W3CAR

Teleseminar on E-Commerce
September 2001 – February 2002

Johannes Walch
Wolfgang Kiess
Emmanuel Castellani
Laurent Marchese

Université de Nice-Sophia Antipolis
IUP MIAGE

Universität Mannheim
Wirtschaftsinformatik

1 Summary

2 Introduction

2.1 Context

The w3car project was initially designed as a component of the Speed3 project, which is a cooperation of DaimlerCrysler (DCX) and the MBDS degree in the Nice-Sophia-Antipolis University. The goal of the DCX Speed3 project is to create a prototype of a UMTS car to be demonstrated on the Monaco Telecom (MT) UMTS network.

2.2 Target of the project

The sub-project assigned to the Nice- Mannheim students can be called “IPAQ Entertainment application”. It addresses the only part of the DCX demand that does not requires complex technologies like Video servers or OSGI architecture : backseats entertainment applications.
The main goal of the project is then to build a Java application on the IPAQ Client that enables the user to download and run leisure programs on the IPAQ.
The main focus of this study is to set up the client Application (mechanisms to download programs over network) ; one simple game should be designed for an example.

3 Architecture

The system is implemented in the classical way of a n-layer architecture with game servers, database, application server (which consists of a webserver with jsp and servlet engine) and client.

Here´s the description of the architecture drawn in Figure 1 :

1) The webpage containing the list with all available games is generated out of the database via JSP and shown in the browser on the IPAQ. The user selects a game by clicking on the link. This leads to the call of a servlet which generates the XML-message which contains the name and the URL of the game (the game can be on a different server, see 4).
2) The message is send to the MessageService on the IPAQ.
3) The message is passed to the Classloader
4) The Classloader retrieves the information out of the message, loads the jar file and unzips it.
5) The main class with the start method is passed to the main window, the game is started.

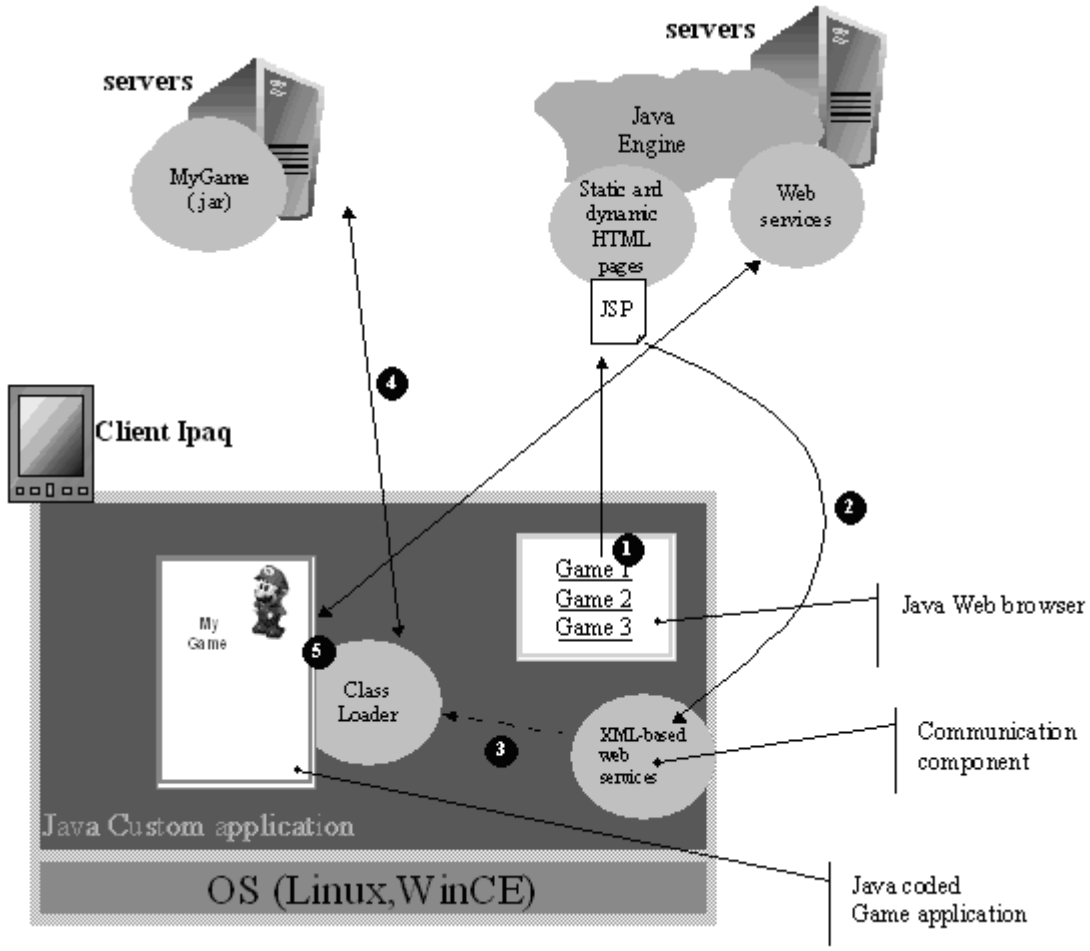


Figure 1 : Overall architecture

3.1 Servers

The server consists of two parts, one for the dynamic creation of the webpages and the servlet part, which produces the messages. The code in the JSP-page uses JDBC to connect to the database and extract the information about the games. The generated HTML-page is then send back to the client. When the users clicks on a link, a GET-request is posted back to the webserver where it is processed by the servlet. It produces the message and sends it to the client via the MessageService.

3.2 Client

In the following figure 2 you can see the class diagramm of the client application.

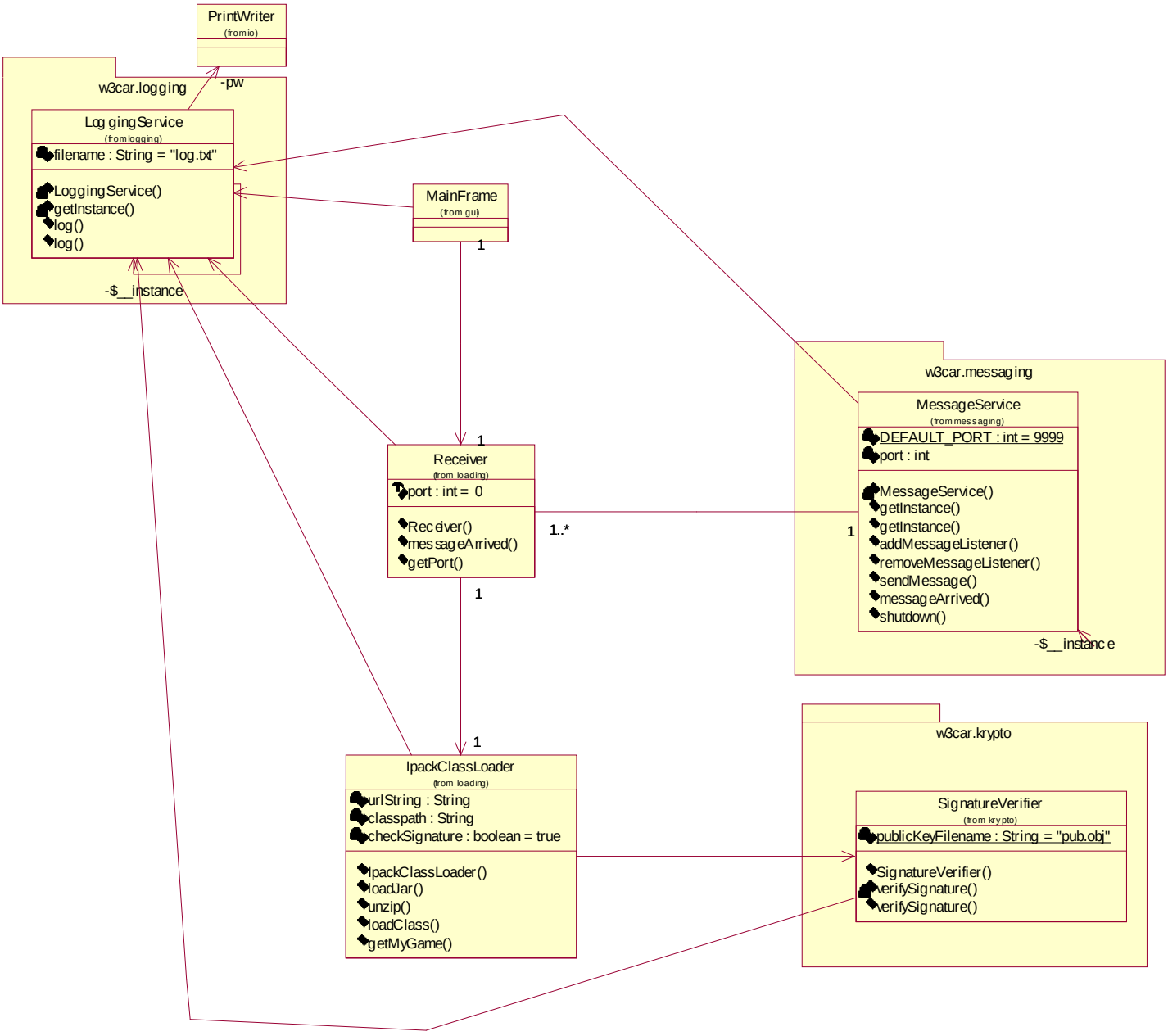


Figure 2 : client architecture

We have the primary class MainFrame, Receiver for receiving messages, w3car.messaging for the messaging service (xml), w3car.krypto for the signature processing, IpackClassLoader for downloading and executing the jar files and finally w3car.logging for our custom logging service.

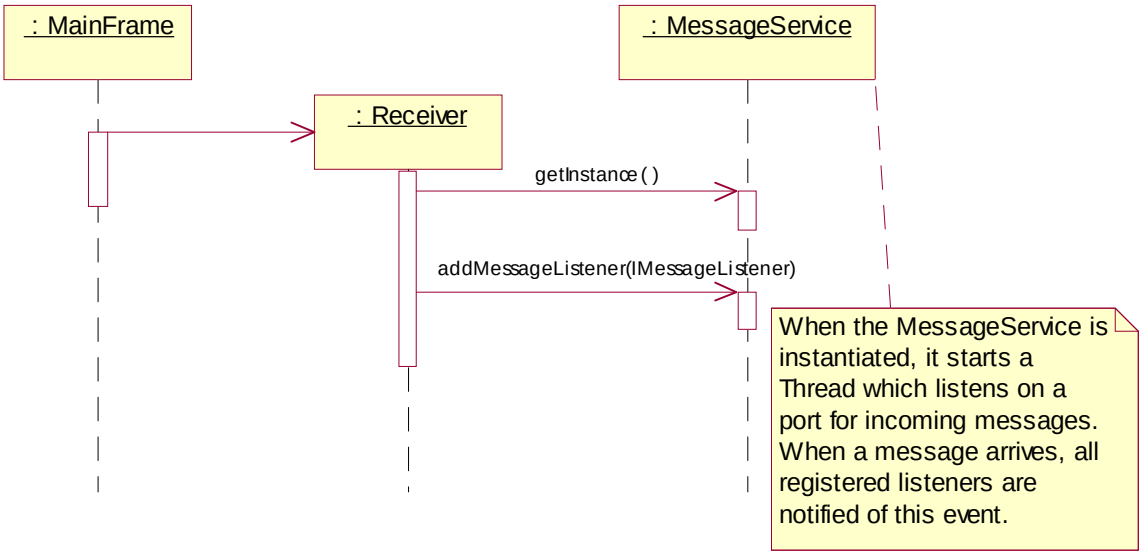
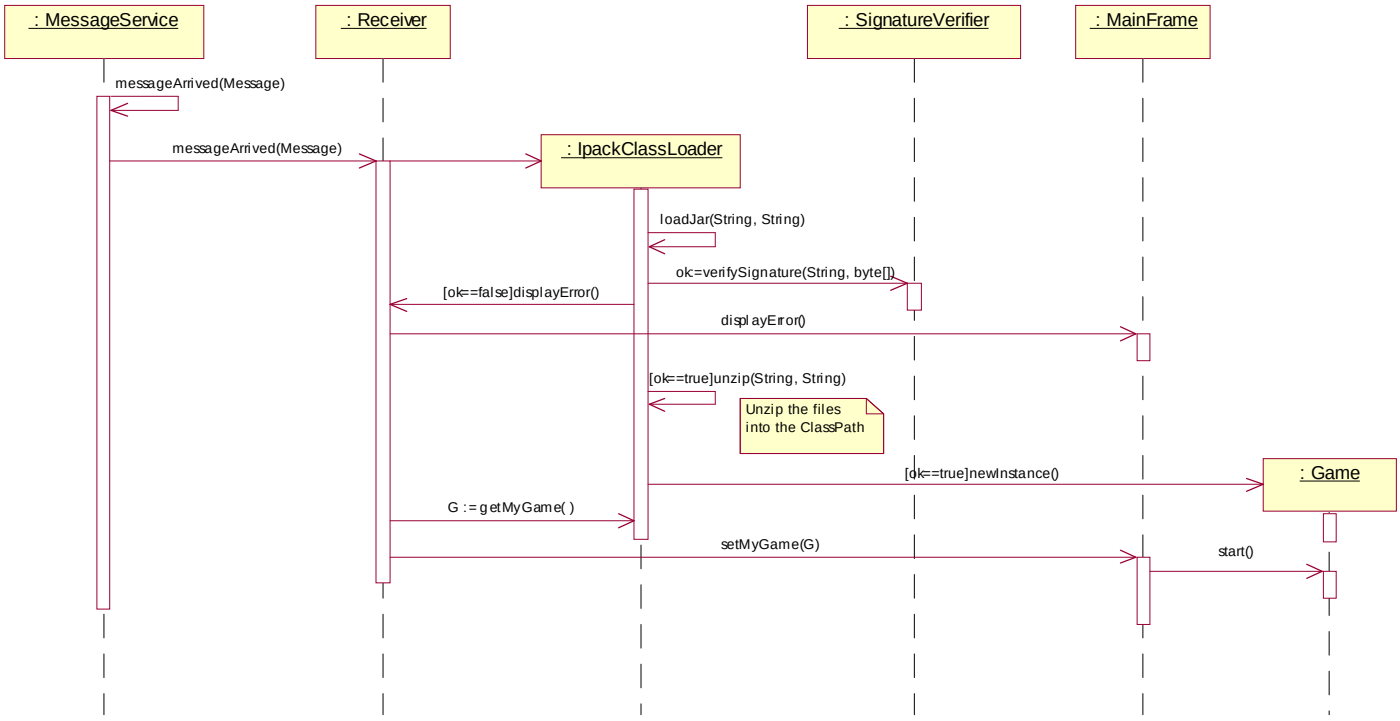


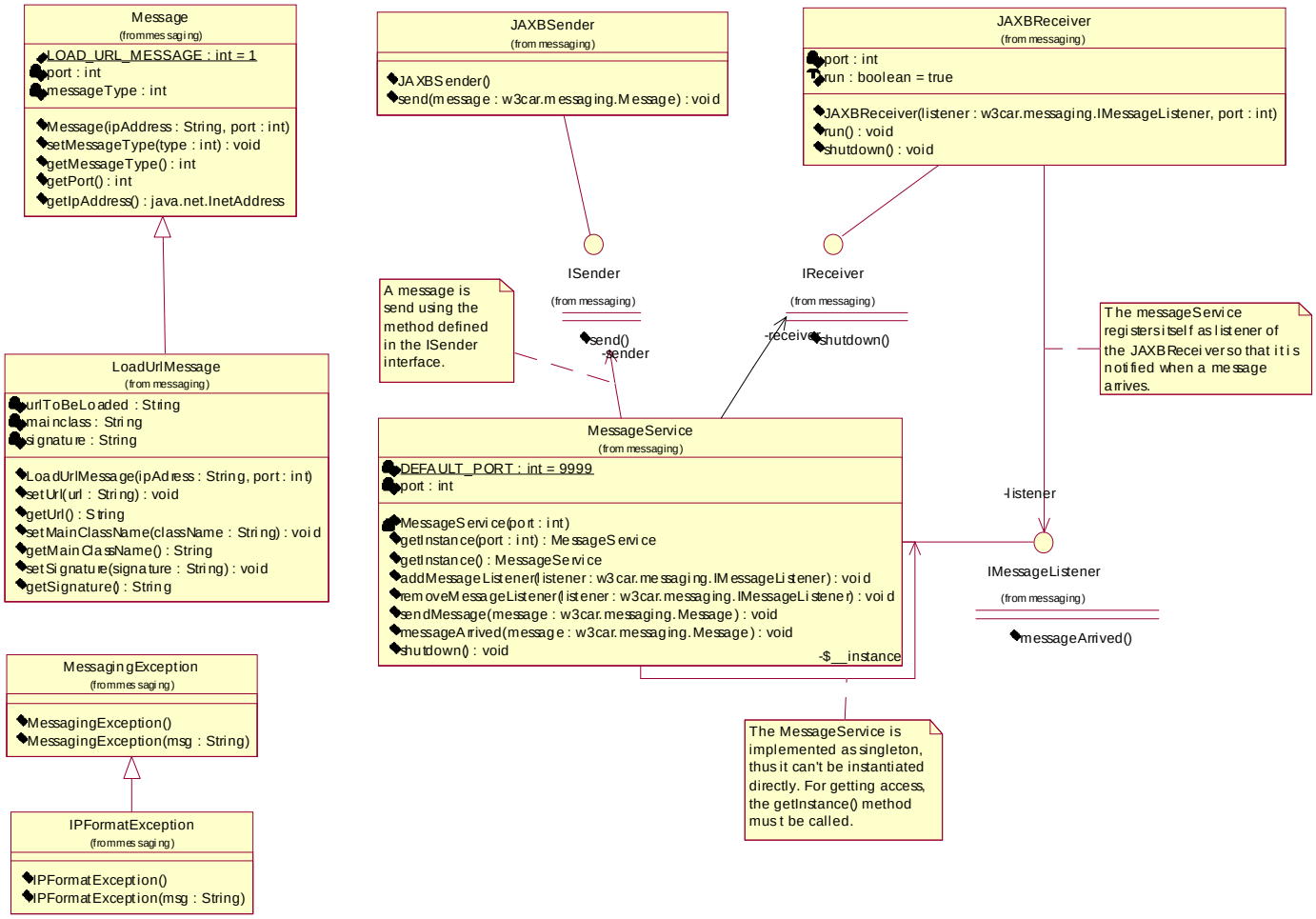
Figure 3 : client startup

The client starts the MainFrame class via its main()-Method. Here, a receiver is instantiated which registers itself as a MessageListener at the MessageService. When a message arrives at the MessageService, it is unmarshaled to a Java-Object and passed to the Receiver. The Receiver then extracts the message and calls the IpackClassloader to load the Jar-file. The jar file is loaded, verified against the signature (which can be found in the message), written to the classpath and unpacked. The classloader creates an instance of the game class. The MainFrame then fetches the reference to the newly created game class through the getMyGame method and loads the class.



pict 4 : receiving a message

3.3 Messaging



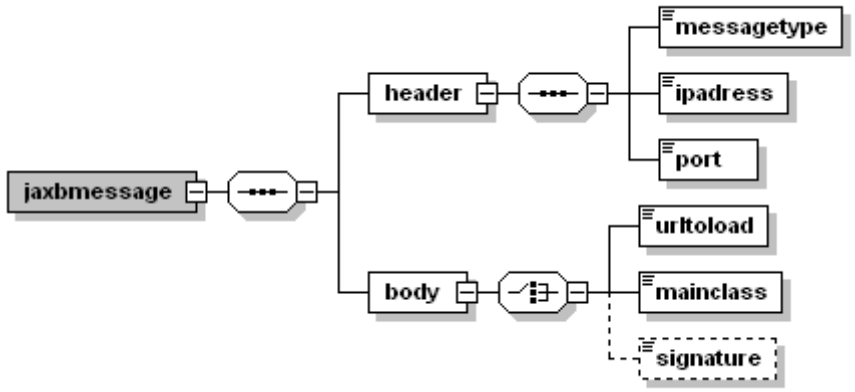
pict 5 : client startup

We introduced a messaging API for the programmer so he can access it transparently regardless of the underlying communication architecture. This API provides generic set and get methods for every parameter of a given message type. Currently there is only one message type "LoadUrlMessage" (see 4.3.1) implemented.
Through this methods the data is transferred to the underlying communications architecture which can though be exchanged without changing the application. It is also required in the API that the receiver side implements an event listener for the arrival of a message.

For our implementation, XML messaging using the CASTOR package which marshals Java Objects into XML Strings is used. This implementation of the messaging API uses the parameters it has received from its own set methods to call the JAXBSender method which transmits the data to the other communication endpoint after converting it to a plain text XML String.
On the receiver side the XML String is then unmarshaled, the event listener gets alerted and can access the transmitted data via the API´s get methods.

The transmission of the XML String via TCP/IP is implemented as a singleton Service which always has a receiver and a sender listening on the fixed port 9999. The target address is extracted from the http request.

3.3.1 Message grammar



This is a representation of the schema-grammar for a LoadUrlMessage. It is used to create valid (in means of XML) String to be transmitted over TCP/IP socket connection.

3.4 Cryptography

The application provides the optional possibility to verify if the loaded jar file was correctly signed by a trusted individual. This was motivated through the danger that probably malicious code which presents itself in a harmless manner to the enduser (virus) could be executed on the IPAQ.

The game developer needs to sign its jar files manually to make use of this service. For this purpose we provide a tool w3carsigner.bat on the server.
It is called via the batch file w3carsigner.bat, the argument is the file which has to be signed. It computes a signature using the DSA key in the w3carkey keystore, converts the byte[] representation of the signature to a hex String and writes it to a file. The name of the file is written to stdout. The signature files need to be installed with the jar files on the web server.

When the webserver tries to send a LoadUrlMessage to the client it tries to read the signature file corresponding to the jar file. If it exists the signature String is extracted and transmitted in the message.

As soon as the client receives a LoadUrlMessage it is checked if there is a valid signature in the message and if yes it checks the signature against the downloaded jar file. For the moment there is only a message on the console if the jar file is correctly signed or not. The game is loaded anyway. It is left for further development to react accordingly e.g presenting a window with different choices to the user. This was not fully implemented because of some problems with the IPAQ JRE (see 7.1).

4 Implementation

4.1 Product Environment

4.2 Software

4.2.1 Server

Operating system	Microsoft Windows 2000 Professional
JDK	Sun Microsystems, Java 2 SDK Standard Edition v1.3.0_02
Webserver	The Apache Foundation Apache HTTP Server 1.3.23
Servlet Engine	The Apache Foundation Tomcat 4.02
Database	MySQL 3.23.41 on Unix

4.2.2 Client

Operating system	Windows CE 3.0
JRE	Personal Java Runtime v1.1 Beta 1
Swing	classes from jdk 1.3 imported and integrated with the JRE

4.2.3 External components

4.2.3.1 Castor 0.9.3

"Castor is an open source data binding framework for Java" (Castor webpage http://castor.exolab.org) . In this project it was used to marshal message objects into XML via the easy way of data binding, transmit them over the network as XML and unmarshal them on the other end of the connection. With this tool it was possible to generate all the necessary classes for the marshalling and unmarshalling out of an XML-schema, which makes it easy to change the whole program just by changing the grammar and regenerate the XML-binding classes.

4.2.3.2 Xerces

Xerces is the XML-Parser used by Castor. It is an apache open source project, more information can be found under http://xml.apache.org/xerces-j/index.html .

4.2.3.3 Browser

First we use a java bean naming IceBrowser. It is a complete webBrowser (http://www.icesoft.no/ICEBrowser/). But because of some problems we finally use some code of a simple webBrowser that we get on the java sun site.

4.2.4 Additional files (non source-files)

The additional files which are necessary for running the program contain configuration information, the keys and signatures for the jar files and so on.

4.2.4.1 generateKey.bat

PURPOSE: a batch file containing the call to the JAVA KEYTOOL for the generation of the keys necessary for the system (it contains some configuration information like the name for the keystore, the algorithm to use)
ARGUMENTS: The keystore name and the password for it
LOCATION: on the computer where the signature should be produced and in the same directory as the keystore (w3carkey)

4.2.4.2 message.xsd

The XML-schema grammar describing the format of a LoadUrlMessage. (see 4.3.1)

4.2.4.3 pub.obj

The serialized version of a public dsa key. If the full cryptographic packages are available on the IPAQ, the keystore file "w3carkey" can be used for extracting the public key.
LOCATION : in the working directory of the JVM with which the signature should be verified, the SignatureVerifier loads the key (this means in our case in the working directory of the application)

4.2.4.4 w3carkey

The keystore file which contains the public and the private key for the signature, it was produced with the java KEYTOOL with the arguments given in the generateKey file
LOCATION: in the directory where the w3carSigner batch file is executed, the signer needs to open the file to retrieve the private key

4.2.4.5 w3carsigner.bat

A batch file for starting the w3carSigner (it needs to access the w3carkey keystore to retrieve the private key)

4.2.4.6 Properties.txt

The property file is used to configure the application. It contains the following parameters:

classpath	one directory in the classpath, this is the place where the Classloader puts the jar file and the extracted classes (in the correct directory structure) on the client.
starturl	this is the url which is seen first in the Browser of the application
browserheight	The height of the Browser / application window, with this the application can be easily adapted to bigger or smaller screens (other Palm devices, a laptop with more space on the display,…)
browserwidth	The width …
separator	The character which separates the directories in a pathname String. This had to be added due to the faulty implementation of Java on the IPAQ.

4.3 Hardware

4.3.1 Server

- Notebook Pentium III 650 MHz
- 256 MB RAM
- 802.11b compatible PCMCIA Wireless LAN Adapter

4.3.2 PDA

- Compaq IPAQ H3660, 64 MB Memory
- PCMCIA Expansion Pack
- 802.11b compatible PCMCIA Wireless LAN Adapter

4.4 Changes in the design

On of the major changes from the design to the implementation is the use of a XML data binding framework based on JAXB instead of using SOAP, the industry standard for XML-communication for the XML communication. This decision was taken due to the fact that SOAP isn't the right protocol for our purpose. SOAP is designed for a request-response behavior (although there is also a one way communication possible) for web services where the client makes a response, a service is executed on the server and a response is delivered. The processing logic for a SOAP message can be very complex and heavy, where in our case the message must be processed on a palm device with very limited processing power and memory.
In the w3car communication, only lightweight content is transmitted (the address of the jar-file, the name of the name class and in a later version a signature for the jar-file) from the server to the client.

5 Experiences

5.1 IPAQ

5.1.1 JRE on IPAQ

Unfortunately the Java support for the target architecture (IPAQ running Windows CE 3.0) is very poor. For production applications we would suggest a architecture with better Java support (either Linux or a special Java OS).

Through some research on the world wide web we found two possibilities :

- Personal Java Runtime (Beta) from sun (development discontinued)
- Different OS than Windows CE 3.0 (Linux or Java OS)
- Insignia JEODE (commercial JRE)

Both JREs have Java support similar to early JDK 1.1.

What made the decision easy is that the integrated Browser classes are based on Swing. We decided to use Personal Java Runtime because it was possible (with some effort) to port swing to

it. Swing uses no architecture specific code because it is a lightweight implementation on top of AWT. All that needs to be done is pack all the necessary class files into a jar file and point the classpath there. We decided not to install another OS with better Java support because there were warning that the IPAQ might be rendered useless if something goes wrong. Due to the already short time left we decided to take the chances.

5.1.2 Porting the Program

First we made several tests on the IPAQ with the different modules we planned to integrate with the following results :

- XML communication libraries work
- Java applications on the IPAQ are very slow
- Swing worked but did not display everything correctly (esp. text using fonts)
- The Personal Java VM is not stable. It crashes the IPAQs Windows CE OS regularly and also the USB connected laptop.

The application was developed on Windows PC running Jbuilder. From time to time the actual version was uploaded to the IPAQ and tested for compliance. During the development process we made the experience that developing an application for another architecture than the one your Development Environment runs at is very complicated.

5.1.3 WLAN connection

We decided to use a simple network configuration for the demonstration of the project we could duplicate at home for development. The IPAQ connects to the Server over a Standard WLAN connection in peer-to-peer mode (ad-hoc).

5.1.3.1 Security

For security a 128bit WEP Encryption is used. This is not satisfactory for high security applications because of some design flaws in the WEP Protocol. We choose it anyway because its hardware support makes it fast and entertainment applications are not high security. If e.g. payment has to be integrated in the system, a higher level SSL/TLS connection would be more suitable.

5.1.3.2 Network configuration

The IPAQ and the PC used for demonstration have PCMCIA cards which make the wireless connection to the university network. The server running the Servlets/JSPs is located in the university of Nice. This setup was used to show the internet compatibility of the implementation.

5.2 Classloader

We use our own ClassLoader to download files located on different servers by giving an URL address and the name of the game. The game is downloaded over HTTP and is instantiated/initialized automatically.

The first implementation we did allowed us to download directly the game without saving it in the file system by using a JarURLConnection. It was working very well but unfortunately the Ipaq doesn't support this class (only the JDK 1.1.8 is installed). So we decided to use an other way to download the game: first we save it in the fileSystem by using this time a standard URLConnection and second we unzip it into the classpath. Finally, the main class is instantiated. We decided for it to use the simple method of the class Class : newInstance(). So, each game must implement an empty constructor.

5.3 WebBrowser integration

To have a visual and dynamic way to get the .jar and the name of the game, we thought to use a webbrowser. For this, we decide first to integrate a complete webbrowser with the IceBrowser bean.
Before to have a problem with the integration in the IPAQ, we had a problem of conflict between the webbrowser Classloader and our Classloader, exactly with the security manager of the IceBrowser.
So we decided to use a simple webbrowser without Classloader. Indeed, we needed only to get the URL and the name of the game to download.

5.4 Games development

The developer need the abstract class Games to develop his games which must extends this class. The abstract class Games extend JPanel, so each game developed can be insert into our MainFrame.
By using this system the developers must redefined all abstract method, so we are sure that all games will have a method start() and a method stop().
So in our ClassLoader we have just to cast the class download into Games, without knowing anything about it.

5.5 Organization

The specification of the project arrived quite late (end of november), thus there wasn't any work possible before the beginning of december 2001. The specification which arrived then was quite different from the one originally established by the team: the initial plan intended a system for an extended UMTS telemetry service in the car which can be used in breakdown situations to establish an audio connection to the next garage and exchange telemetry data. This initial specification can be viewed on the web page of the project.

5.6 Team communication

Our team communication was based on 2 technologies using the english languages:

- Video conferences
- EMail

Video conferences

Video conferencing was new to the team members so there were some difficulties in the beginning, mostly understanding problems. This was due to the sometimes bad audio quality and the use of foreign languages. Over the time everyone got used to the teleconferences and it proved to be an important part of the communication. Very similar to a personal meeting we used the teleconferences for general discussion and important decisions. Also the fixed time schedule for the conferences provided some mandatory delivery dates helping everyone organizing their work.

EMail

A mailing list including all the team members was set up to simplify the communication
Email communication worked good from the beginning because obviously everybody was used to it. The main disadvantage of the Email communication is the large delay. The advantage is that there is no need for everybody to be at a specific location, that it is more easy to understand written foreign languages than spoken ones and the possibility to attach files illustrating what you are trying to explain.

6 Outlook

6.1 Krypto package

The krypto package can be found in w3car.krypto.*. It contains three files, one for signing JAR files, on for verifying these signatures and an additional tools class which is used for converting between byte[] and hex-Strings.
The krypto package produces signatures for the JAR-files with a 1024 Bit DSA key. The problem on the IPAQ is that the version of the JDK is 1.1 without the cryptographic packages (java.security.*, sun.security.provider.*). There is a switch in the Properties file for the krypto package, it must be switched to off for running it on the IPAQ.

7 Additional information

7.1 Webpage

http://w3car.nwe.de
The website is password protected. Use the following login data for access :

Username : w3car
Password : chrysler

The website contains the following things :

- Description of the project
- Contact information
- Downloadable source code ZIP file
- Protocols of the teleconferences
- Various documents and binary files related to the project

7.2 Javadoc

For all the classes, there is a javadoc available which ships together with the application. It´s available at this url : http://w3car.nwe.de/javadoc/index.html

7.3 Contact

Johannes Walch	j.walch@nwe.de
Wolfgang Kiess	wolfgang.kiess@web.de
Emmanuel Castellani	emmanuel_castellani@yahoo.fr
Laurent Marchese	laurent_marchese@yahoo.fr