Semester Thesis

Java on the iPAQ

Nicolas Burri
nburri@student.ethz.ch

Dept. of Computer Science
Swiss Federal Institute of Technology (ETH) Zurich
Summer 2003

Prof. Dr. Roger Wattenhofer
Distributed Computing Group
Advisor: Aaron Zollinger

Contents

Chapter 1

Introduction

Over the last years the capabilities of small handheld computers, better known as PDAs, have increased significantly. Today these small devices have become powerful enough to run programs which were designed to work on bigger hardware (like notebooks) only one or two years ago. Not only the CPU speed of these devices has improved but also additional features like Bluetooth- or WLAN-Support have become widely available and allow a whole new range of applications.

Especially in the field of mobile applications there is a demand for such small hardware devices, since even a slim notebook becomes uncomfortable for the user if it must be carried around all day long. Therefore it makes sense to evaluate the complexity of adapting existing mobile applications in order to run them on one of those much smaller handheld devices.

The task of this semester thesis was exactly to do this by means of a practical example. An existing middle-sized Java application was adapted to run on a Compaq iPAQ and the problems that occurred during this process were evaluated. In this documentation some of the results of this work will be presented and a collection of hopefully helpful tips for further iPAQ projects can be found in the appendix.

Chapter 2

The Given Program

To get a good overview of the capabilities of Java on the iPAQ, a test program was required that made use of several different Java packages under real conditions. The choice fell on the server-less Instant Messenger, developed by the Distributed Computing Group at ETH Zurich. The decision to port exactly this application was based on the fact that its source code was well known and its required Java packages will probably also be core parts of possible future projects.

2.1 The Basic Concept

This special version of an Instant Messenger follows a different approach than today's widely used programs like ICQ or AIM. The basic idea is to have a local ad-hoc wireless communication network which operates without any stationary hardware. There is no central server where the users have to register, and also no WLAN access points are required. The communication is purely based on IP broadcasts, which means that all clients have to join a multicast group known in advance to participate in the network.

The program offers different fuctions to deal with the problems that arise in such an unreliable network. For example it supplies a special multi-hop layer in the communication protocol to solve the problem that in an adhoc WLAN network usually not all clients can reach each other directly, due to their restricted signal strength. This multi-hop layer allows a sender to use other clients as repeaters for messages that cannot reach their target directly. Like this, as long as any path from sender to receiver exists, the message will be delivered.

The client also offers a couple of sniffing functionalities which show the local traffic on different abstraction levels of the communication protocol, and a neat network graph can be opened that plots the topology of the communication network, showing all directly and indirectly reachable communication partners.

2.2 Important Java Aspects

From the Java point of view, the Instant Messenger consists of several packages including more than 60 classes. Three aspects can be seen as key components that are essential for the program to run:

Multicast Socket: As mentioned above the communication is based on UDP/IP broadcasts and therefore the Java `MulticastSocket` class is the vital component for the exchange of messages.

Multi-Threading: Since the Instant Messenger needs to be able to send and receive messages at the same time without locking the GUI, it runs in several threads.

Swing: Though, not as essential as the first two points, Java Swing still represents a key component for the system: The GUI of the Instant Messenger makes use of several Swing components, and a complete replacement of the interface would be very time consuming.

Chapter 3

Getting Started

3.1 How to Choose a Virtual Machine

As every other hardware platform also the iPAQ requires a Java Virtual Machine to run Java byte-code. Several companies offer different Virtual Machines for the iPAQ which all support parts of the outdated Personal Java standard and/or parts of the Java 2 Micro Edition (J2ME). Unfortunately every manufacturer has chosen to include a different subset of Java packages in their product, what makes it almost impossible to decide on one Virtual Machine for all possible tasks. It is therefore inevitable to analyze the program that needs to be adapted and to identify all important Java packages that need to be available, before the work on the code can begin. A good overview of the currently available solutions, their advantages and disadvantages, offers the Semester Thesis of David Mayor from the EPFL. [1]

3.2 Jeode

After some investigations we decided to use Jeode, the Virtual Machine from Insignia, for this project, since it offered all important packages we needed and also its general user feedback found on the internet was mostly positive.

Advantages of Jeode:

- The `java.net` package including the `MulticastSocket` is fully supported.
- Multi-Threading is available, including `synchronized` functions
- Java Swing, even though not officially supported, can be refitted (also see A.2)
- Jeode comes with the iPAQ CD and can be used with an out of the box handheld system running Windows CE 3.0
- A console mode is available. Even though it seems unnatural, there are VMs for the iPAQ that don't offer a simple console for printing.
- Jeode is quite sophisticated and comparatively well tested.

Disadvantages:

- There is only a very poor documentation about the supported packages and system arguments for Jeode available. Also many of the other Virtual Machines suffer from this handicap and often one has to rely on third party information.

Till the end of the work we never regretted to have chosen Jeode, since tests with its archrival, the Virtual Machine from NSIcom called CrEme proved that we would have run into similar problems if we had chosen this alternative. On a side note: Although CrEme officially supports Swing, the menu bar of the Instant Messenger was not visible and the generated frames were always in full screen mode without any close button.

Chapter 4

The Adaptation Process

4.1 The General Problem

Once the Virtual Machine is ready, the real work can begin. The whole development of a Java application for the iPAQ takes place on the PC, since neither compiler nor development environment are available on the handheld device. The major problem that arises from this fact is that debugging becomes very complicated. A program can be compiled and tested on the PC with its Java Development Kit from Sun Microsystems, and once the class files are copied to the iPAQ and run there, a `MethodNotFoundException` will probably be the only output before the program terminates. This happens, since the application uses Java classes which are available on the Virtual Machine of the PC but not on the one of the PocketPC. The only way to deal with this problem is to strip the program to a minimal skeleton and to reactivate the components bit by bit. This means that a lot of files must be copied from the PC to the iPAQ over and over again, which can be quite tedious, since the throughput (especially over USB) can be very low. As soon as a missing component is found and identified, there are two possible workarounds to fix the problem:

4.2 Solution One: Re-Implementation

If a whole class is unavailable, it may be reasonable to write an own partial implementation of the missing object. Due to the rather complicated inheritance of most of the native Java classes, it will usually not be possible to offer a full re-implementation of the object, but at least the missing functionality can be made available. A good example where this solution makes sense, is the missing `LinkedList` class that offers some handy functions and therefore is being used in many Java programs. A partial re-implementation of this class might look like this:

```
public class LinkedList{

  private Vector myVector = new Vector();
  public void add(Object o){ myVector.addElement(o); }
  public int size() { return myVector.size(); }
  public Object removeFirst() {
    Object tmp = myVector.elementAt(0);
    myVector.removeElementAt(0);
    return tmp;
  }
  .
  .
  .
}
```

Advantage: The advantage of this solution is that there is hardly no need to change the code of the application, since usually a simple import is all that needs to be adjusted to fix the problem. As a nice side effect of this clear separation of application and bug fix, it will be possible to port later versions of the same program very efficiently, since the changes in the old version will be easy to reapply to the new one.

Disadvantage: The drawback of this solution is that the re-implementation of the missing objects must be seen as a hack. If at any time the existing code is going to be extended it will be very probable that the current bug fixes will lead to several new problems, especially if manipulated classes are being inherited.

4.3 Solution Two: Code Replacement

The other possible approach is to replace calls to missing classes or methods with calls to existing components. Especially if a certain class is available on the handheld's Virtual Machine but does not support all methods it is supposed to, this solution will usually be the better choice, since in most of those cases the class offers substitutes for the missing methods. For example on Jeode the Vector class is missing the two methods `Object get(int i)` and `void add(Object o)`, but their substitutes `Object elementAt(int i)` and `void addElement(Object o)` are available. Replacing the calls to the missing methods with calls to their functional twins will do the job most of the times.

Advantage: The advantage of this solution is that the resulting code will not "fake" any information to produce a running program, and so follow-up problems are improbable.

Disadvantage: Still, this way to solve the problem has a severe drawback as well. Since a lot of changes at many different places in the source code are necessary, it will be practically impossible to recover them all at a later time. Therefore even a small update of the original application will be very hard to include in the handheld version of the program.

Finally both ways lead to a working iPAQ program, and so the decision which approach to take to deal with a missing component must be made case by case. Unfortunately, once all the missing classes and methods are repaired, there may still be some problems left that have their origin in the implementation of the Virtual Machine or even are the result of a hardware problem. These bugs are very hard to track down, since they tend to produce non-deterministic application behavior. Furthermore, due to the fact that the code of the program seems to be correct, it is difficult to decide where to start searching for the problem. The next section will show the worst bug of that kind that occurred during the adaptation of the Instant Messenger.

4.4 The Worst Problem

4.4.1 The Symptoms

Once the Instant Messenger was adapted to a level where sending and receiving of data became possible, a lot of the incoming messages were rejected by the communication layer of the Instant Messenger with the error message: "Unknown type received".[1] The strange thing about this error was that the same code did not produce any problems on the PC but only on the iPAQ. Also did the situation not get any better even when the whole application was adapted to run on the PocketPC, and so we started to search for the reason behind this bug. Soon it became clear that the incoming messages were already defective at the level of UDP packets, and at the same time a network sniffer on a PC proved that all messages which were transmitted over the network had a correct format.

4.4.2 Expected Reasons

The first suspicion was that the `MulticastSocket` produced the problem, probably due to collisions while receiving the packets. After some intensive testing it became clear that this was not the case, since it was impossible to reproduce the problem within the specially written test program. Of course it was possible to enforce packet loss if there was too much traffic on the network, but all the captured packets were fully correct. The next idea was that the CPU of the iPAQ was overloaded because of the many threads used by the application and that this would lead to corrupted data read from the socket. Unfortunately also this assumption could not be confirmed by any test program, and so we had to try to find the problem somewhere else.

[1]This message was produced, since a high level data packet was detected to be faulty.

4.4.3 The Real Reason

After days of experimenting we finally found the problem lying in the definition of the `DatagramPacket` that was generated to store the incoming data. The length of the data buffer and the `DatagramPacket` itself was set to 0x10000, which means 65536 bytes. This is exactly one byte more than the maximum possible UDP packet size. Calling the `getLength()` method of the `DatagramPacket` before actually receiving data also returned exactly this correct value, but as soon as there was data stored in the object, it became corrupted.

The only plausible explanation we found for this behavior is that the mobile Virtual Machine must have been optimized to run with a minimal amount of memory and therefore does not expect a `DatagramPacket` to be bigger than the maximum UDP packet size. This thesis is supported by the fact that, as soon as we reduced the packet size by one byte, the error disappeared and never showed up again.

Chapter 5

Conclusion

5.1 The Program

The last release of the adapted Instant Messenger is now working on the iPAQ and supports all features of the original version. Tests with several clients on other machines forming a communication network showed that the handheld device is capable of dealing with the traffic of six and more clients. Precise tests to determine the maximum number of clients the iPAQ could deal with were unfortunately not possible, since there were not enough peer computers available. Experiments with more than one instance of the original program running on a computer were also not really helpful, since the instances interfered with each other and the whole network became unstable very fast. The biggest remaining nuisance is the very slowly responding GUI. Swing obviously demands too much computing power to run at a reasonable speed on the iPAQ. For the Instant Messenger application the resulting delay is still acceptable, but for programs that require more interaction with the user, another GUI library will be necessary (also see A.2). Nevertheless Jeode has proved, that it can handle a sophisticated Java application, and that it certainly represents a good choice for further projects.

5.2 Personal Experience

Working on this project was for sure an interesting experience since it was exciting to see what problems can occur if an ”exotic” hardware platform tries to run default software. It turned out that the *”‘Write once, run Anywhere”’* slogan often propagated in the Java context should not be taken too literally, since the degree of portability depends heavily on the quality of the implementation of the Virtual Machine.

There were times when the work was very tedious and frustrating, especially if an unexplainable bug stopped the progress for several days, but solving these bugs in the end was all the more rewarding.

Finally, the information listed in the following appendix, will hopefully help an iPAQ newbie to get started and perhaps to avoid some of the problems we had to deal with.

Appendix A

Tips and Tricks

A.1 Getting Started

Q: Where can I get Jeode and how do I install it on the iPAQ?
Jeode can be found in the corresponding folder on the iPAQ CD-Rom. To install the software, just start the .exe file in this folder, and the installation will be executed automatically. (This requires an installed version of `Active Sync` on the PC)

Q: Where can I get the Win CE/PocketPC Driver for my Cisco Aironet 350 WLAN adapter?
The necessary driver and configuration tool for the Cisco Aironet 350 WLAN adapter can be downloaded from the Cisco homepage [2]. Make sure to get this version: "Windows CE 3.0 (Pocket PC 2002)"

Q: How should I transfer my programs to the iPAQ?
If a lot of small files have to be transferred from the PC to the iPAQ, the throughput over USB is less than 3 kB/s. If the transfer is done over WLAN, at least a value of about 30 kB/s can be reached. Therefore either set-up `Active Sync` to use WLAN or just use the Windows network directly to copy your data to the handheld.
To do this you have to share the PC folder containing your files for the iPAQ over the Microsoft Network (Usually possible by right-clicking the folder). Once the PC is ready start the `File Explorer` on the iPAQ and choose to open a remote source by clicking on the `Open` button at the bottom of the screen. A pop-up window will open and ask for a URL. Enter the path to your shared data folder on the PC (e.g. `\\myPc\theDataFolder\`) and click OK. If everything works correctly, you should be able to copy the files from the PC now. To paste them into the memory of the iPAQ, switch to the folder where you want to put them (choose the first of the three icons in the toolbar at the bottom of the screen to get back to the iPAQ memory) and click on `Edit->Paste`

Q: Which Java compiler should I use?
Even though you can use any Java compiler, using a Sun SDK of version 1.4 or

newer seems to produce problems. A good solution is to get a copy of the outdated `jdk1.1.8` and to compile your program with this development kit. Since Personal Java was based on Java 1.1, it is missing many of the modern Java classes, and using this old compiler will help you to identify those at compile time already and not only when you try to run your application on the iPAQ.

Q: How do I set-up Eclipse to use a special compiler?
Eclipse insists on using its own Java compiler but it allows to specify the Runtime Environment used to run a program. According to this Runtime Environment Eclipse also highlights compatibility problems during development time already. Therefore setting the Runtime to 1.1.8 will help a lot to find nonexistent Java components fast. The Runtime Environment can be changed under

```
Window->Preferences->Java->Installed JREs
```

Of course the JRE 1.1.8 must be installed first before it can be used in Eclipse.

A.2 Jeode

Q: How can I prevent the console from closing when my program terminates?
By default Jeode closes the console as soon as a program terminates (also in case of an exception). To prevent this, add the following parameter to your call:

```
-Djeode.evm.console.local.keep=TRUE
```

Q: I only need to see the first screen-full console output, can I prevent Jeode from scrolling?
Just add this parameter to your program call:

```
-Djeode.evm.console.local.paging=TRUE
```

Q: How do I create a link?
The command line parameters for Jeode become quite long and since the input with the pen is not very efficient, it makes sense to create a link for every of your Java applications. To do this, do the following:

1. Create a link on the iPAQ to the file `evm.exe` (if you have a default installation, this file should be in the folder `\windows\`)

2. Copy this link to the PC using `Active Sync` or the Windows network.

3. On the PC, start a text editor of your choice and open the link file. (Right-clicking on the file will not work) The content of this file should look like this:

   `18#"\Windows\evm.exe"`[1]

[1] A link has always the form **NUM#"URL"** where **NUM** is the number of characters of the **URL** plus 2 for the leading and closing "

4. Add the command line parameters *after* the closing ”. For example the link to one of our test programs looked like this:

```
18#"\Windows\evm.exe" -cp \swingall.jar MCS 100
```

5. Save the link and copy it back to the iPAQ.

Q: Can I use Swing on Jeode?
Insignia officially denies that Jeode supports Swing but clever users have discovered that the incompatibility between Swing and Jeode is caused by a small bug in the Swing 1.1.1 package which can be fixed easily:

1. Get Swing 1.1.1 [4]
2. In the Swing package you downloaded, there will be a `src.zip` file. Unzip it.
3. In the `Javax\Swing` subdirectory there will be a `SwingUtilities.java` file.
4. Go to line 677 of that class:
   `Method m = Class.class.getMethod("getProtectionDomain", null);`
5. Replace this line with:
   `Method m = Class.class.getMethod("getPackage", null);`
6. Save and compile the file.
7. Move the `swingall.jar` file from the download directory to the main directory of the extracted zip file. It should be where the `Javax` directory is.
8. Open up the command prompt and goto this directory.
9. Enter: `jar uvf swingall.jar javax\swing\SwingUtilities.class`
   to update the class in the jar file.

The resulting `swingall.jar` can be included in the classpath to enable Swing support for the application

Q: Does it make sense to use swing on the iPAQ?
Some applications depend heavily on swing and will need this library at any cost. In this case and for some first tests with a partially adapted application it makes sense to use Swing on the iPAQ. For programs that require a lot of user input though, Swing is definitively not the way to go. Besides the dull reaction time on user input, also the startup time of the application becomes ridiculously long. Just to load a test program with one `JFrame` containing one `JButton` and one `JLabel`, the iPAQ needs about 30 seconds. For real applications the startup time will be between 40 and 90 seconds.
Since it is difficult to rely on pure awt and still to get an appealing GUI, the use of special interface libraries is recommendable. An interesting freely available awt

extension is called `thinlet`[5]. Even though it is still under development, this open source project already offers a lot of components, which are all based on AWT. It is therefore compatible with virtually any Java version starting from Java 1.1 and also runs perfectly on Jeode. A very short test with a demo application provided by the thinlet download package showed some very promising results. The application loaded much faster than a comparable Swing program, and also the delay between a click with the pen and the reaction of the GUI was much smaller than with Swing.

Q: Does Jeode support jar files?
Yes, jar files are supported, but there is no `"java -jar"` option to start them. Since Jeode does not read the manifest to find the main class, it is not possible to have a self-starting jar file anyway. To use a jar file, just include it in the classpath and call the main class manually.

A.3 Debugging

Q: Can I use the Personal Java Emulator (pjee) from Sun Microsystems to test my programs?
Maybe... Unfortunately we could not get it to start our program, since it obviously does not support the inclusion of the `swingall.jar` we needed. If your application does not use Swing, you should give `pjee` a try. You can get it for free from here [3]

Q: An exception prints too much information on the console and the interesting part scrolls out of the window. How can I find out where the exception happened?
The best way is to redirect the error stream to a log file. To do this add the following line to your program:

```
System.setErr(new PrintStream(new FileOutputStream("logfile.txt")));
```

Like this all exception messages will be written to the file `logfile.txt` in the root directory of the iPAQ.

Q: My program just stops working but does not terminate either, what now?
Remove all programs from the memory and restart your program and/or remove `-Djeode.evm.console.local.paging=TRUE` from your programm call!
During the development of our application we faced the problem that the program just stopped running at a certain point in the code. It did not crash and terminate but just didn't continue its execution any more. Since the problem happened at a place where a programming error was impossible (setting the text of a label) we started to examine other possible reasons. Finally we found out, that as soon as `Active Sync` was loaded into the memory of the iPAQ, our application failed (even if `Active Sync` was inactive). Removing the synchronisation tool from the memory (`System->Settings->Memory->Applications`) solved the problem. Also the Jeode console parameter `-Djeode.evm.console.local.paging=TRUE` produced a similar

problem for a while.
With the final version of the Instant Messenger the error cannot be reproduced any more, and it is unknown what caused it in the first place.

Q: Why can't my application find its data files?
You are probably trying to use a relative path to your file. Jeode always uses the iPAQ root menu as its working directory, which means all paths need to be written in their absolute form (e.g. `\myProgramFolder\dataFiles\data.dat`)

Bibliography

[1] David Mayor: *Comparaison de Machines Virtuelles Java pour PDA*. Semester Thesis EPFL (2002)
`http://lsrwww.epfl.ch/cavin/work/manet/mayor02.pdf`

[2] Driver and configuration tool for the Cisco Aironet 350 WLAN adapter
`http://www.cisco.com/pcgi-bin/Software/WLAN/wlplanner.cgi`

[3] Personal Java emulator
`http://java.sun.com/products/personaljava/pj-emulation.html`

[4] Download for Java Swing 1.1.1
`http://java.sun.com/products/jfc/download.archive.html`

[5] Thinlet awt extension homepage
`http://www.thinlet.com`