Design and Implementation of the J-SEAL2 Mobile Agent Kernel

Walter Binder
CoCo Software Engineering GmbH
Margaretenstr. 22/9, A-1040 Vienna, Austria
w.binder@coco.co.at

Abstract

J-SEAL2 is a secure, portable, and efficient execution environment for mobile agents. 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. This paper describes the key concepts of the J-SEAL2 micro-kernel and how they are implemented in pure Java.

1. Introduction

Currently, an increasing number of research projects explores mobility in object-oriented systems. Mobile objects, usually referred to as mobile agents, offer 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. Other advantages of mobile computations include the support for offline operation, which allows users of mobile computing devices to minimize their connection costs. 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 object 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, etc.), mobile agent platforms must protect the host from malicious agents, as well as each agent from any other agent in the system. In order to support large-scale distributed applications, mobile agent systems have to be portable and to offer good scalability and performance. Currently, most mobile agent platforms fail to provide sufficiently strong security models, are limited to a particular hardware architecture and operating system, or cause high overhead. In contrast, the design and implementation of the J-SEAL2 mobile agent system, first described in [3], reconcile strong security mechanisms with portability and high performance.

Recently, a large number of Java [7] based mobile agent systems has emerged. 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, such as portable code, language safety and classloader namespaces for isolation, serialization for state capture, and multithreading. In addition to this, high performance Java runtime systems are available for many hardware platforms and operating systems.

Despite these advantages, current Java environments do not provide sufficiently strong security mechanisms for the protection and isolation of mobile agents. Since the vast majority of current mobile agent platforms simply relies on the insufficient security features of Java, these systems are not suited for commercial mobile agent applications. There are no guarantees that the system and mobile agents are protected from each other. Because of pervasive aliasing in the Java Development Kit (JDK), there is no real protection boundary between different components. Furthermore, malicious agents can easily mount denial of service attacks against the platform, possibly even crashing the Java Virtual Machine (JVM) [8]. Moreover, if a misbehaving agent is detected, the platform does not guarantee that the agent can be safely removed from the system.

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 [3] en-

0-7695-0942-8/01 $10.00 © 2001 IEEE

sures 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 (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.

This article is structured as follows: In section 2 we give a brief overview of the J-SEAL2 mobile agent system. In section 3 we state requirements for kernel code written in Java and show how kernel entry and exit can be implemented efficiently. The following 4 sections deal with protection, communication, protection domain termination, and resource control. In each section we state our requirements for the J-SEAL2 kernel and discuss various implementation issues and techniques that help to meet the requirements efficiently. In section 8 we compare the J-SEAL2 kernel to related work. The last section summarizes the current state of implementation of our mobile agent kernel.

2. System overview

J-SEAL2 [3] is a complete redesign of JavaSeal [4], a secure mobile agent system developed at the University of Geneva. 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, Jan Vitek's SEAL Calculus [9]. JavaSeal enables the expression and effective enforcement of security policies, but it incurs rather 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 yields much better performance. J-SEAL2 provides an efficient 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.

The J-SEAL2 micro-kernel maintains a tree hierarchy of agents and services. Each agent and service executes in a protection domain of its own, called a sealed object or *seal* for short. The root of the tree hierarchy is the RootSeal, which is responsible for starting system services and applications. Seals are multithreaded domains. Every seal can run an arbitrary number of *strands* (secure threads) concurrently. The J-SEAL2 kernel ensures that a parent seal may terminate its children at any time. As a consequence, all strands in the child domain are stopped and all memory resources used by the child seal become eligible for garbage collection immediately.

The J-SEAL2 kernel acts as a reference monitor, it ensures that there is no sharing of object references with different seals. Communication between distinct seals requires kernel primitives, objects are passed always by deep copy within so-called *capsules*. J-SEAL2 offers 2 different communication mechanisms, *channels* and *external references*. Channels only supports 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.

3. Kernel code

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 (strands), as well as communication facilities (channels and external references).

3.1. 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. 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 `ThreadDeath` and `OutOfMemoryError`.

`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 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. 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 strand initiates a kernel operation, it enters a privileged kernel mode. When the kernel operation completes (succeeds or fails), the strand leaves the kernel mode and continues execution in user mode.

The main purpose of the kernel mode in J-SEAL2 is to prevent a strand from being stopped while accessing kernel internals. Stop requests are deferred until the strand 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, strand creation, as well as communication requests.

We distinguish between exclusive and shared kernel mode. A strand entering exclusive kernel mode is blocked until no other strand is executing in the kernel. While a strand is executing in exclusive kernel mode, no other strand can enter the kernel. Protection domain termination is always performed in exclusive kernel mode. Therefore, strands 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 strand waiting for the write lock will enter the kernel before strands waiting for a read lock. This property makes sure that domain termination cannot be delayed infinitely.

Since classloading affects the internal state of the Java runtime system, we must ensure that classloading always occurs in kernel mode. A strand must not be stopped while it is loading a class, since this might corrupt some internal structures of the JVM. However, in general classloading occurs asynchronously, dependent on the virtual machine implementation. Therefore, we do not know whether a classloading strand 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 strand is not yet executing in kernel mode. Implementing this operation requires keeping track of all strands 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 strands to and removing strands from the set of strands 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. Because operations requiring exclusive kernel mode are not executed frequently, the kernel lock does not become a significant performance bottleneck. Resource control (see section 7) ensures that an agent cannot enter the kernel arbitrarily. Therefore, the kernel lock cannot be abused for denial of service attacks.

4. 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.

4.1. Requirements

Language safety in Java (a combination of strong typing, memory protection, automatic memory management, and byte-code verification) 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 (strands), as well as all objects allocated by these strands. In general, strands shall only execute in their own protection domain. Special precaution is necessary to allow strands 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 in only 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 byte-code 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 byte-code verification.

4.2. Implementation issues

The J-SEAL2 kernel employs a separate classloader 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 classloader. All other classes are loaded by the protection domain classloader (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 3 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 classloader 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 classloader 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.

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 subpackages.
- 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.

Benchmarks measuring agent startup overhead indicate that verification overhead is less than 9% of the CPU time spent for classloading. These measurements also include the costs for additional verification to ensure proper domain termination (see section 6).

5. Communication

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.

5.1. Requirements

As we have stressed in the previous section, 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.

Channels support communication only between 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 [9]. 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. However, channel communication becomes a significant performance bottleneck, if messages must be routed through a series of protection domains, since communication involves strand switches proportional to the communication partners' distance in the seal hierarchy.

External references are an optimization, they support indirect sharing of objects between distinct seals that are not necessarily neighbours in the hierarchy. An external reference acts as a capability to invoke methods on a shared object. 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., strands 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 [10]. Details on the external reference communication model can be found in [3].

5.2. 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 classloader, 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 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 [3].

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 strand. 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.

6. 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.

6.1. Requirements

When a protection domain is terminated, all strands 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 section 3 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 strand is stopped asynchronously. The idea is to defer stop requests if the strand to be stopped is accessing the kernel. A simple solution is to ensure that no other strand can access the kernel while a strand 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 byte-code 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.

6.2. Implementation issues

Safe strand stopping is achieved through special kernel entry and exit sequences. A strand terminating a protection domain enters exclusive kernel mode. It is blocked until all other strands have left the kernel. Terminating a protection domain is an atomic kernel operation. All strands belonging to the domain are stopped within the same kernel operation. Since the strands to be stopped cannot enter the kernel, this approach enforces safe domain termination. Details can be found in section 3.

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 [8]. 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 byte-code 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 byte-code sequence (4 JVM instructions) to rethrow a caught

`ThreadDeath` exception immediately.

The rewritten exception handlers rethrow `ThreadDeath` exceptions before any locks are released. This is important 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 strand 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), 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 4, does not exceed 10% of the total time for classloading.

7. Resource control

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 ensures that memory limits are not exceeded. A mobile agent kernel must prevent denial of service attacks, such as agents allocating all available memory. For this purpose, accounting of physical resources (e.g., CPU, memory, network, etc.) and logical resources (e.g., number of threads, number of protection domains, etc.) is crucial.

7.1. Requirements

A generalization of CPU Inheritance Scheduling [6], a hierarchical scheduling protocol, fits very well to the J-SEAL2 model of nested protection domains. At system startup the root domain, RootSeal, owns all physical and logical resources, for example 100% CPU time, some amount of virtual memory, the maximum number of strands the underlying JVM is able to cope with, an unlimited number of subdomains, etc. When a nested protection domain is created, the creator donates some part of its own resources to the new domain.

The J-SEAL2 kernel has to control the consumption of all resources it manages (CPU, memory, strands, domains). However, it is not responsible for limiting network usage, 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 control according to application-specific security policies.

Since J-SEAL2 is designed for large-scale applications, where a large number of services and 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.

In certain situations protection domains that are neighbours in the hierarchy may choose to share certain 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 an agent creating a subagent for a certain task. Frequently, the creating agent does not want to donate some resources to the subagent, but it prefers to share its own resources with the subagent.

Resource accounting also affects the kernel communication facilities. A domain must be able to limit the size of incoming messages. Otherwise, malicious domains holding more memory resources could easily mount denial of service attacks by sending large messages. For channels, this means that the receiver can specify the maximum size for incoming messages. Because external references support two-way message exchange (argument and result messages), the callee as well as the caller may limit incoming messages.

7.2. Implementation issues

The J-SEAL2 kernel guarantees accountability, because references to an object exist only within a single domain. Therefore, it is possible to account each allocated object to exactly one protection domain. This feature not only simplifies memory accounting, but it is also crucial for immediate resource reclamation during domain termination.

For portability reasons, we employ byte-code rewriting techniques for memory and CPU accounting. In this approach the byte-code of a Java class is modified before it is loaded by the JVM. Readers interested in an extensive technical report presenting our byte-code transformations may contact the author by E-mail.

Like in JRes [5], code for memory control is inserted before each memory allocation instruction. The J-SEAL2 kernel maintains weak references to allocated objects in order to detect when an object is reclaimed. Enforcing memory limits requires exact pre-accounting for memory resources, i.e., an overuse exception is raised before a strand can exceed the memory limit of the domain it is executing in. In contrast to JRes, which controls 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. Therefore, memory accounting requires synchronization.

CPU accounting is based on an abstract measure, the number of executed byte-code instructions. Instructions for CPU accounting are inserted in every basic block of code. Every strand in the system updates its own CPU account. A high-priority scheduler strand executes periodically in order to ensure that assigned CPU limits are respected. It is responsible for accumulating the accounting data of all strands executing in a set of domains sharing a CPU limit. The scheduler strand compares the number of executed byte-codes with the desired schedule and adapts the JVM thread priorities of individual strands in order to control the CPU consumption of different protection domains.

8. Related work

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 [1, 2].

Like J-SEAL2, the Alta [2] operating system is a micro-kernel design. It provides a hierarchical process model supporting CPU accounting through CPU Inheritance Scheduling [6], 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.

J-Kernel [10] 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 byte-code rewriting [5]. Like J-SEAL2, J-Kernel is implemented completely in Java, only CPU accounting requires native code.

9. Conclusion

We have presented design and implementation issues that must be addressed by Java based mobile agent platforms. 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, it is portable over different operating systems and hardware platforms.

The J-SEAL2 micro-kernel offers strong protection domains, an efficient communication mechanism, and safe protection domain termination with immediate resource reclamation. Currently, we are integrating resource control facilities into the J-SEAL2 kernel. Readers interested in getting an evaluation version of the J-SEAL2 platform may contact the author by E-mail.

References

[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, Mar. 1999.

[2] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, and J. Lepreau. Java operating systems: Design and implementation. Technical Report UUCS-98-015, University of Utah, Department of Computer Science, Aug. 6, 1998.

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

[4] 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.

[5] 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, Oct. 18–22 1998. ACM Press.

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

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

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

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

[10] 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.