Performance Isolation for Component Systems

Master Thesis by:
Ramon Schwammberger

Advisor: Jan S. Rellermeyer
Referee: Gustavo Alonso

December 7, 2009
Contents

1 Introduction and Motivation 5

2 Background and Related Work 7
   2.1 Software Stack 8
   2.2 Hardware Layer and Xen 8
   2.3 Operating System Layer 9
   2.4 JVM Layer 10
      2.4.1 OSGi 11
   2.5 Middleware Layer 12
      2.5.1 R-OSGi 13

3 Isolating Java based Component Systems 13
   3.1 Hardware and JVM Layer 13
      3.1.1 OSGi 14
   3.2 Middleware Layer and Networking 14
      3.2.1 Enhancement of R-OSGi 15
   3.3 Use Case Application 15

4 Evaluation 16
   4.1 Performance of GuestVM 16
      4.1.1 Test Setup 16
      4.1.2 Results 16
   4.2 Performance of Inter-Domain Communication 17
      4.2.1 Test setup 17
      4.2.2 Results 17

5 Conclusions 21

6 Future Work 22
Abstract

Enterprise applications profit greatly from virtualization, which not only allows server consolidation but also coarse-grained isolation of subsystems. Performance isolation prohibits subsystems to affect each other, which is especially good for enterprise applications. Another benefit is security isolation, which protects the system from compromised or misbehaving subsystems. The costs of virtualization are overhead from duplicated foundations with full operating systems and complete software stack. This work presents a lightweight platform for Java\textsuperscript{TM} based component systems which counters the introduced overhead with a small foundation. The performance of the Java Virtual Machine of that platform and the performance of inter-domain communication are evaluated.
Introduction and Motivation

Modular design is a key concept in software and system engineering. Modularization enables decomposition of software or systems into loosely coupled components which encapsulate more tightly coupled elements. This alleviates development, maintenance and reuse of modules. It also simplifies the introduction of isolation as tightly coupled components are hard to isolate. Many programming languages support different namespaces to facilitate modularization of program code. Decomposition of applications opens the door for deployment modularization where components can be reused easily and the system can be composed at runtime rather than at compile time. Such systems are called component systems.

A more coarse-grained form of modularization is virtualization which brings also the benefits of improved resource utilization, security and robustness. The drawback of virtualization is that it introduces overhead because it duplicates the foundations, the operating system and the software stack for each virtualized machine. Enterprise applications make often use of virtualization for the sake of performance isolation. These big and complex systems face many challenges. Despite their size and the usually large number of subsystems, they need to be robust, secure and efficient. Scalability and dependability are put to test as applications grow. Performance isolation helps to cope with those requirements by preventing isolated subsystems to affect the rest of the systems. Only explicitly defined communication is allowed between isolated components.

Best isolation is achieved if each component runs on a separate machine, thus granting exclusive access to time resources, like CPU and devices, as well as access to space resources, like volatile and persistent storage space, to the component. To keep a tap on the number of needed machines and because many servers would be underused if they run only one component, a multi-tenancy solution must be used. Virtualization is such a solution and it preserves strong isolation.
This work presents a lightweight platform for Java based component systems which provides strong isolation of components through the use of virtualization (Chapter 3). The introduced overhead is countered with a small software stack foundation. This component platform is evaluated against a benchmark which simulates an enterprise application in Chapter 4. The next chapter (Chapter 2) starts with some background on virtualization and technologies related to the software stack of component systems.
Chapter 2

Background and Related Work

Today's component systems for Java based applications provide only limited security isolation and almost no performance isolation. This is due to the fact, that all Java components run on the same Java Virtual Machine (JVM) and hence share time and space resources. This makes security isolation in the JVM hard to implement and maintain. Performance isolation is almost impossible to achieve, because a misbehaving application could always selfishly acquire as many resources as possible and thus letting all other applications suffer resource starvation. To prevent this, the system would have to be able to discern a demanding application from a misbehaving application, which is not reliably feasible in practice. Consequently, it is best practice for security isolation to use one JVM per critical application. However, even with multiple JVMs, the operating system's resources are still shared, which can be problematic for performance isolation. If resources cannot be reliably and exclusively bound to JVMs then the use of one machine per critical application is necessary. To counter the increased need of server machines and for more efficient resource usage, virtualization can be used to run each critical application on a virtualized machine.

Enforcing isolation makes communication between isolated entities always explicit and this communication needs to be taken care of. Standard Unix inter process communication (IPC) calls might not always be possible, especially if the communicating entities are not on the same machine. Classic solutions for distributed systems are Remote Procedure Calls (RPC) [28], Java Remote Method Invocation (RMI), the Common Object Request Broker Architecture (CORBA) [29] or the Distributed Component Object Model (DCOM) [30].

Virtualization allows to run multiple virtualized machines (VMs) on one physical machine. The virtualization layer takes then care of multiplexing hardware resources and device accesses. There are different types of virtualization platforms with different degrees of virtualization. Platforms that provide full virtualization need to modify binary code of their guest operating systems at runtime to adapt the hosted guest operating systems to the virtualization platform. The VMware
Workstation [22] is such a platform. Other virtualization solutions, like Xen [25], require the guest operating systems to be modified at compile time. Each virtual machine has to duplicate a whole operating system as well as the software stack that is needed by the applications. This introduces overhead, wastes resources and is prohibitive to using virtualization for performance isolation. There are virtualization platforms like Sun’s Solaris Containers [23] which try to counter this overhead by sharing the operating system to various degrees. This in turn makes isolation more difficult to achieve because of the tighter coupling between the VMs.

2.1 Software Stack

Traditionally, enterprise applications rely on an extensive and often complex software stack. Each layer depends on the layers below and in each layer there is the possibility of unanticipated interactions between its components. To alleviate this burden of maintaining such a software stack, applications with their software stack components are often separated from other applications and their stack components. This provides isolation of configuration as well as isolation of temporal and spacial resources. As applications grow in size and complexity, one can even go a step further and separate the components of an application.

In [24] the authors point out the many advantages of decomposition. Besides the above mentioned advantage of technical compatibility, there are more compatibility issues in terms of legal regulations, security concerns and quality of service requirements. Decomposition can also be used to match the conceptual understanding of a system, to separate the stakeholders or simply to isolate proprietary modules from the rest of the system.

Paper [24] introduces MacroComponents as software components that run in isolation from the rest of the system. The authors argue that virtualization can be a viable instrument to provide performance isolation, provided that the MacroComponents are lightweight. Therefore, it is important to minimize the software stack of components. The VM of MacroComponents are managed by the MacroComponent framework.

The topmost layer in the software stack is the application layer and it has no potential for optimization in a decomposed scenario since it only contains the actual component of the system.

2.2 Hardware Layer and Xen

The lowest level is the hardware level. Traditionally, operating systems have direct and exclusive access to all hardware devices. This is no longer possible when several virtual machines run on the same physical machine. Resource sharing is done by introducing an abstraction layer, which manages hardware access. Xen [25] is such an abstraction layer, also called Hypervisor. Xen allows paravirtual-
ized OSs to run in parallel in so called “domains” on the same physical machine. Paravirtualization as opposed to full virtualization means that the source code of the operating systems needs to be modified to run on the virtualization host. Xen needs a privileged domain, called Domain0, which provides device drivers for the hardware. This design makes the Hypervisor very lightweight. Device drivers of the operating systems running in unprivileged domains, called DomUs, communicate over shared memory with the device drivers in Domain0. This detour creates overhead which is especially big if two DomUs on the same physical machine want to setup a network connection. The communication path starts at one domain, goes through the network stack of the guest operating system and gets handled by the frontend driver for the virtual network interface. The frontend driver sends the data over a ring of shared memory to the backend driver in Domain0. Xen calls this shared memory ring an “IORing”. Domain0’s backend driver relates the data to its network stack, which routes the packets and sends them through the backend driver to the IORing of the target DomU. From there the data has to go through the frontend driver of the second DomU and through its network stack until the network packet is finally received.

The obvious solution to this situation is to enable the two DomUs to communicate directly over shared memory with each other. XWAY [21] introduces an additional layer right above the TCP layer. This layer determines if a network packet can be sent directly over shared memory or if it needs to go through the network. Another solution are Xensockets [33] which provide a socket interface to the operating system, but the communication goes through IORings directly to the target DomU. Xensocket is implemented as a kernel module.

Setting up a Xensocket connection is slightly different from setting up a standard network socket connection. On creation of the Xensocket, the own domainId must be known. To allow another domain to connect to this socket, an IORing with permission for the target domain is setup. The permission is a reference in Xen’s grant table which grants the target domain access to the shared memory region of the IORing. This grant reference has to be delivered out of band to the target domain together with the domainId of the first domain. The target domain can then create a Xensocket and connect to the opened socket of the first domain. This opens the communication channel from the target domain to the first domain. Version 1.0 of Xensockets are one way only.

2.3 Operating System Layer

In a non virtualized environment, an operating system (OS) has direct access to the hardware and the task of the OS is to manage resources and to multiplex the access to the hardware for all the layers above the OS layer. Some approaches reject the management part of the responsibilities, which resulted in the design of the Exokernel [26] and similar. If the OS runs in a virtualized environment then hardware access is already multiplexed by the Hypervisor in the hardware layer. Thus, the operating systems only task is that of a compatibility layer. “Libra”
Performance Isolation for Component Systems

[3] is a library operating system that is tailored for JVMs in virtualized execution environments.

2.4 JVM Layer

The Java Virtual Machine is the platform specific Java language runtime that executes bytecode. This makes Java applications platform independent. There are many sizes of JVMs from different suppliers. The most prominent ones are Sun’s Hotspot JVM and OpenJDK’s implementation, but there are also JVMs like Squawk [13,14] which implements CLDC[27] and targets resource constrained devices like mobile phones or Sun’s SPOT [34]. CLDC has the restriction that it only supports a subset of the Java language and it is not suited for dynamic classloading.

The OSGi platform is a Java application itself and therefore needs a JVM to run. This has consequences in terms of security as Parrend and Frenot [6] point out. The bundles are not enough isolated because they run on a single JVM. Parrend and Frenot elaborate that malevolent bundles could alter shared variables or block resources. The paper about I-JVM [7] tackles those problems by introducing more isolation in a JVM. I-JVM does not need virtualization but has still to deal with explicit communication, because of the introduced isolation. It solves this with inter process communication (IPC) calls. Cloneable JVMs [17] propose the use of virtualization to provide isolation, but the Cloneable JVM approach does not focus on a small software stack but rather on fast instantiation of VMs by cloning them.

Traditionally, JVMs have been implemented in C and thus run natively on a machine. The Kilobyte VM [20], which has its roots in the Spotless JVM [8], is such a JVM. But there are also JVMs which are written in the same language that they interpret, so called meta-circular JVMs. The above mentioned Squawk is such a meta-circular JVM, it is mostly written in Java and executes Java byte code. Squawk’s paragon is Squeak [10], a meta-circular interpreter for smalltalk-80 [18]. Other meta-circular JVMs are Sun’s Maxine [12], JavaInJava [19], and IBM’s Jikes [4] (formerly known as Jalapeño).

There are several JVMs that skip the operating system run directly on the hardware. The Kilobyte Virtual Machine for Java CLDC, Jikes and Squawk are such VMs. A special kind of a “bare-metal”-JVM is GuestVM [16] is based on the meta-circular JVM Maxine[12] and has been enhanced to run on Xen in an unprivileged domain. More precisely, it runs on GUK which is a mini kernel derived from Xen’s Mini-OS kernel.

Other efforts tried to combine the operating system with the Java Virtual Machine to optimize performance, the JX Operating System [9] and JNode [11] are such examples.
2.4.1 OSGi

The OSGi™ Framework [5] is a dynamic module system for Java and has been designed by the OSGi Alliance. Early specifications aimed for small devices as target platform, but the framework was equally successful on big server systems and newer specification versions aim for big and powerful servers. The OSGi platform can be described as an extensible Component platform on top of a standard Java Virtual Machine which must be at least compatible with CDC Foundation 1.0. The platform takes care of the life cycle management of software components, called bundles. A bundle is a collection of Java packages packed in a Java Archive (JAR) together with some metadata and optionally with an activation class to start the bundle.

Besides dependency management and starting, stopping, updating and uninstalling bundles at runtime, OSGi also provides a platform for registering and finding services, thus enabling loose coupling between bundles. OSGi’s flexibility and isolation is achieved by giving each bundle its own classloader, thus introducing separate class spaces for each bundle.

OSGi has a whiteboard pattern [31] that allows bundles to subscribe to announcement or revocations events of services. This enables the bundles to react to changes in the set of offered services. Such events are also fired when bundles get installed, updated or uninstalled and bundles can subscribe to them to take actions if needed. There are several implementations of OSGi: Equinox [35], Knopflerfish[37], Felix [36], Oscar [38], Concierge [1] to name a few. Equinox is the most prominent of those four. It is the OSGi implementation on which the Eclipse IDE is built. Concierge targets small and resource constrained devices and version 3 even outperforms its competitors.

2.5 Middleware Layer

The middleware layer supports the application layer. This is where the life cycle of applications is managed. It is also here that company wide common grounds for all applications are established. Because this layer is often company specific, it is often self made and tends to grow overtime. Reducing this layer to a minimum does not only shrink the software stack but also makes this layer more sustainable and brings economic benefits for the company.
2.5.1 R-OSGi

Remote-OSGi (R-OSGi) [2] is based on OSGi and enables the developer to turn modular applications transparently into distributed applications. It is non-invasive to OSGi, that means it does not need to change the framework. R-OSGi uses dynamic proxy generation to a local proxy bundle which mirrors remote services. Those services are managed by a distributed Service Registry based on SLP. No new failure patterns are introduced, because R-OSGi maps network and remote failures to local bundle failure events. Thus, local bundles are not aware of the remote connection and failure handling does not change.

R-OSGi uses code analysis to determine differences in the set of defined types between local and remote bundles and then uses type injection to provide the service proxy bundle with the missing types, making it self-contained.

Coign [32] for COM/DCOM is the only other system that can turn modular applications into distributed applications. Coign partitions the applications into two parts that can be distributed in a client/server configuration.
Chapter 3

Isolating Java based Component Systems

The following sections present a platform for Java based component systems with strong performance isolation of components. Isolation is achieved through virtualization and the overhead of virtualization is countered by a minimized software stack. This stack consists of a JVM that runs directly on the hardware hypervisor. Thus eliminating the need of an operating system. That JVM is enhanced with an OSGi framework for life cycle management of the components. The middleware layer consists only of R-OSGi which transparently connects the components and takes thus care of the explicitly needed communication because of the introduced isolation. The last section in this chapter presents the use case application against which this platform is evaluated.

3.1 Hardware and JVM Layer

Performance isolation of this component platform is achieved by giving each sensitive component its own machine. Virtualization is used to counter the need of an increased number of machines. This isolation counters all JVM-related security issues of OSGi identified in [6]. The costs of this feature are the virtualization overhead. Virtualization is powered by the efficient Xen Hypervisor. Additionally to the isolation, a lot of flexibility is gained by using Xen. Although, the possibilities of migrating a running component in its virtual machine has not been explored in this work due to time constraints.

Operating systems have the task of a compatibility layer on virtualized systems. This task is taken over by the “bare-metal” JVM GuestVM which runs directly on Xen’s hypervisor. This eliminates the need of an operating system. No operating system also means that the OS is no longer an attack vector for this system. Even though GuestVM is not yet finished and has problems with JAR streams, there were no modifications to the source code necessary after revision 142 of the Mercurial repository. Bugs in the TCP stack and in ObjectStream handler have been fixed by revision 142.
3.1.1 OSGi

The JVM layer has been enhanced with OSGi to provide life cycle management of application modules. Of all the available OSGi implementations, “Concierge” has been chosen because it is small and efficient and because it has been developed at ETH Zurich. Concierge implements most parts of OSGi’s specification version R4.2.

Concierge’s dependency resolving process has undergone big changes since version 3.0 to handle the requirements of OSGi’s specification R4.2, but it had not yet a sophisticated algorithm to find the best supplier for required Java packages. Because of the focus on isolation, there should never be many bundles in one JVM. Accordingly, the algorithm can focus on good best and average case performance. Although the influence of this algorithm on this component platform is expected to be neglectible.

Further modifications were necessary to adapt Concierge to be able to circumvent GuestVMs deficiencies. Because GuestVM can handle neither compressed JARs nor embedded JAR streams, a shell script has been used to flatten OSGi bundles. Embedded JAR files are written into a directory named after the JAR file but without the “.jar” ending. Concierge has been modified to search also for such directories when all other attempts to find the requested resource failed. This is not a permanent solution. If a bundle has already a directory or file with the same name as the jar file then this workaround fails.

3.2 Middleware Layer and Networking

R-OSGi is the only middleware layer component and provides transparent networking for components. Thus taking care of the explicit communication which is needed to cross the introduced isolation.

Xen’s architecture imposes an overhead on network communication for unprivileged domains. If both domains reside on the same machine then this performance penalty is doubled. This overhead of inter-domain network communication is prohibitive to the deployment of components of distributed applications to different virtual machines on the same physical machine. This problem has been solved by enhancing R-OSGi to use Xensockets for fast inter-domain communication when both parties reside on the same physical machine.

Xen prohibits an unprivileged domain to get its own domain identification number (domainID), because that domain could be migrated at any time and the domain should not be able to notice any difference. That domainID is a consecutive numbering of all domains on the physical machine and it is only available in the controlling domain0. Unfortunately, this domainID is exactly what Xensockets need to establish a connection. Therefore this domainID has to be provided when OSGi is started in an unprivileged domain. Otherwise, R-OSGi will not be able to use IORings for fast inter-domain communication and it will default to networking over TCP.
The implementation of Xensockets 1.0 had to be adapted to run on Xen 3.2.1.

### 3.2.1 Enhancment of R-OSGi

To teach R-OSGi to use Xensockets, three new Java classes have been added to R-OSGi. The first new class is Xensock which provides wrappers for the native Xensocket calls. More precisely, Xensock makes JNI calls a shared library “libxensock.so” which then calls the appropriate Xensocket function in the Xensocket kernel module. This indirection was only used for prototyping to facilitate debugging.

The second class is the XenChannelFactory, which is an enhanced TCPChannelFactory. This XenChannelFactory sets up a TCP socket and sends lease messages as well as service request and delivery messages still over TCP. The service request message has been expanded by an integer field which holds the domainId of the domain in which the requesting R-OSGi bundle is installed.

If the receiver of such a service request message also knows its domainId, it will open a Xensocket for the requesting domain. The grant table reference for Xen’s shared memory and the receiver’s domainId are then passed with the service delivery request over TCP to the service requester. The service requester connects to the xensocket of the receiver and has established a one way communication channel to the receiver. The requester opens a new xensocket and grants the receiver the rights to connect to it. This permission is then transmitted to the receiver which connects to the socket and a two way communication over shared memory between the two bundles has been established.

This last message from requester to receiver is the third new class to R-OSGi, the XenSocketMessage class. In the prototype, this XenSocketMessage is sent over TCP. However, there is no need to do so, it would be faster to send that message over the first shared memory channel which is at that point in the protocol already established.

From this moment on, all subsequent messages are sent over the two shared memory channels, except for lease and service request and delivery messages.

### 3.3 Use Case Application

Enterprise applications profit a lot from performance isolation, which is the speciality of this platform for component systems. Hence, TPC-W [39] has been chosen to evaluate the performance of this platform. TCP-W is a benchmark which measuring web interactions for a simulated web application. A two tier e-commerce application provided by the TCP-W project will represent a typical use case application.

The electronic bookstore has a frontend component that is responsible for presentation of all data. The backend component of the system is a database application which provides the data.
Evaluation

Performance of two components of the software stack, which has been introduced in Chapter 3, is evaluated. The first series of tests compare the JVM’s performance against other implementations of Java Virtual Machines. The second series of tests compare R-OSGi’s usage of Xensockets for inter-domain communication to its performance over the TCP stack.

4.1 Performance of GuestVM

The performance of GuestVM is compared against the performance of Sun’s HotSpot JVM, OpenJDK, Jikes and Maxine. The HotSpot JVM and OpenJDK are run in server mode as well as in client mode. Measurements where taken from the specJVM98 benchmark.

4.1.1 Test Setup

The test machine is a OptiPlex GX620 from Dell, equipped with a dual core Intel(R) Pentium(R) 4 CPU running at 3.40GHz. It has 1024MB of RAM with 16KiB L1 and 2MiB L2 cache and it runs on the Linux 2.6.24-24-xen kernel. The tested HotSpot JVM is JDK_1.6.0_15-b03, the version of OpenJDK is JDK_1.6.0_0-b11, Maxine is revision 1122 and Jikes is revision 15761. GuestVM is revision 142 from the repository and relies on Maxine revision 3018. The virtualization layer is Xen-3.2.1.

4.1.2 Results

Figure 4.1 shows that Maxine and Jikes are very close together in terms of performance. They are also only insignificantly slower than Sun’s HotSpot and OpenJDK’s JVMs.

Figure 4.2 shows that GuestVM is up to 26 times slower than its fastest opponents. Even though GuestVM is based on Maxine, it is on average 8 to 9 times
slower than Maxine. GuestVM even failed two tests. Its runtime for _200_check and _228_jack is zero and cannot be compared to the values of other JVMs. The first test failed because of an array out of bounds exception. And the second test failed with a compilation error.

### 4.2 Performance of Inter-Domain Communication

The introduced performance isolation requires bundles to use explicit inter-domain communication to connect to each other. R-OSGi has the task of providing such network connection in a transparent manner. R-OSGi has been configured to use either a TCP connection or a faster inter-domain connection over shared memory. In this suite of tests, JavaParty is used as the benchmarking tool. JavaParty measures the round trip time for ping messages with various payloads.

#### 4.2.1 Test setup

This test machine is from Dalco AG and has an Intel(R) Core(TM)2 Quad core CPU Q9400 which runs at 2.66GHz. Its 4GiB of RAM have 3MiB of L1 and 32KiB of L2 cache. It runs on the Linux 2.6.24-25-xen kernel and has Sun’s HotSpot JDK 1.6.0_15-b03 installed. Xen-3.2.1 provides the virtualization layer. The operating system Debian Lenny is installed in the VMs. The first test run started JavaParty’s sending and receiving components in the same VM. Both modules were configured to use TCP connections only. The second test run put sender and receiver in different unprivileged domains. Communication was still TCP only. The third test run had the same setup as the second one, but this time both modules were instructed to use Xensockets to communicate over shared memory.

#### 4.2.2 Results

Figure 4.3 shows the first couple of JavaParty’s tests. The Xensocket implementation is limited to arrays with a maximal size of 4096 bytes, which is the size of a standard page on the test machine. The R-OSGi adaption of Xensockets was not yet able to circumvent this limitation. The graph shows that using shared memory results in significant performance gain. It is even faster than TCP within the same domain, where the connections still need to go through the network stack of the domain. The current prototype adaptation of Xensockets is a factor of 2.1 to 2.6 faster than TCP inter-domain communication.
Figure 4.1: specJVM98 on different JVMs, not on GuestVM
Figure 4.2: specJVM98 on different JVMs, including GuestVM
Figure 4.3: First couple of ping tests from JavaParty
Chapter 5

Conclusions

This work highlighted that demanding software, such as enterprise applications, profit from performance isolation. The presented prototype of a platform for Java based component systems can provide such isolation. R-OSGi together with shared memory communication and virtualization have proven to be a synergetic combination. However, there might still be some weak elements in the lightweight software stack and evaluations have shown that there is still ample space for improvements. This platform for component systems is not yet a framework for MacroComponents but it demonstrates that the tools to build such a MacroComponent framework are there.
Chapter 6

Future Work

Starting a component on the modular platform involves still manual setup of a virtual machine. R-OSGi could be leveraged to automate this process. R-OSGi would need to run in each administrative domain (domain0) to be able to setup new virtual machines for bundles. R-OSGi could provide services with varying setup policies like “install new bundles in an isolated VM on the machine with the most free resources” or “install on the system that has the longest idle periods”. OSGi bundles have a startlevel to enforce starting sequences. This is used by some implementations to break cyclic dependencies. The startlevel, which is an integer, could be reused to specify that bundles with the same startlevel should go into the same VM. Thus allowing tightly coupled bundles to run in the same VM. Deployment of the bundles to the virtual machine could also be done by R-OSGi. The Xen hypervisor provides not only isolation but also the freedom to migrate virtual machines from one physical machine to another. This possibility could be explored and used to dynamically adapt modular applications to the environment. If a component suddenly needs more resources, then the whole VM could be moved to a machine with more free resources. If the platform for component systems is managed by R-OSGi, then R-OSGi could also trigger migration as well as provide means of transportation for the VM images.

The presented prototype consists of modules and layers, but those need not be visible to users of this component system platform. Thus the whole software stack of this platform should be integrated in a single VM image. The Xensocket kernel modul should be integrated in GuestVM’s GUK kernel and it should be directly accessed by R-OSGi.

GuestVM and Concierge are both implemented in Java. This facilitates cross layer optimization. The Xensockets are also mere prototypes and would benefit from optimizations.
Acknowledgements

I would like to thank Jan Simon Rellermeyer for his guidance and advice.
Bibliography


Performance Isolation for Component Systems

TR-99-73 Sun Microsystems Research Laboratories, Mountain View (CA), (February 1999)


Performance Isolation for Component Systems


[34] Sun Microsystems, Sun SPOT, http://www.sunspotworld.com/

[36] Apache Foundation,

[37] Knopflerfish Org,
    Knopflerfish (OSGi), http://www.knopflerfish.org/

[38] Oscar, Oscar (OSGi), http://forge.ow2.org/projects/oscar/