J-Kernel: a Capability-Based Operating System for Java

Thorsten von Eicken, Chi-Chao Chang, Grzegorz Czajkowski,
Chris Hawblitzel, Deyu Hu, and Dan Spoonhower

Department of Computer Science
Cornell University

Abstract. Safe language technology can be used for protection within a single address space. This protection is enforced by the language's type system, which ensures that references to objects cannot be forged. A safe language alone, however, lacks many features taken for granted in more traditional operating systems, such as rights revocation, thread protection, resource management, and support for domain termination. This paper describes the J-Kernel, a portable Java-based protection system that addresses these issues. J-Kernel protection domains can communicate through revocable capabilities, but are prevented from directly sharing unrevocable object references. A number of micro-benchmarks characterize the costs of language-based protection, and an extensible web and telephony server based on the J-Kernel demonstrates the use of language-based protection in a large application.

1 Introduction

The notion of moving code across the network to the most appropriate host for execution has become commonplace. Most often code is moved for efficiency, but sometimes it is for privacy, for fault-tolerance, or simply for convenience. The major concern when moving code is security: the integrity of the host to which it is moved is at risk, as well as the integrity of the computation performed by the moved code itself.

A number of techniques have been used to place protection boundaries between so-called "untrusted code" moved to a host and the remainder of the software running on that host. Traditional operating systems use virtual memory to enforce protection between processes. A process cannot directly read and write other processes' memory, and communication between processes requires traps to the kernel. By limiting the traps an untrusted process can invoke, it can be isolated to varying degrees from other processes on the host. However, there's little point in sending a computation to a host if it cannot interact with other computations there; inter-process communication must be possible.

The major problems encountered when using traditional operating system facilities to isolate untrusted code are deciding whether a specific kernel trap is permissible or not and overcoming the cost of inter-process communication. The semantic level of kernel traps does not generally match the level at which protection policies are specified when hosting untrusted code. In addition, the objects on which the traps act are the ones managed by the kernel and not the ones provided by the hosting environment. Regarding performance, despite a decade of research leading to a large number of fast inter-process communication mechanisms [4,9,24], the cost of passing through the kernel and of switching address spaces remains orders of magnitude larger than that of calling a procedure.

In the context of mobile code, language-based protection is an attractive alternative to traditional operating system protection mechanisms. Language-based protection rests on the safety of a language's type system, which ensures that the abstractions provided by the language's types are enforced. A type system acts as a simple access control mechanism: it limits the objects that a computation can access (there is no way to "forge a pointer" to an object), and it limits the operations that code can perform on accessible objects.

The attraction of language-based protection is twofold: precision of protection and performance of communication across protection boundaries. Language-based protection mechanisms allow access rights to be specified with more precision than traditional virtual-memory based mechanisms: the data items to which access is permitted as well as the types of accesses permitted can be specified more finely. For example, in Java, access can be granted precisely to individual objects and even only to certain object fields using the `public` qualifier. In addition, with language-based protection, calls across protection boundaries could potentially be as cheap as simple function calls, enabling as much communication between components as desired without performance drawbacks.

But language-based protection alone does not make an operating system. Several projects [1,2,3,8,11,14,35] have recently described how to build protection domains around components in a safe language environment. The central idea is to use object references (i.e., pointers to objects) as capabilities for cross-domain communication. Object references in safe languages are unforgeable and can thus be used to confer certain rights to the holder(s). In an object-oriented language, the methods applicable to an object are in essence call gates. However, this approach, while both flexible and fast, suffers from two limitations: there is no way to revoke access to object references, and there is no way to track which domain owns which objects. This leads to severe problems with domain termination and resource accounting.

The J-Kernel [16] is an operating system layered on top of a Java Virtual Machine (JVM) that introduces additional mechanisms borrowed from traditional operating systems to provide the features missing at the language level. The goal of the J-Kernel is to define clear boundaries between protection domains, which are called *tasks* in the J-Kernel. This makes resource management and task termination tractable, and simplifies the analysis of inter-task communication. The boundaries are established by limiting the types of objects that can be shared between tasks. In the J-Kernel, only special objects called *capabilities* may be shared; all other objects are confined to single tasks. This allows the J-Kernel to equip capabilities with features like revocation, without adding any overheads to ordinary, non-capability objects.

The main benefits of the J-Kernel are a highly flexible protection model, low overheads for communication between software components, and operating system independence. The current J-Kernel implementation is written entirely in Java [13] and runs on standard Java Virtual Machines (JVMs). This implementation was chosen for practical reasons—Java is emerging as the most widely used general-purpose safe language, and dependable virtual machines are widespread and easy to work with. While Java in itself does allow for multiple protection domains within a single JVMs using the sandbox model for applets, that model is currently very restrictive. It lacks many of the characteristics that are taken for granted in more traditional systems and, in particular, does not provide a clear way for different protection domains to communicate with each other.

The J-Kernel also offers a simple form of resource management based on the JRes interface [7]. The interface allows accounting for resource consumption on per-thread and per-task basis. The resources accounted for are CPU time, heap memory, and network traffic. The implementation of JRes contains a small native component for interfacing to the threads package but is mostly based on Java bytecode rewriting.

Language-based protection does have drawbacks. First, code written in a safe language tends to run more slowly than code written in C or assembly language, and thus the improvement in cross-domain communication may be offset by an overall slowdown. Much of this slowdown is due to current Java just-in-time compilers optimized for fast compile times at the expense of runtime performance. However, even with sophisticated optimization it seems likely that Java programs will not run as fast as C programs. Second, all current language-based protection systems are designed around a single language, which limits developers and doesn't handle legacy code. Software fault isolation [34] and verification of assembly language [26, 27, 28, 32] may someday offer solutions, but are still an active area of research.

2 Language-Based Protection Background

In an unsafe language, any code running in an address space can potentially modify any memory location in that address space. While, in theory, it is possible to prove that certain pieces of code only modify a restricted set of memory locations, in practice this is very difficult for languages like C and arbitrary assembly language [5, 27], and cannot be fully automated. In contrast, the type system and the linker in a safe language restrict what operations a particular piece of code is allowed to perform on which memory locations.

The term *namespace* can be used to express this restriction: a namespace is a partial function mapping names of operations to the actions taken when the operations are executed. For example, the operation "read the field `out` from the class `System`" may perform different actions depending on what class the name `System` refers to.

Protection domains around software components can be constructed in a safe language system by providing a separate namespace for each component. Communication between components can then be enabled by introducing sharing among namespaces. Java provides three basic mechanisms for controlling namespaces: selective sharing of object references, static access controls, and selective class sharing.

Selective Sharing of Object References Two domains can selectively share references to objects by simply passing each other these references. In the example below, `method1` of class A creates two objects of type A, and passes a reference to the first object to `method2` of class B. Since `method2` acquires a reference to `a1`, it can perform operations on it, such as incrementing the field `j`. However, `method2` was not given a reference to `a2` and thus has no way of performing any operations on it. Java's safety prevents `method2` from forging a reference to `a2`, e.g., by casting an integer holding `a2`'s address to a pointer.

```
class A {
    private int i;
    public int j;
    public static void method1() {
        A a1 = new A();
        A a2 = new A();
        B.method2(a1);
    }
}

class B {
    public static void method2(A arg) {
        arg.j++;
    }
}
```

Static Access Control The preceding example demonstrated a very dynamic form of protection—methods can only perform operations on objects to which they have been given a reference. Java also provides static protection mechanisms that limit what operations a method can perform on an object once the method has acquired a reference to that object. A small set of modifiers can change the scope of fields and methods of an object. The two most common modifiers, `private` and `public`, respectively limit access to methods in the same class or allow access to methods in any class. In the classes shown above, `method2` can access the public field `j` of the object `a1`, but not the private field `i`.

Selective Class Sharing Domains can also protect themselves through control of their *class namespace*. To understand this, we need to look at Java's class loading mechanisms. To allow dynamic code loading, Java supports user-defined *class loaders* which load new classes into the virtual machine at run-time. A class loader fetches Java bytecode from some location, such as a file system or a URL, and submits the bytecode to the virtual machine. The virtual machine performs a verification check to make sure that the bytecode is legal, and then integrates the new class into the machine execution. If the bytecode contains references to other classes, the class loader is invoked recursively in order to load those classes as well.

Class loaders can enforce protection by making some classes visible to a domain, while hiding others. For instance, the example above assumed that classes A and B were visible to each other. However, if class A were hidden from class B (i.e. it did not appear in B's class namespace), then even if B obtains a reference to an object of type A, it will not be able to access the fields `i` and `j`, despite the fact that `j` is public.

2.1 Straight-Forward Protection Domains: The *Share Anything* Approach

The simple controls over the namespace provided in Java can be used to construct software components that communicate with each other but are still protected from one another. In essence, each component is launched in its own namespace, and can then share any class and any object with other components using the mechanisms described above. While we will continue to use the term *protection domain* informally to refer to these protected components, we will argue that it is impossible to precisely define protection domains when using this approach.

The example below shows a hypothetical file system component that gives objects of type `FileSystemInterface` to its clients to give them access to files. Client domains make cross-domain invocations on the file system by invoking the `open` method of a `FileSystemInterface` object. By specifying different values for `accessRights` and `rootDirectory` in different objects, the file system can enforce different protection policies for different clients. Static access control ensures that clients cannot modify the `accessRights` and `rootDirectory` fields directly, and one client cannot forge a reference to another client's `FileSystemInterface` object.

```
class FileSystemInterface {
    private int accessRights;
    private Directory rootDirectory;
    public File open(String fileName) {}
}
```

The filesystem example illustrates an approach to protection in Java that resembles a capability system. Several things should be noted about this approach. First, this approach does not require any extensions to the Java language—all the necessary mechanisms already exist. Second, there is very little overhead involved in making a call from one protection domain to another, since a cross-domain call is simply a method invocation, and large arguments can be passed by reference, rather than by copy. Third, references to any object may be shared between domains since the Java language has no way of restricting which references can be passed through a cross-domain method invocation and which cannot.

When we first began to explore protection in Java, this *share anything* approach seemed the natural basis for a protection system, and we began developing on this foundation. However, as we worked with this approach a number of problems became apparent.

Revocation The first problem is that access to an object reference cannot be revoked. Once a domain has a reference to an object, it can hold on to it forever. Revocation is important in enforcing the principle of least privilege: without revocation, a domain can hold onto a resource for much longer than it actually needs it.

The most straightforward implementation of revocation uses extra indirection. The example below shows how a revocable version of the earlier class `A` can be created. Each object of `A` is wrapped with an object of `AWrapper`, which permits access to the wrapped object only until the `revoked` flag is set.

```
class A {
  public int meth1(int a1, int a2) {}
}

class AWrapper {
  private A a;
  private boolean revoked;
  public int meth1(int a1, int a2) {
    if(!revoked) return a.meth1(a1, a2);
    else throw new RevokedException();
  }
  public void revoke() { revoked=true; }
  public AWrapper(A realA) {
    a = realA; revoked = false; }
}
```

In principle, this solves the revocation problem and is efficient enough for most purposes. However, our experience shows that programmers often forget to wrap an object when passing it to another domain. In particular, while it is easy to remember to wrap objects passed as arguments, it is common to forget to wrap other objects to which the first one points. In effect, the default programming model ends up being an unsafe model where objects cannot be revoked. This is the opposite of the desired model: safe by default and unsafe only in special cases.

Inter-domain Dependencies and Side Effects As more and more object references are shared between domains, the structure of the protection domains is blurred, because it is unclear from which domains a shared object can be accessed. For the programmer, it becomes difficult to track which objects are shared between protection domains and which are not, and the Java language provides no help as it makes no distinction between the two. Yet, the distinction is critical for reasoning about the behavior of a program running in a domain. Mutable shared objects can be modified at any time in by other domains that have access to the object, and a programmer needs to be aware of this possible activity. For example, a malicious user might try to pass a byte array holding legal bytecode to a class loader (byte arrays, like other objects, are passed by reference to method invocations), wait for the class loader to verify that the bytecode is legal, and then overwrite the legal bytecode with illegal bytecode which would subsequently be executed. The only way the class loader can protect itself from such an attack is to make its own private copy of the bytecode, which is not shared with the user and is therefore safe from malicious modification.

Domain Termination The problems associated with shared object references come to a head when we consider what happens when a domain must be terminated. Should all the objects that the domain allocated be released, so that the domain's memory is freed up? Or should objects allocated by the domain be kept alive as long as other domains still hold references to them? From a traditional operating systems perspective, it seems natural that when a process terminates all of its objects disappear, because the address space holding those objects ceases to exist. On the other hand, from a Java perspective, objects can only be deallocated when there are no more reachable references to them.

Either solution to domain termination leads to problems. Deallocating objects when the domain terminates can be extremely disruptive if objects are shared at a fine-grained level and there is no explicit distinction between shared and non-shared objects. For example, consider a Java `String` object, which holds an internal reference to a character array object. Suppose domain 2 holds a `String` object whose internal character array belongs to domain 1. If domain 1 dies, then the `String` will suddenly stop working, and it may be beyond the programmer's ability to deal with disruptions at this level.

On the other hand, if a domain's objects do not disappear when the domain terminates, other problems can arise. First, if a server domain fails, its clients may continue to hold on to the server's objects and attempt to continue using them. In effect, the server's failure is not propagated correctly to the clients. Second, if a client domain holds on to a server's objects, it may indirectly also hold on to other resources, such as open network connections and files. A careful server implementation could explicitly relinquish important resources before exiting, but in the case of unexpected termination this may be impossible. Third, if one domain holds on to another domain's objects after the latter exits, then any memory leaks in the terminated domain may be unintentionally transferred to the remaining one. It is easy to imagine scenarios where recovery from this sort of shared memory leak requires a shutdown of the entire VM.

Threads By simply using method invocation for cross-domain calls, the caller and callee both execute in the same thread, which creates several potential hazards. First, the caller must block until the callee returns—there is no way for the caller to gracefully back out of the call without disrupting the callee's execution. Second, Java threads support methods such as `stop`, `suspend`, and `setPriority` that modify the state of a thread. A malicious domain could call another domain and then suspend the thread so that the callee's execution gets blocked, perhaps while holding a critical lock or other resource. Conversely, a malicious callee could hold on to a `Thread` object and modify the state of the thread after execution returns to the caller.

Resource Accounting A final problem with the simple protection domains is that object sharing makes it difficult to hold domains accountable for the resources that they use, such as processor time and memory. In particular, it is not clear how to define a domain's memory usage when domains share objects. One definition is that a domain is held accountable for all of the objects that it allocates, for as long as those objects remain alive. However, if shared objects aren't deallocated when the domain exits, a domain might continue to be charged for shared objects that it allocated, long after it has exited. Perhaps the cost of shared objects should be split between all the domains that have references to the object. However, because objects can contain references to other objects, a malicious domain could share an object that looks small, but actually contains pointers to other large objects, so that other domains end up being charged for most of the resources consumed by the malicious domain.

Summary The simple approach to protection in Java outlined in this section is both fast and flexible, but it runs into trouble because of its lack of structure. In particular, it fails to clearly distinguish between the ordinary, non-shared object references that constitute a domain's internal state, and the shared object references that are used for cross-domain communication. Nevertheless, this approach is useful to examine, because it illustrates how much protection is possible with the mechanisms provided by the Java language itself. It suggests that the most natural approach to building a protection system in Java is to make good use of the language's inherent protection mechanisms, but to introduce additional structure to fix the problems. The next section presents a system that retains the flavor of the simple approach, but makes a stronger distinction between non-shared and shared objects.

3 The J-Kernel

The J-Kernel is a capability-based system that supports multiple, cooperating protection domains, called tasks, which run inside a single Java virtual machine. Capabilities were chosen because they have several advantages over access control lists: (i) they can be implemented naturally in a safe language, (ii) they can enforce the principle of least privilege more easily, and (iii) by avoiding access list lookups, operations on capabilities can execute quickly.

The primary goals of the J-Kernel are:

- a precise definition of tasks, with a clear distinction between objects local to a task and *capability objects* that can be shared between tasks,
- well defined, flexible communication channels between tasks based on capabilities,
- support for revocation for all capabilities, and
- clean semantics of task termination.

To achieve these goals, we were willing to accept higher cross-task communication overheads when compared to the share anything approach. In order to ensure portability, the J-Kernel is implemented entirely as a Java library and requires no native code or modifications to the virtual machine. To accomplish this, the J-Kernel defines a class loader that examines and in some cases modifies user-submitted bytecode before passing it on to the virtual machine. This class loader also generates bytecode at run-time for stub classes used for cross-task communication. Finally, the J-Kernel's class loader substitutes safe versions for some problematic standard classes. With these implementation techniques, the J-Kernel builds a protection architecture that is radically different from the security manager based protection architecture that is the default model on most Java virtual machines.

Protection in the J-Kernel is based on three core concepts—capabilities, tasks, and cross-task calls:

- *Capabilities* are implemented as objects of the class `Capability` and represent handles onto resources in other tasks. A capability can be revoked at any time by the task that created it. All uses of a revoked capability throw an exception, ensuring the correct propagation of failure.
- *Tasks* are represented by the Java class `Task`. Each task has a namespace that it controls as well as a set of threads. When a task terminates, all of the capabilities that it created are revoked, so that all of its memory may be freed, thus avoiding the task termination problems that plagued the share anything approach.
- *Cross-task calls* are performed by invoking methods of capabilities obtained from other tasks. The J-Kernel's class loader interposes a special calling convention [1] for these calls: arguments and return values are passed by reference if they are also capabilities, but they are passed by copy if they are primitive types or non-capability objects. When an object is copied, these rules are applied recursively to the data in the object's fields, so that a deep copy of the object is made. The effect is that only capabilities can be shared between tasks and references to regular objects are confined to single tasks.

[1] The standard Java calling convention passes primitive data types (int, float, etc.) by copy and object data types by reference.

3.1 J-Kernel Implementation

The J-Kernel's implementation of capabilities and cross-task calls relies heavily on Java's *interface classes*. An interface class defines a set of method signatures without providing their implementation. Other classes that provide corresponding implementations can then be declared to *implement* the interface. Normally interface classes are used to provide a limited form of multiple inheritance (properly called interface inheritance) in that a class can implement multiple interfaces. In addition, Sun's remote method invocation (RMI) specification [19] "pioneered" the use of interfaces as compiler annotations. Instead of using a separate interface definition language (IDL), the RMI specification simply uses interface classes that are flagged to the RMI system in that they extend the class `Remote`. Extending `Remote` has no effect other than directing the RMI system to generate appropriate stubs and marshalling code.

Because of the similarity of the J-Kernel's cross-task calls to remote method invocations, we have integrated much of Sun's RMI specification into the capability interface. The example below shows a simple *remote interface* and a class that implements this remote interface, both written in accordance with Sun's RMI specification.

```
// interface class shared with other tasks
interface ReadFile extends Remote {
  byte readByte() throws RemoteException;
  byte[] readBytes(int nBytes)
    throws RemoteException;
}

// implementation hidden from other tasks
class ReadFileImpl implements ReadFile {
  public byte readByte() {...}
  public byte[] readBytes(int nBytes) {...}
  ...
}
```

To create a capability in the J-Kernel, a task calls the `create` method of the class `Capability`, passing as an argument a target object that implements one or more remote interfaces. The `create` method returns a new capability, which extends the class `Capability` and implements all of the remote interfaces that the target object implements. The capability can then be passed to other tasks, which can cast it to one of its remote interfaces, and invoke the methods this interface declares. In the example below task 1 creates a capability and adds it to the system-wide repository (the repository is a name service allowing tasks to publish capabilities). Task 2 retrieves the capability from the repository, and makes a cross-task invocation on it.

```
// Task 1:
// instantiate new ReadFileImpl object
ReadFileImpl target = new ReadFileImpl();
// create a capability for the new object
Capability c = Capability.create(target);
// add it to repository under some name
Task.getRepository().bind(
    "Task1ReadFile", c);

// Task 2:
// extract capability
Capability c = Task.getRepository()
  .lookup("Task1ReadFile");
// cast it to ReadFile, and invoke remote method
byte b = ((ReadFile) c).readByte();
```

Essentially, a capability object is a wrapper object around the original target object. The code for each method in the wrapper switches to the task that created the capability, makes copies of all non-capability arguments according to the special calling convention, and then invokes the corresponding method in the target object. When the target object's method returns, the wrapper switches back to the caller task, makes a copy of the return value if it is not a capability, and returns.

Local-RMI Stubs The simple looking call to `Capability.create` in fact hides most of the complexity of traditional RPC systems. Internally, `create` automatically generates a stub class at run-time for each target class. This avoids off-line stub generators and IDL files, and it allows the J-Kernel to specialize the stubs to invoke the target methods with minimal overhead. Besides switching tasks, stubs have three roles: copying arguments, supporting revocation, and protecting threads.

By default, the J-Kernel uses Java's built-in serialization features [19] to copy an argument: the J-Kernel serializes an argument into an array of bytes, and then deserializes the byte array to produce a fresh copy of the argument. While this is convenient because many built-in Java classes are serializable, it involves a substantial overhead. Therefore, the J-Kernel also provides a fast copy mechanism, which makes direct copies of objects and their fields without using an intermediate byte array. The fast copy implementation automatically generates specialized copy code for each class that the user declares to be a fast copy class. For cyclic or directed graph data structures, a user can request that the fast copy code use a hash table to track object copying, so that objects in the data structure are not copied more than once (this slows down copying, though, so by default the copy code does not use a hash table).

Each generated stub contains a revoke method that sets the internal pointer to the target object to `null`. Thus all capabilities can be revoked and doing so makes the target object eligible for garbage collection, regardless of how many other tasks hold a reference to the capability. This prevents tasks from holding on to garbage in other tasks.

In order to protect the caller's and callee's threads from each other, the generated stubs provide the illusion of switching threads. Because most virtual machines map Java threads directly onto kernel threads it is not practical to actually switch threads: as shown in the next subsection this would slow down cross-task calls substantially. A fast user-level threads package might solve this problem, but would require modifications to the virtual machine, and would thus limit the J-Kernel's portability. The compromise struck in the current implementation uses a single Java thread for both the caller and callee but prevents direct access to that thread to avoid security problems.

Conceptually, the J-Kernel divides each Java thread into multiple segments, one for each side of a cross-task call. The J-Kernel class loader then hides the system `Thread` class that manipulates Java threads, and interposes its own with an identical interface but an implementation that only acts on the local thread segment. Thread modification methods such as `stop` and `suspend` act on thread segments rather than Java threads, which prevents the caller from modifying the callee's thread segment and vice-versa. This provides the illusion of thread-switching cross-task calls, without the overhead for actually switching threads. The illusion is not totally convincing, however—cross-task calls really do block, so there is no way for the caller to gracefully back out of one if the callee doesn't return.

Class Name Resolvers In the standard Java applet architecture, applets have very little access to Java's class loading facilities. In contrast, J-Kernel tasks are given considerable control over their own class loading. Each task has its own class namespace that maps names to classes. Classes may be local to a task, in which case they are only visible in that task's namespace or they may be shared between multiple tasks, in which case they are visible in many namespaces. A task's namespace is controlled by a user-defined *resolver*, which is queried by the J-Kernel whenever a new class name is encountered. A task can use a resolver to load new bytecode into the system, or it can make use of existing shared classes. After a task has loaded new classes into the system, it can share these classes with other tasks if it wants, by making a `SharedClass` capability available to other tasks [2].

Shared classes are the basis for cross-task communication: tasks must share remote interfaces and fast copy classes to establish common methods and argument types for cross-task calls. Allowing user-defined shared classes makes the cross-task communication architecture extensible; standard Java security architectures only allow pre-defined "system classes" to be shared between tasks, and thus limit the expressiveness of cross-task communication.

[2] Shared classes (and, transitively, the classes that shared classes refer to) are not allowed to have static fields, to prevent sharing of non-capability objects through static fields. In addition, to ensure consistency between domains, two domains that share a class must also share other classes referenced by that class.

Ironically, the J-Kernel needs to *prevent* the sharing of some system classes. For example, the file system and thread classes present security problems. Others contain resources that need to be defined on a per-task basis: the class `System`, for example, contains static fields holding the standard input/output streams. In other words, the "one size fits all" approach to class sharing in most Java security models is simply not adequate, and a more flexible model is essential to make the J-Kernel safe, extensible, and fast.

In general, the J-Kernel tries to minimize the number of system classes visible to tasks. Classes that would normally be loaded as system classes (such as classes containing native code) are usually loaded into a privileged task in the J-Kernel, and are accessed through cross-task communication, rather than through direct calls to system classes. For instance, we have developed a task for file system access that is called using cross-task communication. To keep compatibility with the standard Java file API, we have also written alternate versions of Java's standard file classes, which are just stubs that make the necessary cross-task calls. (This is similar to the interposition proposed by [35]).

The J-Kernel moves functionality out of the system classes and into tasks for the same reasons that micro-kernels move functionality out of the operating system kernel. It makes the system as a whole extensible, i.e., it is easy for any task to provide alternate implementations of most classes that would normally be system classes (such as file, network, and thread classes). It also means that each such service can implement its own security policy. In general, it leads to a cleaner overall system structure, by enforcing a clear separation between different modules. Java libraries installed as system classes often have undocumented and unpredictable dependencies on one another [3]. Richard Rashid warned that the UNIX kernel had "become a 'dumping ground' for every new feature or facility"[30]; it seems that the Java system classes are becoming a similar dumping ground.

3.2 J-Kernel Micro-Benchmarks

To evaluate the performance of the J-Kernel mechanisms we measured a number of micro-benchmarks on the J-Kernel as well as on a number of reference systems. Unless otherwise indicated, all micro-benchmarks were run on 200Mhz Pentium-Pro systems running Windows NT 4.0 and the Java virtual machines used were Microsoft's VM (MS-VM) and Sun's VM with Symantec's JIT compiler (Sun-VM). All numbers are averaged over a large number of iterations.

Null LRMI Table 1 dissects the cost of a null cross-task call (null LRMI) and compares it to the cost of a regular method invocation, which takes a few tens of nanoseconds. The J-Kernel null LRMI takes 60x to 180x longer than

[3] For instance, Microsoft's implementation of java.io.File depends on java.io.DataInputStream, which depends on com.ms.lang.SystemX, which depends on classes in the abstract windowing toolkit. Similarly, java.lang.Object depends transitively on almost every standard library class in the system.

a regular method invocation. With MS-VM, a significant fraction of the cost lies in the interface method invocation necessary to enter the stub. Additional overheads include the synchronization cost when changing thread segments (two lock acquire/release pairs per call) and the overhead of looking up the current thread. Overall, these three operations account for about 70% of the cross-task call on MS-VM and about 80% on Sun-VM. Given that the implementations of the three operations are independent, we expect better performance in a system that includes the best of both VMs.

Operation	MS-VM	Sun-VM
Regular Method invocation	0.04μs	0.03μs
Interface method invocation	0.54μs	0.05μs
Thread info lookup	0.55μs	0.29μs
Acquire/release lock	0.20μs	1.91μs
J-Kernel LRMI	2.22μs	5.41μs

Table 1. Cost of null method invocations

To provide a comparison of the J-Kernel LRMI to traditional OS cross-task calls, Table 2 shows the cost of several forms of local RPC available on NT. *NT-RPC* is the standard, user-level RPC facility. *COM out-of-proc* is the cost of a null interface invocation to a COM component located in a separate process on the same machine. The communication between two fully protected components is at least a factor of 3000 from a regular C++ invocation (shown as *COM in-proc*). For a comparison to the local RPC performance of research operating systems see the Related Work section.

Form of RPC	Time
NT-RPC	109μs
COM out-of-proc	99μs
COM in-proc	0.03μs

Table 2. Local RPC costs using NT mechanisms

Threads Table 3 shows the cost of switching back and forth between two Java threads in MS-VM and Sun-VM. The base cost of two context switches between NT kernel threads (*NT-base*) is 8.6μs, and Java introduces an additional 1-2μs of overhead. This confirms that switching Java threads during cross-task calls would add a significant cost to J-Kernel LRMI.

Argument Copying Table 4 compares the cost of copying arguments during a J-Kernel LRMI using Java serialization and using the J-Kernel's fast-copy

NT-base	MS-VM	Sun-VM
8.6μs	9.8μs	10.2μs

Table 3. Cost of a double thread switch using regular Java threads

mechanism. By making direct copies of the objects and their fields without using an intermediate Java byte-array, the fast-copy mechanism improves the performance of LRMI substantially—more than an order of magnitude for large arguments. The performance difference between the second and third rows (both copy the same number of bytes) is due to the cost of object allocation and invocations of the copying routine for every object.

Number of objects	MS-VM		Sun-VM	
and size	Serializ.	Fast-copy	Serializ.	Fast-Copy
1 x 10 bytes	104μs	4.8μs	331μs	13.7μs
1 x 100 bytes	158μs	7.7μs	509μs	18.5μs
10 x 10 bytes	193μs	23.3μs	521μs	79.3μs
1 x 1000 bytes	633μs	19.2μs	2105μs	66.7μs

Table 4. Cost of Argument Copying during LRMI

In summary, the micro-benchmark results are encouraging in that the cost of a cross-task call is 50x lower in the J-Kernel than in NT. However, the J-Kernel cross-task call still incurs a stiff penalty over a plain method invocation.

Inspection of the critical code paths shows that in the call path itself, the acquisition of a lock and the management of the thread state contribute most to the cost. While aggressive inlining could reduce that by perhaps a factor of 2x, the high cost of lock operations in modern processors will keep the cost significant. The largest cost of most cross-task calls is likely to be in the copying of the arguments. For small objects the allocation (and eventual garbage collection) dominates the cost and appears difficult to optimize by more than a small factor. A more promising avenue would be to optimize the passing of arrays of primitive (i.e., non-pointer) types by simply sharing the array. This clearly changes the semantics, but does not compromise the integrity of the J-Kernel.

4 An Extensible Web and Telephony Server

One of the driving applications for the J-Kernel is an extensible web and telephony server. The goal is to allow users to dynamically extend the functionality of the server by uploading Java programs, called *servlets* [21], that customize the processing of HTTP requests and telephone calls. Applications that integrate the two forms of communication are of particular interest.

The extensible web and telephony server is made possible by the ongoing merger of networking and telephony. The resulting new wave of commodity telephony equipment is designed to be controlled by a standard workstation. Examples are telephone interface cards with 2 to 24 analog telephone lines and the DSPs to handle real-time tasks, and full telephone switches (PBXs) that can be controlled over a LAN. On the software side, standard telephony APIs (e.g., Windows' TAPI or Java's JTAPI) allow programs to control the equipment in a relatively portable way.

The upshot of these developments is that today a single commodity PC can provide integrated communication services that use both the Internet and the telephone system. This section describes a prototype integrated web and telephony server called the J-Server, which uses the J-Kernel to support the extensibility necessary in this setting. The J-Server core manages a hierarchical namespace of resources where each resource can correspond either to a URL or to a telephone number. Users of the system can upload programs (called servlets) to the J-Server and attach each servlet to one or more resources. The servlets can then handle all HTTP requests and telephone calls that are sent to the resources. Furthermore, servlets may communicate with one another, so that one servlet may provide a service to another servlet. For example, a servlet that uses speech recognition to determine the nature of a call might need to dispatch the call to another servlet once it is clear whom the call is for.

In general, our experience with using these servers has shown that they place high demands on the extension mechanism:

- Protection is important because multiple users create their own independent webs.
- Failure isolation is important because webs are continuously expanded and new features must not disrupt old ones.
- Clean servlet termination is essential.
- Extensions must be able to call other extensions, and these cross-task calls should be cheap so that they can occur frequently.
- Resource management and accounting are necessary.

Our J-Server setup uses two off-the-shelf products: Lucent Technologies DEFINITY Enterprise Communications Server and the Dialogic Dialog/4 voice interface. At the core, the DEFINITY server is a Private Branch Exchange (PBX), i.e. a telephone switch, augmented by an Ethernet-based network controller. The network controller in this PBX allows the routing of telephone calls to be monitored and controlled from a PC over the LAN. Lucent provides a Java interface allowing the J-Server to communicate with the PBX using the Java Telephony API (JTAPI) [18].

The Dialog/4 Voice Processing Board, from Dialogic Corporation, provides four half-duplex, analog telephone interfaces and includes a digital signal processor (DSP) to play and record audio, and to detect and transmit DTMF (dual-tone multi-frequency) signals. Using two Dialog/4 boards, the J-Server is able to make eight simultaneous half-duplex voice connections to the telephone network.

4.1 Extensible Server Architecture

Each servlet in the J-Server runs as a separate task to isolate it from the J-Server core as well as from other servlets. This setup allows new servlets to be introduced into the system or crashed ones to be reloaded without disturbing the server operation. The J-Server core is also divided into several tasks. One task dispatches incoming HTTP requests to servlets. Three tasks handle telephony: the first is responsible for communicating with the PBX via JTAPI, the second deals with the Dialogic voice processing boards, and the third is in charge of dispatching telephony events to the appropriate servlets.

During operation, the J-Server performs a large number of cross-task calls to pass events and data around. This is best illustrated by looking at the handling of a telephone call by a simple voice-mail servlet. When a telephone call reaches the PBX, an event is sent to the J-Server via JTAPI and is handled by the task in charge of PBX events. This task then sends the event information to the telephony dispatch task, which in turn passes the information to the appropriate servlet. The servlet instructs the J-Server to route the call to one of the telephone lines connected to the Dialogic card. At this point, the voice processing task begins generating events, starting with a "ringing" event. These events are passed to the dispatching task, and then on to the appropriate servlet. The servlet proceeds to take the telephone off hook, and to start playing audio data. In this application, the data is passed from the servlet task to the dispatch task and then on to the voice task in one large chunk. When the voice channel has finished playing the audio sample, it alerts the servlet, which, in turn, responds by creating an appropriate response object. When the calling party hangs up, the servlet instructs the voice channel to stop recording. The servlet waits for all recorded data and then saves the sample as a Microsoft WAVE file for later playback.

4.2 Performance

We conducted a set of experiments on the J-Server to evaluate the performance of the J-Kernel in an application setting. The experiments were carried out on a 300MHz Pentium II, running Windows NT 4.0. The Java virtual machine used was Microsoft's VM version 1.1, with SDK 2.01.

Two tests measure the costs of cross-task calls and the frequency of such calls in the J-Server. The first test, *Half-Duplex Conversation*, mimics a 40-second telephone conversation between two people, each of whom is simulated by a servlet. The call initiator "speaks" for two seconds, then "listens" for two seconds, then "speaks" again, while the receiver is doing the opposite. "Speaking" involves sending uncompressed audio data in one kilobyte segments (which is equivalent to 1/8sec of conversation) to the voice processing task, where they are presented to the Dialogic driver, which plays the data over the phone line. "Listening" consists of recording data (in 512byte segments), collecting the data to accumulate two seconds worth of audio (16Kbytes) and writing it out to the disk in a standard, uncompressed WAVE file format.

The second test, *VoiceBox* Listing, measures the performance of the HTTP component of J-Server. Using a web browser, a user makes a request to the servlet, which displays the contents of a voice box directory, formatted as an HTML page. Cross-task calls are made from the HTTP dispatching task to the servlet task. In addition, a simple authentication servlet is used to verify the user's password, resulting in another cross-task call.

Table 5 summarizes the results. The *Initiator* and *Responder* columns correspond to two parties involved in *HalfDuplex Conversation*, while the *Listing* column corresponds to the servlet fetching and formatting information about voice messages. The first five rows present the actual measurement values; the remaining rows contain derived information.

	Half-duplex conversation		HTTP
	Initiator	Responder	Listing
Total CPU time	1503 ms	1062 ms	18.5 ms
Total number of cross-task calls	3085	2671	8
Total cross-task data transfer	962304 bytes	984068 bytes	3596 bytes
Total cross-task call overhead	9.37 ms	8.71 ms	0.037 ms
Total cross-task data copy time	2.91 ms	2.58 ms	0.01 ms
Average CPU time/cross-task call	3.0 μs	3.3 μs	4.6 μs
Average bytes/call	312 bytes	368 bytes	450 bytes
Average copy overhead/call	31 %	30 %	27 %
Average copy bandwidth	331 MB/s	381 MB/s	360 MB/s
Relative cross-task call overhead	0.62 %	0.82 %	0.20 %

Table 5. Selected performance measurements of *HalfDuplex Conversation* and *VoiceBox Listing*.

In all cases the overheads of crossing tasks (which include the cost of copying arguments and then return values) are below 1% of the total consumed CPU time. On the average, crossing tasks costs between 3μs-4.6μs and an average call transfers between 312 (*Initiator*) to 450 (*Listing*) bytes. The cost of copying data accounts for 27%-31% of an average cross-task call. This suggests that overheads of crossing tasks in real applications when no data transfer is involved are between 2.1μs-3.3μs. This is in contrast to the cost of a null cross-task call, measured in an isolated tight loop: 1.35μs.

The achieved bandwidth of copying data across tasks is sufficient for this application. About 85% of the data transferred across tasks in the two experiments is stored in byte arrays of 0.5Kbytes and 1Kbytes; the remaining data are either references to capabilities, primitive data types or data inside objects of the type `java.lang.String`. The average copying bandwidth is 330-380Mbytes/s, which is over 50% of the peak data copying bandwidth of 630Mbytes/s achieved in Java with a tight loop of calls to `System.arraycopy()`.

With respect to scalability, our current hardware configuration supports at most eight simultaneous half-duplex connections. Every connection results in an additional 1.7%-2.3% increase in the CPU load. From this we estimate that, given enough physical lines, our system could support about 50 simultaneous half-duplex connections before reaching full CPU utilization. Since the J-Server demands modest amounts of physical memory, permanent storage and network resources, CPU is the bottleneck from the perspective of scaling the system, so the range of 50 simultaneous half-duplex connections is very likely to be achievable in practice.

While analyzing the performance, it is important to note that Java introduces non-obvious overheads. For instance, in the *Listing* experiment, a large part of the 18.5ms of CPU time can be accounted for as follows. About seven milliseconds are spent in the network and file system Java classes and native protocol stacks. Roughly eight milliseconds are spent performing string concatenations in one particular part of the *VoiceBox Listing* servlet (Java string processing is very slow under MS JVM). These overheads, which are introduced by Java and not easy to avoid, dwarf the relative costs of crossing task boundaries and lead to conclusions that may be not true if the JVM performance improves dramatically.

One conclusion, based on the current performance numbers, is that in large applications similar to J-Server, the task boundaries crossing overheads are very small relative to the total execution time and further optimizing them will bring barely noticeable performance improvements. Second, in applications where crossing tasks is frequent and optimizing cross-task calls actually matters, on the average 30% of task crossing overheads can be removed by wrapping data structures and arrays in capabilities. This avoids copying data between tasks but comes at the expense of increased programming effort. It is important to stress that the conclusions are drawn from observing several experiments based on a single, although complex and realistic, application. More study is needed to fully understand application behavior on top of Java in general and the J-Kernel in particular.

5 Related Work

The Alta and GVM systems [1] developed at the University of Utah are closely related to the J-Kernel. The motivation is to implement a process model in Java, which provides memory protection, control over scheduling, and resource management. Both systems are implemented by modifying a JVM (the free Kaffe JVM is used). In GVM each Java process receives separate threads, heap, and classes. Sharing is only allowed between a process and a special system heap. This invariant is enforced using a write barrier. Each heap is garbage collected separately and a simple reference counting mechanism is used for cross-heap references. Alta implements the nested process model used in the Fluke microkernel [10]. Child processes of the same parent process can share data structures directly: the resources used are attributed to the parent. While the inter-process communication costs of Alta and GVM are higher than in the J-Kernel (see [1]), the custom JVM modifications allow both GVM and Alta to take control of thread scheduling and to be more precise in resource management.

Several major vendors have proposed extensions to the basic Java sandbox security model for applets [20, 25, 29]. For instance, Sun's JDK 1.1 added a notion of authentication, based on code signing, while the JDK 1.2 adds a richer structure for authorization, including classes that represent permissions and methods that perform access control checks based on stack introspection [12]. JDK 1.2 "protection domains" are implicitly created based on the origin of the code, and on its signature. This definition of a protection domain is closer to a user in Unix, while the J-Kernel's task is more like a *process* in Unix. Balfanz et al. [2] define an extension to the JDK which associates domains with users running particular code, so that a domain becomes more like a process. However, if domains are able to share objects directly, revocation, resource management, and domain termination still need to be addressed in the JDK.

JDK 1.2 system classes are still lumped into a monolithic "system domain", but a new classpath facilitates loading local applications with class loaders rather than as system classes. However, only system classes may be shared between domains that have different class loaders, which limits the expressiveness of communication between domains. In contrast, the J-Kernel allows tasks to share classes without requiring these tasks to use the same class loader. In the future work section, Gong et al. [12] mentions separating current system classes (such as file classes) into separate tasks, in accordance with the principle of least privilege. The J-Kernel already moves facilities for files and networking out of the system classes and into separate tasks.

A number of related safe-language systems are based on the idea of using object references as capabilities. Wallach et. al. [35] describe three models of Java security: type hiding (making use of dynamic class loading to control a domain's namespace), stack introspection, and capabilities. They recommended a mix of these three techniques. The E language from Electric Communities [8] is an extension of Java targeted towards distributed systems. E's security architecture is capability based; programmers are encouraged to use object references as the fundamental building block for protection. Odyssey [11] is a system that supports mobile agents written in Java; agents may share Java objects directly. Hagimont et al. [14] describe a system to support capabilities defined with special IDL files. All three of these systems allow non-capability objects to be passed directly between domains, and generally correspond to the share anything approach described in Section 2. They do not address the issues of revocation, domain termination, thread protection, or resource accounting.

The SPIN project [3] allows safe Modula-3 code to be downloaded into the operating system kernel to extend the kernel's functionality. SPIN has a particularly nice model of dynamic linking to control the namespace of different extensions. Since it uses Modula-3 pointers directly as capabilities, the limitations of the share anything approach apply to it.

Several recent software-based protection techniques do not rely on a particular high level language like Java or Modula-3. Typed assembly language [26] pushes type safety down to the assembly language level, so that code written at the assembly language level can be statically type checked and verified as safe.

Software fault isolation [34] inserts run-time “sandboxing” checks into binary executables to restrict the range of memory that is accessible to the code. With suitable optimizations, sandboxed code can run nearly as fast as the original binary on RISC architectures. However, it is not clear how to extend optimized sandboxing techniques to CISC architectures, and sandboxing cannot enforce protection at as fine a granularity as a type system. Proof carrying code [27, 28] generalizes many different approaches to software protection—arbitrary binary code can be executed as long as it comes with a proof that it is safe. While this can potentially lead to safety without overhead, generating the proofs for a language as complex as Java is still a research topic.

The J-Kernel enforces a structure that is similar to traditional capability systems [22, 23]. Both the J-Kernel and traditional capability systems are founded on the notion of unforgeable capabilities. In both, capabilities name objects in a context-independent manner, so that capabilities can be passed from one domain to another. The main difference is that traditional capability systems used virtual memory or specialized hardware support to implement capabilities, while the J-Kernel uses language safety. The use of virtual memory or specialized hardware led either to slow cross-domain calls, to high hardware costs, or to portability limitations. Using Java as the basis for the J-Kernel simplifies many of the issues that plagued traditional capability systems. First, unlike systems based on capability lists, the J-Kernel can store capabilities in data structures, because capabilities are implemented as Java objects. Second, rights amplification [22] is implicit in the object-oriented nature of Java: invocations are made on methods, rather than functions, and methods automatically acquire rights to their *self* parameter. In addition, selective class sharing can be used to amplify other parameters. Although many capability systems did not support revocation, the idea of using indirection to implement revocation goes back to Redell [31]. The problems with resource accounting were also on the minds of implementers of capability systems—Wulf et. al. [36] point out that “No one ‘owns’ an object in the Hydra scheme of things; thus it’s very hard to know to whom the cost of maintaining it should be charged”.

Single-address operating systems, like Opal [6] and Mungi [17], remove the address space borders, allowing for cheaper and easy sharing of data between processes. Opal and Mungi were implemented on architectures offering large address spaces (64-bit) and used password capabilities as the protection mechanism. Password capabilities are protected from forgery by a combination of encryption and sparsity.

Several research operating systems support very fast inter-process communication. Recent projects, like L4, Exokernel, and Eros, provide fine-tuned implementations of selected IPC mechanisms, yielding an order of magnitude improvement over traditional operating systems. The systems are carefully tuned and aggressively exploit features of the underlying hardware.

The L4 μ-kernel [15] rigorously aims for minimality and is designed from scratch, unlike first-generation μ-kernels, which evolved from monolithic OS kernels. The system was successful at dispelling some common misconceptions about μ-kernel performance limitations. Exokernel [9] shares L4's goal of being an ultra-fast "minimalist" kernel, but is also concerned with untrusted loadable modules (similar to the SPIN project). Untrusted code is given efficient control over hardware resources by separating management from protection. The focus of the EROS [33] project is to support orthogonal persistence and real-time computations. Despite quite different objectives, all three systems manage to provide very fast implementations of IPC with comparable performance, as shown in Table 6. A short explanation of the 'operation' column is needed. Round-trip IPC is the time taken for a call transferring one byte from one process to another and returning to the caller; Exokernel's protected control transfer installs the callee's processor context and starts execution at a specified location in the callee.

System	Operation	Platform	Time
L4	Round-trip IPC	P5-133	1.82μs
Exokernel	Protected control transfer (r/t)	DEC-5000	2.40μs
Eros	Round-trip IPC	P5-120	4.90μs
J-Kernel	Method invocation with 3 args	P5-133	3.77μs

Table 6. Comparison with selected kernels.

The results are contrasted with a 3-argument method invocation in the J-Kernel. The J-Kernel's performance is comparable with the three very fast systems. It is important to note that L4, Exokernel and Eros are implemented as a mix of C and assembly language code, while J-Kernel consists of Java classes without native code support. Improved implementations of JVMs and JITs are likely to enhance the performance of the J-Kernel.

6 Conclusion

The J-Kernel project explores the use of safe language technology to construct robust protection domains. The advantages of using language-based protection are portability and good cross-domain performance. The most straightforward implementation of protection in a safe language environment is to use object references directly as capabilities. However, problems of revocation, domain termination, thread protection, and resource accounting arise when non-shared object references are not clearly distinguished from shared capabilities. We argue that a more structured approach is needed to solve these problems: only capabilities can be shared, and non-capability objects are confined to single domains.

We developed the J-Kernel system, which demonstrates how the issues of object sharing, class sharing, thread protection, and resource management can be addressed. As far as we know, the J-Kernel is the first Java-based system that integrates solutions to these issues into a single, coherent protection system. Our experience using the J-Kernel to extend the Microsoft IIS web server and to implement an extensible web and telephony server leads us to believe that a safe language system can achieve both robustness and high performance.

Because of its portability and flexibility, language-based protection is a natural choice for a variety of extensible applications and component-based systems. From a performance point of view, safe language techniques are competitive with fast microkernel systems, but do not yet achieve their promise of making cross-domain calls as cheap as function calls. Implementing a stronger model of protection than the straightforward share anything approach leads to thread management costs and copying costs, which increase the overhead to much more than a function call. Fortunately, there clearly is room for improvement. We found that many small operations in Java, such as allocating an object, invoking an interface method, and manipulating a lock were slower than necessary on current virtual machines. Java just-in-time compiler technology is still evolving. We expect that as virtual machine performance improves, the J-Kernel's cross-domain performance will also improve. In the meantime, we will continue to explore optimizations possible on top of current off-the-shelf virtual machines, as well as to examine the performance benefits that customizing the virtual machine could bring.

Acknowledgments Portions of this paper were originally published by the USENIX Association in *Proceedings of the USENIX 1998 Annual Technical Conference*, New Orleans, Louisiana, June 15-19, 1998.

The authors would like to thank Greg Morrisett, Fred Schneider, Fred Smith, Lidong Zhou, and the anonymous reviewers for their comments and suggestions. This research is funded by DARPA ITO Contract ONR-N00014-95-1-0977, NSF CAREER Award CCR-9702755, a Sloan Foundation fellowship, and Intel Corp. hardware donations. Chi-Chao Chang is supported in part by a doctoral fellowship (200812/94-7) from CNPq/Brazil.

References

1. G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, J. Lepreau. *Java Operating Systems: Design and Implementation.* Technical Report UUCS-98-015, Department of Computer Science, University of Utah, August, 1998.
2. D. Balfanz, and Gong, L. *Experience with Secure Multi-Processing in Java.* Technical Report 560-97, Department of Computer Science, Princeton University, September, 1997.
3. B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. *Extensibility, Safety and Performance in the SPIN Operating System.* 15th ACM Symposium on Operating Systems Principles, p.267-284, Copper Mountain, CO, December 1995.
4. B. Bershad, T. Anderson, E. Lazowska, and H. Levy. *Lightweight Remote Procedure Call.* 12th ACM Symposium on Operating Systems Principles, p. 102-113, Lichtfield Park, AZ, December 1989.
5. R. S. Boyer, and Y. Yu. *Automated proofs of object code for a widely used microprocessor.* J. ACM 43(1), p. 166-192, January 1996.
6. J. Chase, H. Levy, E. Lazowska, and M. Baker-Harvey. *Lightweight Shared Objects in a 64-Bit Operating System.* ACM Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), October 1992.

7. G. Czajkowski and T. von Eicken. *JRes: A Resource Accounting Interface for Java.* To appear in proceedings of the 1998 Conference on Object-Oriented Programming Languages, Systems, and Applications.
8. Electric Communities. *The E White Paper.* http://www.communities.com/products/tools/e.
9. R. Engler, M. Kaashoek, and J. James O'Toole. *Exokernel: An Operating System Architecture for Application-Level Resource Management.* 15th ACM Symposium on Operating Systems Principles, p. 251266, Copper Mountain, CO, December 1995.
10. B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. *The Fluke OSKit: A substrate for OS and language research.* In Proc. Of the 16th SOSP, pp. 38-51, St. Malo, France, October 1997.
11. General Magic. *Odyssey.* http://www.genmagic.com/agents.
12. L. Gong, and Schemers, R. *Implementing Protection Domains in the Java Development Kit 1.2.* Internet Society Symposium on Network and Distributed System Security, San Diego, CA, March 1998.
13. J. Gosling, B. Joy, and G. Steele. *The Java language specification.* Addison-Wesley, 1996.
14. D. Hagimont, and L. Ismail. *A Protection Scheme for Mobile Agents on Java.* 3rd Annual ACM/IEEE Int'l Conference on Mobile Computing and Networking, Budapest, Hungary, September 2630, 1997.
15. H. Haertig, et. al. *The Performance of μ-Kernel-Based Systems.* 16th ACM Symposium on Operating Systems Principles, p. 6677, Saint-Malo, France, October 1997.
16. C. Hawblitzel, C. C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. *Implementing Multiple Protection Domains in Java.* 1998 USENIX Annual Technical Conference, p. 259-270, New Orleans, LA, June 1998.
17. G. Heiser, et. al. *Implementation and Performance of the Mungi Single-Address-Space Operating System.* Technical Report UNSW-CSE-TR-9704, Univeristy of New South Wales, Sydney, Australia, June 1997.
18. JavaSoft. *Java Telephony API.* http://java.sun.com/products/jtapi/index.html.
19. JavaSoft. *Remote Method Invocation Specification.* http://java.sun.com.
20. JavaSoft. *New Security Model for JDK1.2.* http://java.sun.com
21. JavaSoft. *Java Servlet API.* http://java.sun.com.
22. A. K. Jones and W. A. Wulf. *Towards the Design of Secure Systems.* Software Practice and Experience, Volume 5, Number 4, p. 321336, 1975.
23. H. M. Levy. *Capability-Based Computer Systems.* Digital Press, Bedford, Massachusetts, 1984.
24. J. Liedtke, et. al. *Achieved IPC Performance.* 6th Workshop on Hot Topics in Operating Systems, Chatham, MA, May.
25. Microsoft Corporation. *Microsoft Security Management Architecture White Paper.* http://www.microsoft.com/ie/ security.
26. G. Morrisett, D. Walker, K. Crary, and N. Glew. *From System F to Typed Assembly Language.* 25th ACM Symposium on Principles of Programming Languages. San Diego, CA, January 1998.
27. G. Necula and P. Lee. *Safe Kernel Extensions Without Run-Time Checking.* 2nd USENIX Symposium on Operating Systems Design and Implementation, p. 229243, Seattle, WA, October 1996.
28. G. Necula. *Proof-carrying code.* 24th ACM Symposium on Principles of Programming Languages, p. 106119, Paris, 1997.
29. Netscape Corporation. *Java Capabilities API.* http://www.netscape.com.

30. Rashid, R. *Threads of a New System.* Unix Review, p. 3749, August 1986.
31. D. D. Redell. *Naming and Protection in Extendible Operating Systems.* Technical Report 140, Project MAC, MIT 1974.
32. Z. Shao. *Typed Common Intermediate Format.* 1997 USENIX Conference on Domain-Specific Languages, Santa Barbara, California, October 1997.
33. J. S. Shapiro, D. J. Farber, and J. M. Smith. *The Measured Performance of a Fast Local IPC.* 5th Int'l Workshop on Object-Orientation in Operating Systems, Seattle, WA. 1996
34. R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. *Efficient Software-Based Fault Isolation.* 14th ACM Symposium on Operating Systems Principles, p. 203216, Asheville, NC, December 1993.
35. D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. *Extensible Security Architectures for Java.* 16th ACM Symposium on Operating Systems Principles, p. 116128, Saint-Malo, France, October 1997.
36. W. A. Wulf, R. Levin, and S.P. Harbison. *Hydra/C. mmp: An Experimental Computer System*, McGraw-Hill, New York, NY, 1981.