Autonomous Agents and Multi-Agent Systems, 4, 359–384, 2001
© 2001 Kluwer Academic Publishers. Manufactured in The Netherlands.

The JavaSeal Mobile Agent Kernel

CIARÁN BRYCE ciaran.bryce@cui.unige.ch
Object Systems Group, University of Geneva, Switzerland

JAN VITEK jv@cs.purdue.edu
Department of Computer Sciences, Purdue University, West Lafayette, IN

Abstract. Mobile agents show promise as a new distributed programming paradigm in which locality plays a central role—programs that are able to move closer to their data can overcome limitations of connectivity, latency or bandwidth. Mobility also enables distributed systems to evolve; for instance, the deployment of a new service over a network can be programmed as part of the service itself. Of course, moving programs introduces new challenges. One of these is related to program structure: How much of a computation should be moved? Where are the boundaries between mobile and immobile entities drawn? A second challenge is to provide security guarantees: How can the actions of mobile agent be controlled? And what kinds of security properties can we realistically expect to enforce? We answer these questions within the framework of the JavaSeal mobile agent system kernel. JavaSeal provides several abstractions for constructing agent systems in Java. Our basic building block is the *seal* which is a nested encapsulated computation fragment with sharply delineated boundaries. *Strands* are sequential threads of computation bound to a seal. *Capsules* transfer passive seals and objects over communication *channels*; Traffic over channels is regulated by *portals*. We argue that these abstractions are sufficient to program secure mobile agent systems. An electronic commerce application built over our kernel is used as a demonstrator.

Keywords:

1. Introduction

Mobile agents are units of computation that control *where* they execute. This form of mobility is supported by a software infrastructure called a mobile agent system. Mobile agents embody a new approach to distributed programming in which locality is an essential part of the computational model [10, 12]. Controlling locality is particularly important on wide area networks where it may be the key to overcoming variations in communication latency and bandwidth; for instance, a sudden increase in latency may be offset by moving the agent closer to its data [5, 36]. Agents provide additional freedom over traditional distributed programming techniques in that it is not necessary to decide before-hand where each component of a distributed system will run, nor to pre-install these components on all nodes that may potentially run them. Instead, mobile agents can deploy themselves and even reconfigure dynamically in response to changes in the environment such as a server going down, or new resources being added to the system.

Mobile agents are a natural evolution of active objects [3] and mobile objects [26] which were studied in the 1980s. Mobile agents are self-contained computations with their own thread of control which can be moved as units across the network. In the mobility model, bindings to local components and resources are broken off when an agent migrates, and this includes connections to other agents. The internal state of the agent is nevertheless preserved. Two mobility models have been investigated for agents: in the case of strong mobility the entire state of the agents is moved including all running threads [44], while in the case of weak mobility, only a portion of the data state is transferred [6, 22, 29, 37, 38].

This paper reports on the design and implementation of the JavaSeal mobile agent kernel. JavaSeal is not full-featured agent system, but rather a kernel upon which different agent systems can be implemented. Our design goals were to investigate three basic issues that all agent systems must address:

- *Structuring principles* guide the design of agents in that they provide hints about the functionality or data that should be in a particular agent. The agent system must then enforce this structure since without a sharp boundary around agents, mobility runs the risk of introducing errors (not moving the right objects) and inefficiencies (moving too many objects or too much code).
- *Mobility support* is more than a network protocol for exchanging data. It involves stopping a computation in a coherent state, storing that state in an intermediate format suitable for transport, and restarting the computation later from this intermediate representation.
- *Security guarantees* ensure that certain properties will hold throughout execution of agent programs.

JavaSeal provides abstractions to address the above issues. In JavaSeal, mobile agent programs are structured as a hierarchy of *seals*. Seals are encapsulated computations composed of one or more active *strands* of execution as well as zero or more nested subseals. Seals are kept disjoint from one another by a system enforced boundary; their only means of communication is message passing over synchronous *channels*. Channels are controlled by capability-like *portals*. Mobility is supported through a set of kernel operations for stopping seals, writing them to archive files, and later retrieving them from these archives. Application specific services such as secure network protocols or even user interfaces are implemented as user level services on top of JavaSeal.

The simple agent model chosen for JavaSeal has the advantage that its formal semantics are tractable. The Seal calculus [41], a relative of Cardelli and Gordon's Ambients [11], allows reasoning about the behavior of mobile programs. The implementation of JavaSeal was influenced by the nested process model of the Fluke operating system [31] and Alta, its Java counterpart [4].

The main contribution of this work is an implementation of the nested seal computation model and the strengthening of Java's security model. We have found that while the Java language can be used to implement secure systems, the current security architecture is not inherently secure. Writing secure code in Java is error prone and it is too easy to inadvertently break security. In that sense, JavaSeal is an alternative to the standard Java sandbox security mode.

2. JavaSeal architecture

The JavaSeal kernel provides an interface for writing mobile agent systems. We have chosen to include a small set of core services in the kernel—for communication, mobility and protection—and program all higher level services as JavaSeal agents. The kernel design emphasizes security; one of the most significant decisions in that respect is the restriction on object sharing imposed by the kernel.

JavaSeal is implemented as a collection of trusted Java packages running on top of Sun's JDK1.2 virtual machine. It is notable that no changes were required in either the virtual machine or the core JDK classes. The kernel is thus easily portable to other JVMs. Indeed, porting JavaSeal from version 1.1 to 1.2 of the JDK took no more than a couple of days. We now describe the main abstractions provided by the kernel.

2.1. Seals

Seals are encapsulated computations composed of Java objects and threads. Encapsulation protects seals from each other and prevents them bypassing the kernel communication mechanisms. Each seal defines its own name space disjoint from other seals; each object class and thread in a JavaSeal program belongs to one and only one seal. An active seal is thus a structure $\langle O, \mathcal{C}, S, C, P\rangle$ where O is a set of Java objects, $\mathcal{C}$ is a set of classes, S is a set of active threads (strands), C are channels and P are portals. The last two sets are discussed in the next two subsections.

Seals nest to form a hierarchy, thus the set of seals running on the same kernel instance forms a tree, see Figure 1. The root of the seal tree implements the kernel functionality, and is referred to as the `RootSeal`. This hierarchy is central to the design of JavaSeal. Communication between seals is restricted to parent–child interaction which allows to implement security policies by interposition.

Mobility is implemented using the channel based communication mechanism. Moving a seal is done in three steps: the seal is first stopped, its externalizable state is written to a new capsule and finally the capsule is transmitted on a channel to a neighbor. Mobility is not restricted to complets (leaf seals); any seal can be moved. In the case of envlets (non-leaf seals), moving the seal not only implies moving objects but also moving all subseals. Figure 2 illustrates a move within the hierarchy in which a seal migrates to another node of tree. Moves are always initiated by the parent seal. Thus the traditional `moveTo` command provided in most agent systems is implemented as request to the parent seal. The motivation is security as the parent must be able to oversee all externally visible actions of its subseals.

The interface of the `Seal` class is shown in Table 1. This abstract class exposes an interface composed of the following five methods: `currentSeal` returns the name of the seal. Names are used to designate channel end points as well as seals. `parentSeal` returns the name of the parent. `wrap` stops the named subseal and returns a capsule containing a representation of that seal's persistent state. Wrapping does not preserve thread state. `unwrap` extracts a seal from its capsule

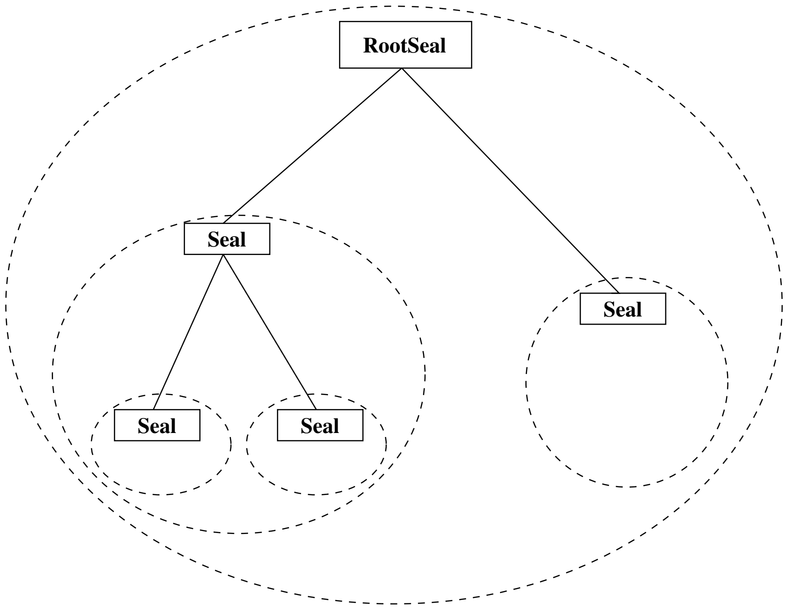

Figure 1. A Seal hierarchy. The `RootSeal` represents the `JavaSeal` kernel. Each seal consists of an instance of the `Seal` class, as well as a set of objects and subseals.

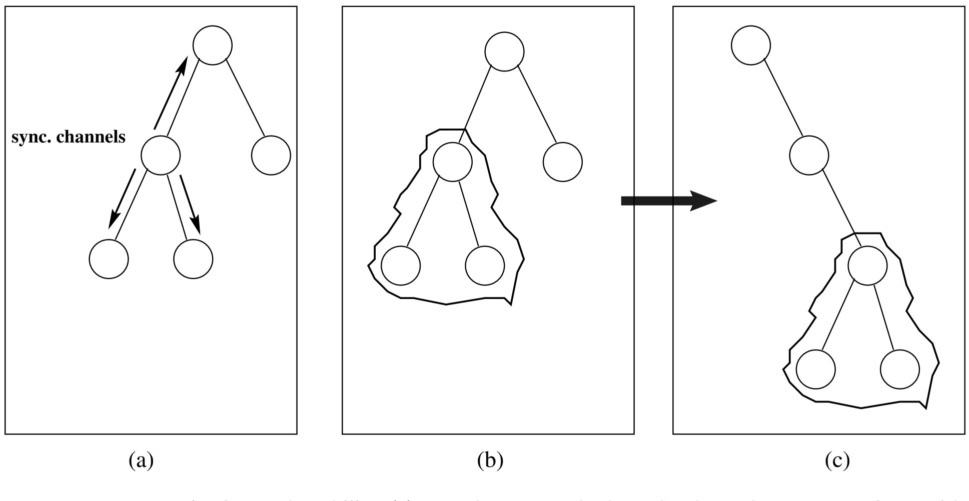

Figure 2. Communication and Mobility. (a) A seal can uses the kernel's channels to communicate with direct neighbors in the seal tree. (b and c) Mobility means that the structure of a seal application can be changed dynamically.

Table 1. The abstract Seal class

```
abstract class Seal implements Runnable, Serializable {
     final Name currentSeal();
     final Name parentSeal();
     final void rename(Name from, Name to);
     final Capsule wrap(Name subseal);
     final unwrap(Capsule cap, Name seal);
     abstract public void run();
}
```

argument, initializes it and starts it under the name given as the second argument. The wrapping operations are consistent with the hierarchical control. A seal may only be wrapped and unwrapped by its direct parent. New seals are thus always rooted in the currently executing seal. When a seal is unwrapped, the run method is invoked with a new execution thread. Unless otherwise specified, all methods in the classes and interfaces shown in the paper are public.

The class is declared abstract to signify that it must be extended in user code with, at least, an implementation for the run method. We now show an example complet that randomly visits a set of network nodes:

```
public class HelloWorld extends Seal {
     public void run() {
         if (Host.name() == "cui.unige.ch")
             Host.println("Home at last");
         else
             Host.moveTo(Host.neighbor());
     }
}
```

The `moveTo` and `neighbor` messages are requests to the parent of `HelloWorld` seal. We postpone the explanation of the `Host` class to Section 2.3.

2.2. Capsules

Capsules are data containers used to transfer objects by value between seals. The motivation for capsules comes from the requirement that objects not be shared between seals. Capsules come in two kinds: data capsules contain objects while seal capsules contain passive seals.

Data capsules. A data capsule is constructed by specifying a root object. The created capsule contains all serializable objects reachable from the root with the exception of `JavaSeal` kernel objects. Capsules are implemented by serializing the root and copying all contained objects. Data capsules do not include code, and to ensure that a capsule does not span seal boundaries, kernel objects are never copied into a capsule. Table 2 shows the interface of the capsule class.

Table 2. The Capsule class

```
    final class Capsule implements Serializeable{
        Capsule(Object obj);
        final Object open() throws ClassMismatchError;
        final void compact();
    }
```

A capsule's contents can only be released into the current seal by invoking `open`. The current seal is potentially a different seal from the one that created the capsule, so to protect seals from viruses, a capsule is not allowed to introduce new classes into a seal. The open call fails if any of the objects in the capsule belongs to a class (or version) that is not part of the set of classes of the current seal.

Seal capsules. Seal capsules contain seals with all of their data, code and subseals. The only part of an active seal that is not retained is thread state. A passive seal is stored in a custom archive format that contains serialized objects as well as a signed collection of bytecode files.

Creating a seal capsule requires serializing a seal tree. Serialization proceeds bottom up, each visited seal is first stopped, its objects are serialized and its classes are added to the archive. With the exception of kernel code, all code used by a seal is included in its archive. Our motivation for this choice are twofold: (a) some applications require off-line operation—this means that lazy class loading is not an option as the agent's origin may not be reachable; (b) versioning support in Java is weak, thus if two different classes were shipped with the same interface and version number, there is no way to determine the correct class version. Our approach guarantees that the right version is always available.

Opening a seal capsule triggers a verification procedure that checks the validity of the archive. First, all bytecode files are verified against their digital signatures, then the bytecode verifier is run on the bytecode, and finally the security checks described in Section 4.1 are applied. If the capsule passes all checks, the topmost seal is extracted and started. Subseals are left in their capsules; the instantiated seal can choose to awaken them if need be.

A drawback of seal capsules is that they are large and the same code may end up being transfered repeatedly. The overhead for a seal capsule is 1.2KB, but a typical agent in our demonstrator application is about 84KB. To alleviate this problem we developed a custom code compressor called Jazz [9] which is able to reduce Java bytecode files to 24% of their original size, that is one half the size of gzipped archive. Jazz is invoked by calling the `compact` method on a seal capsule.

2.3. Channels and portals

Channels provide a synchronous message passing communication model between seals with the primitives operations `send` and `receive`. send takes a channel name, a target seal name and a capsule, and attempts to transfer the capsule on the named channel in the target seal. A matching `receive` specifies the name of local

Table 3. The channel and portal classes.

```
final class Channel {
     static void send(Name chan, Name seal, Capsule caps);
     static Capsule receive(Name chan, Name seal);
}
final class Portal {
     static int status(Name channel, Name seal);
     static open(Name channel, Name seal, int capacity);
     static close(Name channel, Name seal);
}
```

channel and a seal on which to wait for a message. Communication requires a pair of matching offers and an open portal. The portal class has an open method which opens a channel for a specified number of communications. Table 3 presents the interfaces of the two classes.

Communication offers may specify a channel located either in a parent seal or a child seal. The following code fragment tries to send a string to the parent of the current seal:

```
Name req = new Name("Request");
Channel.send( req, parentSeal(), Capsule(new String("Hello")));
```

Since the channel is located in the parent, the parent must first open a portal and also be willing to receive on the channel:

```
Name req = new Name("Request");
Portal.open(req, child, 1);
Channel.receive( req, currentSeal(), val);
```

Note that the second argument to receive indicates the location of the channel and not from where the message comes. Thus, the parent specifies that it is listening to its own `req` channel. If the only portal that is open is for the `child` seal, then this is the only other seal that may write to `req`.

Separating the action of opening a portal and the sending or receiving on the associated channel allows to split the access control logic from the main application behavior. Figure 3 illustrates one particular interleaving of communication offers: the `open` method is non-blocking, thus it always returns, but both the receive and the send are blocking—they only return when the communication has completed.

We can now show the implementation of the Host class. Its role is to encapsulate channel communication within a parent. The implementation presupposes agreement on channel names, that is, the envlet in which the seal using the `Host` class is located must listen on the right channels and agree to provide the requested services.

```
class Host {
   private final Name _moveTo = Name("MoveChan");
   private final Name _system = Name("System");
   void println(String s) {
```

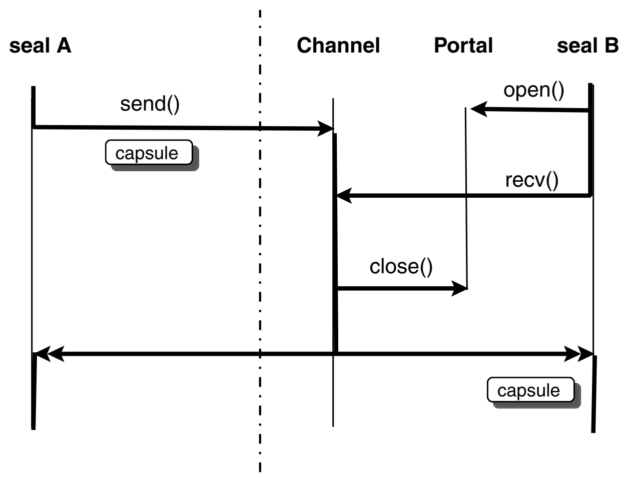

Figure 3. Channel communication.

```
            Channel.send(_system, Seal.parentSeal(),
                                      Capsule(Pair("println",s)));
         }
         void moveTo(String s) {
            Channel.send(_moveTo, Seal.parentSeal(), Capsule(s));
         }
         String name() {
            Channel.send(_system, Seal.parentSeal(), Capsule("name"));
            Capsule cap = Channel.receive(_system, Seal.parentSeal());
            String res = null;
            try {
                 res = (String) cap.open();
            } catch(Throwable t) { Error.report(t);}
            return res;
          }
      }
```

The two channel names used in this example are `MoveChan` for move requests and `System` for requests to the `System` class. In the case of system requests, the request is either a single string that describes the request or a pair composed of a service description and an argument.

2.4. *Strands*

In Java, any thread can be asynchronously interrupted by an invocation of the `stop` method. Asynchronous interrupts are particularly dangerous as a thread may be stopped and leave some key data structures in an inconsistent state. Strands abstract Java threads. They allow interrupts to threads within a seal, but prohibit attempts to stop threads running in other agents or the kernel. This is achieved by ensuring

Table 4. The Strand class

```
final class Strand {
     static Strand create(Runnable target);
     static Strand currentStrand();
     static Strand getStrand(Name subseal);
     void start();
     void stop();
}
```

that strands do not cross seal boundaries, and that a seal is unable to reference a strand in another seal. Strands sometimes execute in the JavaSeal kernel for short sequences of instructions but `stop` signals sent while in kernel mode are delayed until the strand exits the kernel.

Strands are wrappers around Java threads. The interface of the strand class is listed in Table 4. The cost of this approach is that communication between seals requires synchronization and is thus predictably slow. On a 333MHz `UltraSparc` the average cost of sending a message to another seal and getting a reply in return is 414 μ-seconds. In order to improve on synchronization overheads, we are considering implementing strands with a migrating thread model [23, 43].

2.5. *Services*

An agent application and its roaming agents require several kinds of service to function. In a JavaSeal system, there are three general classes of agent service: (i) kernel services that each seal generally requires, e.g., network and simple I/O, (ii) environment services that an envlet offers to its traveling complets, and (iii) application services that are specific to each application that runs over JavaSeal. We look at each category in turn.

Kernel services. Kernel services are services which, while not part of the kernel, are installed by default. The two main examples are network access and simple I/O. Both of these services are implemented within `RootSeal`. The reason for this structuring is security. First, ordinary seals must not be allowed access to the classes needed to implement these services; therefore, a seal that seeks a service must send a request to `RootSeal`. The second reason is to allow `RootSeal`'s security policy to decide whether a particular seal request should be serviced or not.

Environment services. Environment services are those offered to a mobile agent complet on its current host by its envlet. One of the main JavaSeal design goals is to permit a seal to control all externally observable actions of its children in the following two respects. First, a seal can stop and start its subseals. Second, a seal always intercepts all messages sent by subseals and can apply any appropriate security restrictions on these messages. On the other hand, a seal is not able to peek and poke the internals of its children seals. Communication with children can *only* take place by exchanging messages. This last property is instrumental in protecting seals from their environment.

Envlets interpose between requests of a complet and its environment. They can play the role of adapters when the services on the current platform do not match an agent's expectation [25], as well as to interpose extra security checks. An envlet's role is also to act as a proxy for a remote service. For example a very simple envlet is given next that has a single child and interposes on messages sent on the `System` channel. If the subseal requests the name of the current host, the envlet will return a fake name, all other requests are transmitted up the hierarchy (where, supposedly, some seal will actually satisfy them).

```
class Filter extends Seal {
   private final Name _system = Name("System");
   private final Name _child = Name("myChild");
   void run () {
      ...  code to start up the child
      while (true) {
        Portal.open(_system,_child, 1);
        Capsule cap = Channel.receive(_system, currentSeal());
        try {
           String val = (String) cap.open();
           if (val.equals("name")) {
              Portal.open(_system,_child, 1);
              Channel.send(_system, currentSeal(),
                                          new Capsule("localhost"));
              continue;
        } catch (Throwable t) { . . . }
        Channel.send(_system, parentSeal(), cap);
        Capsule res = Channel.receive(_system, parentSeal());
        Portal.open(_system,_child, 1);
        Channel.send(_system, currentSeal(), res);
      }
   }
}
```

Application services. Application services are dedicated to a particular enduser application. In the newspaper application of Section 5, services include a smartcard process that interfaces with a smartcard reader device, a GUI control panel, a HTTP process that interfaces with a netscape browser, and a file server. It is important that an application be allowed to install as wide a range of services as possible without compromising security. Application services do not run within JavaSeal, but in parallel to `RootSeal` over the same JVM in order to protect JavaSeal from programming errors in these services.

3. Mobile agent security

As Chess argues in [13, 40], mobile agents violate many standard assumptions that underlie existing security architectures. The main reason for having code mobility in the first place is to create open systems that can be extended in ways not foreseen when first deployed. Agent systems are built above wide area networks without restriction on the machines that host or create agents. Each of these points is a separate research topic in security: security of extendible systems have been studied by researchers in operating systems, e.g., [7], authentication and trust for open networks have been investigated at length [1], and the issue of controlling the flow of information between computations has been an open problem for more than twenty years [15, 42]. It is therefore not surprising that mobile agent security has proven to be a difficult problem.

We will not address the issue of malicious hosts or environments designed to subvert the programs running on them here. Our position is that they are a risk in any open environment. While solving the general problem may be impossible, solutions have been proposed in the literature for special cases, these include tracing [39, 40], computing with encrypted functions [35] and tamper-proof hardware. Tracing involves having each site visited by an agent send back signed intermediate results that the owning site can verify; the aim of the approach is to detect malicious sites. Encrypted functions operate on encrypted data and produce a result in encrypted form. Unfortunately, these *privacy homomorphisms* only exist for a restricted class of functions. Tamper-proof hardware aims to prevent malicious hosts altogether [45].

The goal of this section is to present the state of the art in Java security. Java contains several weaknesses that had to be overcome in the JavaSeal design. We overview the security mechanisms in the Java languages, and then in the runtime environment. We close the section with an overview of security in other mobile agent systems.

3.1. Agent security basics

Computer security is defined relative to assets to be protected and to a threat model. In the case of mobile agents three kinds of assets may require protection: the data and resources held by the hosts on which agents execute, the agents themselves as they contain sensitive information in their code and data, and lastly, the network that interconnects hosts.

Protecting against breaches of *secrecy* implies preventing sensitive information such as passwords or credit card numbers from being leaked to unauthorized entities. Protection against *integrity* violations requires that agents be prevented from corrupting assets, such as for example erasing the host's hard disk. Finally, guaranteeing *availability* means preventing an agent from consuming inordinate amount of resources that could prevent other agents from executing.

In a mobile agent setting, these attacks can originate from *malicious agents*—agents engineered to subvert the security of the system on which they run, or from *malicious hosts*—agent kernels that mount attacks against visiting agents.

A secure agent kernel is the building block for more powerful policies. Even though this problem has been studied, the issues involved in building a secure kernel are still ill understood. For instance, very few agent systems provide a clear definition of what guarantees their security architecture provides. The task is sometimes pushed onto the operating system, though it is recognized that operating system protection is the wrong level of granularity [19]. Most agent systems take a language-based approach, and rely on the protection mechanisms provided by a safe programming language to enforce security. As we will argue with Java, the fit between the security requirements of agent applications and what languages provide is not perfect.

The traditional components of a secure system are *principals, security policies, protection domains* and *reference monitors* [20]. Principals are entities whose access to information and resources is controlled by security policies; these can be users, servers, IP domains, etc. Security policies grant principals the rights to access resources and use services provided by the host. They also control communication between principals. The granularity of control is an important issue: too fine a granularity implies complex and costly policies. Typically, access control is required only between protection domains which are contexts associated with a single principal 'owning' some objects and resources. Operations that execute entirely within a protection domain need not be checked.

Software systems contain a variety of communication channels over which principals exchange information [28]. *Legitimate channels* are included in a system precisely to let principals exchange information, e.g., sockets. *Storage channels* are elements of an environment that can be read or written by several programs and which can therefore be used to exchange information between these programs. The last category is called *covert channels* which are a means of communication that exploit a visible system characteristic in an unconventional manner. For instance, a program can signal the value of a PIN to its environment by creating a file with exactly that number of bytes. When secrecy must be protected, security architectures strive to prevent storage channels and limit the bandwidth of convert channels.

A reference monitor is a key component in any security architecture. Its role is to enforce the security policy for all access to resources and uses of legitimate channels. A reference monitor must satisfy the following two properties:

- **R1** *Total mediation:* The reference monitor intercepts all operations so that they can be verified by the security policy.
- **R2** *Encapsulation:* The reference monitor is protected from tampering by user programs.

3.2. Java security architecture

Java was designed to protect a host against programs downloaded from the network [21]. The basic assumptions here differ from those underlying mobile agents. For instance, applets are not expected to communicate, so Java can afford to isolate applets rather than provide controlled communication mechanisms. Furthermore, in a web browser there is usually little state that cannot be recovered so it may be an acceptable response to shut down the virtual machine when an application consumes all available memory. In an agent system this is not feasible as it would simply destroy legitimate programs.

Language-based security. Basic security in Java is enforced by safe programming language features such as strong type safety and bytecode verification. At this level, the only guarantee provided is that a Java program is not able to access arbitrary memory locations or to treat a data segment as executable code. The language has built-in access control policies, namely access modifiers, that restrict access to individual fields or entire classes that have been declared private or package-scoped. In addition, methods can be declared final to prevent being redefined in user-extensions.

These language based mechanisms are static. They cannot be changed once a class has been linked into the virtual machine. Moreover, the policies that can be specified are fairly coarse grained. There are four access modes: (1) unrestricted, (2) same package access, (3) same class and subclasses, and (4) same class. There is, for instance, no way to specify a set of packages as having access to some field, nor to prevent an unauthorized class from calling a public method. A class can be restricted to a single package, as in the following code fragment

```
package example;
class SecretAccount extends Account ...
```

Unfortunately, nothing prevents another part of the program from handing out a secret account if the object is widened to a public supertype

```
outsider.leak((Account)new SecretAcount())
```

When this expression is evaluated, an outside object will acquire a reference to an `Account`. The issue here is that subtyping allows to get around the security provided by static access modifiers. A similar problem has been the source of a major security breach in a previous implementation of the Java security architecture [8].

Subtyping can also be used to 'inject' code into another class. That is, when a class expects an object of some type as argument to a method, a client may call that method with any subtype of the expected class. The security risk is that a malicious client may hand out an object specifically designed to break the victim. The only way to rule out such attacks is to rule out subtyping.

The bottom line is that security mechanisms in Java were originally conceived as software engineering mechanisms. They are mainly designed to protect developers working independently on a project from each other. For security, these mechanisms fail to provide assurances and are deceptively hard to use in a disciplined manner.

3.3. The JDK security model

The JDK 1.2 security architecture is used to protect a JVM and the host system from malicious or erroneous applets. Each applet is assigned a trust level, based on the applet's source and signatures, and this level is used to determine the applet's privileges. Basically, classes act as protection domains, class loaders are used to control class name spaces, and security managers implement some of the features of a reference monitor.

Classes are protection domains in the sense that security policies grant permissions to classes and checks are performed by verifying the classes of callers of a method. The drawback of this design is that protection domains are fine grained, potentially every method call may require an access control check. Not only is this unmanageable from a security policy standpoint, but it is also unreasonable on efficiency grounds.

Security policies are consequently enforced by a combination of restriction of the class name space and programmer inserted dynamic access control checks. Name space restriction hides certain classes from untrusted programs. Dynamic access checks are initiated by inserting calls to a security manager within methods which perform sensitive operations. The main drawback of this approach is that class loaders only enforce an initial separation into domains. If applications are allowed to communicate, they can exchange references to previously hidden objects. Furthermore, the explicit access control checks are error prone: forgetting a single check may jeopardize the entire security architecture.

Java lacks any meaningful notion of resource accounting. Thus the security architecture cannot prevent denial of service attacks. It is hard to implement resource accounting at the granularity of a single class, especially with the pervasive sharing that permeates the environment. For instance, threads potentially cross class boundaries on every method call and objects are frequently allocated in one class and used in another.

It is clear from these mechanisms that the Java security architecture, designed specifically to protect hosts from applets, is not appropriate for mobile agent systems where agents must communicate with other agents in a controlled manner.

Secure language extensions and Java agent systems. Despite the orientation of the Java security architecture, several agent systems have been developed over Java.

The J-Kernel is a Java package from Cornell that provides disjoint protection domains that prevent object references being shared between domains [24]. Communication between domains is done using capability objects. A capability object is akin to a remote reference except that when a method is invoked using a capability, all arguments are transfered by deep-copy. Service classes are shared between domains, thus lending themselves to covert channels and security breaches. Protection domains form a flat space across which capabilities can be freely distributed. Once a capability has been given out, its originator retains little control on how the capability is used. A capability, unlike a plain object reference, can be revoked but revocation indiscriminately applies to all domains holding a copy of the same capability.

Mobile agents systems such as Aglets [30], Mole [6] and Ajanta [27] rely on the underlying Java security model for protection against malicious agents. Although an authentication and digital signature model has been proposed for Aglets, none of these systems provides a clear statement of what security guarantees are enforced.

The JRes system from Cornell performs flexible resource accounting of heap memory, CPU time, and network resources consumed by individual threads or groups of threads [14]. If threads are not allowed to cross protection domain boundaries, then this mechanism could be useful to control the resources used by agents.

The JFlow language from MIT [33] introduces a strong security model to Java. JFlow controls the spread of information in a system. It is thus possible to guarantee that a datum labeled as `secret` will not be communicated to an adversary. Unfortunately, at the time of this writing, JFlow is restricted to a sequential subset of Java.

Other systems. We mention only two closely related works here: the Safe-Tcl security model because we briefly considered adopting a similar approach, and operating system research on interposition. Safe-Tcl takes the approach that each program script runs in its own interpreter with a subset of the libraries available and any communication between scripts is left to the operating system [34]. After consideration, we decided that this approach would push security onto the operating system without solving any of the underlying issues. In operating systems research, the issue of defining flexible security policies is a well-known problem. Several research groups have advocated interposition as a technique for enforcing security [16–18]. This technique relies on being able to intercept all requests to the operating system and interposing security checks to decide whether the request should be forwarded to the operating system. The hierarchical model of JavaSeal was designed to support arbitrary levels of transparent interposition.

4. Security in the JavaSeal kernel

In mobile agent systems, agents cannot reliably be associated with end users without assuming digitally signed agents and some widely accepted public key infrastructure. Even then, unless all kernel instances visited by an agent are also trusted, a malicious kernel could inject a virus into any visiting agent or modify its persistent state. The alternative is to treat every agent as a distinct principal and to provide protection mechanisms that isolate agents from each other. This view is adopted by most agent systems which differ only in the degree of isolation and the guarantees provided. At one extreme, Tschudin's `MO`[38] allows agents to execute in a shared address space (with some memory locations being hidden using keys); at the other extreme, `Safe-Tcl` [34] runs all programs in disjoint environments with no direct communication. Java-based agent systems are somewhere in the middle, with agents executing in a shared environment but with language enforced restrictions to control sharing [6, 27, 30].

For efficiency reasons, it is advantageous not to perform access control checks within an agent. This means viewing each agent as executing in its own protection domain, and focusing security measures on the interaction between agents. All accesses to objects belonging to other domains are checked by a security policy enforced by the reference monitor. At the very least, the events that require special attention from the security policy are: (1) agent creation, (2) agent termination, (3) thread creation within an agent, (4) loading of code and data into an agent, (5) message passing between agents. The kernel's role is to provide the hooks for security policies to be defined and to act as a reference monitor.

JavaSeal security measures consist of a kernel enforced boundary to guarantee that the kernel cannot be tampered with by incoming agents, along with three basic security properties: (1) confinement, (2) mediation and (3) faithfulness. The first two properties act together to provide the reference monitor properties R1 and R2. The last property safeguards seals against Trojan horses

4.1. JavaSeal kernel protection

Security measures can only be built on top of a trusted base. Protecting the kernel from tampering implies that the classes that implement the JavaSeal functionality must be protected from attacks originating in user code. These attacks include attempts by seals to invoke methods to which they have not been given access, attempts to modify fields of kernel objects, and attempts to trick the kernel into executing user code under its authority. The kernel consists of a small number of core classes in a package `seal.sys`. The core seal classes refer to classes in the JDK libraries. To prevent tampering with the kernel both sets of classes must be protected.

The kernel was designed to abide by the following three principles:

- *Total state isolation:* No updateable state can be shared between user code and the kernel classes.
- *Well-defined and small interfaces:* The classes and methods exported to user code are few and known in advance.
- *Fixed interfaces:* The kernel interface is made up of objects with fixed implementations.

The combined effect of applying these principles is that we obtain what amounts to a procedural interface between the kernel and user code. The interface cannot be extended dynamically through subtyping, nor can user code inject new behavior into the kernel. Since no updateable state is shared, users can only interact with the kernel through the methods defined in the interface. Finally, insisting on small interfaces simplifies the task of verifying that the code which implements it is actually secure.

Total state isolation is achieved by ensuring that the kernel classes exported to user code do not contain updateable fields. In Java, this corresponds to ensuring that public fields are final and that they refer to objects that themselves have no updateable fields. This restriction extends to class variables. For JDK classes the situation is more complex. The problem is twofold: first, we do not control their definition so it is not possible to enforce the above restrictions, and second, each JDK class may (transitively) refer to a large portion of the JDK libraries. If user code is allowed to update the state of any class that is in this transitive closure, then the update may have an impact on the kernel. Little can be done to prevent this since the JDK classes are out of our control. We therefore isolate JDK classes. Isolation is obtained by preventing user code from linking against any class in the JDK. Each seal is associated with a `SealLoader object` that enforces the constraint that the classes loaded into a seal belong either to the seal itself—and are extracted from its capsule—or belong to a set of safe classes. Safe classes are currently restricted to eight `seal.sys` kernel classes and 25 JDK classes containing no public updateable variables. The majority of these classes are exception classes, and `Object`, `String` and `StringBuffer`, as well as interfaces `Serializeable` and `Runnable`. The seal loader signals an error if any other class is referred to from user code.

We stress that the JavaSeal security model forbids seals from *sharing* JDK classes, and not from *using* these classes. In some cases, fresh copies of these classes can be loaded into the user seal, thus giving the seal its own, unshared, copy of some JDK classes. This is not always possible as the virtual machines forbid the reloading of some classes, so in general the solution to access library classes is to define service seals which are considered trusted and which are given less restrictive seal loaders. Service seals are installed by the local JavaSeal administrator, they cannot migrate, and must be coded with care to avoid security breaches.

One last aspect of state isolation is to ensure that aliasing (objects reachable through more than one variable) cannot be used to mount an attack. Currently, we adopt a simple solution—any updateable object must be copied upon entry to, or exit from, the kernel. An object is not updateable if it is a primitive Java type of if all of its fields are declared final and their types are not updateable.

A well defined and small kernel interface is obtained by limiting the kernel classes accessible to user code (public classes). In `seal.sys` there are only eight public classes, each of which has a small number of methods. This interface is fixed by declaring these classes as final, thus preventing user code from extending them. The only exception is the `Seal` class which has to be extended by users. In this case, all methods that the kernel relies on are declared final.

The kernel interface is fixed by ensuring the following two properties for all arguments to public kernel methods and corresponding return values: (1) All types used in the interface are either primitive types or are declared final. (2) For final classes, we further require that all public fields and method arguments of these classes be final. These restrictions ensure that the kernel will never invoke a method defined in user code. We summarize these restrictions by defining the following notion of safety.

Definition 1 (Fixed). A class is fixed if it is `final` and the types all `public [static]` fields and all arguments to `public [static]` methods are fixed. Primitive types are fixed.

Definition 2 (Update-safe). A class C in package P is update-safe with respect to a package P' where $P \neq P'$, if for each `public [static]` field of type T, the field is `final` and T is either

—a primitive type or
—T is fixed and T is update-safe with respect to P'

Definition 3 (Updateable). A class is updateable if it contains a non`final` field or if one of its fields is updateable. Arrays are updateable.

Definition 4 (Safe). A class C in P is safe with respect to a package P' where $P \neq P'$, if it is update-safe with respect to P' and all updateable arguments of `public [static]` methods are copied.

JavaSeal has been implemented according to these restrictions: all public classes in the `seal.sys` package are *safe* with respect to all classes in user packages.

Selective access modifiers. The requirements for safety given above are somewhat restrictive. They imply that classes like the `Capsule` class which are exported to user code must be final. On the other hand, good object-oriented design principles suggest splitting the generic capsule class into two subclasses: one for data capsules and another for seal capsules.

This suggests that access modifiers provided by the Java language are not powerful enough. We would like to specify that user code should treat `Capsule` as a `final`, but that kernel code be allowed to extend the class as long as the extensions remain transparent to user code. Similarly, for design reasons it is often convenient to split the implementation of core classes into different packages. In Java this means that if fields, methods or classes have to be visible from other parts of the kernel located in the other packages, these fields must be `public`. However, this would make these fields accessible to user code. Again, we would like to specify that some (public) fields be considered as private in user code.

To address these problems, JavaSeal employs *selective access modifiers* which are enforced at load time by the seal loader. Selective access modifiers are used to define multiple views on object classes. They are defined in directive files and have the following grammar:

```
Modifier ::= export | final | private | protected

FQN      ::= ( name .)* name

           | ( name .)* name()

M-Exp    ::= <Modifier> <FQN>;
```

Selective access modifiers provide fine-grained static access control as each class loader can present a slightly different view of the types it is linking agents against. Note that for reasons related to dynamic checks effected by the virtual machine, modifiers can only strengthen existing modifiers. An example directive sequence is the following:

```
export java.lang.Object;
final seal.sys.Capsule;
private java.lang.Object.getClass();
```

The first directive specifies that class `Object` is exported to user code. In other words, seal objects may link against this (shared) class. The second specifies that a class is to be treated as final; thus no subclasses are allowed in the seal. The last directive specifies that a method of a class cannot be used within a seal. These modifiers are read in by the `SealLoader` and all classes loaded are checked to conform to their restrictions.

Denial of service. JavaSeal maintains a mapping from threads to seals. We are considering using an approach similar to JRes [14] to account for the processing time consumed by each seal on a per thread basis. Currently the only safeguard implemented in JavaSeal is to prevent finalizer methods from crashing the virtual machine. A finalizer is a method that runs when an object is about to be garbage collected. If a finalizer fails to terminate the garbage collector will block and eventually the virtual machine will run out of memory. The JavaSeal seal loader implement a simple static analysis of the bytecode to prevent nonterminating finalizers: loops and method calls are forbidden in finalizers. While this is restrictive, we have not encountered practical situations where finalizers were needed.

Termination. Seal termination is achieved by asynchronously stopping all of the strands of a seal and setting all domain-specific kernel pointers to null. To ensure that the seal does not catch the `ThreadDeath` exception, we rewrite the bytecode of all exception handlers susceptible of catching the exception to ensure that the exception is propagated. Memory will eventually be reclaimed by the garbage collector. The restrictions on finalizers ensure that new threads cannot be started in finalizers and that they terminate

4.2. Seal security properties

We propose three security properties as building blocks for more elaborate security policies.

Confinement. Confinement is a property that a seal that has not explicitly been granted communication privileges cannot interact with or affect in any way the behavior of other seals. Confinement is the default state in the system; a seal is confined as long as no portals are opened for it by its parent in the hierarchy.

A seal ceases to be confined as soon as portals are opened. To open a portal a parent must know under which name its subseal is running. This property is quite useful as it is possible to constrain portals to specific parts of an agent's code by controlling the scope of names. For instance, if a name is thrown away right after unwrapping a new seal from its capsule:

```
unwrap(capsule, Name.makeFresh());
```

then the new seal will run under a name unknown to its parent. The parent will therefore not be able to open portal for this seal. Apart from the issue of resource usage, this operation is equivalent to unwrapping a void capsule:

```
unwrap(null, Name.makeFresh());
```

Confinement can be recovered by closing all portals—a form of revocation—or by renaming a seal to a fresh name:

```
rename(thisName, Name.makeFresh());
```

Achieving confinement requires overcoming several difficulties. The biggest challenge is to prevent sharing of objects. Any shared object would break confinement as, even if the object itself is not mutable, it can be locked and locks can be used to communicate. Confinement thus requires a by-deep-copy communication mode, such as JavaSeal capsules, and it also requires that no global variables be shared between seals. This is achieved by loading distinct instances of the classes used by each seal. Furthermore, seals should not be granted access to system variables such as for example the `threadCount` of the Thread class, as they represent low bandwidth storage channels. JavaSeal avoids these threats by isolating seals and preventing any form of sharing. The isolation imposed by seal loaders may appear a bit drastic as we effectively separate each seal from most of the JDK. However, this is the only way to ensure confinement. While program analysis could be used to prove some of the JDK classes free of storage channels, we decided against this solution as it would force to analyze the configuration of each JVM on which JavaSeal is installed and would thus require significant support from the system loader.

Mediation. Mediation states that any request emitted by an agent can be subject to a userdefined security policy. This is achieved by allowing interposition of security code between a seal and the envlet that provides the requested service. Mediation can be either static, which is the case if policies cannot be altered once put in place, or dynamic, in which case different interposition policies can be applied to an agent during its life time. Mediation is not straightforward in Java because of the difficulty of controlling communication via shared objects. JavaSeal with its restricted communication primitives makes mediation easy. To request any service, a seal must send a message to its parent. A security policy can thus be implemented as an envlet that filters requests of its children based on user defined rules.

Faithfulness. Faithfulness guarantees that agents are executed with their own code. This is to prevent injection of untrusted code (code spoofing attacks). In plain Java, faithfulness is endangered by a weak version control model based on easily spoofed version numbers, as well as subtyping based code injection attacks if a victim is tricked into executing a method of a subtype of the expected type. JavaSeal enforces faithfulness with seal loaders. Creating a seal causes all of its classes to be extracted from the seal archive; the seal loader never reuses previously loaded classes. Code injection is prevented by ensuring that the contents of capsules exchanged between agents do not introduce new classes into a seal. One implication of this property is that the set of seal classes is immutable; thus the code portion of a seal archive will not change after migration. This has the advantage that the class can be digitally signed by the seal creator.

5. HyperNews: An agent-based newspaper application

HyperNews is a mobile agent based system developed at the University of Geneva for the distribution of newspaper articles over the Web [32]. The project was founded by Hebdo, a Swiss weekly news magazine. The primary requirement specified by Hebdo was *payment enforcement*; this means that any article read by a client must be paid for.

The HyperNews system was implemented over the JavaSeal agent kernel. The goal of this section is to overview this implementation and to examine how JavaSeal's design accommodated the implementation. We begin with a description of the HyperNews system and then proceed with its implementation.

5.1. The HyperNews system

The goal of HyperNews is the distribution of newspaper articles over the Web [32]. A newspaper provider that publishes an article must be paid by every client who reads that article.

Mobile-agent technology was chosen for HyperNews for the following reasons:

- An article is an object in the object-oriented sense. It is self-contained in that its code verifies a receipt of payment each time that a client reads the article. This code starts a payment process if the client is reading the article for the first time, that is, when an attempt to read the article is made without a receipt.
- Any user can download an article to his site; this means that the object must be able to execute on any site. The connotation of agent in HyperNews is that of a mobile object, and this is one reason why JavaSeal was chosen to implement HyperNews.

HyperNews payment is done with the aid of JavaCard smart cards [32]. The card is periodically loaded with a fixed sum of money. Communication with the smartcard uses a standard card reader attached to the user's PC. When a user reads an article for the first time, the smartcard is contacted, the price of the article is deducted from the smartcard, and a receipt signed using a key only known to the smart card is returned. The receipt is verified on each subsequent access to the article.

The overall structure of the HyperNews system is shown in Figure 4. In this example, the system is configured for a user who receives articles from the news providers

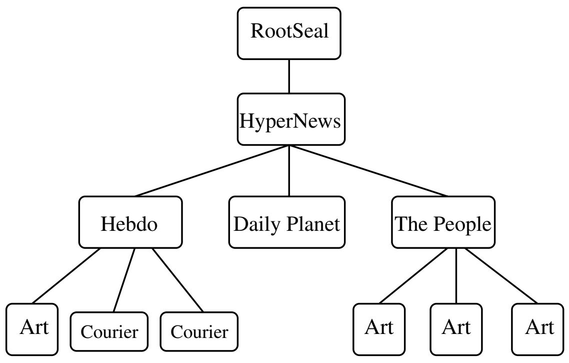

Figure 4. Hierarchy of the HyperNews system.

Daily Planet, Hebdo and the People. A proxy provider environment, known as a *news feed*, is installed on the user's machine for each provider that the user subscribes to. The news feed represents the newspaper provider on the client site. It is responsible for storing articles that belong to that provider, and for downloading information on new articles from the provider site when needed. Communication between a proxy and its provider is encrypted with a symmetric session key known only to the proxy and provider. Messages between the proxy and provider are carried by courier complets.

Article agents do not have a fixed itinerary in HyperNews. Once created at the provider site, they can migrate to *any* client site, and be read after a payment is made. Once an article arrives at a site, a mutual authentication protocol is run between the article and the main HyperNews agent. The article is verified by furnishing an article information string signed by the provider; the HyperNews agent is verified by having it sign a message that the article sends to it on arrival. After successful authentication, the article is passed to the owning provider's news feed. `Read` requests from the user are filtered by the news feed and passed onto the article agent. The article replies after a payment or receipt verification procedure. Thus, article agents operate in "reply-mode."

Courier agents in contrast have a fixed itinerary, as they operate between a client and a bank, certificate authority or provider. These agents are responsible for a number of tasks; these include: obtaining a session key from the bank for the card update procedure, obtaining certificates from the certificate authority for verifying public keys, obtaining a summary of new articles from the provider so that the user can subsequently decide whether to download new articles. A courier agent is created by a news feed envlet on a client site, and then given instructions about its task. It is sent to the destination site, where the main `HyperNews` agent authenticates it. The agent recovers the information it came for and returns to the client site. On return, the mutual authentication protocol is initiated by the news feed between itself and the returning courier agent. If successful, the courier agent delivers the information that it is carrying.

5.2. Implementing HyperNews over JavaSeal

HyperNews on JavaSeal is around 30000 lines of code, not including the encryption and GUI libraries. JavaSeal has proved to be a particularly useful platform for HyperNews for several reasons: the organizational hierarchy of the system is naturally represented by a seal hierarchy, articles and news feeds are protected from one another, and the system continues to function off-line.

As seen from Figure 4, the hierarchy is exploited in HyperNews to organize the articles and courier agents of the system. On bootstrap, the `RootSeal` creates the main `HyperNews` agent. This seal instantiates an envlet for each news feed used by the client. Article and courier agents are stored as complets within their owning news feed envlet.

Concerning security, the security properties of the JavaSeal hierarchy are particularly important to HyperNews. For instance, each article or courier agent and news feed is completely encapsulated, and the information within it cannot therefore be stolen or tampered with by code running on behalf of competing news providers. Further, since communication between seals is only between children and parents, messages exchanged between article or courier agents and their new feed envlets are also secure from other news feed envlets. This guarantee relies on the `HyperNews` envlet which news feeds trust. Since this envlet is a parent of the news feed envlets, it ensures that no messages are exchanged between envlets, and the mutual authentication protocol ensures that arriving courier and article agents do not go to the wrong destination news feed.

There is no situation in HyperNews where the movement of agents requires full thread mobility. In fact, whenever either a courier or article agent moves, the first action that it must take on arrival is to authenticate itself to the system and news feed and to have the news feed authenticate itself to it. For instance, a courier agent that wishes to talk to the Daily Planet envlet must prove to `HyperNews` that it comes from the Daily Planet provider before HyperNews serializes the agent and passes it down to the Daily Planet envlet. JavaSeal is useful here since, not only does it allow `HyperNews` to control all agents that enter and leave the system, but the system is not encumbered with the unnecessary overhead of full thread mobility.

The JavaSeal kernel and HyperNews seals run on the same JVM as the HyperNews services. The services that HyperNews requires include a GUI control panel for the application, an interface to the smart card reader, a storage subsystem for storing articles locally as well as a HTTP interface to a netscape browser. This example illustrates how an application can require a variety of services, but where the choice of service not be restricted by the agent security model, and where the services used do not damage agent security. In HyperNews, the services are started by `RootSeal` on application launch, though outside of JavaSeal. That is, their classes are loaded using the system loader. In this way, there is no restriction on the classes that a service chooses to load; further, an agent seal can still not share data directly with a service, so service protection is not compromised. The architecture is illustrated in Figure 5; the `Bridge` object stores messages exchanged between `RootSeal` and the services.

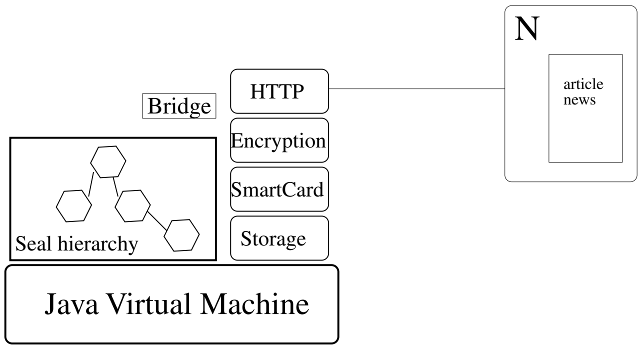

Figure 5. Architecture of the HyperNews services.

Finally, concerning security, much use is made of encryption by the architecture. An article's contents is encrypted with a key known only to the provider and which is stored on the smartcard. During the payment and receipt verification procedure, the key is passed to the news feed, which decrypts the contents, and passes the contents to the browser used to read the article. This key is then destroyed by the news feed in order to reduce the risk of the key being compromised. This approach has a risk of the article key being disclosed if for instance the underlying system has been tampered with. However, it was felt that this risk is worthwhile since an article only costs a few cents, and modifying one's JVM to avoid payment is hardly worth it for such a price.

6. Conclusion

This paper has described the JavaSeal platform—a secure kernel for mobile environments (envlets) and mobile objects (complets). JavaSeal is a kernel in that it offers minimal service functionality. One motivation for this structure is that since services differ between sites, one must be able to build different services over the kernel. JavaSeal is secure in that it isolates agents from one another. Security is built over Java's typing mechanism, and also extends the class loader verifier to ensure that seals do not share forbidden or untrusted classes.

The main lesson that we have learned from the JavaSeal project is that it is possible to implement a secure kernel based on Java. We qualify security in this case as strong separation between agents, and between agents and services. Of course, some covert channels may remain in the kernel though we believe these to be of insignificant bandwidth compared to the storage channels that can exist in the JDK service classes.

Like others [2], we have also learned that full migration—where an agent's thread state is also moved—is not easy in the Sun JDK due to low-level implementation issues. The problem is that full migration means being able to manipulate object locks, and in Sun's JVMs, locking is implemented in machine dependent code. As it happens, we became less convinced of the necessity of full migration during the course of the project. When an agent moves, it typically must execute an authentication procedure, which means interrupting its thread of execution in any case. Further, services on different sites that an agent uses can differ in functionality and security policies, so an agent might not be able to continue running after migration in a transparent way.

Acknowledgments

This research was conducted in the Object Systems Group at the University of Geneva. The implementation of JavaSeal is due to W. Binder, C. Bryce, M. Oriol, and K. Taha. The JavaSeal project was funded by the Swiss National Science Foundation (research grant: SPP-ICS-5003 45335) to whom we extend our gratitude.

References

1. M. Abadi, M. Burrows, B. Lampson, and G. Plotkin, "A calculus for access control in distributed systems (invited)," in J. Feigenbaum, (ed.), *Proceedings of Advances in Cryptology (CRYPTO '91),* vol. 576, Lecture Notes in Computer Science, Springer: Berlin, Germany, 1992, pp. 1–23.
2. Acharya, M. Ranganathan, and J. Saltz, "Sumatra: A Language for Resource-Aware Programs," in *Mobile Object Systems: Towards the Programmable Internet,* vol. 1222, Lecture Notes in Computer Science, Springer-Verlag: Berlin, April 1997.
3. G. Agha, *Actors—A Model of Concurrent Computation in Distributed Systems*, MIT Press: Cambridge, MA, 1986.
4. 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, 6 August 1998.
5. M. Baldi, S. Gai, and G. P. Picco, "Exploiting code mobility in decentralized and flexible network management," in *Proceedings of the First International Workshop on Mobile Agents,* April 1997. pp. 13–26, Berlin, Germany.
6. J. Baumann, F. Hohl, K. Rothermel, and M. Strasser, "Mole—Concepts of a mobile agent system," *World Wide Web*, vol. 1(3), pp. 123–137, 1998.
7. B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers, "Extensibility, safety and performance in the SPIN operating system," in *Proceedings of the 15th Symposium on Operating Systems Principles,* Copper Mountan, Colorado, December 1995, pp. 267–284.
8. B. Bokowski and J. Vitek, "Confined types," in *Proceedings of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '99),* Denver, Colorado, USA, November 1999.
9. Q. Bradley, R. Horspool, and J. Vitek, "JAZZ, compression of Java bytecode," in *CASCON '98*, 1998.
10. L. Cardelli, "Abstractions for mobile computations," in J. Vitek and C. Jensen (eds.), *Secure Internet Programming: Security Issues for Distributed and Mobile Objects,* Springer-Verlag: Berlin, 1999.
11. L. Cardelli and A. D. Gordon, "Mobile ambients," in M. Nivat (ed.), *Foundations of Software Science and Computational Structures,* vol. 1378 in LNCE, Springer-Verlag: Berlin, 1998, pp. 140–155.
12. D. Chess, C. G. Harrison, and A. Kershenbaum, "Mobile Agents: Are They a Good Idea?," in J. Vitek and C. Tschudin (eds.), *Mobile Object Systems-Towards the Programmable Internet,* Lecture Notes in Computer Science, Springer-Verlag: Berlin, Germany, 1997, pp. 25–47.
13. D. M. Chess, "Security issues in mobile code systems," in G. Vigna (ed.), *Mobile Agents and Security,* vol. 1419, Lecture Notes in Computer Science, Springer-Verlag: New York, 1998.
14. G. Czajkowski and T. von Eicken, "JRes: A resource accounting interface for Java," *ACM SIGPLAN Notices,* vol. 33(10), pp. 21–35, October 1998.
15. D. Denning, "A lattice model of secure information flow," *Communications of the ACM,* vol. 19(5), pp. 236–243, 1976.
16. T. Fine and S. E. Minear, "Assuring distributed trusted mach," in IEEE, editor, *Proceedings of the 32nd IEEE Conference on Decision and Control*, San Antonio, TX, USA, 15–17 December 1993, pages 206–217, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, IEEE Computer Society Press: Silver Spring, MD, 1993, pp. 206–217.
17. D. P. Ghormley, D. Petrou, S. H. Rodrigues, and T. E. Anderson, "SLIC: An extensibility system for commodity operating systems," in *Proceedings of the USENIX 1998 Annual Technical Conference,* Berkeley, USA, USENIX Association, 15–19 June 1998, pp. 39–52.
18. D. P. Ghormley, S. H. Rodrigues, D. Petrou, and T. E. Anderson, "Interposition as an operating system extension mechanism," Technical Report CSD-96-920, University of California, Berkeley, 9 April 1997.
19. I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer, "A secure environment for untrusted helper applications," in *Proceedings of the 6th Usenix Security Symposium,* San Jose, CA, July 1996.
20. D. Gollman, *Computer Security,* John Wiley & Sons: New York, 1999.
21. L. Gong, "Java security architecture (JDK 1.2)," Technical report, JavaSoft, July 1997. Revision 0.5.

22. R. S. Gray, "Agent Tcl: A flexible and secure mobile-agent system," Technical Report PCS-TR98-327, Dartmouth College, Computer Science, Hanover, NH, January 1998.
23. G. Hamilton and G. Kougioris, "The spring nucleus: a micro-kernel for objects," in *Proceedings of the Usenix Summer Conference,* Ohio, USA, June 1994, pp. 147–159.
24. C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken, "Implementing Multiple Protection Domains in Java," Technical Report 97-1660, Cornell University, Department of Computer Science, 1997.
25. N. Jamali, P. Thati, and G. A. Agha, "An Actor-based architecture for customizing and controlling agent ensembles," *IEEE Intelligent Systems,* 1998.
26. E. Jul, *Object Mobility in a Distributed Object-Oriented System,* PhD thesis, University of Washington, Computer Science Department, December 1988.
27. N. Karnik and A. Tripathi, "Agent server architecture for the ajanta mobile-agent system," in *1998 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '98),* July 1998.
28. B. W. Lampson, "A note on the confinement problem," *Communications of the ACM*, vol. 16, 1973.
29. D. B. Lange and M. Oshima, "Mobile agents with Java: The Aglet API," *World Wide Web Journal,* 1998.
30. D. B. Lange and M. Oshima, *Programming and Deploying Mobile Agents with Java Aglets,* Addison-Wesley: Reading, MA, USA, September 1998.
31. J. Lepreau, B. Ford, and M. Hibler, "The persistent relevance of the local operating system to global applications," in *Proceedings of the 1996 SIGOPS European Workshop,* 1996.
32. J. -H. Morin and D. Konstantas, "Commercialization of electronic information," *Journal of End User Computing,* vol. 12(2), pp. 20–32, April–June 2000.
33. A. C. Myers, "Jflow: Practical static information flow control," in *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL 99),* 1999.
34. J. K. Ousterhout, J. Y. Levy, and B. B. Welch, *The Safe-Tcl Security Model*, vol. 1419, Lecture Notes in Computer Science, 1998, pp. 217–??
35. T. Sander and C. F. Tschudin, "Protecting mobile agents against malicious hosts," in G. Vigna (ed.), *Mobile Agents and Security,* vol. 1419, Lecture Notes in Computer Science, Springer-Verlag, New York, 1998.
36. W. Theilmann and K. Rothermel, "Disseminating mobile agents for distributed information filtering," in *First International Symposium on Agent Systems and Applications (ASA '99)/Third International Symposium on Mobile Agents (MA '99),* Palm Springs, CA, USA, October 1999, pp. 152–161.
37. Tromsø University and Cornell University, TACOMA Project, http://www.cs.uit.no/DOS/Tacoma/.
38. C. Tschudin, "The messenger environment M0—A condensed description," in *Mobile Object Systems: Towards the Programmable Internet,* vol. 1222, Lecture Notes in Computer Science, Springer-Verlag: Berlin 1997, pp. 149–156.
39. G. Vigna, *Cryptographic Traces for Mobile Agents*, vol. 1419, Lecture Notes in Computer Science, Springer-Verlag: New York, 1998, pp. 137–149.
40. G. Vigna, *Mobile Agents and Security,* vol. 1419, Lecture Notes in Computer Science, Springer-Verlag: New York, USA, 1998.
41. J. Vitek, *The Seal Model of Mobile Computations,* Ph.D. thesis, University of Geneva, 1999.
42. D. Volpano, C. Irvine, and G. Smith, "A sound type system for secure flow analysis," *Journal of Computer Security,* vol. 4, pp. 167–187, May 1996.
43. T. Von Eicken, C.-C. Chang, G. Czajkowski, and C. Hawblitzel, *J-Kernel: A Capability-Based Operating System for Java*, vol. 1603, Lecture Notes in Computer Science, 1999, pp. 369–394.
44. J. E. White, "Telescript technology: The foundation for the electronic marketplace," White paper, General Magic, Inc: Mountain View, CA, 1994.
45. U. G. Wilhelm, L. Buttyàn, and S. Staamann, "On the problem of trust in mobile agent systems," in *Symposium on Network and Distributed System Security*, Internet Society, March 1998.