DISSERTATION

Designing and Implementing a Secure, Portable, and Efficient Mobile Agent Kernel: The J-SEAL2 Approach

ausgeführt zum Zwecke der Erlangung des akademischen
Grades eines Doktors der technischen Wissenschaften

eingereicht an der Technischen Universität Wien
Fakultät für Technische Naturwissenschaften und Informatik

von

Walter Binder
Matr. Nr. 9200514
Braungasse 13, 1170 Wien

Wien, im April 2001

Abstract

Even though the benefits of mobile agents have been highlighted in numerous research works, mobile agent applications are not in widespread use today. For the success of mobile agent applications, secure, portable, and efficient execution platforms for mobile agents are crucial. However, available mobile agent systems do not meet the high security requirements of commercial applications, are not portable, or cause high overhead.

Currently, the majority of mobile agent platforms is based on Java. These systems simply rely on the security facilities of Java, although the Java security model is not suited to protect agents and service components from each other. Above all, Java is lacking a concept of strong protection domains that could be used to isolate agents.

The J-SEAL2 mobile agent system extends the Java environment with a model of strong protection domains. The core of the system is a micro-kernel fulfilling the same functions as a traditional operating system kernel: Protection, communication, domain termination, and resource control. For portability reasons, J-SEAL2 is implemented in pure Java. J-SEAL2 provides an efficient communication model and offers good scalability and performance for large-scale applications. This thesis explains the key concepts of the J-SEAL2 micro-kernel and how they are implemented in Java.

Keywords: Bytecode rewriting, Java, micro-kernel architectures, mobile agent systems, protection domains, resource control, security

Contents

Chapter 1

Overview

1.1 Introduction

Currently, an increasing number of research projects explores mobility in object-oriented systems. Mobile objects, usually referred to as mobile agents, are programs that are able to migrate in a network in order to optimize their consumption of resources, such as network bandwidth, or to accommodate to a changing environment. During migration, a mobile agent preserves some state of its execution. Mobile agent technology offers many advantages for distributed computing:

Mobile agents support resource aware computations. Object migration allows mobile agents to access necessary services locally. Therefore, expensive remote interactions, such as client-server communication over a network, can be minimized. Once a mobile agent has been transferred to a server, it may issue many requests locally at the server. In that way, the use of network bandwidth can be reduced significantly. For instance, frequently mobile agent technology is used to search in an environment, where information is stored in a distributed way (e.g., the Internet). An agent moves directly to an information provider in order to locally filter out the relevant information, which the mobile agent preserves in its state. Consequently, information that is not relevant to the agent's task is not transmitted over the network.

Other advantages of mobile computations include the support for off-line operation. Mobile agent technology allows to create autonomous mobile computations, which are able to survive temporal network failures and disruptions. Especially in the context of wireless networks, which suffer from low bandwidth, high error rates, and frequent disconnections when com-

pared to wireline networks, applications based on mobile agents may help to overcome the shortcomings of these networks. An agent covering a user's working set (i.e., the programs and data the user needs to continue working on a certain task) enables the user to complete a task without maintaining a permanent network connection. As a result, availability and usability of such applications are improved, whereas connection costs are reduced.

As a model for distributed computation, mobile agents ease load balancing and help to improve scalability and fault tolerance. Moreover, an agent-oriented programming model facilitates the design and implementation of complex distributed systems.

In order to enable agent mobility, dedicated execution environments – mobile agent systems – have to be developed and to be deployed widely. For the success of a mobile agent platform, a sound security model, portability, and high performance are crucial. Since mobile code may be abused for security attacks (unauthorized disclosure and modification of information, denial-of-service attacks, trojan horses, viruses, etc.), mobile agent platforms must protect the host from malicious (or badly programmed) agents, as well as each agent from any other agent in the system. In order to support distributed applications, mobile agent systems have to be portable and to offer good scalability and performance. As mobile agent applications typically are deployed in large-scale heterogeneous environments, such as the Internet, mobile agent platforms have to support a wide variety of different hardware platforms and operating systems. Therefore, portability is of paramount importance for the success of a mobile agent system.

However, most mobile agent systems fail to provide sufficiently strong security models, are limited to a particular hardware architecture and operating system, or cause high overhead. As a result, commercial applications based on mobile agent technology are not in widespread use today.

In contrast to popular mobile agent platforms, the design and implementation of the J-SEAL2 mobile agent system [5, 6] reconcile strong security mechanisms with portability and high performance. J-SEAL2 is a microkernel design implemented in pure Java[1] [20]; it runs on every Java 2 implementation (JDK 1.2 or higher) [34]. Considering the variety of hardware used for Internet computing, it is crucial that J-SEAL2 supports as many different hardware platforms and operating systems as possible. For this reason, J-SEAL2 does not rely on native code nor on modifications to the Java

[1] Java is a trademark of Sun Microsystems, Inc.

Virtual Machine (JVM) [26].

J-SEAL2 is a complete redesign of JavaSeal, a secure mobile agent system developed at the University of Geneva [42, 11]. JavaSeal extends the Java programming environment with a model of mobile agents and strong hierarchical protection domains. These extensions are based on a formal model of distributed computation, the Seal Calculus [43]. JavaSeal enables the expression and effective enforcement of security policies, but it incurs high overhead and does not scale well. Due to performance problems (e.g., inefficient communication between different protection domains, enormous agent startup overhead, etc.), JavaSeal is not suited for large scale applications.

J-SEAL2 is compatible with JavaSeal, but offers enhanced security mechanisms and significantly improved performance. J-SEAL2 provides resource control, a new communication model, a high-level communication framework built on top of the micro-kernel, a new component model for services, as well as a flexible and convenient configuration mechanism based on XML [10].

This thesis is structured as follows: In chapter 1 we give a short overview of the Java language [20] and the J-SEAL2 architecture. In section 1.2 we discuss characteristics of Java that can be used to build secure mobile agent systems, and explain how shortcomings of the language with respect to agent security can be overcome. Section 1.3 outlines the structure of the J-SEAL2 micro-kernel and shows an exemplary configuration of the system. Section 1.4 introduces the main abstractions of J-SEAL2 and summarizes the security properties of the kernel.

In chapter 2 we compare J-SEAL2 with related work in the area of Java operating systems. Furthermore, we discuss alternative approaches to resource control in Java, which however are not completely portable.

Chapter 3 is devoted to the protection domain model of J-SEAL2. We explain how to write kernel code in Java and how to achieve protection of individual applications executing within the same Java runtime environment. Furthermore, we address various issues regarding safe domain termination and resource reclamation.

Chapter 4 deals with the communication model of J-SEAL2. We start with summarizing the features of the communication model of JavaSeal [42, 11], which has many deficiencies and imposes an enormous overhead. Therefore, we designed a complementary communication mechanism in the J-SEAL2 kernel, which offers efficient inter-domain communication without sacrificing security. This new communication model enables efficient high-

level communication facilities built on top of the micro-kernel. We conclude this chapter with performance measurements comparing the various communication mechanisms available in J-SEAL2.

Chapter 5 provides an extensive discussion of the resource control model in J-SEAL2. We state our requirements and give a coarse overview of the resulting model. We present the resource control API and explain the implementation techniques we are resorting to. Initial performance measurements back our approach.

In the last chapter we give a glimpse on future investigations and summarize the current state of implementation of the J-SEAL2 kernel, as well as the main contributions of this thesis.

1.2 Mobile Agent Systems in Java

Recently, a large number of mobile agent systems based on Java [20] has emerged[2]. In fact, Java is a good choice for the implementation of execution environments for mobile agents, as it offers many features that ease the development of mobile agent platforms:

- The code of mobile agents has to be represented in a hardware-independent format in order to allow agents to migrate in a heterogeneous environment. The JVM specification [26] defines a *portable code* representation for Java programs (bytecode), which is independent of any particular hardware architecture. Therefore, using JVM bytecode to represent the code of agents enables code mobility, the basis for agent mobility.

- In addition to portable code, Java offers a *serialization* mechanism allowing to capture a mobile agent's state of computation before it migrates to a different host, and to resurrect the agent in the new environment. This kind of state transfer is known as weak mobility, because running threads are lost. The agent is responsible for defining the state of computation to be preserved during migration.

[2]For an (incomplete) list of different mobile agent platforms see *The Mobile Agent List* available via WWW at URL: `http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole/mal/mal.html`. Most of the systems presented there are based on Java.

- Since a mobile agent platform has to execute multiple agents and service components concurrently, it requires a multithreaded environment. Java supports *multithreading* and offers convenient mechanisms to implement various synchronization protocols.

- *Language safety* in Java [46] guarantees that the execution of programs proceeds according to the language semantics. For instance, types are not misinterpreted and data is not mistaken for executable code. In Java safety depends on the following techniques: *bytecode verification* to ensure that programs are well-formed, *strong typing* to guarantee that values are used according to their definition, *automatic memory management* (garbage collection) to prevent memory leaks and errors such as deleting live objects, and *memory protection* to prevent array and stack operations from overflowing. Language safety can be used as a basis to build secure execution environments for mobile agents.

- Java supports the dynamic loading and linking of code. *Class-loader namespaces* can be used in order to isolate the classes of the agent system and of different agents from each other. For instance, with the aid of class-loader namespaces it is possible to prevent agents from substituting system classes with their own code (trojan horses).

- *High performance* Java runtime systems are available for most hardware platforms and operating systems. Therefore, mobile agent platforms written in pure Java are highly portable and may exploit sophisticated compilation techniques and other optimizations provided by the underlying JVM in order to offer good performance and scalability.

Despite these advantages, the Java security model [34] is not suited to protect agents and service components from each other. Above all, Java is lacking a concept of strong protection domains that could be used to isolate agents. Because of pervasive aliasing (i.e., direct sharing of objects) in the Java Development Kit (JDK), there are no protection boundaries between different components. Furthermore, malicious agents can easily mount denial-of-service attacks against the platform, possibly even crashing the JVM. Moreover, if a misbehaving agent is detected, the platform does not guarantee that the agent can be safely removed from the system. Since the vast majority of current mobile agent platforms simply relies on

the insufficient security facilities of Java, these systems are not suited for commercial mobile agent applications.

Some researchers have shown that an abstraction similar to the process concept in operating systems is necessary in order to create secure execution environments for mobile agents [1]. However, proposed solutions were either incomplete or required modifications of the Java runtime system. In contrast, the J-SEAL2 mobile agent micro-kernel ensures important security guarantees without requiring any modifications to the underlying Java implementation. For portability reasons, J-SEAL2 is implemented in pure Java and runs every Java 2 platform.

In traditional operating systems the kernel is responsible for protecting processes from each other. A process cannot access a foreign memory region unless that region has been explicitly declared to be shared. Furthermore, the operating system offers some Inter-Process Communication (IPC) facilities, allowing for a controlled co-operation between different processes. When a process is terminated, the kernel ensures that it is removed from the system freeing all resources the terminated process possesses. In addition to this, the kernel must ensure that the termination of a process does not corrupt any shared system state. Finally, the operating system guarantees that a process can only use the resources (e.g., CPU time, memory) it has been given.

The J-SEAL2 micro-kernel fulfills the same role as an operating system kernel: It ensures protection of different agents and system components, provides secure communication facilities, enforces safe domain termination with immediate resource reclamation, and controls resource allocation.

1.3 J-SEAL2 System Structure

In this section we give a brief overview of the J-SEAL2 mobile agent platform. More information about the Seal model can be found in [43, 42, 11]. Details about the design and the implementation of J-SEAL2 are presented in the following chapters.

J-SEAL2 offers a model of nested protection domains. The J-SEAL2 kernel maintains a tree hierarchy of agents and service components. Each agent and service executes in a protection domain of its own, called a sealed object or *seal* for short. Apart from resource limitations, a parent seal may create an arbitrary number of children seals. The root of the tree hierarchy is RootSeal, which is responsible for starting system services.

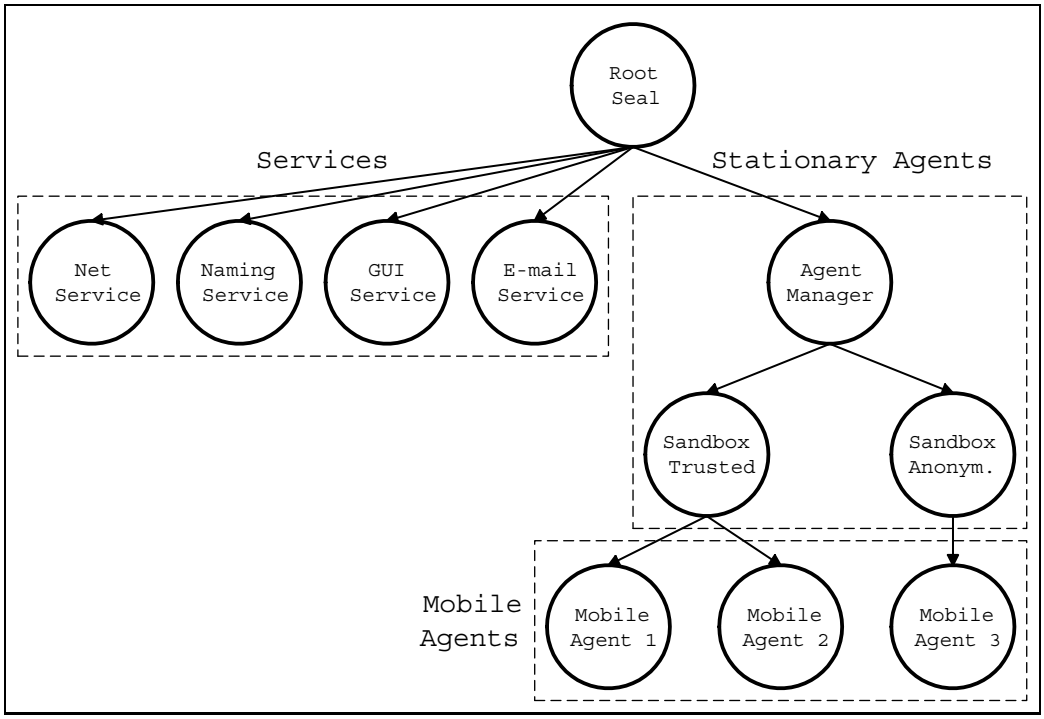

Figure 1.1: J-SEAL2 nested protection domains.

RootSeal reads a XML [10] configuration file describing the service infrastructure to be created. Due to a generic configuration format, each service may specify its own set of configuration parameters (nested parameter structures are supported as well). RootSeal executes a well-defined service registration protocol, where each service seal registers its interfaces in a local naming service maintained by RootSeal. Figure 1.1 shows some frequently used service seals:

- The *network service* receives mobile agents from other hosts and allows to send them out again. Depending on the application, the network service may implement some protocols to authenticate remote hosts. Furthermore, it is possible to install different network services in the same J-SEAL2 platform, which may be important if multiple mobile agent applications execute within a single J-SEAL2 installation. Unlike most other mobile agent systems, the J-SEAL2 kernel does not have any built-in networking support. Therefore, network communication is not further discussed in this thesis.

- The *naming service* provides access to a global service registry. If an agent requires a service not available locally, it contacts the global naming service in order to find out the Internet address (e.g., host name and port) of a remote J-SEAL2 installation offering the required service. Afterwards, the agent employs a network service for migration.

- The *GUI service* provides a simple window manager interface enabling agents to directly interact with users. The GUI service may be used by agents while they execute on the host of their client. In J-SEAL2 server installations the GUI service may be accessed only by dedicated stationary management and monitoring agents interacting with the system administrator.

- The *E-mail service* is used by agents performing some long lasting off-line operations on behalf of their users. For instance, consider a shopping agent searching for a certain product: The user may want the agent to keep an eye on the market for some days or weeks in order to wait for a cheap offer. The agent may use the E-mail service in order to inform the user of new interesting offers.

Having started all service components, RootSeal installs a stationary agent acting as a container for other agents. This stationary agent manager may create additional stationary sandbox managers as children seals, each of them responsible for different types of incoming mobile agents. Figure 1.1 shows a typical configuration, where the agent manager installs two sandboxes: One sandbox executes authenticated, fully trusted agents, while the other one is responsible for anonymous, potentially malicious agents.

The agent manager is granted access to the local naming service. Therefore, it may employ all services. Each seal may define different security policies for its nested domains. For instance, the agent manager may allow the sandbox of trusted agents to access arbitrary services, while the sandbox of anonymous agents may only use the network service in a restricted way.

1.4 J-SEAL2 Security Properties

The J-SEAL2 kernel isolates seals from each other. The kernel acts as a reference monitor ensuring that there is no direct sharing of object references with distinct seals. J-SEAL2 guarantees accountability, i.e., every object belongs

to exactly one protection domain. This feature eases memory accounting and protection domain termination.

Communication between distinct seals requires kernel primitives (see section 4), objects are passed always by deep copy within so-called *capsules*. In that way, the kernel prevents direct sharing of object references between different protection domains. This property is crucial for protection domain isolation, as aliasing between different domains would undermine protection.

J-SEAL2 offers two different communication mechanisms, *channels* and *external references*. Channels only support direct communication between seals that are neighbours in the hierarchy. Channel communication ensures that a parent seal is able to isolate a child completely from other domains. External references allow indirect sharing of objects by different seals, which enables efficient communication shortcuts in deep seal hierarchies. For security reasons, external references are tracked by the J-SEAL2 kernel and may be invalidated at any time. External references are the basis for efficient high-level communication frameworks.

Agents are not allowed to directly use functionality offered by the JDK classes. Instead of employing the JDK, agents have to contact corresponding service seals, succeeding only if all seals on the shortest path in the hierarchy between the agent and the service are granting access to that service. The JDK functionality available to individual service seals can be configured by the system administrator.

Seals are multithreaded protection domains. Every seal can run an arbitrary number of secure threads concurrently. The J-SEAL2 kernel ensures that a parent seal may terminate its children at any time. As a consequence, all threads in the child domain are stopped and all memory resources used by the child seal become eligible for garbage collection immediately.

Summing up, the J-SEAL2 kernel ensures the following important security properties [11]:

Confinement: A seal is isolated from other parts of the system. Its actions cannot affect other parts of the system.

Mediation: It is possible to interpose security code between a seal and the rest of the system, i.e., all messages from an untrusted seal can be intercepted and verified before they are forwarded to other seals.

Faithfulness: Code executed in a seal really belongs to that seal. It is impossible to force a seal to execute foreign code.

Chapter 2

Related Work

In this chapter we compare the J-SEAL2 mobile agent kernel with related work. We distinguish two broad categories of related work on improving the security of Java runtime environments: Java operating systems offering protection domains in order to isolate applications from each other, and Java-based systems which provide mechanisms for resource accounting or control, but do not enforce a strict separation of individual components.

2.1 Java Operating Systems

Java operating systems incorporate a process model in a Java environment, which allows multiple applications to execute concurrently but isolated within a single JVM. In contrast to multiple JVM processes running in different address spaces on a traditional operating system, a Java operating system may support very efficient communication mechanisms within the same address space (i.e., safe sharing between distinct protection domains). Furthermore, a single address space enables the sharing of frequently used classes, such as the JDK and tools like XML parsers [10]. Thus, the memory requirements and the overhead to start new applications are reduced significantly. In addition to these advantages, Java operating systems can be used on embedded or portable devices that do not offer operating system or hardware support to manage multiple processes.

In this section we differentiate the J-SEAL2 kernel from other Java operating systems regarding the supported communication models, the resource control facilities, the domain termination characteristics, and to which extent

the implementation is portable or not.

2.1.1 JavaSeal

JavaSeal [42, 11], developed by the Object System Group at the University of Geneva, is the first implementation of the Seal Calculus [43], a formal model for secure mobile computations in open network environments. Like J-SEAL2, JavaSeal is implemented in pure Java. It provides the same hierarchical model as J-SEAL2, even the APIs for domain creation and termination are compatible.

However, concerning the communication model, JavaSeal supports only synchronous message passing over named channels (see section 4.2). JavaSeal allows neighbour domains to exchange messages by deep-copy, but it does not offer any means for direct or indirect object sharing between different domains. Consequently, JavaSeal incurs high communication overhead in deep hierarchies. Conversely, J-SEAL2 supports external references (see section 4.4), an efficient but secure mechanism for indirect sharing.

While the hierarchical domain model of JavaSeal is a prerequisite for the resource control model presented in chapter 5, JavaSeal does not address denial-of-service attacks. In contrast to J-SEAL2, there is no accounting for physical and logical resources in JavaSeal. Also the formal model of JavaSeal, the Seal Calculus, does not deal with resource control.

JavaSeal aims at ensuring safe domain termination with immediate resource reclamation, but due to shortcomings in the design and implementation of JavaSeal, domain termination may corrupt the shared system state[1], and malicious domains may prevent their termination. Furthermore, in the JavaSeal implementation certain kernel objects encapsulating communication messages are shared between distinct domains for performance reasons. While these objects cannot be accessed by user code, direct sharing may prevent memory reclamation by the garbage collector after the termination of a protection domain. The careful design and implementation of J-SEAL2 encompasses several improvements to ensure that a parent domain can safely terminate its children at any time, and that the garbage collector is able to reclaim all allocated memory resources during the next garbage collection cycle.

[1] E.g., during domain termination a thread may be stopped while it is loading and linking a class. As a result, some structures inside the JVM may be corrupted.

2.1.2 KaffeOS

Our work on the J-SEAL2 mobile agent micro-kernel is related to work on protection in single-language mobile code environments. Especially the Utah Flux Research Group has worked on the design and implementation of secure single address space operating systems implemented in Java [3, 2].

KaffeOS [1, 2] is a Java runtime system which supports the operating system abstraction of processes to isolate applications from each other, as if they were run on their own JVM. Thanks to KaffeOS, a modified version of the freely available Kaffe virtual machine [45], it is possible to achieve resource control with a higher precision than what is possible with byte-code rewriting techniques, where, for example, memory accounting is limited to controlling the respective amounts consumed in the common heap, and where CPU control does not account for time spent by the common garbage collector working for the respective applications.

The KaffeOS approach should by design result in better performance, but is however inherently non-portable. This means that optimizations found in compilers and standard JVMs are not benefited from: in a recent publication [2] the authors report that, in absence of denial-of-service attacks, IBM's compiler and JVM [32] is 2–5 times faster than theirs.

2.1.3 Alta

Developed by the same team as KaffeOS, Alta [40, 3] is a prototype based on the Fluke hierarchical process model, and implemented on the Kaffe virtual machine [45]. The main differences with KaffeOS are that a single garbage collector is responsible for all applications, and that Alta entirely respects the hierarchical process model of Fluke by providing resource control APIs, whereas KaffeOS only retains a more implicit nested CPU and memory management scheme.

Like J-SEAL2, the Alta operating system is a micro-kernel design. Its hierarchical process model supports CPU control through CPU Inheritance Scheduling [18], where a process may donate some percentage of its CPU resources to nested child processes.

However, the Alta design cannot be implemented in pure Java. Alta relies on modifications to the JVM, whereas J-SEAL2 runs on every Java 2 implementation. We are convinced that the portability of a mobile agent platform is crucial for its successful deployment in large-scale commercial projects.

Furthermore, considering the enormous pace of new JVM implementations offering rapidly increasing performance, it is almost impossible to maintain a modified JVM offering sufficient performance.

2.1.4 J-Kernel

J-Kernel [44] is a Java micro-kernel supporting multiple protection domains. In J-Kernel communication is based on capabilities. Java objects can be shared indirectly by passing references to capability objects. However, J-Kernel is lacking the hierarchical model of J-SEAL2. Moreover, in J-Kernel cross-domain calls may block infinitely and may delay protection domain termination. J-Kernel supports per thread memory accounting via bytecode rewriting [14]. Like J-SEAL2, J-Kernel is implemented completely in Java, only CPU accounting requires native code.

2.1.5 Luna

Luna [22, 21] is a Java extension that provides a task model for Java based on a type system, which distinguishes between task-local pointers and remote pointers shared between multiple tasks. Access to remote pointers is controlled by permits that can be revoked at any time. When a task is terminated, Luna revokes all remote pointers into that task. While J-SEAL2 supports only coarse-grained indirect sharing between different domains with the aid of external references (see section 4.4), remote pointers in Luna enable fine-grained direct sharing of individual objects between distinct tasks. However, Luna is not portable, as it is based on an extension to a Java runtime system.

2.1.6 NOMADS

NOMADS [37] is a mobile agent system which has the ability to control resources used by agents, including protection against denial-of-service attacks. The NOMADS execution environment is based on a Java-compatible VM, the Aroma VM, a copy of which is instantiated for each agent. There is no resource control model or API in NOMADS; resources are managed manually, on a per-agent basis or using a non-hierarchical notion of group. Relying on a specialized VM, it follows that the overhead is smaller than with our approach; currently, CPU control is however not implemented.

2.2 Resource Control in Java

There are several lines of research, where libraries, environments, and analysis tools have been designed that can be exploited to prevent denial-of-service attacks.

2.2.1 JRes

JRes [14] is a resource control system which takes CPU, memory, and network resource consumption into account. The resource management model of JRes works at the level of individual Java threads; in other words, there is no notion of application as a group of threads, and the implementation of resource control policies is therefore cumbersome. JRes is a pure resource accounting system and does not enforce any separation of domains; covering this other aspect is the goal of J-Kernel [44], a complementary project of the same research team (see section 2.1.4).

For its implementation, JRes does not need any modification to the JVM, but relies on a combination of bytecode rewriting and native code libraries. To perform CPU accounting, the approach of JRes is to make calls to the underlying operating system, which requires native code to be accessed[2]. For memory accounting, it essentially uses bytecode rewriting, but still needs the support of a native method to account for memory occupied by array objects. Finally, to achieve accounting of network bandwidth, the authors of JRes also resort to native code, since they swapped the standard `java.net` package with their own version of it.

2.2.2 Real-time Extensions for Java

The Real-Time for Java Experts Group has published a proposal to add real-time extensions to Java [9]. One important focus of this work is to ensure predictable garbage collection characteristics in order to meet real-time guarantees. For instance, the specification provides for several memory management schemes, such as areas with limited lifetime or bounded allocation rates, which could be implemented – or at least simulated – with the resource control model described in this thesis.

[2]More precisely, CPU accounting in JRes is based on native threads, a feature not supported by every JVM.

Another real-time system, PERC [28], extends Java to support real-time performance guarantees. To this end, the PERC system analyzes Java bytecodes to determine memory requirements and maximal execution times, and feeds that information to a real-time scheduler. The objective of real-time systems is to provide precise guarantees e.g. for worst-time execution; our focus, on the other hand, is on computing approximated resource consumptions in order to prevent denial-of-service attacks. We are more interested in the relative values of applications, and less in absolute figures. This is confirmed by the fact that we are not trying to estimate their real CPU consumption, but rather to compare the respective number of executed bytecodes.

2.2.3 Java Profilers

Profilers constitute another class of tools that have many things in common with resource control: both intend to gather information about resource usage. Profilers however are designed to help developers optimize the efficiency of their applications, and not to externally control their resource consumption. The Java Virtual Machine Profiling Interface (JVMPI) [36] is an API created by Sun; it is a set of hooks to the JVM which signals interesting events like thread start and object allocations.

Java Usage Monitor (JUM) [16] is a tool which builds upon JVMPI to help the developer determining how much CPU is consumed by the different threads of an application and how much memory they use. JUM needs native code to obtain information from the underlying operating system about how CPU time is allocated, and is therefore not portable.

Interestingly, JUM is able to also account for objects allocated by native code. However, JUM is not able to enforce memory limits. While J-SEAL2 allows for exact pre-accounting of memory resources, where an overuse exception is generated before a thread exceeds its memory limit, a resource control mechanism based on JUM can only react after a memory overuse is detected. In addition to these limitations, JVMPI is an experimental interface, it is not yet a standard profiling interface.

2.2.4 Economic Models

Finally, we mention some approaches that rely on economics-based theories, using virtual currencies to achieve natural load-balancing of concurrent

applications, as well as recycling of unused resources in open distributed environments, with the anticipated side-effect of preventing denial-of-service attacks [39]. Our focus is however more on how to implement the basic resource accounting mechanisms on a specific platform, Java, than on the design of high-level – and distributed – resource allocation policies. Nevertheless, whereas the spirit of this thesis is rather conservative, it does not exclude the application of the presently described techniques to the implementation of open computational markets.

Chapter 3

Protection Domains

3.1 Introduction

The core of the J-SEAL2 mobile agent system is a compact micro-kernel, which offers a minimal set of abstractions necessary to program secure agent environments: protection domains for agents and services (seals), concurrent activities (threads), as well as communication facilities (channels and external references).

In this chapter we present the protection domain model of J-SEAL2, which resembles the process model of a traditional operating system. Protection domains in J-SEAL2 enable the isolation of multiple applications running in the same JVM from each other. Like an operating system, the J-SEAL2 micro-kernel employs a user/kernel distinction in order to maintain system integrity in the presence of protection domain termination.

In section 3.2 we state requirements for kernel code written in Java and show how kernel entry and exit can be implemented efficiently. The following two sections deal with protection and protection domain termination. In each section we state our requirements for the J-SEAL2 kernel and discuss various implementation issues and techniques that help to meet these requirements efficiently.

3.2 Kernel Code

Like a traditional operating system, J-SEAL2 is divided into user and kernel parts. While user mode and kernel mode do not indicate a change in hardware

privileges, they provide distinct environments regarding protection domain termination. In this section we discuss various issues arising from the design and implementation of an operating system kernel in Java.

3.2.1 Requirements

Since seals are multithreaded and certain kernel structures must be accessible from different seals, a kernel synchronization protocol must ensure proper synchronization and prevent deadlocks. Kernel code must not synchronize on objects that are accessible by agents. Otherwise, agents could cause kernel code to block infinitely (i.e., a thread of an agent could initiate a kernel operation after a different thread has obtained a lock on an object, which the kernel also needs). Instead, kernel code shall lock only internal structures, such as private members of kernel abstractions.

Kernel operations are to be performed atomically: they must either succeed or leave the kernel state unchanged. Kernel operations must take care not to cause any uncaught exceptions. In particular, special attention has to be paid to exceptions that can occur asynchronously, such as e.g. `ThreadDeath`, `OutOfMemoryError`, or `StackOverflowError`.

`ThreadDeath` is thrown, if a thread stops another one with the aid of `Thread.stop()`. The stopped thread immediately exits all monitors it holds and throws `ThreadDeath`. Since this may leave objects in an inconsistent state, `Thread.stop()` has been deprecated in the Java 2 platform. However, because the Java 2 platform does not offer any alternative mechanisms for thread stopping, protection domain termination must be based on `Thread.stop()`. The kernel has to ensure that stop requests are deferred while a thread is accessing kernel structures.

`OutOfMemoryError` is thrown whenever the virtual machine runs out of memory and the garbage collector fails to reclaim enough memory. Kernel operations must be designed to avoid `OutOfMemoryError` after the operation has modified some kernel state. A simple solution is to allocate all objects that might be required (worst case estimation) in advance before any changes are made. Following this approach, kernel structures should not employ Java utility classes, such as the collections framework, since these classes allocate objects on demand.

3.2.2 Implementation Issues

Operating systems employ a privileged processing mode for kernel operations. Only a process executing in kernel mode has access to all processor instructions and kernel internals (special memory regions). The J-SEAL2 kernel uses a similar distinction: When a thread initiates a kernel operation, it enters a privileged kernel mode. When the kernel operation completes (succeeds or fails), the thread leaves the kernel mode and continues execution in user mode.

The main purpose of the kernel mode in J-SEAL2 is to prevent a thread from being stopped while accessing kernel internals. Stop requests are deferred until the thread to be stopped leaves the kernel. In addition to deferring stop requests, kernel mode is used to synchronize primitives affecting the J-SEAL2 kernel abstractions, such as protection domain creation and termination, thread creation, as well as communication requests.

We distinguish between exclusive and shared kernel mode. A thread entering exclusive kernel mode is blocked until no other thread is executing in the kernel. While a thread is executing in exclusive kernel mode, no other thread can enter the kernel. Protection domain termination is always performed in exclusive kernel mode. Therefore, threads executing in kernel mode are guaranteed not to be stopped asynchronously.

The J-SEAL2 kernel uses a single-writer/multiple-reader lock for controlling access to the kernel: Entering exclusive kernel mode corresponds to acquiring the write lock, whereas shared kernel mode requires a read lock. The lock implementation ensures that a thread waiting for the write lock will enter the kernel before threads waiting for a read lock. This property makes sure that domain termination cannot be delayed infinitely.

The lock object must be implemented very carefully: When a thread is trying to obtain a read lock or the write lock, it is still executing in user mode. Entering kernel mode means that the lock has been obtained successfully. Thus, the state of the lock object must not be changed before the required lock is really available, otherwise this would be a forbidden modification of a kernel object (the lock object) by a thread executing in user mode. As a result, sophisticated queuing techniques to control the order of kernel entries cannot be implemented easily, since enqueuing a request is a modification of kernel structures.

Since class-loading affects the internal state of the Java runtime system,

we must ensure that class-loading always occurs in kernel mode. A thread must not be stopped while it is loading a class, since this might corrupt some internal structures of the JVM. However, in general class-loading occurs asynchronously, dependent on the virtual machine implementation. Therefore, we do not know whether a class-loading thread already executes in kernel mode or not.

For this reason, the J-SEAL2 kernel offers a conditional kernel enter operation: A shared lock is only obtained, if the requesting thread is not yet executing in kernel mode. Implementing this operation requires keeping track of all threads that are executing in kernel mode. Because entering kernel mode is a very frequent operation (for instance, each communication involves at least one switch into kernel mode), adding threads to and removing threads from the set of threads executing in kernel mode must be highly optimized.

Although kernel entry and exit are extremely frequent operations, measurements indicate that less than 3% of the overall CPU time is spent for obtaining and releasing kernel locks. The benchmark measures communication latency and throughput in deep seal hierarchies, thus it shows the highest possible kernel entry/exit frequency (worst case).

Because operations requiring exclusive kernel mode are not executed frequently, the kernel lock does not become a significant performance bottleneck. Resource control (see chapter 5) ensures that an agent cannot enter the kernel arbitrarily. Therefore, the kernel lock cannot be abused for denial-of-service attacks.

3.3 Protection

The kernel of a traditional operating system ensures that a process can only access its own memory pages. The operating system kernel relies on the memory management unit of the processor in order to detect illegal memory accesses. A mobile agent kernel has to enforce similar protection. The kernel must protect itself as well as each agent from any other agent in the system.

3.3.1 Requirements

Language safety in Java, a combination of strong typing, memory protection, automatic memory management, and bytecode verification (see section 1.2), already guarantees some basic protection, as it is not possible to forge object

references. However, language safety itself does not guarantee protection in a mobile agent execution environment. Pervasive aliasing in object-oriented languages leads to a situation where it is impossible to determine which objects belong to a certain agent and therefore to check whether an access to a particular object is permitted or not. It is crucial to introduce the concept of strong protection domains, similar to the process abstraction in operating systems.

A protection domain draws a boundary around a component (i.e., a mobile agent or a service). It encapsulates the set of classes required by the component, some concurrent activities (threads), as well as all objects allocated by these threads. In general, threads shall only execute in their own protection domain. Special precaution is necessary to allow threads to cross domain boundaries. Furthermore, the kernel must prevent object references from being passed over protection domain boundaries. The kernel ensures that an object reference exists only in a single domain. This property is very important for memory accounting, too.

Each protection domain has associated its own set of classes. In general, different components must not share the same classes, since this would mean to share also the static variables in these classes (i.e., shared static variables would introduce aliasing of object references between different domains, or even worse, if static variables were not final, they could be used as covert communication channels the kernel could not control). However, some classes from the JDK must be shared by all components in order to ensure correct function of the JVM. Nevertheless, mobile agents must not employ JDK classes comprising functionality that undermines protection. For this reason, extended bytecode verification of agent classes is necessary.

Another issue to be addressed by a mobile agent system is protection of resources, such as files or network ports. While in monolithic operating systems the kernel usually deals with resource protection, micro-kernel architectures simply ensure that security policies can be implemented at a higher level. Similarly, a mobile agent micro-kernel does not have to deal with security policies. Rather, it must make sure that only privileged domains can access system resources. For Java, this means that agent domains must not have access to certain core packages, such as `java.io` or `java.net`. Such restrictions can be enforced by extended bytecode verification.

3.3.2 Implementation Issues

3.3.2.1 Class-loading

The J-SEAL2 kernel employs a separate class-loader namespace for each protection domain. A global configuration defines the set of classes to be shared by all domains. These classes are loaded by the JVM system class-loader. All other classes are loaded by the protection domain class-loader (replicated classes).

In order to ensure proper function of the Java runtime system, all JDK classes are shared. This does not introduce security problems, as the extended J-SEAL2 verifier assures that agents cannot use dangerous JDK functionality. Since replicating classes limits performance and increases agent startup overhead as well as memory consumption (above all, the same methods are compiled multiple times), the J-SEAL2 kernel is designed to minimize replicated kernel classes. In the current implementation only three small classes from the communication subsystem are replicated. This is necessary to ensure that serialized object graphs received by a protection domain are resurrected using the class-loader of the receiving domain.

Agent classes as well as classes from the J-SEAL2 library are replicated. To minimize the overhead of replicating library classes, the J-SEAL2 class-loader can be configured to cache the class-files residing in certain library packages. Since loading a class-file from disk is the most significant performance bottleneck, a proper caching configuration yields a speedup in agent creation by more than factor 2.

3.3.2.2 Extended Bytecode Verification

Agent class-files are verified by a special J-SEAL2 verifier in order to ensure that the agent does not use certain JDK and kernel classes. By this means the kernel protects itself and the underlying JVM from being corrupted by malicious or badly programmed agents. Each protection domain can have its own directives declaring which classes may be referenced. Directives include the following types of restrictions:

- Access can be restricted to certain packages, eventually including sub-packages.
- Access to individual classes can be allowed or forbidden.

- Access to individual class members can be allowed or forbidden.
- Extension of non-final classes can be prohibited.
- Agent classes must reside in a particular package or in a subpackage thereof.

The possibility to prevent the extension of certain classes and to control access at the level of individual class members helps to structure the J-SEAL2 kernel in a clean way. For example, we used multiple packages to separate different parts of the kernel. Public access modifiers were necessary to allow the interoperation of distinct kernel components. The directives ensure that agents cannot access certain kernel internals that had to be declared public for software engineering reasons. More generally, we think it is important to distinguish between software engineering practices and security engineering techniques: Java access modifiers and subtyping are very useful for software engineering purposes, whereas security engineering takes place in the specification of directives.

To ensure that a given class-file does not violate a particular set of directives it is sufficient to verify that the constant pool of the class-file does not refer to forbidden classes or members and that extension of the superclass is not prohibited. Each member (field, method, constructor) that is accessed by a method/constructor of the verified class has an entry in the class-file constant pool. Thus, it is not necessary to verify method/constructor code.

The verification algorithm consists of two passes: In the first pass, the constant pool is parsed and a constant pool representation is created. In the second pass, the constant pool representation is scanned for class references, member references, and type signatures. Class references as well as class names in type signatures must denote allowed classes. Member references have to specify allowed members. An efficient array representation of the constant pool as well as an optimized internal representation of the verification directives help to minimize verification costs.

Benchmarks measuring agent startup overhead indicate that verification overhead is less than 9% of the CPU time spent for class-loading. These measurements also include the costs for additional verification to ensure proper domain termination (see section 3.4).

3.4 Domain Termination

Operating systems provide means to terminate running processes. All memory resources the terminated process had allocated before become available to other processes. The operating system kernel must ensure that neither its own resources nor any other shared resources are corrupted when a process is killed.

3.4.1 Requirements

When a protection domain is terminated, all threads belonging to that domain have to be stopped. In Java the only means to stop a running thread asynchronously is `Thread.stop()`. However, this operation has been deprecated in the Java 2 platform, as it is inherently unsafe. When a thread is stopped, it exits all monitors immediately and throws `ThreadDeath`. As a consequence, objects may be left in an inconsistent state.

In chapter 3.2 we have already stated requirements for kernel code to ensure that shared resources, such as internal structures of the kernel and of the Java runtime system, cannot be corrupted when a thread is stopped asynchronously. The idea is to defer stop requests if the thread to be stopped is accessing the kernel. A simple solution is to ensure that no other thread can access the kernel while a thread is stopping another one.

When a thread is stopped, there is no guarantee that the stopped thread will really terminate. Depending on the executed code, `ThreadDeath` might be caught or a `finally{}` clause might execute an infinite loop. Since the kernel must ensure immediate resource reclamation when a protection domain is terminated, special verification is necessary to ensure that agent code cannot prevent or delay domain termination. Thus, exception handlers of agents must immediately rethrow caught `ThreadDeath` exceptions. J-SEAL2 offers a class-file rewriting tool to modify exception handlers accordingly. At runtime the extended bytecode verifier simply checks whether all agent classes have been rewritten correctly.

In addition to these restrictions, agents must not define finalizers or class finalizers. These special methods are invoked by the garbage collector before an object or a class is reclaimed. If these methods contained some infinite loops, they would hang up the whole system.

3.4.2 Implementation Issues

Safe thread stopping is achieved through special kernel entry and exit sequences. A thread terminating a protection domain enters exclusive kernel mode. It is blocked until all other threads have left the kernel. Terminating a protection domain is an atomic kernel operation. All threads belonging to the domain are stopped within the same kernel operation. Since the threads to be stopped cannot enter the kernel, this approach enforces safe domain termination. In order to prevent the corruption of internal structures inside the JVM, class-loading always occurs in kernel mode. Details about the kernel mode can be found in section 3.2.

In order to prevent agents from delaying their termination, the J-SEAL2 verifier ensures that agent methods do not use forbidden Java constructs. The method signatures of all defined methods must be different from `finalize()` and `classFinalize()`. Furthermore, the exception tables of all methods are examined in order to determine the types of caught exceptions. Agents are not allowed to catch `ThreadDeath`. If an exception handler catches a supertype of `ThreadDeath` (i.e., `Throwable` or `Error`), the handler code has to determine whether the actual type of a caught exception is `ThreadDeath`. In this case, the handler must rethrow `ThreadDeath` immediately.

The JVM supports a special catch type in the exception table to indicate that all exceptions are caught by a particular exception handler. Java compilers use this feature to compile `finally{}` clauses and `synchronized{}` statements [26]. The J-SEAL2 verifier ensures that these special exception handlers include code to rethrow a caught `ThreadDeath` exception immediately, too.

Because exception handlers catching all exceptions are not present in the original Java code, we have implemented a bytecode rewriting tool to process agent classes generated by standard Java compilers. This tool has to be used by J-SEAL2 programmers before they package their agents. It examines all exception handlers that could potentially catch `ThreadDeath` exceptions and inserts a short bytecode sequence (4 JVM instructions) to rethrow a caught `ThreadDeath` exception immediately. Our implementation is based on the bytecode engineering library by Markus Dahm [15].

The rewritten exception handlers rethrow `ThreadDeath` exceptions before any locks are released, because the `monitorexit` instruction of the JVM might throw `NullPointerException` or `IllegalMonitorStateException`.

However, since we are rethrowing `ThreadDeath` before releasing locks, an eventually locked object will never be unlocked and could cause a deadlock, if another thread tried to lock it later. Since there is no direct sharing of objects between different protection domains and because kernel code must not lock objects that are accessible by agents (see section 3.2), only objects belonging to the terminated agent may be left in a locked state. This is not a problem, since the agent is removed from the system anyway.

The J-SEAL2 verifier simply checks whether all agent classes have been rewritten accordingly. Benchmarks show that our verification mechanism does not introduce much overhead at runtime. Even for complex agents the total verification costs, including constant pool verification as described in section 3.3, does not exceed 10% of the total time for class-loading.

Chapter 4

Communication

4.1 Introduction

In operating systems co-operating processes can exchange messages through Inter-Process Communication (IPC) facilities provided by the kernel. Process protection ensures that IPC is the only means to exchange information over process boundaries. As a mobile agent kernel isolates agents from each other, it must also provide some means for inter-agent communication in order to allow agents to collaborate.

As we have stressed in the previous chapter, a secure mobile agent kernel has to provide strong protection domains. An agent executes within a protection domain, it is isolated from the rest of the system. If agents need to exchange some messages, all communication partners have to ask the kernel to establish a communication channel. The kernel ensures that communication partners can only access certain channels if they have the necessary permissions.

The J-SEAL2 kernel offers two different communication facilities, channels and external references. In both cases, messages are passed by value. The kernel creates a deep copy of the message before it passes it to the receiver. In that way, the kernel prevents direct sharing of object references between different protection domains. This property is crucial for protection domain isolation, as aliasing between different domains would undermine protection. Furthermore, resource accounting is largely simplified by the fact that every object reference exists in only a single protection domain.

This chapter is organized as follows: In section 4.2 we summarize the features of the channel communication mechanism, which corresponds to the

communication model in JavaSeal [42, 11]. In section 4.3 we discuss the severe shortcomings of the channel mechanism, which are solved by the new external reference communication model presented in section 4.4. The following section deals with issues arising from the implementation of channels and external references in Java. In section 4.6 we outline Inter Agent Method Calling (IAMC), a high-level communication protocol implemented efficiently on top of the J-SEAL2 micro-kernel. Finally, in section 4.7 we compare the performance of the different communication facilities in J-SEAL2.

4.2 Channels

In the JavaSeal kernel [42, 11] synchronous message passing through named channels is the only inter-agent communication mechanism. This model only supports direct communication between seals that are neighbours in the seal hierarchy. If two neighbour seals issue matching send and receive communication offers, the kernel passes the message from the sender to the receiver. Details about the channel matching algorithm can be found in [43, 42, 11].

With the aid of channel communication it is possible to achieve complete mediation. This means that it is possible to intercept all messages going in and out of an agent. Channel communication ensures that a parent seal is able to isolate a child completely from other seals.

J-SEAL2 supports the same channel operations as JavaSeal, but offers some improvements, such as optional timeouts for all synchronous channel primitives, as well as an asynchronous send operation.

4.3 Limitations of Channels

Even though channel communication is simple and conceptually clear, it is the most significant performance bottleneck inherent to the JavaSeal architecture.

If a seal wants to talk to another one, which is neither parent nor direct child, intermediate seals have to actively take part in the communication. Since threads cannot cross seal boundaries, communication requires dedicated threads in intermediate seals for message routing. Therefore, communication between seals involves thread switches proportional to the communication partners' distance in the seal hierarchy. Furthermore, every com-

munication act requires entering the JavaSeal kernel, which is also expensive due to synchronization.

Analyzing applications using JavaSeal revealed that for the vast majority of communications intermediate seals do not interpose any security policy, but they only forward communication requests. For instance, an agent manager seal grants a child agent access to a service. For this purpose, the agent manager has to start a thread receiving the child agent's messages from a certain channel and forwarding them to the service.

In addition to this inefficiency, the blocking nature of channel communication complicates the JavaSeal programming model significantly. The failure of a communication partner results in the deadlock of the other partner, unless channel communication is used very carefully. In effect, each seal has to run dedicated supervisor threads stopping those threads that are blocked executing channel primitives for a long period of time. This programming technique is error-prone, cumbersome, and incurs high overhead. Furthermore, it is difficult to determine useful values for timeouts, since they also depend on the varying system load.

The J-SEAL2 communication model is designed to overcome all of these deficiencies inherent to JavaSeal channel communication.

4.4 External References

Inside a single JVM method invocation on objects enables extremely fast communication between different software components without involving any thread switches. However, direct sharing of object references with distinct protection domains breaks the security properties mentioned before and complicates per protection domain resource accounting (CPU time and memory allocation) enormously.

The problem is that an object reference once handed out to a foreign protection domain can be neither retracted nor invalidated. While the reference is alive, it prevents the shared object from being reclaimed by the garbage collector. Furthermore, a thread calling methods of a shared object may be stopped asynchronously leaving the shared object in an inconsistent state.

The J-SEAL2 kernel introduces external references in order to overcome the communication inefficiency of JavaSeal without sacrificing security. Ex-

ternal references allow indirect sharing[1] of objects between different seals that are not necessarily neighbours in the hierarchy. An external reference acts as a capability to invoke methods on a shared object. A seal creates an external reference for some object and passes it out to another seal in order to share the object. External references encapsulate references to shared objects. They are tracked by the J-SEAL2 kernel and may be invalidated at any time deleting the encapsulated object references.

When an external reference is passed to another seal, the receiver gets a copy of the communicated external reference. The sender may invalidate that copy (as well as, in a synchronized way, recursively all further copies of that copy) at any time with immediate effect, i.e., threads calling methods through the copy immediately leave the callee's protection domain and throw an appropriate exception in the caller's domain. This property clearly distinguishes external references from capabilities in the J-Kernel [44]. Details of the external reference communication model are presented in the following subsections.

4.4.1 Terminology

We use the following terms to describe the semantics of external reference communication:

Shared object: A Java object (of arbitrary type) for which an external reference has been created.

Owner of a shared object: The seal which has created an external reference in order to share an object maintained by the seal.

Strong reference: A reference to a Java object. While there are strong references to an object, the object cannot be garbage collected.

Weak reference: A Java 2 reference object encapsulating an object reference [34]. Weak references do not prevent objects from being reclaimed by the garbage collector.

[1] The properties of different sharing models (copying, direct sharing, and indirect sharing) are explained in [3].

4.4.2 Properties of External References

In the following we explain the semantics of external references in J-SEAL2. We define rules for the creation, scope, and copying of external references, as well as for method invocation and for external reference invalidation.

Creation

CR-1: External references are created explicitly for objects to be shared with other seals. Initially, an external reference is valid (i.e., it can be used to invoke methods on the shared object).

CR-2: External references encapsulate strong references to shared objects. As long as strong references to a valid external reference are maintained, the shared object behind the external reference is kept alive. This property ensures that the owner of a shared object need not take care to keep the shared object alive. If it has created and passed out an external reference, the shared object cannot be garbage collected as long as the external reference is alive and remains valid.

CR-3: The J-SEAL2 kernel keeps track of all external references available in each seal. This property is important for implicit external reference invalidation during seal termination (see invalidation rule I-5). In order not to prevent external references from being reclaimed, the J-SEAL2 kernel employs weak references to track external references.

Scope

S-1: External references are completely decoupled from the seal hierarchy. Seals need not be in any particular relationship (like parent–child) in order to share objects with the aid of external references.

S-2: External references cannot be used as remote references, they are valid only within the J-SEAL2 platform where they have been created. When a seal is wrapped (serialized for transmission over the network), it loses all external references it holds.

Copying

CO-1: External references can be passed around in the seal hierarchy. They can be contained in capsules, which requires a dedicated copying algorithm for external references.

CO-2: When an external reference is copied, the copy is registered in the original external reference. We call the original external reference *ancestor* of the copy, whereas the copy is denoted as *descendant* of the original. An external reference may have any number of descendants. A copy always has exactly one ancestor. Only external references that are created explicitly for a shared object have no ancestor (see creation rule CR-1). Thus, an external reference and its direct and indirect descendants form a tree hierarchy decoupled from the seal hierarchy.

CO-3: Since an ancestor must not prevent its descendants from being reclaimed by the garbage collector, the ancestor employs weak references to the descendants. On the other hand, a descendant uses a strong reference to its ancestor preventing the ancestor from being garbage collected as long as the descendant is alive. As we will see in the section about external reference invalidation, it is crucial that the complete ancestor path of an external reference remains accessible while the external reference itself is alive.

CO-4: If a seal receiving an external reference already maintains an ancestor of the received external reference, it reuses the ancestor.[2] This property ensures that the ancestor path of an external reference is acyclic; it is closely related to the shortest path in the hierarchy to the owner of the shared object. We will see the importance of this property in the section about external reference method invocation. Note that if

[2]Copying rule CO-4 allows us to use external references as capabilities that cannot be forged. For instance, consider a GUI window manager service offering operations for window opening and closing. The open primitive returns an external reference to a new window, which can be used to manipulate the window contents. Furthermore, this external reference acts as a capability to close the corresponding window. The close operation requires an external reference to an open window as an argument. The implementation of the window manager service may use Java reference comparison in order to decide whether a given external reference refers to an open window. Language safety in Java [46] guarantees that it is not possible for any seal to guess a capability for a window belonging to another seal. Therefore, a seal cannot close the windows of other seals.

an external reference is passed to a neighbour seal, the 'shortest path property' guarantees that it is sufficient to check whether the receiving seal already maintains the external reference's ancestor (if it has an ancestor). Therefore, ensuring that the ancestor path has no cycles imposes only minimal overhead.

Method invocation

M-1: An external reference allows threads to invoke methods on a shared object concurrently, no matter in which seal the shared object resides. The caller has got to specify a method signature and to provide a capsule containing the arguments. The result is encapsulated, too. In effect, using capsules for arguments and for results ensures that there is no direct sharing of object references with different protection domains.

M-2: External references contained in an argument or result capsule[3] are treated specially: In order to ensure that the ancestor path of an external reference is related to the shortest path in the hierarchy between the owner of the shared object and the seal receiving the external reference, the J-SEAL2 kernel simulates a step-by-step forwarding of the external reference on that shortest path.

M-3: If an external reference is invalidated (see the invalidation rules below), threads calling through that external reference immediately leave the callee seal (i.e., the owner of the shared object) and throw an appropriate exception in the caller seal. This property is crucial in order to allow immediate memory reclamation when the owner of the shared object is removed from the hierarchy (see invalidation rule I-5). Note that this property allows shared objects to offer blocking operations without delaying protection domain termination.

M-4: While a thread executes a method of a shared object in a foreign seal, the reference to the thread object is not available. This property is an important detail, since it prevents the shared object from storing the

[3]Note that allowing external references to be passed over existing external reference connections (either as method argument or as return value) enables callback interfaces. For instance, in J-SEAL2 the local naming service is accessed via external references. The result of a successful query is an external reference to the requested service.

thread reference in its state (i.e., direct sharing with different seals) and from manipulating (e.g., stopping) that thread.

M-5: While a thread executes a method of a shared object, the seal the thread belongs to may stop the thread asynchronously. Therefore, the shared object must be designed in a way that does not allow a calling thread to leave the shared object in an inconsistent state. Currently, there are two options to ensure the consistency of a shared object: The shared object may use 'self-communication' with channel primitives [43, 42, 11] in order to dispatch a request to a thread belonging to the owner of the shared object, or it may employ Inter Agent Method Calling (IAMC), a high-level communication protocol implemented on top of external references (see section 4.6).

Invalidation

I-1: A seal may invalidate an external reference it holds at any time. This operation atomically invalidates all direct and indirect descendants.

I-2: A seal may atomically invalidate all external references it holds together with their descendants. This operation explains why the J-SEAL2 kernel keeps track of all external references in each seal (see creation rule CR-3).

I-3: Having passed out an external reference to a particular neighbour, a seal may atomically invalidate that copy[4] and its descendants. That is, a seal passing out an external reference has always the right to invalidate the forwarded external reference.

I-4: A seal may atomically invalidate all copies of external references passed out to a certain neighbour seal.

I-5: When a seal is removed from the hierarchy, invalidation operation I-2 is triggered implicitly[5] in order to ensure complete and immediate

[4]Note that passing out a an external reference does not imply that the receiver gets a copy of the external reference. If it already maintains an ancestor, it reuses the ancestor. However, the invalidation operation only works on copies.

[5]Note that also descendants of external references that had been passed through the terminated seal are invalidated, because the ancestor path is cut.

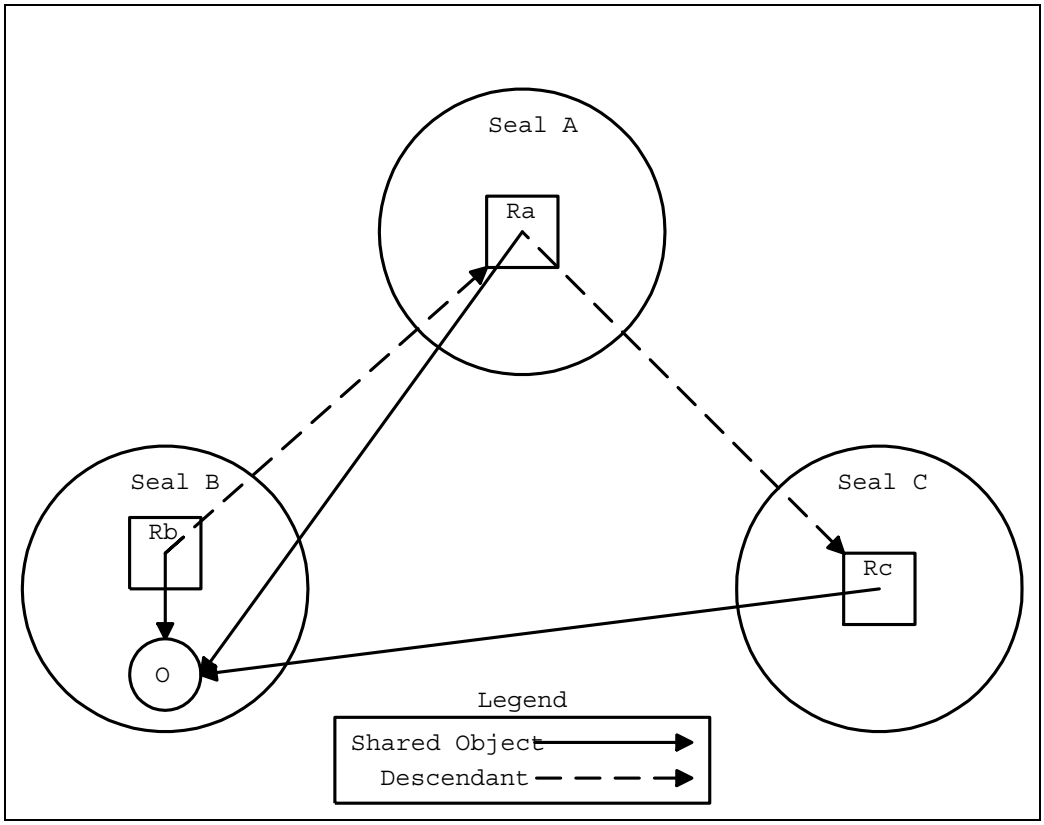

Figure 4.1: J-SEAL2 external references.

memory reclamation. That is, after a seal has terminated, no other seal can keep shared objects alive inside the terminated protection domain.

I-6: When a thread executing a method of a shared object invalidates the external reference it is calling through, the invalidation operation (I-1, I-2, I-3, or I-4) is guaranteed to complete before the thread throws an exception. This property also holds, if the thread terminates the seal it belongs to (see invalidation rule I-5).

4.4.3 Examples of External References

The following examples illustrate how external references support direct communication between seals that are not in a parent–child relationship (see figure 4.1).

We have three different seals, seal A is parent of seals B and C. Seal B provides a shared object O, which shall be made accessible to seal A. For

this purpose, seal B creates an external reference Rb for the shared object O and passes it to seal A (either via channel operations or invoking a method of an existing external reference). Seal A receives Ra, which is a copy of Rb. As long as Ra remains valid, seal A is able to directly invoke methods on the shared object O.

Now assume seal A decides to grant seal C direct access to the shared object O. Thus, seal A forwards Ra to seal C, which receives the copy Rc. Now all three seals may concurrently call methods on the shared object O.

Seal B may invalidate Rb (Rb, Ra, and Rc) or only the descendant of Rb passed to seal A (Ra and Rc). Seal A may invalidate Ra (Ra and Rc) or only the descendant of Ra forwarded to seal C (Rc), but not Rb. Seal C may only invalidate Rc.

Now assume seal C invokes a method on shared object O passing Rc within the argument capsule. Thus, the J-SEAL2 kernel has to copy Rc step-by-step on the shortest path in the hierarchy to seal B (i.e., C–A–B). Copying Rc to seal A yields Ra, since seal A already holds the ancestor of Rc. Seal B receives Rb in the argument capsule, as it already maintains the ancestor of Ra.

4.5 Implementation Issues

Since in J-SEAL2 there is no direct sharing of object references between different protection domains, all communication involves the copying of messages. J-SEAL2 employs Java serialization to create a deep copy of an object graph of serializable objects. Therefore, only serializable objects can be passed between different domains. The kernel ensures that a communicated serialized object graph is deserialized within the target domain. Thus, the deserialized object graph only refers to classes from the receiving domain.

As serialization and deserialization are expensive operations, J-SEAL2 offers optimizations for certain object types that are frequently used in communication messages, such as Java primitive types, arrays of primitive types, as well as strings. Primitive types do not include object references, they can be passed within simple wrappers. The classes for arrays of primitive types and for strings are always loaded by the system class-loader, thus we need not take care whether they are copied within the target domain. For arrays of primitive types, we can use array cloning or `System.arraycopy()`. For strings, there is a special constructor taking another string as argument. All

these optimizations are performed by the J-SEAL2 kernel, they are transparent to the programmer. Performance measurements[6] show that capsule optimizations improve the performance of capsule creation and opening by a factor of 20–50.

External references complicate the serialization and deserialization of object graphs. External references are not serializable, but they have to be treated in a very special way. Since the kernel keeps track of external references and their copies, they must be separated from the rest of the serialized object graph. The kernel employs a dedicated copying algorithm for external references, details can be found in section 4.4.2 and in [5].

The J-SEAL2 implementation of communication channels ensures that a message is copied to the receiving protection domain only after a communication match. Send and receive requests are treated as equivalent communication offers. They are inserted in kernel queues residing in the same protection domain as the issuing thread. The kernel checks neighbour domains for matching offers. Only if the search is successful, the kernel copies the message to the receiving domain. This implementation is very different from traditional message passing, where a sender directly inserts a message into the receiver queue. By separating sender and receiver queues the J-SEAL2 kernel ensures that an agent cannot mount a denial-of-service attack against a neighbour domain by filling its receiver queues with messages, eventually causing the receiver to exceed its memory limits.

4.6 Inter Agent Method Calling (IAMC)

IAMC is an efficient and convenient communication framework based on external references. Following the J-SEAL2 micro-kernel design, IAMC is built on top of the kernel, it is a library package and may be replaced or supplemented by other communication frameworks.

IAMC aims at simplifying access to services provided by other seals. In J-SEAL2 all service registration protocols and service access are based on IAMC. In order to export a certain service a seal has got to specify a Java interface defining the service methods. Furthermore, the seal must provide an appropriate implementation of the interface, either an object of a class

[6] In this benchmark we created capsules consisting of a string, a byte array, and a long value. The sizes of the string and the array were chosen independently between 0 and 1000.

implementing the interface, or the seal itself may implement some service interfaces. The seal creates an IAMC dispatcher, which can be made accessible to foreign seals via external references. An IAMC dispatcher accepts service requests from external reference method invocations and dispatches them to the service implementation.

The IAMC dispatcher controls the concurrency factor of the service by managing a set of worker threads to handle incoming requests. This approach also solves the problem that a method invocation through an external reference may leave the shared object in an inconsistent state, if the calling thread is stopped asynchronously. A calling thread simply inserts the request into a queue managed by the dispatcher and blocks until the request has been processed or an exception has been thrown by a worker thread. Thus, the shared object is only accessed by threads belonging to the same seal. Only the request queue has to be protected in order to ensure state consistency. The IAMC dispatcher implementation solves this problem employing 'self-communication' with channel primitives [43, 42, 11].

Clients of a service receive external references to the IAMC dispatcher. For convenience, they may wrap an external reference into a stub class implementing the service interface. The IAMC stub is responsible for argument/result marshalling/unmarshalling (argument capsule creation and result capsule opening) and providing the method signature necessary for external reference method invocation. J-SEAL2 offers a generator tool, which automatically creates IAMC stubs for Java interfaces.

Summing up, IAMC is an easy to use communication framework, which takes advantage of external references. It allows to shortcut communication paths in the seal hierarchy[7], while at the same time all seals on the shortest path in the hierarchy between the client and the service seal have the right to break the communication with immediate effect (external reference invalidation). Implicit external reference invalidation during seal termination ensures that clients are not blocked infinitely, if the communication partner fails or moves away (in the case of a mobile agent). Thus, IAMC communication is inherently mobility aware.

[7] In effect, IAMC communication involves exactly two thread switches (handing over the request to a worker thread and passing back the result to the calling thread), independent of where the communication partners are located in the seal hierarchy.

n	1	2	3	4	5	6	7	8	9	10	11	12
Sync. ch.	90	170	251	350	431	521	601	691	781	871	961	1051
Async. ch.	30	60	90	120	150	181	220	240	281	320	451	581
Ext. ref.	10	10	10	10	10	10	10	10	10	10	10	10
IAMC	100	100	100	100	100	100	100	100	100	100	100	100

Table 4.1: J-SEAL2 communication performance (time in microseconds).

4.7 Evaluation

In this section we compare the performance of the various communication mechanisms offered by the J-SEAL2 platform. For this purpose, we measured the time it takes to transmit a capsule containing a Java long integer over n seal boundaries ($1 \leq n \leq 12$)[8].

All measurements were collected with Sun's Java 2 Software Development Kit, Standard Edition, version 1.3.0 (Hotspot Client VM) on a Windows NT 4.0 workstation (Intel Pentium II, 400MHz clock rate). In order to avoid errors in measurement due to garbage collection or method compilation, we show the median of 101 measurements.

The results in table 4.1 confirm that for external reference and IAMC the communication costs are independent of where the communication partners are located in the seal hierarchy, whereas channel communication overhead increases linearly with the communication partners' distance. Since IAMC involves exactly two thread switches, the IAMC communication costs[9] are roughly comparable with channel communication over two seal boundaries. Asynchronous channel communication is up to 3 times faster than synchronous channel communication, because it allows the JVM scheduler to avoid some thread switches. The deterioration of performance of asynchronous channel communication for $n > 10$ is due to increased scheduler activity (preemption).

Considering a typical J-SEAL2 configuration for a large-scale electronic commerce application (e.g., see figure 1.1 on page 11), an agent has to communicate at least over four seal boundaries in order to access a service (agent–

[8]In this benchmark we were repeating each capsule transfer 1000 times, since in Java the accuracy of measurement is 1 millisecond.

[9]Note that the IAMC measurements include the time to create, to transfer, and to open a capsule, as well as the time for dispatching and method invocation, while we only measured capsule transfer time for channel and external reference communication.

sandbox–sandbox manager–RootSeal–service). In this case, external reference communication is 35 (12) times faster than synchronous (asynchronous) channel communication. Even though IAMC is a high-level communication protocol, it is still 3.5 (1.2) times faster than synchronous (asynchronous) channel communication. Note that if a service invocation returns a result object, the overhead for channel communication is about twice as high as stated in table 4.1, whereas external reference and IAMC communication do not incur any additional overhead.

Chapter 5

Resource Control

5.1 Introduction

Operating system kernels provide mechanisms to enforce resource limits for processes. The scheduler assigns processes to CPUs reflecting process priorities. Furthermore, only the kernel has access to all memory resources. Processes have to allocate memory regions from the kernel, which verifies that memory limits for the processes are not exceeded. Likewise, a mobile agent kernel must prevent denial-of-service attacks, such as mobile agents allocating all available memory. For this purpose, accounting of physical resources (i.e., memory, CPU, network bandwidth, etc.) and logical resources (i.e., number of threads, number of protection domains, etc.) is crucial.

Whereas J-SEAL2 [5, 6] is primarily designed for mobile agents, the approach described here is in many ways applicable to other distributed programming paradigms practiced in Java, since the mobile agent paradigm is very comprehensive in terms of involved issues and technologies. The techniques employed in J-SEAL2 could thus greatly improve stability and security in the execution of Java Applets, or traditional distributed applications, where strong protection domains and resource control mechanisms are often needed. Further potential use cases include technologies such as World-Wide-Web server extensions (Java Servlets [35]) and Java application servers (e.g., Enterprise JavaBeans containers [33]).

The great value of resource control is that it is not restricted to serve as a base for implementing security mechanisms. Application service providers may, for example, need to guarantee a certain quality of service, or to create the support for usage-based billing, in order to amortize investments in

hardware and software set at customers' disposal. The basic kernel extensions described in this chapter will be necessary to schedule the quality of service or to support the higher-level accounting system, which will bill the clients for consumed computing resources. This thesis is however restricted to the kernel extensions that were necessary to add resource control to J-SEAL2; faithful to the micro-kernel approach, J-SEAL2 relegates to the higher levels the mechanisms which do not absolutely have to be part of the kernel.

This chapter is organized as follows. The next section presents the design goals and the resulting resource control model, and section 5.3 the corresponding APIs. Section 5.4 explains our implementation techniques, for which section 5.5 presents some performance measurements.

5.2 Objectives and Resulting Model

The ultimate objective of this work is to enable the creation of execution platforms, where anonymous mobile agents, or more general programs, may securely coexist without harming each other, and without harming their environment. Examples of such platforms are user-extensible databases [19] or decentralized e-commerce and trading systems as, for example, in [23]. Java Applet execution platforms – World-Wide-Web browsers – as well as embedded Java devices also need such guarantees. The desire to deploy this kind of platforms translates into the following requirements:

- Sufficiently abstract concepts, in order to make mapping of policies into implementations more straightforward, and with a view to making resource control and eventual billing more manageable.
- Accounting of low-level, physical resources as well as higher-level, logical resources, such as threads.
- Prevention against denial-of-service attacks, which are based on CPU, memory, or communication misuse.
- Fair distribution of resources among concurrent domains, even outside the context of malicious activities.
- Fine-grained load-balancing of mobile agent applications on a cluster of machines.

Since some aspects of resource control are to be manageable by the application developer, it is important that the general model integrates well with the existing J-SEAL2 programming model [5]. The resource control facilities shall reflect the hierarchical system structure. Hierarchical process models have been used successfully by operating system kernels, such as the Fluke micro-kernel [17]. The Fluke kernel employs a hierarchical scheduling protocol, CPU Inheritance Scheduling [18], in order to enforce scheduling policies. In this model, a parent domain donates a certain percentage of its own CPU resources to a child process. Initially, the root of the hierarchy possesses all CPU resources.

A general model for hierarchical resource control, such as e.g. Quantum [27], fits very well to the J-SEAL2 hierarchical domain model. At system startup the root domain, RootSeal, owns by default all resources the Java runtime system allocates from the underlying operating system, for example, 100% CPU, the entire virtual memory, unlimited network usage, the maximum number of threads the underlying JVM [26] is able to cope with, an unlimited number of subdomains, etc. Moreover, the root domain, as well as other domains loaded at platform startup, are considered as completely safe, and, consequently, no resource accounting will be enforced on them. This default behavior may however easily be overridden if specific configurations should require accounting even for trusted domains.

When a nested protection domain is created, the creator donates some part of its own resources to the new domain. Figure 5.1 illustrates the way resources are either shared or distributed inside a seal hierarchy. In the formal model of J-SEAL2, the Seal Calculus [43], the parent seal supervises all its subdomains, and inter-domain communication management was the main concern so far. Likewise, in the resource control model proposed here, the parent seal is responsible for the resource allocation with its subseals. This produces a nested structure, where the parent seal is initially the sole owner of its resources, and it may either share them or dispatch fractions of them to its subseals. However, the sum of all resources within a protection domain, e.g., in the *Untrusted application* of figure 5.1, remains constant.

Our resource control model stems from further design goals, such as portability and transparency: the next subsections are dedicated to describing these.

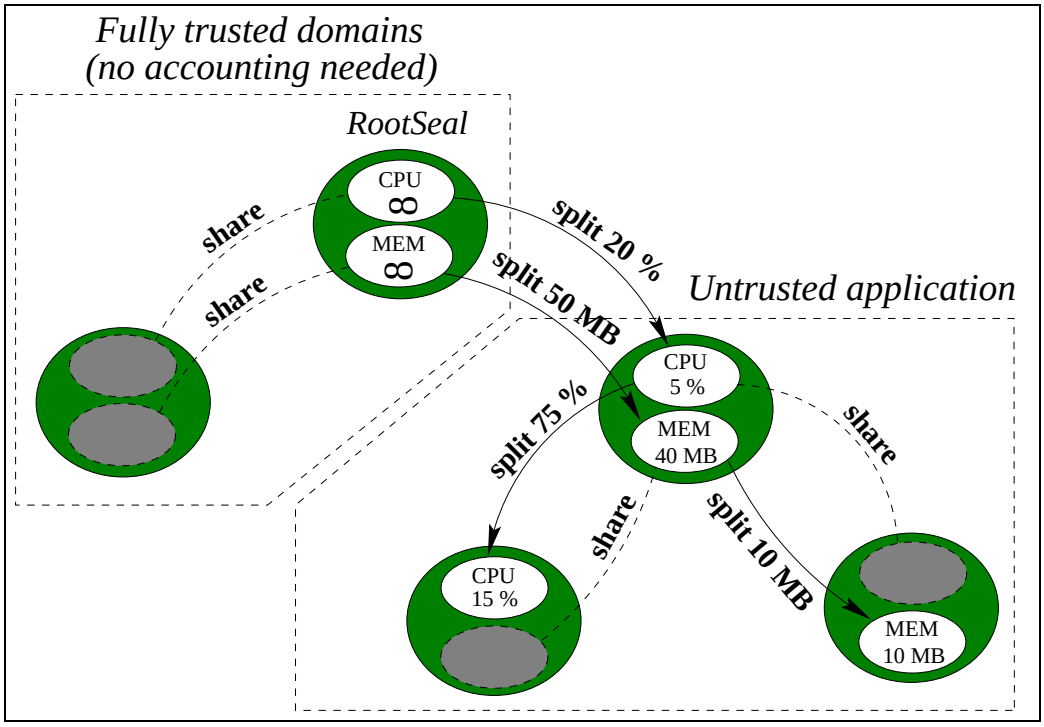

Figure 5.1: Illustration of the general resource control model.

5.2.1 Portability and Transparency

Portability is crucial for the success of any mobile agent platform. There are already some Java-based systems offering resource control facilities, such as Alta [40] and KaffeOS [1, 2], which we discussed in chapter 2. However, they rely on modified Java runtime systems, which are not portable. As a result, these systems are not suited for large-scale applications that have to support a wide variety of different hardware platforms and operating systems. Our goal is to provide a general-purpose model which is not dependent on specific implementation techniques, and to explore primarily completely portable solutions. This entails that we have to cope with certain restrictions and with performance levels sometimes inferior to those of existing realizations. Our portable approach will nevertheless show its advantages in the longer term: our solution will always perform somewhat slower than the fastest JVMs without resource control mechanisms, but, on the other hand, we will be able to exploit the latest techniques in Java implementation optimizations, which will often not be possible with non-portable implementations.

A related important requirement of our resource control model is that

unmodified agent applications must be able to execute on our platform. In other words, resource control must be transparent to applications which do not explicitly manage their pool of resources.

For portability reasons, it should also be stressed that the goal of this work is not to implement any kind of real-time guarantee. The resources that are managed and distributed internally to the JVM are thus entirely dependent upon what the JVM process itself is given by the underlying operating system.

5.2.2 Minimal Overhead for Trusted Domains

Since J-SEAL2 is designed for large-scale applications, where a large number of services and mobile agents are executing concurrently, design and implementation must minimize the overhead of resource accounting. Some domains, such as core services, are fully trusted. Their resource consumption need not be controlled by the kernel.

5.2.3 Support for Resource Sharing

In certain situations protection domains that are neighbours in the hierarchy may choose to share some resources. In this case, resource limits are enforced together for a set of protection domains. As a result, resource fragmentation is minimized. For example, consider a mobile agent creating a subdomain for a certain task. Frequently, the creating domain does not want to donate some resources to the subdomain, but it rather prefers to share its own resources with the subdomain. A property of our approach is that if a domain has unlimited access to a resource, this means that it is sharing it with RootSeal.

5.2.4 Managed Resources

Within each untrusted protection domain, the J-SEAL2 kernel shall account for the following resources:

- CPU_RELATIVE defines the relative share of CPU. It is expressed as a fraction of the parent domain's own relative share, but takes a slightly different meaning when the parent itself is a trusted domain; the precise semantics is exposed in section 5.3.2.

- MEM_ACTIVE is the highest amount of volatile memory that a protection domain is allowed to use at any given moment.
- THREADS_ACTIVE specifies the maximal number of active threads by protection domain at any moment. Uncontrolled creation of threads has to be avoided, as it results in increased load for the scheduler; it may even crash the JVM, as there is currently no standard Java construct allowing one to inquire about the maximum number of threads a JVM implementation is able to cope with.
- THREADS_TOTAL limits the number of threads that may be created throughout the lifetime of a protection domain, as thread creation is an expensive (kernel-level) operation.
- DOMAINS_ACTIVE specifies the maximal number of active subdomains a protection domain is allowed to have at any given moment. This limit is to minimize management overhead inside the kernel by controlling the complexity of the seal hierarchy at any time.
- DOMAINS_TOTAL bounds the number of subdomains that a protection domain may generate throughout its lifetime, as domain creation and termination are expensive kernel operations.

Note that the kernel of J-SEAL2 is not responsible for network control. This is because the micro-kernel does not provide access to the network. Instead, network access can be provided by multiple services. These network services or some mediation layers in the hierarchy are responsible for network accounting according to application-specific security policies. Let us stress that the network is not a special case, since J-SEAL2, thanks to its homogeneous model, may limit communication with any services, e.g., file IO.

Another resource kind that could be expected in the above list of kernel-managed resources is the total amount of CPU allocated to a given protection domain throughout its lifetime. It is however not clear what the unit of measurement should be for this resource, while still preserving a completely hardware-independent model. The main objective of this kind of resource accounting would be to prevent applications from indefinitely cluttering up platforms; in a heterogeneous set of servers it makes however more sense to express total lifetime abstractly as the wall clock time elapsed since the

application was started, than as the number of consumed CPU cycles. Using as unit of measurement the amount of executed Java bytecodes, although portable, was also regarded as too low-level. Measuring wall clock time can be achieved at the application level, by establishing a controlling domain with sufficient rights to kill all misbehaving applications; this is a viable approach, since in J-SEAL2, when a parent disposes of a child seal, all resources are guaranteed to be freed properly. Accounting of total CPU time was therefore discarded from the kernel.

Finally, there is also no such resource as MEM_TOTAL, a limit to the accumulated amount of memory used throughout the lifetime of a protection domain. It could be needed to prevent the kind of denial-of-service attacks where a malicious domain creates a lot of dynamic objects in order to keep the CPU busy with garbage collection. Its implementation would however require maintenance of an additional counter, which we preferred to avoid. Instead, J-SEAL2 will take preventive action by charging an abstract amount of CPU as a compensation for the garbage collection induced by each object created.

The six basic resource types retained for management by the J-SEAL2 kernel are discussed in more detail in the API section below.

5.3 API

In this section we give an overview of the resource control API provided by the J-SEAL2 kernel. A detailed specification of the API can be found in [7].

There are 2 kernel abstractions dedicated to resource control: A resource object of type `Res` represents a resource of a certain type available for a protection domain. Resource sets of type `ResSet` ease the management of multiple resources. Furthermore, the kernel class `Seal`, which supports domain creation and termination, has been extended to allow a parent domain to restrict the resources of its children.

5.3.1 Definitions

In this section we provide some definitions, which simplify the description of the resource control API. In the following definitions let S denote an arbitrary domain in the hierarchy.

Root `Res` object: A root `Res` object of the domain S is a `Res` object responsible for resource control in S. A root `Res` object is returned by an invocation of the method `getCurrentRes` in class `Res` (for details see the following section).

Descendant `Res` object: A descendant `Res` object D of the domain S is the result of splitting a root `Res` object R of S. R is also called the parent `Res` object of D. When a descendant `Res` object is used in a `ResSet` object to create a nested domain, it will be used for resource control in the created child domain.

Note that these definitions are relative to the domain S. A descendant `Res` object D of the domain S is a root `Res` object in a child C of S, if D was in the `ResSet` object used for creating C. When we use the terms root and descendant `Res` objects in the description of a method, we implicitly assume `Res` objects of the domain invoking the method.

5.3.2 Class `Res`

For each type of resource, a protection domain has an associated root `Res` object reflecting how much of the resource the domain has been granted. A `Res` object defines a resource limit and provides information on the current resource usage in order to support resource aware computations. It offers an operation allowing a domain to split up some part of the resource. This operation yields a new descendant `Res` object that may be donated to children domains. The root domain, RootSeal, creates an initial `Res` object for each type of resource during startup. RootSeal distributes resources to service components and to application domains according to a configuration provided by the system administrator. Table 5.1 summarizes the interface of a `Res` object.

The static method `getCurrentRes` returns the root `Res` object for a given type of resource of the invoking domain. In order to indicate the requested resource type, the constants `CPU_RELATIVE`, `MEM_ACTIVE`, `THREADS_ACTIVE`, `THREADS_TOTAL`, `DOMAINS_ACTIVE`, and `DOMAINS_TOTAL` (i.e., relative CPU share, active memory in bytes, as well as active and cumulative threads and subdomains) are used. The information, for which type of resource a `Res` object is responsible, is permanently associated with the `Res` object in order to prevent the programmer from mixing up different types of resources

```
public final class Res {
  public static final int
    CPU_RELATIVE = 0,
    MEM_ACTIVE = 1,
    THREADS_ACTIVE = 2, THREADS_TOTAL = 3,
    DOMAINS_ACTIVE = 4, DOMAINS_TOTAL = 5;

  public static Res getCurrentRes(int type);
  public int getType();
  public long getLimit();
  public long getUsage();
  public Res split(long limit);
  public void setLimit(long limit);
  public void combine();
}
```

Table 5.1: The `Res` API.

by mistake. The `getType` method returns the type of resource a `Res` object is representing.

`getLimit` returns the resource limit of a `Res` object. A negative value means that there is no resource limit. Concerning the semantics of the resource limit, the relative CPU share (`CPU_RELATIVE`) is treated differently from all other resource types. A relative CPU share of n means that domains created with the corresponding `Res` object may use at most a fraction of $\frac{n}{\text{sum of all CPU limits in the system}}$ of the CPU time available to domains with a CPU limit ≥ 0[1]. `getUsage` returns the resource consumption of all domains sharing the same root `Res` object. A negative value means that the J-SEAL2 kernel does not account for the resource.

As the `Res` API does not expose any public constructor, the `split` operation has to be used in order to create descendant `Res` objects that may be donated to subdomains. `split` may be invoked only on root `Res` objects. It returns a new descendant `Res` object responsible for the same type of resource as the root `Res` object, which becomes the parent of the descendant.

[1] In our current implementation, this resource is controlled by periodic sampling of the amount of executed bytecode instructions. The precision of the measurement is implementation dependent; there is indeed a bias induced by the fact that the CPU resource is not allocated by absolute values, but by relative shares, while in the implementation, the reference value is the aggregated consumption measured among untrusted domains and is not, as could be expected, the resource taken as a whole.

```
public final class ResSet {
  public static ResSet getCurrentResSet();
  public ResSet copy();
  public Res getRes(int type);
  public void setRes(Res r);
  public void combine();
}
```

Table 5.2: The `ResSet` API.

The descendant `Res` object has the resource limit, which was passed to `split` as argument, and an initial resource usage of zero. The resource usage of the parent `Res` object is incremented by the limit given to the descendant.

The `setLimit` method provides a mechanism to modify the resource limit of a `Res` object. The new resource limit is passed as argument. The resource usage of the parent `Res` object is adjusted accordingly. A parent domain may use descendant `Res` objects in order to monitor the resource usage of children domains. With the aid of `setLimit`, the parent is able to adjust the resource limits for the children domains.

The `combine` operation allows to merge `Res` objects that have been split before. If it is invoked on a root `Res` object, `combine` has no effect. If it is called on a descendant `Res` object, the descendant is combined with its parent `Res` object, i.e., the resource usage of the parent object (if it is accounted for) is reduced by the limit of the descendant. The descendant `Res` object is marked as invalid and cannot be used anymore. Combination is only possible, if the descendant `Res` object is not used by any subdomain (i.e., all subdomain created with the descendant `Res` object must be terminated before).

5.3.3 Class `ResSet`

A `ResSet` object offers a convenient way to manage all resources given to a domain. It holds exactly one `Res` object for each type of resource. Table 5.2 summarizes the public interface of a `ResSet` object:

The static method `getCurrentResSet` returns a `ResSet` object with the root `Res` objects of the domain the calling thread is executing in. This `ResSet` object may be used to access the individual `Res` objects of the domain. The `copy` method creates a shallow copy of a `ResSet` object. The copy

```
public class Seal {
  public static void unwrap(WrappedSeal wrapped, String sealname,
                            ResSet resources);
  public static void unwrap(WrappedSeal wrapped, String sealname) {
    unwrap(wrapped, sealname, ResSet.getCurrentResSet());
  }
  ...
}
```

Table 5.3: The `unwrap` methods of class `Seal`.

contains the same references to `Res` objects as the original `ResSet` object. The `getCurrentResSet` and `copy` methods are the only mechanisms allowing to allocate new `ResSet` objects. There is no public constructor, because the API enforces the constraint that a `ResSet` always holds exactly one `Res` object for each type of resource.

The `getRes` method return the `Res` object for a given type of resource. The argument is a resource constant defined in the class `Res`. The `setRes` method replaces the `Res` object in the set, which has the same resource type as the `Res` object given as argument. The `combine` method offers a convenient way to invoke `combine` on all `Res` objects in the set.

5.3.4 Class `Seal`

The `Seal` abstraction provides methods for domain creation (unwrapping) and removal (wrapping or disposing). Table 5.3 summarizes the `unwrap` methods of the `Seal` class. Other methods are not shown, because they are not affected by the resource control extension.

The `unwrap` method with 3 arguments requires a wrapped representation of the subdomain to create (corresponding to the serialized state of a mobile agent), its name, as well as a `ResSet` object with the resources for the new subdomain. The `unwrap` operation with 2 arguments implicitly shares the resources of the unwrapping domain with the created child domain.

When a new child domain is created, the parent's `DOMAINS_ACTIVE` and `DOMAINS_TOTAL` `Res` objects are charged for the created subdomain, while the child's resource objects are charged for the CPU time consumed for unwrapping (involving class-loading and linking), for memory allocation, as well as for the child's initializer thread.

```
long MB = 1024*1024;

ResSet rP = ResSet.getCurrentResSet();
Res cpu = rP.getRes(Res.CPU_RELATIVE);
Res mem = rP.getRes(Res.MEM_ACTIVE);

ResSet rA = rP.copy();
long cpuA = (long)(cpu.getLimit()*0.75);
rA.setRes(cpu.split(cpuA));

ResSet rB = rP.copy();
rB.setRes(mem.split(10*MB));

Seal.unwrap(childA, nameOfChildA, rA);
Seal.unwrap(childB, nameOfChildB, rB);
```

Table 5.4: Resource control example.

5.3.5 Example

The code fragment in table 5.4 demonstrates how the resource control API is used to control the resources of children domains. This example corresponds to the *Untrusted application* depicted in figure 5.1 on page 48.

A parent domain, which has limited CPU and memory resources, creates 2 subdomains: One child domain (*childA*) gets 75% of the parent's CPU resources and shares the memory resources with the parent, while the other child domain (*childB*) receives 10 MB of active memory and shares the CPU resources with the parent.

5.4 Implementation Issues

In this section we present the techniques we are using for the implementation of the resource control model discussed in the previous sections. Since accounting for logical resources, such as active and cumulative threads and subdomains, requires only minor modifications to a few J-SEAL2 kernel primitives, we focus on accounting for physical resources, such as memory and CPU consumption.

5.4.1 No Direct Sharing

Since its initial release the J-SEAL2 kernel has been designed to ease the integration of resource control facilities. It guarantees accountability, i.e., user-visible objects belong to exactly one protection domain. References to an object exist only within a single domain[2], i.e., in J-SEAL2 there is no direct sharing of object references between distinct domains. Therefore, it is possible to account each allocated object to exactly one protection domain. This feature not only simplifies resource accounting, but it is also crucial for immediate resource reclamation during domain termination.

5.4.2 Bytecode Rewriting

In our approach we employ bytecode rewriting techniques both for memory and CPU accounting. This is because it is to our understanding the only entirely portable way to implement the needed accounting mechanisms. It is unrealistic to expect the source code of every application to be available for modification. Moreover, if we want guarantees against denial-of-service attacks, we cannot rely on foreign code to perform any voluntary self-limiting operations, whereas if we modify its bytecode before it starts executing, we can 'oblige' it to provide any information needed by the kernel and to obey any restriction imposed on it by the environment. Instead of rewriting bytecode for CPU control, the J-SEAL2 kernel might, for example, ask the underlying operating system for information about the CPU consumption of each thread, but this is possible only when Java threads are directly mapped into operating system threads. Another approach would be to run a modified JVM; the arguments against this are however exposed in section 5.2.1. A further discussion of existing (and non-portable) approaches is to be found in chapter 2.

In our implementation, the bytecode of a Java class is modified before it is loaded by the JVM [26]. Code for memory accounting is inserted before each memory allocation instruction (for details, see section 5.4.9). CPU accounting uses an abstract measure, the number of executed bytecode instructions. Therefore, code for CPU accounting is inserted in every basic block of code (details are presented in section 5.4.10).

[2]The only exception to this rule are `Res` objects (see section 5.3.2) used for resource sharing.

Rewriting for memory accounting must be done before rewriting for CPU accounting, because memory accounting inserts additional bytecode instructions to enforce memory limits, while accounting CPU consumption does not involve any object allocation.

5.4.3 Domain Types

The resource control model supports trusted domains that have unlimited access to certain types of resources. For performance reasons, the J-SEAL2 kernel does not account for the consumption of these resources. Regarding CPU and memory accounting, we distinguish 4 types of domains:

NO-ACC: Domains without memory limit and without CPU control may execute unmodified Java code, as they do not need to execute any accounting instructions.

CPU-ACC: Domains without memory limit, but with CPU control have to execute CPU accounting instructions. However, code for memory accounting is not required in such domains.

MEM-ACC: Domains with a memory limit, but without CPU control have to execute memory accounting instructions. However, code for CPU accounting is not required in such domains.

CPU-MEM-ACC: Domains with a memory limit and with CPU control have to execute accounting code for memory allocation as well as for CPU consumption.

5.4.4 Accounting Objects

In MEM-ACC and in CPU-MEM-ACC domains `MemAccount` objects represent memory limit and current usage. In CPU-ACC and in CPU-MEM-ACC domains objects of the type `CPUAccount` maintain CPU consumption. These objects are used only by the J-SEAL2 kernel, they are not accessible by user code. Each thread has associated the `MemAccount` object and a `CPUAccount` object of the domain it is executing in; `null` values indicate that a domain does not need a `MemAccount` or `CPUAccount` object. Java thread-local variables (instances of `java.lang.ThreadLocal`) are used to implement this association. The `MemAccount` and `CPUAccount` classes offer a static method

`getCurrentAccount`, which returns the corresponding accounting object of the domain the calling thread is executing in.

Because access to `MemAccount` and above all to `CPUAccount` objects may be extremely frequent, accessing these objects from thread-local variables in every method would cause a significant performance penalty[3]. Therefore, non-native methods are rewritten in order to pass the necessary accounting objects as additional arguments. Native methods are excluded from rewriting, because we cannot account for memory allocated and CPU time consumed by native code. We are relying on modern inter-modular register allocation algorithms implemented by state-of-the-art JVMs to minimize the overhead of passing the accounting objects through the whole method call-graph.

As an example for the rewriting process, consider method `a` given in table 5.5. The rewritten[4] version of method `a` for a CPU-MEM-ACC domain is given in table 5.6. Here we are only presenting the additional arguments, while the inserted accounting code is discussed in sections 5.4.9 and 5.4.10. In this example, method `a` receives two additional arguments for the `CPUAccount` and `MemAccount` objects[5]. The additional arguments are passed to all invoked methods/constructors (in this example to method `b`).

```
void a(int x) {
  b(null, x);
}
```

Table 5.5: Method `a` before rewriting.

```
void a(int x, MemAccount mem, CPUAccount cpu) {
  b(null, x, mem, cpu);
}
```

Table 5.6: Method `a` rewritten for a CPU-MEM-ACC domain.

[3]In Sun's JDK 1.3 implementation thread-local variables are managed as hash-maps, i.e., each access to a thread-local variable requires a hash-map lookup.

[4]For the sake of easy readability, we present rewriting transformations at the Java level, even though the implementation works at the JVM bytecode level.

[5]Note that in a CPU-ACC or MEM-ACC domain only one additional argument would be necessary to hold the accounting object.

5.4.5 Callbacks from Native Code

Native code invoking Java methods complicates the resource control implementation, because the native code is not aware of the accounting objects to be passed to Java methods as extra arguments. The following three scenarios of Java method invocation by native code are particularly important:

- Thread creation: The Java runtime system (native code) invokes the `run` method of a thread object when a thread is started with the aid of the `start` method.
- Static initializers: Static initializers are invoked directly during class-loading, i.e., they are invoked by native code.
- Reflection: The `invoke` method of `java.lang.reflect.Method`, as well as the `newInstance` method of `java.lang.reflect.Constructor` are native methods.

When the thread invoking a Java method from native code has already set its thread-local accounting objects, it is sufficient to provide for each method an additional one with the same signature, which takes the required accounting objects from thread-local variables and passes them to the rewritten method. In the rewriting example given in tables 5.5 and 5.6 we have to supplement the rewritten method with method `a` in table 5.7. Note that when a constructor is rewritten according to this scheme, the invocation of another constructor of the same class or of a constructor of the superclass has to antecede the lookup of the accounting objects.

```
void a(int x) {
  MemAccount mem = MemAccount.getCurrentAccount();
  CPUAccount cpu = CPUAccount.getCurrentAccount();
  a(x, mem, cpu);
}
```

Table 5.7: Solving callbacks from native code.

However, when a new thread starts executing the `run` method, the thread-local accounting objects have not been initialized yet. As protection domains in J-SEAL2 do not have direct access to the class `java.lang.Thread` but have to employ a safe wrapper class instead [6], the wrapper initializes the

thread-local accounting variables with the accounting objects of the protection domain the new thread belongs to. These objects are passed to the constructor of the wrapper by the J-SEAL2 kernel.

When a new protection domain is created, the J-SEAL2 kernel allocates a new initializer thread with the accounting objects for the new domain. While starting this thread, the thread wrapper initializes the thread-local accounting variables and starts to load the classes of the new protection domain. The class-loading already happens in the accounting context of the new domain.

5.4.6 Shared Classes

As discussed in section 3.3.2.1, the J-SEAL2 kernel distinguishes between shared and replicated classes. Shared classes are loaded by the system class-loader (they exist only once in the JVM), while replicated classes, such as the classes of a mobile agent, are loaded by the class-loader of a protection domain (they are reloaded in each domain). All JDK classes as well as most classes from the J-SEAL2 kernel are shared. Certain J-SEAL2 library classes that are frequently used may be shared as well, in order to avoid the overhead of reloading them multiple times.

In the Java 2 platform [34] it is not possible to load a JDK class with a class-loader different from the system class-loader. Depending on the JVM implementation, certain core JDK classes (e.g., `java.lang.Object`, `java.lang.String`, `java.lang.Throwable`, etc.) are assumed to exist only once in the system. Replicating such classes crashes the JVM. Furthermore, the class-loader API of JDK 1.2 specifies that all classes in the `java` package or in a subpackage thereof can only be defined by the bootstrap class-loader. As a consequence, we have the following constraints for accounting for resources used in the JDK:

- All JDK classes are loaded by the system class-loader; there is only a single version of each JDK class.
- Since the same JDK class may be used in different types of domains (NO-ACC, CPU-ACC, MEM-ACC, or CPU-MEM-ACC), JDK classes have to include the accounting code for all domain types.
- The rewriting of JDK classes must be off-line (e.g., during the installation of the J-SEAL2 platform), because JDK classes are always loaded

by the system class-loader, which we cannot modify.

The example in table 5.8 shows how method `a` given in table 5.5 would be rewritten, if it were defined in a shared class. A method with the same signature as the original method dispatches to the appropriate implementation, when it is invoked from native code. For each type of domain, there is a different method implementation. In this example we distinguished the signature of the NO-ACC implementation from the dispatcher method by adding a dummy argument of type `NoAccount`. The compilers of state-of-the-art JVMs may be able to remove this useless argument.

```
void a(int x) {
  MemAccount mem = MemAccount.getCurrentAccount();
  CPUAccount cpu = CPUAccount.getCurrentAccount();
  if (cpu == null)
    if (mem == null) a(x, (NoAccount)null);
    else a(x, mem);
  else
    if (mem == null) a(x, cpu);
    else a(x, mem, cpu);
}
void a(int x, NoAccount _no)  { b(null, x, _no); }
void a(int x, CPUAccount cpu) { b(null, x, cpu); }
void a(int x, MemAccount mem) { b(null, x, mem); }
void a(int x, MemAccount mem, CPUAccount cpu) { b(null, x, mem, cpu); }
```

Table 5.8: Rewriting methods in shared classes.

Alternatively, it is possible to rename the NO-ACC implementation. This approach complicates rewriting, since a table of renamed methods of shared classes has to be maintained, but it has the advantage that replicated classes of trusted domains (e.g., classes of an authenticated, fully trusted agent) can be rewritten very efficiently, because only method signatures in the constant-pool [26] are affected, whereas the method code remains unchanged (in contrast, passing the extra `NoAccount` argument requires additional bytecode instructions).

5.4.7 Optimizations

Rewriting shared classes as discussed in the previous section increases the code size by more than factor 4. Because the increased code size affects

the memory requirements and the startup overhead of the J-SEAL2 kernel (more methods may be compiled), the following optimizations are being implemented:

- A leaf method, which does not allocate any objects, requires neither MEM-ACC nor CPU-MEM-ACC implementations (i.e., the NO-ACC implementation can be used in MEM-ACC domains, and CPU-MEM-ACC domains can employ the CPU-ACC implementation).
- As a generalization of this optimization, we do not need to provide MEM-ACC and CPU-MEM-ACC implementations for methods without any memory allocation instructions, if they invoke only methods satisfying the same condition.
- We can optimize the code of shared kernel classes by hand in order to minimize the overhead for resource control (e.g., when allocating a set of objects, we can account for the total size of these objects at once).

5.4.8 Rewriting Abstract Methods

There are two different approaches for dealing with abstract Java methods (including interface methods) in shared types (classes or interfaces):

1. Abstract methods are not rewritten. This approach simplifies the rewriting process, but invoking a method on a variable of a (static) type, in which the called method is declared as abstract (e.g., interface method call), incurs high overhead, because the dispatching method has to access the accounting objects from thread-local variables.
2. Abstract methods are rewritten. For each abstract method in a shared class, the type signatures of the 4 possible implementations (NO-ACC, CPU-ACC, MEM-ACC, CPU-MEM-ACC) are added. As a result, a class implementing the abstract method has to provide all 4 implementations, even if the implementing class is a replicated one (in this case, 3 implementations may be dummies). This approach allows to pass the accounting objects directly to the invoked method, no matter whether it is an interface method or not.

The J-SEAL2 implementation follows the second approach, since method calls on interface types are very frequent in Java programs. Thus, we can avoid the invocation of the dispatcher method.

5.4.9 Memory Control

Memory control has to limit the allocation of heap memory, as well as the size of the execution stacks of running threads.

5.4.9.1 Heap

Enforcing memory limits requires exact pre-accounting for memory resources, i.e., an overuse exception is raised before a thread can exceed the memory limit of the domain it is executing in. In contrast to JRes [14], which maintains a separate memory limit for each thread, J-SEAL2 enforces a single memory limit for a multithreaded domain or even for a set of domains in the case of resource sharing.

Because a single `MemAccount` object has to maintain the memory consumption and limit of a set of domains sharing the same memory resources, access to the `MemAccount` must be synchronized. Furthermore, accounting for an object as well as its allocation and initialization has to be an atomic action.

Before the object is allocated, J-SEAL2 ensures that the memory limit is not exceeded and updates the `MemAccount`. If the memory allocation fails, if the constructor raises an exception, or if the allocating thread is terminated asynchronously, we have to ensure that the modification of the `MemAccount` is undone. Otherwise, other threads or even other domains (using the same `MemAccount`) could suffer from memory leakage. Details on the rewriting scheme for memory allocation instructions can be found in [7].

When the garbage collector reclaims an object, we have to update the `MemAccount` that has been charged for this object. For this reason, the `MemAccount` maintains a weak reference for each allocated object, which does not prevent the object from being reclaimed. When an object referenced by a weak reference is garbage collected, the weak reference is enqueued in a reference queue, which can be polled by the `MemAccount` implementation (for details see [7]).

Object Size The size of an object is calculated from the number of fields for each Java basic type, the number of fields holding object references, a constant for the object overhead, as well as a constant for the accounting overhead (i.e., the overhead for maintaining a weak reference to the allocated object). For arrays, the actual size must be computed from the array

dimensions available on the execution stack. Depending on the Java runtime system, the overhead for array objects may be larger than for non-array objects, because of the size information stored within arrays.

Constants for the object overhead and for the size of Java basic types and object references are managed in a configuration file by the system administrator. Since in general the administrator does not know the object representation of the underlying Java runtime system, a tool helps to approximate these constants (e.g., by avoiding garbage collection and measuring the difference of allocated memory before and after creating certain types of objects). However, object alignment is not taken into account.

Optimizations While our approach works for objects as well as for arrays, we are also implementing an optimization for non-array objects: Similar to JRes [14], in each allocated object we store a reference to the corresponding `MemAccount` object. Rewritten finalizers are responsible for updating the `MemAccount` when an object is reclaimed by the garbage collector. Thus, we can avoid the significant overhead of maintaining weak references, which is particularly important for small objects.

For arrays, such an optimization cannot be implemented in pure Java. However, in practice the overhead for accounting for allocated arrays is not a serious problem, because arrays frequently are large objects (compared to the accounting overhead they cause).

5.4.9.2 Stack

The computation of recursive methods may rapidly blow up the execution stack of a thread without allocating a single object. Especially if domains are allowed to create large numbers of threads, an attacker could easily create a bunch of threads, and in each thread create a very deep call-stack forcing the system to use large amounts of memory (precious memory, which cannot be garbage collected until the methods return).

Most proposals for resource control in Java, like e.g. JRes [14], do not take the memory consumption of the execution stacks into account. Our implementation supports control of stack memory as an optional feature. During the installation, the system administrator has to decide whether stack control shall be enabled. When untrusted domains are allowed to create only a small number of threads and the underlying JVM allocates execution stacks

that cannot expand dynamically, it is sufficient to charge the `MemAccount` for the maximum stack size[6] when a thread is created.

However, if the JVM allows execution stacks to grow up significantly, special effort is necessary in order to limit the size of the stack. For this purpose, we rewrite non-native methods to pass an additional counter, indicating the amount of memory the thread is allowed to use on the stack. On method entry, this counter has to be reduced by the number of local variables and the maximum stack consumption of the invoked method[7]. For each method, this information is available in the Java class-file [26]. If the counter becomes negative, an appropriate exception is raised. The counter can be an integer that is passed by value. Therefore, a good register allocator will help to keep the overhead small. As a further optimization, leaf methods (i.e., methods that do not invoke any other method) may omit the check of the counter.

5.4.10 CPU Control

For CPU control, we are accounting the number of executed bytecode instructions for each thread running in a CPU-ACC or CPU-MEM-ACC domain. A high-priority scheduler thread, which is part of the J-SEAL2 kernel, executes periodically in order to ensure that assigned CPU limits are respected. The scheduler thread calculates the number of executed bytecode instructions for each set of domains sharing a CPU limit by summing up the CPU consumption of all threads executing in a domain in the set. The scheduler compares the number of executed bytecodes with the desired schedule. If a set of domains has exceeded its CPU limit, the priorities of threads executing in these domains are lowered.

5.4.10.1 Class `CPUAccount`

In contrast to a `MemAccount` object, which is shared by all threads executing in a domain with memory accounting, each thread running in a domain

[6] In order to approximately determine the maximum stack size of a JVM implementation, we employ a calibration program executing a recursive method until a `StackOverflowError` occurs. The maximum stack size corresponds to the product of the maximum recursion depth and the size of a stack frame of the recursive method.

[7] A Just-in-Time compiler will completely remove the Java stack when it creates code for a register machine. Nevertheless, the number of local variables and the maximum stack consumption of a method can be used as an approximation for the size of a stack frame of the method.

with CPU accounting has associated its own `CPUAccount` object. Since CPU accounting occurs very frequently, it is important that multiple threads do not have to synchronize on a common accounting object. As only the scheduler thread makes any scheduling decisions, it is sufficient to account for each thread separately. The scheduler is responsible for accumulating the accounting data of all threads executing in a set of domains sharing a CPU limit.

A `CPUAccount` object simply maintains an integer counter, which is updated by the thread owning the object. Table 5.9 shows some parts of the `CPUAccount` implementation[8]. Because the scheduler thread has to read the counter value, we are using a volatile variable in order to force the JVM to immediately propagate every update from the working memory of a thread to the master copy in the main memory [20, 26].

```
public final class CPUAccount {
  public volatile int usage;
  ...
}
```

Table 5.9: The `CPUAccount` implementation.

In general, updating the counter requires loading the `usage` field of the `CPUAccount` object from memory (it is volatile), incrementing the loaded value accordingly, and storing the new value in the memory. A counter update requires about 6 bytecode instructions.

5.4.10.2 Scheduler

In this section we describe how the scheduler thread computes the CPU consumption of a set of domains, and how it employs different JVM priority levels in order to prevent CPU overuse. However, we do not present a particular scheduling algorithm, because we are still experimenting with different policies.

For each `CPUAccount` object, the scheduler thread always stores the value of the counter it has read most recently. The scheduler calculates the difference between the current value and the previously stored value in order

[8]For instance, we omitted the static `getCurrentAccount` method mentioned in section 5.4.4.

to determine the amount of bytecode instructions executed during the last time-slice (because of the lack of synchronization, the scheduler must not reset any `CPUAccount` object). If a thread has not existed before, the scheduler assumes the previously stored value to be zero. When a thread terminates, its `CPUAccount` object is not disposed of immediately, but it is maintained until the scheduler has examined it.

The scheduler has to deal with an overflow in the counter of a `CPUAccount` object. The size of the counter must be large enough so that its full range cannot be used in a single time-slice. For current JVMs and a reasonably small time-slice, a Java `int` is sufficient. However, in future high-performance systems, `CPUAccount` objects may have to maintain `long` values[9].

We are using different JVM priority levels to control the CPU consumption of individual domains. As protection domains in J-SEAL2 do not have direct access to the class `java.lang.Thread` (they have to use a safe wrapper class instead [6], which does not offer any mechanism to change the priority of a thread), a user-level thread cannot raise its own priority.

Even though the Java language specification [20] does not define any scheduling policy, current JVM implementations respect assigned thread priorities. Many JVMs employ fixed priority scheduling, where a low-priority thread cannot execute, if there is a high-priority thread ready to run. The J-SEAL2 kernel uses the distinct JVM thread priority levels as follows:

- MAX_PRIORITY: The maximum priority is reserved to JVM internal tasks, such as handling weak references. J-SEAL2 does not run any threads with the maximum priority.

- MAX_PRIORITY-1: J-SEAL2 uses this priority level for kernel-level operations in order to prevent priority inversion, i.e., when a high-priority thread is waiting for an exclusive kernel lock (see section 3.2) because of a low-priority thread T executing in kernel mode, the priority of T is temporarily boosted until thread T releases the kernel lock.

- MAX_PRIORITY-2: This priority level is used by the J-SEAL2 scheduler thread.

- NORM_PRIORITY–MIN_PRIORITY[10]: The scheduler assigns these

[9] For a `long` variable, the volatile declaration is crucial, because some JVMs do not treat non-volatile `long` values atomically [26].

[10] In this description we assume that NORM_PRIORITY < MAX_PRIORITY-2.

priority levels to threads according to the CPU consumption of the corresponding domain and the assigned CPU share. Threads that are executing in NO-ACC or in MEM-ACC domains are always assigned NORM_PRIORITY. If a domain exceeds its CPU limit, the priorities of its threads are reduced (or at least the priorities of those threads overusing the CPU). If a domain does not consume its assigned CPU resources, the priorities of its threads may be increased again (but never exceeding NORM_PRIORITY). We are experimenting with different scheduling algorithms regarding the history of CPU consumption.

5.4.10.3 Rewriting Algorithm

In the description of the rewriting algorithm we use the following definition of an accounting block, which is related to the concept of a basic block of code. In order to minimize the accounting overhead, we are considering blocks of maximal length. An accounting block is a bytecode sequence fulfilling the following constraints:

- If a bytecode instruction, which is neither a method/constructor invocation nor a JVM subroutine invocation, changes the control-flow non-sequentially (e.g., method return, exception raising, branch, JVM subroutine return, etc.), it must be the last instruction in the accounting block. That is, with the exception of method/constructor and JVM subroutine invocations, only the last bytecode instruction in the block may change the control-flow non-sequentially. A method invocation does not terminate an accounting block, because otherwise the average block size would be reduced significantly, as method invocations are very frequent in object-oriented programs.

- Only branches to the begin of the block are allowed. There is no bytecode instruction branching to another instruction in the same method, which is not the first one in its block. Furthermore, the first instruction of an exception handler must be always the first instruction in its block.

The bytecode rewriting algorithm involves the following 4 steps (an efficient implementation may perform multiple steps together):

1. Method/constructor invocations are rewritten to pass the `CPUAccount` object as extra argument. Because the `CPUAccount` is always the last argument[11], it can be pushed onto the stack immediately before the method/constructor invocation instruction.

2. An accounting block analysis (similar to a basic block analysis in traditional compilers) partitions the method code into a set of accounting blocks. Each block has an attribute indicating the accounting size of the block. Initially, this attribute holds the number of bytecode instructions in the block[12]. Furthermore, a control-flow graph with the accounting blocks as nodes has to be constructed, if optimizations are to be performed in order to minimize the accounting overhead. Without any optimizations, accounting instructions have to be inserted into every block.

3. Optimizations, such as those presented in following section, analyze the control-flow graph in order to detect situations where acounting for multiple different blocks may be combined. The optimizations may decrement the accounting size attribute of one block and add it to the accounting size of another block. If the accounting size of a block becomes zero, it does not require any accounting instructions.

4. For every block with a positive accounting size, accounting instructions are inserted at the begin of the block. The only exception to this rule is the first block in a constructor: The invocation of another constructor of the same class or of the superclass has to antecede the accounting code. The included instructions add the accounting size of the block plus the number of inserted accounting instructions to the `CPUAccount` object. For performance reasons, updates of the `CPUAccount` object are not synchronized.

This approach ensures that a thread is charged for at least the number of bytecode instructions it executes. For each accounting block, a thread is

[11]Since rewriting for memory accounting is done before rewriting for CPU control, the `MemAccount` argument is passed always before the `CPUAccount` object.

[12]In order to improve the accuracy of measurement, the J-SEAL2 administrator may configure a weighting of bytecode instructions (integer values) according to their complexity. To simplify matters, we assume that all bytecode instructions have a weighting of 1.

charged for the number of instructions in the block, before it executes these instructions (pre-accounting). When an instruction, which is not the last one in its accounting block, raises an exception, the thread has been charged for more instructions than it has consumed. However, since the number of executed bytecode instructions is only an approximation of the exact CPU consumption, and because exception handling is expensive on many JVM implementations, this possible inexactness does not pose any problem.

5.4.10.4 Optimizations

In order to minimize the accounting overhead, the rewriting algorithm may perform certain optimizations. Since CPU accounting is closely related to profiling and tracing of programs, profiling techniques can help to insert accounting instructions in a way to minimize the accounting overhead [4].

If classes are rewritten off-line, such as shared JDK classes (see section 5.4.6), the optimization algorithm may perform some complex and time-consuming analysis. However, for replicated classes, only simple optimizations are possible, since these classes are rewritten on-line. In the following paragraphs we present some simple rules that are well suited for on-line optimization.

In the following optimization O1 we assume that the accounting block B has n ($n > 0$) predecessors A_i ($1 \leq i \leq n$) in the control-flow graph. We denote the accounting size attributes of B and A_i as b and a_i.

O1: If all A_i are different from B, and for each A_i the only successor is B, then all a_i are incremented by b and b is set to zero.

For the following optimizations O2 and O3 we assume that the accounting block A has n ($n > 0$) successors B_i ($1 \leq i \leq n$) in the control-flow graph. We denote the accounting size attributes of A and B_i as a and b_i, the minimum accounting size $\min b_i$ as b_{min}, and the maximum accounting size $\max b_i$ as b_{max}.

O2: If all B_i are different from A, and for each B_i the only predecessor is A, then a is incremented by b_{min} and all b_i are decremented by b_{min}. Consequently, the value of at least one b_i becomes zero.

O3: If all B_i are different from A, and for each B_i the only predecessor is A, and the difference $b_{max} - b_{min}$ does not exceed a given threshold T,

then a is incremented by b_{max} and all b_i are set to zero. Less formally: If the values of the accounting size attributes of successor blocks are not too much different, the common predecessor block accounts for the longest successor block. This optimization is an aggressive version of rule O2. The threshold controls the aggressiveness of this optimization. A threshold T means that a thread executing a block B_i may be charged for up to T bytecode instructions, which it did not execute. In general, T should not be smaller than the number of bytecode instructions necessary to update the `CPUAccount` object (a thread would be charged for the update instructions, if the optimization was not applied). In order to find effective values for the threshold, we can perform static analysis of typical Java programs (the smallest value T allowing to avoid a significant fraction of the accounting code).

The optimization rules O1, O2, and O3 aim at combining the accounting for a set of blocks that represent conditional statements, but they do not allow to remove the accounting code from loops. For instance, rules O1 and O2 (or alternatively, O1 and O3) may be applied to optimize the accounting for `if-else` statements. However, these rules are not sufficient to reduce the accounting overhead for `if` statements without a matching `else`. Therefore, we are working on further optimizations.

In general, multiple optimization rules can be applied to a given control-flow graph. The order of application is important, since it may affect the quality of the accounting code. Most importantly, the optimization algorithm must ensure termination. In particular, certain loops allow an infinite application of rule O1. The following heuristics help to guide the optimization process:

- An optimization rule may be applied only if the application increases the number of blocks with an accounting size attribute of zero. Since the number of blocks in a method is finite, obeying this rule ensures termination of the optimization algorithm.
- Optimization O1 shall be applied before optimizations O2 and O3.
- Optimization O3 shall be applied before optimization O2. There is no need to apply optimization O2, if optimization O3 (which is more aggressive) succeeds on a certain node in the control-flow graph.

- If there are leaf nodes in the control-flow graph, they should be considered first, afterwards their predecessor nodes, etc.

While optimizations O1, O2, and O3 aim at removing accounting code from certain blocks, the following rule O4 helps to reduce the overhead of accounting by caching the counter maintained by the `CPUAccount` in a local variable. This optimization improves performance only for certain JVM implementations (measurements are given in section 5.5). Optimization O4 must be considered after application of the rules O1, O2, and O3.

O4: In general, a block with a positive accounting size requires accounting instructions to load, update, and store the `usage` field of the `CPUAccount` object (see section 5.4.10.1). We introduce a local variable `localUsage` caching the value of the `usage` field in order to avoid reloading this field in every accounting block. The following algorithm marks exactly those accounting blocks that have to reload the `usage` field of the `CPUAccount` object. All other blocks may directly update the `localUsage` variable and propagate the new value to the `usage` field of the `CPUAccount` object.

- Initially, we mark the first block in the method, in each JVM subroutine, and in each exception handler.
- If a block contains a method/constructor invocation, all of its successors in the control-flow graph are marked.
- If a block with an accounting size attribute of zero is marked, all of its successors have to be marked as well.

The algorithm terminates, if no further blocks can be marked.

5.4.11 Accounting for Garbage Collection

In order to prevent denial-of-service attacks by causing the garbage collector to consume a considerable amount of CPU time (e.g., an attacker may create a lot of garbage without exceeding its memory limit), the J-SEAL2 kernel has to account for the time spent by the garbage collector. Only CPU-MEM-ACC domains can be charged for the garbage they produce, because accounting for garbage collections requires the information, which domain has allocated a certain object (such information is not available in NO-ACC or CPU-ACC

domains), and because the time spent by the garbage collector affects the CPU consumption of a domain (CPU consumption is not measured in NO-ACC or MEM-ACC domains).

Since the exact CPU time spent by the garbage collector is not known, we are using an abstract measure. The J-SEAL2 administrator defines a rough approximation of the number of bytecode instructions required to reclaim an object. Before an object is allocated, the J-SEAL2 kernel charges the `CPUAccount` object of the allocating thread. That is, a domain has to 'pay' for the garbage it eventually will produce at the time it 'buys' an object. This simple approach has the advantage that a CPU-MEM-ACC domain is charged for all garbage it produces, even if the domain has already terminated when some objects are reclaimed.

5.4.12 Compensating for Native Code

With the aid of bytecode rewriting techniques, it is not possible to account for memory allocation and CPU consumption in native code. Untrusted applications are not allowed to bring native code libraries into the system. Concerning JVM-provided standard operations, the J-SEAL2 kernel tries to compensate for resources used by native code and prevents untrusted domains from using certain functionality leading to a significant resource consumption by native code. In the following we describe some important cases of resource consumption in native code and how J-SEAL2 solves them:

- Class-loading: The Java runtime system manages an internal table of loaded classes. Memory for compiled methods is allocated by the Just-in-Time compiler, which is usually implemented in native code. However, the set of classes untrusted domains (e.g., mobile agents) are allowed to access is limited and known to the J-SEAL2 kernel. Therefore, the kernel accounts for the classes using an approximation, which is proportional to the size of the class-files.

- Deserialization: J-SEAL2 uses Java serialization in order to create messages to be transferred across domain boundaries. When the receiving domain opens a message, it is being deserialized using the class-loader of the receiving domain to resolve class names. The class `java.io.ObjectInputStream` employs native methods to allocate objects without invoking their constructors. J-SEAL2 solves this hurdle

by storing the amount of objects for each type, which is part of the serialized object graph, in the message. The receiver performs resource checks before deserializing the message. Note that we are not directly storing the size of the message, because the message may be deserialized on a different host using a JVM with a distinct object representation (e.g., consider a mobile agent carrying a message to another location).

- Object cloning: Java supports a way to create a shallow copy of an object of a type implementing the interface `java.lang.Cloneable`. The shallow copy is allocated by a native method. A simple solution is to forbid untrusted domains to use object cloning. Another somewhat more complicated approach is to rewrite invocations of the `clone` method accordingly.

- Reflection: The Java reflection API provides a mechanism to indirectly create a new instance of a class. The `newInstance` method of the class `java.lang.reflect.Constructor` is native. J-SEAL2 prevents untrusted domains from using the reflection API. However, note that objects allocated by a constructor invoked with the aid of the `newInstance` method would be accounted for (see section 5.4.5).

5.5 Evaluation

Because the integration of our resource control model in J-SEAL2 is still in progress, we are currently not able to provide performance and scalability evaluations of real applications running in a J-SEAL2 environment with resource control. Nevertheless, in this section we present some performance measurements proving that the overhead due to accounting is acceptable on modern JVM implementations.

While in J-SEAL2 the overhead for memory control is comparable to the overhead caused by JRes[13] [14], the overhead of CPU control based on bytecode rewriting techniques has to be examined carefully, because such an approach has not been used before.

JRes [14] uses native code for CPU accounting, although the authors mention that CPU accounting could be accomplished with the aid of byte-

[13]For an application allocating a new object every 250 bytecode instructions, the overhead for memory control is less than 18%, if no memory limit is exceeded.

code rewriting techniques. The authors argued that the resulting execution time would be prohibitive when a reasonable degree of accuracy was to be achieved. However, our initial performance measurements show that the overhead due to our completely portable implementation of CPU accounting is not prohibitive on modern JVM implementations[14].

We have implemented a bytecode rewriting tool that performs the necessary transformations of Java classes to support resource control. The tool was designed to add resource accounting instructions into arbitrary Java applications, to create an extended version of the JDK, and to modify mobile agent applications in J-SEAL2. Our current bytecode rewriting tool supports off-line transformations of arbitrary Java classes.

There are several low-level bytecode engineering frameworks written in Java (e.g., BCA [24], JOIE [13], BIT [25]), as well as higher-level frameworks, such as e.g. Javassist [12]. Our bytecode rewriting tool is based on BCEL (Byte Code Engineering Library, formerly called JavaClass) [15], which allows bytecode manipulations of Java classes and is also entirely written in Java. We chose BCEL since it is one of the most mature bytecode intrumentation frameworks and provides a powerful and intuitive API that is well adapted for our requirements.

We measured the standard SPEC JVM98 benchmarks [38] on a Linux platform (Intel Pentium III, 733MHz clock rate, 128MB RAM, Linux kernel 2.2.16) with IBM's JDK 1.3 implementation, which includes one of the best Just-in-Time compilers available today. We measured the overhead due to CPU accounting in three different configurations:

- U_{bench}-U_{jdk}: Unmodified benchmarks on an unmodified JDK.
- R_{bench}-U_{jdk}: Rewritten benchmarks on an unmodified JDK.
- R_{bench}-R_{jdk}: Rewritten benchmarks on a rewritten JDK[15].

For each measurement, table 5.10 shows the execution time of the benchmark in seconds (rounded to 3 decimal places), as well as the speedup of the original code compared to the rewritten version (rounded to 2 decimal

[14] We are not measuring the overhead for CPU control incurred by the scheduler, as it can always be kept small by choosing an appropriate time-slice.

[15] Modern JVMs allow to run user-defined library classes with the `-Xbootclasspath` option.

Benchmark	U_{bench}-U_{jdk}		R_{bench}-U_{jdk}		R_{bench}-R_{jdk}	
_227_mtrt	5,823	(1,00)	7,336	(1,26)	7,685	(1,32)
_202_jess	7,779	(1,00)	9,145	(1,18)	11,590	(1,49)
_201_compress	19,130	(1,00)	23,156	(1,21)	23,468	(1,23)
_209_db	26,740	(1,00)	27,777	(1,04)	31,352	(1,17)
_222_mpegaudio	8,694	(1,00)	12,425	(1,43)	12,575	(1,45)
_228_jack	8,184	(1,00)	8,771	(1,07)	11,487	(1,40)
_213_javac	14,150	(1,00)	15,618	(1,10)	22,853	(1,62)
Geometric Mean	11,286	(1,00)	13,296	(1,18)	15,514	(1,37)

Table 5.10: SPEC JVM98 benchmarks measuring the overhead of CPU accounting (time in seconds).

places). All results represent the median of 101 different measurements. Furthermore, we also computed the geometric mean for each configuration. We rewrote about 520 Java class-files for the CPU-aware version of the SPEC JVM98 benchmarks, and about 5400 class-files for the extended version of the JDK.

The results in table 5.10 show that the overhead due to CPU accounting is about 40%, if we rewrite applications as well as the whole JDK. With an unmodified JDK, the overhead can be halved. Note that we did not apply any optimizations to reduce the accounting overhead. Since the implementation of the optimization algorithm is still in progress, we could not evaluate the performance of the optimized rewritten code with the standard JVM98 benchmarks. Thus, we measured the following well-known micro-benchmarks, which we rewrote by hand:

Fib: This is the recursive algorithm for the calculation of fibonacci numbers. We used this benchmark to calculate the 35th fibonacci number.

Sort: This is bubble-sort, an iterative sorting algorithm for arrays. It consists of 2 nested loops, where the inner loop exchanges 2 adjacent array elements, if they are not in the desired order. We used this benchmark to sort an array of 10000 `int` values in ascending order. Initially, the input array was sorted in descending order.

Table 5.11 summarizes our measurements, which were collected on a Windows NT 4.0 workstation (Intel Pentium II, 400MHz clock rate) with 3 different JVM implementations. In order to minimize the impact of compilation

		Sun JDK 1.2.2 Classic (JIT)		Sun JDK 1.2.2 Hotspot Server 2.0		IBM JDK 1.3 Classic (JIT)	
Fib	original	1031	(1,00)	1032	(1,00)	991	(1,00)
	rewritten	1773	(1,72)	1502	(1,46)	1522	(1,54)
	O3	1432	(1,39)	1232	(1,19)	1131	(1,14)
Sort	original	1212	(1,00)	1352	(1,00)	782	(1,00)
	rewritten	2434	(2,01)	2564	(1,90)	2323	(2,97)
	O1+O3	1752	(1,45)	2143	(1,59)	1623	(2,08)
	O1+O3+O4	1513	(1,25)	2294	(1,70)	1543	(1,97)

Table 5.11: Micro-benchmarks measuring the overhead of CPU accounting (time in milliseconds).

and garbage collection, all results represent the median of 5 different measurements. For each measurement, table 5.11 shows the execution time of the benchmark in milliseconds, as well as the speedup of the original code compared to the rewritten version. We measured code rewritten without any optimizations, as well as the code resulting from the application of the optimizations rules presented in section 5.4.10.4.

The fibonacci benchmark consists of 5 accounting blocks. Optimization O3 with a minimum threshold $T_{min} = 11$ allows to combine accounting for the whole method in the first block, i.e., this optimization avoids 80% of the accounting code. Because the optimized fibonacci method accounts for all instructions in the first block of the method, optimization rule O4 cannot reduce the accounting overhead anymore.

Our measurements for the recursive fibonacci method show that our optimizations allow to reduce the accounting overhead to 14–19% on modern JVM implementations, such as Sun's Hotspot VM and IBM's Classic VM. These results also indicate that the overhead of passing the additional `CPUAccount` argument is reasonably small.

The bubble-sort method comprises 10 accounting blocks. A combination of the optimization rules O1 and O3 with a minimum threshold $T_{min} = 6$ allows to remove the accounting code from 5 blocks, i.e., these optimizations avoid 50% of the accounting code. Furthermore, the optimization O4 can be applied to all accounting blocks but the first one in the method.

The bubble-sort benchmark shows that optimization rule O4 is beneficial only on some JVM implementations, such as Sun's and IBM's Classic VMs,

whereas on Sun's Hotspot VM this rule has a bad impact on the performance. While the performance on IBM's Classic VM suffers significantly from the volatile `CPUAccount` counter (removing the volatile declaration reduces the accounting overhead drastically), the performance impact of the volatile variable is rather small on Sun's Hotspot VM. Nevertheless, for our benchmark programs, IBM's JVM implementation offers the best overall performance in absolute terms.

Chapter 6

Conclusion

This chapter concludes the thesis. Section 6.1 presents the current state of implementation of the J-SEAL2 mobile agent system, section 6.2 gives a glimpse on future investigations, and section 6.3 summarizes the main contributions of this thesis.

6.1 State of Implementation

The first version of the J-SEAL2 kernel was completed in autumn 1999. It supported the efficient external reference communication model (see section 4.4), but like JavaSeal [42, 11], it did not provide any mechanisms for resource control. After an extensive period of testing, the stable kernel was packaged and released in January 2000. This release included some exemplary service components, such as a network service for agent migration, a graphical user interface service based on a HTML browser component enabling agents to interact with users, as well as some simple IO services. In early 2000 we profiled benchmarks and demo applications running under J-SEAL2. Our measurements revealed some hotspots in the kernel, such as the kernel locking mechanism explained in section 3.2.

After some considerable optimization efforts, we released the current version of the J-SEAL2 kernel in March 2000. This release, which has been evaluated and tested by many researchers throughout the world, has proven to be highly stable and efficient. Recently, this version of the J-SEAL2 kernel has been used for the implementation of an agent-based workflow trading

package	description	classes	lines of code	bytes
seal.sys	kernel API	13	2304	22481
seal.sys.exception	exception classes	12	137	2031
seal.sys.extref	external references	4	921	6866
seal.sys.directive	directives parser	8	3154	34406
seal.sys.kernel	internal kernel classes	20	2590	32352
seal.sys.kernel.caps	optimized capsule classes	20	1675	13342
seal.sys.loader	class-loader and verifier	7	1280	14554
seal.sys.util	utility classes	3	300	3808
seal.framework.iamc	Inter Agent Method Calling	3	507	7252
seal.framework.init	service registration protocol	1	137	2287
seal.framework.interf	local service registry	7	275	7056
total		98	13280	146435

Table 6.1: J-SEAL2 core components.

architecture[1] [29, 30, 31].

Table 6.1 summarizes the core components of the current J-SEAL2 release. For each package of the kernel and of the libraries, the table gives a short description, the number of classes and interfaces, the number of lines of code including full JavaDoc documentation and comments, as well as the size of the compiled class-files in bytes. For the agent programmer, only the libraries and the packages `seal.sys`, `seal.sys.exception`, and `seal.sys.extref` are available. All other packages are used only internally by the J-SEAL2 kernel. A minimal configuration for devices with limited resources may be obtained by omitting the libraries and the packages `seal.sys.extref`, `seal.sys.directive`, and `seal.sys.kernel.caps`. However, in such a configuration it is not possible to employ the efficient external reference communication model.

Readers interested in getting an evaluation version of the J-SEAL2 platform, including network and GUI services, developer documentation, as well as some small demonstration applications, may contact the author by E-mail[2].

[1] Detailed information on this project is available via WWW at URL: `http://anaisoft.unige.ch/`

[2] The author's current E-mail address is `w.binder@coco.co.at`.

6.2 Future Work

Since autumn 2000 we are working on the resource control model for J-SEAL2 together with the TiOS group at the University of Geneva. The design phase was finished in October 2000, a recent article comprises the resulting model [8]. Initial performance measurements proving the feasibility of our approach are given in section 5.5.

We have implemented a bytecode rewriting tool for off-line rewriting of the JDK and arbitrary Java applications in order to evaluate the performance impact of CPU accounting. The tool is not yet complete, as memory accounting and optimizations to reduce the performance penalty of CPU accounting are not fully implemented. Rory Vidal, a student at the University of Geneva who is preparing his diploma thesis, has implemented the bigger part of the bytecode rewriting tool. Together with Jarle Hulaas and Alex Villazón, co-designers of the resource control model, we are adapting the J-SEAL2 micro-kernel for resource control and integrating the resulting bytecode rewriting tool into the kernel's class-loader.

Regarding CPU control, we are experimenting with different scheduling algorithms in order to find the best trade-off between accounting accuracy and little overhead. Concerning the CPU accounting scheme with its many optimization tricks, we have to show that no denial-of-service attack will get unnoticed, and that a client will not be charged for much more than it actually consumed. Moreover, we are also considering an additional implementation of the J-SEAL2 architecture on top of implementations of the Real-Time Specification for Java [9], which contains certain concepts that can be used for resource control.

Also on our immediate todo-list is the development of high-level programming tools in order to support a friendlier event notification mechanism than the overuse exceptions generated by the J-SEAL2 kernel. User-specified thresholds should enable applications to receive warnings in a timely manner before the actual overuse happens.

For a high-level programming model, it is also important to have a simple uniform mechanism to control all kinds of resources. Whereas the resource control API presented in section 5.3 only deals with kernel-managed resources, the high-level model has to include resource control for service access (e.g., network and file IO), which is based on an extension of the communication model of chapter 4. This extension provides a homogeneous API

to associate security policies with external references, which can be used, for example, to limit the size of communication messages, to restrict the number of messages, or to limit the communication bandwidth.

Other important features to be offered in a future J-SEAL2 release include remote configuration and administration without disruption of the agent platform. We are integrating Alex Villazón's meta-programming model [41] into the J-SEAL2 kernel, which can be used, for instance, to provide a flexible architecture for debugging and monitoring mobile agent applications. A graphical user interface will allow to monitor and configure multiple J-SEAL2 platforms from a single place. A load-balancing service will help to maintain server clusters for large-scale applications.

6.3 Summary

We have presented design and implementation issues that must be addressed by Java-based mobile agent platforms. Security, portability, and high performance are crucial for the success of a mobile agent system in large-scale distributed commercial applications. For security reasons, a mobile agent system has to be structured in a similar way as an operating system, where the kernel is separated clearly from all other parts of the system. The kernel is responsible for protection, communication, protection domain termination, and resource control.

The J-SEAL2 mobile agent system is based on a micro-kernel architecture providing the necessary security features for commercial mobile agent applications. The J-SEAL2 kernel is implemented in pure Java, thus it is portable over different operating systems and hardware platforms. J-SEAL2 offers strong protection domains and safe domain termination with immediate resource reclamation.

The J-SEAL2 micro-kernel supports a new communication model, allowing efficient communication in a hierarchy of protection domains. The communication model ensures security, it prevents direct sharing of object references with distinct protection domains, and ensures that communication paths may be invalidated at any time. A high-level communication protocol implemented on top of the kernel supports convenient Inter Agent Method Calling (IAMC). Performance measurements prove that J-SEAL2 communication incurs only small overhead and scales well.

In order to prevent denial-of-service attacks, currently we are integrating

a new resource control model into the J-SEAL2 kernel, covering physical resources, such as CPU and memory, as well as logical resources, like threads and subdomains. In order to maintain complete compatibility and portability, the implementation is based on bytecode rewriting techniques. Initial performance measurements back our approach.

As part of this thesis, the author designed and implemented the current release of the J-SEAL2 kernel, including the external reference communication model, and developed techniques enabling portable resource control in Java.

Acknowledgements

Many thanks to my PhD supervisors Rudolf Freund and Andreas Krall, who supported me throughout my study, for enlightening discussions and comments on earlier drafts; to Klaus Rapf for contributing many ideas and for enabling my research on mobile agent systems in his company; to Jarle Hulaas and Alex Villazón for a fruitful collaboration on the design and implementation of the resource control model; to Rory Vidal for his work on a bytecode rewriting tool for CPU accounting; to Jan Vitek for contributing elegant solutions to some difficult problems; to Ciarán Bryce for inspiring discussions regarding the communication model; to Julien Francioli and Patrik Mihailescu for helpful comments on early drafts dealing with techniques for resource control.

Bibliography

[1] G. Back and W. Hsieh. Drawing the red line in Java. In *Seventh IEEE Workshop on Hot Topics in Operating Systems*, Rio Rico, AZ, USA, Mar. 1999.

[2] G. Back, W. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI'2000)*, San Diego, CA, USA, Oct. 2000.

[3] G. Back, P. Tullmann, L. Stoller, W. Hsieh, and J. Lepreau. Techniques for the design of Java operating systems. In *Proceedings of the 2000 USENIX Annual Technical Conference*, San Diego, CA, USA, June 2000.

[4] T. Ball and J. Larus. Optimal profiling and tracing of programs. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 59–70. ACM Press, Jan. 1992.

[5] W. Binder. J-SEAL2 – A secure high-performance mobile agent system. In *IAT'99 Workshop on Agents in Electronic Commerce*, Hong Kong, Dec. 1999.

[6] W. Binder. Design and implementation of the J-SEAL2 mobile agent kernel. In *The 2001 Symposium on Applications and the Internet (SAINT-2001)*, San Diego, CA, USA, Jan. 2001.

[7] W. Binder, J. Hulaas, and A. Villazón. Resource control in J-SEAL2. Technical Report Cahier du CUI No. 124, University of Geneva, Oct. 2000. `ftp://cui.unige.ch/pub/tios/papers/TR-124-2000.pdf`.

[8] W. Binder, J. Hulaas, A. Villazón, and R. Vidal. Portable resource control in Java: The J-SEAL2 approach. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, Tampa Bay, Florida, USA, Oct. 2001.

[9] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java.* Addison-Wesley, Reading, MA, USA, 2000.

[10] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. Web pages at `http://www.w3.org/TR/1998/REC-xml-19980210`, Feb. 1998.

[11] C. Bryce and J. Vitek. The JavaSeal mobile agent kernel. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA, Oct. 1999.

[12] S. Chiba. Load-time structural reflection in Java. In *ECOOP*, pages 313–336, 2000.

[13] G. Cohen, J. Chase, and D. Kaminsky. Automatic program transformation with JOIE. In *1998 USENIX Annual Technical Symposium*, pages 167–178, 1998.

[14] G. Czajkowski and T. von Eicken. JRes: A resource accounting interface for Java. In *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98)*, volume 33, 10 of *ACM SIGPLAN Notices*, pages 21–35, New York, USA, Oct. 18–22 1998. ACM Press.

[15] M. Dahm. Byte code engineering. In *Java-Information-Tage 1999 (JIT'99)*, Sept. 1999. `http://bcel.sourceforge.net/`.

[16] F.-X. Le Louarn. JUM, a Java Usage Monitor. Web pages at `http://www.iro.umontreal.ca/~lelouarn/jum.html`.

[17] B. Ford, M. Hibler, J. Lepreau, R. McGrath, and P. Tullmann. Interface and execution models in the fluke kernel. In *Proceedings of the Third*

Symposium on Operating Systems Design and Implementation (OSDI-99), pages 101–116, Berkeley, CA, USA, Feb. 22–25 1999. Usenix Association.

[18] B. Ford and S. Susarla. CPU Inheritance Scheduling. In *Usenix Association Second Symposium on Operating Systems Design and Implementation (OSDI)*, pages 91–105, 1996.

[19] M. Godfrey, T. Mayr, P. Seshadri, and T. von Eicken. Secure and portable database extensibility. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD-98)*, volume 27,2 of *ACM SIGMOD Record*, pages 390–401, New York, USA, June 1–4 1998. ACM Press.

[20] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

[21] C. Hawblitzel and T. Von Eicken. Tasks and revocation for Java (or, hey! you got your operating system in my language!). Draft, Nov. 1999. `http://www.cs.cornell.edu/Info/People/hawblitz/hawblitz.html`.

[22] C. Hawblitzel and T. Von Eicken. Type system support for dynamic revocation. In *ACM SIGPLAN Workshop on Compiler Support for System Software*, May 1999. `http://www.cs.cornell.edu/Info/People/hawblitz/hawblitz.html`.

[23] J. Hulaas, L. Gannoune, J. Francioli, S. Chachkov, F. Schütz, and J. Harms. Electronic commerce of internet domain names using mobile agents. In *Proceedings of the Second International Conference on Telecommunications and Electronic Commerce (ICTEC'99)*, Nashville, TN, USA, Oct. 1999.

[24] R. Keller and U. Hölzle. Binary component adaptation. In E. Jul, editor, *ECOOP '98—Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 307–329. Springer, 1998.

[25] H. B. Lee and B. G. Zorn. BIT: A tool for instrumenting Java bytecodes. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (ITS-97)*, pages 73–82, Berkeley, Dec. 8–11 1997. USENIX Association.

[26] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[27] L. Moreau and C. Queinnec. Design and semantics of Quantum: a language to control resource consumption in distributed computing. In *Usenix Conference on Domain-Specific Languages (DSL'97)*, pages 183–197, Santa-Barbara, CA, USA, Oct. 1997.

[28] K. Nilsen. Java for real-time. *Real-Time Systems Journal*, 11(2), 1996.

[29] A. Schacke. Agentenbasierte Realisierung eines föderierten Handelssystems für Workflowausführungen. Diploma thesis, IFI, University of Zurich, 2001.

[30] M. Schönhoff and H. Stormer. Trading workflows electronically: the ANAISOFT architecture. In *Proceedings of Datenbanksysteme in Büro, Technik und Wissenschaft (BTW'2001)*, Oldenburg, Germany, Mar. 2001.

[31] H. Stormer. Task scheduling in agent-based workflow. In *International ICSC Symposium on Multi-Agents and Mobile Agents in Virtual Organizations and E-Commerce (MAMA'2000)*, Wollongong, Australia, Dec. 2000.

[32] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time compiler. *IBM Systems Journal*, 39(1):175–193, 2000.

[33] Sun Microsystems, Inc. Enterprise JavaBeans Technology. Web pages at `http://java.sun.com/products/ejb/`.

[34] Sun Microsystems, Inc. JAVA 2 Platform, Standard Edition. Web pages at `http://java.sun.com/j2se/1.3/`.

[35] Sun Microsystems, Inc. Java Servlet Technology. Web pages at `http://java.sun.com/products/servlet/`.

[36] Sun Microsystems, Inc. Java Virtual Machine Profiler Interface (JVMPI). Web pages at `http://java.sun.com/j2se/1.3/docs/guide/jvmpi/index.html`.

[37] N. Suri, J. M. Bradshaw, M. R. Breedy, P. T. Groth, G. A. Hill, R. Jeffers, T. S. Mitrovich, B. R. Pouliot, and D. S. Smith. NOMADS: toward a strong and safe mobile agent system. In C. Sierra, G. Maria, and J. S. Rosenschein, editors, *Proceedings of the 4th International Conference on Autonomous Agents (AGENTS-00)*, pages 163–164, NY, June 3–7 2000. ACM Press.

[38] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. Web pages at `http://www.spec.org/osg/jvm98/`, 1998.

[39] C. F. Tschudin. Open resource allocation for mobile code. In *Proceedings of The First Workshop on Mobile Agents*, Berlin, Germany, Apr. 1997.

[40] P. Tullmann and J. Lepreau. Nested Java processes: OS structure for mobile code. In *Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, Sept. 1998.

[41] A. Villazón. A Reflective Active Node. In H. Yasuda, editor, *Active Networks. Second International Working Conference on Active Networks (IWAN 2000)*, volume 1942 of *Lecture Notes in Computer Science*, pages 87–101, Tokyo, Japan, 2000. Springer-Verlag.

[42] J. Vitek, C. Bryce, and W. Binder. Designing JavaSeal or how to make Java safe for agents. Technical report, University of Geneva, July 1998. `http://cui.unige.ch/OSG/publications/OO-articles/TechnicalReports/98/javaSeal.pdf`.

[43] J. Vitek and G. Castagna. Seal: A framework for secure mobile computations. In *Internet Programming Languages*, 1999.

[44] T. Von Eicken, C.-C. Chang, G. Czajkowski, and C. Hawblitzel. J-Kernel: A capability-based operating system for Java. *Lecture Notes in Computer Science*, 1603:369–394, 1999.

[45] T. Wilkinson. Kaffe - a Java virtual machine. Web pages at `http://www.transvirtual.com/`.

[46] F. Yellin. Low level security in Java. In *Fourth International Conference on the World-Wide Web*, MIT, Boston, USA, Dec. 1995.