Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS

Author: Ondrej Mihályi

Development of browser-based desktop-oriented web applications

Department of Software Engineering

Supervisor: RNDr. Petr Hnětynka, Ph.D.

Study program: Computer Science

Acknowledgments

I would like to thank my supervisor, RNDr. Petr Hnětynka, Ph.D., for his patient support, useful commentaries and corrections, and valuable advice.

CONTENTS

Chapter 1: Introduction....6
1.1 Motivation....6
1.2 Objectives of this thesis....7
1.3 Structure of this document....7

Chapter 2: Evolution of rich web applications....8
2.1 Evolution of internet standards and popular technologies....8
2.2 Rich internet applications....8
2.3 Convergence of Web and Desktop applications....9
2.4 On the gap between web and desktop applications....11
2.5 Pervasive adaptable rich internet applications....12
2.6 Challenges in development of desktop-oriented web applications....14
2.6.1 Data persistence and replication....14
2.6.2 Adaptability to ambient environment....15
2.6.3 Manageable development and deployment of application....17

Chapter 3: Architectural model and Implementation....18
3.1 Introduction to the development framework for PARIAs....18
3.1.1 Model of PARIA deployment....18
3.1.2 Isolation of data layer from logic....19
3.1.3 Synchronization and storage of data....21
3.1.4 Service oriented features....24
3.1.5 Final overview of the framework architecture....26
3.2 Implementation of the development framework....27
3.2.1 Main module....27
3.2.2 Data layer package....29
3.2.3 Filesystem module....29
3.2.4 Persistence module....30
3.2.5 Internal Derby database module....32
3.2.6 Hibernate synchronization module....33
3.2.7 Servlet bridge....35
3.2.8 Internal HTTP server module....37
3.3 Development tool based on Eclipse platform....38
3.3.1 Description of Eclipse PDE platform....38
3.3.2 OSGi module configuration and development....38
3.3.3 Running and debugging applications within the IDE....39
3.3.4 Building the final application....40

3.3.5 Building web application package.....41
3.3.6 Building desktop application package.....42
3.3.7 Installation as a desktop package from within a web application.......46
3.3.8 Tooling for development of user interfaces.....47

Chapter 4: Realization of a prototype application.....49

4.1 Prototype application.....49
4.1.1 Architecture of prototype application.....49
4.1.2 Data layer.....51
4.1.3 User interfaces of prototype application.....52
4.2 Evaluation of PARIA framework in the context of prototype application. .54

Chapter 5: Conclusion and future work.....56

5.1 Conclusion and evaluation.....56
5.2 Future work.....56

References.....58

Appendix.....60

List of figures.....60
Acronyms and terms.....60
Enclosed CD.....62

Title: Development of browser-based desktop-oriented web applications

Author: Ondrej Mihályi

Department: Department of Software Engineering

Supervisor: RNDr. Petr Hnětynka, Ph.D.

Supervisor's e-mail address: hnetynka@dsrg.mff.cuni.cz

Abstract: In accordance with new trends in development of web-based services, applications that are running on the Internet tend to be more complex and feature rich. This happens while computers and network bandwidth are becoming more and more capable. We may see lots of different, often inventive, approaches in rich web application development, which try to surpass the constrai nts of web environment. Unfortunately, few technologies or tools provide support for ubiquitous access to data. This thesis aims to propose a development method of rich web applications, which transparently provide ubiquitous access to user data in many various environments and conditions.

Keywords: web applications, Java, ubiquitous data, adaptability to environment, pervasive computing

Názov práce: Vývoj desktopovo orientovaných webových aplikácií dostupných cez webový prehliadač

Autor: Ondrej Mihályi

Katedra (ústav): Katedra softwarového inženýrství

Vedúci diplomovej práce: RNDr. Petr Hnětynka, Ph.D.

E-mail vedúceho: hnetynka@dsrg.mff.cuni.cz

Abstrakt: V súlade s modernými trendami vývoja webových služieb, aplikácie, ktoré bežia na internete, sa stávajú komplexnejšími a bohatšími na vlastnosti. To všetko sa deje, zatiaľ čo sa počítače a kapacita sieťových prenosov čoraz viac zlepšuje. Môžeme pozorovať veľa často inovatívnych spôsobov vývoja aplikácií, ktoré sa pokúšajú preklenúť obmedzenia webového prostredia. Žiaľ, málo technológií alebo nástrojov poskytuje podporu dostupnosti k dátam z ľubovoľného prostredia. Táto diplomová práca má za cieľ navrhnúť metódu pre vývoj košatých webových aplikácií, ktoré poskytujú transparentný prístup k užívateľským dátam v rôznych prostrediach a podmienkach.

Kľúčové slová: webové aplikácie, Java, všadeprítomné dáta, adaptívnosť k prostrediu

Chapter 1: Introduction

1.1 Motivation

In accordance with new trends in development of web-based services, applications that are running on the Internet tend to be more complex and feature rich. This happens while computers and network bandwidth are becoming more and more capable. We can clearly see the ascend of AJAX as a standardized technology, which helps to create interactive and highly responsive web applications. Such technologies help to cope with the natively request-response based character of the HTTP protocol for serving pages. This trend is further backed by growing number of JavaScript libraries, which ease the development of dynamic web components. The rise of multiple third-party widgets and component libraries, which seek to provide the desired eye candy to rich multimedia applications, is also evident. Moreover, on many occasions, lots of more or less standardized browser plugins, such as Adobe Flash, Java applets, or Microsoft Silverlight, are widely utilized to bring media and interactivity directly to the user in a pleasant manner.

In the light of these circumstances, contemporary form of the Internet is often vastly different from what it used to be like when it was in its beginnings. Initially, it was designed around simple technologies like HTTP and HTML. Today, modern web applications have the potential to compete with standard desktop programs, and still provide easily accessible Internet-based services. Such applications create a comfortable abstraction of an application as a tool accessible anywhere (running on Internet), without the need to tie it to a specific computer. However, nowadays many standardized technologies are far beyond developers' needs to create highly usable, media-centric, visually appealing web applications.

As a result, we may see lots of different, often inventive, approaches in rich web application development, which try to surpass the constraints of web environment. Many of them go positively hand in hand with new evolving standards produced by organizations such as W3C, and technologies provided by many commercial vendors. Unfortunately, few technologies or tools provide support for ubiquitous access to data, and possibly no substantial effort has been made to bring such tools to Java platform yet. Tools to help develop applications deployable both in standard web environment and on a desktop are also scarce.

1.2 Objectives of this thesis

The thesis aims to propose a development method of rich web applications, targeting their ability to run in both web and desktop environments. This proposition is based on analysis of contemporary technologies and suitable development strategies. The method should simplify development of applications, which transparently provide ubiquitous access to user data in many various conditions. Among these conditions are connectivity to a central network storage and type of available runtime environment.

The emphasis is put on developing web applications, which rely only on features available in standard web browsers. Applications may be extended by installing a runtime support in a desktop environment. Provided that, they can make native features of local system accessible to users and behave like usual desktop programs. They also can provide a local storage of data in case of connection outage.

The development method integrates suitable approaches, standards and Java opensource technologies. It is supported by a development framework and tools implemented and described in the thesis. These simplify and assist in development of ubiquitous web applications, which are based on Java platform and common Java and web standards.

1.3 Structure of this document

Chapter 2 discusses evolution of internet applications and their convergence to desktop applications. It clarifies common terms and analyzes the differences and similarities between web and desktop applications. The rest of the chapter introduces a new term for the next generation web applications and discusses the challenges needed to overcome to build such applications.

Chapter 3 proposes architectural model and a method to develop these applications. It discusses and supports the solutions to the challenges mentioned in Chapter 2. It later describes a development framework integrating these solutions into a development platform. Chapter 3 is finalized by introduction to development tool to support development of applications based on the framework.

Chapter 4 describes how the framework from Chapter 3 can be used to build a functional application. On an example of a prototype application, this chapter also evaluates the advantages of using the framework over other available frameworks and tools.

Chapter 2: Evolution of rich web applications

2.1 Evolution of internet standards and popular technologies

In the beginnings of internet era in early 1990-ties, there were no web applications as we know them now. Internet standards evolved in order to support simple request-response WWW sites. These were based on HTTP protocol for transport of web pages, HTML language to describe and present the content. Later, additional standards were specified, which enabled growth of web sites into interactivity through scripts and visual richness (animated images, mime-types, etc.). Soon it was discovered, that richer applications are managed better through dynamic scripts rather than using plain html files. Thus, web sites evolved into more complex internet portals, making use of both browser and server side scripting languages (Javascript, PHP, Perl, Python). After asynchronous XML communication had been introduced in HTTP protocol, it represented a significant step in interconnecting browser and server script engines. It alleviated applications of the necessity to retransfer whole state of user interface upon posting some information to the server. Thus applications began to evolve into more and more interactive, feature-rich and responsive services. The abrupt expansion and acceptance of AJAX as a new approach to creation of web applications confirms that asynchronous communication had been eagerly awaited

Java, as a web platform, has been evolving for years in parallel to the evolution of modern technologies. It offers solutions not only targeting regular web applications, but it has continually evolved into mature enterprise platform. It has support for majority of internet-related technologies and even brings unique concepts in application development and deployment. Servlets, JSP, EJB, Web Start, Java FX and Java browser applets are some of those worth mentioning.

2.2 Rich internet applications

Most modern web applications with rich user interface are referenced at as rich internet applications. In [Mor08], based on a Macromedia white paper [All02], the author summarizes RIAs as internet applications, which should support as many aspects as possible of a classical Desktop Application offering a "rich user experience". Further in the text, he remarks that the definition of RIAs does not mention a Web browser. This implies that the browser functions only as a widely used media and a common platform to access the internet application. It is not required that all RIAs are browser based.

Many existing rich web applications are built around more or less standard and widespread browser extensions or plugins. These plugins, if built solely on such extensions, could be possibly deployed and run even outside of a browser. In fact, there already are efforts to provide no-browser runtime environments for RIAs. It is possible by providing a standalone version of a certain browser plugin, as presented by Adobe AIR. Another solution is to provide whole underlying internet browser engine that can be encapsulated into the application (Mozilla XULRunner). Or a web application can be encapsulated into it a container to come closer to desktop experience (Mozilla Prism, and also Adobe AIR). In Java world, Java FX provides a solution to build applications, which are executed primarily outside of browser through Java Web Start, with plans to support deployment through Java Applets. Thus the solution is to provide runtime support in browser and outside of browser, similarly to Adobe AIR.

2.3 Convergence of Web and Desktop applications

To provide richer user experience, many web applications present user interface, which resembles interface of classical desktop applications (DA). Some web applications even tend to be a web alternative to DA and their user interface strives to be as complex and feature rich as desktop user interfaces. There are tools to create web applications close to as responsive as desktop counterparts, to bring some form of data storage and user customization, even to access local desktop environment.

In the thesis, we will refer to some widely used but not commonly defined terminology regarding web and desktop paradigms. These terms are often vaguely used and their exact meaning varies in different conditions. We will comply with web and desktop naming conventions summarized in [Mor08].

Desktop applications are applications that need to be installed on an operating system. A classical desktop application is, for example, a word processing application like Microsoft Word, or a graphics editor like Adobe Photoshop. Desktop applications with Internet connection or *Internet-enabled applications* are applications that use network support, but also run offline, perhaps with limited functionality. For example an e-mail client like Mozilla Thunderbird or Microsoft Outlook.

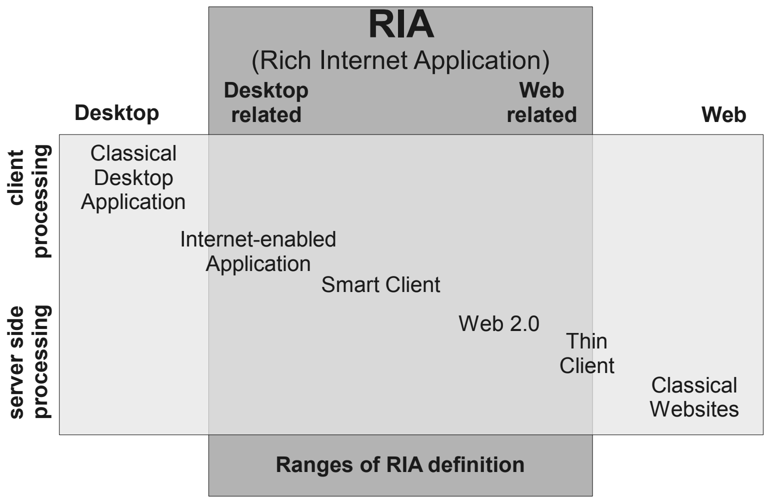


Figure 1: RIA categorization (from [Mor08])

On the other hand, there are classical w*ebsites*, W*eb 2.0 applications* and t*hin clients*. They rarely have to be installed by a user. They can be started and loaded via network. A running connection is mostly required to use them. Websites, W*eb 2.0 applications* or thin Clients very often run in a Web browser.

The technologies described by the term Rich Internet Applications (RIA) are located in the middle or between more Web-related and more Desktop-related technologies. Many technologies and kinds of applications can be summarized as RIAs depending on the point of view. Some run inside a Web browser, others without. But a common characteristic aspect is, that RIAs benefit from the best of traditional Desktop Applications and of Web applications.

The convergence of desktop and web applications is exemplified by mature rich internet applications, which offer functionality nearing the features of desktop applications. Such web applications are often embedded into their own web universe and provide the same functionality and data all over the Internet. Examples of such applications are office tools in Google office package. These tools provide full featured browser-based user interface to edit documents in similar way to desktop office packages, such Microsoft Office. The difference is that the documents are stored in remote web folders, so that they are accessible anywhere.

2.4 On the gap between web and desktop applications

Traditional web applications often have to tackle with problems unknown to desktop applications. They are unable to run without the Internet connection, there are issues with synchronizing desktop (browser) part of the application with the server side. Some RIAs try to overcome these problems. They extend the possibilities of web applications with frameworks and solutions, which support coping with these problems to some extent.

Probably the most familiar representative of such frameworks is Gears under auspices of Google software department (gears.google.com). This project provides for extended features with the help of an extra browser plug-in for each mainstream browser and OS platform, and with the help of a JavaScript API library to access the features. Applications developed under this framework may be built using standard Javascript and server-side frameworks, such as GWT, DOJO, ZK, and others. They are provided with an SQL-like JavaScript database API for local storage of data, tools for detection of online and offline states, tools for synchronization of local and remote data after going online, and other additional functions accessible from JavaScript. The heart of Gears is the LocalServer module, which functions as a transparent proxy inside the browser plug-in. The LocalServer module handles all remote requests of a Gears-aware application and substitutes remote http server when the connection is lost. Thus the application can be started even if no connection is available from the beginning. This solution is solid and offers reasonable tools to build a desktop-oriented application. However, it is built for cost of tight integration of the plug-in with the browser. Thus there is need to maintain different versions of it for all supported browsers and platforms, what may prove itself as a bottleneck in larger adoption.

HTML 5 offline technology, on the other hand, provides more standard way to cope with loss of internet connection. This solution provides similar features to Google Gears without the need to rely on a non-standard browser plugin. However, HTML 5 specification has not yet been officially released. It has not been adopted by majority of browser vendors and thus is not yet a working solution. Firefox in version 3 supports a subset of HTML 5 offline specification, but this support is still experimental.

Java FX solves the problems of web applications in a different way. Applications built on Java FX are run as standard Java applications. They can thus be granted access to desktop through the Java Runtime Environment. These applications are distributed through Java Web Start and executed in a sand box. This sand box limits some of the features available to desktop applications.

However application in a sand box can be granted all the standard privileges if users allow it. Security is ensured by allowing to grant certain privileges only to signed applications. Java FX provides a platform powerful enough to build solutions to overcome usual problems of web applications mentioned above. However, it does not provide any concrete solutions or tools to help in development. Another disadvantage of building RIA on top of Java FX is that the platform is built on top of standard Java platform and native operation system features. Java FX seems to be promising in future, but nowadays only support for some platforms supported by Java platform has been provided. RIAs aiming to be multi-platform still have to be built around standard Java technologies.

2.5 Pervasive adaptable rich internet applications

Existing efforts are definitely close to bridge the gap between web and desktop applications. Even though, they in general don't focus on synchronization of application data between online and offline states. Nor they offer any substantial advantage from running the application in a desktop environment apart from putting it outside of a browser, so that it can be run and administrated as a desktop application. Therefore, to distinguish between traditional RIAs and applications targeted in this thesis, we will introduce a special category of RIAs referenced at as Pervasive Adaptable Rich Internet Applications (PARIA). These rich internet applications are developed with the emphasis on ubiquitousness of application and data availability and on adaptability to advantages and disadvantages of available desktop or web browser platform.

The term pervasive comes from pervasive computing. It is a model of a software interaction, where a particular instance of an application is understood mainly as a portal to ubiquitous data accessible from any other instances of the application on any supported devices. As stated in [GDLX04], pervasive computing is motivated by the observation that computing and networking technologies are becoming increasingly powerful and affordable. The goal is to provide people with universal access to their information and seamlessly assist them in completing their tasks. This enables that, in contrast to conventional computing environments, people focus on their activities and not on the computers.

Ideally, [UWas02] proposes that pervasive applications should be built according to three key axes: separation of application data and functionality, availability of any resource/data at any time, and a common system platform available across many different environments. These three design aspects should be common to all pervasive applications in order to become viable and highly usable in real world. This work takes these aspects into consideration and attempts to embed them into proposed model of development of PARIAs.

Further, the term adaptable embraces the manner, in which PARIAs cope with limitations and drawbacks of interconnected environment and availability of application data under these limitations. It also stands for the capability to adapt to certain environments and provide the most of it with regard to available resources, permissions and the nature of the platform.

The range, in which an application adapts to the ambient environment, may be virtually limitless. It may access simple resources, such as files or programs, on desktop or a device. It may be able to install and integrate itself into the native platform. It may run with a native, and thus different, more responsive user interface, which is not bound to e.g. web browser limitations. And the list goes on. This all can be accomplished by a unique architecture of the application, which provides means to decouple application's functions into separated modules. These modules are then detected and used as necessary and appropriate. As an example, we could imagine a mail program, which could operate as a standard web-mail when accessed through browser. But after it detects that a supportive desktop platform is available, it could provide more suitable means to attach files from local computer to outgoing messages, or add a notification to system tray when new mails arrive. Nowadays, there are means to achieve this kind of behavior, for example when a program is designed as a Java applet and manually detects whether permission to access file on hard disk is available. However, there are even now many challenges in practicing this behavior, because few to none technologies exist with primary focus on development of such applications, especially in the Java world.

2.6 Challenges in development of desktop-oriented web applications

There are many challenges that emerge, when attempting to design and create an application that would match a model of PARIA. They are discussed in the remaining of this chapter in order to establish a solid theoretical background for the proposition of a model and a development framework for PARIAs.

2.6.1 Data persistence and replication

PARIAs are in essence data-centric applications with support for ubiquitous data availability. Their foremost feature is to persist data not only within a device, but distribute it along whole universe of devices and platforms that could be used to connect to the data. These are namely desktop computers connected to the Internet and all devices that can run applications available on the Internet.

It is essential that data are accessible in any condition and that the process of manipulation of data is optimized to the extent, which is allowed by available underlying platform. Therefore, if feasible, the process of distribution of data has to be backed-up by a locally stored replicas of the data on desktop clients. This would also improve the chance to safely cope with connection outages. In the case when internet connection is not available, it also has to be ensured that changes to data made locally will be synchronized with data on the internet later on. Data synchronization is crucial, because it is required in order to provide the same user's data everywhere the user connects to the network. Data synchronization is a process, which aims to establish consistency among data from different sources and the continuous harmonization of the data while the sources are connected. It is to be triggered each time an application goes online and has to resolve any inconsistencies between locally stored data and data on remote storages. The implementation of data synchronization could be a little simplified if we consider only centralized topology apart from peer to peer synchronization, but still poses a significant challenge.

With a view of data as an arbitrary information, another inconvenience arises. It has to be decided, which data are to be persisted across the network, how they should be manipulated locally by the application, and in which form they should be transferred and stored. [UWas02] discusses the relationship between data and functionality and differentiates between active and passive data. In particular, the encapsulation of data into active objects is evaluated and then refused as an inflexible and insecure approach. On the other hand, it is insisted on separation of data and functionality, as it is usually easy to accomplish and much more suitable and transparent for data distribution.

In almost all cases, when developing a software application, which manipulates with some structural data, it seems very reasonable to keep the data separate. Data should be stored in structures separated from logic. They, together with logical components and runtime environments, should serve as a unifying abstraction similar to combination of a file system and nested processes in traditional operating systems. It is the data, which are manipulated by users or by automatic operations, and which have to be stored between multiple executions of the program or shared among various entities. At the end, it is the data that users access and work with. Applications provide means to access the data. If properly separated from the rest of the application, data are even accessible by different programs and may be further processed beyond the capabilities of the original application.

One of the most important aspects of PARIAs is the emphasis on ubiquitous data availability. The data-centricity of an application is so profound concern that it has to be built into the very basis of inner architectural model of the application. This ensures that all data, which should be persisted and shared, are easily distinguished from runtime data and application logic. While in a generic software program, the separation of data and the rest of application is possible but optional, the model of PARIA literally requires some form of separation of data, so that it can be extracted from application instance and transferred over the network.

In addition to this, if the application should be built using a certain framework for data transfer and synchronization, the data layer must be transparently accessed and described. In an object-oriented environment, the data layer may be perceived as isolated objects with a clearly defined interface to access data. An example of such data isolation is a DOM tree. It is well specified, can be exported as a XML document for storage and transportation and it is widely supported in JavaScript, Java and other common platforms.

2.6.2 Adaptability to ambient environment

The challenge with making an application adaptable is primarily related to the overhead of detection of available resources. It has to be done either at application start-up or even while the application is running, dependent on volatility of resources. In conditions, where unavailability of certain resources is rather extreme exception, such resources are usually abstracted in a form, which simplifies development. Remote invocation, such as CORBA or RMI, may serve as an illustration of this abstraction. Masking remote resources as local is easily understandable and very suitable in situations with high certainty that remote resources are almost as available as local resources. As the difference between levels of availability of local and remote resources grows, remote invocation usually fails to provide enough tools to cope with non-standard conditions.

In various environments, specifically the combined desktop and web environment, a high abstraction of resources is not flexible enough as to help build an adaptable architecture. In particular, desktop client may have more features available in its runtime environment. Web client is not tied to one computer and may provide unlimited accessibility to data for the price of limited features, in the name of security or other reasons. There is hardly any suitable way to abstract the features that are available in one platform or in the other, but not both. Even similar features may be implemented differently. There has to be a way to adapt at any point, when resources become or cease to be available or when the underlying platform changes.

The two essential cases, which have to be cared of, is modules and services. Modules are basically available always in the particular environment and are either loaded or not loaded when application starts. Services may be loaded at any time but may happen to be inaccessible for some time during application execution.

Modules encapsulate the functionality, which is common only to certain environments and thus cannot be took granted universally. The availability of such modules is thus either manually wired into application configuration, or detected at application startup. Their availability doesn't change throughout execution of the application and thereby they may be statically referenced if once present. Sometimes there may be different modules with the same functionality, but each with different implementation, which is either limited to certain environment or seeking some sort of optimization. Static modules may represent features like:

- direct access to files stored on local computer
- persistent data storage
- system tray icon
- authentication (in case that data storage provides for more users)

Services, on the other hand, have to be detected each time they are requested, because there is no way to ensure their availability. Services encapsulate features, which rely on circumstances that are not fully under control of the application. The have, otherwise, all the characteristics of modules, but because of their volatile nature, they have to be accessed and treated differently. With services, there is much more emphasis on handling of the situations, in which required services are unavailable . In such cases, it is needed to provide for supplemental treatment, so that application may further operate under limited conditions and normal flow of operations can be restored as soon as the service becomes available. The single most important example of such service in PARIA is open network connection to the central data storage.

2.6.3 Manageable development and deployment of application

Another difficult task for a PARIA, which is not that apparent, is to provide users with suitable method to start and access the application, no matter in which environment it is used. In the case of a single web client, the solution is rather simple, as the application is simply accessed and executed by loading a web page into a browser. However, if user should install and use a desktop client, she has to be provided with a simple solution.

This goal implies that a PARIA has to be developed around a widely used platform that is common to many devices and environments. For one reason, it simplifies the development by focusing on one platform. This brings its fruit in maintaining a single code-base and thus less errors in final product. Another reason is that the final code may be executed on wide variety of software platforms. There already exist widespread platforms that fulfill the needs of PARIAs, among them web browser and Java platforms. These, if combined together by suitable means and technologies, may provide a powerful capable platform. For the sake of pervasive applications, an example of utilization of Java platform is one.world ([GDLX04], [UWas02]). It extends Java platform and provides a single application programming interface (API) and a single binary distribution format with a single instruction set. These can be implemented across the range of devices in a pervasive computing environment, which are supported by Java platform.

Furthermore, the common platform should provide means of manipulating with data so that they are separated from logic. The form of data storage and representation should be easily accessible and integrated into the platform. It should be implemented in a standard manner easily understandable by developers and should not introduce any unfamiliar concepts or constructs that would limit productivity.

Chapter 3: Architectural model and Implementation

This chapter proposes solutions to problems and challenges, which were posed in previous text, founding them on existing standards and technologies. The experience from a case study in [AGX02] shows that ubiquitous applications *have many aspects in common* and thus can benefit from a common architectural model. Based on this assumption, we describe a framework providing competent infrastructure for building PARIAs, so that they can be built according to a proposed architectural model. In the end, a tool for building PARIAs based on this infrastructure is described and demonstrated.

3.1 Introduction to the development framework for PARIAs

3.1.1 Model of PARIA deployment

The applications targeted in the thesis, are in their principle web applications and therefore their primary deployment is on the internet. As such, it has to be made possible to build the application as a standard Java web application. Internet browser thus plays the important role of an access gateway. Although, it is itself a very limited platform to provide for deployment of all pervasive and adaptable features, unless it is extended by a necessary plugin or a desktop runtime. Some forms of RIA are distributed as a package, which is inherently dependent on presence of some extra runtime, such as applications based on Java FX and Java Web Start, or on Adobe Flash. After their preconditions are met, applications can be easily extended as far as the environment and granted privileges allow. This work is, however, based on the assumption that a pervasive web application should be accessible from any standard web browser regardless any non-standard runtime support. This assumption concludes in necessity of deploying the final application in two separate builds, which are afterwards integrated for seamless and adaptable distribution. One of the builds should be a web application based on standard browser environment present natively in all modern browsers. The other build should targeted the desktop computer environment. The presence of the latter enables users to choose to download and extend the application with features available only within the desktop environment. This enables users to retain access to the application regardless of the quality of internet connection. The Java platform provides for ideal common ground to develop both web server component and desktop component, as it is extensively supported by many mature tools and enterprise vendors on the web platform. It is also widely available and present on almost all desktop computers and even other computer devices.

Deploying a web application using Java servlets is the most convenient form of Java web deployment. It will be considered as the only suitable form of deployment for our model of PARIA. Java servlets are well supported by all Java web application servers and their use is in detail specified in [Sun07]. On the other hand, the choice how to deploy and make available the desktop client is not so apparent. The simplest solution is to deploy the desktop part as a traditional Java application packed in a jar Java archive. Nevertheless it is not very flexible and usable, therefore we make use of a more suitable Java Web Start technology in this work. For our purposes, it is much more convenient than a Java applet, because an applet is too much tied to a browser and the web part of an application. Furthermore, it is nowadays the most widely adopted form of serving Java applications on the internet.

Java Web Start technology frees the developer from concerning with how the client is launched. It provides means to package an application as a component, which can be either launched from a web page or from desktop environment. Additionally, this technology provides a mechanism that automatically installs applications into desktop environment. Moreover, it supports a scheme that enables a Web server to independently distribute and update client code ([Kim01]). The application is thus accessible also if the page, from which it was distributed, is not available at that time.

3.1.2 Isolation of data layer from logic

In the Java platform, the most natural way to implement the data layer is to use simple Java objects. Among available tools helping with building the data layer on Java objects are various forms of object libraries. These libraries provide interfaces to manipulate and isolate the data layer.

Our approach is to use the Java reflection API and a subset of standard Java interfaces. Java Beans specification ([Sun97]) and Java Collections Framework ([Zuk01]) have been chosen to standardize the form of data objects. These two specifications are powerful enough to design various complex custom data structures, which are still universally accessible. This solution frees the programmer from the requirement to adjust to a certain API and data structures available in the library. it also opens the possibilities to extend the data layer with event handling, easy object persistence and much more, all using tools built around the same specifications. On the other hand, the DOM API was also evaluated as a viable alternative for data layer. It provides its own methods and interfaces to store objects in a tree-like structure and also provides means to store and export the data. However, it appeared to be much more usable when working with tree-like data and persistent storages based on XML. Combination of Java Beans and JCF (Java Collections Framework) has been evaluated as more convenient in all other situations. Java Beans and JCF are both designed directly for Java environment. They are well specified and supported by many mature tools that are available. Whole data layer is built around simple Java objects and that provides a familiar way to work with data with almost no real restriction on how the data structure has to be implemented.

In general, there are two approaches to implement access to isolated data layer. First, and probably more evident, is to keep all data separated in one place and provide for transparent accessibility to Java objects through Java Beans property accessor methods and JCF interfaces. In this case, instances are free to manipulate with all the data that is accessible. On the other hand, it contains limitations concerning information about lifetime of the data. Second approach is to expose events to listeners when the data is changed. Thus data are not directly accessible, only changes on data are processed by external listeners. These have to store the previous replicas of data for themselves to rebuild whole data layer if needed.

These two approaches of data isolation and accessibility also have an impact on the way the data are synchronized and transferred across multiple data storages. In case when actual state of data layer is directly accessed and no object lifetime information is available, all the data have to be transferred and compared to the data in the other storage. On the contrary, the second approach leads to transferring only the changed data, which may be inserted or updated in the other storage. This storage has to contain an existing replica of data layer before the change.

Both approaches are used in our framework. They don't interfere with one another and may be implemented at the same time. Runtime objects represent the separated data, which is easily accessible through accessor methods of objects. Thanks to the many tools available for standard Java objects and Java Beans specification, this data layer can be extended to emit events when the data is changed. This can be achieved using events in Java Beans specification together with bytecode manipulation libraries, such as Javassist. There are tools to help manipulate data in a pleasant and familiar manner with little effort and code (using e.g. JXPath for searching through data). Event processing is, on the other hand, the core part in data synchronization. Events are triggered after modified objects are persisted, and a special listener is used to store meta-data about changed objects. These meta-data are later accessed directly by the synchronization mechanism to find out about changed objects.

3.1.3 Synchronization and storage of data

In a PARIA application, there are two types of storages, in correlation with two types of client instances. The web client operates on a central remote storage, which is placed on a remote server (most likely on the same machine as the web server). In the case of desktop client, there is a local data storage. It is steadily accessible and is used primarily when remote storage is not available. Thus, the shared storage of PARIA is oriented around one central master storage and multiple local slave storages.

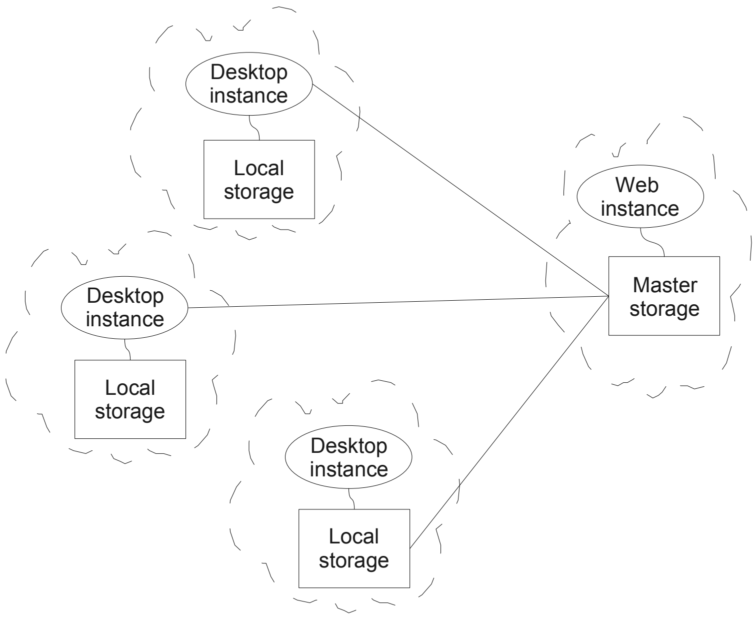


Figure 2: Topology of local and remote storages – one master storage and many local storages. Desktop instance accesses both master and local storages

The topology of storages is very important for design of data synchronization strategy. Based on the assumption that PARIAs are always built around central web application, and for the sake of simplicity, this work considers only master-slave synchronization. It is most commonly needed in normal usage. The central master point serves as a checkpoint for all other slaves. Thus there's only need for two-way synchronization of data between master and slave. Additional synchronization among client instances may be elaborated in future work, but is beyond the scope of this thesis.

A two-way synchronization strategy may differ in whether the process of data synchronization is synchronous and happens instantly, or it is asynchronous and happens invisibly in the background. In either case, the data must be stored locally to instantly make accessible all the data when the connection goes off. Provided that the synchronization happens synchronously in online conditions, data changes are sent through the network to appear in the remote storage and are locally stored as well at the same time. Remote data are always fresh and thus there is no need for additional synchronization. However, local operations have to be blocked until data changes are transferred to remote side. Utilizing the asynchronous strategy, local storage is treated as primary and is lazily synchronized with the remote storage in the background. This is done either by logging operations on data and sending the log in a non-blocking way, or periodically scanning for differences between local and remote storage.

If we want to have the desktop client to behave and feel as close to a native desktop application as possible, the synchronous data synchronization is not an option. It doesn't cope well with latency in network environment and may cause the application to slow down or even block itself for some time. On the contrary, the asynchronous strategy is very suitable and is more flexible in high latency network environment, or even environment with frequent connection dropouts. The data are always stored locally and thus always accessible. The synchronization mechanism stays the same in online condition and also when the connection is reestablished.

Approach that is used in this work is to completely separate the process of local data manipulation and the process of data synchronization with remote server. Thus, synchronization of data is completely independent from the actual data manipulation. It also doesn't require that the remote server is available, and it might be triggered at any time if appropriate events happen (e.g. at application startup, after connection becomes alive, previous synchronization attempt failed,and so on…). From the development point of view, it is also very suitable that the process for data synchronization is run separately in a background thread, independent of main thread of the application. In such case, both were developed separately and combined in a suitable manner as two intercommunicating processes, which share data through a database layer. Applications that are based on our framework are thus easier to test, to define and check preconditions and postconditions, and to recover from failures.

An alternative approach would be to transform the simpler synchronous strategy to an asynchronous one. This can be done by logging all the local changes into a queue and later propagating all the changes in background in the order of their creation. This may be referred to as a lazy synchronization, because it simply changes the remote storage in the same way as the data were changed locally, except in a latent manner. Nevertheless, there are substantial reasons why this approach has not been chosen. This approach may expose difficulties if the remote data were changed while the desktop client was in offline state. That may lead to conflicts, which are tough to resolve only with the data stored in the queue of changes. Moreover, logging all possible types of data alteration introduces even greater complexity.

The web part of a PARIA application, as a central point of all distributed instances of the application, has its own storage, which has the precedence before other local storages. This storage represents the actual state of shared data, which is known to all other distributed instances. On the other side, there are storages that belong to local instances of the application and they represent only the actual state of data within the particular local instance. Based on this fact, there is only need to initiate the synchronization of central and local storage from local instances The central storage is considered always fresh. The synchronization module hence has to be included only in local instances. On the other hand, the web instance has to provide access to persisted data and any required additional meta-data through a network protocol

Another issue needed to be resolved was which data and on which layer will be actually shared. This appears to be a subtle matter, because shared data may or may not include in-memory structures not yet persisted into the storage. However, if considered that only final and approved changes are worth synchronizing, shared data layer is best built only on a defined subset of persisted data. Our decision was hence to put the process of synchronization on the level of persisted data in storages. Thus the process of data persistence and data synchronization overlap. They both may make use of various advanced features of persistent databases, such as transactions or unified data retrieval. This decision is supported further by additional arguments. First, in research preceding this work, no viable form of suitable in-memory data synchronization could be constructed using existing stable Java tools. More often there exist tools, which help with replicating persisted data between databases. Another argument is that achieving transparent data accessibility is easier through common database interfaces than by Java interfaces. A proved database solution involves in-built support for atomic access, security and enforcement of data singularity and persistence. There even exist very mature Java-based database solutions, which provide means for extending the data layer with data synchronization mechanisms.

The data and synchronization layer in our framework for PARIA development utilizes Hibernate Java persistence framework and DB4o replication system ([Drs09]). They together with an underlying SQL database provide a robust Java object persistence platform. Java Beans and JCF are fully supported by Hibernate Its engine is then extended by DB4o event listeners, which utilize the event based approach to isolated data in order to make data synchronization more efficient. DB4o listeners record data changes on both master and local database separately. When the data synchronization is later invoked, only the changed data are transferred. In order to ensure that all data were replicated, it is also possible to amend this basic behavior with a less frequent complete scan of data for changes between the two databases.

3.1.4 Service oriented features

Ambient environments, in which PARIAs run, may vary, and some services may even be heavily volatile. As such, not only it is suitable to design the architecture of PARIA based on modularity, but it has to be also designed to adapt if certain modules or services are not available. This problem is too complex to develop a solution for each application independently. Fortunately, the pattern of software modules and components used as building blocks has evolved gradually throughout software development history. Lots of mature projects and activities have emerged to support this pattern. This allows us to use a certain common well established framework to build the needed modular architecture.

The thesis aims to utilize the most common and widespread technologies. The proposed architecture is based on OSGi platform ([OSG07a]), as it is one of the most popular and widely supported platforms oriented on services and modularization, with enough development tools built around it. Moreover, according to [HC04], *the initial focus of OSGi was the market of home services gateways, where the vision was that a house would contain a home-area network and most, if not all, household devices would be connected to this network.* The fact that vision is close to the mission of pervasive and adaptable applications even more supports the decision for OSGi framework.

Now let's have a look on how the needs of PARIA, which emerge from the requirement for adaptability, are fulfilled and implemented using OSGi.

According to the discussion in 2.6.2, we logically divide application components into those, which are hard-configured to be available in certain deployment, and those, which are not known before application startup if they are available and have to be detected either once upon application invocation (if the detected feature is assumed to be always present) or periodically throughout its execution (if the feature may disappear at any time). The first we call modules, the second we call services. In our implementation, services are not independent components. They are only interfaces provided by underlying modules, if the required conditions to administer the service are met. Modules present separated software components and may be differently configured in either web application deployment or desktop deployments (even different configurations for various desktop platforms are possible).

The OSGi service framework provides a simple, light-weight framework for creating service-oriented applications. It introduces a concept of extensible application components, called bundles. In our framework, each application module is implemented as such a bundle. These bundles are loaded and launched automatically according to the configuration — the underlying framework runtime detects and solves dependencies, controls life-cycle of bundles and provides utility mechanisms and additional bundles for the developer. To accomplish that, the framework runtime uses a custom class loader. This class loader also manipulates classpaths and handles dependency resolution of bundles upon their loading. Separate classpath is kept for each bundle.

Modules register their services through OSGi service layer. They publish each service under a well-known Java interface in OSGi service registry. Services are later retrieved from this registry by modules, which want to use it. In our implementation, modules, which provide some functionality to other modules, automatically register this functionality as an OSGi service. This kind of service we call *static* service. Both hard-coded components and detected services are thus discovered by other modules via the same mechanism in a unified manner. To even more simplify working with modules and services, all are expected to be possibly non-present by dependent modules. This enforces that the access to modules and services is error prone, while still simple enough. Static services are marked by special empty Java interface and may be assumed to never disappear, but this behavior is discouraged.

3.1.5 Final overview of the framework architecture

The architecture of the framework is based on separate OSGi bundles, which run inside of an OSGi framework. These bundles together form the necessary infrastructure to develop applications. Developers include their application components by adding new bundles into the framework.

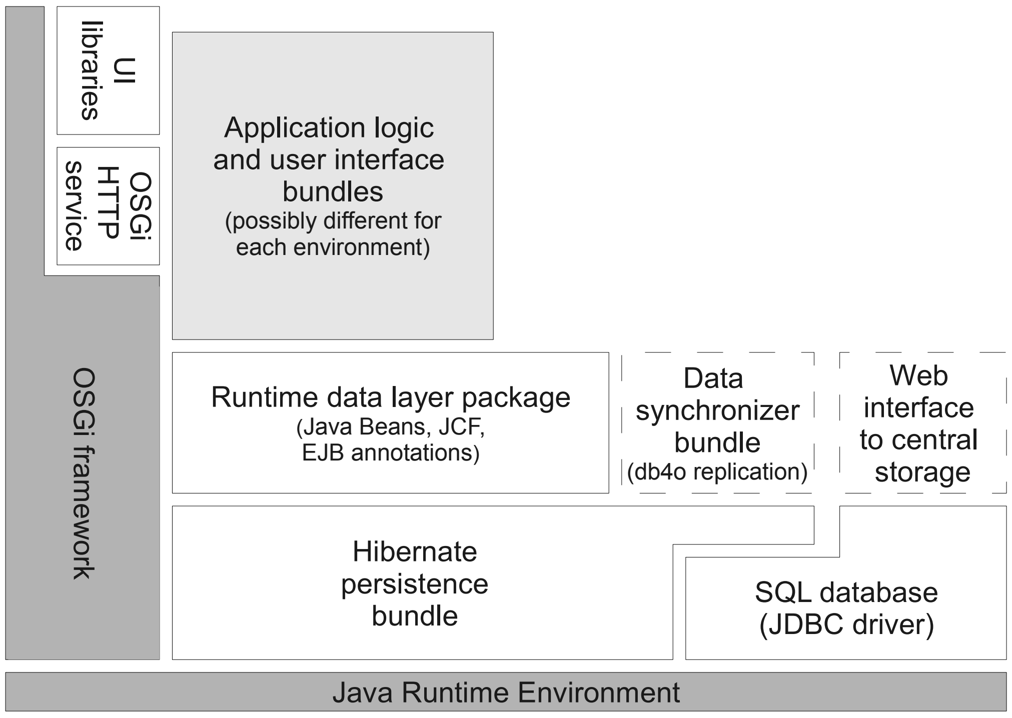

UI libraries
OSGi HTTP service
Application logic and user interface bundles
(possibly different for each environment)
OSGi framework
Runtime data layer package
(Java Beans, JCF, EJB annotations)
Data synchronizer bundle
(db4o replication)
Web interface to central storage
Hibernate persistence bundle
SQL database (JDBC driver)
Java Runtime Environment

Figure 3: Schema of framework with all the key parts. Shows OSGi component layers in horizontal axis, and data layers in vertical axis.

The most robust part of PARIA framework consists of data-centered components. These components provide a multi-layered data model built on opensource Java technologies and a JDBC protocol. Figure 3 illustrates how the main framework components are related. The vertical axis shows all the layers of data model. On top is the application logic and its user interface, which are individual to each PARIA application. Application logic operates on runtime data layer. This data layer consists of plain objects, which either comply with Java Beans specification or provide JCF interfaces. These objects are tied to the Hibernate engine. The runtime data layer and Hibernate are tied through an exported data layer package. It contains class definitions of all data layer objects. All the modules, which operate on the data layer, import this package to access objects class definitions. Hibernate handles changes on objects and provides automatic persistence mechanism. For this mechanism to work, it only needs that persisted

Java classes are marked by EJB annotations. This requirement is necessary to map object classes to underlined SQL tables. Hibernate stores objects using a JDBC driver, which connects to a relational database. This database is configurable and doesn't have be a part of the Java application.

The data synchronizer bundle runs on top of Hibernate engine separately from all others.. This bundle is present only in local desktop instances. It periodically replicates data in a background thread with the central storage. On the other side — on the central web instance, there is a web interface to connect to the central database. This is realized through an open database port, or more conveniently, through an HTTP database bridge.

There are also other OSGi bundles, which are not meant to support the data model. These bundles contain UI libraries, provide OSGi HTTP service to deploy servlets, and other additional functionality.

3.2 Implementation of the development framework

3.2.1 Main module

OSGi manifest fragment: Main bundle

```
Bundle-SymbolicName: cz.cuni.mff.paria.main; singleton:=true
Import-Package:
   javax.servlet;version="2.4.0";resolution:=optional,
   org.eclipse.equinox.servletbridge;version="1.1.0";resolution:=
   optional
Export-Package: cz.cuni.mff.paria.logging,
   cz.cuni.mff.paria.main, cz.cuni.mff.paria.serviceregistry,
   cz.cuni.mff.paria.services, javax.persistence
Bundle-ActivationPolicy: lazy
```

Exported services: Main bundle

- `cz.cuni.mff.paria.services.DesktopPackageProvider`

OSGi bundle `cz.cuni.mff.paria.main` is the main module in our framework. It is by default included in dependency list of all other framework bundles. It contains common utility packages and libraries, which are used by other modules, such as logging or EJB annotations. Its main purpose is to tie together all framework modules and services, to monitor their activity and availability and provide access to them.

It contains a simple hash-based service registry implemented by `ServiceRegistry` class. It wraps standard OSGi services mechanism and simplifies access to available services that were published by other modules. It also ensures that the main module and all available modules are started before they are used. `ServiceRegistry` makes use of a standard OSGi `ServiceTracker` utility class. It provides methods to acquire a reference to object that implements required interface or to test weather an implementation is available..If methods provided by `ServiceRegistry` in main module are not sufficient to fine tune the application behavior, standard OSGi service mechanism still may be used. In this way, services are accessed using full name of the Java interface as a key (return value of `Class.getName()`). However, when using `ServiceRegistry`, it ensures that the whole OSGi framework with all the modules configured to start is started prior to prompting for service reference in other modules. Static services are hence started first, and only then the reference is returned by `ServiceRegistry`. These services are thus always available from the start also if modules are launched in parallel.

The main module also exports interfaces for common framework services, which are provided by other framework modules or could be provided by custom modules. These are grouped in `cz.cuni.mff.paria.services` package:

- **`interface`** `FileSystem` **`extends`** `StaticService`
  - a convenience service to distinguish weather underlying filesystem is configured to be accessed in standard way, or it is forbidden
- **`interface`** `Persistence` **`extends`** `StaticService`
  - service that provides methods to persist and retrieve Java objects
- **`interface`** `DesktopPackageProvider` **`extends`** `StaticService`
  - service providing URL to the JNLP descriptor file of available desktop client

This list of services is not comprehensive and includes only basic services needed to build a simple PARIA application. It could be extended by other helpful services in the future. Other services may be added by the developer of final application and need not be defined in main module. Also some modules in our framework offer their own specific services.

The main module also exports its OSGi activator class, which provides methods to transparently access common system and framework functions. For example, this class is used to stop the whole OSGi application with all the modules in the proper form by calling `stopApplication()`.

3.2.2 Data layer package

The data layer classes are expected to be accessed from every module that needs to work with the data objects and thus is required to have their class definition on its classpath. Since OSGi bundles have completely separated classpaths, there have to be some form of bridge between classpaths of the module, which implement the data layer and of other modules, which work with data objects.. This is done through an OSGi package import. It is arbitrarily decided, that all classes that implement data objects are encapsulated in `cz.cuni.mff.paria.datalayer` package and this package is exported by the original OSGi bundle. All other bundles, which require data classes on their classpath, declare that they import, and thus require, this package. Upon resolving bundles, OSGi framework then automatically bridges these bundles with appropriate bundle, which exports the data package. It is also required that Java classes that should be persisted must be annotated by EJB annotations, so that the persistence engine knows the relations among data and knows how to save them in a database.

The data layer package is also required to provide `BeanRegistry` class with a single static method named `getRegisteredClasses()`. This method is used to acquire list of all classes which compose the data layer. In our framework, it is used to initialize Hibernate object mappings to relational database based on EJB and/or Hibernate annotations, and to know which classes to replicate through synchronization mechanism.

3.2.3 Filesystem module

OSGi manifest fragment: FS bundle

```
Bundle-SymbolicName: cz.cuni.mff.paria.bundles.fs
Require-Bundle: cz.cuni.mff.paria.main;bundle-version="1.0.0"
```

Exported services: FS bundle

- `cz.cuni.mff.paria.services.FileSystem`

Module `cz.cuni.mff.paria.bundles.fs` is a simple module, which doesn't provide any extra functionality but is important as an example of how features may be configured using services to be present or forbidden. This module is intended to be present in every deployment of application, which should have access to underlying filesystem. Usually it targets desktop deployment, where the application runs under user's privileges and access to filesystem is eligible, so that user may directly manipulate with files on her computer. On the contrary, web part of application is usually deployed at public server and thus available for everyone, and access to server's filesystem is required to be prevented. This different configuration is conveniently represented by availability of the filesystem service. If all the filesystem dependent functions of the PARIA application are executed only after successfully acquiring reference to the FS service, then the filesystem is accessed only if a FS module (and thus the FS service) is configured to be available at application runtime.

Code snippet 1: Using FS service to access filesystem

```
if (ServiceRegistry.isAvailable(FileSystem.class) ) {
        /* do something dependent on filesystem,
         * e.g. show UI button to load files
         */
}
```

3.2.4 Persistence module

OSGi manifest fragment: hibernate persistence bundle

```
Bundle-SymbolicName:
   cz.cuni.mff.paria.bundles.hibernatePersistence
Require-Bundle: cz.cuni.mff.paria.main;bundle-version="1.0.0"
Import-Package: cz.cuni.mff.paria.datalayer
Export-Package:  cz.cuni.mff.paria.bundles.hibernatepersistence,
```

(and all packages from hibernate and hibernate-annotations class libraries, all packages from DB4o class libraries)

Exported services: hibernate persistence bundle

- `cz.cuni.mff.paria.services.Persistence`

Persistent data storage is accessed through an OSGi service represented by `Persistence` interface. We implemented the persistence module as an OSGi bundle called `cz.cuni.mff.paria.bundles.hibernatePersistence`. This bundle is built on Hibernate persistence libraries and stores the Java objects in an underlying SQL database.

Upon activating the bundle, it initializes and starts Hibernate engine with a default configuration and with any other provided configuration files if available. To initialize Hibernate session factory properly, the bundle activator scans through the list of classes obtained from datalayer's `BeanRegistry`, and adds them to Hibernate annotation configurator. The mappings to relational database are then created from EJB annotated classes. The process of initialization is concluded by inserting db4o configuration, so that db4o synchronization is available later.

Hibernate initialization is configured through a configuration file `hibernate.cfg.xml` included in the bundle as a bundle resource. To enable custom configuration of Hibernate in order to connect it to a certain SQL database, and without the need to amend the HibernatePersistence bundle itself, we make use of a very handy feature of OSGi called bundle fragments. This technique allows to modify the host bundle by attaching a fragment of the bundle to it. Fragments in OSGi are in principle of the same physical structure as usual bundles, however, they cannot be activated like normal bundles because they logically complement the host bundle, which has its own activator. Upon bundle resolution, OSGi framework attaches all related bundles to their host bundle, logically joining them into one resulting bundle by combining their manifests. Fragments are allowed to alter any part of bundle's manifest and any resources.

The code in Hibernate bundle initialization attempts to read custom configuration file `hibernate.dbcfg.xml`, which is not present in the bundle itself. This provides a hook for developers using our framework to insert their configuration file — the file `hibernate.dbcfg.xml` is simply put into the fragment's classpath and the fragment is attached to Hibernate core bundle. Configuration fragments may also include necessary JDBC drivers and class libraries.

3.2.5 Internal Derby database module

OSGi manifest fragment: Derby hibernate bundle fragment

```
Bundle-SymbolicName:
   cz.cuni.mff.paria.bundles.derbyHibernateFragment
Fragment-Host: cz.cuni.mff.paria.bundles.hibernatePersistence
```

Additional notes: hibernate persistence bundle

- adds `hibernate.dbcfg.xml` hibernate configuration file into host's classpath

This module represents a compact solution for local storage of desktop instances. It is designed to be included in a desktop instance to supply the persistence module in situations, where external SQL database is not applicable.

This module includes only hibernate configuration file specific to embedded derby database and the derby DB class library. It is configured to launch embedded derby database upon loading of JDBC driver from hibernate. The database is represented by a file on the disk under current directory. It is created automatically if the file is not present.

3.2.6 Hibernate synchronization module

OSGi manifest fragment: database synchronization bundle

```
Bundle-SymbolicName:
   cz.cuni.mff.paria.bundles.hibernateSynchronization
Require-Bundle: cz.cuni.mff.paria.main;bundle-version="1.0.0",
   cz.cuni.mff.paria.bundles.hibernatePersistence;bundle-
   version="1.0.0", cz.cuni.mff.paria.datalayer;bundle-
   version="1.0.0"
Export-Package: cz.cuni.mff.paria
                          .bundles.hibernatesynchronization
```

Imported services: database synchronization bundle

- `com.db4o.drs.ReplicationEventListener`
- `cz.cuni.mff.paria.bundles.hibernatesynchronization`
  `.ReplicationProgressListener`

Exported services: database synchronization bundle

- `cz.cuni.mff.paria.bundles.hibernatesynchronization`
  `.SynchronizerController`

Synchronization of data persisted by Hibernate module is implemented in a separate bundle `cz.cuni.mff.paria.bundles.hibernateSynchronization`, which should be configured to activate upon desktop client start-up. It then spawns a background thread and cares for data synchronization itself without any need for additional treatment.

In this work, we implement synchronization based on Db4o Replication System. This replication system was built primarily for the DB4o object database, but has been later extended to support also Hibernate object persistence. We take advantage only of its support for Hibernate to Hibernate data replication. The synchronization thread connects to two distinct database through Hibernate. One of the connections is taken from Hibernate persistence module configuration (thus the hibernatePersistence module is included as a dependency in the import list), and the other is read from a configurations file provided by a custom bundle fragment, copying the mechanism from hibernatePersistene bundle, only the files are prefixed by `sync.` string.

The synchronization process is composed of the following:

- Creation of SQL tables used to store additional meta-data about modified data
- Storing of additional meta-data about modified objects upon Hibernate commits
- Replicating object deletions
- Check for objects created or modified in remote central storage and their replication
- Check for objects created or modified in local storage and their propagation to the central storage
- Resolution of conflicts between central and local storages.

Creation of necessary SQL meta-data tables and storing additional meta-data is tightly coupled with the module providing Hibernate persistence. This is because if the persisted data don't include additional information about the history of their changes required by db4o replication system, it is later impossible to easily append it to the stored data. DB4o engine is hooked to the Hibernate engine by listeners, which react on data modification events and store information about object revisions into the same database. The synchronization mechanism later reads the information about changed objects and then transfers necessary data to make the two databases consistent.

The synchronization module itself periodically starts a synchronization thread, which attempts to make both databases consistent. The synchronization is performed in three phases. First, the deletions of objects, which were deleted on either side, are propagated to both databases, so that no needed objects are transferred later. If there are any conflicts in this phase, it is assumed that deletion has stronger preference than modification — if an object is deleted, this action is usually meant to be permanent, apart from a change in object properties, and thus it is generally safe to expect that the deletion in one of the storages happened before the modification in the other storage. This is also the default behavior of DB4o replication system. After, the synchronization process checks for new and modified objects. This check ids performed against the remote central database first. This is for two reasons. Central storage is considered the actual state of all distributed data and hence should not be modified before the local database is filled with the centrally stored data. This is to ensure that the local instance has access to the most fresh data available. Second reason is more pragmatic — some conflicts during replication phase may arise, and it is more convenient and safe to solve most of them before any data is uploaded to the central storage. Hence, it is allowed to modify other non-conflicting objects during conflict resolution and add these objects to the other locally modified data before they are synchronized with the remote storage.

When conflicts happen during replication process, the whole process is intercepted by a listener, which handles conflict resolution. In the default configuration, conflicts are resolved automatically by synchronization module using in-built DB4o replication mechanisms — the central storage is considered as the master, and thus whole object in the local storage is overwritten by the remote version. This conflict resolution strategy is very simple and may in some peripheral situations cause data losses. However, we also provide a way for developers to implement their own strategy. It is as simple as to implement the interface `com.db4o.drs.ReplicationEventListener` and register it as an OSGi service in any bundle activator. The synchronization thread prompts main module's `ServiceRegistry` for all the services implementing this interface and adds them as listeners to replication events. Using `ServiceRegistry` ensures that all bundles are started prior to searching for the services. Only if no such services are found, the default strategy is used. The behavior of the synchronization thread may be further fine-tuned through exported OSGi service called `SynchronizerController`.

Because synchronization itself may be a length background process, we provide a simple mechanism to monitor its progress. This mechanism is based on OSGi services. This time, the use of services is reversed on the basis of white-board model. The synchronization process doesn't register a monitoring service, on the contrary, available published services are used as progress listeners. The synchronization process, whenever it makes some progress, sends notifications to all present services implementing monitoring interface. Modules, which want to monitor the synchronization process, register service named `SynchronizationProgressMonitor`. Through this interfaces they then receive progress information about synchronization.

3.2.7 Servlet bridge

As discussed in 3.1.1, complete PARIA deployment consists of two separate builds, one of them is a Java web application, and the other is a standard Java desktop client distributed as a Java Web Start package. While configuration and execution of the desktop client is very close to launching a standard standalone Java desktop application, the web application has to be inserted in an external Java application container. Meanwhile, OSGi framework is generally implemented as a standalone runtime unaware of being started from within a servlet container, and this causes difficulties when attempting to access the underlying container. Additional work had to be done to bridge a servlet container and OSGi framework in a manner consistent with the OSGi principles. In order to accomplish this task, we need to amend the underlying framework environment. To avoid direct modification of a framework source code, which would also bring a dependency on a certain framework implementation, we utilized an advanced feature of OSGi framework specification — extensibility of the framework itself by an extension bundle. This bundle then extends the framework environment through the basic OSGi bundle and services mechanisms.

The Equinox OSGi implementation project provides a very convenient and easy to use solution to implement a servlet bridge through the use of the Http Service specification, which is part of the OSGi service compendium ([OSG07b]). The idea is basically as follows: the chosen OSGi framework is encapsulated into a servlet component and launched upon servlet initialization at the web application start-up. The launcher servlet automatically provides a framework extension bundle, which then registers a Http service. The implementation of this service has direct access to underlying servlet and its container. By registering this service, it makes the access to the servlet container available to other modules of the application, creating the final bridge between them and the underlying container. Framework modules, which need to export their servlets to a Http servlet container, thus have a transparent way to publish their servlets, regardless of weather the container is implemented as an internal server module, or as a bridge to the underlying servlet container. This solution then enables to encapsulate any part of PARIA application, which has to be built as a Java web application using servlets, into a standard OSGi module, which simply publishes exported servlets through the OSGi Http service interface.

3.2.8 Internal HTTP server module

OSGi manifest fragment: internal jetty server bundle

```
Bundle-SymbolicName: cz.cuni.mff.paria.bundles.internalHttpServer
Import-Package: org.eclipse.equinox.http.jetty;version="1.1.0"
Require-Bundle: cz.cuni.mff.paria.main;bundle-version="1.0.0"
```

Exported services: database synchronization bundle

- `org.osgi.service.http.HttpService`

If we compare the differences between the web application and the desktop client environment, the manner, in which the user interfaces to applications are traditionally served and implemented in each of them, is cardinal. It causes a lot of code duplicity if two separate user interfaces have to be implemented in each case. In order to avoid the need to duplicate work on user interface, we integrated an internal http server module, which provides a transition for the interface developed for web environment, so that it could be deployed also in the desktop deployment using little extra effort.

The internal http module is built over embedded Jetty Java http servlet container and exposes itself through the same OSGi Http service as the servlet bridge mentioned above. It is meant to be included in the desktop deployment to substitute for the servlet bridge, which is not available outside of web deployment.

The module imports `org.eclipse.equinox.http.jetty` package, which provides for means to configure and launch the embedded http server itself. This package can be found in a bundle with the same name, which is included in the Eclipse PDE platform. Otherwise it can be easily built, because it is used only for launching the server through a single method, which has following signature:

- `org.eclipse.equinox.http.jetty.JettyConfigurator.startServer(java.lang.String, java.util.Dictionary)`

Custom configuration to the jetty container can be supplied again through a bundle fragment, which contains file `http.properties` within the classpath. This file is a simple properties file, which is read as `java.util.Properties` instance and put as the second parameter to the call of `startServer` method.

3.3 Development tool based on Eclipse platform

3.3.1 Description of Eclipse PDE platform

The Eclipse platform was chosen as a base for development tool for building PARIA applications because it is a mature and well-known development tool for Java and web applications and provides features and tools very useful in building such development tool. The Eclipse IDE is easily extensible by additional third party plugins, provides well designed tools to build such plugins. Moreover, this extensibility is based on the OSGi specification, and the tool itself can be easily turned into an OSGi development platform.

The Eclipse IDE is built around a modular architecture, due to which the development environment is extensible by many various plugins and tools. In latest releases (since the beginning of the third generation) of the product, the modular architecture is implemented as a full OSGi framework according the specification, which is called Equinox. This framework is further extended by additional custom concepts to provide for the development of richer tools and applications for the Eclipse platform. However, these features are implemented transparently through standard OSGi mechanisms and don't break the specification. It is easy to amend the default configuration to turn the Eclipse plug-in framework into a standard OSGi framework. The Eclipse Plug-in Development Environment provides tools for development of various types of extensions to the Eclipse IDE itself. Most of these extensions are based on OSGi components, such as bundles, and the Eclipse PDE platform allows even to restrict development of plugins to OSGi-only features. Thus, the tooling allows to develop plain OSGi bundles as restricted Eclipse plugins, and OSGi bundle fragments as restricted Eclipse fragments. It also contains some helpful additional tools and concepts: launch configuration for an OSGi application, running and debugging such application, grouping OSGi bundles into features, building and exporting bundles and features, and lot more.

3.3.2 OSGi module configuration and development

In our framework, the only OSGi elements used are OSGi services, bundles and bundle fragments. The OSGi services are easily created and administered by Java source code in bundles and the PARIA framework provides some utility classes and tools to make manipulation with services even easier. On the other hand, configuration of bundles and bundle fragments is done through OSGi manifests and is much more complicated.

The Eclipse PDE provides for very useful visual tools to configure and develop OSGi bundles and fragments. These tools are capable to edit almost all the needed features of OSGi manifests in visual way, so that the developer is not required to know the proper syntax of manifest headers. Worthy of mentioning are

- defining of bundle activator, ID and version number
- listing the dependencies on other required or optional bundles or imported packages
- listing of packages that the bundle exports
- modifying bundle's classpath
- choosing the host bundle for bundle fragment

Every bundle and bundle fragment is developed as a separate Eclipse PDE project, bundle as a Plug-in project and bundle fragment as a Fragment project. Any bundle project in workspace may be listed as a prerequisite of another bundle project through OSGi dependencies, or an exported package may be imported by another project. The dependent projects are then automatically interconnected according the dependency tree, resolving thus Java imports from other bundles.

All the PDE projects must be created with a standard OSGi framework, apart from using Eclipse framework, which is selected by default. The project creation wizard allows for easy change of this setting, and also to set up some basic configuration details of the bundle manifest.

3.3.3 Running and debugging applications within the IDE

Before we can run an application composed of OSGi bundles, we have to configure which bundles and how they are to be launched. This could be a tough thing to do manually, because usually the final application consists of greater number of bundles. Eclipse IDE provides us with a helpful tool to configure the bundles. This is done by setting up a launch configuration for an OSGi application. The launch configuration dialog enables to choose which modules should be included in the launched application and helps with automatic dependency resolution. This launch configuration can be used to launch applications in run or debug mode. It is also helpful for other tasks later, for example to setup a feature project.

3.3.4 Building the final application

We deploy the final PARIA application in two separate packages. The web package has a form of a Java web application and whole PARIA is in fact embedded into a bridging servlet. On the other hand, the desktop package has to be built as Java Web Start application, with the necessity to pack all program components into signed downloadable packages accompanied with JNLP manifest files. The structure and contents of final packages are so different, that we have to provide two distinct mechanisms to build them.

Whole PARIA application consists of separated modules and each different deployment is built including a subset of these modules..Some of the modules are common for both web and desktop deployments, the others are targeted to be used in only one environment. Modules which apply to data layer, persistence and provide common functionality, are those, which could be common. It is a good practice to bind such common modules together into a common feature project and include this feature into all features exported From the modules available in our framework, the following may be common to both deployments:

- `cz.cuni.mff.paria.bundles.hibernatePersistence`
- `cz.cuni.mff.paria.datalayer`
- `cz.cuni.mff.paria.main`

Modules applicable to only the web environment are not very numerous and account only to support for data synchronization initiated by client, and to bridging the application with underlying servlet container. The servlet bridge, provided by the equinox.servletbridge project, serves as a bundle, which publishes underlying servlet container as an OSGi http service. However, it is not deployed in a standard way like all other modules are and the use of this bridge will be explained later in section describing building of web package. There is thus only one web-only module, and it is even optional (`cz.cuni.mff.paria.bundles.hibernateHttpBridge`). It provides access to the central database transparently through the http protocol, if the standard database connection protocol between desktop instances and web instance is not applicable.

Most non-common modules target only desktop environment. They are either not applicable to the web environment or a different implementation has to be used to account for the differences. These include local filesystem switch module to allow access to local files, synchronization module to turn on automatic data synchronization with central instance, and other utility modules. In our framework, following modules are to be used mainly in the Java Web Start environment:

- `cz.cuni.mff.paria.bundles.hibernateSynchronization`
- `cz.cuni.mff.paria.bundles.fs`
- `cz.cuni.mff.paria.bundles.derbyHibernateFragment`
  - embedded derby SQL database integrated into Hibernate persistence module as a configuration bundle fragment, database files are automatically created in Web Start cache
- `cz.cuni.mff.paria.bundles.internalHttpServer`
  - embedded jetty http server to help with using the same web user interface on a local machine as is used in web deployment
- `cz.cuni.mff.paria.jnlp`
  - utility module providing access to features available only in Java Web Start environment, such as running JNLP instances as singletons

Custom parts of the application are built and later integrated into the framework as custom OSGi bundles using all the advantages of OSGi framework.

3.3.5 Building web application package

The web part of an application is built and deployed as a standard web application. This web application is initiated by a servlet, which launches and bridges OSGi framework from within context of servlet container. Hence, whole framework with application modules has to be included in the built web application package and made accessible from bridging servlet.

The build process requires creation of a feature project, which includes all the necessary modules and a necessary servlet bridge feature. The build itself is performed using `webappBuilder.xml` Ant build script. This build script takes advantage of Ant tasks exported by Eclipse PDE platform and by our customized Web Start Eclipse plugin to automatically export the web package feature and equinox framework configuration. All the exported files and packages are bundled into the structure of final web application, under WEB-INF folder, together with web.xml manifest and a the servlet bridge library.

All the necessary build tools and files are held in a special Eclipse project, which is based on `org.eclipse.equinox.servletbridge` Eclipse project provided by equinox.servletbridge team. This project includes class library `servletbridge.jar` containing servlet, which is used to launch and extend the equinox framework, the Ant script for building, and template directory structure, which is to be customized and which is copied into the build directory of the web application by the building script..In order to configure the building process, developers need to adjust the Ant build script to match their setup, and edit the web.xml application descriptor in the template directory. In Ant build file, two main Ant targets have to be configured. First one, named `pdeExportFeatures`, uses `pde.exportFeatures` task to export all the bundles of the target web package. It is only needed to adjust the name of the feature, which is to be exported. The exported feature is created as an Eclipse feature project, and has to include another feature with all the necessary OSGi modules needed to bridge OSGi framework with the underlying servlet container in order to provide an OSGi http service. This feature is called `org.eclipse.equinox.servletbridge.feature` and contains `http.registry, http.servlet, http.servletbridge` Equinox OSGi modules together with all the dependencies. The second target is used to help create equinox framework configuration file automatically from an OSGi launch configuration and uses `paria.equinox.exportConfigurationFile` Ant task, which was included into the Web Start plug-in to support the build mechanism in our framework. It is only needed to prepare a launch configuration with a unique name and paste its name into the build file. Both mentioned Ant targets are configured through named Ant properties, which are set in Eclipse Ant launch configuration dialog. These are:

- **`paria.webfeatures`** — name of the web package feature (features)
- **`paria.weblaunchconfiguration`** — name of the template OSGi launch configuration

After execution of the properly configured build file, the web package is built and is prepared to be deployed to a servlet container.

3.3.6 Building desktop application package

The desktop build uses bundle packages called features, a concept introduced by Eclipse PDE, to bind all the exported plugins together and export them in one step. Exported plugins and features are signed with a keystore, which is either supplied by developer or a default keystore included in our development tool is used.

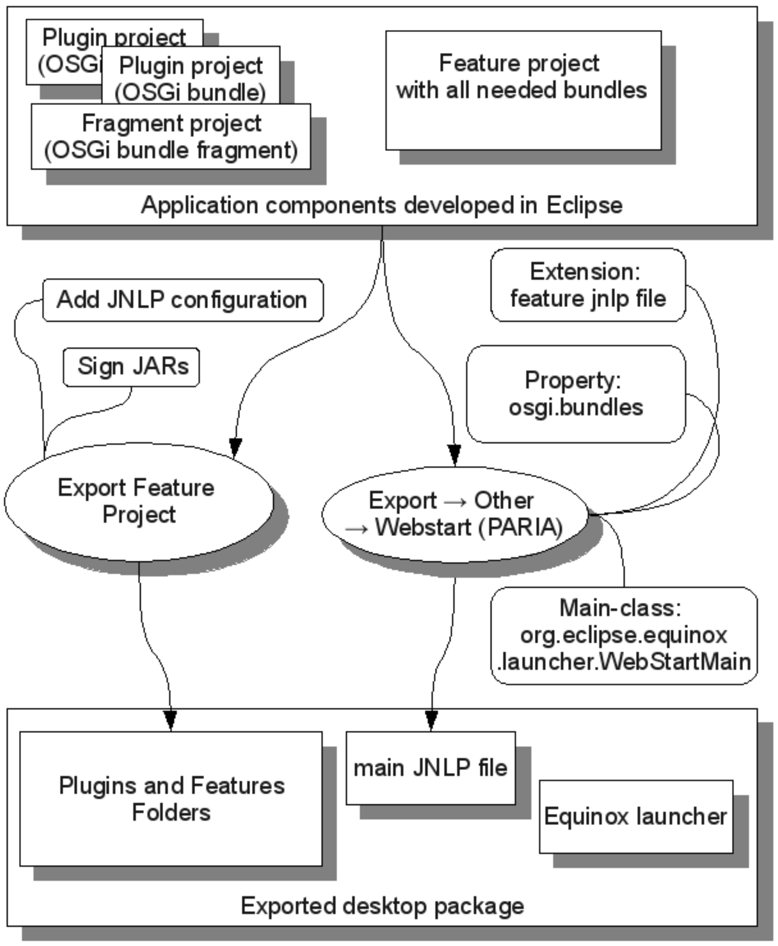
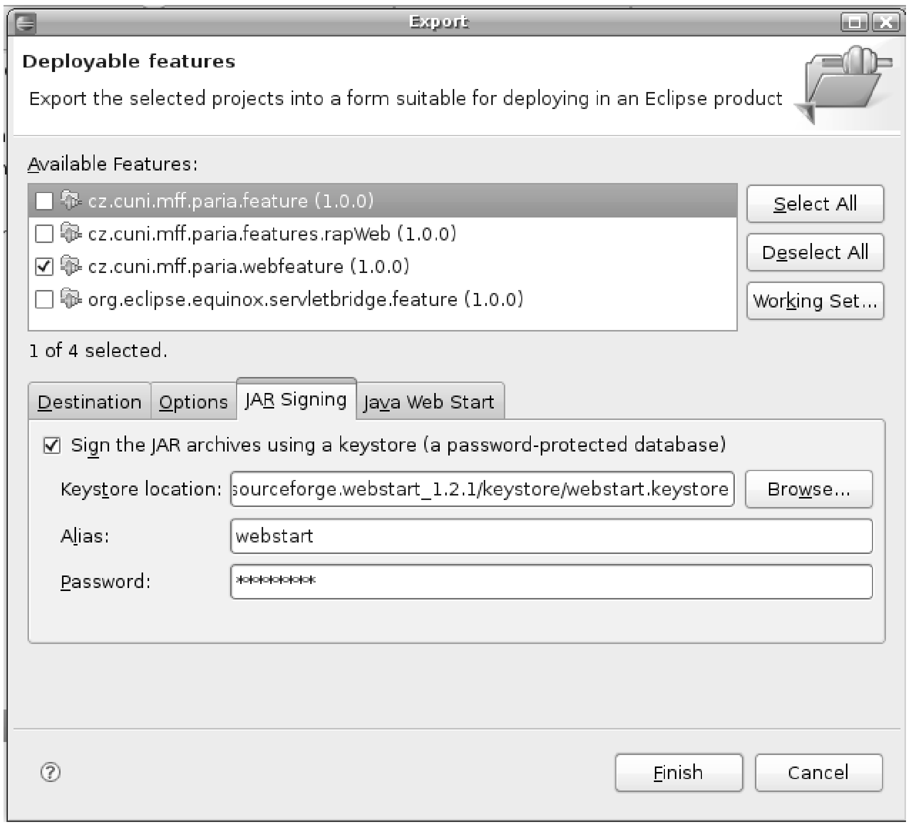

Plugin project (OSGi bundle)
Plugin project (OSGi bundle)
Fragment project (OSGi bundle fragment)
Feature project with all needed bundles
Application components developed in Eclipse
Add JNLP configuration
Sign JARs
Extension: feature jnlp file
Property: osgi.bundles
Export Feature Project
Export → Other → Webstart (PARIA)
Main-class: org.eclipse.equinox .launcher.WebStartMain
Plugins and Features Folders
main JNLP file
Equinox launcher
Exported desktop package

Figure 4:Scheme of building process of desktop application instance. The final product is a deployable Web Start application with JNLP manifest file

It is necessary to sign all the packages that are to be deployed through Java Web Start, because only then it is possible that the JNLP application is granted all the privileges a desktop application can have. After bundles are exported as a signed JNLP package, the export is completed by adding a simple Equinox launcher, which is used to launch whole OSGi framework runtime. This is a small JAR class library, which is included in our framework for convenience. Real equinox OSGi implementation is present in Eclipse PDE as a bundle and was automatically exported with the previous feature export. Finally, main JNLP descriptor is created using a modified Web Start Eclipse plug-in, originally developed by Jack Li Guojie (*http://sourceforge.net/projects/webstart*).


Export

Deployable features

Export the selected projects into a form suitable for deploying in an Eclipse product

Available Features:

- [ ] cz.cuni.mff.paria.feature (1.0.0)
- [ ] cz.cuni.mff.paria.features.rapWeb (1.0.0)
- [x] cz.cuni.mff.paria.webfeature (1.0.0)
- [ ] org.eclipse.equinox.servletbridge.feature (1.0.0)

1 of 4 selected.

Destination | Options | JAR Signing | Java Web Start

- [x] Sign the JAR archives using a keystore (a password-protected database)

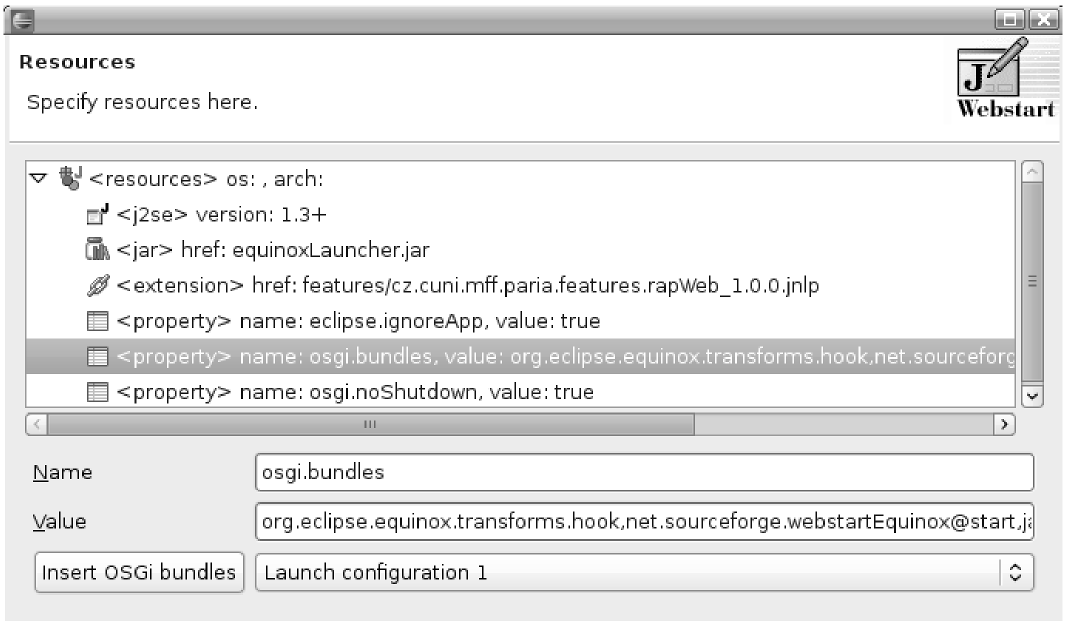
Keystore location: sourceforge.webstart_1.2.1/keystore/webstart.keystore

Alias: webstart

Password: *********

Figure 5: Wizard for configuring export of feature. All components have to be packaged as individual JAR archives and signed using a keystore.

In order to export signed bundles, we first create a feature project representing the configuration of modules present in final desktop build. This feature should consist of `org.eclipse.osgi` plug-in and any other custom plugins (bundles) we need to include in our build. It may be easily filled with appropriate plugins from an OSGi launch configuration. After the feature project is configured, we use a standard feature export dialog. It is important how export is configured through this dialog. The option to package as individual JAR files have to be selected, so that the application can be later deployed through Java Web Start. Further, it is also as important to select option to sign JAR archives. The Web Start for Eclipse plug-in provides a legacy keystore under plugins/net.sourceforge.webstart_1.2.1/ keystore/webstart.keystore path with both alias and password being "webstart". The Java SDK provides tools to create and sign keystores and developers are encouraged to use their custom keystores.



Resources

Specify resources here.

```
<resources> os: , arch:
    <j2se> version: 1.3+
    <jar> href: equinoxLauncher.jar
    <extension> href: features/cz.cuni.mff.paria.features.rapWeb_1.0.0.jnlp
    <property> name: eclipse.ignoreApp, value: true
    <property> name: osgi.bundles, value: org.eclipse.equinox.transforms.hook,net.sourceforg
    <property> name: osgi.noShutdown, value: true
```

Name: osgi.bundles

Value: org.eclipse.equinox.transforms.hook,net.sourceforge.webstartEquinox@start,j

Figure 6:Resources page of JNLP file export wizard. Resources are preconfigured for deployment under Equinox OSGi implementation. Bundle configuration is easily generated from OSGi launch configurations.

Exported OSGi modules are turned into complete Web Start application through a JNLP file export. This is accomplished through a standard Eclipse export dialog under options Other → Webstart (PARIA). This dialog is preconfigured to contain all the necessary configuration to create a JNLP file. It is only needed to fill custom values and do two necessary amendments: in resources dialog page, the link to the JNLP file of the feature exported previously has to be corrected, and `osgi.bundles` property has to be added. This property configures Equinox runtime to load and launch the necessary plugins, which the application consists of. The dialog provides option to fill in the bundle configuration from an OSGi launcher configurations, if any were made ahead. After exporting the main JNLP file and all the necessary preceding steps, the desktop package build is complete.

Example snippet of an exported JNLP file

```
<jnlp>
  <information>
    <vendor>Ondrej Mihályi</vendor>
    <offline-allowed/>
    <shortcut online="false">
      <desktop/>
      <menu submenu="PARIA applications"/>
    </shortcut>
  </information>
  <security>
    <all-permissions/>
    </security>
    <resources>
      <jar download="eager" href="equinoxLauncher.jar" main="true"/>
      <extension href="features/cz.cuni.....rapWeb_1.0.0.jnlp"/>
      <property name="osgi.bundles" value="..[bundle configuration].."/>
      <property name="osgi.noShutdown" value="true"/>
      <property name="eclipse.ignoreApp" value="true"/>
    </resources>
  <application-desc
   main-class="org.eclipse.equinox.launcher.WebStartMain"/>
</jnlp>
```

3.3.7 Installation as a desktop package from within a web application

After building the desktop package and the web package, these two have to be interconnected into a complete PARIA package. The desktop package is inserted into the web application folder of the web package and the web.xml application descriptor is configured to map a URL to the main JNLP descriptor of inserted desktop package.

In order to interconnect the two packages on the application level, so that the web part of application knows, where the desktop package is deployed, we designed a simple solution that fits into the OSGi architecture. The solution is to utilize already present OSGi services mechanism and supply a service to discover presence of deployed desktop package and its URL. For this to work, servletbridge servlet is accessed from the main module of our framework and a servlet context parameter called `paria.desktopPackage.jnlpURL` is read for the link to JNLP file. Then an OSGi service is published, which provides this link to other application modules. This link is meant to be used by web user interface module, which then may provide methods to launch the Web Start instance directly from web instance.

3.3.8 Tooling for development of user interfaces

From the perspective of user interfaces, regular web and desktop applications differ noticeably in the platform, which is the base for the user interface. Desktop applications run as separate programs using available graphical desktop resources and windowing toolkits, and they serve the user interface as one or couple of decorated windows. On the contrary, standard web applications are served through and tied to a browser environment, what poses limits on how the user interface is developed. In PARIAs, this concludes into the necessity to either supporting multiple different user interface modules, or to find a solution as to how to bring the user interface, which is designed to be run in browser, to the desktop platform.

One of the approaches to unify user interfaces is to build the interface on a common platform made available in both environments. This is implemented as a common runtime, which can be made available in browsers through a plugin and as a library or standalone executable on desktop. Some of the tools discussed in provide such platform, for example Adobe AIR or Java FX. In the thesis we target PARIAs, which are not explicitly dependent on any non-standard browser plug-in and can be accessible through any bare internet browser. Thus this approach is not considered as an adequate solution. However, it still may be used in case it is appropriate to require additional browser plugins.

We will further discuss approaches, which avoid such requirements. First, the web user interface can be built using an API, which is supported by two different backends, each one for a different environment. This is not widely supported by many user interface toolings, nevertheless the combination of Rich Client Platform (RCP) and Rich Ajax Platform (RAP) integrated into Eclipse IDE provides tools to support multiple backends to a unified API. The two different backends are included in the final application as OSGi bundles, and in the build configuration, the appropriate backend bundles are exported for web and desktop deployments. Since standard RCP bundles are based on platform dependent SWT toolkit, it is encouraged to substitute standard SWT bundles with a bundle built on SWTSwing libraries, which create a bridge between SWT API and platform agnostic Swing UI libraries. This is exemplified by `cz.cuni.mff.paria.lib.swingswt` module included in our framework.

Another adequate approach is to bring a servlet container into the desktop environment and serve user interface again through a standard web browser. This solution is the most powerful and supports any web technology used in development of user interfaces. On the other hand, it may appear less intuitive to users and involves the necessity to launch a browser to provide the interface through. Our framework supports this approach from the ground by providing a module with embedded jetty servlet container named `internalHttpServer`, which publishes a standard OSGi http service. This module is to be included in the desktop instance to provide the same http service, which is regularly available in the web instance.

Chapter 4: Realization of a prototype application

4.1 Prototype application

To illustrate the capabilities and features of our PARIA framework and tools, we will describe a prototype application. This application is a simple program for taking and managing textual notes. It runs on the internet as a web application accessible through a standard web browser. It can be downloaded and installed to the desktop as a JNLP package. The desktop package then runs either as an normal desktop program or in a browser window as web application through an embedded http server. The data are automatically synchronized with the Internet.

4.1.1 Architecture of prototype application

The core of the prototype application is based on Eclipse Rich Client Platform technology (RCP). This platform is generally used in development of standard desktop applications based Eclipse technologies. Another Eclipse technology, Rich AJAX Platform (RAP) enables to build web applications based on UI API analogous to RCP API. We have chosen these two technologies to build the prototype application to illustrate the possibilities of adaptability and extensibility of PARIA applications. While the web application instance provides standard web user interface through a browser, desktop instance may optionally provide a more convenient native user interface. This is achieved by different configurations of OSGi bundles.

Web instance of application is built as servlet deployed through the OSGi HTTP service into the underlying servlet container. This service is provided by the servlet bridge included in PARIA framework. The rest of the application is based on Hibernate module, Data layer module and the core RCP application executed in RAP environment. Hibernate module is connecting to a distinct SQL server on the network.

Desktop instances are distributed from the web instance through the JNLP protocol, which is linked to from a menu button. They are optionally served through a native windowing interface in RCP environment, or through a browser using embedded instance of jetty http server. Hibernate persists objects in embedded derby SQL database.

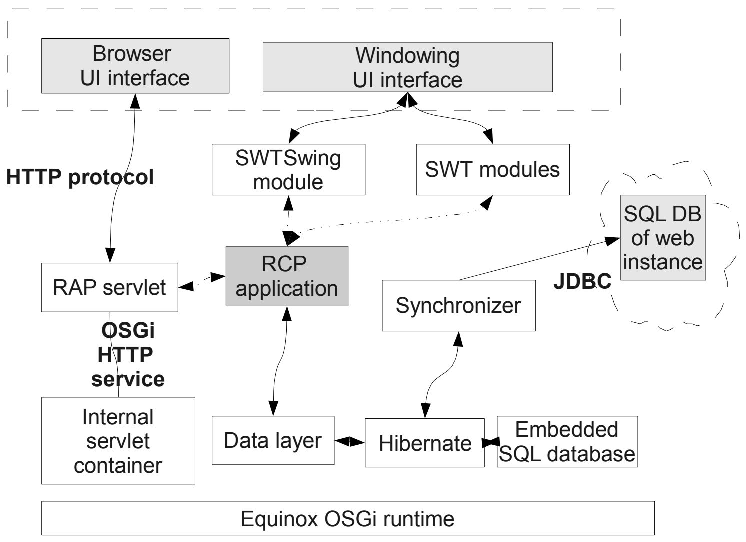


Figure 7: Architecture of web instance of prototype

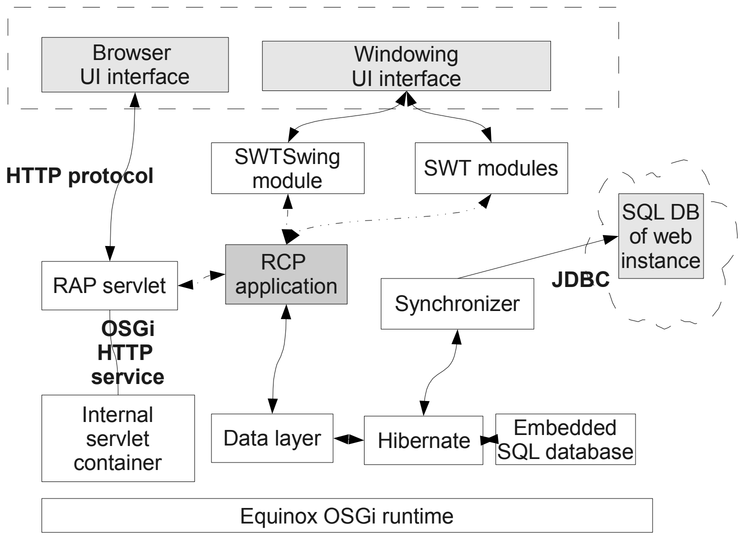


Figure 8: Architecture of desktop instance of prototype

4.1.2 Data layer

Data layer of the application is composed of three Java Bean classes and `java.util.List interface`, which is used to create one-to-many relations between them. The application manipulates with textual notes stored in folders. One folder stores zero or more notes, a note belongs to exactly one folder. We use an additional bean to track all the folders. This bean is used as the root of the data hierarchy. Following table summarizes bean classes, including EJB annotations.

Class description: RootBean

```
@Entity public class RootBean {
   @OneToMany List<FolderBean> folders;

   @Id @GeneratedValue public long getId();
}
```

Class description: FolderBean

```
@Entity public class FolderBean {
   @OneToMany List<NoteBean> notes;

   @Id @GeneratedValue public long getId();

   public String getName();
}
```

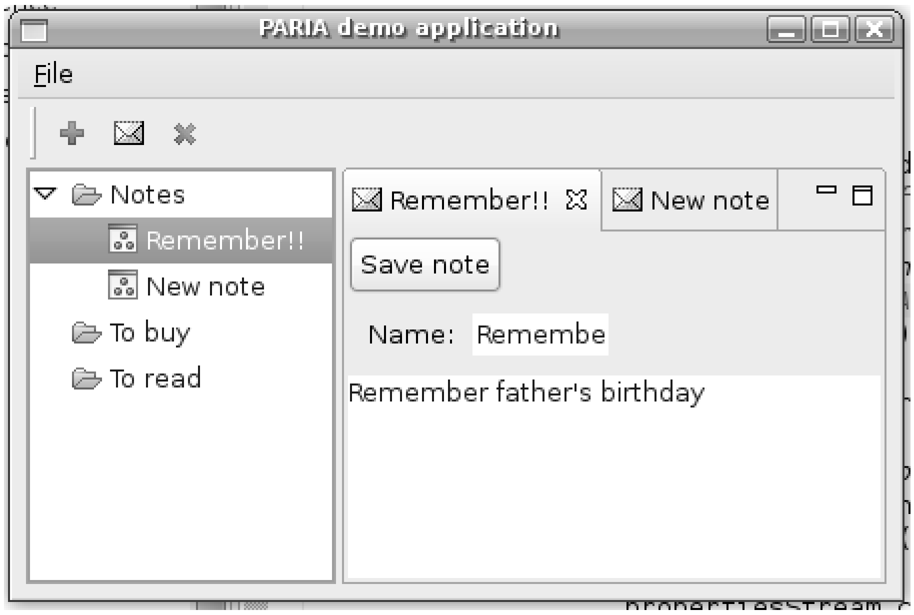
Class description: NoteBean

```
@Entity public class NoteBean {
   @OneToMany List<NoteBean> notes;

   @Id @GeneratedValue public long getId();

   public String getName();
   public String getText();
}
```

4.1.3 User interfaces of prototype application

Prototype application has been designed to show the ability to serve different user interfaces in web and desktop environment. This is in general accomplished by developing two different user interface modules for each deployment. However, we took advantage of Eclipse RCP and RAP technologies to simplify the development.

The user interface of application is built on packages, which are provided by both SWT bundles and RAP bundles. Exported Java packages in these two bundle packages are not equal and thus the final application code had to be adapted to support both of them. We use OSGi mechanisms to configure, which bundle packages are inserted in the environment. Underlying UI libraries are packaged into bundles, which export the necessary Java packages. This allows for substituting one UI engine for another. Additionally, we use the SWTSwing library and package it into an OSGi bundle. This bundle allows to bridge SWT API with native Java Swing UI libraries. SWT binary libraries are not platform independent and have to be supplied with native OS libraries. The SWTSwing module helps us to free the final application from this OS specific dependencies. The final result is that the same application code runs through a servlet and browser in web instance, and can run as a windowing application on desktop. This is illustrated by following screenshots of in different configurations.



Remember father's birthday

Figure 9:Screenshot of desktop instance running on SWT backend.

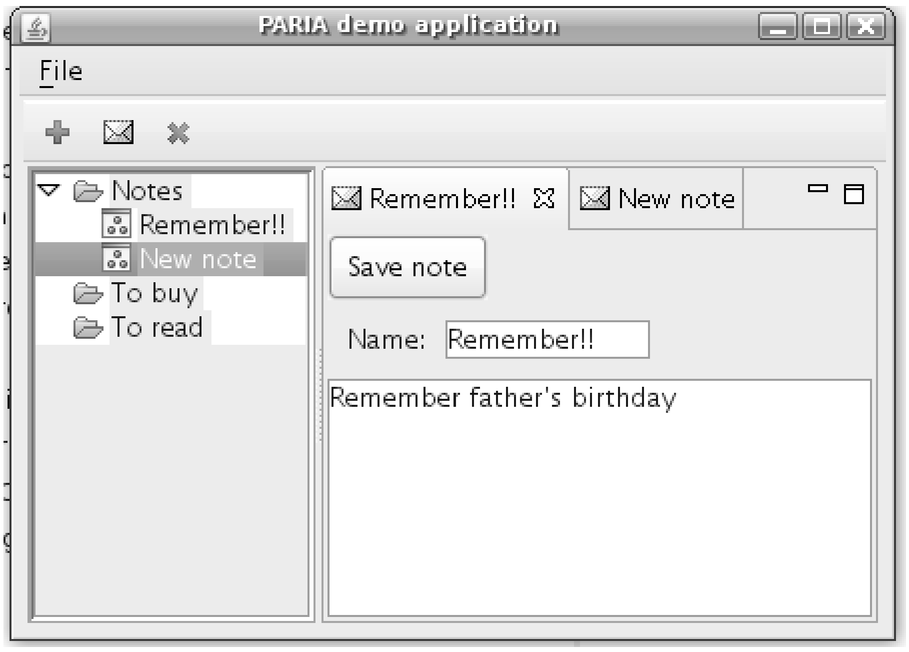


Figure 10:Screenshot of desktop instance running on pure Java SWTSwing backend.

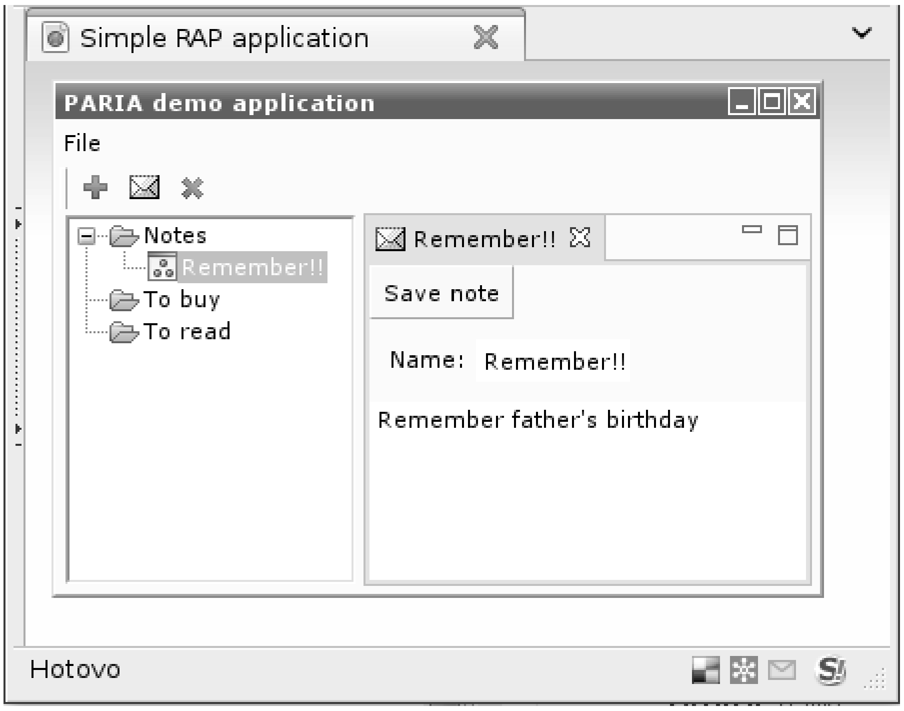


Figure 11:Screenshot of desktop or web instance running on servlet RAP backend.

4.2 Evaluation of PARIA framework in the context of prototype application

The prototype application showed that our PARIA framework provides a usable concept and tools to develop desktop-oriented web applications. The development tools simplify the development of PARIAs to the extent that it does not involve much more complexity than development of usual Java web applications.

A worthy aspect of building the application on our framework is that it involves many clear design patterns and principles. Separation of data into beans helped to build the application around a central object model. This allowed easy integration of model-view-controller design pattern into the application through a simple event registry. Thanks to the OSGi layer, it is now possible to build application on multiple modules and also extend it in future development. The `ServiceRegistry` in main framework bundle also helped to keep track of module references. The positive side-effect of this is cleaner and more readable code when accessing features available in distant parts of application. Utilization of OSGi framework also introduces features that are unmatched by other similar tools. It enables to build application modules, which are able to fit into the rest of the application and cooperate with other modules. This mechanism is valuable even outside of our framework in building custom application modules. The ability to adapt application user interface to target environment demonstrates its power. To develop this behavior using other technologies, such as Google Gears, is much more complicated, if not impossible.

Over Google Gears and Adobe AIR, our framework has the advantage that it is built on plain Java platform. This makes the final application more available, because Java Runtime Environment is nowadays present on virtually every piece of a computer device. Moreover, the framework provides an infrastructure and a legacy module for automatic data synchronization, which helps with building a shared data layer very simple. This is not provided by any other known tool to that extent. Google Gears provides a local SQL database with simple connection API, however, synchronization with remote server is supported rather by form of guidelines and some utility tools. Java FX provides again only a partial solution for building PARIAs. It highly depends on JRE and thus is suitable only for desktop instance development. As a result, it could be used as a technology complementing the PARIA framework, and not competing it.

Development of the application showed also some imperfections and stability issues in data synchronization mechanism. This was expected since the framework is not yet a mature project for production use. It cannot compete in stability and reliability with well established projects like Google Gears, Java FX or Adobe AIR, all backed by big commercial companies. Although to this day the framework as a whole is not sufficiently stable to build production-ready applications, it still has significant advantages over other tools available in the universe. Moreover, developers are free to supply their solutions into the framework through OSGi mechanism and thus bypass unfavorable, possibly unstable, legacy modules.

Chapter 5: Conclusion and future work

5.1 Conclusion

We built our tools by bringing together various Java opensource technologies. The additional effort to glue them together to an integrated framework was not comparable to the effort to build any of its major components. This proves that the Java platform and surrounding technologies create a solid ground for modern application development. Many advanced specifications and guides are being prepared by both commercial and noncommercial organizations, mainly Sun Microsystems and IBM. Thanks to them, it is possible to integrate most mature software components together easily. Moreover, the OSGi mechanism allows for flexible and configurable integration of the components.

The proposed method and tools for development of desktop-oriented web applications illustrate how next generation network applications may be developed. Using already existing and mature Java tools and applications, it was possible to create platform for applications, which are ubiquitously accessible and can compete with desktop applications at the same time. These applications combine the best from web and desktop world.

As modern web technologies develop, there will be more demand for applications, which provide access to the same user's data through any computer device connected to Internet. The Google company supports this direction with the many web applications accessible through any internet browser. Our solution extends the range of web applications over the limitations of web environment. It gives method and tools to develop web applications that can extend themselves to become more a desktop application than a web application. Final applications have the potential to unify the desktop and web paradigm into only one type of application necessary in the interconnected future.

5.2 Future work

Our framework was built as a proof of concept to support the development method of web-applications adaptable to desktop environments. As such, it leaves much space for improvement through development of additional OSGi modules or substituting existing components with more advanced ones.

Applications built with our existing framework are already able to provide adaptable user interfaces to access user data anywhere on the Internet. This raises security concerns, which are yet not covered. Data available on the Internet should be protected by some form of authentication layer running on the web instance. This layer would be supported by user interfaces, which would provide means to input authentication data, and by synchronization process, which would pass these data to the server to authenticate transactions.

Another aspect, which could be much improved, is the data synchronization mechanism. Our framework contains a simple data replication through DB4o Replication System. This process of synchronization periodically checks for changed data and replicates them. It could be much better optimized, for example by triggering synchronization attempt after data changes, or by compressing the transferred data. Also more complex solutions to treating data conflicts could be elaborated and supplied by more intelligent automatic conflict handlers.

In the future, we expect that more and more modern web technologies and solutions will develop and begin to be widely accepted. Lots of projects presented by Google and other major companies prove that this trend is inevitable. One of recent Google's moves — Google App Engine — may provide a very useful platform for deploying Java web applications in the future. It is meant to provide sandbox environment built on distributed resources. If appropriately adapted for deployment in this environment, it would represent a ideal deployment platform for ubiquitous web applications.

In the world of new technologies, we believe that our framework has full potential to evolve and not become obsolete. Many emerging technologies don't compete with, but may be combined with our framework. For example, Java FX may be used to build desktop instances, Google Web Toolkit can be a base for web user interfaces. The key concept of our framework is to provide a simple synchronization of shared data and this will not be provided by most of available tools for a while.

References

[All02] Jeremy Allaire (2002): Macromedia Flash MX – A next-generation rich client. Macromedia, Inc.
http://www.adobe.com/devnet/flash/whitepapers/richclient.pdf

[AGX02] Larry Arnstein, Robert Grimm, Chia-Tang Hung, Jong Hee Kang, Anthony LaMarca, Gary Look, Stefan B. Sigurdson, Jing Su, and Gaetano Borriello (2002): Systems support for Ubiquitous Computing: A case study of two implementations of Labscape. Pervasive Computing, Pages 30-44.

[Drs09] DB4o object database project: db4o Replication System (dRS) version 7.0 – Product Information.
http://www.db4o.com/about/productinformation/drs/.
Accessed: February 2009

[UWas02] Robert Grimm, Janet Davis, Ben Hendrickson, Eric Lemar, Adam MacBeth, Steven Swanson, Tom Anderson, Brian Bershad, Gaetano Borriello, Steven Gribble, David Wetherall (2002): Systems Directions for Pervasive Computing. University of Washington, USA.

[GDLX04] Robert Grimm, Janet Davis, Eric Lemar, Adam Macbeth, Steven Swanson, Thomas Anderson, Brian Bershad, Gaetano Borriello, Steven Gribble, and David Wetherall (2004): System Support for Pervasive Applications. ACM Transactions on Computer Systems, Vol. 22, No. 4, Pages 421-486.

[HC04] Richard S. Hall and Humberto Cervantes (2004): An OSGi implementation and experience report. Consumer Communications and Networking Conference, 2004. CCNC 2004. First IEEE, Pages 394- 399.

[Kim01] Steven Kim (2001): Java Web Start - Developing and distributing Java applications for the client side. IBM developerWorks.
http://www.ibm.com/developerworks/java/library/j-webstart/

[Mor08] Florian Moritz (2008): Rich Internet Applications (RIA): A convergence of user interface paradigms of web and desktop - exemplified by JavaFX. University of Applied Science Kaiserslautern,Germany. *http://www.flomedia.de/diploma/*

[OSG07a] The OSGi Alliance (2007): OSGi Service Platform - Core Specification.
http://www.osgi.org/Download/Release4V41

[OSG07b]	The OSGi Alliance (2007): OSGi Service Platform - Service Compendium. *http://www.osgi.org/Download/Release4V41*
[Sun97]	Sun Microsystems, Inc (1997): JavaBeans™ API specification. *http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html*
[Zuk01]	John Zukowski (2001): Java Collections Framework. IBM developerWorks. *http://www.ibm.com/developerworks/edu/j-dw-javacoll-i.html*
[Sun07]	Sun Microsystems, Inc (2007): Java™ Servlet Specification. *http://jcp.org/aboutJava/communityprocess/mrel/jsr154/index2.html*

Appendix

List of figures

Figure 1: RIA categorization (from [Mor08])....10
Figure 2: Topology of local and remote storages – one master storage and many local storages. Desktop instance accesses both master and local storages....21
Figure 3: Schema of framework with all the key parts. Shows OSGi component layers in horizontal axis, and data layers in vertical axis....26
Figure 4:Scheme of building process of desktop application instance. The final product is a deployable Web Start application with JNLP manifest file....43
Figure 5:Wizard for configuring export of feature. All components have to be packaged as individual JAR archives and signed using a keystore....44
Figure 6:Resources page of JNLP file export wizard. Resources are preconfigured for deployment under Equinox OSGi implementation. Bundle configuration is easily generated from OSGi launch configurations....45
Figure 7: Architecture of web instance of prototype....50
Figure 8: Architecture of desktop instance of prototype....50
Figure 9:Screenshot of desktop instance running on SWT backend....52
Figure 10:Screenshot of desktop instance running on pure Java SWTSwing backend....53
Figure 11:Screenshot of desktop or web instance running on servlet RAP backend....53

Acronyms and terms

Adobe AIR	*Runtime, which lets developers use web technologies to build rich Internet applications that run outside the browser.*
AJAX	*Asynchronous JavaScript and XML*
Ant	*Java build tool. (http://ant.apache.org/)*
CORBA	*Common Object Request Broker Architecture*
DB4o	*Java opensource object database*
DOJO	*AJAX Javascript toolkit (http://www.dojotoolkit.org/)*
DOM	*Document Object Model (http://www.w3.org/DOM)*
Eclipse PDE	*Eclipse Plug-in Development Environment. A platform used to develop Eclipse plug-ins from within the Eclipse IDE. (http://www.eclipse.org/pde/)*
EJB	*Enterprise Java Beans*
Equinox	*OSGi implementation developed by the Eclipse Foundation. It is used to power modular architecture of Eclipse IDE in latest releases*
Flash	*Adobe Flash Player (http://www.adobe.com/products/flashplayer)*
GWT	*Google Web Toolkit. (http://code.google.com/webtoolkit/)*

Hibernate	*Opensource Java persistent framework (http://www.hibernate.org/)*
HTTP	*Hypertext Transfer Protocol*
Java	*http://java.sun.com*
Java FX	*http://java.sun.com/javafx/*
Java Web Start	*Technology developed by Sun Microsystems, which enables to deploy web applications as downloadable components accessible through internet browser.*
Javascript	*Standardized scripting language present in all major browsers.*
Javassist	*Class library for editing Java bytecodes (http://www.csg.is.titech.ac.jp/~chiba/javassist/)*
JDBC	*Java database connectivity (http://java.sun.com/javase/technologies/database/)*
Jetty	*Opensource embeddable Java servlet container (http://www.mortbay.org/jetty/)*
JNLP	*Java Network Launch Protocol. Used to describe features, contents and launch configuration of applications deployed through Java Web Start.*
JRE	*Java Runtime Environment*
JSP	*JavaServer Pages*
JXPath	*Interpreter of an expression language called XPath to graphs of Java objects (http://commons.apache.org/jxpath/)*
Microsoft Silverlight	*http://silverlight.net*
OS	*Operating System*
OSGi	*Java-based service platform (http://www.osgi.org)*
PARIA	*Pervasive Adaptable Rich Internet Application*
Perl, Python	*Dynamically typed programming languages with good support for textual data and web technologies. Both often used as server-side scripting languages. (www.perl.org, www.python.org)*
Pervasive Computing	*Pervasive computing is a software model bearing in mind increasingly ubiquitous connected computing devices in the environment and focuses on distributed data rather than on one computer device. Also Ubiquitous Computing.*
PHP	*Server-side HTML embedded scripting language (www.php.net)*
RAP	*Rich AJAX platform. Eclipse web UI project based on RCP. (http://www.eclipse.org/rap/)*
RCP	*Rich Client Platform. Eclipse UI project. (http://www.eclipse.org/rcp/)*
RIA	*Rich Internet Application*
RMI	*Remote Method Invocation*
RPC	*Remote Procedure Call*

SWTSwing	*Port of the SWT graphical toolkit to Swing. (http://swtswing.sourceforge.net)*
ZK	*RIA and AJAX toolkit (http://www.zkoss.org/)*

Enclosed CD

The CD enclosed with the thesis includes full source code of development framework and demo applications. It also includes a separate Eclipse plugin and whole Eclipse distribution for Linux and Windows with all the tools needed to build the source code and use the framework.

File `README.html` on the CD includes more detailed information about contents of the CD and how to use them.