirements to a mediation component and an analysis of existing possibilities for OSGi internal and external communication, this section will now introduce some approaches on how messaging could be integrated into OSGi. These approaches will than be evaluated to select the most promising one for further investigation. This section will discuss the following integration approaches:

- A special messaging service for asynchronous messaging
- Distributed OSGi with the Apache CXF - JMS transport
- Asynchronous callbacks
- An Event Admin service based approach

---

1 Further information on Java serialization can be found in [29].
2 OSGi also supports a so called **dynamic-import** directive which enables a bundle to dynamically import foreign packages into the local class loader as soon as they are needed. However this directive has some serious limitations with regard to the dynamic behavior of the framework. A more detailed discussion on this can be found in section 4.1.4.3 in the next chapter.
3.7.1 Messaging Service Approach

The basic idea of the messaging service approach is to define the API for an specialized messaging service. This would basically mean to introduce the JMS API in form of a service version into OSGi.

Some of the JMS concepts could be adapted to suite better into the dynamic OSGi environment. Such an adaption could for example be done for the JMS message listeners. They could be realized using the Whiteboard Pattern\(^1\) which would make it easier to handle the dynamic behavior of OSGi.

The provided service and the used services of this approach and their interactions are shown in Figure 3.5. The mediation component bundle would provide the *messaging service* implementation. The service could be used by other bundles to send messages to topics or queues. A bundle which wants to receive messages would register a *message handler service*. The mediation component would listen for such services and deliver messages from the messaging system to them.

An evaluation of the approach on its compliance to the defined requirements is shown in table 3.2. Summarized this approach would introduce the full messaging functionality into OSGi but it also introduces a completely new API and therefore supports no possibility for transparent usage.

![Diagram of the messaging service approach](image)

**Figure 3.5:** The messaging service approach

---

\(^1\) The Whiteboard Pattern is explained in appendix A.1
### 3.7 Definition and Analysis of Possible Integration Approaches

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Compliant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1 - Basic Functionality</td>
<td>full</td>
<td>The service would be specially designed to provide the required functionality.</td>
</tr>
<tr>
<td>R2 - QoS</td>
<td>full</td>
<td>The service could be specially designed to fulfill all quality of service requirements.</td>
</tr>
<tr>
<td>R3 - Fit into OSGi</td>
<td>partial</td>
<td>The approach is not introducing any completely new concepts into the container. However it is not reusing an existing API with a messaging compatible semantic.</td>
</tr>
<tr>
<td>R4 - Portability / Interoperability</td>
<td>full</td>
<td>The service could be specially designed in such a way that the service consumer can define the message format. The underlying transport technology can also be hidden.</td>
</tr>
<tr>
<td>R5 - Transparent Usage</td>
<td>no</td>
<td>A OSGi application which uses this service for asynchronous messaging would completely rely on the existence of this service. It also needs explicit knowledge that it communicates with an external messaging system as the service is only intended for external communication.</td>
</tr>
<tr>
<td>R6 - Discovery</td>
<td>full</td>
<td>Discovery could be fully supported as the service can be designed appropriately.</td>
</tr>
</tbody>
</table>

#### Table 3.2: Evaluation of the messaging service approach.

3.7.2 Distributed OSGi with the Apache CXF - JMS Transport

The reference implementation of distributed OSGi is based on Apache CXF\(^1\) [13]. Per default it uses the combination of SOAP and HTTP for the communication with remote systems. Besides this common combination, CXF supports the transport of messages over a messaging system via JMS instead of HTTP [14].

The idea of this approach is to use the JMS transport to provide access to a messaging system via service calls. The basic steps when communicating with the messaging system would be the following:

1. An OSGi bundle “X” calls a Service which is provided as a proxy for a remote service by the distribution software (DSW). The call blocks until the DSW has called the remote service.

2. The DSW marshals the call into a SOAP message and transfers the message to the remote service. In the case of a normal Web-Service the transport mechanism would be HTTP. In this case the distribution software uses the messaging system to transfer the SOAP message.

3. A DSW on the service provider side obtains the message from the messaging system, unmarshals the contents and calls the appropriate local service.

---

\(^1\) Apache CXF is an open source Web-Service framework. Further Information can be found on the CXF homepage [2].
4. The return value of the service invocation is marshaled as a SOAP message and is transferred back to the service consumer DSW via the messaging system.

5. The consumer DSW unmarshals the response and returns the results from the call as the return value of the original service call from the service invoking bundle $X$.

The crux of this approach is that the messaging system is only used as a transport mechanism to realize synchronous remote method invocations on top of it. Due to this most of the functionalities of the underlying messaging system are not available. A more detailed evaluation is given in table 3.3.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Compliant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1 - Basic Functionality</td>
<td>partial</td>
<td>Distributed OSGi only provides synchronous communication based on method calls. Only this semantic is available in this approach to communicate with the messaging system.</td>
</tr>
<tr>
<td>R2 - QoS</td>
<td>partial</td>
<td>The DSW can support many QoS features. However these features are designed for service invocations and are therefore not always suited for the communication with a messaging system.</td>
</tr>
<tr>
<td>R3 - Fit into OSGi</td>
<td>partial</td>
<td>Distributed OSGi fits perfectly into OSGi. However it is build for synchronous remote procedure call communication and not for asynchronous message based communication. Therefore the user would have to use the wrong paradigm to communicate with a messaging system.</td>
</tr>
<tr>
<td>R4 - Portability / Interoperability</td>
<td>partial</td>
<td>The messages transferred via the messaging system are specifically created for service invocations. Therefore they are meant to call a specific remote message and are not intended to be processed as a simple self contained message.</td>
</tr>
<tr>
<td>R5 - Transparent Usage</td>
<td>partial</td>
<td>The usage of a remote service would be transparent but with the drawback that the messaging functionality is also completely hidden from the user.</td>
</tr>
<tr>
<td>R6 - Discovery</td>
<td>full</td>
<td>Distributed OSGi provides methods for service discovery. Those methods are intended to discover the remote endpoint but could also be used to discover the messaging system.</td>
</tr>
</tbody>
</table>

**Table 3.3:** Evaluation of the distributed OSGi approach

### 3.7.3 Asynchronous Callbacks

The previous integration approach introduced the possibility to use normal OSGi services to access a remote messaging system. One of the major drawbacks of this approach was the synchronous behavior of OSGi services. This approach now introduces a concept which avoids this synchronous behavior. As the approach represents simply an extended version of the distributed OSGi approach the description is therefore limited to the major changes and the resulting changes in the evaluation.
3.7 Definition and Analysis of Possible Integration Approaches

The basic idea of this approach is to let the OSGi service calls against the DSW return immediately. The results would be delivered asynchronously by invoking a provided callback or service reference. The basic steps when communicating with the messaging system would be the following:

1. An OSGi Bundle “X” calls a service which is provided as a proxy for a remote service by the distribution software (DSW). The call returns immediately. Besides the normal parameters, the bundle would need to provide information on a callback method or service to deliver the response to.

2. The DSW marshals the call into a SOAP message and transfers the message to the remote service via the messaging system.

3. The service provider handles the message (See steps 3 and 4 form the previous approach).

4. The DSW receives the response message and unmarshals it. It delivers the response to the specified callback function or service.

Therefore the concept is similar to the previous one with the exception that the communication is now asynchronous. Additional drawbacks compared with the previous approach are the requirement of additional knowledge of the changed service semantic by the service consumer and the usage of the callback concept which is not common in OSGi. Table 3.4 evaluates the concept in more detail.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Compliant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1 - Basic Functionality</td>
<td>partial</td>
<td>Much like the previous approach only that the communication is now asynchronous.</td>
</tr>
<tr>
<td>R2 - QoS</td>
<td>partial</td>
<td>Same as in the previous approach.</td>
</tr>
<tr>
<td>R3 - Fit into OSGi</td>
<td>no</td>
<td>The approach doesn’t fit into OSGi as it breaks the semantic of one of OSGi’s core functionalities (OSGi services). Besides this it introduces the concept of callbacks which is not common for OSGi.</td>
</tr>
<tr>
<td>R4 - Portability / Interoperability</td>
<td>partial</td>
<td>Same as in the previous approach.</td>
</tr>
<tr>
<td>R5 - Transparent Usage</td>
<td>no</td>
<td>The usage of such a service requires special knowledge about its changed semantic.</td>
</tr>
<tr>
<td>R6 - Discovery</td>
<td>full</td>
<td>Same as in the previous approach.</td>
</tr>
</tbody>
</table>

Table 3.4: Evaluation of the asynchronous callback approach

3.7.4 Event Admin Service Based Approach

As shown in section 3.4.1 the OSGi Event Admin service provides limited messaging functionality inside one OSGi container. This approach now facilitates the Event Admin API to integrate remote messaging into OSGi.

Generally there are two possibilities for the realization of the mediation component in this approach. The first possibility is to replace the Event Admin implementation with
a service implementation provided by the mediation component. Besides this possibility, the mediation component could interact as a normal event producer or consumer with an existing Event Admin implementation. This approach is described in the following section, followed by a description of the first possibility.

### 3.7.4.1 The Mediation Component as a Client to the Event Admin

The mediation component needs to access relevant events to forward them to the messaging system as messages. It also needs the possibility to inject messages which it receives from the messaging system as events so that the event handler services can receive those events.

This functionality can be realized by the mediation component if it simply acts as a “client” to the Event Admin (Figure 3.6). The mediation component can implement and register one or more event handler services. Those services will than provide access to the events that are intended to be forwarded to the messaging system. In the other direction the mediation component can use the Event Admin service to publish events which originate from a messaging system. This way the normal Event Admin implementation can be used and the mediation component can simply be added to the system.

### 3.7.4.2 The Mediation Component as Event Admin Replacement

Besides the less invasive approach of the mediation component to act next to the Event Admin, another possibility is to replace the Event Admin implementation.

![Figure 3.6: The mediation component as a client to the Event Admin](image-url)
The replacement would than provide remote messaging functionality besides the normal Event Admin functionality (Figure 3.7). This way the implementation would have full control on how the events are delivered. On the other hand this requires that the complete Event Admin functionality is implemented by the mediation component.

3.7.4.3 Conclusion of the Event Admin Approaches

Both approaches introduce basic messaging functionality into OSGi without the need for a new API. Furthermore they enable legacy bundles which until now only used the Event Admin service for internal communication to communicate with external messaging systems.

The mayor drawback of both approaches is the absence of queue style messaging and QoS features like guaranteed delivery. A detailed evaluation of the approach is shown in table 3.5.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Compliant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1 - Basic Functionality</td>
<td>partial</td>
<td>The Event Admin API provides only topic based messaging functionality. The optimal case would be if also queue based messaging is supported.</td>
</tr>
<tr>
<td>R2 - QoS</td>
<td>no</td>
<td>The Event Admin API provides no semantic for guaranteed delivery or other reliability functionalities.</td>
</tr>
<tr>
<td>R3 - Fit into OSGi</td>
<td>full</td>
<td>The approach uses an already existing OSGi API and doesn’t introduce any new concepts.</td>
</tr>
<tr>
<td>R4 - Portability / Interoperability</td>
<td>full</td>
<td>The Event Admin hides the specialties of the concrete messaging system completely behind its own API. Events are defined as self contained “messages” and are well suited to be translated into for example JMS Map Messages.</td>
</tr>
<tr>
<td>R5 - Transparent Usage</td>
<td>full</td>
<td>Bundles which used the Event Admin for internal communication could also be used with a remote messaging system and vice versa.</td>
</tr>
<tr>
<td>R6 - Discovery</td>
<td>full</td>
<td>As all the configuration needed for the connection to a messaging system is concentrated in the mediation component, the component could easily interface with a discovery system.</td>
</tr>
</tbody>
</table>

Table 3.5: Evaluation of the Event Admin approach
3.8 Selection of an Integration Approach

The evaluation of the approaches rule out the distributed OSGi and the asynchronous callback approach. The Event Admin approach and the messaging service approach show a similar count of supported and unsupported features. This prevents an easy decision. A closer look at the supported requirements of both approaches shows that they are mostly oppositional (Table 3.6). The messaging service has its strength in the functionality whereas the Event Admin approach provides a better integration into OSGi and transparent usage. Therefore a combination of both approaches could be capable to fulfill the full set of requirements.

The combination of both approaches could be realized by utilizing the Event Admin approach for the basic functionality and transparent usage of a messaging system with a limited feature set. An additional messaging service with a specialized API could provide the additional functionality which is not covered by the Event Admin. Both APIs could even be merged to provide an extended Event Admin API which could be usable with legacy bundles as well as with bundles which need the full functionality of a messaging system.

As this combination promises the best compliance with the defined requirements it is selected as the basis for further research in the following chapters.

Table 3.6: Results from the evaluations of the different integration approaches
CHAPTER 4

Key Concepts

In the previous chapter the Event Admin approach together with an additional messaging service was selected. In this chapter a concept for this approach is created to further evaluate its capabilities and to provide the foundation for the exemplary realization.

As a foundation for the concept, this chapter starts with a discussion of several realization options of the concept. The aim of this process is to determine the appropriate options to combine the advantages of the Event Admin approach and the messaging service approach. Based on the discussions from this first step, the complete concept is defined in the second step. The capabilities and limits of the concept are evaluated based on the previously defined requirements. The created concept will then be used in the following chapter for the realization.

In section 3.1 the general mediation component was introduced. As the mediation component is now discussed in the context of the Event Admin approach it will from now on be called the Event Distribution System (EDS) and both designations can be considered synonymous. Furthermore events and messages will have a relatively similar meaning in the following discussions. Messages are the interchanged data structures between the EDS and a messaging system. Events are the data structures used inside OSGi for the communication between the Event Admin, event consuming or producing bundles and the EDS\(^1\).

4.1 Realization Options

This section will now discuss several different realization options for the conception. The results will be used to define the complete concept in the following section.

As already outlined two general possibilities for the realization of the raw Event Admin approach are available. One approach would replace the Event Admin and the other would introduce the Mediation Component as a client to the Event Admin. The

---

\(^1\) The conversion between events and messages that needs to be done by the EDS will be discussed in section 4.1.4.
following discussions will cover both approaches to determine which approach is more suitable.

4.1.1 Basic Functionality

As defined in the first requirement, the basic functionality of the messaging integration is to provide access to messaging topics and queues. It needs to be possible to send and receive messages in both communication styles with their appropriate semantic.

The OSGi Event Admin provides topic based publish/subscribe semantic to send and receive events. It has no build in support for queue based communication. Consequently the EDS has to obey this limitation if it acts as a client to the Event Admin.

If the EDS replaces the Event Admin it has full control over the event delivery. Due to this it could realize queue functionality by changing the behavior of the event delivery. However this would break with the semantic of the Event Admin API.

The main advantage of the usage of the Event Admin API is the interoperability with legacy bundles and the transparent usage as already discussed during its analysis. The integration of queue based semantic by breaking the Event Admin semantic would break this interoperability. Bundles which would use such a modified Event Admin would not necessarily work with the normal Event Admin. Therefore the implementation of queue semantic hidden behind the Event Admin API should be avoided. Possible solutions for this matter are discussed in the following subsections for the event producing and the event consuming side.

4.1.1.1 Event Producing

As the API of the Event Admin is defined with a topic based semantic, queue based semantic can’t be realized with it. Therefore an additional API seems to be an appropriate solution to provide queue based messaging. Such an API could provide a method like `sendEventToQueue(Event e)`. To allow a bundle to use the new API together with the old Event Admin API, the new API could extend the interface of the Event Admin. Thereby an extended Event Admin API would be available for clients with the requirement of the extended functionality. This would also have the advantage that those clients would explicitly depend on an extended interface. If queue based communication would be hidden behind the old API, the client could not be able to directly distinguish between the normal Event Admin and its extended version.

4.1.1.2 Event Consuming

When events are consumed from a topic, the normal event handler mechanism and its semantic is sufficient as it was designed for exactly this purpose. However when events are obtained from a queue, than the event handler mechanism doesn’t provide the appropriate semantic. Therefore the reception of events should take place either by a new concept or by a modified version of the event handlers.
An event handler defines a simple interface with a method that is called when an event is delivered from a topic. Exactly the same functionality is needed for the reception of events from a queue. Therefore the event handler interface could also be used with queues but with different service properties when the event handler is registered. The new service property could be named `event.queue` and would replace the `event.topics` property. Therefore no new API is needed and the different semantic would be explicitly chosen by the developer when he registers the handler.

This would also be a compatible approach with the existing Event Admin because the specification explicitly defines that event handlers without the `event.topics` property must be ignored by the normal Event Admin [36, Section 113.4].

Another service property that can be defined for an event handler is the `event.filter` property which enables the handler to filter the delivered events based on their properties. This service property normally acts in addition to the `event.topics` property. However this property could also be used to filter events which the handler receives from a queue.

Consequently this service property provides a similar semantic as JMS Message Selectors which can be used to filter the messages delivered by a messaging server to a messaging client. However this possibility will not be covered in further detail as it is not part of the required basic functionality.

The proposed solutions for the event publishing and the event consuming provide the basic functionality without breaking with the concepts of the Event Admin. Furthermore no new concepts are introduced and a clear separation between the mechanisms with topic and queue semantic is given.

### 4.1.2 Reliability Aspects

#### 4.1.2.1 Guaranteed Delivery

As outlined during the analysis of the Event Admin approach the Event Admin API provides no mechanisms for guaranteed delivery. However those functions are crucial for an enterprise grade messaging system. Therefore this section discusses how or if such functionality can be realized on top of the Event Admin API or if it needs to be realized by the extended API.

**Sending Messages to a Messaging System**

When a bundle sends an event to the Event Admin it has no way of knowing if there is any receiver registered for this kind of event. This is also the case when a bundle wants to send an event as a message to the messaging system and the EDS acts as a client behind the Event Admin. The sender can not know if the event was handled by the EDS or if it was discarded because no appropriate handler was registered.

Even if the EDS receives the event, the event producer can’t be informed of a possible failure during the event forwarding to the messaging system. The situation is shown in figure 4.1. It shows the calls which would take place between the components when
an event is send to the Event Admin, forwarded to the EDS and is finally delivered to the messaging system. The figure shows the asynchronous `postEvent(...)` call where the delivery of the event to possible handlers takes place asynchronously. Therefore the caller can’t be informed of any failures that might occur during the event delivery.

The Event Admin API also provides the synchronous `sendEvent(...)` call which blocks until the delivery of the event is complete. However the Event Admin is not allowed to throw any exceptions from this method. He is explicitly required to catch all exceptions which might occur during the event delivery [36, Section 113.8.1]. Therefore the caller won’t be informed about any possible failures during the processing and the delivery of the event.

**Figure 4.1:** Asynchronous event delivery: The sending bundle has no possibility to determine if the event was received by the EDS and delivered to the messaging system

The other possibility would be that the EDS replaces the Event Admin. In this case all event producers would directly interact with the EDS and the event delivery would therefore be guaranteed. However the EDS has still no possibility to notify the sender if the event can’t be forwarded to the messaging system\(^1\). This is once again caused by the fact that the specification forbids the Event Admin to throw exceptions from the `send` and `post` methods. If the EDS Event Admin implementation would misbehave at this point and throw runtime exceptions to the sender it would break the compatibility with bundles which are not prepared to catch exceptions at this point.

Based on those cognitions, guaranteed delivery for the event sending case via the Event Admin is not possible in an appropriate way. Therefore it needs to be realized via the specialized API of the extended Event Admin. The API could define a method for sending events that blocks until the event is delivered to the messaging destination. When the delivery fails, the method could throw an exception. The declaration of the method is shown in Listing 4.1. The semantic of this approach is similar to JMS where the send method blocks and throws an exception in the case of a failure.

---

\(^1\) The event might for example not be forwarded because of missing configuration on the EDS. Therefore the EDS might not know if the event is intended to be forwarded and is therefore only delivered to local event handlers or is discarded if no handlers are available.
4.1 Realization Options

Listing 4.1: Definition of a blocking fault aware `postEvent` method.

```java
class ExtendedEventAdmin {
    /* [...] */

    // The postEvent method blocks until the event was delivered to
    // the corresponding messaging destination. If the delivery fails,
    // the method throws an exception to notify the caller.
    public void postEvent(Event e) throws MessagingException;
}
```

Receiving Messages from a Messaging System

A similar situation is given when the EDS is acting as a client to the Event Admin and receives a message from the messaging system. The EDS needs to acknowledge the successful reception and processing of the message to the messaging system. However, the EDS has no way of knowing if the message was delivered to the receiving bundle (Figure 4.2) if it acts as a client to the Event Admin. If the receiving bundle was not available at this moment, the event might be lost. Even if the event was delivered, the processing of the event might fail and the EDS won’t be notified about this fact. Consequently the EDS would need to acknowledge the message blindly without knowing if it was delivered to and processed by the receiving bundle.

A possible solution for this is to let the EDS deliver the event directly to the event handler service. Thereby the EDS would know if the delivery was possible. It can also check for exceptions thrown during the `handleEvent(...)` callback and acknowledge the message only if no exception occurred.

A second possibility would be to replace the Event Admin implementation and thereby deliver the events directly. However this would provide the same result as the previous solution with the drawback that the complete Event Admin implementation needs to be built into the EDS.

Consequently the possibility to deliver the events directly without replacing the Event Admin is chosen here.

![Figure 4.2](image-url): Asynchronous event reception: The EDS has no information if the delivery of the event to the ultimate receiver was successful but needs to acknowledge the reception and successful processing of the message.
4.1.2.2 Durable Subscriptions

The Event Admin doesn’t provide any mechanisms similar to durable subscriptions. When events are transported via the normal Event Admin implementation, they will be discarded if no event handler is available when the Event Admin tries to deliver the events. However, if the EDS directly delivers the events, an event handler can be registered with additional service properties. These properties could then be used to identify the subscription and to mark it as durable. Consequently, the EDS would be able to track the subscription and to preserve events when the subscriber is not available. Once the subscriber becomes available again, it could deliver them. This provides the possibility to, for example, update a message receiver bundle without risking any message loss. Listing 4.2 shows such a registration with exemplary service properties.

```
Dictionary props = new Properties();
props.put("messaging.id","4711");
props.put("messaging.durable",true);
props.put("event.topics","flightInfo");
ServiceRegistration registration = bundleContext.
    registerService(EventHandler.class.getName(),new MyHandler()
    (),props);
```

When such a subscription is not automatically revoked when the event handler is unregistered, another possibility for the revocation needs to be provided.

JMS provides a special unsubscribe method which needs to be called by the client if he wants to end a durable subscription. Such direct calls are not possible as the client is not utilizing any special API to create a durable subscription. Consequently, another solution needs to be found to end such a subscription, preferably without the need for a special API. Two possibilities are discussed in the remainder of this section.

- **Property Based Unsubscribe**
  
The handler would in this case be registered with a set of properties to define it as a durable subscription. If the bundle wants to revoke the subscription it could change the properties of the event handler before he unregisters it. The EDS would use a listener for event handlers to be informed about such changes and could remove the subscription. Listing 4.3 shows such an unsubscribe process based on the registration from listing 4.2.

---

1 How the EDS could handle failed deliveries for topic-based communication is discussed in the next section 4.1.2.3.
4.1 Realization Options

Listing 4.3: Revoke a durable subscription by changing the service registration

```java
// Update the properties revoke the durable subscription
Dictionary props = new Properties();
props.put("messaging.id","4711");
props.put("messaging.durable",false);
props.put("event.topics","flightInfo");
registration.setProperties(props);
// Remove the EventHander from the service registry
registration.unregister();
```

- Time Based Unsubscribe

The registration of the durable event handler could be extended by additional properties which define a timeout for the subscription. This value would specify that the subscription ends when the handler is not registered for a longer period of time than the specified maximum. Such a policy could also be defined system wide for all subscriptions to avoid stale subscriptions from misbehaving bundles which might simply “forget” to unsubscribe.

The property based unsubscribe provides a solution which gives the developer full control over the subscription. Furthermore this approach features the same semantic for the subscription and the revocation without the need for a different API. The time based approach can be seen more as a fallback solution as the developer has no possibility to explicitly revoke a subscription. Consequently the most reasonable solution seems to be the property based approach.

4.1.2.3 Handling Messages for Multiple Receivers

One part that has been left open so far is what happens when the EDS receivers one message from the messaging system but has multiple receivers inside the OSGi container. The EDS needs to acknowledge the message once it was received and processed. When only one receiver is available the acknowledgment can take place once the event was delivered. If there are multiple receivers the decision if the acknowledgment can be send is more complex and depends on the semantic of the used messaging destination. This situation is discussed in the following two sections for queue and topic based communication.

A Queue with Multiple Receivers

If a message is received from a message queue and the EDS has the choice to deliver the message to one of multiple possible recipients the message can be acknowledged once it was delivered to the chosen one. If the delivery to this receiver fails the EDS could return the message to the messaging system by not acknowledging it. Besides
this the EDS could first try to deliver the message to other possible receivers and only return the message if no receiver was able to successfully process the message.

A Topic with Multiple Receivers

If the EDS receives a message from a messaging topic and has multiple receivers in the OSGi container to deliver the resulting event to, it can only acknowledge the reception once the delivery was successful for all recipients. However a problem occurs when the delivery was successful for a few receivers and failed for at least one receiver. In this case the EDS can’t simply acknowledge the message. To solve this issue the EDS could use one subscription for the messaging topic per internal receiver. Thereby the EDS would receive the message once for each receiver and could acknowledge the reception separately.

Durable subscriptions

The previous section on durable subscriptions left open how the EDS could handle the case where an internal durable subscription is not available but the EDS receives a message from the messaging system for it. This problem could be solved in two different ways.

The EDS could queue the events that need to be delivered to the durable subscription and acknowledge the reception to the messaging system immediately. It could then deliver the events once the receiver is available again. However this implies that the EDS needs to take care that the events are really delivered to the receiver. Therefore the EDS would for example have to store them in a persistent storage area or take other precautions to avoid to loose the events.

The other possibility would be based on the previously described approach to use one topic subscription for each internal receiver. This could also be done for durable subscriptions. Therefore the EDS would create a durable subscription at the messaging system for each local durable subscription. If the local subscription becomes unavailable the EDS could stop to receive messages on this subscription. Once the internal subscription becomes available again, the EDS could reactivate the durable subscription to the messaging system and all missed messages would be delivered by the messaging system.

A concrete decision which of the approaches should be used in each of the three situations is not made here because it is considered implementation specific as it doesn’t influence the usage of the EDS.
4.1 Realization Options

4.1.3 Configuration

The Event Distribution System is intended to selectively forward events as messages to messaging systems and vice versa. To fulfill its duties, the EDS needs to be configured appropriately. Therefore it needs to know which events should be forwarded, where they should be forwarded to and vice versa.

This section will cover different approaches how the configuration data can be applied and which configuration data should be specified. Furthermore it defines how discovery can be realized.

The section will begin with the following descriptions of the possibilities how configuration data can be applied to the EDS:

- Configuration files
- An EDS configuration service
- Configuration data from the event handler registrations and the events
- The OSGi Configuration Admin

4.1.3.1 Configuration Files

The necessary configuration information could be stored in configuration files which could than be loaded by the EDS during it’s start-up. However this would require that the EDS reloads the configuration every time it changes. Furthermore other bundles need to “leave” the framework to change the configuration files and than trigger an EDS specific mechanism to load the changes.

To eliminate the drawbacks of the external files the configuration files could be placed inside normal bundles and the EDS could realize the Extender Pattern\(^1\). The EDS would scan all bundles for an appropriate configuration file and apply their contents during runtime. Configuration changes could be realized by reloading the bundle with a different configuration file. However this still doesn’t provide a possibility to change the configuration data programmatically.

4.1.3.2 EDS Configuration Service

The EDS could provide a special service for its configuration. This service would provide a specialized interface to apply the necessary configuration information programmatically.

This has the drawback that the configuration options could only be set programmatically and that the configuration information would be located in the program code of the configuring bundle\(^2\). Furthermore the bundle would have to use a completely new interface. Moreover this approach causes problems for the usage of legacy bundles as

---
\(^1\) The Extender pattern is described in appendix A.
\(^2\) The bundle could of course load the configuration data for example from a property file. However this would make the implementation needed to configure the EDS more complicated.
they wouldn’t use the service to apply the configuration on their own. In this case an additional bundle would be needed to apply the configuration information needed by the legacy bundle.

Another major drawback is that the configuration data would need to be explicitly reapplied by the configuring bundle if the EDS restarts. This extends the complexity of a configuring bundle as it would have to monitor the life cycle of the EDS.

4.1.3.3 Configuration Data from the Event Handler and the Event

Another possibility would be to specify the configuration data together with the registration of the event handler services or as additional properties on the events that should be forwarded. Both possibilities are discussed in the following two subsections.

Configuration via the Event Handler Service Registration

An event handler could be registered with additional service properties like the connection information of the messaging system and the name of the topic or queue it wants to obtain messages from. The EDS would find those services via the service registry and use the additional information to configure itself for the forwarding.

This approach shares some of the drawbacks pointed out for the EDS configuration service as the configuration can also only be applied programmatically. On the other hand in this case there is no need to reapply the configuration data if the EDS restarts as the data is available via the registration.

However the approach has a major drawback: It is only possible to set configuration information by registering an event handler. If the bundle which wants to receive messages is a legacy bundle where the handler registration is not changeable, another bundle would have to register a “dummy” event handler to provide the configuration information. Consequently this dummy would also receive events from the EDS which results in a problem when the handler is supposed to obtain messages from a queue. In this case each message is only delivered to one of the receivers and the real event handler would loose messages.

Configuration via Event Properties

When an event is intended to be forwarded to a messaging destination, the event producer could add the necessary configuration data as additional properties to the event. When the EDS receives the event it could use those additional properties to configure itself to forward the event.

Once again the configuration is only possible programmatically. If the configuration data would be added to every event, the EDS could restart without loosing the configuration as it will be added again when the next event is about to be forwarded. However in this case the deletion of the configuration poses a problem as the EDS has no way of knowing if a received event was the last one related to this configuration. Furthermore a problem with the configuration for legacy bundles exists as the bundle
itself wouldn’t produce events with the needed configuration data. If another bundle would send dummy events to configure the EDS it would also produce dummy messages which would be forwarded to the messaging destination. In addition this solution would produce a race condition if the EDS is restarted¹.

4.1.3.4 The OSGi Configuration Admin

The OSGi compendium specification defines the Configuration Admin (CA) as an unified mechanism to configure bundles in OSGi [36, Section 104]. The Configuration Admin is specified as a service with which configuration data can be set and “configuration receivers” can receive this configuration data. A service which wants to receive configuration information needs to implement either the ManagedService or the ManagedServiceFactory interface and register itself at the service registry. The CA realizes the Whiteboard pattern² to apply the configuration to the “listening” services. Figure 4.3 shows the CA with its relations to other components.

The Configuration Admin manages sets of configuration data where each of those sets has its own Persistent Identity (PID). Those configuration sets are a simple set of key value pairs. The CA can store those configuration sets in a persistent storage area so that they survive for example a framework restart.

If a service wants to obtain configuration data from the CA it has to register itself at the service registry as follows: The service must be registered with the service property service.pid and it must be registered with one of the above mentioned two interfaces. When the CA finds such a service it searches for a set of configuration data that has the same PID as the service. If a configuration set is found, it is applied to the service.

![Figure 4.3: The basic functionality of the OSGi Configuration Admin](image)

¹ The configuration bundle would need to reapply the configuration data before the legacy bundle sends the next event. If the legacy bundle sends the event first, the event would not be forwarded to the messaging system as the EDS has no appropriate configuration.

² The Whiteboard pattern is described in appendix A.
The two possible interfaces `ManagedService` and `ManagedServiceFactory` represent two different configuration modes of the CA:

- **ManagedService**
  This interface is intended to be used for a service that only needs one set of configuration data. Therefore one service instance is configured with one set of configuration information.

- **ManagedServiceFactory**
  This interface is intended for services that need to obtain multiple sets of configuration information. This mechanism might for example be used when the service acts as a factory for other services and each of the created service instances should have its own set of configuration data.
  The managed service factory instance needs to be registered with its own PID. All configuration blocks that are applied to the factory service also have their own PID and the PID of the factory defined as their `factory PID`.

The Configuration Admin supports configuration updates during runtime. The changes are applied to the managed services in the same way as they are applied when the configuration is defined for the first time. For this purpose both interfaces define an `updated(...)` method which is called with the current configuration information. The interface of the managed service factory is defined as shown in listing 4.4. In the case of the managed service factory the `updated(...)` method is called for each of the configuration sets assigned to this factory. Therefore for all configuration sets that have the PID of the factory assigned as their factory PID. The `deleted(...)` method is called every time a configuration set of this factory is deleted.

The Configuration Admin supports the dynamic handling of configuration data in a standardized way. However the configuration is only possible programmatically but in this case it only needs to be done once even when the EDS restarts as the CA will reapply the information as soon as the service is available again.

**Listing 4.4: The ManagedServiceFactory interface for the Configuration Admin**

```java
public interface ManagedServiceFactory{
    // called when a set of configuration data is deleted
    public void deleted( String pid );

    // Returns a simple name for the factory
    public String getName( );

    // Called for every update and for every new set of suitable
    // configuration data
    public void updated( String pid, Dictionary properties )
        throws ConfigurationException ;
}
```
Usage of the Configuration Admin for the EDS

The EDS could register a managed service factory to obtain multiple sets of configuration data from the Configuration Admin. Due to this the EDS would be configurable via OSGi’s standard configuration mechanism. Further evaluation of this approach is done in the following section.

4.1.3.5 Conclusion on Configuration Mechanisms

Each of the possibilities has its own advantages and disadvantages in certain use cases. However the Configuration Admin provides a simple standardized interface to define configuration parameters and to apply them to configurable parties. In addition this configuration approach integrates perfectly with OSGi as it uses OSGi’s standard configuration mechanism.

The only major disadvantage is the missing possibility to specify configuration information in a non programmatic way. This problem could be solved when an additional bundle would realize the Extender Pattern. It would therefore scan for configuration files located in bundles and apply the contained configuration data to the Configuration Admin. This way a standardized service can be used for the configuration of the EDS and a specialized file format could be used as a loosely coupled extension to this default configuration mechanism.

4.1.3.6 Configuration Parameters

The Configuration Admin supports arbitrary sets of key value pairs as configuration data. This section will now outline some general configuration parameters. It is not intended to provide a full list of all possible configuration options. It is rather intended to give an impression of the kind of configuration that is necessary.

The EDS should support multiple simultaneous connections to different messaging systems. Therefore the EDS needs to be able to handle multiple sets of configuration data where each set configures one particular forwarding. Such configuration blocks need to be identifiable by the EDS, therefore they should have an own identifier such as a unique name. The PID defined by the Configuration Admin is not suitable for this purpose as the data necessary to fully configure a forwarding could be split between multiple configuration sets from the CA which would have a different PID\(^1\). In this case the EDS has know possibility to no about their relation.

Each complete configuration block needs information on the following topics:

- Connection information for the messaging system and the messaging destination where the EDS should connect to.

\(^1\) This could for example happen if a basic configuration is specified by the application developer and the configuration is later extended with deployment specific information like the address of the messaging system.
• Information on the forwarding direction. Therefore if messages should be send to the messaging system or if they should be received from it or both.
• The semantic (queue or topic) of the connected messaging channel.
• If events should be forwarded to the messaging system: Information from which topic they should be selected and if an additional filter should be applied.
• If messages should be received as events: The event topic where the events should be delivered to.

In addition to this general configuration information, additional EDS implementation specific information might be supplied.

The configuration properties could be grouped together based on their purpose. Therefore all messaging system related information could form a group as well as all Event Admin related information. Such a grouping would provide the advantage that the EDS could easily decide which part of a forwarding is affected by configuration changes. This would make it easier to realize the EDS in such a way that for example the following is possible:

The EDS is configured to forward events from a local topic without any special guaranteed delivery options. The configuration of the forwarding needs to be changed because the EDS should connect to a different messaging system. Due to the separation of the configuration data the EDS can decide that it only needs to reinitialize the connection to the messaging system. The other part of its implementation could continue to receive events and queue them until the messaging part becomes available again. In this case no events would be lost.

The grouping of the configuration properties could simply be achieved by adding a prefix to the name of the configuration parameter like “messaging.” or “event.”. Listing 4.5 shows an exemplary configuration where the configuration properties are extended by a prefix. The shown configuration would configure the EDS to obtain events from the topic flightInfo and to forward them to a messaging topic FlightInformation. The listing also shows a custom implementation specific parameter with its own prefix.

The shown parameters define the necessary information to setup a forwarding. However additional properties are necessary if the configuration should define for example properties for a service discovery or QoS aspects. The QoS properties could follow the principle used by distributed OSGi which uses SCA intents for this purpose.

Listing 4.5: An exemplary configuration block for the EDS

```
# A name to identify this configuration block
name=FlightInformation

# The EDS should send events to the messaging system
direction=send
```
4.1 Realization Options

1. # The EDS should forward events from the event topic 'flightInfo'
   event.topic=flightInfo
2. event.filter=
   # The EDS should forward the events to ....
3. messaging.url=tcp://localhost:61616
4. messaging.user=root
5. messaging.password=secret
6. messaging.subject=FlightInformation
7. messaging.style=topic

# ** Possible implementation specific configuration values **
# In this case the EDS implementation could support different messaging clients. This configuration would specify that the ActiveMQ client should be used.
8. myImpl.binding=activemq

4.1.3.7 Alternative Configuration Possibilities

Besides this relatively static configuration for the selection of messages and events that should be forwarded, another approach would be possible. The configuration could specify rules or patterns to for example map dynamically created event topics to matching messaging destinations. Therefore it could specify something like “Forward events from topic /orders/<OrderId> to the messaging queue order_<OrderId>”. This more dynamic approach is also possible with the selected Configuration Admin mechanism. However this approach is considered as a specialized solution and is therefore not discussed in further detail.

4.1.3.8 Discovery

The previous sections defined how the configuration data can be applied manually. This section now introduces a concept how discovery could be integrated with the defined Configuration Admin based configuration approach.

The basic idea is to discover a broker based on information about the messaging channel the EDS needs to talk to. In the case of messaging, discovery is explicitly not about the final message receiver as it is for tightly coupled systems like for example Web-Services. In the case of messaging, the client should only be interested in the messaging channel and the message broker which provides this channel.

For the discovery the EDS would need some initial configuration which defines that it should look for a messaging system that provides the required messaging channel. Such a configuration is shown in listing 4.6. Based on these properties a discovery system could try to discover the missing configuration options.

The grouping of the configuration properties allows the EDS implementation to configure the event handling part and to queue events for later delivery while the discovery is still in progress.
The remainder of this section will cover a possible approach for a general integration of discovery with the Event Distribution System and the Configuration Admin.

Integration of Discovery

The previous two sections outlined a configuration mechanism based on the Configuration Admin as the central point to define configuration data for the EDS. The Configuration Admin supports Configuration Plugins which can intercept and modify configuration information before it is delivered to the configuration receiver. Those plugins are realized as services which need to implement the ConfigurationPlugin interface. The CA listens for such services via the service factory and calls the method modifyConfiguration(...) before it delivers the configuration.

A discovery bundle could register a configuration plugin and thereby intercept all configuration data for the EDS. When the bundle finds a configuration block like for example the one shown in listing 4.6 it could try to discover the missing properties. If the discovery succeeds, the bundle could modify the configuration appropriately and return form the callback method. The Configuration Admin would than apply the complete configuration to the EDS.

However it is likely that the discovery a slow process. Therefore it might be better to realize the mechanism slightly different (Figure 4.4). The discovery bundle could do the discovery asynchronously and therefore return from the callback immediately. When it later receives possible results from its discovery attempts it can apply the additional configuration information using the normal Configuration Admin service. Such a solution provides the EDS with the possibility to for example setup the forwarding as far as the configuration is already available\(^1\). Furthermore it is possible for the discovery to change the configuration after a first discovery run when it discovers new information.

The approach of realizing the discovery outside the EDS provides several advantages over the alternative to integrate the functionality directly into the EDS:

- The discovery is completely independent of the concrete EDS implementation.
  This even without the need for the definition of a specialized discovery interface.
- The EDS can be used with or without the discovery and the discovery can be added, changed or removed during runtime without any influence on the EDS.

---

\(^1\) This especially with regard to the previous example for partial configuration updates from page 50.
4.1.4 Event Message Conversion

The previous sections assumed that the EDS somehow converts between events and messages while it forwards them. This section now covers some aspects of this conversion process.

The section will start with a proposal for a message format, continue with a discussion on how messaging related information can be transferred together with the events and concludes with an approach for the needed Java serialization in OSGi.

4.1.4.1 A Suitable Message Type

The selection of a suitable message type is based on the JMS message types. JMS supports several different message types which are shown in table 4.1. The Map Message which contains a simple map of key value pairs as its body has basically the same structure as an event. Therefore a Map Message seems to be the appropriate choice.

However the information stored in an event can basically be mapped to all the supported message types where each of them might be better suited for a particular situation. On the other side the conversion between multiple message formats is the job of a messaging system. Therefore the Map Message seems to be a good choice as a “default” message type as it provides an easy and natural mapping for events.

4.1.4.2 Messaging Related Information

Events have no designated header like a JMS message. However a bundle which communicates with a messaging system via events might need access to the information contained in the header of the original message for example to get or set a correlation id. Therefore a mechanism is needed to transport the header information together with the event.
<table>
<thead>
<tr>
<th>Message Type</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TextMessage</td>
<td>A message which contains a simple string as its body.</td>
</tr>
<tr>
<td>MapMessage</td>
<td>The message body is a set of key value pairs where the key needs to be a string and the value needs to be a serializeable object.</td>
</tr>
<tr>
<td>StreamMessage</td>
<td>The body of this message contains a sequential list of primitive values like int or float.</td>
</tr>
<tr>
<td>ObjectMessage</td>
<td>The message simply contains a serializeable Java object as its body.</td>
</tr>
<tr>
<td>BytesMessage</td>
<td>Such a message contains an array of bytes as payload to for example transfer binary formats.</td>
</tr>
</tbody>
</table>

Table 4.1: The different message types supported by JMS

The header information which can be added to a JMS message is a set of simple key value pairs. Therefore the header elements could also be stored in the body of an event. To separate them from the payload, a prefix like "edsMessageHeader." could be used.

When the EDS receives an event it could look for such properties and map them to the header fields of the message. For the other direction the EDS could simply extract the header information from the message, add the prefix to their names and store them as additional properties in the event.

Besides this possibility the EDS could define an “advanced event” by extending the original event interface from the Event Admin. This new event type could then support an additional set of properties as its header. However in this case a bundle which wants to send or handle such events would explicitly depend on this class and would therefore be unusable when the EDS bundle is not installed. Therefore the other option that uses the plain event with additional properties seems to be the better solution.

4.1.4.3 Serialization and OSGi

The analysis of the different class loader structure of OSGi showed that the deserialization of objects is more complicated than normally. As this especially concerns the EDS which needs to deserialize objects which it might extract from received messages this section will outline two possible approaches for the deserialization.

Dynamic Imports

The Dynamic-Import property can be specified in the bundle manifest to define that a bundle wants to import packages on demand. This means that the import takes place on demand in the active state of the bundle when it is already running. The option allows to specify package names with a wild card to thereby import packages without knowing their exact name.

When a bundle uses dynamic imports, the framework tries to dynamically resolve requests for classes that are not available in the class loader of the bundle. If the package of such a class matches the pattern specified by the dynamic import property
the framework tries to find the package of the class. It uses the set of the exported packages of the other bundles in the framework to find the missing package. When the framework finds a matching export it adds it to the local class loader and the bundle can use this package and therefore the classes from it.

The dynamic import mechanism could be used by the EDS to find and access the needed classes for the deserialization of objects delivered via the messaging system. However this import mechanism has the major disadvantage that it can only add the relation to the foreign package in a static way. Therefore once a package is dynamically included from another bundle, the package can’t be replaced by for example a newer version because the old package can’t be removed from the class loader. The only possibility to access the new version would be to restart the EDS bundle. After the restart, the EDS could import the currently available version of the package.

Additional Configuration

Besides the possibility of the dynamic import mechanism another approach is possible. The configuration provided for the EDS to setup the forwarding from or to the messaging system could be extended by an additional parameter. This parameter could specify one or more bundles which provide the classes that the EDS needs to deserialize objects it receives from the messaging system. Therefore the EDS could search for the specified bundles and access their class loader to deserialize the objects. This would provide a more dynamic approach as the configuration of which bundles should be used as “class source” could be changed during runtime. Therefore this approach is chosen over the dynamic imports approach.

4.2 Resulting Concept

Figure 4.5 shows the complete concept for the EDS and its interactions with the surrounding bundles. The decisions that let to this concept are based on the discussions in the previous sections and they are summarized in the following paragraphs.

The basic messaging functionality is realized on the event sending side by the Event Admin API for topic based non reliable communication (Figure 4.5 Part 1). For topic and queue based reliable communication the extended Event Admin API is provided directly by the EDS (Figure 4.5 Part 2). For the reception of events the concept of the event handler mechanism of the Event Admin is used. Event handler for topic and queue based communication are separated from each other by their service properties.

Guaranteed delivery for the event producer is realized via blocking methods provided in the extended Event Admin API which can throw exceptions to notify the sender of possible failures. To ensure the delivery of events, the EDS will deliver them directly to the appropriate event handlers. Based on this the EDS can acknowledge the reception of messages based on their successful delivery. Durable subscriptions are realized by
additional service properties for the event handler registration. The revocation of such a subscription is also realized by special service properties.

The configuration is based on the chosen Configuration Admin approach (Figure 4.5 Part 3). The EDS provides a managed service factory to receive configuration information from the Configuration Admin. The configuration information can be supplied by any bundle using the normal Configuration Admin. Optionally the discovery can be realized as a standalone bundle which interfaces with the CA. Another optional bundle can provide configuration file based configuration by realizing the Extender Pattern. Some basic configuration options are specified in listing 4.5. The configuration options will be grouped together based on their purpose.

Messaging related information will be stored as additional properties in the events as it provides better compatibility with legacy bundles. To resolve the class loading issue, the usage of an additional configuration property to specify the location of the class definition is chosen over the dynamic import approach. The concrete realization on how the event message translation takes place and which message type is used is considered as implementation specific as it doesn’t change how other bundles use the EDS.

4.2.1 Evaluation

This section will now evaluate the concept regarding its fulfillment of the requirements defined during the analysis.

4.2.1.1 Basic Functionality - R1

The defined concept allows to send and receive messages with topic and queue semantic. It provides the possibility to use multiple messaging systems and to provide the messaging functionality to multiple bundles in parallel. Therefore the basic functionality as defined in the first requirement is fulfilled by the concept.

4.2.1.2 Quality of Service - R2

The concept provides mechanisms to guarantee the message delivery when sending and receiving messages. The used mechanisms are oriented on JMS. However the reliability functionality is only available when sending messages via the extended API. When receiving messages the reliability is always given as the EDS directly handles the delivery.

4.2.1.3 Fit into the OSGi Framework - R3

The integration concept reuses the existing Event Admin and extends its functionality based on the same concepts. Therefore the general concept fits into OSGi and is capable of dealing with the dynamic behavior of the framework.

The concept defines a non invasive approach for the integration as none of the framework components need to be changed. It should therefore be possible to realize this concept in such a way that only a couple of bundles need to be loaded into the framework to provide the remote messaging functionality.
Figure 4.5: Abstract Architecture of the EDS
The realization of the event reception is based on the Whiteboard Pattern. However durable subscriptions break with the concepts of the pattern as they require the EDS to track their life cycle. Therefore the implementation of the EDS is more complex than it would be with the plain Whiteboard Pattern.

4.2.1.4 Portability and Interoperability - R4

The concept itself only uses standard OSGi mechanisms and no implementation specific features of a concrete OSGi container. Therefore an implementation can be created in such a way that it is usable on every standard OSGi container and with every standard implementation of the compendium services. Furthermore the concept doesn’t introduce anything that prevents an implementation for limited environments.

The interoperability with foreign messaging applications should be possible if the implementation uses a standardized message format like the proposed JMS Map Messages.

Additionally the concept completely hides the concrete underlying messaging system and its API.

4.2.1.5Transparent Usage - R5

The concept allows the completely transparent usage of non reliable messaging via the Event Admin API. Additional functionality needs to be used explicitly by obtaining a special extended Event Admin service.

Moreover if a bundle only uses the Event Admin API it can be used with a remote messaging system or with the local event delivery mechanism. The decision if the events are forwarded to a messaging system are not made by the bundle but by the appropriate configuration data and the EDS.

4.2.1.6 Discovery - R6

The concept supports discovery as an additional component that is independent of the concrete EDS implementation. This additionally allows an easy adaption of existing discovery mechanisms to the EDS as they can be developed separately.

4.2.2 Conclusion

The evaluation shows that the extended Event Admin concept is capable of fulfilling all defined requirements. However many of the aspects relevant for the fulfillment of the requirements are specific to the implementation. Therefore the next chapter documents a proof of concept realization. It will show if the key elements of the concept are realizable and if the realization of those concepts is possible in a portable and interoperable manner.
After the definition and evaluation of a concept for the chosen extended Event Admin integration approach this chapter will describe an exemplary realization of this approach. The produced implementation will be used in the next chapter to show how it can be used in a business scenario.

The remainder of this section will describe which functionality of the concept is realized by this implementation. This is followed by a description of the design of the implementation. Based on this, further information regarding the concrete implementation is given which is followed by a short conclusion.

The exemplary implementation only covers a part of the whole concept. In detail it only realizes functionality that is based on the plain Event Admin and doesn’t integrate the functionality of the extended interface. Therefore guaranteed delivery features are limited to message reception as the Event Admin has no support for it. Furthermore no mechanisms for durable subscriptions are implemented.

The configuration is realized as defined in the concept. Therefore the implementation uses the Configuration Admin and defines configuration groups. However the optional discovery features are not realized but the implementation provides simple version of the Extender Pattern based configuration file scanner to show how such optional components integrate with the EDS.

The implementation uses JMS Map Messages and a custom text based message format. The reasons for the two different formats are explained in the following section. Furthermore the implementation doesn’t realize any of the described Java serialization related mechanisms and has no support for any messaging specific parameters attached to the events.

More information on what is supported can be found in the descriptions of the implementation in the following sections.

Due to the cooperation with two industry partners the following additional requirements for this realization are given:

- The implementation must be usable on a mobile device. Therefore the implementation must be compatible with a Java ME CDC/FP
The implementation must be able to communicate with Apache ActiveMQ

5.1 Technology and Design Decisions

This section will now document the chosen technologies and the resulting architecture of the exemplary implementation.

5.1.1 Additional Requirements

Due to the requirement that the EDS has to be able to communicate with an Apache ActiveMQ messaging system the ActiveMQ client libraries need to be used by the implementation. However an analysis of those libraries revealed that they are not usable in the in a JavaME environment. Therefore an alternative solution is used for the limited environments. The EDS will use STOMP, a simple text based messaging protocol, here. The reasons that lead to the decision to use STOMP are documented in appendix B. Based on this decision the EDS has to support two different communication mechanisms:

- An Apache ActiveMQ client library based approach that can be used in a normal Java runtime.
- An STOMP based approach which can be used to communicate from within JavaME.

One possibility for the realization would be to simply implement both mechanisms directly in the EDS. However the STOMP library is not usable via JMS. Therefore the EDS would have explicit dependencies on the STOMP implementation and on JMS even if only one of them is used. This problem can be avoided if the EDS is modularized. Therefore the STOMP and JMS parts could be realized in different bundles and the EDS could communicate with them over a generalized messaging service interface. In this case the EDS would have no dependency on a protocol specific API and the different implementations could be added and removed during runtime. As this approach provides the needed flexibility it is used in the exemplary implementation. The resulting architecture of the EDS is shown in figure 5.1.

As the EDS has to choose one concrete protocol it introduces an additional configuration parameter which determines which protocol should be used. The value of the parameter is then matched to a service property of the messaging service to find the needed implementation.

5.1.2 Bundles

The exemplary implementation is realized in the following bundles:

- **EventDistributionSystem Bundle**
  Implements the EDS functionality as specified by the concept in the previous chapter and provides the Interface definitions for the messaging service. It has no dependency to JMS or STOMP.
5.1 Technology and Design Decisions

Figure 5.1: The general design of the EDS exemplary implementation. The parts realized by this implementation are shown in a darkened color.

- **ActiveMQ Wrapper Bundle**
  Realizes the `MessagingService` interface and provides access to an ActiveMQ broker based on the Apache ActiveMQ client libraries.

- **STOMP Wrapper Bundle**
  Realizes the `MessagingService` interface to provide access to messaging systems based on the STOMP messaging protocol. It uses the STOMP client library bundle.

- **STOMP Client Bundle**
  Provides the actual implementation of the STOMP protocol via a proprietary API.

- **Configuration File Scanner Bundle**
  Scans installed bundles for appropriate configuration files and applies their configuration information via the Configuration Admin to the EDS.

The group of the EDS, the STOMP Wrapper and the STOMP bundles is intended to be usable with JavaME. The combination of the EDS, the ActiveMQ Wrapper with it’s dependencies and the Configuration File Scanner are intended for normal Java runtime environments.

5.1.3 Event Message Conversion

As the implementation is only intended to demonstrate the general principle of the integration approach the conversion process between events and messages is implemented
in a simple manner. It is therefore only supported to serialize and deserialize Java objects that are directly resolvable by the wrapper bundles class loader as the conversion takes place in the wrapper components. Therefore only the basic classes from the “java.” package are available as possible event contents. The possible contents are limited even more for the STOMP wrapper as it uses a very simple text format which is discussed in the next section.

5.1.4 Message Format

The implementation uses two different message formats. This is caused by the fact that the STOMP implementation is not based on JMS and therefore doesn’t support the JMS message types. Due to this the mobile implementation uses a simple text based message format. This format would also be usable by the ActiveMQ wrapper but this part also supports the proposed JMS Map Message which is therefore used instead.

The text format used by the STOMP wrapper is a very simple serialization of the properties of an event. Each property key and the string representation of the property value is concatenated with a separator in between. The resulting property strings are then concatenated and the whole string is prefixed by an capital E to form the complete message. The format is specified in the extended Backus Naur Form in listing 5.1.

Listing 5.1: The syntax of the text based message format in EBNF

```
1   key = Name of the property
2   value = String representation of the property's value
3   seperator = ','

5   property = key seperator value seperator
6   message = 'E' seperator {property}
```

5.2 Implementation

The following sections describe the concrete implementation of the components. All components can be build using Apache Maven. The steps needed to build and deploy them is described in appendix C. The source of the implementation is available on the enclosed CD-Rom in the directory exemplaryImplementation. Besides this the source has been published as project Lightsabre on FuseForge\(^1\).

5.2.1 EDS

The implementation of the EDS is divided into three Java packages as shown in figure 5.2. The contents of the packages are explained in the following paragraphs.

---

\(^1\) http://lightsabre.fusesource.org
The configuration package contains the implementation of the managed service factory. Therefore an instance of this class is registered as a service to receive configuration information from the Configuration Admin.

The internal package contains the classes which implement most of the EDS’s application logic. Each instance of the class ForwardingContext represents one loaded forwarding configuration to mediate between an messaging destination and the internal Event Admin based mechanisms. The EDS can handle multiple forwarding contexts and thereby communicate with different messaging destinations in parallel.

If a context is configured to send events to the messaging system, it references an instance of the class EventSubscriber. This subscriber implements the EventHandler interface from the Event Admin. The instance is registered as a service to receive the events that should be forwarded to the messaging system from the Event Admin. The event subscriber holds a reference to an EventSender object which handles the forwarding of the events to the messaging system.

The realized implementation of the event sender queues the events which it receives from the subscriber and asynchronously sends them to the messaging system. To deliver the event the sender uses an instance of the Connection class from the messaging package. The instance of this class is provided by the used implementation of the messaging service.

If the context is configured to receive events from the messaging system, it references an instance of the DirectEventPublisher class. The class implements theEventListener interface from the messaging package. This instance is registered to a Connection object to receive events from the messaging service. The publisher than delivers the events directly to appropriate event handler services. The implementation only uses one

Figure 5.2: Simplified class diagram of the implementation of the EDS. The colored packages are provided by the OSGi framework.
session for the reception of messages regardless how many event handlers will receive the events. The acknowledgment of a message takes place only if the delivery was successful for all recipients.

The messaging package contains the MessagingService interface and all related interfaces. The messaging service is realized by each of the wrapper bundles and allows the EDS to obtain a new connection to a messaging destination. The needed configuration information for the connection to the messaging destination is given as a parameter to the call. The interface of the MessagingService interface is shown in listing 5.2. The obtained instance of the Connection class allows the EDS to send events as messages to the messaging system and to register EventListener instances to the connection to receive messages in the form of events (Listing 5.3). Therefore the complete conversion between messages and events is realized in the implementation of the messaging service. The expected behavior of the messaging service implementation is documented in section 5.2.2 for the STOMP wrapper implementation. The behavior of the ActiveMQ wrapper is similar.

The EDS implementation allows partial updates of the applied configuration data in a limited form. In particular it is able to setup the event reception even if the configuration of the messaging service is not yet complete. The EDS queues received events in the EventSender class which delivers them as soon as the connection to the messaging system is established.

The EDS implementation is intended to be usable in a JavaME environment. Therefore no Java 5 language features or classes from the Java 5 class library are allowed. However the implementation uses language features like generics and Java 5 classes from the java.concurrent package to ease the development. Consequently the implementation needs to be translated to Java 1.4 and the concurrency classes need to be added to the implementation. This translation is done by the Retrotranslator tool from Taras Puchko. Further information on this process can be found in [40] and in appendix C.2.1.2.

**Listing 5.2: The MessagingService interface defined by the EDS implementation**

```java
public interface MessagingService {
    public String getName();
    public Connection getConnection(Dictionary<String, String> properties);
}
```

**Listing 5.3: The Connection interface defined by the EDS implementation**

```java
public interface Connection {
    void sendEvent(Event e) throws MessagingException;
    void registerEventListener(EventListener el) throws MessagingException;
    void close();
}
```
5.2 Implementation

5.2.2 STOMP Wrapper

The STOMP Wrapper bundle realizes the messaging service interface defined by the EDS. The implementation creates one connection for each unique set of configuration data. Therefore the implementation caches the connections based on the name of the configuration set. Due to this the EDS can request the connection multiple times. The implementation will only close the connection after all references where closed.

Furthermore the STOMP implementation checks the provided configuration data if it is sufficient to create a connection. If that is not the case, the `getConnection(...)` call returns `null`. In this case the EDS will cache all events that it receives until the connection is setup and the events can be delivered. The EDS will try again to get a connection once the configuration data is updated.

The message event translation is realized based on the previously specified text message format. Therefore the object types that can be serialized are limited to objects which have a simple text representation like an integer or a string.

As the STOMP wrapper is based on a general messaging protocol it is usable with every messaging system that supports STOMP. However the wrapper supports one ActiveMQ specific feature which can optionally be used. The supported feature is ActiveMQ’s retroactive consumer subscription\(^1\) which can be activated by a special configuration property\(^2\).

Furthermore the STOMP and the ActiveMQ wrapper both support connections to messaging queues. However the EDS implementation has no support for queue style communication yet. Therefore the wrapper simply adds the possibility to forward an event that was published to the Event Admin to a messaging queue without any special semantic.

5.2.2.1 STOMP Client

The Stomp client bundle realizes the STOMP protocol in a simplified form. It supports the sending of messages and the subscription to topics and queues. Besides this it is implemented in such a way that it requests an acknowledgment for each sent message. However further descriptions are omitted here as the concrete implementation of the protocol is not relevant for the realization of the integration concept.

The implementation of the STOMP wrapper and of the STOMP client both use Java 5 language features. Therefore they both need to be translated to Java 1.4 to be usable with JavaME.

5.2.3 ActiveMQ Wrapper

The ActiveMQ wrapper realizes the same functionality that was already documented for the STOMP wrapper. Therefore a detailed description of its implementation is omitted.

---

1 Retroactive consumers are explained in the next chapter as part of the demo.
2 The support for this feature was added for the demo scenario.
However the event message conversion is based on JMS Map Messages in this case. Therefore all serializable Java objects can be packaged into a message by this wrapper. However the possible object types for the reception of messages is limited to classes directly available to the wrappers class loader.

5.2.4 Configuration File Scanner

The configuration file scanner loads configuration information from active bundles which contain appropriate configuration files. Therefore the scanner registers a synchronous bundle listener at the OSGi framework to be notified about all bundle state changes.

Whenever a bundle enters the active state the scanner looks for property files in the bundles OSGI-INF/messaging/ folder. The properties from each of the found files are then loaded and applied to the Configuration Admin to configure the EDS.

When a bundle leaves the active state the scanner deletes the appropriate configuration data from the Configuration Admin. The configuration file scanner thereby realizes the Extender Pattern as described in appendix A.2.

5.3 Conclusion

The implementation was tested with Apache Felix and Eclipse Equinox. It supports the at the beginning of this chapter described functionality in both containers. Therefore the implementation shows that the core concepts specified for the Event Admin integration approach are easily realizable. Furthermore it shows that the implementation provides the functionality without changing any parts of the OSGi framework or of an existing Event Admin implementation.

The realization shows that the usage of the Configuration Admin provides the possibility to extend the configuration process by for example a configuration file scanner. The basic concept behind partial configuration changes during runtime is also realized by the possibility to apply the configuration for the messaging destination independently form the other configuration information.

Reliability mechanisms are only shown by the implementation in a very limited form. However the implementation shows the concept for the reliable message reception by directly delivering the resulting events to the event handlers and by only acknowledging the reception when the delivery was successful.

The next chapter will use this implementation to show how it can be used in practice to realize a demo scenario where multiple OSGi based subsystems use this implementation to communicate with a foreign system via messaging.
After the conception and exemplary realization of a messaging extension for OSGi, this chapter will show how this newly developed extension can be used in a business scenario.

The scenario will be outlined in the next section followed by a detailed description of the participating systems and their functionality. This description of the components will then be extended by a technical description of their realization in the next section. The chapter concludes with a summary of the cognitions on how the new messaging extension can be used in such a scenario.

6.1 The Flight Information System

The aim of the Flight Information System is to distribute information about the departing aircrafts on an airport. The system will provide information on the flights on the well known flight status displays which can be found on nearly all airports nowadays. In addition to the public displays, the system provides the passengers with the possibility to stay informed on their flight via their mobile phone.

On the backend side the system enables airlines to change information for scheduled flights. Such changes could for example include the changing of the aircraft type if the airline can’t fill the currently scheduled plane and wants to switch to a smaller one.

The system is divided into four subsystems, the flight management system as the central management component, the flight information displays, the mobile clients and a simple web interface for the airlines. All four parts are shown in figure 6.1 together with their interactions which will be explained in the following paragraphs.

- **The Flight Information Manager**

  The Flight Information Manager (FIM) is the central component and dispatches the status information on the flights to all interested clients. It handles changes from the airlines and informs the clients about the results. For this purpose the FIM keeps a list of all flights. Changes are broadcasted via a messaging system to the mobile clients and the information displays. The changes from the airlines are also supplied via a messaging system.
6.2 Realization

This section now covers how the subsystems of the scenario are realized and how they interact with each other in detail. It will also cover if and how the EDS is used by them. Figure 6.2 shows all demo parts with a simplified representation of their internal components and their interactions.

A description how the parts can be build and deployed can be found in appendix C.2. The source code of all components can be found on the enclosed CD-Rom in the directory `airportDemo`.

---

**Figure 6.1:** The basic subsystems in the demo scenario and their relations

- **The Airline Interface**
  The airlines can request changes of the flight information via a simple web application. Furthermore they can schedule new flights. The changes will be transported via a messaging system to the FIM. The FIM then changes the flight information and distributes the changes.

- **The Flight Status Display**
  A Flight Status Display shows the current state of the scheduled flights in a simple list. The display is realized as a simple GUI application which receives information on the flights from a messaging system.

- **The Mobile Client**
  Customers which want to stay informed on their flight wherever they are on the airport can install the mobile client application. The application will inform them on all changes regarding their flight. For this purpose it will provide the user with a simple input possibility where he can enter his flight number. The information on the flights will be distributed to the application by the FIM via a messaging system.
6.2 Realization

6.2.1 Flight Information Manager

The Flight Information Manager is realized as a Java Business Integration (JBI)\(^1\) application which is deployed in an Apache ServiceMix\(^2\). ServiceMix is shipped with an integrated Apache ActiveMQ message broker which is used for the messaging based communication with the other subsystems. The connection between the JBI application and the messaging system is realized by an Apache Camel\(^3\) based JBI component.

The FIM manages a list of all current flights. Changes are received from a messaging queue. The changes are applied to the internal state and the new list of flights is broadcasted via several distinct messaging topics. One topic distributes a complete list of all flights and their states. This topic is used by the Flight Information Displays and the airline interfaces. The other topics are dynamically created for each flight. They are intended for the mobile clients and will be explained together with the mobile clients implementation in section 6.2.4.

---

1 JBI defines a runtime architecture for components which interact over a mediated message exchange with each other. The message transport is realized by a Normalized Message Router (NMR). All communication between the components takes place via this NMR. (cp. \[16\])

2 [http://servicemix.apache.org](http://servicemix.apache.org)

3 Apache Camel is a routing engine which can process messages from ambiguous sources. Camel is often used to realize Enterprise Integration Patterns as they are defined in the book from Gregor Hohpe\[20\]. Further information on Camel can be found in [1].
The FIM is realized by two JBI components: A management bean and a routing component (Figure 6.2). The bean stores the flight list, applies changes and broadcasts the results. All communication takes place via JBI with the routing component. Therefore the bean has no direct contact with the messaging system.

The routing component is based on Apache Camel and mediates between the internal JBI messages and the messaging system. It defines two routes. The first receives change messages from the Flight Changes message queue as JMS Map Messages, converts them into an internal XML format and forwards them to the bean as JBI messages. The second route receives the flight list JBI messages from the management bean. It converts this message into a JMS Map Message and forwards it to the Flight Information messaging topic. Additionally it extracts the changed flight from the list and creates a JMS Text Message which only contains the information on this flight. This message is than published to the appropriate flight specific topic for the mobile clients.

The FIM is the only non OSGi application\(^1\) in this demo as all other parts of the demo are using OSGi. This shows how OSGi applications which use the EDS can communicate with this foreign system using messaging.

6.2.2 Airline Web Interface

The Airline Web Interface is realized as an OSGi application which uses the OSGi HTTP service to provide the web frontend (Figure 6.3). The application communicates changes to the FIM over the Flight changes message queue which is provided by the ActiveMQ Broker of the FIM. Furthermore it subscribes to the Flight Information topic of the broker to display a recent list of the flights in the web interface (Figure 6.2).

The web interface provides the possibility to make changes to existing flights and to schedule new flights. The communication with the messaging system is realized by the EDS and its ActiveMQ binding. The application is deployed in an Apache Felix OSGi container which runs on a Java 6 runtime.

The OSGi HTTP service allows the application to register a servlet. The servlet handles the HTTP requests and generates the appropriate HTML code to provide the user interface. The servlet communicates with a backend service which realizes the application logic of the airline interface. Therefore the service provides the possibility to request flight changes, schedule new flights or to get a recent list of all flights. This backend service hides the communication with the Event Admin from the web interface.

The backend receives events which contain the current list of all flights and caches the list so that the web interface can request them at any time. When the web interface requests a change or a scheduling of a flight, the backend crafts a new event which contains the requested action together with the necessary parameters. The event is than delivered to the Event Admin. As the communication with the messaging system

\(^1\) The statement is not exactly true as Apache ServiceMix which is used as the runtime for the FIM and Camel is based on OSGi. However OSGi only provides the foundation for the higher functions of the ServiceMix and is not directly involved in the realization of the FIM.
6.2 Realization

The web interface of the airline's management application is realized via the EDS, it receives those events and forwards them as messages to the Flight Changes queue on the ActiveMQ broker. The EDS also subscribes to the general Flight Information topic. The received messages are converted into events and delivered to the event handler of the backend service. The EDS is configured by the configuration file scanner which loads the configuration information from files contained in the airline bundle.

6.2.3 Status Display

The flight status display is also realized as an OSGi application. It uses a simple Swing based GUI to display the current status information on the flights (Figure 6.4). The status information is received from the general Flight Information topic. The status display also uses the EDS for the communication with the messaging system which is also configured by the configuration file scanner. The application is deployed in an Eclipse Equinox OSGi container which runs in a Java 6 runtime.

The status display bundle shows a simple window which contains a list of the flights. The list content is obtained from a backend class. The backend receives the events from the EDS and updates its cached flight list so that the GUI can request it at any time. The Observer Pattern\(^1\) is used to notify the GUI about changes in the list.

The airline and the status display application both use a retroactive subscription to always obtain the current state of the flight list as soon as they are started. The concept of a retroactive subscription is described at the end the next section.

6.2.4 The Mobile Client

The mobile client is also developed as an OSGi application. As it is intended to be used on mobile devices it has to cope with their limited Java runtime environments.

---

\(^1\) Further information on the Observer Pattern can be found in [51, Section 7.2.2].
The client is based on Sprint Titan which is a mobile OSGi implementation. Titan is based on a Java 2 ME CDC and supports an embedded version of the Eclipse Rich Client Platform (eRCP) and the Standard Widget Toolkit (eSWT)\(^1\). This demo uses the Titan Stack for Windows Mobile. For the development and the demo the Windows Mobile 6 Device Emulator was used. Furthermore the mobile client also uses the EDS to communicate with the message broker. However it will not use the ActiveMQ binding but instead the STOMP binding.

The configuration of the EDS is done directly via the Configuration Admin. Therefore the application will dynamically set the needed configuration information to connect to the required messaging topic. The mobile client will not subscribe to the global Flight Information topic like the other clients. It will subscribe to a special topic for the flight it is interested in. Therefore the client will only receive changes for his flight which reduces the amount of data that needs to be transferred to each client. The message broker will therefore provide one messaging topic for each flight. The naming of the topics follows a simple schema:

```
/flight_<FlightNumber>
```

Which would be resolved for a flight with the number “LX4711” to the following:
```
/flight_LX4711
```

The client is thereby able to subscribe to the right messaging topic only by knowing the flight number and general connection information for the messaging system. Due to this the client doesn’t need to connect to an additional information source to obtain a list of flights and their related messaging topics.

The mobile application uses the eRCP and eSWT capabilities of Titan to provide a graphical user interface (GUI). The GUI allows the user to enter his flight number into a pop-up dialog. Afterwords the interface shows status information on the entered flight (Figure 6.5).

When the user enters his flight number, the flight number is applied to a backend class. The class triggers an update of the EDS configuration so that the EDS subscribes to the appropriate messaging topic from which it will receive the information on the entered flight.

The backend class also registers an event handler. The EDS delivers the messages from the messaging topic to this handler. The handler than updates the flight information

---

\(^1\) Further Information on Sprint Titan can be found in [44] and [50].
which is cached in the backend class. After this is done the backend notifies the GUI that the information has changed. The notification is once again based on the Observer Pattern.

6.2.4.1 Retroactive Consumer

As the FIM will only send messages to the topics when a change occurred, the mobile client needs a way to obtain the current information on the selected flight when it connects for the first time. This is realized by a special feature of the ActiveMQ broker. The messaging topics are configured with the last image recovery policy and the client subscribes to them as a retroactive subscriber. Due to this configuration a connecting messaging client will always get the last message which was send to the topic. The client can thereby obtain the current state of the flight without the need to communicate with an additional system and even without implementing any additional application logic for the retrieval.

The usage of this configuration also avoids possible problems with lost connections over wireless communication systems as the client will always get the most recent
message when it reconnects\textsuperscript{1}. Further information on ActiveMQ\textquoteleft s concept of retroactive subscriptions and recovery policy\textquoteleft s can be found in [10, 12].

6.3 Reliability

This section will exemplarily show the reliability mechanism realized by the EDS when receiving messages from the messaging system by analyzing the start-up of the Flight Information Display (FID) under two different circumstances.

When the OSGi framework starts, it loads and starts all required bundles for the status application. To ease the descriptions it is assumed that all bundles were successfully started before the bundles of the FID are started. The following paragraph describes the start-up of the FID where the configuration file scanner applies the configuration information and no other configuration is already available from the Configuration Admin.

The EDS configuration for the FID is contained in the application logic bundle of the FID. Once this bundle is started it registers an event handler to receive messages as events. During the start-up the configuration scanner will find the configuration information in the bundle and apply it to the Configuration Admin. The Configuration Admin will then deliver the information to the EDS. Based on the configuration the EDS creates a retroactive subscription for a messaging topic and immediately receives a message with the current flight status. The message is then delivered to the FID backend event handler service as an event.

This would be the normal start-up of the FID. Now the same start-up is described but without the configuration file scanner bundle. Here it is assumed that the configuration information is stored in an persistent storage of the Configuration Admin implementation and is therefore available immediately after the start-up of the admin. Additionally it is still assumed that all other bundles start before the FID bundles.

Once the framework starts the EDS and the Configuration Admin, the CA will apply the configuration information to the EDS (Listing 6.1 lines 1-8). The EDS configures itself as described and receives the first message. Now it has to deliver the message to the event handler but the FID bundles are not yet started. Therefore the event handler is not available. The normal Event Admin would now discard the event and the message would be lost. However the EDS only acknowledges a message to the messaging system when the resulting event was successfully delivered. Consequently the message will not be acknowledged in this case and the messaging system will retry the delivery (Lines 10-12). Once the FID bundles are started and the handler is registered the EDS can deliver the event and therefore deliver the message (Line 14-18).

\textsuperscript{1} This would also be possible with a durable subscription but such a subscription has a mayor drawback. If the connection is lost the broker will store all missed messages until the subscriber connects again. Then all missed messages would be delivered. This also means that the broker needs to keep track of the messages for each subscription. This is especially a problem if the mobile clients don\textquoteleft t end their connection when they are leaving the airport. The broker would have no chance to know if the client has only connection problems or if it left.
Consequently the implemented mechanisms for reliable message delivery work as expected.

Listing 6.1: Log output of the EDS when the event delivery fails

```java
EDS: Configuration updated: /* properties omitted */
EDS: **** created new ForwardingContext: [FlightInfo]
EDS: updateContext
EDS: DIRECTION: receive
EDS: creating EventPublisher
ActiveMQ: creating new connection
com.fusesource.lightsabre.activemqWrapper.ActiveMQConnection<init> INFO: new connection established
ActiveMQ: creating consumer
/*...*/
ActiveMQ: Delivery of the event failed -> won't acknowledge the message
/*...*/
ActiveMQ: Delivery of the event failed -> won't acknowledge the message

/* The Flight information display bundles start */
/*...*/
EDS: Directly delivering Event for Topic [flightstatus] to handler of bundle FlightStatusDisplay
/* The event handler of the Flight Information Display receives the event */
Handle Event -> org.osgi.service.event.Event [topic=
flightstatus]
```

6.4 Conclusion

The demo shows that the EDS can be used on the sending and receiving side of a messaging system as well as in normal Java environments and embedded or limited devices. It also showed that the implementation is usable on three different OSGi implementations Equinox, Felix and Spring Titan. The separation of the protocol specific messaging client implementation from the messaging client makes it easy to use a different communication protocol on the mobile device than in the normal Java environments.

The different message formats don’t pose a problem as a conversion between them is easily possible as shown by the Camel component of the FIM. Furthermore the interaction with the non FIM shows that the communication with foreign systems is possible.

Due to the reduced implementation of the EDS the reliability mechanisms are only available for the reception of messages. However as shown in section 6.3 these mecha-
nisms work as expected. The fact that no reliability mechanisms are available in the implementation when sending messages are only relevant for the airline interface where the change messages might be lost if the EDS is not available or is unable to send them.

A summarizing evaluation of the results of the analysis, the conception, the realization and this demo can be found in chapter 8 Evaluation but first the next chapter gives a quick introduction into some projects and publications that are related to this thesis.
CHAPTER 7

Related Work

This chapter gives a quick overview over some related projects and publications.

The research for related work showed that much effort was made in the area of synchronous service based remoting for OSGi and less attempts were made so far to integrate asynchronous communication. However the following sections present some publications and projects which provide messaging functionality within OSGi or are directly related to the concepts developed in this thesis.

7.1 Eclipse Communication Framework

The Eclipse Communication Framework (ECF) [18] provides a communication infrastructure to ease and generalize the usage of different communication paradigms and protocols. ECF supports point-to-point and publish/subscribe communication and features several protocols from HTTP to XMPP and many client API’s including an Apache ActiveMQ client. It is based on OSGi and provides besides its own proprietary API an implementation of the distributed OSGi specification.

ECF is intended to be used for Eclipse rich client applications or Eclipse plug-ins. However it is not limited to the Eclipse platform and can therefore also be used as a communication framework for other OSGi based applications.

7.1.1 ECF’s Distributed Event Admin Service

Since June 2009 the ECF community provides an example implementation of a simple distributed Event Admin based on their communication framework1. The implementation replaces the normal Event Admin with a custom implementation which forwards events between other remote Event Admins (Figure 7.1). Due to ECF the implementation can be used with various protocols like ActiveMQ or XMPP.

However the implementation is limited to the Event Admin API and therefore only provides limited access to the messaging functionality and its reliability mechanisms as already documented for the conception of the extended Event Admin approach in

---

1 This implementation was released after the implementation developed in this thesis was made public on a Fuse-Forge project page.
chapter 4. A short documentation on their implementation and further information can be found in [17].

![Distributed EventAdmin Service](Image Source [17])

**Figure 7.1:** The distributed Event Admin provided by ECF (Image Source [17])

### 7.2 Spring JMS

The Spring framework is a Java framework that provides a unified API layer above several common Java API's. Additionally it supports configuration based dependency injection and mechanisms for aspect oriented programming (cp. [6, Page 1-2]).

One of the API’s that are supported by Spring is JMS. Spring provides a simplified API to use JMS to communicate with messaging systems. It introduces a Template Pattern\(^1\) based approach to use the functionality of JMS. The approach hides the differences between JMS 1.0.1 and the current JMS 1.1 API. Furthermore it provides the same API for point-to-point and publish / subscribe messaging and allows the messaging related configuration to take place outside of the program code.

Spring’s JMS integration is also usable from within OSGi. Thereby it supports a Spring specific integration of asynchronous messaging in OSGi. Further information on Spring and its JMS integration can be found in [6] and in [43].

---

\(^1\) The Template Pattern uses the concept of a template object which takes care of the resource handling. The template for example handles the creation of a connection to the messaging destination and closes the connection and related resources when they are not needed anymore. While the connection is established the template object executes the code that needs those resources. This code is normally provided by an object that implements a special interface and is assigned to the template object. (Based on [6, Page 118-120])
7.3 R-OSGi

The R-OSGi research project\(^1\) is founded by the ETH Zurich. “R-OSGi allows a centralized OSGi application to be transparently distributed at service boundaries by using proxies.” [41, Page 5] Therefore R-OSGi uses a similar approach like distributed OSGi for the distribution of OSGi services.

Besides the general possibility for remote service invocations, R-OSGi features a distribution of the Event Admin. R-OSGi uses a normal “local” implementation of the Event Admin. To distribute events across the participating OSGi containers the event handler services are distributed. Therefore each event handler service registration from each OSGi container is exported to all other containers. Due to this each local Event Admin sees all registered event handler services of the whole distributed system and delivers its events to all of them (cp. [41, Page 12]).

Consequently R-OSGi only offers a distributed Event Admin without the aim to introduce asynchronous communication based on messaging systems.

7.4 Apache Project Proposal “Aries”

The Aries project was recently proposed to become a project of the Apache Software Foundation. The aim of the project is to provide implementations of the current and future OSGi EEG specifications. Furthermore the project aims to develop additional extensions which are not covered by the OSGi standard. One particular aim is to realize “Message-driven Blueprint components” [21]. However a comparison with the concept developed in this thesis is not possible yet as no detailed information on the integration approach is published so far.

Further information on the project can be found in the project proposal [21].

7.5 Proxy Based Mobile Messaging

The implementation of the EDS in this thesis facilitates STOMP for direct communication with the messaging system. [27] introduces a proxy based concept for the communication between the mobile client and the enterprise system. The introduced proxy component converts “mobile” messages to “enterprise” messages and vice versa. A mobile message is a simple lightweight message which is transferred via HTTP between the proxy and the mobile client. Enterprise messages are JMS messages. Due to this approach a specialized messaging mechanism can be used for the mobile client but the communication with the enterprise system takes place with the usual mechanisms.

\(^1\) http://r osgi.sourceforge.net/
CHAPTER 8

Evaluation

This chapter will shortly summarize the results of the different evaluations of the concept, the realization and its usage in the scope of the demo scenario. The summary is based on the fulfillment of the initially defined requirements.

• **R1 – Basic Functionality**
  The defined concept for the realization of the extended Event Admin integration approach supports the full functionality that was specified in the first requirement. The realization of the approach supports most of the basic functionality and the demo application in chapter 6 demonstrated how it can be used. The only unimplemented functionality is queue style communication.

• **R2 – Quality of Service**
  Guaranteed delivery is fully supported by the defined concept on the event sending side via an extended API and on the receiving side via the normal event handler mechanism. Other QoS aspects are not covered as they are explicitly excluded.
  The realization only supports reliable message reception as the extended API is not implemented. Furthermore durable subscriptions are not implemented. The demo scenario showed that the reliable message delivery concept works as expected.

• **R3 – Fit into the OSGi Framework**
  The developed concept fits well in OSGi as it reuses existing concepts wherever possible. All parts of the concept where developed in such a way that they can handle the dynamic behavior of the framework. The realization of the concept showed that an implementation is possible in a non invasive manner. Therefore the messaging functionality can be added to an OSGi container without the need to change the container itself.

• **R4 – Portability and Interoperability**
  The concept only uses standardized OSGi functionalities and therefore doesn’t depend on any specific implementation of the OSGi standard. The realization of the concept produced an implementation which should be usable with every standard OSGi container. The implementation was successfully used in the demo
scenario with Apache Felix, Eclipse Equinox and Sprint Titan. Furthermore the
demo showed that the concept and its exemplary implementation is usable in the
limited JavaME CDC environment.

• **R5 – Transparent Usage**
The selected combination of the Event Admin and the messaging service approach
allowed the development of a concept that reuses an existing OSGi API to provide
basic messaging functionality in a transparent way. This of course with the
drawback that not all messaging features are available, like for example the in
this thesis covered guaranteed delivery. This missing extended functionality is
supported by an additional API. This makes a bundle which only uses the Event
Admin API usable with local event based messaging and with a remote messaging
system. The usage of the implementation of the concept in the demo scenario
showed that the remote messaging can be used in a transparent manner as all
of the components only used the known mechanisms of the Event Admin to
communicate with the messaging system.

• **R6 – Discovery**
Discovery mechanisms are fully supported by the developed concept as an optional
extension to the configuration process. The loose coupling of the discovery
provides the possibility to realize discovery mechanisms independent from the
Event Distribution System. However the realization of the concept didn’t cover
an implementation of a discovery mechanism.

The evaluation shows that the concept fulfills all requirements. Key elements of the
concept have been realized and where successfully tested in the scope the demo scenario.
However several other parts of the concept have not been tested yet. The next chapter
concludes the thesis and it will thereby mention some still open points.
Chapter 9

Conclusion

The OSGi Framework is emerging in the enterprise market and therefore needs to support common features of enterprise systems. The OSGi Enterprise Expert Group is working on appropriate extensions. Recently Distributed OSGi was introduced which integrates transparent synchronous service based remote communication. The new standard explicitly doesn’t cover asynchronous message based remote communication. This thesis developed and evaluated the extended Event Admin approach for the integration of asynchronous messaging into OSGi to fill this gap.

To find possible integration approaches, different possibilities for OSGi internal and external communication were analyzed. Based on those cognitions four integration approaches were presented and evaluated with regard to previously defined requirements.

From the presented integration approaches, two approaches could be clearly eliminated. Each of the two resulting approaches has its strength in a different area of the requirements. The Messaging Service approach has better support for reliability and the basic messaging mechanisms but has no support for transparent usage. The Event Admin approach features a transparent integration but has only limited support for the basic messaging and reliability mechanisms. Therefore a combination of these two approaches seemed to be capable of fulfilling all defined requirements. This combination, the so called extended Event Admin approach was therefore selected for further evaluation.

Based on this decision the key concepts for a possible realization of the extended Event Admin approach where deducted from several realization alternatives. The resulting concept used the API of the OSGi Event Admin to support a seamless integration of basic messaging functionalities. An extended API was specified as an addition to the Event Admin API support the full messaging functionality including reliability mechanisms. The evaluation of the results of this combined approach showed that it is able to fulfill all of the defined requirements.

The developed concept was used to create a simplified realization of the extended Event Admin approach. The realization features the basic non reliable message sending via the Event Admin and the reliable message reception via direct delivery to the event handler services. Thereby the implementation showed that the key concepts behind the approach are realizable and that they provide the expected functionality. The
implementation was than used to realize a demo scenario to show that the approach is usable in a business scenario.

The demo scenario realizes a flight information system for an airport which distributes status information on departing flights to information displays and mobile clients. The scenario combined the new OSGi messaging extension with a non OSGi management application. Furthermore it showed that the implementation of the extended Event Admin approach is usable with various different OSGi framework implementations and in the resource constrained environment of mobile devices.

The final evaluation of the results from the different parts of the thesis showed that the initially defined requirements where met or could be met when the implementation of the concept is extended appropriately.

9.1 Open Points and Outlook

A promising approach for the integration of asynchronous messaging in OSGi was presented in this thesis. However the approach only covered the core messaging functionality. Therefore many aspects like for example transactions, security or performance where left open. Consequently those aspects provide topics for further research in this area.

The realization of the developed integration concept only covers some key elements of the whole concept. Furthermore the implementation was done as a proof of concept implementation and has therefore for example only limited exception handling. Consequently the implementation needs to be extended appropriately if it shall be used in a productive environment. The implementation was made publicly available as an open source project\(^1\) to feature such future developments.

Results produced by this thesis where presented at the OSGi Developer Conference Europe 2009 [5] in Zurich which resulted in positive feedback. Furthermore the results will be used by the Enterprise Expert Group as input for their ongoing standardization efforts. At the time of this writing the EEG has already started with the integration process of asynchronous communication into OSGi by gathering requirements. With regard to this, developments in the area of asynchronous communication mechanisms for OSGi can be expected in the near future.

\(^1\) http://lightsabre.fusesource.org
APPENDIX A

Commonly used OSGi Patterns

This appendix quickly introduces two OSGi related patterns which are often used by the OSGi framework itself and which are also used for the realization of the extended Event Admin approach.

A.1 The Whiteboard Pattern

The Whiteboard Pattern provides an OSGi oriented realization of the functionality of the Listener Pattern.

In the Listener Pattern, multiple event listeners register them self at an event source. The event source has to keep track of all registrations and notifies them when an event occurs. However this pattern is hard to realize in a dynamic environment such as OSGi where bundles can appear and disappear all the time. Therefore the event source and the event listener would have to track the status of each other to detect if one of the components has been removed or changed.

The Whiteboard Pattern provides the same functionality as the Listener Pattern but in a way that eases the handling of the dynamic behavior of OSGi. The pattern uses the OSGi service registry to keep track of all registered listeners. Therefore it defines that listeners register them self as a service with the service registry. An event producer will than use the service registry to find appropriate listeners once it has an event to deliver (Figure A.1).

Due to this concept the event producer doesn’t have to keep track of the listeners and the listeners don’t even have to know the event producer. Thereby the producer and consumer implementation is rather simple as most of the work is done by the OSGi framework.

The descriptions on the Whiteboard Pattern in this section are based on [34] where also further information can be found.
A.2 Extender Pattern

The Extender Pattern provides a concept for automatic configuration of bundles based on the activation and removal of other bundles.

OSGi provides a synchronous bundle listener by which a bundle can be informed about events raised for other bundles. Such events include the start-up and the stopping of a bundle. Therefore a registered handler can synchronously\textsuperscript{1} scan the manifest or other files contained in the starting bundle. Based on gathered information the listening bundle can configure itself or other components. As the listener is also notified when a bundle is stopped or removed it can revert the configuration applied based on this bundle.

Due to this mechanism OSGi applications can adapt automatically if other components are installed into the runtime. For example could an Extender Pattern based HTTP-Service bundle scan for configuration data in a newly installed bundle which defines that one of the bundle’s classes is designated to be registered as a servlet. In this case the HTTP-Service implementation could automatically register the servlet and unregister it when the bundle is removed.

The descriptions are based on [22] where also further information regarding the Extender Pattern can be found.

\textsuperscript{1} In this case synchronously means that the start-up of the bundle is stopped until the callback of the synchronous bundle listener returns. Further information on synchronous bundle listeners can be found in the OSGi specification [35, Chapter 4].
The realization of the Event Distribution System in chapter 5 has to provide access to a messaging system from within the limited environment of JavaME. Therefore this chapter discusses some messaging protocols and implementations to determine if they are usable with JavaME. The protocols addressed in this chapter are limited to protocols supported by Apache ActiveMQ as the implementation is intended to be used with it. However this section is also not covering all protocols usable with ActiveMQ, a full list of supported protocols is available in [8].

Whenever concrete components or capabilities of ActiveMQ are mentioned in this chapter, they relate to the at the time of this writing current ActiveMQ version 5.2.

B.1 Protocols

ActiveMQ supports beside others the following communication protocols which are very briefly evaluated here: OpenWire, STOMP, REST and AMQP.

OpenWire

OpenWire is ActiveMQ’s native binary protocol. It is implemented by the ActiveMQ client libraries. The specification of the protocol is publicly available.

The ActiveMQ client libraries require Java 1.5. Therefore the client libraries are not directly usable in a JavaME environment. A translation of the Java byte code from version 1.5 to version 1.4 is possible. However tests showed that the client uses classes and methods which are only available in the Java 5 class library.

An own implementation of OpenWire would be possible. However the protocol is rather complex. Therefore even a basic implementation of OpenWire for the JavaME environment seems not to make sense in the scope of this thesis.

STOMP

STOMP is a simple text based messaging protocol. It is specially designed to be easy to implement and implementations are available for many programming languages
and platforms (cp. [49]). ActiveMQ has full support for STOMP (cp. [11]). Further information on STOMP and its capabilities can be found in [49].

Gozirra is an extremely lightweight STOMP client implementation. Tests showed that it is usable within JavaME.

Further information on STOMP and Gozirra can be found in the following section B.2.

REST

“ActiveMQ implements a REST-ful API to messaging which allows any web capable device to publish or consume messages using a regular HTTP POST or GET.” [9] Therefore the mobile client could easily communicate with the messaging system via simple HTTP requests. However ActiveMQ’s REST API only supports to consume messages via a pull mechanism. Therefore the mobile client would have to periodically ask the message broker for new messages (cp. [9]).

AMQP

The Advanced Message Queuing Protocol (AMQP) is an approach to standardize an open wire level messaging protocol (cp.[52]). ActiveMQ has limited support for AMQP build in. However at the time of this writing the development of the AMQP protocol support in ActiveMQ was stopped (cp. [7]).

B.1.1 Selection of a Protocol

The ActiveMQ client libraries are not usable within a JavaME environment as they have dependencies to unsupported classes. Besides this the client libraries are relatively big and therefore don’t suite the limited environments of mobile devices.

The AMQP provides a promising approach for a standardized messaging protocol. However the support of the protocol by ActiveMQ is at the moment of this writing still limited.

REST and STOMP are both suited for limited environments as both are lightweight approaches. However REST has some disadvantages compared to STOMP regarding the reception of messages. Therefore STOMP is chosen as the protocol for the implementation.

B.2 The STOMP Implementation

The Gozirra STOMP implementation is usable with JavaME. However it has no support for message acknowledgments even though they are supported by STOMP. Furthermore it has dependencies to special authentication related Java API’s that are only available

1 http://www.germane-software.com/software/Java/Gozirra/
as an optional extension to JavaME CDC\(^1\). Consequently an own Java implementation of STOMP was created as part of the implementation of the Event Distribution System. The new STOMP library supports to send and receive text messages and implements message acknowledgments. Furthermore it has no special external dependencies except for classes that are available in a JavaME CDC.

\(^1\) In particular it has dependencies to classes from the package \texttt{javax.security.auth}. 
This appendix describes the necessary steps to build and deploy the implementation of the EDS and the components of the demo scenario.

The following descriptions for the deployment require basic knowledge on the usage of the Eclipse Equinox or the Apache Felix OSGi runtime. Therefore knowledge on how to install, start and stop bundles in the selected runtime is required to follow the descriptions.

C.1 Exemplary Implementation

C.1.1 Build

The source code of the implementation from chapter 5 can be found in the directory exemplaryImplementation on the enclosed CD-Rom. The complete source can be build using Apache Maven which will automatically download all dependencies. The commands shown in the following listing build the complete implementation with all sub-projects:

```
cd exemplaryImplementation
mvn clean install
```

After the successful build, the bundles of the different sub-projects can be found in the `target` directory of each sub-project. A description of the different bundles can be found in chapter 5.

C.1.2 Deployment

The directory Distribution/ConfigurationFiles/target/ contains the four configuration files listed in table C.1 after a successful build. Each of the configuration files loads all needed bundles of the EDS and their dependencies into the appropriate OSGi container. Afterwards the EDS can be used in this container. To use one of the configuration files, its contents need to be appended to the configuration file of the used OSGi framework.

Additional documentation on the usage of the EDS can be found in [26] which contains a getting started guide for the usage of the EDS.
C. Build and Deployment

<table>
<thead>
<tr>
<th>File Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>equinox_activemq.config.ini.append</td>
<td>Loads all needed bundles of the EDS with its ActiveMQ wrapper into an Eclipse Equinox container.</td>
</tr>
<tr>
<td>equinox_stomp.config.ini.append</td>
<td>Same as the previous only with the STOMP wrapper.</td>
</tr>
<tr>
<td>felix_activemq.config.properties.append</td>
<td>Loads all needed bundles of the EDS with its ActiveMQ wrapper into an Apache Felix container.</td>
</tr>
<tr>
<td>felix_stomp.config.properties.append</td>
<td>Same as the previous only with the STOMP wrapper.</td>
</tr>
</tbody>
</table>

**Table C.1:** Automatically created configuration files for the deployment of the EDS.

C.2 Demo Scenario

C.2.1 Build

The source-code of the demo application from chapter 6 can be found in the directory `airportDemo` on the enclosed CD-Rom. All components except the mobile client can be build with Apache Maven. The mobile client is provided as an Eclipse project.

C.2.1.1 Flight Information System, Airline Interface and Flight Status Display

The build of the flight information system, the airline interface and the status display is based on Maven. Therefore these components can be build with the following commands issued in the `airportDemo` directory:

```
cd airportDemo
mvn clean install
```

After the successful build the created bundles and JBI components can be found in their appropriate target directories.

C.2.1.2 Mobile Client

The mobile client is provided as an Eclipse project in the folder `airportDemo/MobileClient`. To build the project with Eclipse, the IDE needs to contain the Sprint Titan Developer Tools. This tool set provides the necessary class libraries and the possibility to bundle and install the project on the mobile device. The installation and usage of this tool set is described in [45].

Conversion of the EDS for the Mobile Client

For the deployment of the mobile client the EDS and its STOMP wrapper need to be converted to Java 1.4. The conversion needs to be done for the following three bundles:

```
EventDistributionSystem-0.5-SNAPSHOT.jar
StompWrapper-0.5-SNAPSHOT.jar
StompClient-0.5-SNAPSHOT.jar
```
The conversion of the listed bundles can be done with the Retrotranslator\(^1\). For the demo the version 1.2.9 of the translator was used. The conversion of a bundle is done with the following command:

```
java -jar retrotranslator-transformer-1.2.9.jar -srcjar <jarFileToTransform>
```

After the conversion the bundles have dependencies to the following two JAR files which are a part of the Retrotranslator distribution:

```
retrotranslator-runtime-1.2.9.jar
backport-util-concurrent-3.1.jar
```

To make the libraries available to the bundles, they need to be added to each of the bundles jar files. Furthermore the bundle class path in the bundle manifest needs to be extended appropriately:

```
Bundle-ClassPath: .,retrotranslator-runtime-1.2.9.jar,backport-util-concurrent-3.1.jar
```

After those steps are made the EDS with its STOMP wrapper can be used in a JavaME runtime.

C.2.2 Deployment

C.2.2.1 Flight Information System

During the build a ZIP archive named fim-sa-1.0.0.zip was created in the FIM/camel-sa/target folder. This file contains all the necessary components and configuration information for the FIM. To deploy this file into an Apache ServiceMix 4.x it needs to be copied into the `<ServiceMix>/deploy` folder. Once ServiceMix is started, the file will be automatically installed. Information on how to obtain a current version of Apache ServiceMix and how to start it can be found in [3].

C.2.2.2 Airline Interface

The following description is based on the deployment using Apache Felix. However each other OSGi 4.1 runtime is also usable.

To deploy the airline interface, an Apache Felix runtime with the EDS and its ActiveMQ wrapper is used. The runtime can be setup as described in section C.1.2. Besides the bundles for the EDS, an implementation of the OSGi HTTP service is needed. The Demo was tested with version 1.0.1 of the Apache Felix implementation which can be installed with the following command in the Felix runtime:

```
```

---

\(^1\) [http://retrotranslator.sf.net](http://retrotranslator.sf.net)
To configure the Felix HTTP service implementation to use another port than port 80, the following configuration setting needs to be added to the Felix configuration file. Afterwords the framework has to be restarted:

```
org.osgi.service.http.port=8090
```

After the Runtime is configured the following two bundles of the airline interface have to be installed and started:

```
Airline/FlightManagerWeb/target/FlightManagerWeb-1.0.0.jar
Airline/FlightManager/target/FlightManager-1.0.0.jar
```

After both bundles are installed and started, the web interface is reachable via http://localhost:8090/fim/index.

C.2.2.3 Flight Status Display

The status display consists of a single bundle:

```
StatusDisplay/target/statusDisplay-1.0-SNAPSHOT.jar
```

The bundle needs to be deployed in a Felix or Equinox runtime together with the EDS and its ActiveMQ binding. After the start-up of the flight status display bundle a Swing window is opened by the bundle which will show the list of flights. No additional configuration is needed.

C.2.2.4 Mobile Client

The mobile client application can be deployed in a Sprint Titan runtime together with the EDS and its STOMP binding. In addition the following deployment package needs to be installed in the runtime which relaxes the security restrictions for unsigned bundles:

```
airportDemo/MobileClient/com.prosyst.expand.untrusted_sp.dp
```

For the demo realization the Windows Mobile 6 emulator together with the Titan runtime for Windows Mobile was used. Detailed information on the deployment and the installation process of the Titan runtime on a Windows Mobile emulator can be fond in [45]. Once the mobile client is deployed, the application can be started from the menu entry Other > FlightInfo.
Bibliography


