
Electronic Commerce Research, 1: 131–148 (2001)

J-SEAL2—A Secure High-Performance Mobile Agent System

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

Abstract

Even though the advantages of mobile agents for distributed electronic commerce applications 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, popular mobile agent systems do not meet the high security requirements of electronic commerce 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 model of Java, although it is not suited to protect agents and service components from each other.

In contrast, J-SEAL2 is a mobile agent system designed to meet the high security, portability, and performance requirements of large-scale electronic commerce applications. J-SEAL2 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. This paper focuses on the design of the new communication model in J-SEAL2, which allows convenient, efficient, and mediated communication in a hierarchy of strong protection domains.

Keywords: mobile agent systems, security, protection domains, communication models, Java

1. Introduction

Recently, many research projects have highlighted the benefits of mobile agent technology for large-scale Internet applications, such as distributed electronic commerce frameworks. Mobile agents support resource aware computations; agent 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. 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 computing, 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 agents, as well as each agent from any other agent in the system. In order to support large-scale distributed electronic commerce applications, mobile agent systems have to be portable and to offer good scalability and performance.

However, current mobile agent platforms fail to provide sufficiently strong security models, are limited to a particular hardware architecture and operating system, or cause high overhead. As a result, electronic commerce frameworks 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 reconcile strong security mechanisms with portability and high performance. J-SEAL2 is a micro-kernel design implemented in pure Java[1] (Gosling, Joy, and Steele, 1996); it runs on every Java 2 implementation (JDK 1.2 or higher) (Sun Microsystems). 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 Virtual Machine (JVM) (Lindholm and Yellin, 1999).

J-SEAL2 is a complete redesign of JavaSeal, a secure mobile agent system developed at the University of Geneva[2] (Vitek and Bryce, 1999; Vitek, Bryce, and Binder, 1998). 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 (Vitek and Castagna, 1998). 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 electronic commerce applications.

J-SEAL2 is compatible with JavaSeal, but offers enhanced security mechanisms and significantly improved performance. J-SEAL2 provides 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 (Bray, Paoli, and Sperberg-McQueen, 1998).

This paper is structured as follows: In Section 2 we discuss mobile agent systems implemented in Java. We show that the security model of Java is not sufficient to protect agents and service components. However, the Java language offers many features allowing to implement a strong security model on top of Java. In Section 3 we give a short overview of the J-SEAL2 mobile agent system and present its model of hierarchical protection domains. In Section 4 we discuss security features of the J-SEAL2 kernel. Section 5 is the core of this paper: We focus on the design of the new efficient communication model in J-SEAL2, which is mainly responsible for the high performance and scalability of the system. Fur-

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

[2] The University of Geneva is not engaged in J-SEAL2.

thermore, we describe an easy to use high-level communication mechanism built on top of the J-SEAL2 micro-kernel. In Section 6 we provide some performance measurements comparing the different communication mechanisms supported by the J-SEAL2 platform. In Section 7 we compare the J-SEAL2 kernel with related work. In Section 8 we present our plans for future improvements of the J-SEAL2 kernel. The last section summarizes the current state of implementation of our mobile agent kernel.

2. Mobile agent systems in Java

Recently, a large number of mobile agent systems based on Java (Gosling, Joy, and Steele, 1996) has emerged.[3] 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 heterogenous environment. The JVM specification (Lindholm and Yellin, 1999) defines a *portable code* representation for Java programs (byte-code), which is independent of any particular hardware architecture. Therefore, using JVM byte-code 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.
- 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 (Yellin, 1995) 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: *byte-code 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 dynamic loading and linking of code. *Classloader namespaces* can be used to isolate the classes of the agent system and of different agents from each other.

[3] 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.

For instance, with the aid of classloader 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 (Sun Microsystems) 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 (Back and Hsieh, 1999). 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 native code or modifications to the underlying Java implementation.

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 cooperation 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. A detailed discussion of these issues can be found in (Binder, 2000).

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 (Vitek and Bryce, 1999; Vitek, Bryce, and Binder, 1998; Vitek and Castagna, 1998). Low-level implementation techniques are discussed in (Binder, 2000).

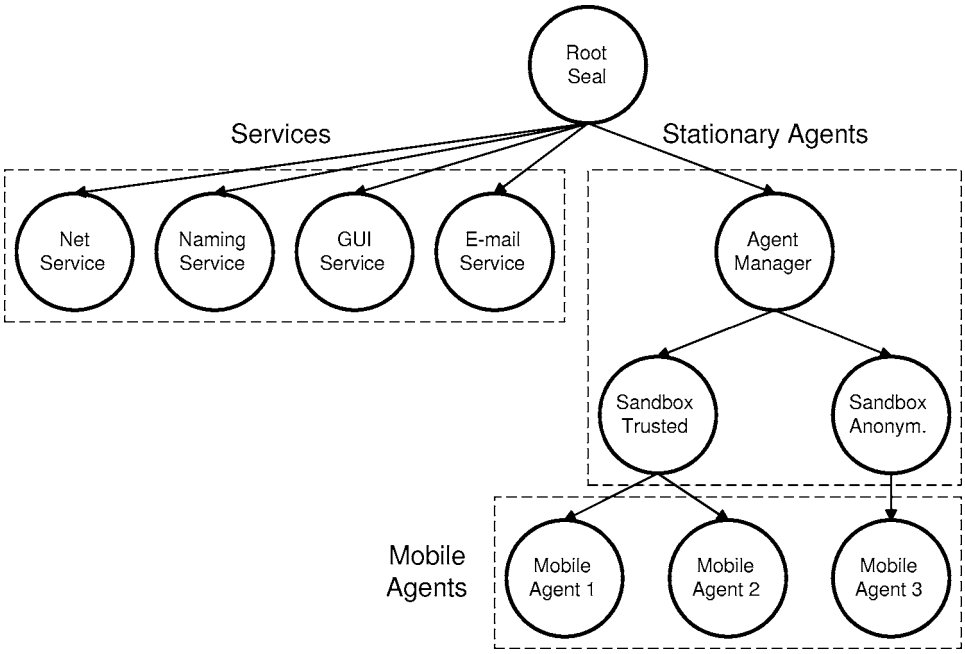

Figure 1. J-SEAL2 nested protection domains.

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 the RootSeal, which is responsible for starting system services.

The RootSeal reads a XML (Bray, Paoli, and Sperberg-McQueen, 1998) 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 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.
- 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 offline 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 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.

4. Security

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 5), 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.

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 *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.

Summing up, the J-SEAL2 kernel ensures the following important security properties (Vitek and Bryce, 1999):

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.

5. Communication model

5.1. Channel communication

In the JavaSeal kernel (Vitek and Bryce, 1999; Vitek, Bryce, and Binder, 1998) synchronous message passing through named *channels* is the only inter-agent communication mechanism. This model only supports direct communication between seals that are neighbors in the seal hierarchy. 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.

5.2. Limitations of channel communication

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 strands cannot cross seal boundaries, communication requires dedicated strands in intermediate seals for message routing. Therefore, communication between seals involves strand switches proportional to the communication partners' distance in the seal hierarchy. Furthermore, every communication 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 strand 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 strands stopping those strands 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.

5.3. External references

Inside a single JVM method invocation on objects enables extremely fast communication between different software components without involving any strand 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 strand 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. External references allow indirect sharing[4] of objects with different seals. 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.

5.3.1. 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 (Sun Microsystems). Weak references do not prevent objects from being reclaimed by the garbage collector.

[4] The properties of different sharing models (copying, direct sharing, and indirect sharing) are explained in (Back et al., 1998).

5.3.2. Specification. 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 form 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.[5] 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 an external reference is passed to a neighbor 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 strands 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, too, is encapsulated. 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[6] 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), strands 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 strand executes a method of a shared object in a foreign seal, the reference to the strand object is not available. This property is an important detail, since it prevents the shared object from storing the strand reference in its state (i.e., direct sharing with different seals) and from manipulating (e.g., stopping) that strand.

[5] 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 (Yellin, 1995) 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.

[6] Note that allowing external references to be passed over exisiting 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.

M-5 While a strand executes a method of a shared object, the seal the strand belongs to may stop the strand asynchronously. Therefore, the shared object must be designed in a way that does not allow a calling strand 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 (Vitek and Bryce, 1999; Vitek, Bryce, and Binder, 1998; Vitek and Castagna, 1998) in order to dispatch a request to a strand 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 5.4).

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 neighbor, a seal may atomically invalidate that copy[7] 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 neighbor seal.

I-5 When a seal is removed from the hierarchy, invalidation operation I-2 is triggered implicitly[8] in order to ensure complete and immediate 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 strand 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 strand throws an exception. This property also holds, if the strand terminates the seal it belongs to (see invalidation rule I-5).

5.3.3. Examples. The following examples illustrate how external references support direct communication between seals that are not in a parent–child relationship (see Figure 2).

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.

[7] 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.

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

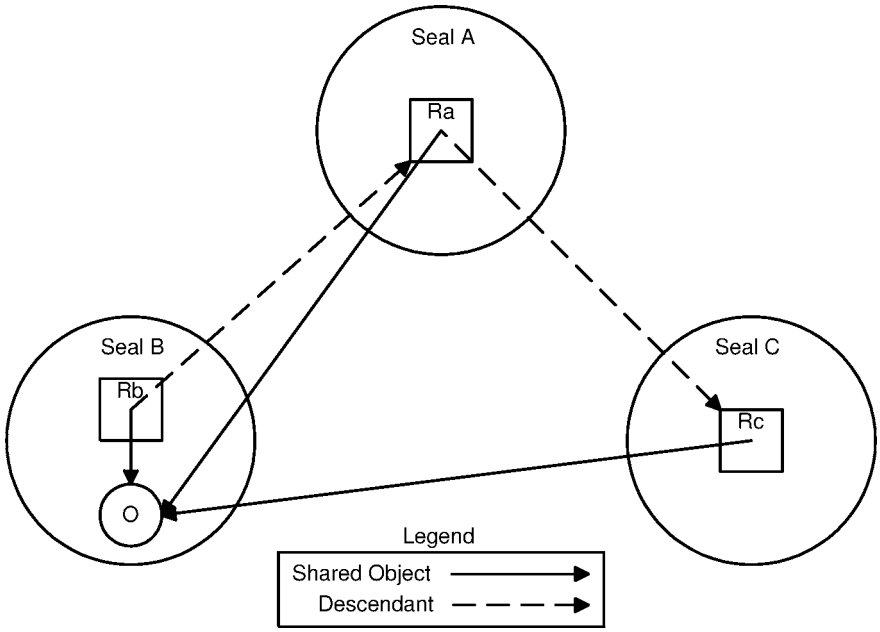

Figure 2. J-SEAL2 external references.

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*. In the same way, if the invoked method of shared object *O* returns a capsule containing *Rb*, seal *C* receives *Rc*.

5.4. 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 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 strands 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 strand is stopped asynchronously. A calling strand 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 strand. Thus, the shared object is only accessed by strands 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 (Vitek and Bryce, 1999; Vitek, Bryce, and Binder, 1998; Vitek and Castagna, 1998).

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,[9] 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.

6. Performance measurements

In this section we compare the performance of the various communication mechanisms offered by the J-SEAL2 platform. For this purpose, we measure the time it takes to transmit a capsule containing a Java long integer over n seal boundaries ($1 \leq n \leq 12$).[10]

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

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

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

Table 1. J-SEAL2 communication performance. All values are given in microseconds.

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

Pentium II, 400 MHz 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 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 strand switches, the IAMC communication costs[11] 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 (see, e.g., Figure 1), an agent has to communicate at least over four seal boundaries in order to access a service (agent–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 1, whereas external reference and IAMC communication do not incur any additional overhead.

7. 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 (Back and Hsieh, 1999; Back et al., 1998; Tullmann and Lepreau, 1998).

Like J-SEAL2, the Alta (Back et al., 1998; Tullmann and Lepreau, 1998) operating system is a micro-kernel design. It provides a hierarchical process model supporting CPU accounting through CPU Inheritance Scheduling (Ford and Susarla, 1996), where a process may donate some percentage of its CPU resources to nested child processes. However,

[11] 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.

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 electronic commerce 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.

J-Kernel (Hawblitzel et al., 1998; Hawblitzel and von Eicken, 1998) 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 (Czajkowski and von Eicken, 1998). Like J-SEAL2, J-Kernel is implemented completely in Java, only CPU accounting requires native code.

Luna (Hawblitzel and von Eicken, 1999a, 1999b) 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, 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.

8. Future work

In order to prevent denial-of-service attacks, such as agents allocating all available memory, the next J-SEAL2 release has to support resource control facilities for physical resources (i.e., memory, CPU time, network usage, etc.) and for logical resources (i.e., number of strands, number of nested protection domains, etc.). The resource control model has to reflect the hierarchical structure of the J-SEAL2 system.

Hierarchical process models have been used successfully by operating system kernels, such as the Fluke micro-kernel (Ford et al., 1996). The Fluke kernel employs a hierarchical scheduling protocol, CPU Inheritance Scheduling (Ford and Susarla, 1996), 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 generalization of CPU Inheritance Scheduling fits very well to the J-SEAL2 hierarchical domain model. At system startup the root domain, RootSeal, owns all physical and logical resources, for example, 100% CPU time, some amount of virtual memory, unlimited network usage, 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.

As there is no standard Java extension for resource accounting and because portability is crucial for the success of J-SEAL2 in large-scale electronic commerce projects, we are implementing accounting for physical resources with the aid of byte-code rewriting. For memory accounting, we are inserting resource checks before every memory allocation instruction. For CPU time accounting, we are placing accounting instructions in every basic block (e.g., in every loop) in order to gather execution statistics that are used by a high-priority scheduler thread to reassign thread priorities. While byte-code rewriting already has been used successfully for memory accounting (Czajkowski and von Eicken, 1998), currently there are no experiences with CPU time accounting via byte-code rewriting. We will have to experiment with different algorithms in order to find the best trade-off between accounting accuracy and little overhead.

Furthermore, we are also considering an additional implementation of the J-SEAL2 architecture on top of implementations of the Real Time Specification for Java (The Real Time for Java Experts Group), which contains some concepts that can be used for resource control. For an in-depth discussion of techniques for resource control in Java environments see (Binder, 2000).

Other important features to be offered in a future J-SEAL2 release include remote configuration and administration without disruption of the agent platform. Furthermore, we are integrating 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.

9. Conclusion

We have shown that security, portability, and high performance are crucial for the success of a mobile agent system in large-scale distributed electronic commerce 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. For portability reasons, J-SEAL2 is implemented in pure Java. The J-SEAL2 micro-kernel offers strong protection domains and safe domain termination with immediate resource reclamation. It 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. Currently, we are integrating resource control facilities into the J-SEAL2 kernel.

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.

References

Back, G. and W. Hsieh. (1999). "Drawing the Red Line in Java." In *Proceedings of the 7th IEEE Workshop on Hot Topics in Operating Systems*. Rio Rico, AZ. Available via WWW at URL: `http://www.cs.utah.edu/flux/papers/redline-hotos7.ps.gz`.

Back, G., P. Tullmann, L. Stoller, W.C. Hsieh, and J. Lepreau. (1998). "Java Operating Systems: Design and Implementation." Technical Report UUCS-98-015, University of Utah, Department of Computer Science. Available via WWW at URL: `http://www.cs.utah.edu/flux/papers/javaos-tr98015.ps.gz`.

Binder, W. (2000). "Design and Implementation of the J-SEAL2 Mobile Agent Kernel." In *Sixth ECOOP Workshop on Mobile Object Systems: Operating System Support, Security and Programming Languages*. Cannes, France. Available via WWW at URL: `http://cui.unige.ch/~ecoopws/ws00/papers/js.ps`.

Bray, T., J. Paoli, and C.M. Sperberg-McQueen. (1998). Extensible Markup Language (XML) 1.0. W3C Recommendation. Available via WWW at URL: `http://www.w3.org/TR/1998/REC-xml-19980210`.

Czajkowski, G. and T. von Eicken. (1998). "JRes: A Resource Accounting Interface for Java." In *Proceedings of the 1998 ACM OOPSLA Conference*. Vancouver, BC. Available via WWW at URL: `http://www.cs.cornell.edu/slk/papers/oopsla98.pdf`.

Ford, B., M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. (1996). "Microkernels Meet Recursive Virtual Machines." In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI'96)*. Available via WWW at URL: `http://www.cs.utah.edu/flux/papers/fluke-rvm.ps.gz`.

Ford, B. and S. Susarla. (1996). "CPU Inheritance Scheduling." In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*. Available via WWW at URL: `http://www.cs.utah.edu/flux/papers/inherit-sched.ps.gz`.

Gosling, J., B. Joy, and G.L. Steele. (1996). *The Java Language Specification*. The Java Series. Reading, MA: Addison-Wesley. Available via WWW at URL: `http://java.sun.com/docs/books/jls/html/index.html`.

Hawblitzel, C., C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. (1998). "Implementing Multiple Protection Domains in Java." In *Proceedings of the 1998 USENIX Annual Technical Conference*. New Orleans, LA. Available via WWW at URL: `http://www.cs.cornell.edu/slk/papers/usenix98-final.pdf`.

Hawblitzel, C. and T. von Eicken. (1998). "A Case for Language-Based Protection." Technical Report TR98-1670, Department of Computer Science, Cornell University. Available via WWW at URL: `http://www.cs.cornell.edu/slk/papers/TR98-1670.pdf`.

Hawblitzel, C. and T. von Eicken. (1999a). "Type System Support for Dynamic Revocation." In *ACM SIGPLAN Workshop on Compiler Support for System Software*. Available via WWW at URL: `http://www.cs.cornell.edu/Info/People/hawblitz/hawblitz.html`.

Hawblitzel, C. and T. von Eicken. (1999b). "Tasks and Revocation for Java (or, Hey! You got your Operating System in my Language!)." Draft. Available via WWW at URL: `http://www.cs.cornell.edu/Info/People/hawblitz/hawblitz.html`.

Lindholm, T. and F. Yellin. (1999). *The Java Virtual Machine Specification*, 2nd ed. Reading, MA: Addison-Wesley. Available via WWW at URL: `http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html`.

Sun Microsystems. JAVA 2 SDK, Standard Edition. Available via WWW at URL: `http://java.sun.com/products/jdk/1.2/`.

The Real Time for Java Experts Group. Real Time Specification for Java. Available via WWW at URL: `http://www.rtj.org`.

Tullmann, P. and J. Lepreau. (1998). "Nested Java Processes: OS Structure for Mobile Code." In *Proceedings of the 8th ACM SIGOPS European Workshop*. Available via WWW at URL: `http://www.cs.utah.edu/flux/papers/npmjava-esigops98web.ps.gz`.

Vitek, J. and C. Bryce. (1999). "The JavaSeal Mobile Agent Kernel." In *Proceedings of the Joint Symposium ASA/MA'99, 1st Internat. Symposium on Agent Systems and Applications (ASA'99) and 3rd Internat. Symposium on Mobile Agents (MA'99)*. Palm Springs, CA, USA. Available via WWW at URL: `http://cui.unige.ch/OSG/publications/OO-articles/TechnicalReports/99/kernel.pdf`.

Vitek, J., C. Bryce, and W. Binder. (1998). "Designing JavaSeal or How to Make Java Safe for Agents." In D. Tsichritzis (ed.), *Electronic Commerce Objects*. University of Geneva, pp. 105–126. Available via WWW at URL: `http://cui.unige.ch/OSG/publications/OO-articles/TechnicalReports/98/javaSeal.pdf`.

Vitek, J. and G. Castagna. (1998). "Towards a Calculus of Secure Mobile Computations." In *Workshop on Internet Programming Languages*. Chicago, IL. Available via WWW at URL: `http://cui.unige.ch/OSG/publications/OO-articles/TechnicalReports/98/calcSecure.pdf`.

Yellin, F. (1995). "Low Level Security in Java." In *Fourth International Conference on the World-Wide Web*. MIT Press.

Walter Binder was born in Vienna on April 14th 1974. From 1992 to 1997 he studied computer science at the Technical University of Vienna. From 1997 to 1998 W. Binder worked with the Object Systems Group at the Centre Universitaire d'Informatique of the University of Geneva. There he developed the JavaSeal mobile agent kernel, which was later used in some electronic commerce applications. Since December 1998 W. Binder is working for CoCo Software Engineering GmbH, where he is developing the J-SEAL2 mobile agent platform. At the same time he is a Ph.D. student at the Technical University of Vienna. His research interests are in the area of object systems, operating systems, distributed and parallel computing, security, compilation techniques, and logic programming.