Java on a Linux Mobile Phone - Sharing a Virtual Machine

Torbjörn Andersson, E03
e03ta@student.lth.se

Lunds Tekniska Högskola
June 3, 2009

Abstract

This report investigates aspects of running Java SE on mobile Linux devices, with special focus on situations where many applications can be expected to run simultaneously. A multi-tasking Java environment where independent applications execute in a shared virtual machine is proposed and technologies for achieving this are investigated. A prototype system, designed entirely in Java as a middle layer between the Java Virtual Machine and the applications, is described. The main concept used in the prototype, classloader-based isolation, is described in detail along with the problems associated with it and possible solutions for these problems. It is shown by the prototype that major savings of memory footprint can be achieved by using it in cases where multiple Java applications are run simultaneously. Improvements of application startup time was not achieved by the prototype, but the possibility of such improvements is investigated in the report. Also, the potential of increased performance when using inter-process communication techniques within a virtual machine compared to communication between operating system processes is described.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Java technology has gained much success in the area of mobile phone software. It it especially popular for developing games and other entertainment applications, and for applications that can be downloaded from the cellular network. The most common utilities in mobile phones, such as phone books and messaging applications, are however often created as C code, which is compiled for the specific platform. Until now, Java applications for mobile devices have been using Java ME, which is a scaled-down version of Java, suited for the simpler and slower hardware that mobile devices have compared to workstation PCs. Java ME is, due to its limited functionality, not suited for applications requiring better access to hardware or system software interfaces.

Recently, some different initiatives have announced mobile platforms that claim to use Java as the primary language for implementing all applications in the system. These initiatives describe more advanced Java solutions than Java ME. It is also clear that the processing power, memory capacity etc of mobile devices have increased to levels making it possible, at least on high-end devices, to run more advanced Java solutions such as Java SE. This report studies the possibilities for running Java SE on Linux-based smartphones, with focus on reducing processing load, memory footprint and optimizing other aspects of a system in cases where multiple applications written in Java are used simultaneously. These aspects are especially important in the area of high-end mobile devices, smartphones, where multiple small applications should be able to run simultaneously in an environment where system resources should be saved as much as possible and where the processing capacity is low compared to normal workstation PCs.

1.1 The problem

Even though modern high-end mobile phones have the hardware capacity to run Java SE applications, the performance is quite low. A typical smartphone would only have the capacity to run a small number of Java SE applications simultaneously, and even then with quite bad performance. Also, on mobile devices, hardware resources should be treated as scarce resources, since efforts should be done to hold power consumption, weight and production cost of the devices down. The current situation could be acceptable if the phone only runs a single Java application at the time, such as many feature phones supporting Java ME programs do today. However, for example if the applications that a user expects to find in a mobile phone, such as phonebooks, messaging clients etc, were also written in Java and run as stand-alone applications, there would be a need to run many Java applications simultaneously and the system would quickly become overloaded. The overload is due to the fact that all Java applications carry an additional overhead to the specific application code. Java applications need a Java Virtual Machine to run, and this machine has memory requirements, requires time to start and so on. Especially when running small Java applications, the overhead due to the virtual machine is significant. Removing the virtual machine would open up for big increases in Java performance. There are techniques for compiling Java code into native executables. However, doing so would remove some Java features that are desired, such as platform independence, which makes this an unattractive solution. There is a need for a virtual machine running in the system to provide proper support for Java. There is however nothing in this stating that each application really must have its own virtual machine, which means that resources could be saved if all Java applications instead shared a common instance of a virtual machine.

1.2 The idea

A potential way of saving system resources and increasing performance when running Java is to run all applications in a single Java Virtual Machine (JVM). Doing so saves memory, since only one Java Virtual Machine would have to be started, even when there are multiple Java applications running. Depending on the way this would be implemented, this could also be true for standard library classes and even application classes. Also, initializing and loading the virtual machine takes time and processing power, which also would be possible to save. Another aspect is that when applications need to communicate with eachother, running them in the same JVM

gives a possibility of optimizing the communication between them, avoiding costly context switches between operating system processes and calls to the operating system kernel. These aspects are of course mostly relevant only when running multiple applications at the same time, although a single application of course could benefit from a JVM that is already running in terms of startup time. However, it is likely that multiple applications are run in parallel in a mobile phone, and part of the intention of the project is to make it possible to use Java implementations of these applications without creating unnecessary overhead.

Running multiple applications in a single operating system process is affected by one big problem. That is that all the application will share a single address space, and the operating system will not be able to protect the applications from eachother. This means that the system controlling the launching of new applications within the process should implement some sort of system for such protection. The Java language has a special property in that it is a type-safe language. This means that the data types of objects are checked before operations are performed on them and that objects can not be cast to different types without being compatible with that type. This also includes that it is not possible, for example, to create a pointer pointing to a memory address and typecast it to an object, thus accessing an object that might be located on that address. There is actually no possibility at all to create Java code that directly modifies memory addresses on the heap. The result is that references to objects cannot be created in other ways than copying the original reference to the object. When this property is present, running multiple applications in a single process becomes possible without the risk of the applications accessing eachothers data, without needing an explicit system to control the memory area. This means that a Java runtime has the potential of running multiple applications in one virtual machine instance, with increased performance compared to the ordinary model of one virtual machine per application. There are different ways this can be accomplished, and different project that have tried these ways. The various solutions are investigated further in this report, and one of these is selected as a base for a prototype system described in the report.

Chapter 2

Background

The purpose of this chapter is to provide some background to the project, motivating why research in this area is interesting. Using Linux as an operating system for a mobile phone is a trend that has gained a lot of attention lately. There are many different projects focusing on this, and many of the big mobile phone manufacturers have expressed interest in Linux as a mobile phone operating system. In this chapter some of the more interesting projects in this area are presented. Java technology, on the other hand, has for quite some time been common in the mobile world. Java ME is a very common technique that most mobile phones produced today support. Recently, however, some projects that intend to deliver more extensive Java solutions for mobile units have emerged and gained a lot of attention. These are also described in this chapter. Pure Java SE has not gained any particular attention in the mobile world until now, much due to high system requirements which mobile phones previously have not been able to meet. However, modern smartphones nowadays have computing powers on par with what personal computers had when Java was first released, therefore it might seem as a natural choice to use Java SE on modern smartphones. Open mobile Linux platforms such as the OpenMoko platform provide a good environment for creating and experimenting with a Java SE enabled smartphone. The OpenMoko telephone and software platform, as well as the Java Runtime Environment currently available for it, will also be presented in this chapter.

Even if the computing power of mobile phones has increased to the levels needed to run Java SE applications, the system resources are still not more than barely enough for this. Especially if multiple applications are running at the same time, this imposes a very high load on such systems. Also, since resources are more valuable on a mobile system, much due to the limited battery power, it is more important than on a normal workstation computer to hold the consumption of system resources down. The idea

presented in section 1.2 could significantly reduce memory requirements as well as processing time when running multiple Java application simultaneously, and the idea is therefore worth investigating further. Previous research in that specific areas is however presented in chapter 3, "Related Work".

2.1 Java

The Java programming language was released in 1995 by Sun Microsystems as an object oriented programming language. The Java language is part of a Java platform including not only the language in itself, but also the Java Runtime Environment, including a Virtual Machine and a core class library, development resources such as compilers and debuggers as well as the Java Bytecode format for storing half-compiled Java code in a platform independent manner. The Java Virtual Machine is a software that takes Java bytecode representing a program and compiles and runs this at the same time. The Java bytecode format is platform independent, and will run on any system for which there is a Java Virtual Machine. The standard class library is also the same on all platforms, and is mostly implemented in Java as well.

The Java language has a syntax that reminds of C and C++, but there are also some differences. In Java, the programmer is not allowed to handle memory management explicitly. Instead, all objects are automatically allocated on the heap when they are created, and then deallocated automatically when there are no more references to them. Also, there is no pointer arithmetic or possibility to create a reference to an object without copying a reference that is already accessible. Objects cannot be cast to a different type if they are not compatible with that type according to the built in inheritance system of the Java language. These factors all contribute in making Java a type-safe language. These properties are maintained when the source code is compiled into Java bytecode, and checked at runtime, which means that these properties will be checked even when running bytecode compiled from another language or handcrafted bytecode. The property of type safety is absolutely necessary for making the goal of this project achievable. Since this project aims at creating a system that runs multiple, not necessarily trusted applications in a single address space, any code that can have direct access to the memory area would be able to access and modify all information in all applications running in the system. For example, such a system would not work for programs written in C. This is also the reason why it is problematic to allow native code in such a system, since

this code would be able to access data belonging to other applications in the system.

2.2 Java on mobile devices

Java has gained a lot of popularity on mobile devices. This has happened due to the Java Platform, Micro Edition (Java ME, formerly known as J2ME). Java ME is a Java Platform aimed at mobile and embedded devices. It consists of a subset of Java, containing only those APIs that are useful and permittable on the target platform. There are different flavors of Java ME which aim at different categories of devices, such as smartphones, simpler mobile phones and embedded system with no or limited user interface. Java ME is used on mobile phones to create games and other applications for the devices. Especially using Java ME for games has gained much popularity. There are today a great number of devices, especially mobile phones, that have the capability of running Java ME applications.

As mobile devices such as mobile phones have gained more computing power, these platforms have become capable of running more advanced Java versions and software. Java ME has been released in more advanced versions than the original version, and features have also been added by mobile phone manufacturers. During this process, Java ME has become more and more similar to Java SE. Some recent initiatives have however showed that the trend goes towards replacing Java ME with something else, more similar to Java SE in the functionality. Sun Microsystems have, for example, announced the new system JavaFX Mobile as a part of their new JavaFX line of products. JavaFX Mobile is an operating system based on a Linux kernel, and is aimed at running applications written entirely in Java. The API:s available are similar to Java SE, but backwards compatibility to let Java ME applications run on the system is also an important part. JavaFX Mobile is based on products from SavaJe Technologies, which was purchased by Sun in 2007. According to [1], at least the SavaJe OS, included in the technology purchased from SavaJe Technologies, has support for running multiple Java applications in a single JVM instance, as attempted by the project described in this report. JavaFX Mobile has yet not been released for use in any commercial products.

Another initiative in the area of providing more advanced Java solutions to the mobile world is the Android operating system. It was created by the Open Handset Alliance, a business alliance with Google being the most notable member. The Android platform is based on a Linux kernel and uses Java as the programming language for applications. A Software Development Kit for the system has been released, but there are no phones

using the system commercially available yet. It is worth noting, however, that the Android platform does not run Java in the ordinary fashion. The Java syntax is used for the programming language, but the APIs available are not compatible with the Java SE or ME APIs. Also, the code is compiled to a different bytecode format, which is run by the Dalvik Virtual Machine included in the system.

2.3 Mobile Linux platforms

As seen previously, Linux has gained some attention in the mobile segment recently. Products such as JavaFX Mobile and Android use a Linux kernel as a base for their respective mobile phone operating system. However, there are also some other initiatives in the area that are not focused on Java as the primary programming language. For example, the LiMo foundation has created a platform for using Linux on mobile devices. Also, the Norwegian company Trolltech, recently acquired by Nokia, has developed Qtopia, which is an application platform for mobile devices using Linux. Motorola is an example of a company that has released a long line of mobile phones based on Linux, but such phones are also manufactured by a number of other well-known companies. There are also projects using Linux with the purpose of creating a mobile platform that is as open as possible to external developers, such as the OpenMoko project. That specific project is closely related to the Neo line of mobile phones from First International Computer (FIC), and the project is presented in more detail in section 2.4.1. The Apple iPhone is not based on Linux, but rather reuses components from Mac OS X. This means that this also is a device using an operating system originally created for a workstation computer, which then has been transferred to the mobile device.

2.4 The reference platform

This section describes the platform used as a target platform for the prototype system described in chapter 5. First the OpenMoko platform is described, and then the Java Runtime Environment, consisting of the Cacao JVM and GNU ClassPath, that is available for the platform. It is however important to understand that the prototype is not especially tied to this platform but can run on many different platforms with little or no modification. The technologies used are not tied to Linux or OpenMoko. Parts of the technology is tied to GNU ClassPath, which is the Java Standard Library that has been used, although it can easily be ported to any standard library where the source code is open and can be modified and recompiled.

Also, some problems still remaining in the prototype are due to problems in the Cacao JVM. Except for this, the prototype does not depend on the Java Runtime Environment either, as long as it is Java SE compatible. The technologies used in the project could be applied to platforms running other operating systems as well, even though testing of such functionality has not been included in the project. The D-bus Inter-process Communications system, which is referenced in different places of this report, is however a Linux-centered product that is not normally used on other platforms. On the other hand, it is not included in the prototype described in this report but rather used as an example of an external IPC system for communication with other processes running on the platform.

During development, a Linux-based PC has been used. To provide an environment as similar as possible to that of the OpenMoko phone, the same Java Runtime Environment, including the same versions[1] of the software, has been used. Testing has been performed on both platforms during all phases of the work.

2.4.1 The OpenMoko Linux Mobile Phone

OpenMoko [2] is a project attempting to create a mobile phone software stack entirely under an Open Source license. It uses a Linux kernel and other well-known tools that turns the system into a complete Linux distribution, in much the same fashion as Linux distributions targeting workstation PCs. OpenMoko is currently run as a small company, maintaining the project, with the support of many open source developers participating in the project. The project is tightly connected to the Neo telephone series, developed by First International Computer, FIC. The first telephone in the series was the Neo1973 prototype telephone, which is a smartphone with touchscreen and no keyboard, and is equipped with an ARM-processor at 266 MHz, 64 MB Flash memory and 128 MB of RAM, as well as support for GSM networks, Bluetooth and various other technologies. It will be followed by a new model, called the Neo FreeRunner, which, in addition to upgrades to the current hardware specifications, also comes with Wireless LAN Connection and a GPS system. The OpenMoko software stack is primarily focused towards this platform, but is also intended to be possible to use on other devices. Similarly, the Neo devices could also be used with other software stacks than OpenMoko. The OpenMoko platform and the Neo telephones are excellent for use as platforms for development of new software targeting Linux-based mobile devices, since it is an open environment with few restrictions to what users are allowed to do with the

[1] Of course, the versions used on the PC and on the phone are compiled for different processor architectures

hardware and software. These properties made the OpenMoko platform running on the Neo1973 mobile phone a natural choice for a target platform during the project described in this report.

2.4.2 The Java Runtime Environment

Jalimo

The Jalimo project [3] attempts to create a free runtime environment for Java applications to be used on mobile Linux-based devices. It targets a couple of different platforms, of which OpenMoko is one. Jalimo consists of a collection of open source software packages which implement different parts of the Java Runtime Environment, as well as some other related products. These products are packaged together in a common distribution system and tailored to work with the targeted platforms. Jalimo contains the two virtual machines Cacao JVM and JamVM, of which the former uses Just-In-Time compilation and the latter is interpreter-based. It also contains the GNU ClassPath library, which together with a JVM becomes a working Java Runtime Environment. Also included in the project are some packages providing different components for graphical user interfaces, as well as other functionality, such as bindings to the D-bus Inter-Process Communications system. The Jalimo project is not locked to these software packages however, components may be changed when other products become available for the target platforms.

Cacao JVM

The Cacao JVM [4] is a Java Virtual Machine, that uses Just-In-Time compilation, which means that the bytecode is compiled into machine-specific code the first time that every piece of code is run. Earlier virtual machines used interpreting, which means evaluating and executing each line of bytecode every time it is run, without caching the translation to machine code between the times the code is run. Cacao was originally created in 1997 as a research project, but in 2004 the project was released under a GPL license, and is now developed further as an open source project. It is available for a couple of different processor architectures, including the ARM processor, which is the reason for it being included in the Jalimo project. The JVM is also available for Intel x86-architectures, which makes it possible to run it on ordinary workstation PC:s as well. In addition to Linux, it can also be run on some other operating systems, such as FreeBSD and Mac OS X. Cacao is designed to use GNU ClassPath as its standard class library.

GNU ClassPath

GNU ClassPath [5] is an open source standard class library that is used by many Java Virtual Machine implementations. For example, both virtual machines included in the Jalimo project, Cacao JVM and JamVM, use GNU ClassPath as their core class library. As will be shown later in this report, using an open source class library was necessary for the project described in this report. This is because the prototype system (described in chapter 5) required changes to one of the core java classes to work properly, which would not have been possible if the source code for those classes was not available.

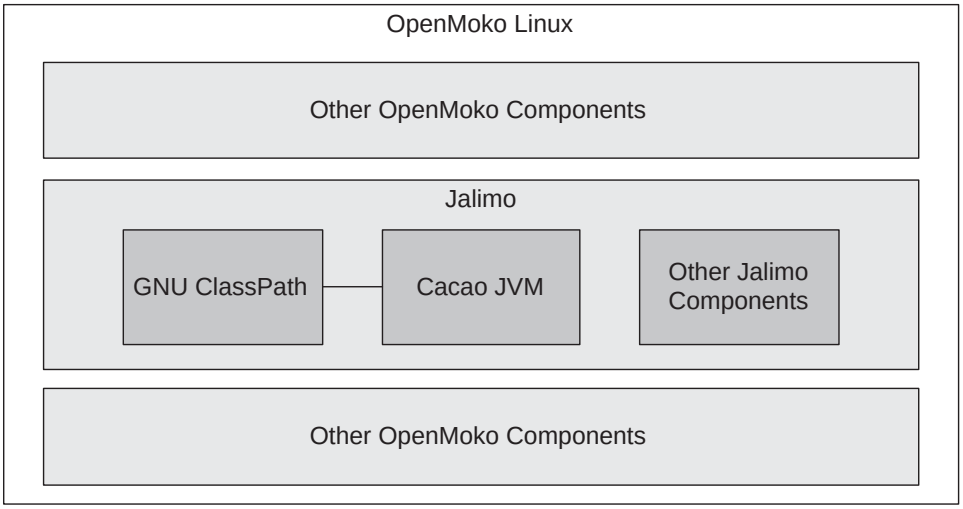

Figure 2.1: The relationship between the platform components

Chapter 3

Related work

In this chapter, different projects that have made attempts at implementing isolation between application in a single Java Virtual Machine are presented, as well as some different techniques that could be used to accomplish this isolation. Some of these products were chosen for use as a base for the prototype system described in this report, and therefore are described in greater detail later on in the report. These are, however, also presented here, so that comparisons can be made with the other techniques that could be used to solve the problem.

3.1 The JKernel System

The JKernel [6, 7] is a system for isolation between multiple applications running in a single instance of a Java Virtual Machine. It was developed at Cornell University in 1998, and has not been subject to any further development since then. JKernel is implemented entirely in Java, and is therefore compatible with any platform that has a Java SE compatible Java Runtime Environment. The JKernel is based around the use of classloaders (described in detail in chapter 4) to provide separation between applications, or Tasks, as they are called in JKernel. The method is fairly simple to understand and implement, but there are some problems that it does not address. The JKernel compensates for some of the problems with isolation using classloaders by complementing this with extra functionality, such as replacing some selected classes from the standard library with wrappers with extended functionality. However, there are still some aspects that can not be addressed when using this method, which means that the JKernel is not necessarily a good solution for all potential uses of shared Java Virtual Machines. Since the JKernel was selected as a base for the prototype described in this report, there is a special chapter presenting this in more

detail. (See chapter 4.) It is presented briefly here only to provide the reader with some basic knowledge about this system before presenting alternative ways the preferred isolation could be achieved.

3.2 Other Classloader-based solutions

The JKernel is not the only product that uses classloaders to provide isolation between applications, processes or similar abstractions. In fact, this is a technique that is commonly used in many systems that need to provide this kind of safety or isolation between different part of code, in various applications. ([8] chapter 14 gives example of this.) Most systems using this technique, however, are different kinds of application servers or other products aimed at larger computer systems. There are some systems based on this technique that are intended for normal desktop use, to save system resources and startup time when using several java-based utilities at the same time. (Example: Echidna, see section 3.2.1.) However, probably due to the fact that mobile units only recently have achieved the computing power to run more advanced Java systems, as well as the fact that most operating systems for mobile devices have been locked down by manufacturers, little development of systems providing isolation through class loaders have been seen aimed at mobile devices[1]. The use of Java in a mobile device has however some similarities to application servers, in the sense that they both suffer from the same lack of resources. In a server system, resources consumed by a single application should be reduced, to provide as many users as possible simultaneous service. In a mobile device, resource requirements should be reduced due to low hardware prestanda, as well as to save battery power. When running multiple applications on the mobile device, as users might be expected to do if given the possibility, this situation is actually quite similar to that of an application server, in the sense that reducing the resource requirements for the entire software system is very valuable, thus motivating the use of the same techniques. Depending on the intended use of the product, the different softwares using the class loader technique implement different sets of features on top of the class loader isolation. JKernel, being originally aimed at an application server environment with great risk of interception with malicious code or users, has put effort into some extra protection, while others, such as the Echidna project, have focused less on protection.

[1] Also, Java ME does not support class loaders

3.2.1 Echidna

Echidna [9, 10] by Luke Gorrie is a software aimed at single-user workstation environments. It provides means to run different applications in a single JVM, and uses class loader based separation for the internal implementation. The purpose of the software is to let the user run various java utilities in the environment, to save system resources that might otherwise have loaded the system too much, and giving the user full control of which applications are started. Since the system trusts the user not to start any malicious programs, there are no security measures in the system, except for the class loader system which is needed to make the system work at all. There is also no built in system for Inter-Process Communication, and no possibility to forbid the use of certain classes from the standard library or otherwise. Due to the complete lack of security features, the Echidna system was not a good candidate to base a prototype system on. However, the system might still be interesting to study, since it can give a good hint on the upper bound of the performance that can be gained by using this method.

3.2.2 IBM Long Running Environment

A US patent [11] filed by IBM in 2000 and issued by the patent authorities in 2005 describes a solution that shows some similarities to the JKernel system both concerning its purpose and the technical solution. The technology described in the patent is implemented in the "Long Running Environment", LRE, which is part of certain software in IBM's WebSphere product line. In a similar fashion as the JKernel, LRE provides the possibility to run several different applications in a single instance of a Java Virtual Machine. The system is, just as JKernel, implemented on top of the existing Java Runtime Environments, thus being compatible with all standard environments. Both products use classloaders to create separation between applications running in the system. Both systems also bring in some extra protection features against such problems that the classloader separation cannot fix, such as calls to System.exit() and modifications to the system for the standard output stream. Both systems also analyses and modifies bytecode internally. This means that, at a first glance, these systems look very similar. The LRE system also keeps track of user interface objects such as windows, to be able to terminate these when terminating an application running in the system. Those features were missing in the JKernel system, but have been added in the prototype described in chapter 5.

There are however some important differences between the LRE and the JKernel. Both use classloaders for the basic separation between appli-

cations, which must be considered as an established method for doing this. The major differences instead lie in the way the extra protection is performed. LRE uses bytecode rewriting techniques to remove calls to, for example, the System.exit() method and replace these calls with callbacks to terminate the application doing the call. In a similar way, code is added to register created windows and associate these with a running application, so that the central system can terminate these windows when termination an application. In the JKernel, such tasks instead are performed through wrapping of the affected classes, and using the classloader system to present user applications with the wrapper classes instead of the original ones. Windows were not registered from the beginning, but the implementation introduced with the prototype described in this report (described in chapter 5) replaces the original affected standard library classes entirely with a compatible version. JKernel also has bytecode rewriting features, but these are used for entirely different purposes.

It should be noted that the JKernel was presented in 1998, which means that the technologies present in both these products should be considered as prior art, since the IBM patent was filed in the year 2000, at least as long as the technologies are used for the same purpose in both products. The described functionality of using bytecode rewrital to implement protection against troublesome methods in the standard class library is never used either in the original implementation of the JKernel or in the prototype described in this report. The technology could be interesting for that prototype, especially as there are bytecode rewriting facilities in the JKernel, but since bytecode rewrital is slow in the JKernel and currently not bringing very much functionality, the target should instead be to remove this. Doing that, the technology would not be very interesting for the prototype any longer. Also, since it is a patented technique, a permission would have to be retrieved from IBM before using this technology.

3.3 Using a modified Virtual Machine

As stated earlier, using class loaders to provide isolation between different applications, or tasks, running in the same Java virtual machine is a solution that is simple and platform independent, but unfortunately contains a few flaws that are not easy to fix. The basic problem here is that the standard library classes in Java must be loaded by the built in system class loader, which cannot be replaced or instantiated in multiple instances. To make that possible, the virtual machine would have to be modified, which has been done in some project. By modifying the virtual machine, or implementing a new virtual machine, it is possible to bypass the problem that

the class loader approach suffered from. The main drawbacks are however that this results in a system that only works on the platforms to which the virtual machine has been ported, as well as the fact that virtual machine internals tend to be much more complex than the java code required to implement a class loader-based solution. When doing this kind of modifications to a virtual machine, it would be preferred if the modifications followed some sort of standard to provide compatibility between virtual machines from different suppliers. This section describes the JSR 121 specification request, which provides an API suitable for application isolation, as well as some products based on it.

3.3.1 JSR 121

JSR 121 [12] has been implemented in a few different products. However, for various reasons, none of the studied products was considered suitable for use in this project. The JSR 121 has made its way into some Java ME implementations. The products based on Java SE are on the other hand still quite immature, and there are currently no released products implementing the JSR 121 for Java SE.

The JSR 121 Java Specification Request is a request for a unified API for products that handle multiple Java applications running simultaneously. The basic requirement on any product implementing this API is such that the isolation between the different applications would be sufficient for the task this project is focused on. However, the JSR does not specify how this should be implemented, which means that it can be implemented by, for example, running the applications in different virtual machines. Such an implementation, while providing isolation between application, would provide this project with little of value, since no performance gains would come from such a solution. There are, however, some research projects that have resulted in prototype versions of Java Virtual Machines implementing the API specified by the JSR 121 in a solution where the applications run in a single JVM instance. These were studied within this project, but in the end a classloader-based solution was selected for the prototype presented later in this report. A classloader-based solution could however also be made to comply with the JSR 121 API, provided that a way could be found to establish good enough isolation between the applications running in the system. No software implementing the JSR 121 for Java SE has been released in a final version, those described below being prototypes or in beta stages. There are however Java ME implementations that use technique based on this JSR.

3.3.2 Sun MVM

The Multi-tasking Virtual Machine [13, 14], MVM, from Sun is a Java Virtual Machine that claims to deliver proper isolation between applications running in the same virtual machine. The MVM implements the Isolate API defined by the JSR-121. The flaws that can be seen in classloader-based solutions are not present in this solution, and at the same time, the overhead created when using one virtual machine per application is not there either. This is accomplished by using a modified JVM. MVM is also constructed in such a way that all of the functionality delivered by the standard library is available without restrictions to applications running in the system. Also, native code can be used in the system, by securely running it as an isolated operating system process. The MVM is however still in its prototype stages, and the prototype is only available for Solaris/SPARC environments. Porting it to new platforms would mean lots of work, work which would have to be repeated for each different target platform. Using the MVM for isolation creates a system that cannot be ported to another, ordinary, JVM, which might be preferred when targeting platforms other than those which Sun decides to port the MVM to. The source code for MVM is available to work with, but under a more restrictive license than most open source software, so porting the MVM to new platforms is not necessarily an interesting alternative either. For these reasons, it was decided not to use the MVM as a base for the prototype presented in chapter 5. Also, the MVM does not include a system for IPC, which means that such functionality must be implemented using traditional IPC methods, thus not providing the faster IPC mechanisms that could be created when all applications run in the same shared memory area.

3.3.3 KaffeOS/JanosVM

KaffeOS

KaffeOS [15] is a product based on the Kaffe Virtual Machine, which has been modified to support multiple simultaneous applications. It implements an abstraction of operating system processes within the JVM, and also reuses many concepts from normal operating systems. KaffeOS has been focused on resource accountability and possibility to terminate processes cleanly. For example, the system has separate memory heaps for each process. Internally, it uses classloaders to separate processes from eachother, in a similar fashion as JKernel and Echidna. Some standard

library classes are loaded by the classloader associated with each process in the JVM, while others are shared between all processes. Loading standard library classes this way would not have been possible without using a modified Java Virtual Machine. The Kaffe Virtual Machine, which is the base for KaffeOS, is also quite slow compared to other virtual machines.

JanosVM

JanosVM [16, 17] is a Java Virtual Machine based on the KaffeOS. It is intended as a base package to construct a multitasking Java environment, to make it possible to create such systems without having to create or modify a JVM. One addition that has been made to the KaffeOS system is that JanosVM supports the Isolate API defined by the JSR 121. Also, JanosVM is constructed to give possibility to IPC functionality within the JVM, although no API aimed towards application programmers has been created for this.

3.3.4 Some Java ME implementations

There are also Java ME implementations that support multitasking within a single JVM, which is performed by using a JVM that is specialized for this. For example, the CLDC HotSpot Implementation from Sun (from version 1.1.2) includes such support [18], and is based on the technology from the MVM virtual machine, presented in section 3.3.2. Multitasking in a single JVM in Java ME attempts to achieve the same performance gains as in Java SE. There is, however, sometimes also the problem of using an underlying operating system that does not support multitasking in itself, which means that running multiple JVMs sometimes is not possible. Using a classloader-based solution is not possible on a Java ME system either, since custom classloaders are prohibited in Java ME. Thus, the only possibility to run multiple java applications simultaneously is often to use a JVM including this feature.

3.4 Inter-Process communication systems

Inter-Process Communication, IPC, is a term that covers all communication between different processes in a computer system. There are different forms of inter-process communication, such as message passing, shared memory or remote procedure calls. These are techniques that require cooperation with the operating system, which sometimes can create unnecessary overhead. For example, with message-passing, the messages will in

some way or another pass through the operating system on their way between processes. When running multiple applications in a single Java Virtual Machine instance, IPC can be performed with less overhead between these application and without concern about the operating system, since these applications technically are part of the same Operating System process. IPC within a single JVM could potentially be reduced to the level of an ordinary function call, which is a very small operation compared to context switches between operating system processes the way IPC normally requires.

Since all the applications running in a single JVM instance actually share the same memory area, large data structures could potentially be sent between applications just by passing references, thus avoiding heavy copying operations. The shared memory area, in its turn, is possible due to Java being a type-safe language, as explained in section 2.1. The conclusion is that a system running several applications in a single JVM has the potential to great increases in IPC performance, compared to ordinary IPC system. That is the reason why this is studied as a topic in this report.

The JKernel, which has been used as the base for the prototype that is presented in this report, has a built-in IPC system called Capabilities, which draws advantage of the described properties to increase IPC performance, and which will be studied in the report. The Capabilities system, however, has no possibility of communicating outside the Java Virtual Machine, and it also has some limitations in its functionality. Therefore, another IPC system, called D-bus, will also be described. The prototype, described in chapter 5, currently has no D-bus support, but adding it to the system should be possible. Also, it might be possible to modify the D-bus system to run on top of the Capabilities system, so that applications running in the same instance of the prototype could use it and still benefit from the advantages of running in the same JVM.

3.4.1 D-bus

The D-bus [19] is a IPC system based on passing of messages. It acts as a message bus to which processes can sign up with an unique identifier. After that, messages can be sent through the D-bus system using that identifier, and the D-bus system will pass the messages on to the correct process. In D-bus, data in messages is treated as specific data types, and not just as a stream of bytes. Also, the system supports sending a message and getting an associated answer back. The structure of messages and replies are defined by interfaces, in a fashion much like the interface concept in java. That way, D-bus can be viewed as a system for performing Remote Procedure Calls, that is, sending a call with a set of data as parameters, and then

getting a reply back with another set of data, corresponding to a return value.

D-bus is a product centered around Linux and Unix environments, and is currently not available for e. g. Windows environments. Recently, the team behind the OpenMoko mobile Linux platform has decided to use this as the primary IPC system for the OpenMoko platform. The system is also used by a variety of programs for the Linux platform. This makes the D-bus especially interesting to study when developing software targeting mobile Linux platforms. Since there is a Java interface for D-bus, it might be interesting to study also in this project, to see if it is a suitable platform to use to interconnect a JVM running multiple applications with native software or java applications running in their own JVM instances.

Java D-bus

There is a java port [20] of the D-bus system, which should be possible to use together with the Linux D-bus system. This java port is of course not designed for use in a JVM that is shared between multiple application, but rather for communication between applications running in separate JVM instances. Communication can currently be performed using either TCP/IP or Unix sockets. This means that the potential performance boost to IPC that a shared JVM could bring is not achieved using this IPC system. However, the system could give the applications in the shared JVM a possibility to communicate with other applications. Also, with the way the D-bus system works, it might be evaluated if an add-on to that system could be implemented, providing a message switching functionality so that messages that should be passed between applications running in the same JVM would not need to be passed through the operating system.

3.4.2 JKernel Capabilities

The JKernel, described in section 3.1, contains a built-in IPC system called Capabilities [6]. It only works between tasks running in the same JKernel instance, but it shows the potential performance gains when running inter-process communication within a single instance of a Java Virtual Machine. The JKernel Capabilities provides the possibilities to share objects between tasks in a safe way and to do method calls across task boundaries. Since the tasks are all running in the same operating system process, there is no need for communication between them to go through the operating system kernel as inter-process communication normally does. Instead communication between tasks does not take very much more resources into account than ordinary method calls within applications, since it is performed inter-

nally in the JKernel system as method calls. Also, since all the tasks run in the same physical address space, with isolation provided through the safe language, data can be sent between tasks by just giving access to references. This reduces the need to copy large amounts of data that might be necessary to send between tasks. This IPC system is interesting to study and to compare with other IPC solutions, due to the potential performance gains it brings over conventional IPC systems. Capabilities are described in more detail in section 4.3 in the next chapter.

Chapter 4

The JKernel - classloader-based isolation

The JKernel system [6, 7], described briefly in section 3.1, was created at Cornell University in 1998 and was originally intended for use in application server-like environments. When studied, it became apparent that the JKernel could be useful as a base for a prototype system providing isolation between Java applications also on a mobile platform, since it contains most of the technology that would be needed in the prototype. Compared to other products using similar technology, such as Echidna (described in section 3.2.1), JKernel is a quite large system, containing many function that extend the functionality of the system. Some of these were considered necessary, or at least useful, in the prototype, thus ruling out simpler products such as Echidna. It might however be worth noting that some of the features in the JKernel might be of less use in this application, and instead reducing performance of the system for no good use. On the other hand, JKernel is released as open source under a BSD-like license, permitting anyone to do most about anything with the code, which means not only that the prototype could be used for anything without the explicit permission from the original authors, but also that the structure could be reused for creating an implementation including only the features needed and nothing more.

From this background, the JKernel was selected as a suitable platform for the prototype system. For this reason, the JKernel is presented here with more detail than given about the different topics in chapter 3, "Related Work". The most important techniques used in the JKernel are also described in this chapter. This chapter focuses on the original distribution of the JKernel, although on some occasions references are made to the extensions to the JKernel that are described in chapter 5, "The prototype

system". References to changes to the JKernel system introduced with the prototype are clearly marked as such, and they are only introduced when describing issues that are not properly addressed by the original JKernel implementation.

4.1 Isolation in JKernel

This section describes the techniques used for isolation between tasks in JKernel. The primary means of isolation is the use of classloaders, which is described in detail in section 4.1.1, "Isolation with classloaders". Isolation using classloaders unfortunately has some drawbacks, and to fill in a few of the holes left open, the JKernel has some extra features, which are described in section 4.1.3, "Extra isolation features in the System class". Even with these features, though, there are still some flaws left which the JKernel system will not be able to handle, and these are described in more detail in the section 4.1.2, "Isolation flaws".

4.1.1 Isolation using classloaders

Using classloaders is a fairly common technique for achieving isolation between applications in a singe JVM. The classloader is the part of the java system that loads new classes into the system. The classloader that is used can be replaced, and different classloaders can be used at the same time to load different classes. In Java, a class is uniquely identified at runtime by the combination of its name and the classloader that was used to load it, even though only the class name is visible in the java source code. This means that if you would want to run different applications in a single JVM, then you could assign a different classloader[1] to each application and load all the application with this, and the applications would then never share any class even if they both would use a class with the same name that was originated from the same class file. The need for this separation between different applications is most clearly visible when looking at static fields and methods in a class. By separating the classes used by different applications, a statically defined field would be accessible from everywhere inside an application, but another application running in the same JVM and using the same class (as defined in the source code) would see a different instance of the static field. This is due to the fact that at runtime, these applications actually do not use the same class, but rather different classes with the same name, which was accomplished by the use of multiple classloaders. The

[1] A different classloader object is sufficient, the classloaders do not have to be of different classes.

source code below demonstrates the problem that can arise when multiple applications use the same class containing static fields. Two applications both using the class MyClass will read and overwrite eachother's value of the static variable x. When loading this class with two different classloaders, both applications can have their own instance of the variable x.

```
public class MyClass
{
    private static int x;

    public void putX(int i){
        x = i;
    }

    public int getX(){
        return x;
    }
}
```

4.1.2 Isolation flaws

Worth noting about classloaders is that using a custom classloader is only permitted for user classes. Standard library classes should be loaded by the system class loader, which cannot be replaced, and thus, static fields and methods in these classes will be shared by all applications in the JVM. Trying to go around this by locating the standard class library on disk and loading the standard classes from their original bytecode as if they were user classes will not work, since many classes actually only can exist in one version due to connections with the virtual machine and the outside world through native code. Also, as can be seen later on in section 4.2 concerning resolvers, the JKernel uses eager class loading to recursively load all user classes before starting an application. Applied to the standard library classes, this would lead to recursively loading major parts of the library before the application can start, which would completely waste all resource savings that the project has the potential to achieve. Disabling eager loading could technically be done, but it is not recommended since usage of forbidden classes would not be discovered until after an application has started. Therefore, the sharing of static fields and methods in the standard library classes has to be accepted when using this method.

The fact that static fields are shared between standard library classes used in different Tasks is one of the major problems with the JKernel and similar solutions. However, it can be debated whether this is a problem that has a major impact on the usability of the system or not. Many static fields or function in the standard library classes have characteristics that make

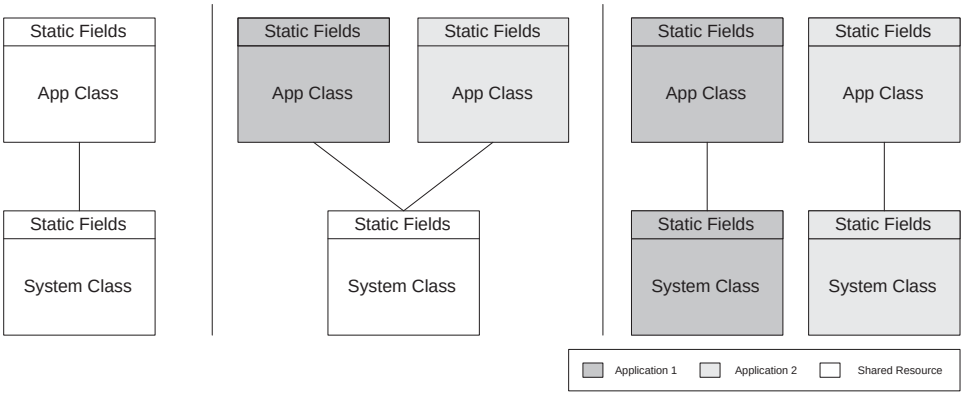

Figure 4.1: Shared resources with different levels of isolation. In the leftmost figure, processes run without isolation and share all static fields. In the middle, a solution where application classes but not system classes are isolated is shown. The rightmost figure shows the preferred case where all classes are isolated between applications, which however is difficult to achieve using classloaders.

them harmless from an isolation perspective. Many times these static fields only provide information that can safely be shared between Tasks, such as system time etc, or they might be used to provide access to system resources that per definition can exist only in one instance. Static fields are not commonly used as a communications channel, as to provide a link between different classes that for some reason cannot share access to the same references. Such usage would be an example of a use of static fields that would compromise isolation security and functionality when running in the JKernel. During the work in this project, some special places in the standard library classes have been identified where there are particular problems due to shared static fields. For example, the System.out, System.in and System.err fields are stored as static fields in the System class. As a workaround for this, the JKernel has a replacement system class that can be used with a custom classloader, thus providing the possibility of creating different instances of these static fields. Also, the method System.exit() is a static method[2] that has the potential to affect other Tasks when called. However, since this method couldn't be allowed to run, even if it was not declared as static, it would be necessary to completely disable this functionality, and possibly replace it with code to terminate the calling Task instead, since that would logically correspond to terminating the JVM when running in a single-application environment. The specific modifications concerning the System class are presented in greater detail in section 4.1.3.

[2]Static methods are generally not a problem, unless there are also static fields that can be affected or accessed using them.

Another java feature that would have the possibility to disturb other running Tasks are so called modal dialogs. These are dialog windows that disable access to other windows in the application until the user has responded to the content of the modal dialog. These will block access also to windows created by other tasks, which is not what the user might expect. Currently, no system to solve this is implemented, either in the original JKernel or in the prototype described in chapter 5.

4.1.3 Extra isolation in the System class

The class java.lang.System contains some of the most common static fields and methods that are likely to create isolation problems in a classloader-separated system of applications. Therefore, the JKernel implements countermeasures concerning this specific class. The methods described in this section could also be used to prevent uses of other problematic methods or fields in other standard library classes, if such problems emerge.

As described above, the JKernel cannot load the standard library classes with a custom classloader. Therefore, static fields and methods in these classes will be shared between the tasks running in the system. Perhaps the most obvious place in which this creates problems is the System class. The fields System.out, System.err and System.in would all be shared between all running tasks, which would create trouble since the output from all tasks would end up in the same terminal, as well as giving the possibility for the tasks to send input to eachother and read eachother's output. To prevent this, the JKernel contains a replacement class for the System class, which can be loaded with a custom classloader. This replacement class is presented to the tasks by making the classloader return this class rather than the original System class. The replacement class can thereafter access the original System class and delegate function call to this if it is safe to do so. There is also a system to differentiate between tasks, thus letting some tasks perform dangerous operations like System.exit(), while preventing others from doing so.

One problem with this method for replacing the System class is that it relies on the JKernel classloader system to load the System class and not to let the system classloader do this. When a java class references another class, it will use its own classloader to resolve the other class. As seen previously, standard library classes are loaded with the system classloader. The result of this is that when a standard library class references the System class, it will use the system classloader to resolve it, which will lead to the original System class being resolved rather than the replacement version. This will mean that using for example the System.out field will not work, since it refers to the field "out" in the original System class, which not only

is not the same as in the tasks instance of the replacement System class, but also is a stream that cannot be used as the usual output stream, since it has been replaced by the JKernel to provide output and input streams that work through capabilities[3].

The problem when accessing the System class from within standard library classes becomes even more apparent when looking at a situation where a standard library class calls the System.exit() method. When this happens, a call to the original function, which will terminate the Java Virtual Machine, thus terminating all running tasks, will be made without any check of the tasks permission to do so at all. This situation is not at all hypothetical, but a realistic situation for example when using the method setDefaultCloseOperation in the class JFrame, thereby configuring the JFrame to call System.exit() when it is closed. To prevent this, the prototype described in chapter 5 uses a SecurityManager, described in section 5.2.5. The original JKernel system did not address this issue at all.

4.1.4 Conclusion on classloader-based isolation

Since only a small number of flaws created due to the lack of isolation within standard library classes have been discovered, and since these most often can be accounted for by extra functionality such as described in section 4.1.3, it seems as the JKernels usefulness is not heavily restricted by this problem. There has not been any complete or otherwise extensive review of the Java standard library classes included in this project, therefore no guarantees can be made that there are no other problems, solveable or not, created through isolation flaws. However, there are techniques such as replacing standard library classes that can be used to solve such problems, at least when there is a reasonably small amount of classes where the problems exist. If problems emerge when the problematic standard library class is accessed from other standard library classes, the replacement methods used for the System class will not work. In such cases, there still is the possibility to implement a solution in a way similar to the replacement techniques used for the Window class, described later on in section 5.2.3. No such techniques have however been used in the original implementation of the JKernel, and these also cannot be used when using a standard class library that for some reason cannot be modified.

[3]JKernel Capabilities are described in detail in section 4.3

4.2 Resolvers

The Classloader system in JKernel uses a system of so-called resolvers to determine which classes it is allowed to load. This functionality is needed to prevent the system for loading classes that might threaten the system security. A resolver is an object that can return bytecode or a class object for an already loaded class, and there are also resolvers that can filter other resolvers and thus regulate which classes can be used. This is used to regulate which user classes the applications running in JKernel are allowed to load, but also to regulate which standard library classes can be used. The latter could be seen as the more important of the two tasks, since the standard library classes can access JVM internal functions and do other things that normally is not implemented in Java code. Since some of the functionality in the standard library classes has the potential of destroying the isolation between applications, a carefully compiled list of allowed and disallowed classes is needed. Also, it is necessary to prevent non-trusted applications in the system from extending their own rights or launching new applications with more rights than their own by creating and installing its own resolvers.

If none of the installed resolvers can produce the class that the classloader wants to load, the class loading will fail, and the application will terminate. This is not a very big problem, since normally all user classes are loaded before the program is allowed to start, thus not creating any program terminations while the program is actually running[4]. This happens because the JKernel eagerly loads all necessary classes recursively. This is true for user classes, but system classes will not be loaded recursively. System classes are loaded with the system classloader, and this will cause other system classes that are referenced from the first one to load using the normal java class loading system, which means that they will be lazily loaded as well as not being affected by JKernel resolvers and their restriction. The resolvers must therefore be even more carefully designed, so that no blacklisted system classes can be retrieved or accessed through a system class that is allowed to be loaded. An application that was terminated due to a failed resolver lookup cannot run in JKernel unless the resolvers are modified, which means that if this happens the only alternative is to run the application in its own virtual machine.

[4]If class resolving fails, either the system is misconfigured or the user is trying to run a program using unauthorized classes, in which case the program should not start.

4.3 Capabilities

The IPC system included in the JKernel is called Capabilities. This is an IPC system in which it is possible to share objects between tasks and invoke methods in objects located in another task. By using Capabilities, objects can be shared between tasks, and the task that created the shared object can later on revoke the shared object, which means that it can no longer be used by the task that it was seeded to - using a capability after it has been revoked will cause an exception to be thrown. To create a capability, an interface and an implementation must be created. The interface should define which methods and fields should be accessible to the remote task, that is, the task that did not create the capability. The implementation then implements this interface. To set up the capability, the sharing task will call the other tasks seed-method with the name of the implementation class. A reference to an instance of the implementation class will be returned and at the same time a stub class following the same interface will be created in the remote task. This way, methods can be called from both tasks acting on the same object, providing a way for data to be shared. The remote task has however no possibility to get access to the reference to the original implementation object, and when the capability is revoked the stub object will no longer be able to communicate with the real implementation object in the calling class. It was intended when the JKernel was created that the system should be as similar as possible to the Java RMI interface [6].

When methods are called through capabilities, this is performed like an ordinary method call in Java, only with a little overhead for checking that the capability has not been revoked. This is a major performance advantage compared to ordinary IPC systems. Normally, IPC systems will have to send their data through the operating system kernel, which will lead to a number of switches between processes. This can consume quite a lot of system resources, at least when compared to a method call within a single process. Of course this performance gain can only be made when running an IPC system within a single process the way this project attempts to do. Another performance gain is also possible, due to the fact that the JKernel system uses shared memory. When using large data structures as function call parameters, these can be passed by reference instead of being passed by copy as in most IPC systems. That way, the copying of large amounts of data through the operating system kernel to another process gets reduced to a simple passing of a reference, which can potentially save very large amounts of system resources.

The JKernel uses the capability system internally when creating new tasks. The objects that are shared are the context that a task needs, such as the standard output and input streams, as well as some resolvers for shared

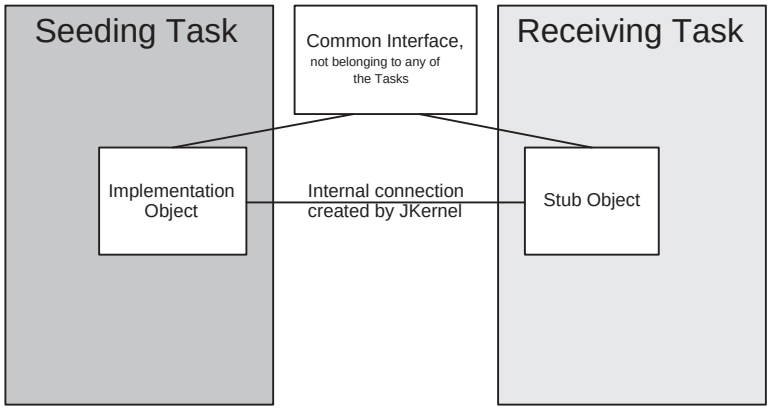

Figure 4.2: Overview of sharing of objects through capabilities

resources. In the JKernel, the seeding functionality is limited so that each task can only be seeded once. It could be changed to allow multiple seedings, but making the different seeds in a task aware of eachother requires a somewhat complex solution. Currently, all objects that should be seeded to a task need to be bundled in one data structure that is seeded to the task by its parent before starting it[5]. This means that all objects that should be seeded must be known by the tasks parent, as well as being known before the task has started. No new capabilities can be set up between a task and its parent after the task has started. It also means that a task only will be able to communicate only with its parent and its children, thus creating a tree-like structure of connections between tasks. Due to this limitation, the Capabilities system can be considered somewhat crippled at the moment.

One potential way to create an IPC system that works between all application would of course be to allow multiple seedings to tasks. This is, however, rather complex, not only to the application programmer that would use it, but also when considering the implementation aspects in the JKernel. Such a solution needs to have a system to give tasks access to other tasks Task objects. Also, it is necessary to prevent seeding capabilities to tasks that have not explicitly permitted this, since the capabilities otherwise would become a security risk.

A better way would be to provide another IPC system, for example a message-based one, on top of the current Capabilities-system, thus letting the messages pass through the tree of parents and children to the right task. The proposed prototype described in chapter 5 will create a flat tree with a main task with all the other tasks as its children, not allowing normal tasks

[5] A task has no code to run before it has been seeded. Therefore, the single allowed seed has to be used to provide this code.

to create children. This way, the distance between two tasks in the system would only be two steps, with the main task as the only middle node. An interesting structure of a messaging system could be for example the one used in D-bus, described in section 3.4.1. It might also be a good idea to make such a system compatible with an existing system that would work outside the JKernel, to allow communication with programs that cannot run in the JKernel. The d-bus might be suitable for this, if extended with a module running in the main task that would switch messages so that messages between processes running in the same JKernel instance would not have to be passed on into the operating system. However, no such efforts has been done either in the original JKernel system or in the proposed prototype described in chapter 5.

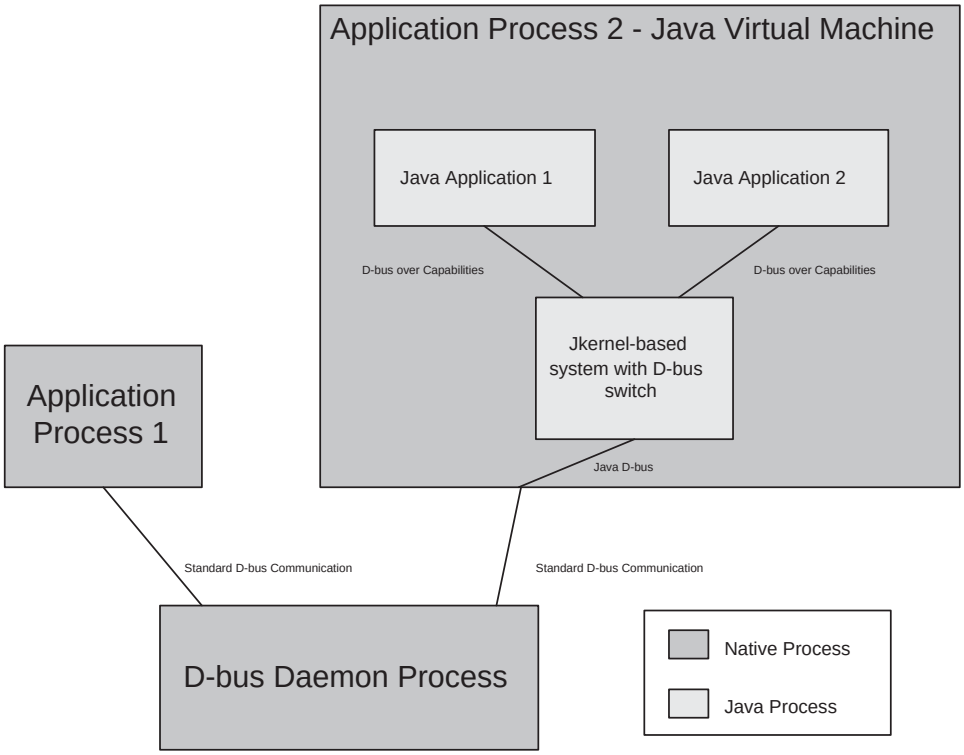

Figure 4.3: Switched D-bus in Java, concept overview

Chapter 5

The prototype system

To prove the feasibility of a system for running multiple application in a single Java Virtual Machine with proper isolation between them, as well as to demonstrate the potential performance gains such a system could introduce compared to running multiple applications in their own instances of the virtual machine, a prototype system has been implemented. The prototype system is based on the JKernel system from Cornell University for its basic functionality. It was then extended with functionality that is preferred on a mobile platform. Also, some modifications had to be made to the original JKernel system, mostly because this was developed in 1998, and several changes to the Java system have been introduced since then, which the system must account for. This chapter describes the prototype, focusing especially on the changes that have been made to the original JKernel system and the functionality that has been added. Problems that are not solved in the prototype are also presented in a separate section.

5.1 The underlying system

The system that the prototype runs on is primarily the OpenMoko platform, using the Jalimo Java environment, which in its turn uses the Cacao JVM and the GNU ClassPath class library to create a Java runtime Environment. The OpenMoko platform and the Java Runtime Environment are described in sections 2.4.1 and 2.4.2 respectively. Much of the testing, however, was performed on the development workstation, which runs Ubuntu Linux, and using Cacao JVM and GNU ClassPath. The versions of the JRE components are the same on the development workstation as on the OpenMoko telephone - differences are just that the Cacao JVM is compiled for different processor architectures, and possibly some differences that can

have been added by the Jalimo team to either of the components on the OpenMoko platform.

The OpenMoko platform, although in development, is a quite stable product. The performance when running Java program using the combination of JVM and Class library supplied by the Jalimo project is however not very good. Except for performance issues, the only major problem concerning the system that the prototype runs on is a problem with the Cacao JVM. The Cacao JVM does for some reason not implement the functionality needed to stop running threads, that is when the method stop() in the class java.lang.Thread is called. Even though the class is located in the GNU ClassPath package, it will in its turn make a native call into the virtual machine, which in the current implementation of cacao results in a call to an empty function, thus letting the thread continue to run. The method Thread.stop() is deprecated in the java standard classes API, but it still should be implemented, which means that this should still be considered a bug in the Cacao JVM.

The possibility to stop running threads is essential to the JKernel-based prototype described in this chapter. The suggested way of stopping threads in java is nowadays to use some sort of an exit condition, designing the thread code to finish whenever this condition becomes true. However, since the JKernel allows user applications to load their own code, which was not necessarily designed to run in the JKernel from the beginning, no such demands can be put on the code. Also, demanding this kind of special conformance to the system would mean that the application programmer becomes responsible for protecting the JKernel against his own application, thus opening up possibilities for writers of malicious code to create threads that cannot be terminated. Many applications also terminate themselves by calling the System.exit() method, which normally would terminate the JVM, meaning that all threads would be stopped without any need for a system for clean exits. In the JKernel, however, the System.exit() calls gets blocked (see section 4.1.3 concerning this) and translated into a request to terminate the calling task instead. To terminate a single task without closing down the virtual machine and thus terminating all other running tasks, some different things need to be done, among them terminating all threads started by the task. This system clearly will leave the threads running after termination of the task, when using the current version of the Cacao JVM. Therefore, the system is not really useable as long as this bug is present in the used JVM. Also, due to the fact that the bug was present in the development environment, and due to the fact that testing cannot easily be performed in another JRE since the prototype is partially tied to the GNU ClassPath class library, the termination functionalities in

the prototype have not been possible to test and to develop. This means that these would probably need some work if it is chosen to develop the system further on a JRE that supports thread termination.

5.2 JKernel modifications and extensions

The JKernel is used as the base for the prototype described in this report. However, the JKernel was not instantaneously useable in its original form. A number of changes have been required to make the JKernel run on a modern Java Virtual Machine, as well as to make it possible to take applications that use modern-day java code and have been compiled with a modern java compiler and run these as tasks in the JKernel. On top of this, some functionality had to be added and changed to turn the JKernel into a useable environment for the intended mobile platform. JKernel was originally supposed to be extended with functionality for the different things it might be used for, but the original intent was to use the system in various forms of servers that need to use java code, such as application servers and similar applications. When it was written in 1998, there was probably no thought of that it could be useful in a mobile device such as a mobile phone. Therefore, a number of modifications and extensions have been necessary to do. This section describes the most important modifications and extensions to the JKernel system needed to create the prototype.

5.2.1 Modifications for compatibility with today's Java

The JKernel was originally created in 1998, and was intended to run on Java Runtime Systems compatible with Sun's Java 1.1. Since then, the JKernel has not been subject to any active development and maintenance from the original team. This means that any features added to Java in later revisions have not been considered by the JKernel team. Running a java application written for an older Java revision in a new Java Runtime Environment is generally not a problem. However, since JKernel aims to control other applications, which might use newly introduced Java features or be compiled with a modern Java compiler, the JKernel needs to be able to deal with those new features. For example, since the JKernel classloader examines and rewrites the bytecode of all the classes that are loaded with it, it must be aware of the new features in the bytecode. Also, since the resolver system contains lists of which standard library classes a task running in JKernel is allowed to use, these lists should also be up to date with later revisions of the Java system. This is necessary to make sure that no dan-

gerous classes are permitted, while at the same time not reducing the set of available classes unnecessarily much.

The parts of the JKernel handling bytecode were originally only prepared for Java 1.1 bytecode, which has since then been extended with some extra attributes. The system was partially prepared for handling this, by designing the bytecode analyzing parts so that unknown attributes are preserved just as they were in the original file. This works for a number of the new attributes that have been added. However, in some cases, it is not possible to just copy the data. Instead the meaning of the data must be understood and modified before putting it into the resulting bytecode. The most common case when this is needed is when the data in the attribute consists of references to data on other positions in the data structure holding the bytecode[1], positions which may have moved around during the bytecode rewrital. Support is implemented by creating handler systems for the new attributes using the same techniques as older attributes were handled, and adding these new attribute as possible cases in the places in the code where attributes are identified. This means that these modifications are spread out all over the bytecode handling classes, and not very easy to identify as parts of the modifications of the JKernel introduced for this prototype. An example of a difficult attribute to handle is the InnerClasses attribute, which contains information about the inner classes that a class might have. This attribute has a variable length depending on the number of inner classes in the class, as well as references to strings in another place in the bytecode file containing the names of these classes. Therefore, the InnerClasses attribute is a good example of the techniques used to handle new attributes.

Another thing that needs to be updated with new Java revisions is the lists containing standard library classes that the tasks are allowed to use. The lists are used by the resolvers in the JKernel to tell the difference between classes that are safe to use by tasks running in the system and classes that the tasks should not be allowed to use. If the lists are used with another java revision than they were created for, some of the allowed standard library classes may have been extended with functionality that threatens the security of the system. This threat might seem rather unlikely at a first glance, but it should be noted that the blocking lists only work for the classes that are directly referenced from the application code. Forbidden classes can still be referenced internally from other standard library classes that are on the list of allowed classes. Another aspect is that when new features enter the standard library, the corresponding classes should be evaluated to see if they can be allowed in the JKernel system. Otherwise,

[1] Typically a byte array, a stream or a file

the JKernel system will slowly get crippled since new functionality that applications use never get introduced. Therefore, the lists should be updated for each new Java revision. The original JKernel package, for example, did not support any of the packages for graphical user interfaces, such as AWT or Swing. It should be noted that the lists used in the prototype have not been developed using any particular analysis of the standard library, but rather just extending the original set of allowed classes with those classes needed to load the applications used during testing. There is no guarantee that either the classes on the original lists or the ones added with the prototype actually are safe to use without threatening the security of the system.

5.2.2 The task launch system

The largest module that the prototype adds to the JKernel is the task launch system. It is a system that lets the running tasks initiate new tasks in a secure way, by delegating initialization requests to a centralized task loading system. Tasks cannot be allowed to freely start their own subtasks, since the launch system needs to have control over the created tasks and which resolvers these tasks get. The launch system needs to keep track of all running tasks to make it possible to terminate a running task at any time[2]. The system is based around capabilities to send requests from tasks to the launch task.

Every task is equipped with a TaskLoader object, which is a capability giving access to methods to send requests back to the launch task. From the perspective of a task, the TaskLoader interface is the interface to the entire Launch System. Through this interface a task can send requests for new tasks to be initiated, as well as requests to start native programs, which will then be run as ordinary operating system processes. A task that cannot run in JKernel, due to for example use of classes that the resolver doesn't allow, can instead be run in a separate JVM instance using a request to run the JVM as a native program. The native programs started through JKernel will not have any possibility to access the launch system or any other parts of the JKernel system, since they are running as separate operating system processes, and in the current implementation the JKernel is not connected to any operating system-wide IPC system. The TaskLoader interface also gives access to methods for requesting other special events such as the termination of other tasks and native processes that have been started[3]. The system will however perform a verification first, to check that the calling

[2]Terminating tasks currently doesn't work anyways, due to the problem described in section 5.1

[3]Currently, it is only possible to terminate all tasks at once, and not an individual task.

task has the permission to perform the request. In the prototype, the first task started after the launch system is given the privilege to perform such requests, but other tasks will be blocked from this feature. This behaviour could however be changed if needed.

For convenient access to the TaskLoader system from within tasks, a singleton is available in every task, giving access to this whenever needed. This is necessary, since the user code of every task is expected to be initialized with a main method in the same manner as normal Java applications, which only can take strings as input parameters and not other objects. The solution is also attractive for convenience reasons, providing applications with access to the TaskLoader system from anywhere in the application code. There is also a class called TaskStartRequest which packs all information about a task that should be started in a single package. It is used in the internal workings of the launch system, but it can also be used in tasks for convenience reasons, especially since it can set the parameters that were not specified by the task implementation to default values. Unfortunately, the TaskStartRequest class currently cannot be sent as an argument through the capabilities system, so the information will have to be extracted from the object again before sending the request to the launch system.

Internally, there is much more functionality in the launch system than just the TaskLoader class. The internal functionality is hidden away to the tasks, to prevent tasks from bypassing the TaskLoader class and setting up their own instance of the launch system. The launch system starts up the same way as any other task, and is used as the first task started by the JKernel system. The launch system in itself contains no GUI, since the system should not be dependant on a particular GUI solution. Instead, the launch system starts by initializing itself, and then goes on by initiating its first subtask, which supposedly is some sort of main menu program that creates a user interface to let the user start other applications. After that, the launch system will only initialize new tasks when requested to do so by other tasks. The first task that is started is given the privilege of requesting special functionality in the launch task, such as task and process termination, system termination etc. This privilege can be assigned to the first task, since before that task is started, only the launch system is running in the JKernel, which means that it can be the privilege of the launch task to choose what the first task will be. Currently the launch system takes the name of the first task as a startup argument, but this could of course be changed to being hardcoded, read from a configuration file or retrieved in some other way.

The task launch system sets up a thread that listens for requests from other tasks. This is done by using the implementation of the TaskLoader

interface as a monitor, waiting for requests from the tasks through the capability system. The internal launching of new tasks and handling of other types of requests is centered around the ProgramLoader class. This class controls all initializing, termination and registering of tasks and native processes, although some functionality is delegated to other classes. There is also a class called TaskRegister which acts as a storage class that contains information about all the tasks in the system as well as references to objects that should exist in a single copy in the whole system. The TaskRegister is a singleton class, to make sure that there is only one instance storing this valuable information, making it possible to for example create new ProgramLoader objects without losing the information about the system and the running tasks. Both these classes have restricted visibility to prevent other tasks in the system to access them.

The launch system keeps track of which tasks were created, but also which windows were created by these tasks. This is because the windows must specifically be closed when a task is terminated, otherwise they will still be visible after terminating the rest of the task. First the windows are closed, then the Task is closed. The JKernel system takes care of much of the different things that should be fixed when terminating a Task, such as revoking capabilities, terminating threads etc. However, since the JKernel system was not prepared for tasks using windows based on the java.awt functionality[4], this feature was added in the launch system instead. The window-tracking is extensively described in the next section.

5.2.3 The window-tracking system

One necessary add-on to the JKernel system to make the prototype work properly was to add a system that keeps track of created windows and the tasks that created them. During testing of the JKernel system, it turned out that when a task created a window, this window would stay on the screen even after the task was terminated. A window can be disposed of by calling the method dispose() in the java.awt.Window class. This function cannot be trusted to the individual task, but must be performed from the launch system. To accomplish this, the launch system must keep track of every window created in the tasks, as well as information about which task a specific window belongs to[5]. The launch system cannot be aware of which windows the tasks running in the JKernel will create. Registering new windows cannot be assigned to the individual task, since that code is not trusted with such responsibility and not necessarily originally de-

[4] AWT also acts as a base for other GUI systems such as Swing or SWT

[5] Keeping track of which task a window belongs to is only necessary when a single application should be terminated.

signed for use in the JKernel system. This means that the registering functionality has to be located somewhere in the standard library classes. As described in section 4.1.3, "Extra isolation in the System class", there are ways to extend the functionality of standard library classes in the JKernel by creating a wrapper class with the same name, which the classloader system then returns instead of the original class. However, a task can create windows using a lot of different classes, of which some, such as in the case of swt, might not even be located in the standard library. This means that very many classes would have to be wrapped, and also that if one class is forgotten or was unknown when the system was implemented, this class would create windows that were not registered. All of these classes share the common property that they inherit the java.awt.Window class, which makes that class the natural place to implement the functionality. There is only one major problem: Since the classes inheriting the Window class often are standard library classes, they will use the system classloader to load the Window class, without querying the JKernel classloading system. Therefore, if the method used for replacing the System class was used, the wrapper class would never load and the functionality would not work.

Since there is no way for a wrapper for the Window class to be implemented and loaded with the JKernel classloader system, the alternative that is left is to make the system resolver return a version of the Window class that implements the desired functionality. Luckily, this can be done by providing the JVM at startup with directives of extra locations to look for the standard library classes, and if there is a class called java.awt.Window in such a location, that will load instead of the original Window class. The problem with doing so is that when this has been done there is no way to retrieve the original Window class any longer. This means that the new Window class cannot be implemented as a wrapper for the old one, but instead it needs to contain the entire functionality of the class. Since the Java Runtime Environment that is used in the development setup uses the GNU ClassPath standard library, which is open source, a new version could be created using the original code with the desired functionality added to it. It must, however, be noted that this cannot be done with all standard library implementations, since the source code is not always available for this, as well as the licensing rules might not permit such changes. The prototype does, however, use this solution to introduce a modified version of the Window class into the standard class library. It is necessary to use directives when starting the virtual machine to set a new path where the modified Window class can be found, rather than having installed a modified version of GNU ClassPath, since the modifications should only be there when using the JKernel and not when running other java programs.

The most important part of the modifications to the Window class is the functionality for storing references to all windows that have been created. This is stored as a static data structure within the class, in which references to new window objects are stored when their constructor gets called. Another thing that should be stored is some sort of knowledge of which JKernel task created each window. This is more difficult to store. Since java.awt.Window is a standard library class, it is preferred not to make it depend on the JKernel implementation, which means that it cannot have any knowledge of the task concept. One thing that it on the other hand can know about is the concept of classloaders, since this is a feature that is built in into Java. The approach therefore is to store the classloader that the task it belongs to uses, and then let the JKernel system figure out which task this corresponds to. The Window class, being a standard library class, is of course always loaded with the system classloader, so storing that information is of no use. What is needed is a reference to a JKernel classloader, which actually can be retrieved.

The SecurityManager class contains a method called getClassContext(), which returns an array of the classes that were passed on the way to the current point in the program, in a way similar to how exception stack traces work. That method has the "protected" visibility grade, which makes it only visible internally in the SecurityManager class and in classes inheriting that class, but by creating a class that inherits the SecurityManager class, this can be accessed through a retriever method[6]. Using the retrieved array of Class objects, the classloaders of these classes could be examined in order, and the first classloader that is not the system classloader is assumed to be the classloader of the initiating task. Thus, a pair consisting of a Window object and a ClassLoader object can be stored in the data structure for later retrieval by the prototype's launch system. The launch system can thereafter associate a window with a task by comparing the classloader associated with the window with that associated with the task. The Task class in JKernel does for security reasons not allow the retrieval of a reference to its classloader, but it has been extended with a method that, given a classloader reference, can tell whether this is the classloader belonging to the task or not.

[6]To prevent misuse of this class, it is implemented as a private inner class of the Window class. However, there is nothing stopping an application programmer from creating a similar class by himself.

5.2.4 New resolvers

To provide som extra functionality related to class loading, three new resolver classes have been implemented. These add features that were missing in JKernel, but with simple implementations provide very useful functionality.

ZipResolver

The ZipResolver is a resolver that much resembles the FileResolver class provided with the original JKernel package. Whilst the FileResolver retrieves bytecode from files in the filesystem, the ZipResolver retrieves bytecode from files stored in a zip file. Zip files are archives to store other files, thus packaging a number of files into one single file suitable for e. g. downloading over the Internet, and the zip format also gives the possibility of compressing the files stored within. The ZipResolver can store a list of zip files which are searched in the order they appear for a requested file. The most common format used for storing java classes today is however the jar file. Luckily, the jar format is compatible with the zip format, which means that the ZipResolver can be used also for jar files.

JarResolver

The JarResolver is quite similar to the ZipResolver. It is specialized in reading bytecode from jar files, and, contrary to the ZipResolver, it can only handle one jar file in an instance of the JarResolver class. The jar format is quite similar to the zip format, but one difference is that jar files can be configured to be "runnable". That is, a jar file can be used as an input parameter when starting a virtual machine, and then a main() method in a file inside the jar file, that was registered as the main class, would be run. The content of the jar file would be put on the class path. The JarResolver has the capability of extracting the info about the main class of the package, and when requested to resolve the name of the jar file, this class would be returned. Otherwise, it works like the ZipResolver.

InvertedFilterResolver

The JKernel package contains a resolver called the FilterResolver. When a FilterResolver is created it is supplied with another resolver and a list of names of classes. When the FilterResolver is called with a request, it forwards this to its internally stored resolver only if the name of the requested class is in the supplied list, this way creating a filter with a whitelist. It

is frequently used within the JKernel implementation, but is useful anywhere resolvers may be created. The InvertedFilterResolver is similar to this resolver, it only inverts the condition determining whether the request should be passed on to the internal resolver or not. Thereby a blacklist functionality is created where requests are only passed on if the name is not in the list. This is very useful in some situations, but it should be noted that the blacklisting will leave any class that is not on the list free to be loaded, including classes unknown to the resolver, when the FilterResolver on the other hand blocks everything that is unknown. Therefore, the InvertedFilterResolver should always be combined with some resolver that prevents unknown classes from loading, such as a File- or ZipResolver pointing to a folder or archive file with known content, a FilterResolver or something similar.

5.2.5 Security Managers

There are some different classes inheriting the SecurityManager class provided with the extensions to the JKernel package. These are Java 1.1 style security managers. Nowadays, the security system has been replaced, but the SecurityManager still acts as a frontend for the new system. All calls to the SecurityManager from the other standard library classes go through the old interface, which is necessary to provide backwards compatibility with older software and SecurityManager implementations. The JKernel originally disallowed any installation of security managers. This was necessary to change to fix the problem when calling the System.exit() method (see section 4.1.3).

BaseSecurityManager

The BaseSecurityManager disallows any attempt to terminate the JVM, and also disallows replacing the current security manager. Otherwise, everything is passed on to the SecurityManager that it inherits, which is the EmptySecurityManager. The BaseSecurityManager is the default security manager and is installed when the prototype starts up.

EmptySecurityManager

The EmptySecurityManager class is a security manager that basically allows everything that is asked to it. This way, it emulates the previous situation where there was no security manager as closely as possible. By using this security manager, there is technically a security manager installed, although it will never stop any applications from doing anything.

LoggingSecurityManager

The LoggingSecurityManager is really a tool that was used during development of the prototype system. This security manager logs all requests that it gets, including input parameters for security checks that have such. The LoggingSecurityManager has been useful for determining possible places to fetch different events that the JKernel system might want to be aware of, debugging purposes and similar usages, but is not useful to an end user of the system. It is still included with the prototype, since it displays what would happen if applications were allowed to install their own security managers. With a security manager like this one an application running in the system could retrieve very much information about other running applications. This shows why applications cannot be let to install custom security managers. It might be possible to implement a Task-aware security manager system, so that requests to the security manager would be sent to another security manager associated with the task, but no such functionality has been implemented yet.

5.3 Remaining problems

There are some problems remaining that have not fully been addressed in the prototype system. These are presented here, along with the effects each one of these has on the prototype system.

Problem with the Classloader approach

The use of classloaders to provide isolation between tasks running simultaneously in a single Java Virtual Machine has a major flaw, which is that this separation is not possible when using standard library classes. Such classes will have a single instance of every static field that is shared among all running tasks in the virtual machine. This problem has been described thoroughly in section 4.1, "Isolation in JKernel". The problem could be seen as the biggest problem with the approach, as it cannot be solved with the way Java Virtual Machines currently work. The effect of the problem in practice might be quite small, however, which means that the system probably still could be useful for certain tasks.

Missing functionality in the JVM

The Java Virtual Machine used in the development platform had some missing features which should normally be included in a Java Virtual Machine. The missing features are associated with termination of running

threads, which means that the only way a thread can be terminated is by running to an end voluntarily. This makes it impossible to implement proper functionality for terminating tasks. Due to this, the system for terminating tasks is not very well tested or developed. Also, any task that tries to exit itself by using a call to the method System.exit() will probably not terminate correctly when running in the JKernel system as long as this problem persists. Since this problem would go away if the system was used on a proper JVM, this problem could be considered less serious that the previous one.

Problems when tasks have access to other tasks classloaders

If a task gets access to a classloader belonging to another task, it can use this classloader to load its own classes so that they partially belong to the other task. For example, the method Class.forName() can take an arbitrary classloader as an argument and use that for loading classes, or the classloader object itself could be used to create classes. A class loaded this way can access static methods and fields in classes loaded by the other task, which clearly is a security violation. Also, depending on how the other task is designed, other data may be extracted from the other task using the static fields and methods. This problem was discovered in a late stage in the project. Some different possible solutions to the problem has been studied, but none have been implemented in the prototype.

Trouble using some functionality in ActionListeners

Classes implementing the ActionListener interface have a method called actionPerformed(), which is called automatically as a result of for example clicks on buttons in AWT- or Swing-based windows. In this method, some JKernel functionality does not work, such as calling methods on capabilities. The exact reasons for this happening are unclear, but it seems like the executing thread is not considered as a part of any task. The thread was initialized from somewhere in the standard library classes, and thus, bypasses the Thread wrapper class in the JKernel system, which work much like the System class described in section 4.1.3, and instead directly creates a thread by using the original Thread class in the standard library. That way, the thread has not been associated with any task, which affects the possibility to use capabilities. It should be noted that, since the fields System.out, System.in and System.err are passed through capabilities, these do not work in that special thread either. A workaround until the problem is fixed is to make the actionPerformed method synchronized with another

method in the same class, in which another thread is waiting, and letting that thread react to the event instead.

Performance problems due to bytecode analysis

The JKernel goes through all bytecode that is loaded by its own classloaders. This is a process that is quite time consuming, currently making the startup time of applications roughly equal to when running them in a separate JVM, and in some cases even slower. The technology is in itself promising since no JVM startup is required for starting a new program, as well as letting the system load standard library classes only once even when used by several applications. The bytecode analysis and repacking is not really necessary to use the classloader approach for isolation. The reasons for reworking the bytecode is probably to remove finalizers, do some renaming of certain classes as well as to add some functionality for a resource accounting interface. That interface in itself is not a part of JKernel but rather implemented in some other products based on JKernel, which means that it is not of any practical use in the prototype. Also, the capability system might be partially dependant on the rewriting. Removing the rewriting, however, seems difficult, so therefore it is currently left in the JKernel. Significant performance gains could however be expected if it was removed. The performance is further discussed in section 6.2, "Performance".

Potential other problems

During the work with the prototype, different problems affecting the functionality or the security of the JKernel has shown up. It is impossible to claim that the isolation system is perfectly secure without knowing every detail about Java and carefully studying all classes and methods in the standard library. Therefore, even if all the problems that have been presented in this report were solved, this is no guarantee that the system actually delivers proper isolation between tasks. Also, more extensive studies than the project has allowed for would be needed for the prototype to reach a state where the quality of the isolation could be determined with confidence.

5.4 Future work

To turn the prototype into a useable product there are still some things that need to be done. These things can basically be divided into some different categories, which are presented in this section.

Problems that need to be fixed

In section 5.3, "Remaining problems", some different problems left to be solved in the prototype are described. The first problem mentioned, the problem that custom classloaders cannot be used to load standard library classes, will persist as long as the current solution for isolation is used. If this cannot be accepted, then another approach has to be used. The other problems could however be solved, which will be necessary to make the prototype really useful. Especially, the problem with missing functionality in the JVM must be solved, since it makes task termination impossible to perform correctly. After that, the task termination system will need to be checked and further developed, which can not be properly done as long as the problem persists. Also, all problems concerning the security of the isolation system should be addressed, while issues creating difficulties for task programmers and seemingly irrational behaviour, such as the trouble concerning the ActionListener interface described above, have to be evaluated if these need to be corrected or not.

Undiscovered problems

To provide good security and functionality in the JKernel, the standard class library should be inspected for java features that could potentially let a task with malicious intent interfere with other tasks or the JKernel system. Some problems that could threaten the system security have been discovered and most of them are also addressed in the prototype system. However, a complete review of the standard class library would be necessary to make certain that there is no functionality creating such problems left that the prototype does not take into consideration. When such functionality is discovered, countermeasures ranging from disabling a class in the resolver system to switch out the entire class in the class library for a patched version are available, so preventing the dangerous functionality from running should be easy in most cases. However, since the problem must be known to be addressed, the big problem are the unknown parts of the standard library, which therefore need to be explored. Also, the resolver system needs to be configured properly, as to provide as much standard library functionality as possible, while hiding away those classes that tasks cannot be allowed to use.

Tailoring for a specific need

Before using the prototype for any specific purpose, there will be a need to review the system to make it behave in the preferred way. There are a lot of details that could be configured differently depending on the intended

use of the system. An example of such a detail is the way the resolver system works. Here, the set of allowed classes, packages and search paths where code can be loaded from should be defined in the desired way. If the resolver system cannot be configured as preferred, it might be desired to implement new resolvers with the preferred functionality. Other aspects where it might be interesting to change the default behaviour include the access to the launch system, where it might be desired to grant more than one task the extended privileges, as well as the security manager system, where the default security settings could be changed or it might be decided to extend the system with a solution to provide different security managers for different tasks.

Other work

There is also the possibility that other things might be desired to change in the prototype to turn it into a useable product. The prototype was designed to demonstrate the technique, and it is not a finished product. Turning the prototype into a finished product would be a project of greater magnitude than this project, and it could be expected that many things in the prototype would be considered for modification and extension if they should be used in a finished product.

Re-implementation of the system

As described in section 5.3, the JKernel-based prototype does not live up to the technique's promises of reduced application launch time. This is due to features of the JKernel which can be debated whether they are useful enough to the system to motivate the heavy load that they create. None of the features requiring bytecode rewriting are central to the classloader-based isolation concept, so it might be possible to create a software that excludes this step but still brings the desired functionality, with potential performance gains primarily concerning startup time as a result. Removing the bytecode rewriting features from JKernel would require a major rewrital of the software, but it could be interesting to evaluate the possibility to write a new software without those components. The JKernel could act as a model for such a software, providing a good structure that could be reused, but with a rewrital from scratch, there is an opportunity to fix things which are not optimal in the prototype implementation. The development of the prototype has also brought attention to many different problems which would not have been obvious if work was started from scratch without any previous experience of this kind of products. If such a re-implementation is done, it also could be investigated if the new features

added to Java since the JKernel was originally written could be used to implement the JKernels functionality in better ways. The primary reason for doing such an implementation is however to retrieve the potential performance gain primarily in application startup time described earlier, which the prototype does not deliver.

Chapter 6

Results

This chapter presents an overview of the results achieved during this project. First, the functionality of the prototype system described in chapter 5 is evaluated. Then, the results of performance measurements on the prototype are presented and analysed. Finally, the general aspects of the project, such as choice of approach, lessons learned from the project and other aspects concerning the topic in general are summarized. This chapter wraps up the content of the report. Final conclusions are however presented in chapter 7, "Conclusions".

6.1 Prototype functionality

The prototype presented in chapter 5 provides means to run multiple applications in a single instance of a Java Virtual Machine. It shows that the goal to some extent was achievable using the selected approach. There are however some things that do not work as good as intended.

The major flaw of the system is that static fields in classes belonging to the standard class library are shared between applications running in the system. There is no way around this problem with the approach of classloader based isolation such as in the JKernel.

Another, smaller flaw is the one concerning the termination of threads, which currently is not possible in the JVM used in the development environment. The problem in practice makes the system unusable, but the problem should be easy to fix with either a patched version of the JVM or by using a different JVM, therefore making it a less important problem than the previous.

Other than that, there are a number of smaller defects making the system insecure or affecting the way programs that interact with the prototype system must be designed. These are described in more detail in section 5.3

of this report. Especially issues concerning the security of the isolation between application must be solved for the prototype to be considered as properly working.

To verify that the prototype actually works as intended a bigger evaluation of it must be done. This evaluation must include verifying the function of all classes in the standard class library, to check that they cannot be used to breach security of the system and that they are safe to be allowed for use by applications running in the system.

There are also currently some performance issues with the prototype. This is primarily due to some features that are present in the JKernel software, which are not necessary for the intended usage of the prototype, but cannot be removed without a major rewrital of the JKernel code. It might therefore be a good idea to view the prototype as a demonstration system and a source of experience rather than as a useful product. If it is decided to use this technology in a product, creating a new implementation using the prototype as a model should be considered.

6.2 Performance

To illustrate the savings of system resources that the prototype can and cannot achieve, some simple benchmarks have been performed. All values have been measured using a single run, which means that there are no guarantees of the repeatability of these results. The measurements are shown here to provide a simple demonstration of the possibilities of the used techniques, and should not be viewed as a good analysis of the performance of the system. The results are likely to depend on a large number of factors, such as the applications that are tested, the Java Runtime Environment that is used and other factors. However, the results show clearly that significant savings of RAM footprint can be made with the prototype in some cases, although no such effects could be seen concerning startup time.

Testing was performed using small applications, which tend to gain more from a system such as the prototype than larger applications, since the overhead for the Java runtime is a proportionally larger part of the system when running smaller applications. However, since the target platform is a mobile phone, most applications in the system might be expected to be of a fairly small size similar to the applications used in the benchmarks. The tests were performed using two programs. One is the EmptyBench, which is a program that does nothing except for what is necessary to measure its performance. It therefore investigates the special case of running programs with exceptionally low own footprint, making the JRE as large part

of the total footprint as possible. It is used in two slightly different versions, where the measurements of memory consumptions are performed using a version that halts itself when started. The other program is called JCalc. It is a simple program working as a simple pocket calculator, and it's size could be somewhat comparable to normal applications running on mobile phones. JCalc uses Swing for its graphical user interface.

In the tables presenting the measured results, sometimes a value is denoted as DNF. This means "Did Not Finish", which means that the test for one reason or another was not possible to perform. The typical reason for this is that the mobile platform runs out of memory before the test is finished. A workstation PC has so-called swap memory, which means that if the memory gets full, the system can move parts of the information stored in RAM to the harddrive, which lets the system continue working, although with lower performance. A mobile platform does not typically contain a hard drive, but rather a flash memory or some other memory for data and program storage. Such memories are not suitable for use as swap devices, and therefore the amount of memory that can be allocated cannot exceed the amount of physical RAM in the device. When the RAM is depleted, and no memory can be freed, no operations needing to allocate memory will be possible, such as starting new processes. In the case of a Linux-based mobile phone, this will probably result in the phone software crashing or hanging. To illustrate the amount of memory that would have been required to perform the measurements triggering such behavior, the same tests were performed on a workstation PC.

In the tests, a number of instances of the same applications were used. A more correct approximation of the expected use of the prototype in a real mobile phone environment would be to start a number of different applications together and perform the measurements on that. However, this brings factors into account such as the size of the programs compared to eachother, and the overlap in use of standard library classes between the applications. When studying the loading of application-specific classes the difference here does not matter, since these classes will be loaded once for each task anyways, as the prototype currently works. On the other hand, when studying loading of standard library classes, the use of multiple instances of the same application will introduce a 100% overlap among these classes, which of course is not true in most cases where different applications are used. The overlap could however be large between applications, especially if two applications use the same system for the graphical user interface, since those systems typically contain large numbers of classes which are interconnected through inheritance and interfaces, and are probable to be used in most applications using such systems.

6.2.1 Memory footprint

Prototype	Shell script	Test app	instances
79826	DNF	HaltingEmptyBench	100
36491	92496	HaltingEmptyBench	10
DNF	DNF	JCalc	100
41180	DNF	JCalc	10
34844	92116	JCalc	5
32183	24834	JCalc	1
23694	0	None	0

Table 6.1: Memory usage in kB on the mobile platform

Prototype	Shell script	Test app	instances
80873	1347888	HaltingEmptyBench	100
33697	134789	HaltingEmptyBench	10
101091	2695776	JCalc	100
40436	269577	JCalc	10
37066	134789	JCalc	5
33697	26958	JCalc	1
23588	0	None	0

Table 6.2: Memory usage in kB on a workstation PC

Measurements of memory consumption are displayed in tables 6.1 and 6.2. The tests have been performed using the prototype to launch a number of instances of a single applications simultaneously, and then noting how much memory the entire JVM process uses according to the operating system. The values are given by the "top" program in Linux, which can display system resource consumption for all processes running in a Linux system. The program gives the values in percent of the total available amount of ram in the system, with three significant digits, which then has been converted to kilobytes. Since the PC used for the measurements in table 6.2 has about 4 GB of available RAM, compared to 128 MB on the openmoko phone, this means that the values in table 6.2 have a significantly lower precision than those in table 6.1. These measurements have then been compared to measurements when an equal number of the same applications are started in separate JVM instances. The measurements on the PC are included in this report to display the amount of memory that would have been needed to actually run the tests which could not finish on the mobile platform.

It can be seen from the values in table 6.1 that when several instances of an application are created, there are considerable savings in the amount of

RAM needed for JVM processes. The biggest saving displayed in table 6.1 is 62%, which takes place when running five simultaneous instances of the JCalc program. However, table 6.2 shows that in the cases when the tests could not run on the openmoko phone, the savings are even greater. In the extreme case of running 100 multiple JCalc instances, as much as 96% of the ram is saved by using the prototype. In that case the test could not performed either using the prototype or the shell script when running on the phone. The memory consumption is however when using the prototype just beyond the point of what the mobile device can handle, while the shell script test consumes more memory than what many modern workstation PC:s have accessible today. A more reasonable task would be to run ten simultaneous JCalc instances, in which case the prototype would reduce the memory consumption by 85%. What the numbers would be if ten different applications similar in size to the JCalc application is hard to determine. The savings would be smaller, since there would no longer be a 100% overlap between the standard library classes used by the applications. However, since JCalc uses Swing for its graphical user interface, it loads a lot of different classes associated with this that other application using the same system for the GUI would reuse to a large extent.

An idea of the proportion between memory use for the virtual machine, the standard library classes and the application classes and instance data can be created from the statistics in table 6.1. It can be seen that the virtual machine along with the prototype system consumes a large part of the used memory when the prototype is running, at least when running a moderate number of applications in it. The amount of memory required for only the virtual machine and the prototype system, including its main application can be seen in table 6.1 as the bottom-most result, where the number of applications is specified as zero. This number is roughly equal to the memory consumption of a single JCalc instance running in its own virtual machine. Running one instance of the JCalc application in the prototype increases the needed amount of memory with a couple of megabytes, but the increase is still rather small compared to the startup footprint of the prototype. After that, creating more instances of the JCalc application adds on even smaller parts to the overall memory consumption. It can be seen that also with five running JCalc instances the system still has a lower footprint than two virtual machines running the JCalc side by side. Therefore, it is obvious that the prototype provides memory savings already with only two applications running in the system. A reasonable conclusion on this is that the prototype can be expected to deliver memory savings in all cases where more than one application is run at the same time.

6.2.2 Application startup performance

Prototype	Shell script	Test app	instances
4m4s	DNF	EmptyBench	100
25s	22s	EmptyBench	10
1m19s	DNF	JCalc	10
46s	55s	JCalc	5
12s	12s	JCalc	1

Table 6.3: Startup time on the mobile platform

Some measurements of application startup performance are displayed in table 6.3. As presented in this report, the use of a system letting multiple applications share a single Java Virtual Machine has the potential of reducing startup time for applications, since there would be no delays waiting for the JVM to initialize before starting to run the application. Also, many classes that an application needs to load could already have been used by another application, which would make it unnecessary to load these again. When using classloaders to provide isolation the way it is performed in the prototype, application classes have to be loaded individually for each application, while standard library classes are loaded only once and then reused. This means that the startup penalty that every application will have to pay every time it starts is only the loading of application classes, which should be a small job compared to also loading standard library classes and initializing the virtual machine.

The numbers displayed in table 6.3 does not show any significant time savings during application start. Savings are shown in some cases, while in others the startup time gets longer. No certain conclusion can be drawn from the numbers on whether the startup time in normal use cases would be any better or not when using the prototype. The theoretical advantages are there, however, which means that the result can be explained for one of two reasons. Either the effect achieved by those advantages is minimal, or something else consuming system resources has been introduced. It is not likely that the effect really would be of such little significance that it would not be visible in the measurements.

There is however another potential source of delays that has been introduced with the prototype. The JKernel, which acts as a base for the prototype, has built in facilities for analyzing and rewriting bytecode that is loaded through the system. Basically, bytecode is translated into a data structure and then transformed back to a stream of bytes again. During this process, some changes are done to the bytecode. Due to lacking documentation on the original code, it is not entirely clear what purpose all these

changes serve. It does, however, seem like the changes include disabling of finalizers, doing some renaming of certain classes as well as adding features needed for a resource accounting system. The resource accounting system is not useable unless support for it is implemented in an application running on top of the JKernel, such as the task launch system described in section 5.2.2, which means that modifying the bytecode for this currently is pointless in the prototype. Doing such bytecode rewrital is a quite complex task, which takes some time to perform. Since this must be done every time a class not belonging to the standard class library is loaded, this acts as a startup penalty affecting every application that starts, much like JVM initialization would do when performing normal application launch. It seems like this penalty is so big that even with the removal of the other penalties, the system is still not faster than starting every application in its own JVM. Therefore, it must be evaluated if the bytecode rewrital really is necessary and wanted in the product, since it is not really essential to the classloader-based isolation concept in itself.

6.3 General results

As was seen in chapter 3, "Related work", there are some different ways the task of running multiple applications in a single JVM can be performed. The selected approach to use classloaders to provide separation between applications is in its basic form a simple way of solving the problem. It has the advantage of being platform independent, both with respect to the Java Virtual Machine and the underlying operating system. However, the method includes a number of problems that need different workarounds, which makes the task more complex. The prototype, based on the JKernel, is a large software with many classes, primarily to provide extra protection system on top of the classloader-based isolation. Using a modified virtual machine instead is an interesting approach, especially as the functionality could be preferred to have integrated with the virtual machine. That approach would also have provided better possibilities to provide proper isolation between applications. However, a modified JVM would probably have been an even larger and more complex project than the JKernel. Also, the system would not be portable between different platform, since the JVM would have to be ported to each platform first, while the described prototype is easily moved between platforms since it is an implementation made entirely in java. For modifications to the Java runtime to become popular and used, they should be a part of standardised Java, for example by being implemented through a JSR. JSR 121 provides an API for a system such as the one this project has attempted to create, although it does

not define any properties regarding the implementation of the API. This means that an implementation will not necessarily give the benefits that this project is looking for. The JSR 121 is not a part of any major release of Java SE either, which will be necessary for it to gain broad acceptance among JVM creators. Implementing a solution not following the JSR 121 would be unwise, since this only further contributes to dividing the Java world into more and more flavors which are more or less incompatible with eachother. In the meanwhile a classloader-based solution creating a middle layer between the JVM and the applications might be a good alternative.

The prototype described in chapter 5 shows that a system for running multiple Java applications isolated from eachother in a single JVM can be implemented by using only Java as the programming language. However, it also shows that the isolation is not as complete as it could have been, and that there still are some possibilities for applications to disturb each other. Although there are problems persisting in the prototype, most of there are believed to be solveable. The big problem that is clearly associated with the method itself is the problem of using shared standard library classes. As suggested in the IBM patent [11] described in section 3.2.2, this problem might be possible to work around using bytecode rewriting techniques. However, the bytecode rewriting taking place in the JKernel is currently quite slow, resulting in slow startup performance of applications. Rewriting the bytecode of those standard library classes requiring this would of course only add to this slowness. An implementation doing this would need to have a faster bytecode rewriting system than what is available in the prototype, alternatively making sure that only a small number of classes would need this rewrite.

Apart for the prototype system, the project has also resulted in a certain amount of knowledge concerning the subject. Working with the prototype has brought attention to a couple of different pitfalls likely to show up during development of this kind of systems. This knowledge would be useful in for example an attempt to implement a new product solving the studied task, which might implement only those features really wanted for a mobile environment and leaving the rest out. Also, the classloader techniques described in this report are used in different sorts of application servers and other similar systems, which means that the knowledge would be of value also when working with such systems, even though it might seem very different from a mobile environment at a first glance.

Chapter 7

Conclusions

The project has shown that it is possible to, using the methods described in this report, implement a system allowing different independent Java applications run in a single instance of a Java Virtual Machine without very much disturbance between them. However, it has also been shown that the isolation between applications is not complete. There are still ways applications may potentially disturb eachother and potential ways for malicious code to gain access to data belonging to other applications running in the system. Solutions and workaround for some problems associated with the main isolation functionality have been described in this report, while some other problems have been described, along with suggestions for possible solutions or motivations to why these problems are likely to be solveable and why they are not solved in the prototype described by the report. One big problem, which the chosen approach cannot address easily, is also described. Other approaches for a solution, which might be able to address all the known problems, have been presented. However, only the approach using classloaders for providing isolation was examined in greater detail, since this was chosen as a base for the described prototype system.

The potential performance gains that motivated the project from the beginning are partially present in the described prototype. As intended, the prototype accomplishes significant savings in terms of overall memory footprint when running multiple Java applications simultaneously. The reduction in startup time of applications that also was a motivation for the project is however not present in the described prototype. This is however related to implementation-specific details that are not directly related to the use of classloader for isolation. The gains possible by not having to initialize new JVM instances are present, but other things in the system slows the prototype down to about the same level as when running application in separate virtual machines. The prototype has an IPC system which draws advantage from the fact that applications are running in the same operat-

ing system process. This system is not freely useable for communication between applications in its current form, and is presently only used for initializing new tasks in the prototype system. An interface for another IPC system that takes advantage of this system should however be possible to add.

Other than the performance issues, it is shown that the prototype system works as intended for the most part. Problems do however persist and, especially as the performance is not as good as i could have been, it is suggested that the possibility to develop a new system using selected parts of the structure of the prototype as a template is investigated. In such a product, those features slowing down the prototype without providing adequate value could be removed, leaving only the preferred set of features. The prototype is however in itself not suitable for such a modification.

The project has also resulted in a knowledge base of typical problems and issues when working with this kind of product. This could be a valuable resource, both if used for constructing a new application solving the problem in a better way and for use in other products that might seem to have an entirely different purpose but use the same technology, such as for example application servers.

Bibliography

[1] *SavaJe OS: Solving the problem of the Java Virtual Machine on Wireless Devices*
SavaJe Technologies,
2002

[2] *OpenMoko homepage*
http://www.openmoko.org,
2008

[3] *Jalimo*
http://www.jalimo.org,
2008

[4] *Cacao JVM homepage*
http://www.cacaojvm.org,
2008

[5] *GNU ClassPath*
http://www.gnu.org/software/classpath/,
2008

[6] Thorsten von Eicken, Chi-Chao Chang, Grzegorz Czajkowski, Chris Hawblitzel, Deyu Hu and Dan Spoonhower
JKernel: a Capability-Based Operating System for Java
Cornell University,
1999

[7] *JKernel homepage*
http://www.cs.cornell.edu/slk/jkernel.html,
2008

[8] Alex Kalinovsky
Covert Java - Techniques for Decompiling, Patching, and Reverse Engineering
2004

[9] *Echidna on SourceForge.net*
http://sourceforge.net/projects/echidna/,
2008

[10] *Echidna as a Java Shared VM surrogate*
http://www.beanizer.org/site/index.php/en/Articles/Echidna-as-a-Java-Shared-VM-surrogate.html,
beanizer.org,
2008

[11] Matthew Paul Chapman
US Patent No. 6851112 - Virtual Machine Support for multiple Applications
IBM Corporation,
2000

[12] *JSR 121: Application Isolation API Specification*
http://jcp.org/en/jsr/detail?id=121,
Sun Microsystems,
2008

[13] Grzegorz Czajkowski and Laurent DaynÃĺs
Multitasking without Compromise: a Virtual Machine Evolution
Sun Microsystems,
2001

[14] Janice J. Heiss
The Multi-Tasking Virtual Machine: Building a Highly Scalabe JVM
Sun Microsystems,
2005

[15] Godmar Back, Wilson C. Hsieh and Jay Lepreau
Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java
School of Computing, University of Utah,
2000

[16] *JanosVM User's Manual and Tutorial Version 1.0*
Flux Research Group,
School of Computing, University of Utah,
2003

[17] *The Janos Project homepage*
Flux Research Group,
School of Computing, University of Utah,
2008

[18] *Connected Limited Device Configuration HotSpot Implementation Multitasking Datasheet*
Sun Microsystems,
2004

[19] *D-bus homepage*
http://www.freedesktop.org/wiki/Software/dbus,
freedesktop.org, 2008

[20] Matthew Johnson
Java D-Bus Implementation Documentation
http://dbus.freedesktop.org/doc/dbus-java/
freedesktop.org,
2008