

Technische Universität Darmstadt

Department of Electrical Engineering and Information Technology
Department of Computer Science (Adjunct Professor)
Multimedia Communications Lab
Prof. Dr.-Ing. Ralf Steinmetz

Design and Implementation of a Service-oriented, Dynamic and Modular P2P-based Multimedia Online Community Platform using OSGi

Studienarbeit

Submitted by

Luciana Mara de Carvalho Alvite

on
15. October 2008

Advisor: Prof. Dr.-Ing. Ralf Steinmetz

Tutor: Kálmán György Graffi

KOM-S-0279

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 15. Oktober 2008 .....

Contents

Ehrenwörtliche Erklärung **iii**

1 Introduction **1**
- 1.1 Online Communities . . . 2
  - 1.1.1 History . . . 3
  - 1.1.2 Characteristics . . . 3
  - 1.1.3 Maintenance costs . . . 5
- 1.2 Peer-to-Peer Networks . . . 5
- 1.3 LifeSocial.KOM . . . 6
- 1.4 Motivation . . . 9
- 1.5 Objectives . . . 10
- 1.6 Overview . . . 10

2 Requirements Analysis **11**
- 2.1 Analysis of the LifeSocial.KOM Architecture . . . 12
- 2.2 Modular Software Design . . . 13
  - 2.2.1 JAR files . . . 14
- 2.3 Dynamic module interaction . . . 16

3 The OSGi Framework **19**
- 3.1 Introduction . . . 19
- 3.2 Practical Uses . . . 20
  - 3.2.1 The Eclipse IDE . . . 20
  - 3.2.2 BMW 5er Series . . . 20
- 3.3 Advantages . . . 21
- 3.4 OSGi technology . . . 21
- 3.5 Bundles . . . 22
  - 3.5.1 The Manifest File . . . 24

3.6 OSGi: a service oriented framework 25
3.6.1 Dynamic interation between modules - Declarative Services . . . 26
3.6.2 Dependency Resolution 27
3.7 Implementations 28
3.8 Summary 29

4 Architecture Design **31**
4.1 Tools 31
4.2 Bundle activation 32
4.3 Modules 34
4.4 Commands architecture 39
4.5 Extension mechanism 41
4.5.1 Example: a Blog Plug-in 42

5 Conclusions **45**
5.1 Conclusion 45
5.2 Future Work 47

Appendices **53**

A List of LifeSocial commands **53**

B Starting the platform **57**
B.1 Test mode 57
B.2 Live mode 58

List of Figures

1.1 Daily Reach MySpace and Facebook 2
1.2 Facebook functionalities 4
1.3 Overview of the LifeSocial.KOM architecture 7
1.4 Overview of LifeSocial.KOM plug-ins 8

2.1 JAR dependencies 16
2.2 Linkage error example 16

3.1 OSGi dependencies 22
3.2 OSGi Layers 23
3.3 Simple example of information sharing in OSGi 24
3.4 Services in OSGi 25
3.5 Example of declarative services in OSGi 27
3.6 Component activation in OSGi 28

4.1 Plug-in activation 32
4.2 Package dependency in LifeSocial.KOM 35
4.3 Service biding in LifeSocial.KOM 36
4.4 Plug-in dependencies in LifeSocial.KOM 38
4.5 Example of the command architecture 39
4.6 Example of command activation in LifeSocial.KOM 40
4.7 User interface of the MySpace Blog 42

5.1 Cyclic dependency in OSGi 48

List of Tables

3.1 Comparison of OSGi Implementations 28

A.1 Network commands 53
A.2 Login Plugin commands 53
A.3 Profile Plugin commands 54
A.4 Friends Plugin commands 54
A.5 Groups Plugin commands 55
A.6 Live Chat Plugin commands 55
A.7 Messaging Plugin commands 56
A.8 Search Plugin commands 56
A.9 Photos Plugin commands 56

Listings

2.1 Excerpt from the current LifeSocial.KOM implementation 13
3.1 Manifest File 24
4.1 LifeSocial.KOM User Interface 37

Chapter 1

Introduction

Overview

1.1 Online Communities 2
1.2 Peer-to-Peer Networks 5
1.3 LifeSocial.KOM 6
1.4 Motivation 9
1.5 Objectives 10
1.6 Overview 10

In a world where people are spending an increasingly higher amount of time online and assuming (real or fictional) online identities, the mimic of real world alike entities and services in the virtual world becomes a natural process. Humans always had the need to socialize and to feel a sense of belonging. For this reason people join groups with similar interests where they can meet like-minded peers. This basic human need is reflected nowadays in different forms on the virtual world, including online communities. In this context we can say online communities mirror club/associations in the real world. Even before the Internet became popular, chat rooms were one of main services offered by systems that allowed users to go online (e.g. *BBSes*[1]), together with games and download and upload of software. The foundation for what would later revolutionize the world was being built.

Moreover, with reallocation of people worldwide becoming more and more common nowadays, the need for non-expensive and efficient ways of maintaining contact with family and friends located miles away becomes important. Online communities offer

[1] Bulletin Board Systems, are considered the precursors of the World Wide Web. A BBS was a system users could connect to using a modem to download and upload software, play games and read news. Some also offered IRC chat rooms and other features. Because users could only connect to the system using a phone line, the users were mainly concentrated in the same region

the possibility of keeping in touch with several people at the same time through built-in functionalities like "moods", "bulletins", "blog" posts, songs you are currently listening to and other forms of informing the world about things happening in your life. This surely eases the task of keeping in touch with several people and makes friends and family feel they are closer to the user in ways that an E-mail, for example, can not. A quick visit to their profiles is enough to keep users updated on their friend's lives and "multicast" events, changes and even new habits in their own lives.

1.1 Online Communities

Online communities (also known as *online social networks* [2]) have become ubiquitous in the Internet world. Sites like MySpace[MyS03], Facebook[Fac04], Orkut[ork04], Friendster[Fri02], Xing[Xin03], StudiVZ[stu05], etc are among the most popular sites on the web [Ale08a]. Figure 1.1 shows that MySpace and Facebook have each a user reach varying from 6% to 11%, what is quite a lot considering the immense amount of sites available on the web.

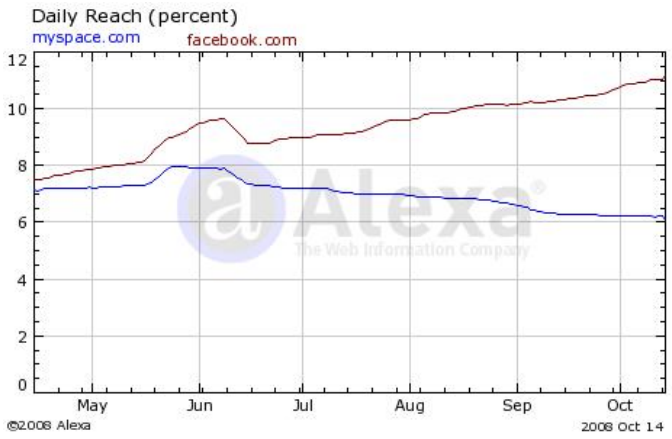


Figure 1.1: Daily Reach MySpace and Facebook. source: Alexa.com

The reasons for the popularity of online communities are, like we mentioned in Section 1, various: keeping in touch with old friends, finding like-minded people all over the world, expressing themselves, finding new friends, exchanging ideas, etc. The audience for these services is varied and nowadays even secret American spies have their own

[2] We use the terms *online communities* and *online social networks* interchangeably throughout this text

online community. The Office of the Director of National Intelligence [DNI08] recently launched A-Space [Sha08], a platform for intelligence community agents to exchange ideas, strategies, plans, videos and other top secret relevant information.

1.1.1 History

We can see from the beginnings of some of the most popular social networks that the focuses and user base of the online socialization vary.

classmates.com [cla95] is considered the first online community ever created. The main purpose of this social network is to help users keeping in touch with old school colleagues. classmates.com is still the 3rd most visited online community in the USA as of May 2008 [Cha08].

Facebook started as a network of students from some of the best universities in the USA (Harvard and other Ivy League universities) as a platform for communication, similar to the German-based **StudiVZ**. Some years later the network was opened to everyone with an E-Mail address and it grew quickly from there, breaking the 63 Million users mark [Swa08].
Today, Facebook is the most popular social network in countries like Canada [Ale08b] and the UK [Ale08c] and second in the USA [Ale08d] and its members base include people with all kinds of educational background and lifestyles, including musicians, artists and Hollywood celebrities [Phi07].

MySpace was created in 2003 with a more commercial focus and, after initially becoming popular with independent music bands and their fan base (MySpace provides special profiles and tools that make it attractive for musicians to advertise their work), achieved mass popularity in a relatively short time. Nowadays, MySpace has more than 110 million users worldwide and has more than 24 regional versions, with content aimed at the specific region [Swa08].

1.1.2 Characteristics

The list of social network platforms is endless. But some of the main characteristics all these platforms share and that make them interesting for the everyday Internet user are:

- Free of charge

- Friendly interface

- Diverse features and interaction capabilities - blogs, Instant-messaging, photos, videos and, more recently, externally developed applications through an open API [Tom08] [Fac07] (*Plug-Ins*)

- Customization - users are free to personalize their profile to more accurately represent their online identity

These features make these platforms extremely popular with different groups of people. Online communities have users of all age groups, ethnicities, cultural backgrounds, lifestyles, etc. In fact, a quick browsing through the profiles and other services offered by social networks shows the heterogeneity of its users. One can find anything from well designed profiles with several gadgets to very simple profiles of people who are more interested on other services like blogging or groups. Figure 1.2 exemplifies this. While some users might intensively share pictures and use external applications to keep in touch with their families, others might never use such services and prefer to send group messages and share their thoughts in groups or blogs.

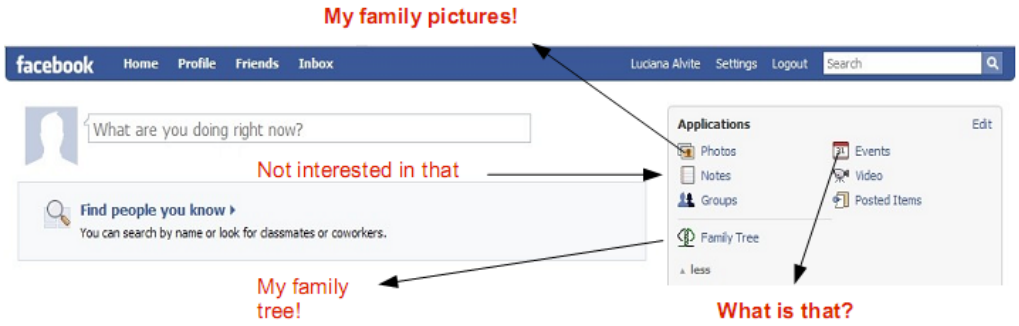


Figure 1.2: Facebook functionalities

It is important for the platform to be able to cope with this diversity and offer users a comprehensive range of services to make the community attractive to as many people as possible. On the other hand, it is important that the platform remains lightweight and provides a straightforward friendly user interface. Furthermore, since the functionalities every member uses every time they access the platform will probably be different, there is no need to load all the services at start-up. Therefore the platform should be able to activate services dynamically on demand.

1.1.3 Maintenance costs

The costs of maintaining online communities are substantial. Myspace for example handles more than 40 billion pages views a month [Car07], and the possibility of adding features like videos and song playlists to a profile certainly adds a considerable amount of traffic to its servers. Rich Miller has recently mentioned on his article "Infrastructure Costs Nearly Killed MySpace"[Mil07]:

> While the site (MySpace) has faced constant data center challenges due to its rapid growth, the biggest budget buster was probably bandwidth, since MySpace has always swapped large amounts of video and image files, especially with YouTube and Photobucket.

The expenses involved such traffic intensive services certainly make providers of such platforms look for ways of reducing costs and at the same time speed up loading times and provide a better user experience. The incorporation of Equinix's [Equ98] peering capabilities in its own video content distribution architecture [Mil06a] clearly shows how important traffic management is to a social network of MySpace's size.

In fact, Myspace has recently announced [Hig08] it will be using *Gears*[3] [Goo07] to optimize its search and email functionalities and at the same time decrease server requests.

These attempts made by online community providers to reduce maintenance costs and also increase availability clearly shows that

1.2 Peer-to-Peer Networks

Peer-to-Peer networks have gained extreme popularity in the past years and have become the object of intense discussion involving various industry segments. Peer-to-Peer applications account for a big part of Internet traffic nowadays and the tendency is for this number to grow. A report [BS08] made by the Discovery Institute [TT94] estimates that by 2015 video downloading and Peer-to-Peer file sharing applications will be responsible for 100 exabytes [4] of traffic only in the USA. Peer-to-Peer networks are the basis for different kinds of applications, including (legal or illegal) sharing of files[Coh01], communication[S.A03] or video streaming[Inc05], etc. We assume the reader is familiar with Peer-to-Peer networks concepts throughout this text. More information about Peer-to-Peer systems can be obtained at [SE05].

[3] Gears is an open source browser extension that lets developers create web applications that can run offline and allows applications to have access to the user's own CPU and storage, which are then used to act as a local server, to cache and provide application resources (HTML, JavaScript, images, etc.)

[4] 1 exabyte = 10^18 bytes

As we discussed in Section 1.1.3, the costs for maintaining an online community platform are high and applying Peer-to-Peer technologies can be an efficient way of reducing these costs, since every peer will be contributing resources such as storage space and computing power to the platform. Another advantage of employing Peer-to-Peer technologies to an online community is related to availability. Since there is no central server anymore, it becomes harder for the platform to be severely affected by power shortages or peak usage problems observed before[Mil06b].

1.3 LifeSocial.KOM

We at KOM are currently developing a new generation of online social networks called LifeSocial.KOM. The basic idea is to combine online communities and Peer-to-Peer networks to build a resilient robust scalable platform that is able to provide the functionalities found in popular online communities at lower costs. Built on top of an overlay Peer-to-Peer network, LifeSocial.KOM allows every user to contribute with a part of its resources to the network, therefore decreasing the costs for the platform provider, as we saw in Section 1.2. Services are offered to the user in the form of an extendible plug-in architecture, hence allowing developers to add new functionalities to the platform.

A short explanation of the architecture is presented next, but it gives by no means all the details of the design, since they are out of scope of this work and can be found elsewhere. It is strongly recommended that the reader is familiar with the architecture of the LifeSocial.KOM before proceeding. More information about LifeSocial.KOM can be found at [Pod08] or [GPM+08]. Figure 1.3 illustrates the architecture of the platform and Figure 1.4 gives an overview of the available plug-ins.

- **Information Cache**
  We can see in Figure 1.3 that all plug-ins communicate only with the *Information Cache* component and other plug-ins. The Information Cache is designed to act, as the name suggests, as a cache for data requested by plug-ins. In the LifeSocial.KOM architecture, the plug-ins are pro-active, meaning they actively check if the information they requested has already arrived, instead of waiting to be informed about the arrival. If the data is not yet available, the plug-in is then free to do other things while waiting. This avoiding blocking and gives the plug-in the freedom to decide what data is still relevant. Imagine we want to display a profile for an user. Requesting it directly from the network and waiting for the results to be avail-

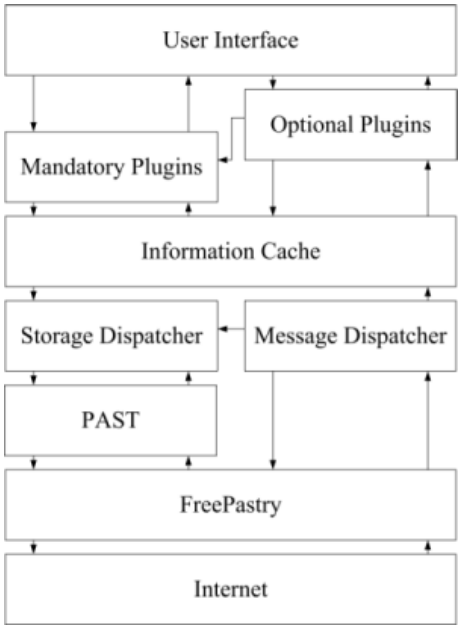


Figure 1.3: Overview of the LifeSocial.KOM architecture

able is not the best method for doing it, since it can take some time for the profile to arrive and the plug-in will be blocked during this time, when it could be performing other operations. The LifeSocial.KOM architecture avoids this situation by giving plug-ins the freedom to decide frequently polling the Information Cache to check for data arrival. Now imagine that before the profile arrives, the user decides to load another profile instead. The previously requested profile is not relevant for this plug-in anymore so the plug-in can simply stop polling the Information Cache for this data and focus on the newly requested profile. The plug-in will not be informed when the old request finally arrives and if the data is not accessed anymore it will be then eventually automatically discarded by the cache.
Another advantage of having a cache is that data is concentrated in one place and when a different plug-in needs the same data that was already requested by another plug-in (such as login data, profile information, etc), no new subsequent requests are made, since the data is already available in the cache. This avoids the generation of extra traffic for data objects that are already available locally.

Other components of the network are the *Storage Dispatcher* and the *Message Dispatcher*. We briefly discuss the functionalities of these components next.

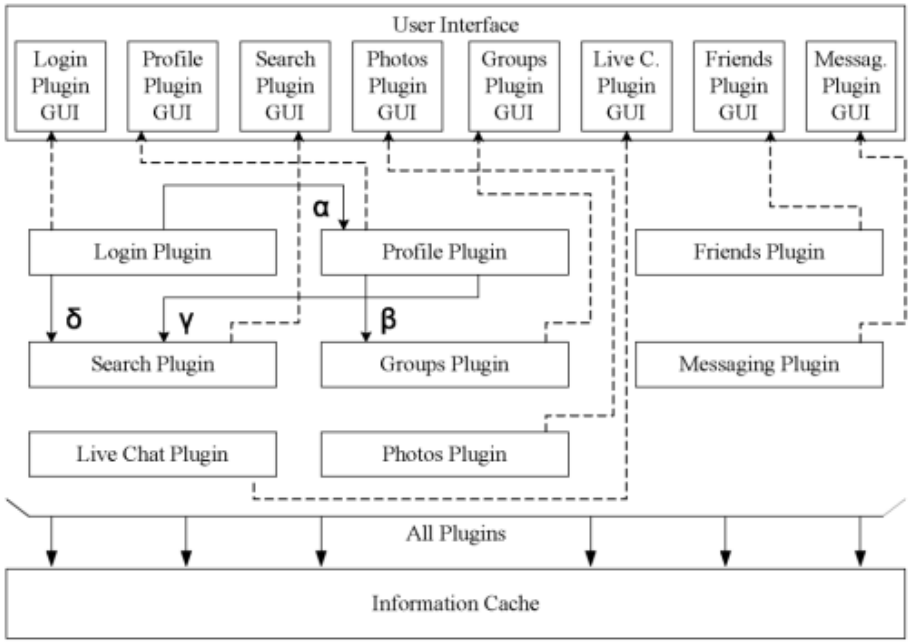


Figure 1.4: Overview of LifeSocial.KOM plug-ins

- **Message Dispatcher**
  The Message Dispatcher component is responsible for delivering user-to-user messages and is used by plug-ins that provide some form of communication service to the users, such as *Messaging* and *Live Chat*. It supports the offline delivery of messages by storing them in the Peer-to-Peer network when the recipient of the message is not online at the moment the message is sent. It also supports message history, by storing an object associated with the user name and all the messages associated with the user. The storage of objects in the network is done by the *Storage Dispatcher* component.

- **Storage Dispatcher**
  The Storage Dispatcher component is responsible for storing data objects in the Peer-to-Peer network. Since there is no central server, it is primordial that a reliable distributed mechanism is employed to be able to store data objects and to make sure data is not lost. For this purpose we use PAST[DR01], [RD01b], which is a large-scale, peer-to-peer archival storage facility built on top of FreePastry [Frec],

[RD01a]. The Storage Dispatcher interface allows other components to store, retrieve and delete items from the network. So lets think of an example like we mentioned when describing the Message Dispatcher, where the user is offline when a certain message is sent. In this situation, the Message Dispatcher from the sender of the message is informed of that and sends a request for the Storage Dispatcher to store the message in the network. The recipient will then be informed that new messages are available on the next login and is then able to read the messages sent while he/she was offline. Other plug-ins (e.g. Photos, Profile, etc) use exclusively the Storage Dispatcher for information retrieval, since they do not offer user-to-user communication services.

The current architecture of the platform is analyzed in Section 2.1.

1.4 Motivation

In a system like LifeSocial.KOM, it is important that changes in the environment (e.g. new versions of applications, introduction of new functionalities, changes in application states, etc) are dealt with in a robust manner to provide flexibility and a better user experience. This requires a system that dynamically reacts to changes and adapts to new conditions at runtime. Moreover, taking into consideration the variety of interests of its member base, the system should be able to provide only the services actually being used at any time and avoiding loading modules, saving memory and reducing the start-up time. Finally, to achieve extendibility, it is fundamental that we have a modular system with well defined interfaces and strong separation of concerns that allows the targeted development of plug-ins.

As we saw in the previous Section, the current architecture of LifeSocial.KOM successfully employs Peer-to-Peer technologies to reduce the costs associated with the maintenance of an online community. However, the architecture of the platform does not allow the incorporation of new functionality at runtime. In order to change the set of plug-ins available, the platform has to be recompilated and restarted. This clearly is not a desired characteristic, since this would mean that for every change the users would have to download a new version of the application. Considering the features desired in an online community as discussed in Section 1.1.2, it is expected that new applications will be added to the platform on a regular basis. It is therefore primordial that the system can be easily updated and extended. We can say that the present extension mechanism of LifeSocial.KOM does not meet the these expectations. Therefore a redesign of the architecture is necessary to comply with these requirements arises.

1.5 Objectives

We can summarize the objectives of this work as:

- Analysis of the technologies currently available that support modularization and the dynamic interaction between modules
- Design of the new LifeSocial.KOM architecture based on these technologies
- Implementation of the new architecture, including the adaptation of all current LifeSocial.KOM functionalities, including all its plug-ins
- Testing of the implementation under several scenarios

1.6 Overview

In Chapter 2 we analyze the current architecture of the LifeSocial.KOM platform, discuss mechanisms that can be used to achieve our objectives and highlight the requirements of our platform. In Chapter 3 we introduce the OSGi Framework, which provides the desired extendibility requirements and graciously takes care of the intrinsic dynamic present in a platform like LifeSocial.KOM. In Chapter 4 we present the implemented architecture based on the OSGi framework and elaborate on other design choices made. In Chapter 5 we summarize our contribution to the LifeSocial.KOM project and present relevant future work that could be done in the area.

Chapter 2

Requirements Analysis

Overview

2.1 Analysis of the LifeSocial.KOM Architecture 12

2.2 Modular Software Design 13

2.3 Dynamic module interaction 16

In this chapter we discuss the requirements for a dynamic modular P2P based online community.

We saw in Section 1.1.2 some of the characteristics that make online communities attractive to a wide range of users. These characteristic can be translated to the following requirements for an online community:

- Usability
- Flexibility
- Extendibility

It is primordial that an online community offers an overall positive user experience. This include fast response times, a straightforward friendly user interface and a considerable collection of applications that can be started on demand to satisfy the needs of its diverse member base. The latter can be achieved by providing an easy extendible platform that allows developers to add new functionality(*plug-ins*). Extendibility requires a modular architecture with strong separations of concerns and a straightforward interface that

gives developers the freedom to develop new plug-ins in an easy manner. It is also fundamental that new functionality can be added to the system on the fly, without requiring recompilations and/or restarts of the platform.

2.1 Analysis of the LifeSocial.KOM Architecture

The current architecture of LifeSocial.KOM has the following characteristics:

- Distribution as one single .jar [1] file - this provide no modularization and makes the development of new plug-ins a hard task, since a developer does not have a standard interface for plug-in registration and to declare dependency on services offered by other components of the architecture. Furthermore, the addition of new plug-ins require changes in the main system, as it can be seen in 2.1. If we consider that several plug-ins are going to be being developed simultaneously, plug-in integration could easily become a nightmare

- Static plug-in integration - the set of available plug-ins is defined at compile time, which makes it impossible to integrate new plug-ins on demand after the platform has been started. The introduction of new plug-ins at runtime is not supported. Change in plug-ins require recompile, redistribution and restart of the platform.

- All available plug-ins are loaded at start-up - once new plug-ins have been added to the platform, they are all loaded when the platform is started. This wastes resources and increases start-up time, contributing to a negative user experience.

- No support for dynamic adaptability - changes in the environment(e.g. plug-ins suddenly becoming unavailable) is not foreseen. It is assumed that all components of the system are available at all times, what might not reflect reality or be desired. This can cause unexpected behavior.

Listing 2.1 shows how plug-ins are incorporated into the system in the current LifeSocial.KOM architecture.

We can see that the introduction of new plug-ins would need require the platform provider to change the main system to add the new plug-ins (assuming the main system is not available to developers) by creating a new plug-in instance at compile time and registering it with an internal plug-in registry. Now lets imagine a real situation where new applications are launched every day. This extension architecture is not flexible

[1] Java ARchive

```
public class MainSystem {
  public static void main(String[] args) {
    LoginPlugin login = new LoginPlugin();
    ProfilePlugin profile = new  ProfilePlugin();
    (...)
    registerPlugin(login);
    registerPlugin(profile);
    (....)
}
```

Listing 2.1: Excerpt from the current LifeSocial.KOM implementation

enough and clearly not suitable to be used in the dynamic context of an online community.

As we can see, the present structure of the LifeSocial.KOM platform clearly does not meet the requirements we have defined for our online community. Therefore we need to look for new mechanisms that can be employed to achieve a more flexible and dynamic system and redesign the architecture using these mechanism. We discuss some of the options currently available to help us design our new architecture in the following sections.

2.2 Modular Software Design

Modularity is an important characteristic of systems designed to be extensible, reusable and maintainable. It stimulates a strong separation of concerns, information hiding and decoupling, some characteristics that are highly desired in Software Development. Modules are pieces of software that together provide some kind of functionality. They usually define an interface which is exposed to other modules and hide the internal implementation details and information that should not be available to others. Ideally, a module should not have to care at all about other modules, but to achieve this in large system would mean having one big module that encapsulates all the system functionalities (so it does not depend on other modules). This would though make software extension and maintenance a difficult task, since a more granular separation of concerns is inexistent at the system level (everything is inside the module). The key is to find a compromise when choosing what to modularize so that a substantial level of decoupling is achieved, while at the same time allowing targeted development and a relatively easy extendible architecture.

By having a modular architecture, with each module providing a well defined interface that executes a certain task, it becomes easier to extend the system, since developers of

new modules only need to know how to interact with a certain module it needs and does not have to care about its internal details.

We can summarize the desired characteristics for a module as:

- **High Cohesion** A module should perform one well defined task and not have multiple unrelated responsibilities. High cohesion facilitates the reusability of modules and makes modules more reliable and understandable.

- **Low Coupling** A module should not have to care about internal details of other modules it depends on. The interaction between modules should be realized through a well defined interface. This stimulates *Information Hiding*.

- **Unity** A module should be seen as whole. One must able to deploy, install, change and update a module as one single unity. The internal parts of the modules should not be used separately.

Java, being an Object-oriented programming language, intrinsically offers modularity in the *class* level, by encapsulating methods and variables that are highly related to each other. But an application is usually obviously not made by only a few class or even packages. It is usually made of several groups of classes and packages that are responsible for a providing some kind of function. Therefore we need to think about modularization in the system level and how to encapsulate classes and packages that together provide some functionality. In LifeSocial.KOM, a plug-in, for example, implements its features using several packages and classes, but also needs to use functions provided outside the scope of its own classes, such as communicating with the Information Cache and with other plug-ins. So we could think of a plug-in as a module that uses functions from other modules.

In Java, the standard unit of deployment are .JAR files [SM04]. .JAR files aggregate into one single file classes that together provide some kind of functionality and a MANIFEST file that provides meta information that determine how the file is used. We analyze the deployment of .JARs as a solution for modular design in the next section.

2.2.1 JAR files

A possible solution to achieve for modularization could be to split the project into different .JAR files. So we could for example have separate .JARs for each plug-in, for the Storage Dispatcher, the Message Dispatcher, etc.
The problems related with this approach are the following:

- **Flat Classpath**
  The MANIFEST of a .JAR file might determine its dependencies using the *Class-Path* header. Once all the .JARs of a system are loaded, the classpath becomes then flat, and the internal boundaries between .JARs is essentially forgotten. One of the problems of this is when 2 libraries have classes with the same name. Inside one module this can be avoided by defining a naming schema. But when we talk about a system composed of several modules, there is no way of determining the naming schema used by external libraries and naming conflicts might occur.

- **No mechanism for version dependency**
  The MANIFEST file does not provide any way of specifying dependency version information. Lets imagine a situation like the one depicted in Figure 2.1, where an application is dependent on 2 other libraries (X.jar and Y.jar) that themselves depend on a Z.jar library. X expects and has been tested against version 1.1 of the Z library and Y expects version 1.2. If we add both version of Z to the classpath, the one coming first will be used and the other one will be ignored. In this case, we assume version 1.1 was declared first in the classpath, so only the classes with different names from version 1.2 would be loaded. Now imagine the situation depicted in Figure 2.2 where Y manages to use the class Foo.class from version 1.2, itself is dependent on class Bar.class. Since version 1.1. was first in the classpath, it shadows the classes with the same name from version 1.2, so Foo.class will be using Bar.class from version 1.1. This would most likely cause a *LinkageError*.

- **No support for runtime dependency resolution**
  When loading a .JAR, all its dependencies must be available at that moment, or a *ClassNotFoundException* will be thrown. It is not possible to resolve the dependencies after the module is loaded.

- **No information hiding**
  In Java, classes can be declared either *public* or *default*. The *default* modifier makes the class visible only to other classes in the same package. But in a module we usually have several packages that must use classes from other packages, thus classes have to be declared public. But declaring a class public would mean that other modules would also have access to it, thus breaking the *Information Hiding* requirement.

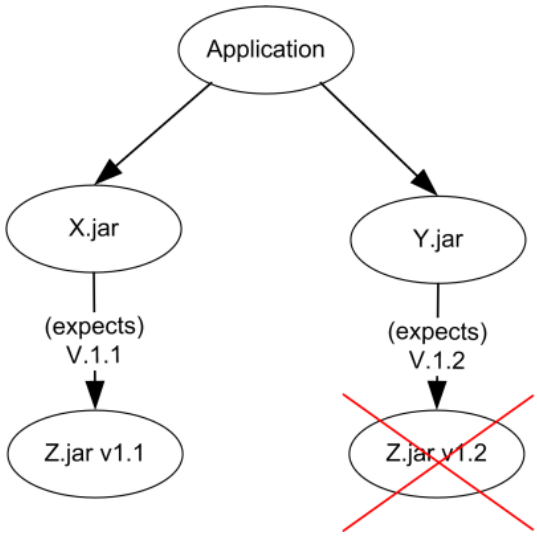


Figure 2.1: JAR dependencies

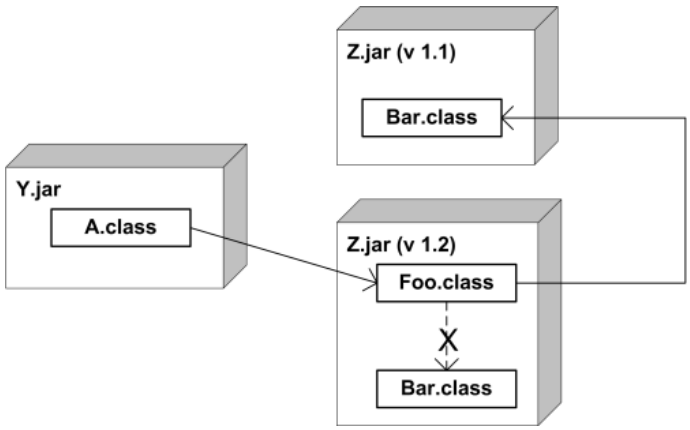


Figure 2.2: Linkage error example

2.3 Dynamic module interaction

It is important that a platform like LifeSocial.KOM can dynamically adapt to the changes in the environment (e.g. changes in the states of modules, new module versions, new functionalities, etc). It is clear that in an online community platform changes in environment have to be dealt with dynamically and without requiring a restart of the application.

In the next Chapter we present a framework that allows us to fulfill these requirements.

Chapter 3

The OSGi Framework

Overview

3.1 Introduction **19**
3.2 Practical Uses **20**
3.3 Advantages **21**
3.4 OSGi technology **21**
3.5 Bundles **22**
3.6 OSGi: a service oriented framework **25**
3.7 Implementations **28**
3.8 Summary **29**

3.1 Introduction

Initially aimed at the home automation environment [Kri07], the non-profit *OSGi Alliance* was founded in March 1999 [All99] and has grown substantially since then. Nowadays application areas that use the OSGi technology range from Mobile Phones to Integrated Development Environments (IDEs). The OSGi Alliance is formed by several big players like IBM Corporation[Cor89] [1], Sun Microsystems Inc. [SM], Red Hat[RH95], NEC Corporation[Cor99], Hitachi[Hit10], Électricité de France[ldF46], Samsung Electronics [Gro38], Ericsson[Eri76], Nokia[Cor65], Motorola[Inc28] and Telefónica [dEnS24]. The diversity of industry segments represented in the alliance indicate how broad are the areas the framework finds application to. As German full members we have the Deutsche Telekom[Gro96], ProSyst Software GmbH[Gmb97], SAP AG[AG72], Siemens

[1] Marks are trademarks of their respective companies

AG[AG47] and Software AG[AG69]. ProSyst Software GmbH is a German company entirely dedicated to providing OSGi-based solutions to areas that vary from vehicle Telematics to Home Automation. The purpose of the alliance is to specify a framework for the development of inter-operable, multi-platform, extendible and manageable applications and services based on its component integration platform.

The OSGi Alliance provides the specifications of the framework (currently on its 4th release), but it does not provide a specific implementation of it. There is a core specification [All07]that defines the standard services and components of the framework and expert group specifications, such as for the mobile environment, that focus on the needs, characteristics and limitations (such as computing power) of the specific environment.

3.2 Practical Uses

Although the OSGi is not (yet) extremely popular, there are relevant examples of applications developed using the framework. In this section we present some of these applications.

3.2.1 The Eclipse IDE

Since its version 3.0, the Eclipse (cite) IDE Runtime is based on the OSGi Framework (cite http://www.eclipse.org/osgi/). Eclipse was developed on top of *Equinox* [Fou03], which itself was also developed by the Eclipse Foundation as an implementation of the OSGi specifications. Equinox complies with the core OSGi Service Platform Release 4 [All07] and its latest stable release is 3.4 [Fou08]. In Eclipse, everything [2] is a Plug-in and they interact with the Runtime through a well defined interface. The Plug-in mechanism of Eclipse gives the developer flexibility when designing new plug-ins for the platform. This means that everything in Eclipse is created the same way and can be activated or deactivated as needed, without influencing other features. In the case of Plug-In dependencies, they are resolved automatically and the required Plug-Ins are started as needed. We use those ideas, which are some of the main ideas of OSGi, in our work to make LifeSocial.KOM an easy extendible, dynamic and robust platform that can be accessed from different devices.

3.2.2 BMW 5er Series

Since 2003 the new BMW 5-series and 6-series employs OSGi to provide in-car Entertainment, Communication, GPS Navigation, Telematics , Car Configuration and Climate

[2] except for a small kernel

Control functionalities.

3.3 Advantages

The OSGi Technology is compatible with our goal because it supports the development of applications with the following characteristics:

- Modular Architecture
- Well Defined Interface
- Straightforward Extendibility
- Service-oriented Architecture
- Robustness in Dynamic Environments
- Support for Hot Deployment
- Automatic Dependency Resolution
- Application Life Cycle Management

We discuss these characteristics in the next Sections.

3.4 OSGi technology

The OSGi technology basically specifies a dynamic module system for Java. It is a service oriented framework and is intended for platform-independent, high available, dynamic systems that must deal with changes in the environment in am automatic and robust way. In OSGi, every module has its own private class-path and class loader, thus solving the versioning dependency problem presented in Section 2.2.1, as it is depicted in figure 3.1. Using OSGi, modules X and Y could declare a dependency on an specific version of module Z and, since they have private class-paths, the simultaneous use of different version of the same module is possible.

The functionality of the framework is divided in the following layers:

- Module Layer - defines the class loading policies and the controlled sharing of information between modules.

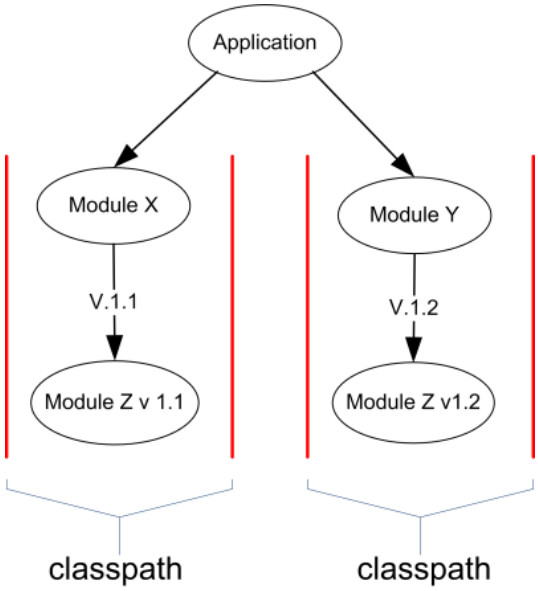


Figure 3.1: OSGi dependencies

- Life Cycle Layer - responsible for dealing with the dynamic interaction between modules. Modules can be dynamically installed, started, stopped, updated and uninstalled and changes are reflected in other modules at runtime.
- Service Layer - provides a *Service Registry* that allows modules to communication through services.
- Security Layer - the service layer is based on the Java 2 security architecture and provides mechanisms for the deployment of applications in security aware environments. It defines for example mechanisms for the signing of .JAR files.

Figure 3.2 gives an overview of the OSGi layers. We elaborate on bundles in the next Section.

3.5 Bundles

The main unit of deployment in the OSGi is the *Bundle*. A bundle is basically a .JAR file comprising the .class files and other resources like a *Manifest File* that contain basic information about the bundle together with meta data describing the its dependencies and packages it exports. All packages which are not explicitly exported are private and other

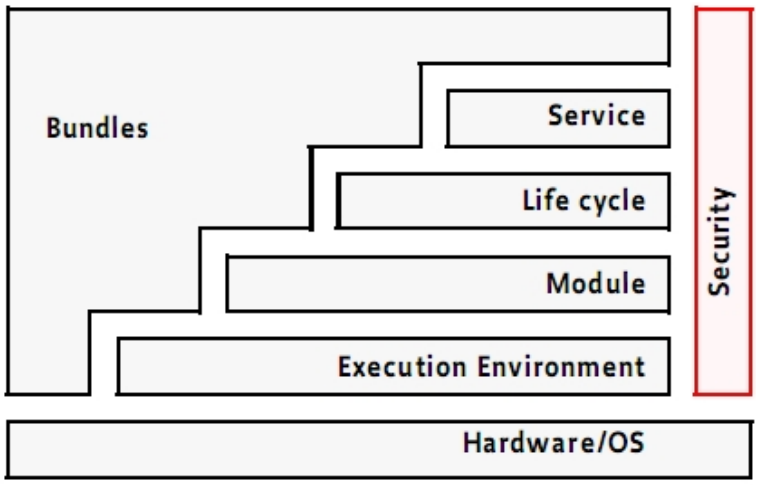


Figure 3.2: OSGi Layers

bundles have no access to them. This allows bundles to hide implementation details and export only an API to other bundles. Additionally, bundles can include version information for itself and for each of its package as meta data, so that other bundles can use this information to further specify the dependency as we saw in Section 3.4. Figure 3.3 shows the interaction between bundles in the module level. As we can see, the Foo bundles exports a package named foo.api by using the *Export-Package* header in the MANIFEST file and the Bar bundle imports this package using the *Import-Package* header. The developer of the Bar bundle can then program against the imported interface. Notice that the package *foo.impl* is **not** exported, meaning the Bar bundle has no access to the details of the actual implementation of the API defined in the *foo.api* package.

We get into more details about the structure and properties of manifest files in Section 3.5.1. These are some of the characteristic we have defined as desired in our platform, as seen in Chapter 2. The dependencies between Bundles are managed automatically by the framework. We elaborate on dependency resolution in Section 3.6.2.

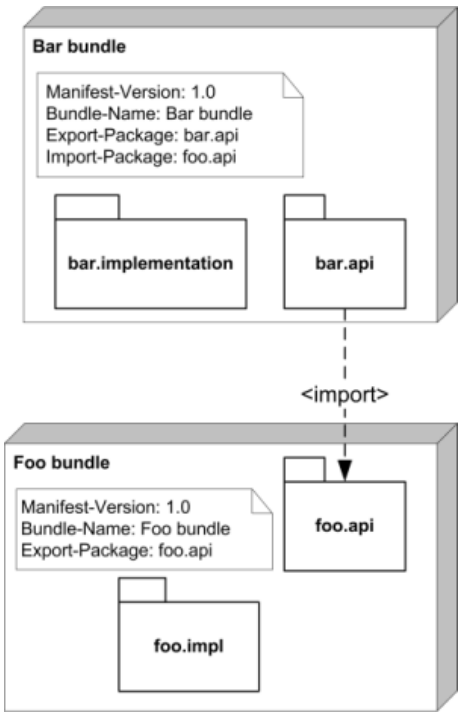


Figure 3.3: Simple example of information sharing in OSGi

3.5.1 The Manifest File

Every bundle must contain a Manifest File in its .JAR File. The Manifest File contains information about the bundle, like name and description, an unique identifier, version number and required packages and bundles. Listing 3.1 shows how a typical Manifest file looks like:

```
Bundle-Description: Profile Plugin
Bundle-Version: 1.0.0
Bundle-SymbolicName: de.tud.kom.lifesocial.profileplugin
Export-Package: de.tud.kom.lifesocial.profileplugin.api
Import-Package: de.tud.kom.lifesocial.loginplugin.api;version="1.0.0"
Require-Bundle: Utilities;bundle-version="1.0.0"
```

Listing 3.1: Manifest File

A bundle is uniquely identified within the OSGi Service Platform through its symbolic name and version. It is important to notice the difference between the *Import-Package* and the *Require-Bundle* headers. Be careful when using the *Require-Bundle* header, since it

declares a dependency on the whole bundle, so all the packages exported by that bundle (*Utilities*, in this case) will be imported. This is not always desired. Sometimes we are only using one functionality of the module and importing all its packages is not necessary.

3.6 OSGi: a service oriented framework

The OSGi is a service oriented framework, which means that usually bundles offer services to the platform by registering them with the *Service Registry* when the bundle is started. Once the service is registered, other bundles are able to request those services by consulting the Service Registry. This mechanism is illustrated in Figure 3.4.
The bundles offering the services are only activated once a request for the service is made. The advantage of this approach is that unused bundles are not loaded, therefore saving memory and resources.

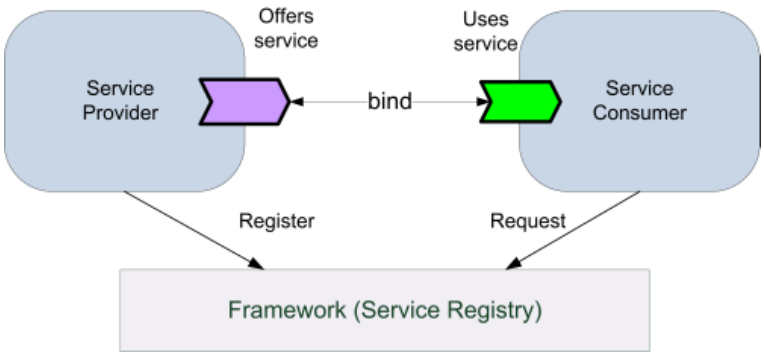


Figure 3.4: Services in OSGi

Bundles offer services in the form of an interface that clients can then request to have access to the methods offered by the service. The internal implementation of the service is transparent for the client, so when using different implementations of the same service the client should notice no difference in the form it interacts with it. The interface is the same for both services. Bundles can also use filters to specify extra information about services they are interested on.
The OSGi Framework defines a number of standard services implementations applications. Some examples are:

- Log Service - Logging features

- HTTP Server - Provides a web server that supports Servlets and static files
- User Admin - Support for user authentication
- Configuration Management - Support for real time configuration management
- Wiring Admin - Grouping ("wiring") of services in functional blocks

3.6.1 Dynamic interation between modules - Declarative Services

In a dynamic environment like a Peer-to-Peer network, it is unpredictable when, if and for how long services are going to be available. Therefore it is important for the platform to be resilient to dynamic changes in the environment and be able to react accordingly at runtime. When a service suddenly becomes unavailable, the Framework automatically tries to find alternative services that fulfill the requirements imposed by the bundle. If none is found, the bundle is automatically deactivated, performing user defined operations during the process. The OSGi offers mechanisms to deal with such dynamics of systems, the most powerful of them being the so-called **Declarative Service**.

The declarative services mechanism allows the developer to define *components* that typically offer some service to other modules and explicitly specify their dependencies in the form of an .XML file. This satisfies the requirements for the dynamic interaction between modules, as defined in Section 2.3. If a service becomes available only after a service consumer bundle was started, the framework automatically informs the interested bundle the service is now available, allowing therefore the **late binding** of services.
Figure 3.5 shows a simple example of the use of declarative services in OSGi. We can see that the Foo bundle offers a service under the "foo.doSomething". The actual service is implemented by the foo.doSomethingImpl which is not visible to the consumers of the services. The Bar bundle then declares its dependency on the service offered by the Foo bar by declaring the interface it expects and the methods that should be called on service binding/unbinding. Once the service is available, the service registry calls by reflection the setFoo() method of the Bar bundle, which takes an instance of the service as parameter. Bundles can define operations related to the initialization of service in the bind and unbind methods, such as assigning the given instance of the interface to a local variable for later references to the system. Note that the Bar bundle will only be activated after the service it needs are available. A more detailed example for Declarative Services taken from our implementation is given in Section 4.2.

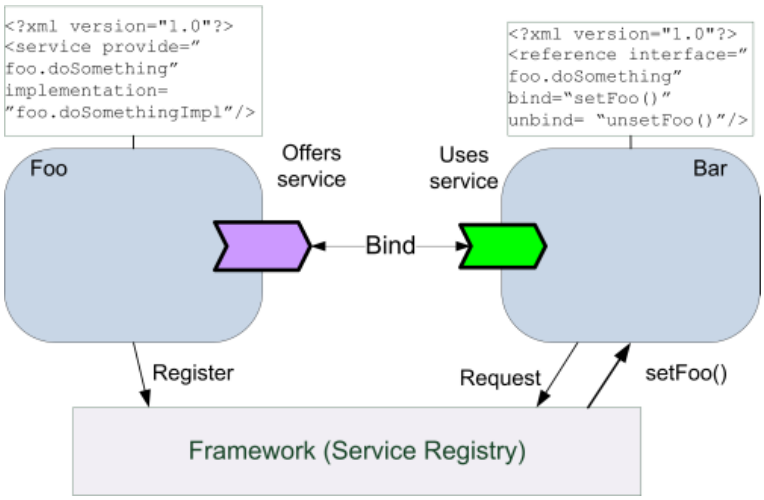


Figure 3.5: Example of declarative services in OSGi

3.6.2 Dependency Resolution

The OSGi Framework manages dependencies between components in an automatically and scalable way. When a components is started, the framework automatically tries to resolves its dependencies by matching *Import-Package* against *Export-Package* headers. The state diagram 3.6 illustrates the activation of components. Once the dependencies of a components are satisfied, it is moved from the INSTALLED state to the RESOLVED one. The services offered by the components are then ready to be used. Once a service is requested, the component is activated (ACTIVE state) until no more bundles are using its services (back to the RESOLVED state) or its dependencies become unsatisfied, when the component is the deactivated (UNSATISFIED). Once the a component is deactivated, its services are longer available.

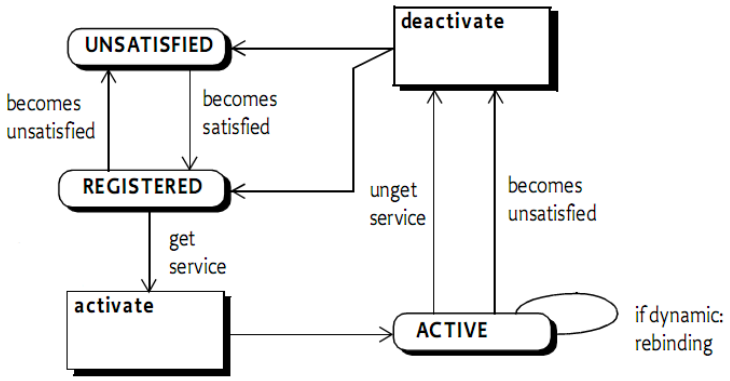


Figure 3.6: Component activation in OSGi[1]

3.7 Implementations

As noted in Section 3.1, the OSGi Alliance does not provide an implementation of the specifications, therefore to implement OSGi based software we need to choose an appropriate implementation. A summary of the currently available *certified* implementations of the OSGi specifications is given in Table 3.1.

We chose to work with the Equinox implementation because it is free, stable, well

Name	Developer	License	Last Release	Other Information
Equinox	Eclipse Foundation	EPL[2]	17.06.2008	
Makewave Knopflerfish Pro	Makewave	BSD [3]	08.06.2008	GUI[4] for Bundle Management
mBedded Server	ProSyst	Proprietary[5]	20.06.2008	GUI, embedded services
OSGi R4 Solution	Samsung	Proprietary	?	
SuperJ Engine Framework	HitachiSoft	Proprietary	08.04.2008	

Table 3.1: Comparison of OSGi Implementations

documented and easily integrable into our IDE of choice Eclipse. It is important to note that bundles should be able to work with any implementation which is fully compliant with at the least the core OSGi specifications, so the choice of one implementation over the other should not have a big impact. Some implementations do offer extra features, so it is important to employ only the standard features if the developer wants to assure total compatibility. Eclipse offers the possibility of defining a *target platform* (e.g. standard) when developing bundles.

3.8 Summary

In this section we presented some of the basic concepts of the OSGi Framework which will be relevant for our work. We saw why the OSGi framework is suitable to be used in our design and the features it offers that satisfy the requirements defined in Chapter 2. It is important to note that the OSGi Framework offers several other standard services implementations not mentioned here, like remote management, UPnP support, initial provisioning, services for mobile platforms, etc. The flexibility and extendibility of the OSGi Framework makes it applicable in many branches, as noted in Section 3.1. More information about the OSGi Framework can be found on [All99] and aQute [aSC87], a software consultancy company from the OSGi Director of Technology, Peter Kriens. The company's website offers several useful tutorials about OSGi. In the next Section we discuss the architecture of our implementation.

[1] modified from: OSGi Service Platform Service Compendium, Version 4.1. Page: 300
[2] Eclipse Public License
[3] Berkeley Software Distribution
[4] Graphical User Interface
[5] ProSys also has an Equinox based OpenSource Implementation

Chapter 4

Architecture Design

Overview

4.1 Tools **31**

4.2 Bundle activation **32**

4.3 Modules **34**

4.4 Commands architecture **39**

4.5 Extension mechanism **41**

In the previous chapters we discussed why a dynamic module system for a platform like LifeSocial is relevant and presented a framework that allows us to achieve such requirements. In this chapter we present how we employed OSGi in our architecture and discuss other design choices made.

4.1 Tools

As mentioned in Section 1.3, LifeSocial.KOM uses FreePastry[Frec] as implementation of the Pastry[RD01a] overlay Peer-to-Peer network and PAST[DR01] [RD01b] as the distributed storage system. In Section3.7 we also mentioned that we chose Equinox [Fou03] as implementation do the OSGi specifications because it integrates nicely with our IDE of choice, Eclipse. Besides these tools, we used the Eclipse PDE[1] for the development of Bundles. The Eclipse PDE was chosen because it provides a Graphical User Interface for MANIFEST file editing. An alternative to the PDE for the creation and diagnosis is the tool Bnd [aSC06], developed by Peter Kriens from the OSGi Alliance.

[1] Plugin Development Environment

4.2 Bundle activation

Figure 4.1 shows an example of how plug-in activation could be implemented in a Graphical User Interface. It uses MySpace's [MyS03] user interface as reference, since LifeSocial.KOM does not have a GUI yet. A bundle is started when the user clicks on the links offered in the toolbar or when he/she adds an application. When a bundle is started, the following steps occur:

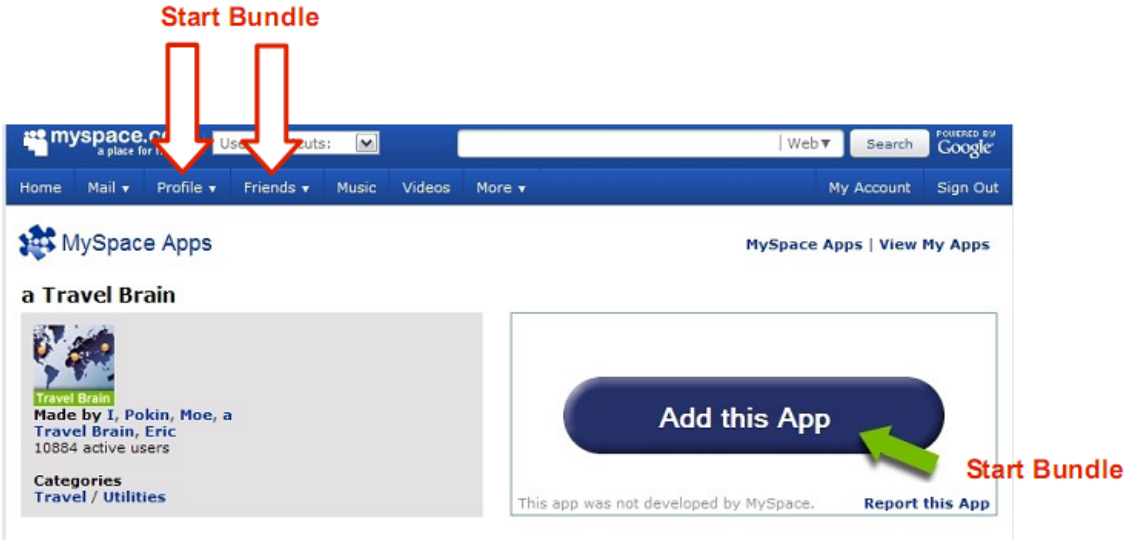


Figure 4.1: Example of plug-in activation

- **Dependency resolution**
  The first step the framework takes is to check if the dependencies of the bundle are available. If they are not, the platform returns an appropriate error message and the bundle is not activated. If all dependencies are satisfied, the next step is to bind the services this bundle utilizes. This exemplified in Figure 4.2 for the *Profile Plugin*.

  The only plug-in the *Profile Plugin* requires is the *Login Plugin*. As we can see in Figure 4.2, it declares this dependency in its MANIFEST file by using the **Import-Package** header. The *Login Plugin* exports through the **Export-Package** header a package called *de.tud.kom.lifesocial.loginplugin.api*, that defines an API for interaction with this plug-in. Notice that the actual implementation of the *Login Plugin*, contained in the *component* package, is not visible to the *Profile Plugin*, since this package is not exported.

- **Service Binding**
  If the bundle requires services from other modules, the next step is to bind these

services. This is done by calling the *bind* method defined in the component definition of the plug-in. Figure 4.3 exemplifies this. This example elaborates on the package dependency presented in Figure 4.2.
The MANIFEST file of the *Login Plugin* remains the same, but in the *Profile Plugin* we added a new header *Service-Component*, that indicates the location of the .XML file that defines this component using Declarative Services as we saw in Section 3.6.1. In the component definition file (*profileplugincomponent.xml* in this case) is where service dependencies are declared and specific information about the dependency is given. We can see that the component has a **reference** key , where it declares the interface of the service it requires (de.tud.kom.lifesocial.loginplugin.api.ILoginPlugin) and the names of the bind and unbind methods that will be called when the services becomes available and unavailable respectively (setLogin and unsetLogin). The bind method takes an instance of the requested service as parameter and usually performs initialization operations related to service binding. In our example, setLogin takes a ILoginPlugin as parameter and simply stores the passed instance in a local variable for future use. The unbind method does the exact opposite and it is called when the *Login Plugin* suddenly becomes unavailable. Inside this method we can define operations to be done before the component is deactivated. In our case it just sets the local ILoginPlugin reference to null.
Moreover we can see that the **cardinality** of the dependency was set as 1..1, meaning the *Profile Plugin* has a **mandatory** and *unary* dependency on the *Login Plugin*. Declaring a dependency mandatory means that the service is **indispensable** for the component to be activated. In other words, the component will not be activated if the *Login Plugin* interface is not available. The second value of the cardinality key defines the number of instance of the service that this plug-in accepts. Defining it unary means that if there were more Login services available, the *Profile Plugin* would not bind to them. The 1..1 cardinality makes sense for the *Profile Plugin* because the plug-in needs user information to function (mandatory dependency) and it is only going to serve one user at any time(unary dependency).

All possible values for the cardinality property are:

- 0..n - optional and multiple
- 1..n - mandatory and multiple
- 0..1 - optional and unary
- 1..1 - mandatory and unary

The last property defined for this reference is the **policy**. The policy determines how a component will react when the service it is bound to is updated or unregistered. In our example, the dynamic policy indicates that, should the *ILoginPlugin* service the component is bound to be unregistered or updated, the framework should automatically bind it to alternative target services, if available.

The possibles values for the **policy** property are:

- static - the component is deactivated when the service it is bound to changes, even if there are alternative target services to replace it. The component will have to be reactivated to bind to one of the alternatives in this case.
- dynamic - the framework automatically binds the component to alternative target services when a service is update/unregistered. The component is only deactivated if the Service Runtime Component can not find a replacement for the service

- **Services Registration**
  Once the component is activated, the services it offers are registered with the *Service Registry*. In Figure 4.3 this is illustrated by he presence of the <service> key. In this case, the *Profile plugin* offers a service whose interface is defined by the de.tud.kom.lifesocial.profileplugin.api.IProfilePlugin interface and implemented by the de.tud.kom.lifesocial.profileplugin.component.ProfilePluginComponent class. Again, we can see that actual implementation is not made available to other modules, only the service interface.

4.3 Modules

In the redesign of the platform we divided the functionality in the following modules:

- **Logger**
  The logger module provides logging functionalities to other modules. It is currently implemented using the *LogService* from the OSGi framework and configured to display error and warning messages to the console and messages of all levels to a log file. We recommend that all the modules have an *optional* dependency on this module, since all the modules are not influenced by the absence of it.

- **Core**
  The Core module contains common interfaces used by other plug-ins to implement

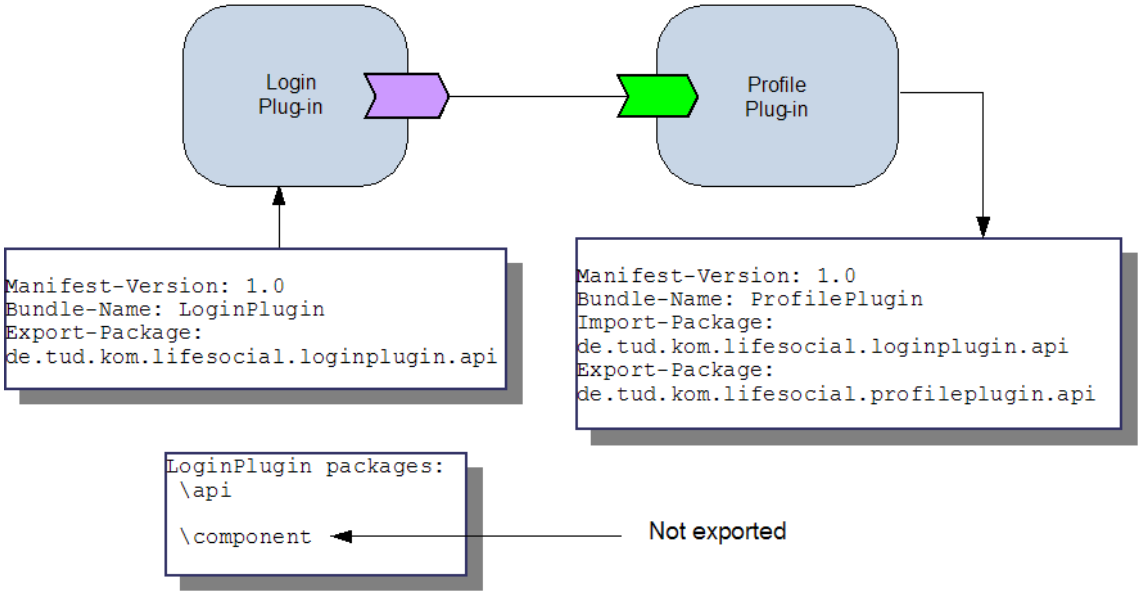


Figure 4.2: Example of Package dependency in LifeSocial.KOM

functionalities, such as the *Plugin* and the *Observer* interfaces, which all plug-ins must implement. It also contains the basic library required by the platform, such as the FreePastry library. Therefore, nearly all other modules depend on this module.

- **Utilities**
  The Utilities module provides useful utilities that are used by other modules, such as a *HashGenerator* and the *PropertiesContainer*, which defines global system properties such as standard bind port, standard file names and status messages. Many modules have a dependency on this module as well.

- **Network**
  We encapsulated the network functionalities (such as connecting and disconnecting to the FreePastry network) in one module, so in the future one could eventually plug another Peer-to-Peer network to the system by exchanging the network module. The network module depends on the *Core* and *Utilities* modules and has an optional dependency on services from the *Logger* module.

- **FileSystem**
  The system module is responsible for assuring the folder structure in the working directory is correct when the platform is started and performing clean up operations

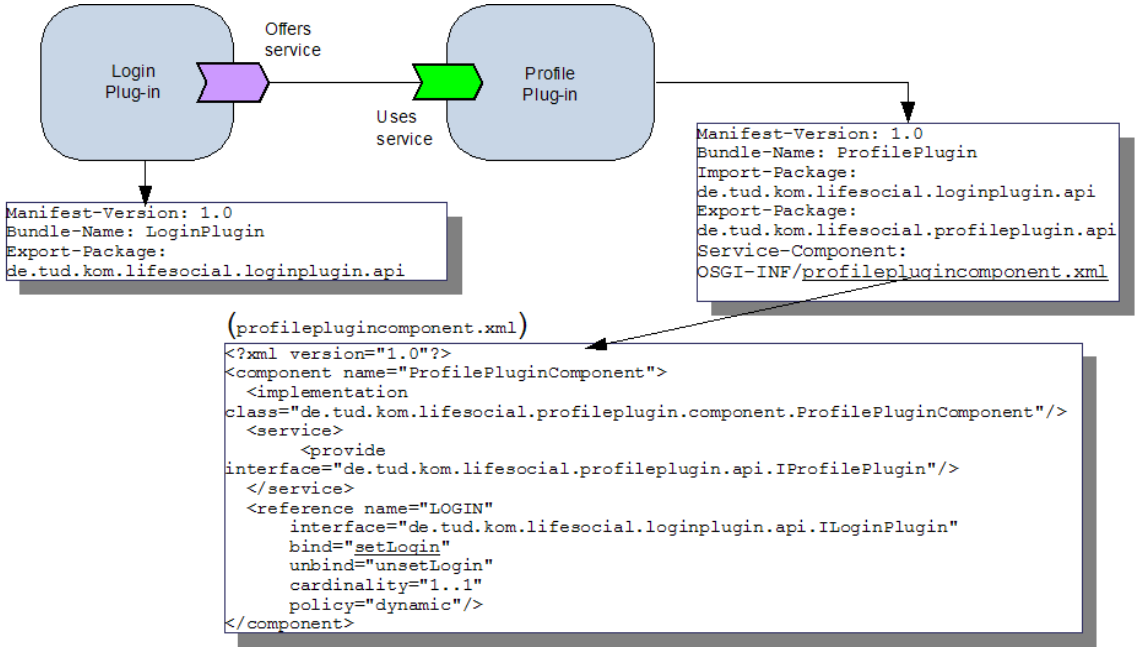


Figure 4.3: Example of service binding in LifeSocial.KOM

of data objects stored locally by the distributed storage utility when it is shutdown. Upon start-up, the working directory has to contain a folder named "default" with the default picture in it. The *FileSystem* modules depends on the *Utilities* module and has an optional dependency on services from the *Logger* module.

- **Local Storage**
  The Local Storage module is used by plug-ins to store and retrieve data objects locally. This module depends on the *Core* and *Utilities* modules and has an optional dependency on services from the *Logger* module.

- **Storage**
  The *Storage* module encapsulates the *Information Cache*, the *Storage Dispatcher* and other support classes that are used by these components. We left the Information Cache together with the Storage Dispatcher because both components are closely related. As explained in Section 1.3, the Storage Dispatcher is responsible for saving, retrieving and deleting data objects from the distributed storage (PAST). The Storage module has a mandatory dependency on the *Core*, *Utilities* and *System* modules as well on the *Network* module, since it needs the network to obtain and

store data objects from/to the network and has an optional dependency on services from the *Logger* module.

- **Message Dispatcher**
  The *Message Dispatcher* remained basically the same, except for the introduction of OSGi related features. Notice that the *Message Dispatcher* is not essential for the platform to work. Only plug-ins that offer messaging capabilities will depend on this component. It is important to note that plug-ins that offer user-to-user communication features depend *only* on the *Message Dispatcher*, which itself depend on the *Storage* module. This allows the system to continue offering real time communication even if the *Storage* is currently unavailable. The *Message Dispatcher* module depends on the *Core*, *Utilities* and the *Network* module, since it needs the network to send user-to-user messages. It also has an optional dependency on the *Storage* module, to be able to provide storage capabilities in case the user is offline when the message is sent and an optional dependency on services from the *Logger* module.

- **Email Sender**
  The Email sender module is used to send platform e-mails to users, e.g. when using the password recover function or sending newsletters. It currently supports only Google Mail™[Inc04] accounts and it uses the "Secure iNet Factory" library™[JSC03] from JSCAPE™[JSC99].

- **User interface**
  We implemented the user interface as an extension of the Equinox console that allows the user to enter LifeSocial.KOM commands using the syntax illustrated in Listing 4.1.

```
execute <LifeSocial command> <arguments>
```

Listing 4.1: LifeSocial.KOM User Interface

  General help and a list of all currently available commands is displayed by entering "help" and specific help for a command can be obtained through "gethelpfor" <Life social command name>.

  The *User interface* module depends on the *Core* and *Utilities* modules and on 0-n *Commands* components (the Commands architecture is explained in Section 4.4) as well as on the *org.eclipse.osgi.framework.console.CommandProvider* interface, since it is currently implemented as an extension of the standard Equinox console.

- **Plug-ins**
  All the plug-ins maintained their functionalities with the exception of the *Search Plugin*, which was changed to provide slightly better search functionalities. When the platform is started, only the *Login Plugin* is needed, which is usually the first action the user has to perform when accessing the services of an online community. All other plug-ins have therefore a dependency on the *Login Plugin*, since they usually need user information to function. The only exception so far is the *Search Plugin*, which can be used also without being logged in.
  Figure 4.4 depicts the plug-in dependency in our architecture. We left the *Core*, *Utilities* and *Logger* modules out for clarity.

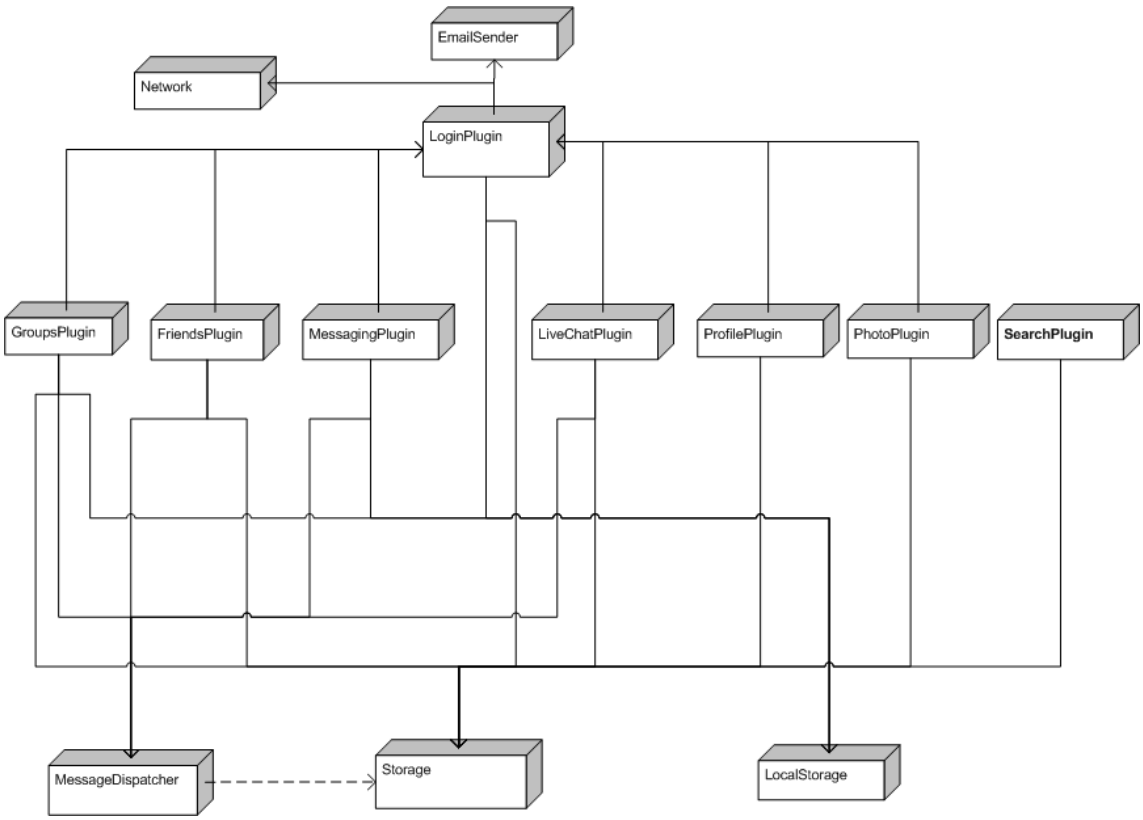


Figure 4.4: Plug-in dependencies in our architecture

4.4 Commands architecture

In our implementation every plug-in has its own set of commands which are made available to the platform only after the corresponding plug-in is activated, since it does not make sense to have the commands available if the plug-in is not available.

The class diagram 4.5 illustrates the command architecture for the *Profile Plugin*. As we saw in Section 4.3, the *User Interface* module depends on 0 or more *Commands* components representing the commands of the plug-ins which are currently activated. The *Commands* interface act as an aggregator of 1 or more *CommandHandlers* for a certain plug-in and act as a proxy for command execution. It contains a identifier of the plug-in it is associated with and maps commands to the corresponding *CommandHandler*. It also offers other functionalities such as help for all plug-in's commands and checking command availability. The *CommandHandlers*, are the ones for actually interacting with the plug-in when executing the program and returning appropriate status messages. Currently, *CommandHandlers* return status messages as *String* objects, to facilitate its incorporation in a future Graphical User Interface.

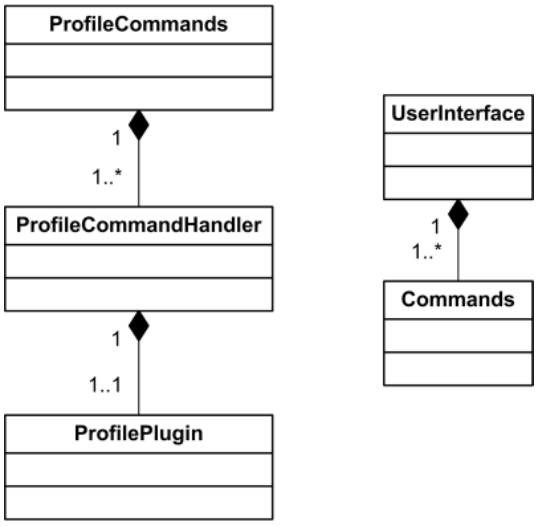


Figure 4.5: Example of the command architecture

A list of the LifeSocial.KOM commands currently available (obviously only when the associated plug-in is activated) is given in Appendix A.

The following sequence diagram illustrates an example of plug-in activation in our implementation and how the resolutions for the *CommandHandlers* and *Commands* objects occurs.

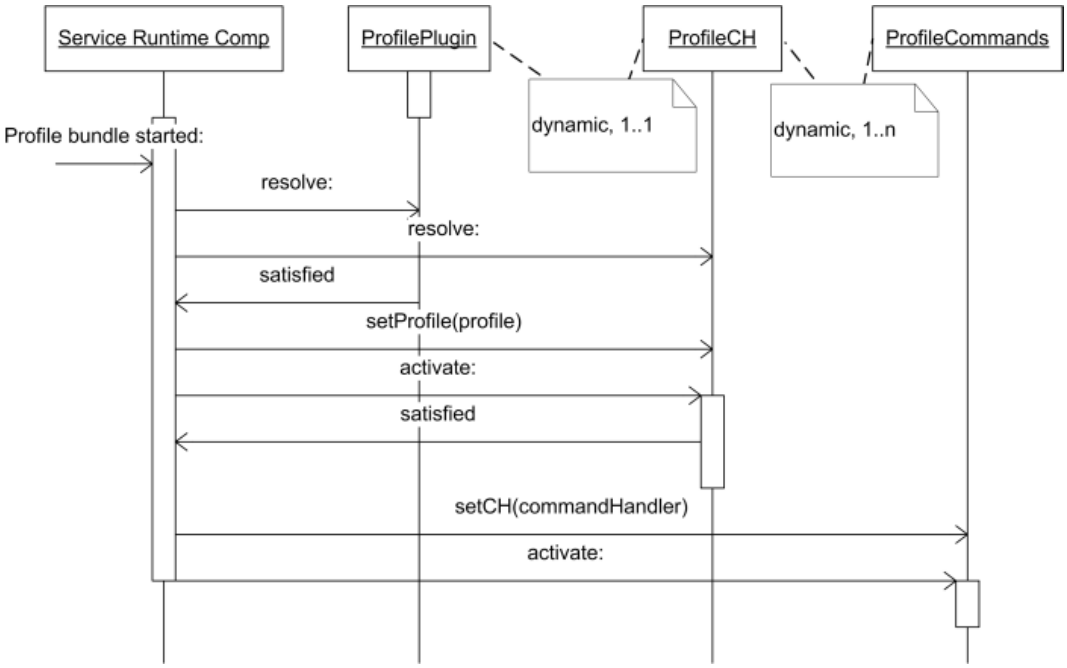


Figure 4.6: Example of command activation in LifeSocial.KOM

When the *Profile Plugin* is first started, the *Service Runtime Component* tries to resolve the dependencies of the *ProfileCommandHandler* (dependent on the *Profile Plugin*) and *ProfileCommand* (dependent on the one or more *ProfileCommandHandler* instances) components. Assuming the dependencies of the *Profile Plugin* are satisfied, so the dependency of *ProfileCommandHandler* is also satisfied and its *setProfile* bind method is called by reflection and the *ProfileCommandHandler* is then activated, since it only depends on the *Profile Plugin*. When the *ProfileCommandHandler* is activated, the dependency of the *ProfileCommand* component is satisfied, its *setCH* method is called and the component is activated. Once the component is activated, the commands for the *Profile Plugin* are then available to the platform. As a last note, it is important to point out that when developing new plug-ins, developers should assure that functionalities which are not available (for example, because one of its optional dependency is not available) also do not offer its command(s) to the platform.

4.5 Extension mechanism

In this Section we explain the steps involved when creating new plug-ins for LifeSocial.KOM.

1. **Dependency definition**
   To create a new plug-in, the developer has to first analyze its dependencies and the services it will eventually provide to other modules. For example, if the plug-in is meant to provide user-to-user communication, it will depend on the Message Dispatcher and possibly on the Login Plugin. When implementing the plug-in, the appropriate MANIFEST and component .XML files have to be created based on this information.

2. **Data objects creation**
   The next step is to create the data objects that will be used by the plug-in. The *Core* module contains the basic interfaces for *Storable* (*SharedItem*, to be used when data is to be stored in the network using the *Storage Dispatcher*) and *Message* (*BasicMessage*, to be used when sending user-to-user messages using the *Message Dispatcher*) data objects. These objects have a *Header* and a *Payload* associated with them.

3. **Definition and creation of commands**
   The next step is to define what commands the plug-in will offer to the platform and create the correspondent *CommandHandlers* and *Command* components and linking them. When developing the plug-in, the developer has to make sure the commands currently available to the platform reflect the state of the plug-in. If the plug-in is not able to provide some functionality at the moment, then the *CommandHandler* associated with the command should not be available.

4. **Interface definition and component implementation**
   Finally, the developer can implement plug-in functionality by implementing the *Plugin* and *Observer* interfaces. The naming format we suggest for exported services we suggest is:
   de.tud.kom.lifesocial.<plugin_name>.api.<Interface> - for the exposed interface and
   de.tud.kom.lifesocial.<plugin_name>.component.<Plugin_name>Component - for the implementation class.

Special attention is required when declaring dependencies mandatory or optional. When declaring a *mandatory* dependency, the developer should keep in mind that making a dependency mandatory means that the plug-in will not be activated until that dependency is satisfied. Hence a module should only use mandatory dependency for services that are essential for its functions (classical example: the *Login Plugin)*. For example, imagine a plug-in that allows the user to build its family tree. It could have a mandatory dependency on the *Profile Plugin*, since it needs it to display the profiles of the tree members but an extra functionality of accessing the family pictures directly from the plug-in could have an *optional* dependency on the *Photos Plugin*, since its basic functions would still work without the *Photos Plugin*.
On the other hand, when developing plug-ins that have *optional* dependencies, keep in mind that if a dependency is optional, the module will be activated even if this service is not available. Therefore extra care must be taken when calling methods from that service. Always make sure the optional dependency is set before calling its methods to avoid null point exceptions.

4.5.1 Example: a Blog Plug-in

In this section we exemplify the plug-in extension mechanism by outlining the steps involved in designing a new plug-in. Imagine the case of a plug-in that offers blogging services as shown in Figure 4.7 for the MySpace blog interface.



Figure 4.7: User interface of the MySpace Blog

1. **Dependency definition**
   A blog plug-in would need to store and retrieve blog entries to/from the distributed storage, eventually have access to some of the profile fields and links to other plug-in such as the *Messaging Plugin* and the *Login Plugin*. The dependencies of the plug-in would thus be:

   - **Storage Dispatcher (mandatory)**
     The plug-in would *need* the Storage Dispatcher to be able to retrieve previous blog entries and store new ones.
   - **Login Plugin (optional)**
     In online communities, blog entries can be made public, which means that users do not have to be logged in to *read* them. However, to be able to *post* entries and have access to other user specific blog information (e.g. "kudos", "blog history", etc) the user must be logged in and the blog plug-in should have some kind of mechanism to inform the users of that, for example by forwarding them to the login page when they click on links like "My Blog", "Post Blog", etc.
   - **Profile Plugin (optional)**
     To be able to show user specific information 4.7, it is necessary to have access to profile fields. The dependency would be optional for the same reason mentioned for the Login Plug-in and also because the absence of profile information does not influence the functionality of the blog plug-in, even when posting blogs in this case.
   - **Messaging Plugin (optional)**
     As it can be seen in Figure 4.7, the MySpace blog toolbar also contains a link to the "mail" feature. To be able to provide similar functionality we would have to be dependent on the *Messaging* plug-in as well.

2. **Data objects creation**
   The next step would be to define data objects for the blog plug-in. Since blog entries would need to be stored, the data objects would have to extend the *SharedItem* class and its payload would have to implement the *Payload* interface. The developer is then free to implement extra blog functionalities on these classes (e.g . "get blog entry", "get blog colection", etc.).

3. **Definition and creation of commands**
   Next, a *CommandHandler* component has to be created for each command offered

by the plug-in. The *CommandHandler* are responsible for actually executing the command, as we saw in Section 4.4. the *BlogCommands* would then aggregate all the *CommandHandlers* and make the commands available to the platform when the plug-in is activated.

The *CommandHandlers* that are available to the platform should reflect the current status of the user. For example, it does not make sense to offer a command to *display* the user's own blog history if the user is not logged in, since the plug-in would have no way of obtaining the history for them. Instead, the user interface could display generic blog links (e.g. "My Blog") and after the user successfully logs in the plug-in can then make all the blog commands available.

4. **Interface definition and component implementation**
   Lastly, the developer has to create the actual blog interface and its implementation. The implementation will use its dependencies to perform the functions offered by the blog, e.g. obtaining blog entries from the network by using the *Storage Dispatcher*, displaying profile information from the *Profile Plugin*, etc.

Chapter 5

Conclusions

Overview

5.1 Conclusion **45**

5.2 Future Work **47**

In this section we analyze our solution, discuss some lessons learned during the development process and give an overview of interesting topics still open to be researched in the future.

5.1 Conclusion

The objective of this work was to develop a modular architecture for LifeSocial.KOM that reacts to dynamic changes in the environment. We chose the OSGi framework as basis to achieve our goals and redesigned the architecture of LifeSocial.KOM to exploit the OSGi features and transform the platform in a dynamic modular system.

The new architecture of LifeSocial.KOM provides a well defined extension mechanism as we saw in Section 4.5 that dynamically adapts to changes in the state of modules. Dependencies between modules can now be more fine-grained and modules can add version information to the modules and packages it depends on. Moreover, it is possible to update plug-ins and add new ones at runtime, a great advantage in comparison with the previous architecture, considering that in a real world environment the re-compilation and redistribution of the platform each time a plug-in is modified or added is impracticable.

The current extension mechanism allows targeted development of plug-ins and the

interaction between modules is performed through well defined interfaces, what allowed us to achieve a high level of *Information Hiding*. Modules are now more cohesive as well and loose coupling was achieved by redesigning modules to obtain a stronger separation of concerns that allowed us to remove weak dependencies.

The new architecture of LifeSocial.KOM also provides a new command architecture that allows commands to be dynamically available to the platform. Now, the commands for a plug-in reflects its state, thus being available only after the plug-in is activated. The search functionalities of the platform were also slightly changed to provide better results.

Another advantage of the new implementation is that all employed mechanisms are also part of the OSGi Mobile Compendium, thus allowing the platform to run in mobile devices such as cell phones, PDA's, Smartphones, etc.

The extension possibilities for the platform are endless. One can even create components inside plug-ins that provide extra functionality, like for example photo editing, enhanced profile capabilities, etc.

During the development of this work, considerable experience was gained about the OSGi technology and its applications possibilities, which are summarized next.

- **Framework dependency**
  One disadvantage of the new implementation is that it needs the OSGi Framework to run. We are however convinced that the benefits introduced by the use of the OSGi framework far outweigh its drawbacks.

- **Require-Bundle header**
  When developing bundles, one must be careful when using the *Require-Bundle* instead of the *Import-Package* one. The developer should keep in mind that when requiring a bundle instead of specific packages, the module will be importing all exposed packages from the bundle. This is usually not necessary or even desired. Furthermore, when using the *Require-Bundle* header, there is no way of defining package dependency version information. This is specially relevant when adopting the Eclipse PDE for bundle development, since it tends to stimulate the use of Require-Bundle header. As a rule of thumb, always choose *Import-Package* over *Require-Bundle* unless **really** necessary, what is usually not the case.

- **Cyclic dependencies**
  Another risk the developer might run into is producing modules that have cyclic dependency on each other. This might causes deadlocks, since when the bundles are started both of them will be waiting for the other one to be activated to satisfies its dependencies, but the this is not possible since they depend on each other. An example of cyclic dependency that existed in the previous architecture is depicted in Figure 5.1. When registering a user, the *Login Plugin* needed *Profile Plugin* to create a new basic profile. That was the only moment the *Profile Plugin* was used, being thus a weak dependency. We solved this problem by separating concerns and delegating the call to createUserProfile to the command level. Now, when executing the <register> command, the basic user Items are created using the *Login Plugin* and, if successful, the *Profile Plugin* is called to create a new user profile. An advantage introduced when avoiding cyclic dependencies is that it "forces" modularization and separation of concerns.

- **Small pool of complex examples**
  The number of complex examples using OSGi and specifically using Declarative Service is very limited and the ones available are not that well documented. There are tutorials available, but they usually are simple and do not attend the needs for a bigger project. Therefore one must rely on the specifications, but the examples contained in them are somehow abstract and simple. Hopefully this work will contribute to the pool of relatively complex projects designed using OSGi and Declarative Services.

5.2 Future Work

There are some issues that still need to be researched more deeply.

- **Extend Functionalities**
  The functionalities offered by LifeSocial are currently limited. Certain popular functionalities common in online communities like videos, blogs, games, quizzes, etc are still missing and are desired to make LifeSocial a more attractive platform.

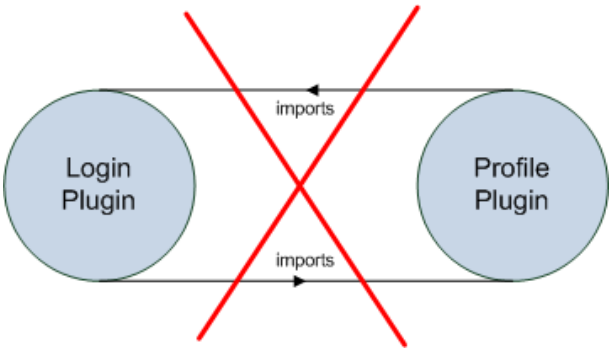


Figure 5.1: Cyclic dependency in OSGi

- **Plug-in repository**
  So far, the platform has no support for the download of new modules in any way. The modules are redistributed using USB sticks or by e-mail. It would be useful to have a form of repository where new bundles and versions can be obtained. So the user could browse the repository and choose to install new applications, a service offered by the most popular social networks. It should be noticed that in this case, a mechanism to store the plug-ins each user has installed on their account would have to be developed, since when talking about optional plug-ins, different users might have different applications installed. This could be implemented by storing the list of installed plug-ins using the distributed storage utility and retrieving it when the member logs in.

- **Graphical user interface**
  The current implementation of LifeSocial.KOM offers a command line user interface implemented as an extension of the Equinox console. This is obviously not the appropriate user interface for an online community, which should allow the users to graphically visualize the information. Therefore a graphical user interface is essential for a consistent platform. During this project the foundations for the integration of the platform in a GUI were built, in the form of Command Interfaces that return Status messages aiming to be integrated to a GUI. A graphical user interface for LifeSocial.KOM is currently under development at KOM [Gra08].

- **Web browser support**
  So far our platform was developed as a stand alone Java application. To be able to truly integrate with the Internet environment and become a real alternative to current online communities, it is fundamental that user can access the platform through a web browser. *Gears*[Goo07] could be a promising technology to be more deeply researched as an alternative.

Appendices

Appendix A

List of LifeSocial commands

This appendix lists the commands currently available in the LifeSocial.KOM platform. The status messages returned for each case are found and can be changed in the *PropertiesContainer* class of the *Utilities* module.

Network commands		
Command name	**Arguments**	**Description**
connect	(no args)	connects using localhost and port 9001
connect	<localport>	connects using localhost and given port
connect	<localport> <bindaddress>	connects using given host and port
connect	<localport> <bindport> <bindaddress>	connects using given host and ports
connectp	<localport> <bindport>	connects using localhost and given local and boot ports
disconnect	(no args)	disconnects from the LifeSocial network

Table A.1: Network commands

Login Plugin commands		
Command name	**Arguments**	**Description**
login	<username> <password>	logs in. Must be connected first
logout	(no args)	logs the current user out
register	<username><password><email><country><city>	creates a new user account
loginstatus	(no args)	shows the username of the currently user
recover	<username> (or/and) <email>	recover user password
changepass	<old pass><old pass repeat><new pass>	changes the user password

Table A.2: Login Plugin commands

Profile Plugin commands		
Command name	**Arguments**	**Description**
editprofilefield	<field name><new value>	changes the value for the given field
requestprofile	<username>	loads the profile of the user
showcurrentprofile	(no args)	returns the previously requested profile
showmyprofile	(no args)	returns own profile
storemyprofile	(no args)	stores own profile

Table A.3: Profile Plugin commands

Friends Plugin commands		
Command name	**Arguments**	**Description**
getmyfriendslist	(no args)	returns user's own friends list
getmyfriendslistandwait	(no args)	requests own list and waits for its arrival
getfriendslisthost	(no args)	returns the username of the current list owner
loadfriendslist	<username>	loads the list of friends of the given user
friendslistloaded	(no args)	indicates whether the requested list is loaded
getfriends	(no args)	returns the list of user names of the current friends' list
requestfriend	<username><reason>	sends an friendship request to the defined user
acceptfriend	<request _key><initiator>	accepts the request and adds the user to own list
declinefriend	<key> <initiator>	declines the given friendship request
deletefriend	<user name>	removes a friend from the friends' list
getfriendsmsgs	(no args)	returns the friends msgs/requests
friendsmsgavail	(no args)	indicates whether new friends msgs are available

Table A.4: Friends Plugin commands

Groups Plugin commands		
Command name	**Arguments**	**Description**
loadgroups	<user name>	loads a list of groups for the given user
usergroupsloaded	(no args)	indicates whether the list of user groups is completely loaded
getgroupslist	(no args)	returns the list of groups currently loaded
loadmygroups	(no args)	loads my list of groups
myusergroupsloaded	(no args)	indicates whether the list of user groups is completely loaded
getmygroupslist	(no args)	returns my list of groups currently loaded
joingroup	<group name>	adds the current user to the members list of the given group
leavegroup	<group name>	deletes the current user from the members list of the given group
creategroup	<group name>	creates a new group if the group name is free
groupcreated	(no args)	informs if the previous group creation was a success
loadsinglegroup	<group name>	loads the list of members for the defined group
singlegrouploaded	(no args)	indicates whether the requested single group is already loaded
showcurrentgroupmembers	(no args)	returns a list of group members of the currently available group
groupsmsgsavail	<no args>	indicates if new unseen group msgs are available
groupsgetmsgbox	(no args)	returns a list of keys of group msgs
getgroupsmsg	<key of a msg>	returns a single group msg for the given key
getgroupshost	(no args)	returns the user name whose group list is currently available
sendgroupmsg	<groupname><msg>	sends msg to all participants of the given group

Table A.5: Groups Plugin commands

Live Chat Plugin commands		
Command name	**Arguments**	**Description**
livechathistoryusers	(no args)	returns list of usernames that have a chat history with the user
livechathistoryforuser	<username>	returns the history for chat with given username
livechatmsgsavail	(no args)	checks whether new chat messages are available
livechatsendmsg	<recipient> <message>	sends a chat message to the given user
getchatmsgs	(no args)	returns chat messages which are available

Table A.6: Live Chat Plugin commands

Messaging Plugin commands		
Command name	**Arguments**	**Description**
getinbox	(no args)	returns message ids available in inbox
showinboxmsg	<msg_key>	returns the content of an 'inbox' message for the given message key
getsentmsgs	(no args)	returns message ids available in sent box
showsentmsg	<msg key>	returns 'sent' message for the given message key
checkmsgs	(no args)	loads undelivered messages for the Messaging plugin
deleteinboxmsg	<msg key>	deletes an inbox message with the given message key
deletesentmsg	<msg key>	deletes a sent message with the given message key
sendmsg	<username> <msg>	sends a message (Messaging plugin)

Table A.7: Messaging Plugin commands

Search Plugin commands		
Command name	**Arguments**	**Description**
search	<criteria> (search criteria. e.g. city)	gets a list of users using search criteria
getsearchresults	(no args)	gets a list of users found with previous 'search'

Table A.8: Search Plugin commands

Photos Plugin commands		
Command name	**Arguments**	**Description**
addphoto	<alkey><title><path><comms>	adds a photo to the defined album. path is absolute
deletealbum	<album key>	deletes an album with the given key
createalbum	<title> <comments>	creates a new album
albumsloaded	(no args)	indicates whether the requested list of albums was loaded
deletephoto	<album key> <photo key>	deletes a photo with the given key
editalbum	<album key><title><comments>	changes album information. 'null' keeps a field unchanged
editphoto	<title> <comments>	changes the info for the current photo. 'null' keeps a field unchanged
getalbums	(no args)	returns a list of albums keys
getalbumshost	(no args)	returns the user name of the albums host
loaduseralbums	<username>	loads the albums for the given user
loadphoto	<photo_key>	loads a photo with the defined key
photoloaded	(no args)	indicates whether the requested photo is already loaded
showphoto	(no args)	shows the currently loaded photo

Table A.9: Photos Plugin commands

Appendix B

Starting the platform

Before starting the platform, the user first has to make sure the directory where LifeSocial.KOM is started contains a folder called *"default"* with the default image in it.

B.1 Test mode

Once the platform has been started (remember you need to have an OSGi implementation installed), the first step is to connect to the LifeSocial.KOM network using one of the *connect* methods described in A. When running the platform in a LAN (usually the case while testing), the user should first use the *connect* method without arguments. This will create a new ring using the localhost and the standard port (currently 9001, this can be changed on the *PropertiesContainer* class available in the *Utilities* module). The second step is to register a new user, since in test mode the ring disappears after all the clients are shut down. Repeat this procedure for a few clients to be able to retrieve profiles, add friends and user other functionalities. Notice that after registering, a user is automatically logged in and is ready to use platform services.

We can summarize the steps for starting the LifeSocial.KOM platform in test mode as:

- Check if default folder present
- Start the platform
- Connect to the network. *connect* (no arguments) for first client and *connect* <*local port*> for the others
- Register a new user

- Repeat the steps above for a few users
- Start using the platform

B.2 Live mode

When the platform is running over the Internet, it is assumed that the ring will persist between accesses, so there is obviously no need of registering every time. In a live environment, the user will need to change the current FreePastry settings (which is configured in test mode to be run in a LAN). In the network module, the parameter *nat_search_policy=never* set when connecting to the network has to be changed to *nat_search_policy=prefix* or simply deleted if the user has the *freepastry.params* file present. More information about FreePastry's parameters can be found on [Frea] and specifically about NAT settings on [Freb]. When connecting to the LifeSocial.KOM network over the Internet, the *connect* <*localport*> <*bindaddress*> can be used (assuming other users are using the standard port), or *connect* <*localport*> <*bindport*> <*bindaddress*> if they are not. The bind address is the IP of a node which is already connected to the ring. After connecting to the network, the user can then login and starting using its services.

Bibliography

[AG47] Siemens AG. Siemens website, 1947. Available from: http://www.siemens.com [cited 2008.10.18].

[AG69] Software AG. Software ag website, 1969. Available from: http://www.softwareag.com [cited 2008.10.18].

[AG72] SAP AG. Sap website, 1972. Available from: http://www.sap.com [cited 2008.10.18].

[Ale08a] Alexa. Alexa top 500 sites global, October 2008. Available from: http://www.alexa.com/site/ds/top_sites?ts_mode=global&lang=none [cited 2008.10.17].

[Ale08b] Alexa. Alexa top sites canada, October 2008. Available from: http://www.alexa.com/site/ds/top_sites?cc=CA&ts_mode=country&lang=none [cited 2008.10.17].

[Ale08c] Alexa. Alexa top sites united kingdom, October 2008. Available from: http://www.alexa.com/site/ds/top_sites?cc=GB&ts_mode=country&lang=none [cited 2008.10.17].

[Ale08d] Alexa. Alexa top sites usa, October 2008. Available from: http://www.alexa.com/site/ds/top_sites?cc=US&ts_mode=country&lang=none [cited 2008.10.17].

[All99] OSGi Alliance. Osgi - the dynamic module system for java, 1999. Available from: http://www.osgi.org/ [cited 2008.10.18].

[All07] OSGi Alliance. Osgi service platform release 4, May 2007. Available from: http://www.osgi.org/Specifications/HomePage?section=2#Release4 [cited 2008.10.18].

[aSC87] aQute Software Consultancy. aqute website, 1987. Available from: http://www.aqute.biz [cited 2008.10.18].

[aSC06] aQute Software Consultancy. Bnd - bundle tool, 2006. Available from: http://www.aqute.biz/Code/Bnd [cited 2008.10.19].

[BS08] George Gilder Bret Swanson. Estimating the exaflood - the impact of video and rich media on the internet - a zettabyte by 2015?, January 2008. Available from: http://www.discovery.org/a/4428 [cited 2008.10.17].

[Car07] David F. Carr. Inside myspace: The story, January 2007. Available from: http://www.baselinemag.com/c/a/Projects-Networks-and-Storage/Inside-MySpacecom [cited 2008.09.06].

[Cha08] Marketing Charts. Top 10 us social-network and blog-site rankings issued for may, May 2008. Available from: http://tinylink.com/?Kp77So2iBF [cited 2008.10.17].

[cla95] classmates. classmates.com website, 1995. Available from: http://classmates.com/ [cited 2008.10.17].

[Coh01] Bram Cohen. The bittorrent protocol specification, April 2001. Available from: http://www.bittorrent.org/beps/bep_0003.html [cited 2008.10.18].

[Cor65] Nokia Corporation. Nokia website, 1865. Available from: http://www.nokia.com [cited 2008.10.18].

[Cor89] International Business Machines Corporation. Ibm website, 1889. Available from: http://www.ibm.com [cited 2008.10.18].

[Cor99] NEC Corporation. Nec website, 1899. Available from: http://www.nec.com [cited 2008.10.18].

[dEnS24] Telefónica de España S.A. Telefonica website, 1924. Available from: http://www.telefonica.com [cited 2008.10.18].

[DNI08] DNI. Office of the director of national intelligence, October 2008. Available from: http://www.dni.gov/ [cited 2008.10.17].

[DR01] Peter Druschel and Antony Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *HotOS VIII*, May 2001. Available from: `http://freepastry.rice.edu/PAST/hotos.pdf.zip` [cited 2008.10.18].

[Equ98] Inc Equinix. Equinix website, 1998. Available from: `http://www.equinix.com/` [cited 2008.10.17].

[Eri76] Telefonaktiebolaget L. M. Ericsson. Ericsson website, 1876. Available from: `http://www.ericsson.com` [cited 2008.10.18].

[Fac04] Facebook. Facebook website, February 2004. Available from: `www.facebook.com` [cited 2008.10.17].

[Fac07] Facebook. Facebook platform launches, May 2007. Available from: `http://developers.facebook.com/news.php?blog=1&story=21` [cited 2008.10.17].

[Fou03] Eclipse Foundation. Equinox website, 2003. Available from: `http://www.eclipse.org/equinox/` [cited 2008.10.18].

[Fou08] Eclipse Foundation. Equinox download website, June 2008. Available from: `http://download.eclipse.org/eclipse/equinox` [cited 2008.10.18].

[Frea] FreePastry. Adjusting freepastry parameters. Available from: `http://freepastry.org/FreePastry/tutorial/params.html` [cited 2008.10.20].

[Freb] FreePastry. Freepastry nat/firewall configuration. Available from: `http://freepastry.org/FreePastry/tutorial/nat.html` [cited 2008.10.20].

[Frec] FreePastry. Freepastry website. Available from: `http://freepastry.rice.edu/` [cited 2008.10.18].

[Fri02] Friendster. Friendster website, March 2002. Available from: `http://www.friendster.com/` [cited 2008.10.17].

[Gmb97] ProSyst Software GmbH. Prosyst website, 1997. Available from: `http://www.prosyst.com` [cited 2008.10.18].

[Goo07] Google. Gears website, May 2007. Available from: `http://gears.google.com/` [cited 2008.10.17].

[GPM+08] Kalman Graffi, Sergey Podrajanski, Patrick Mukherjee, Aleksandra Kovacevic, and Ralf Steinmetz. A distributed platform for multimedia communities. In *IEEE International Symposium on Multimedia (ISM '08)*, page 6, Berkley, USA, Dec 2008. IEEE, IEEE Computer Society Press. Available from: www.kom.tu-darmstadt.de/~graffi/pub/Kalman.Graffi_2008_IEEE.ISM.08_A.Distributed.Platform.for.Multimedia.Communities.pdf [cited 2008.10.18].

[Gra08] Kalman Graffi. Kalman graffi website at kom, October 2008. Available from: http://www.kom.tu-darmstadt.de/en/people/staff/kalman-graffi/ [cited 2008.10.17].

[Gro38] Samsung Group. Samsung website, 1938. Available from: http://www.samsung.com [cited 2008.10.18].

[Gro96] Deutsche Telekom Group. Deutsche telekom website, 1996. Available from: http://www.telekom.com/ [cited 2008.10.18].

[Hig08] Stacey Higginbotham. Myspace uses gears to grind down server costs, May 2008. Available from: http://gigaom.com/2008/05/28/myspace-uses-gears-to-grind-down-server-costs/ [cited 2008.10.17].

[Hit10] Ltd Hitachi. Hitachi website, 1910. Available from: http://www.hitachi.com [cited 2008.10.18].

[Inc28] Motorola Inc. Motorola website, 1928. Available from: http://www.motorola.com [cited 2008.10.18].

[Inc04] Google Inc. Google mail website, March 2004. Available from: https://mail.google.com/ [cited 2008.10.19].

[Inc05] Zattoo Inc. Zatoo website, 2005. Available from: http://zattoo.com/ [cited 2008.10.18].

[JSC99] JSCAPE. Jscape website, 1999. Available from: http://www.jscape.com/ [cited 2008.10.19].

[JSC03] JSCAPE. Secure inet factory, 2003. Available from: http://www.jscape.com/secureinetfactory/ [cited 2008.10.19].

[Kri07] Peter Kriens. Peter kriens discusses osgi, September 2007. Available from: `http://www.infoq.com/interviews/osgi-peter-kriens` [cited 2008.10.18].

[ldF46] É lectricité de France. Edf website, 1946. Available from: `http://www.edf.com` [cited 2008.10.18].

[Mil06a] Rich Miller. Myspace shifts to peering for video delivery, November 2006. Available from: `http://www.datacenterknowledge.com/archives/2006/11/10/myspace-shifts-to-peering-for-video-delivery` [cited 2008.10.17].

[Mil06b] Rich Miller. Power outage knocks myspace.com offline, July 2006. Available from: `http://news.netcraft.com/archives/2006/07/24/power_outage_knocks_myspacecom_offline.html` [cited 2008.10.17].

[Mil07] Rich Miller. Infrastructure costs nearly killed myspace, May 2007. Available from: `http://www.datacenterknowledge.com/archives/2007/05/04/infrastructure-costs-nearly-killed-myspace/` [cited 2008.09.06].

[MyS03] MySpace. Myspace website, August 2003. Available from: `http://www.myspace.com` [cited 2008.10.17].

[ork04] orkut. orkut website, January 2004. Available from: `http://www.orkut.com/` [cited 2008.10.17].

[Phi07] Sarah Phillips. A brief history of facebook, July 2007. Available from: `http://www.guardian.co.uk/technology/2007/jul/25/media.newmedia` [cited 06 Sep 2008].

[Pod08] Sergey Podrajanski. A peer-to-peer based framework for large-scale social networking platforms, June 2008.

[RD01a] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001. Available from: `http://freepastry.rice.edu/PAST/pastry.pdf.zip` [cited 2008.10.18].

[RD01b] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *18th ACM Symposium on Operating Systems Principles (SOSP'01)*, pages 188–201, October 2001. Available from: `http://freepastry.rice.edu/PAST/past-sosp.pdf.zip` [cited 2008.10.18].

[RH95] Inc. Red Hat. Red hat website, 1995. Available from: `http://www.redhat.com` [cited 2008.10.18].

[S.A03] Skype Technologies S.A. Skype.com website, August 2003. Available from: `http://www.skype.com/` [cited 2008.10.18].

[SE05] Ralf Steinmetz and Klaus Wehrle (Edits.). *Peer-to-Peer Systems and Applications*. Springer, September 2005. Available from: `http://www.springer.com/computer/communications/book/978-3-540-29192-3` [cited 2008.10.18].

[Sha08] Larry Shaughnessy. *CIA, FBI push "Facebook for spies"*, September 2008. Available from: `http://www.cnn.com/2008/TECH/ptech/09/05/facebook.spies/index.html` [cited 2008.09.06].

[SM] Inc. Sun Microsystems. Sun website. Available from: `http://sun.com` [cited 2008.10.18].

[SM04] Inc. Sun Microsystems. Java archive (jar) files, 2004. Available from: `http://java.sun.com/j2se/1.5.0/docs/guide/jar/index.html` [cited 2008.10.19].

[stu05] studiVZ. studivz website, October 2005. Available from: `http://www.studivz.net/` [cited 2008.10.17].

[Swa08] Jon Swartz. Social-networking sites going global, February 2008. Available from: `http://www.usatoday.com/money/industries/technology/2008-02-10-social-networking-global_N.htm` [cited 2008.09.06].

[Tom08] Tom. Myspace open to developers, January 2008. Available from: `http://www.all4myspace.com/news/MySpace-Open-to-Developers` [cited 2008.10.17].

[TT94] Discovery Institute's Technology and Democracy Project (TDP). Tdp website, 1994. Available from: `http://www.discovery.org` [cited 2008.10.17].

[Xin03] Xing. Xing website (named openbc/open business club until november 17th, 2006), August 2003. Available from: `http://www.xing.com/` [cited 2008.10.17].