Incommunicado: Efficient Communication for Isolates

Krzysztof Palacz   Grzegorz Czajkowski[†]   Laurent Daynès[†]   Jan Vitek

S^3Lab, Dept of Computer Sciences, Purdue University, West Lafayette, IN, USA
[†] Sun Microsystems Laboratories, 2600 Casey Avenue, Mountain View, CA 94043, USA

ABSTRACT

Executing computations in a single instance of safe language virtual machine can improve performance and overall platform scalability. It also poses various challenges. One of them is providing a fast inter-application communication mechanism. In addition to being efficient, such a mechanism should not violate any functional and non-functional properties of its environment, and should also support enforcement of application-specific security policies. This paper explores the design and implementation of a communication substrate for applications executing within a single Java^{TM} virtual machine modified to enable safe and interference-free execution of isolated computations. Designing an efficient extension that does not break isolation properties and at the same time pragmatically offers an intuitive API has proven non-trivial. This paper demonstrates a set of techniques that lead to at least an eight-fold performance improvement over the in-process inter-application communication using standard mechanisms offered by the Java^{TM} platform.

Keywords

Application isolation, inter-application communication.

1. INTRODUCTION

Running multiple computations in a single instance of the Java virtual machine (JVM^{TM}), for instance executing many servlets in a Web server, has the potential for improving overall system performance and scalability by sharing some of the virtual machine's internal data structures. Such collocation also creates opportunities for better management of resources and elegant control policies at the language level. The main difficulty in delivering collocation is that the platform must provide strong isolation guarantees to ensure that if one computation fails or misbehaves, other computations will not be disrupted or prevented from performing their assigned tasks.

OOPSLA'02, November 4-8, 2002, Seattle, Washington, USA.

The application isolation API defines the basic functionality that can be used to create and manage mutually disjoint computations within the JVM. The key abstraction proposed is that of an *isolate*[1]. Isolates are instances of the `Isolate` class, which provides the means to start and stop an isolated computation.

The goal of our project, code-named *Incommunicado* due to the conflicting needs of keeping applications disjoint while allowing them to interact, is to explore the design space of communication infrastructures for isolates. The presented design is by no means definitive, nor are we in a position to advocate its inclusion in the isolation API. Rather, we seek to gain experience with the costs and benefits of a particular scheme as well as to provide a flexible and efficient platform for further experimentation.

Designing a communication substrate for isolates is challenging for several reasons. New communication mechanisms cannot interfere with other features offered by the underlying language or by its particular implementation. This item is particularly important: any new feature may have subtle interactions with, for instance, the automatic memory manager, which in turn may impact the safety of the language. Any communication mechanism should be general enough to accommodate the many different application requirements, such as different security policies [11], and resource limits [3, 8]. Yet it must remain efficient, so that the benefits of collocation are not drowned by the communication costs. In this respect, it is essential to use a high-performance virtual machine for experimentation. Using low-quality virtual machine implementations, or virtual machines without dynamic compilers, may skew the picture of the relative costs. Implementing the mechanism in a modern, fast virtual machine is much more time consuming, but leads to performance answers meaningful for practical use. Similarly, bytecode editing approaches are not only plagued by performance problems, but typically must prohibit the use of certain languages features [13].

Another important guiding principle for our implementation is to *pay as you go.* In other words, applications that do not communicate should not suffer any slowdown due to the

[1] The application isolation API, currently under review as JSR 121 [14], has not been finalized as of this writing. The name *isolate* was chosen in order to avoid further overloading of terms such as *task*, *process*, *domain*, etc.

presence of the new mechanism. This principle is key for practical acceptance.

Incommunicado is a new communication substrate for isolates that has been designed to provide a minimal interface for isolate communication and was implemented in the Multitasking Virtual Machine (or MVM) [7]. MVM exhibits many features we believe will be present in future virtual machines. In particular, it is a single-process, high-performance, full-featured virtual machine hosting multiple tasks in an interference-free way, with clean application termination and resource reclamation facilities. MVM has been designed to demonstrate that multitasking in a safe language can be practical and efficient.

The design of Incommunicado can be characterized by:

- **Simplicity** – Incommunicado is inspired by the Java^{TM} remote method invocation API (RMI), a model that is already familiar to programmers [9].
- **Efficiency** – communication costs in our system are between 8 and 70 times smaller than when locally using RMI. Thus we feel justified in advocating the use of the substrate for performance critical applications.
- **Security** – policy-neutral hooks are provided for implementing application-specific policies. The policies can be specified simply and run efficiently.
- **Non-intrusiveness** – the functional and non-functional properties of the underlying virtual machine were preserved. In particular, we were careful to preserve isolation and termination.

This paper shows how to use the new facilities and details the cross-isolate method invocation package which is the centerpiece of our implementation. The main contributions of this work are a description and performance evaluation of an isolate communication mechanism that addresses the above requirements while simplifying program development.[2]

2. APPLICATION ISOLATION API

The application isolation API provides the means of creating and managing isolated computations (isolates), written in the Java programming language. An isolate, constructed as an instance of the `Isolate` class, encapsulates an application or a component. The goal of the isolate API, and the main difference with servlets and applets, is that isolates guarantee strict isolation between programs. Isolates have disjoint object graphs, sharing objects is forbidden, and each isolate has its own version of a static state of each class it uses. This form of isolation guards application against various form of interference. No special coding conventions need to be followed within an isolate, nor is there a need for recompilation or any other modification to the bytecode. From a program's point of view, starting an isolate is equivalent to starting a new JVM and gives the programs the same rights: applications executed as isolates have full access to all features of the JDK^{TM} and to all constructs of the Java programming language, controlled by standard permissions. From an implementation point of view, running multiple isolates in the same virtual machine enables sharing of internal virtual machine (VM) data structures, bytecode and in some cases compiled code. No particular techniques are prescribed to realize isolates, and implementation strategies can range from running the JVM in a separate process for each isolate to executing all applications within a single multi-tasking JVM in a single process.

The isolate API can be used to start new applications as isolates and to manage their life-cycle. For instance, a Web server can choose to start each servlet as an isolate, while servlets themselves can be oblivious of the fact that they are run as isolates.

2.1 The Isolate API

The `Isolate` class provides a simple interface. Isolates are created by specifying a class name and an array of string arguments:

```
Isolate isl = new Isolate("MyClass", args);
```

The only requirement is that the specified class must have a `main()` method just like a Java application executed from the command line. A newly created isolate is inactive, its creator must call `start(Link[])` to inject a new thread into the isolate with an array of communication links to other isolates.

The `Isolate` class provides methods to terminate the execution of isolates, `exit()` and `halt()`, the former is equivalent to termination of the VM with `Runtime.exit()`, while the latter is equivalent to `Runtime.halt()` which performs a forced shutdown without finalization. Unlike the deprecated `stop` method of `java.lang.Thread`, isolate termination is guaranteed to leave the virtual machine and JDK code in a consistent state. Thus, isolate-based applications are better suited to interruptible tasks than for instance applets or servlets.

2.2 The Link API

Links, which are part of the Isolate API, provide a low-level communication layer designed for high bandwidth communication of basic data types (byte arrays, byte buffers, serialized objects, sockets, and strings). Communication between isolates is done through instances of subclasses of the abstract `Link` class. Links are one- or two-way communication channels between a pair of isolates that transport instances of the class `LinkMessage`. The simplest case of a send-receive sequence over links is coded as follows:

```
// sender isolate
LinkMessage message;
A data = new A();
message = LinkMessage.newSerializableMessage(data);
link.send(message);
...
```

[2] The interface presented in this paper is not a part of the JSR 121 API.

```
// receiver isolate
LinkMessage message = link.receive();
A data = (A) message.getSerializable();
```

Links are created by invoking the static method `newInstance` with a pair of isolates as arguments. Thus the following code snippet creates a one way connection between the current isolate and a newly created isolate:

```
Link lnk = Link.newInstance(Isolate.currentIsolate(),
                            new Isolate(aclass, args));
```

Note that both end-points of a link must exist[3] before creating the link. This causes a slight difficulty for setting up the initial communication topology. Passing an array of links to the `start` method solves this problem. Thus, in the above example the isolate `isl` can be bootstrapped by calling

```
isl.start(new Link[]{link});
```

Once communication has been set up in this fashion, changes in the interconnection topology can be effected by exchanging links (in a message over an existing link). For completeness, we mention the existence of the `EventLink` class, which provides a channel for receiving notification of isolate life-cycle events (currently three events types are supported: `starting`, `stopping`, `terminated`).

2.3 The Isolate Security Model

As mentioned above isolates provide protection against unintentional sharing, which has been the cause of numerous security breaches (see for instance [24]). The communication API does not require an isolate to accept incoming message (receive operations are explicit). Such provisions are needed to prevent certain kinds of denial of service attacks. The remaining forms of inter-isolate interference are related to uncontrolled use of computational resources, such as CPU and heap memory. The API provides a `IsolatePermission` class that extends the `BasicPermission` class of the Java platform security infrastructure. It controls the creation and stopping of isolates, inter-isolate communication, listing of all isolates, and retrieving an isolate's context.

3. ISOLATE COMMUNICATION WITH XIMI

Incommunicado offers a high-level inter-isolate communication substrate called XIMI (for *C*ross-*I*solate *M*ethod *I*nvocation). Initially our goal with XIMI was to provide a simple and flexible programming model for inter-isolate communication. We chose to model XIMI on RMI, a well known component of the Java platform. The version of XIMI presented here is significantly different from our earlier design. When we started working on XIMI (summer 2001), the Isolate API did not specify how isolates were to communicate. This has since then been addressed by the Link API. Another motivation for revising our design was that our experience with the XIMI programming model suggested that compatibility with RMI is difficult to achieve and negates some of the advantages of Isolates. This section introduces the revised XIMI programming model. Implementation issues will be discussed in Section 4. We start by contrasting XIMI with RMI.

3.1 Why not RMI?

The abstraction of remote procedure call (RPC) has proven to be versatile [4], and has been adopted for a variety of software and hardware platforms. Communication mechanisms inspired by RPC but customized for a particular environment, such as RMI [9], have emerged. Their existence provides a convenient way for programmers to utilize network capabilities via an API in the spirit of the programming language at hand.

While remote method invocation is syntactically identical to local method invocation, there are significant semantic differences. Remote objects can only be manipulated using references of the interface type `java.rmi.Remote` or any other interface that extends it. Arguments to remote method invocations as well as their return values are passed by deep copy, following the semantics of serialization. Remote objects are exchanged by remote references, and stubs are created as replacement for remote objects to forward invocations. Beneath this high-level interface lie three layers of implementation:

- *stub layer*: provides (compile-time) automatically generated implementations of sub-interfaces of `Remote`, so-called stubs. These stubs forward invocations to the actual, programmer-supplied implementations of these sub-interfaces using the transport layer.
- *remote reference layer*: is responsible for determining the identity of the remote object, whether the remote object is replicated or not, and whether the remote object is currently instantiated or has to be instantiated.
- *transport layer*: is responsible for connection management, encoding and dispatching invocations over the wire.

RMI is a general purpose protocol for distributed communication across administrative domains. Thus, with RMI, Java virtual machines with potentially different internal data format, object layouts, and class representations are able to exchange data. In the case of Isolate communication much of this generality is merely overhead.

For isolates collocated within the same JVM several of these differences disappear. For instance, data formats and object layouts are identical on both communicating parties. Furthermore, network errors need not be taken into account, and machine failures are likely to be simultaneously fatal for both sides. Thus, there is little motivation for forcing programmers to catch errors that will not occur.

For this reason we have chosen to design XIMI for speed rather than versatility, with the understanding that applications requiring a more expressive protocol may have to fall back on RMI. XIMI provides an application layer interface comparable to RMI's application layer. The semantics

[3] By "exist" we mean that the isolates have been created, but they need not have been started.

of isolate communication follows the call semantics of RMI but with some objects passed by copy and other by cross-isolate references. APIs providing access to the lower layers of RMI are not supported. For example, XIMI does not have equivalents of classes and methods providing programmatic access to the transport layer of RMI. These classes and methods were omitted because their functionality (such as setting up and managing connections or monitoring their "liveness") is either not applicable or performed differently.

3.2 The XIMI Communication Substrate

Incommunicado provides a simple interface to inter-isolate communication. The Isolate API has been modified (i) to add a new method to the `Isolate` class (ii) to define two cross-isolate objects called `Portal` and `DeferrablePortal`, and (iii) to add a new security manager class called `IsolateSecurityManager`. The new interface is given in Figure 1.

`Isolate(String, String[], String)` constructs an isolate; the first argument is the name of the main isolate class and the last argument is the name of a subclass of `IsolateSecurityManager` (or `null`). The only constraint is that the main class must have a static `main(String[])`.

Communication between isolates is done through *portals*. A portal, an instance of a non-public class extending the `Portal` abstract class, is at the receiving end of a connection. To communicate, a server isolate must create a portal object and send that portal through an existing communication channel (either as a message over a link or as an argument to another portal). Each portal has a *target* object and one or more external `stub` objects. The portal and target objects 'live' within the server isolate, while the stubs are located in client isolates. Cross-isolate calls have semantics similar to RMI in that portal objects are passed by reference (involving the creation of stubs), while all other objects are always copied maintaining the semantics of serialization, even though the implementation avoids the overhead of actual serialization. Returns are treated in a similar fashion. If a method called through a portal throws an exception, the exception will be serialized and returned to the calling isolate.

The semantics of cross-isolate method invocation are that the caller will always block. On the callee side, the semantics depend on how the portal was created. The static method `Portal.newPortal()` creates a plain portal, while the method `Portal.newDeferablePortal` creates a deferable portal. Both methods take an interface, a target object and boolean. They differ in their behavior with respect to external calls. A plain portal will always forward calls to the target object, thus creating a new thread within the target isolate to handle the external call (see Section 4.4 for implementation details)[4]. A deferrable portal defers the execution until an explicit `accept()` call from within the isolate. Thus the call is handled by an existing thread within the isolate. The `accept` method is blocking thus if there is no pending call on the portal, the current thread will wait until one occurs. If a call is issued on an isolate with several threads blocked on the particular portal, one of these will be selected randomly.

Each portal has an *exported interface* which must be an interface implemented by the portal's target. Unlike RMI which requires that the exported interface extend `Remote`, any interface may be chosen at portal creation time. This facilitates inter-isolate communication by allowing any object implementing the interface to be used as the target of a portal. Stub objects created from a given portal have a reference to the portal's target and forward invocations. All portals support methods to get and set the target object, as well as a `close` method which closes a portal. Pending calls are allowed to complete but no new calls will be processed. A portal can also be copyable, meaning that isolates holding one of the portal's stubs may send that stub to another isolate. If the portal is not copyable, then its stubs will not be serialized. A portal can thus be associated with multiple stub objects.

```
final public class Isolate {
    public Isolate(String classname,
                   String[] args,
                   IsolateSecurityManager sm);
    public void start(Object[] portals);
    static public Object[] getPortals();
}


public abstract class Portal {
    static public Portal newPortal(Class iface,
                                   Object target,
                                   boolean copyable);
    static public Portal
        newDeferrablePortal(Class iface,
                                   Object o,
                                   boolean copyable);
    void close();
    void setTarget(Object tgt);
    Object getTarget();
    Class getExportedInterface();
    void accept() throws InterruptedException;
}


public abstract class IsolateSecurityManager {
    void checkInvokeFromIsolate(Isolate src,
                                Method m)
        throws AccessControlException;
    void checkClassDefinition(Isolate src,
                              String classname)
        throws AccessControlException;
    final public Isolate getParent(Isolate src);
    final public Isolate getCurrent();
}
```

Figure 1: Incommunicado interfaces. In the case of Isolate, we only present new methods. Implementations of Portal and DeferrablePortal are private.

[4] The portal interface does not mandate creation of threads *per se*, a thread pool could as well be used by an implementation of XIMI.

```
interface Map {
    Object put(Object name, Object obj);
    Object get(Object name);
    ...

interface Converter {
    Printable prepare(Document doc);
    ...

class ConverterImpl
    implements Converter {
    ...
```

```
class PrintServer {
    static public void main(String[] args) {
        Map nameSrv = (Map) Isolate.getPortals()[0];
        Portal conv = Portal.newPortal(Converter.class,
                                       new ConverterImpl(),
                                       true);
        nameServ.put("converter", conv);
        ...

class App {
        ...
        Document doc = new DocImpl();
        Converter conv = (Converter) nameSrv.get("converter");
        Printable file = conv.prepare(doc);
```

Figure 2: An example of inter-isolate communication. PrintServer registers a Converter with the name server. App, running in another isolate, looks up the converter and invokes prepare() in the PrintServer.

Isolates have two methods to bootstrap communication. The `start(Object[] portal)` method is called by the isolate's creator to inject a number of stubs into a newly created isolate (the semantics of `start` are identical to that of a cross-isolate call). Then from within an isolate, `getPortals()` can be used to obtain all portals. The array of objects returned may contain remote stubs as well as plain objects. A name server object can simply be passed as an argument as shown in Figure 2. Thus XIMI differs from RMI in that the `java.rmi.Naming` functionality is not required.

Implementations of the `IsolateSecurityManager` class must provide the following two methods, `checkInvokeFromIsolate` and `checkDefineClass`, to respectively check that a particular invocation is legitimate and that the target isolate is allowed to load a class while unpacking a message received from another isolate. The class further provides two methods, `getCurrent` and `getParent`, to respectively get the isolate that is the target of the operation, as well as its creator.

3.2.1 *Example: Servers*

Figure 2 illustrates isolate communication with one isolate, running the `PrintServer` class, providing a document conversion service, and a client running `App`. The two isolates are connected by a name server, an object implementing the standard `Map` interface. The name server is a stub for an object living in yet another isolate. The class `PrintServer` is thus able to export a conversion service from its `main` method. When it calls the name server's `put` method with a string and the converter portal, the string is passed by copy while the portal is converted to a stub. The portal was created in copy mode since the name server must be able to forward stubs to isolates requesting them. Without this, any attempt to hand out stubs would fail. The class `App` of Figure 2 is an application that uses the name server to get a cross-isolate reference to a converter. The variable `conv` is actually a copy of the stub stored in the name server.

3.2.2 *Example: Futures*

Another use case for portals is to combine them with *futures* [17]. A future is an object that stands in for the result of a computation. Futures decouple computation of intermediate values from the main control flow of a program, a future may be computed in the background. The main computation need only block if, when it needs the result, the background task has not completed. For instance in the previous example, the class `App` was forced to wait for the document conversion to terminate, with futures the same program can be written as:

```
Callable obj = new Callable() {
            Object call(Object arg) {
                    return conv.prepare((Document) arg);
           }};

Future future = new BasicFuture(obj, doc);
future.run();
...
Printable file = (Printable) future.get();
```

The application can now perform arbitrary actions between the time `run()` is invoked and the result is requested with `get()`. In Figure 2 the client blocked until completion of `prepare()`.

On the server side, the choice whether to have (i) one thread per request, (ii) a thread pool, or (iii) sequential processing is made by specifying a portal class. If `conv` is a stub created from an instance of `Portal`, calls to `prepare()` will be concurrent. On the other hand, if a deferrable portal had been used, along with the following code in `main()`, calls would be serialized.

```
while (true) { conv.accept(); }
```

A thread pool implementation can be derived by extending the above with logic to manage a set of threads.

3.2.3 *XIMI Class Stubs*

XIMI simplifies application development by avoiding the intermediate step of stub generation. RMI's requirement of a remote stub class compiler, `rmic`, is an extra step in the development cycle. For each remote method in an interface extending `Remote`, `rmic` generates a method in the stub class with the same signature that marshals its arguments,

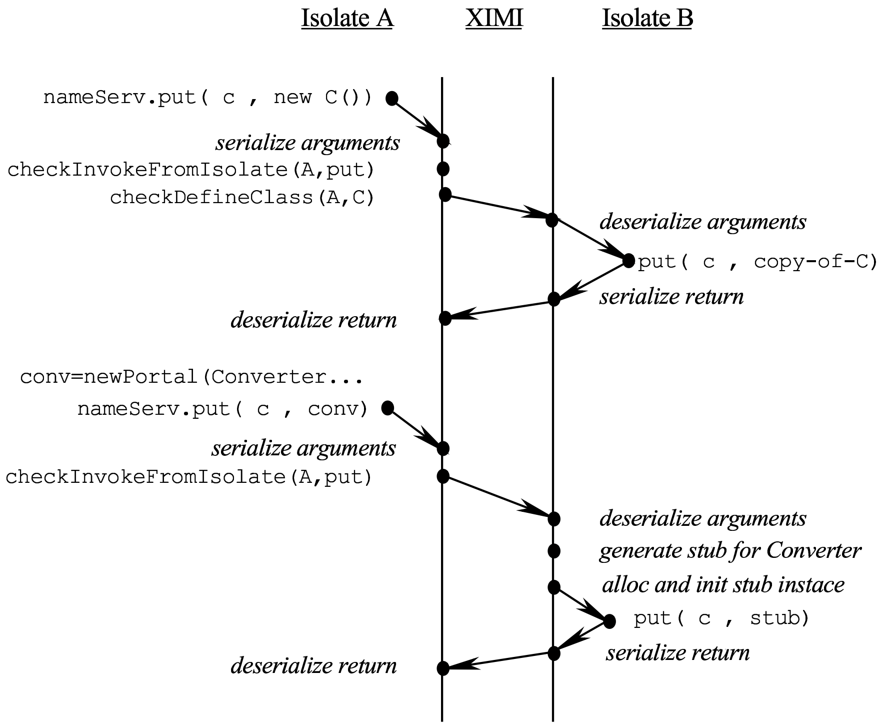

Figure 3: Overview of XIMI communication. Calls to the security manager and copies are explicitly indicated, we assume that the security manager for B is located in its parent B.

sends them to the remote object and unmarshals the return value it receives. Whenever an exported remote object is passed as a parameter or return value in a remote method call, the stub for that remote object is passed instead and the stub class has to be available for loading in both client and server. In XIMI stubs are generated dynamically, on demand, hence no preprocessing is required and no special tools need be invoked during development, nor is necessary to ensure stub availability at class loading time.

3.2.4 Fast loading

Our implementation is based on MVM which provides a fast loading mechanism that bypasses full class loading, including the fetching, parsing and verification of the class file. Full class loading is required only by the first isolate that loads a given class. Subsequent loads of the same class in other isolates reuse the previously created run-time system data structures, thus considerably speeding loading [7]. The current version of MVM limits fast loading to the default class loader. Future MVM versions will lift this restriction and allow fast loading for user-defined class loaders. XIMI takes advantage of fast class loading.

3.3 Enforcing Security Policies with XIMI

The security requirements of isolate communication differ from RMI in at least two respects. First, controlling network connections is a non-issue. Second, efficiency is crucial —not only should applications that do not require a security manager not pay for it, but those requiring security managers should not experience overheads that would dwarf the performance gains of XIMI. For these reasons, Incommunicado introduces a subclass of `SecurityManager` called `IsolateSecurityManager` that provides policy-neutral hooks that allow the implementations of a variety of security policies for controlling communication between isolates. Furthermore cross-isolate references can be used like capabilities as described next.

3.3.1 Capabilities

Capabilities are a well known access control mechanism [18] used in operating systems as well as some agent systems (e.g. [22] and [13]). A capability is an unforgeable token that grants certain access rights to its owner. Some authors have advocated the use of plain objects as capabilities [26, 11], under the rationale that references can not be manufactured and their type describes what can be done with the object. While this approach can be successful in certain cases, objects lack two important characteristics found in most capability-based systems: revocation and copy control. XIMI references behave as capabilities. Revocation can be achieved by closing a portal. The expression

```
portal.close();
```

will ensure that no more calls can be issued through the stubs associated with this portal. Calls in progress will not be affected. Copy control is meant to restrict the flow of capabilities between isolates:

```
Portal port = Portal.newPortal(Printer.class, obj, false);
```

will create a portal whose stubs can not be copied. Thus if an isolate acquires the stub, it will not be able to send it to another isolate via XIMI.

3.3.2 Interposition

Instances of `IsolateSecurityManager` and its subclasses are able to interpose on relevant XIMI operations and throw a `AccessControlException` if the current security policy is breached. The security exception is then serialized and rethrown in the caller. The interface of this class, given in Figure 1, consists of two methods: `checkClassDefinition(Isolate src, String cl)` is invoked every time a new class is about to be loaded as a result of a XIMI call. While this method may appear redundant given the normal security check on class loading, this is not the case since there is no easy way to check what event triggered a class load (stack inspection could be used, but it is rather inconvenient). The arguments to the method are the name of the class about to be loaded, the isolate that caused the load and the isolate in which the class will be loaded. The invocation of this method occurs during deserialization. If the security manager throws an exception the entire XIMI call is aborted. The other method in the security manager interface is `checkInvokeFromIsolate(Isolate src, Method met)`, called once for each XIMI method invocation. Its arguments are the originating isolate and the reflection object describing the method about to be invoked.

Security policies are chosen by the current isolate and dynamically associated with newly created isolates. In other words the same application can have different policies at different times. For instance,

```
isl = new Isolate("Application", null,
                  "RelaxedSecurityManager");
```

creates a new isolate running the `Application` class with an instance of `RelaxedSecurityManager`.

We will now illustrate some applications of the proposed API with examples from the literature.

3.3.3 The JavaSeal security model

The JavaSeal mobile object system [25] provides an abstraction called a seal (for *sealed object*), which plays a similar role to isolates. Just as isolates, seals are disjoint computations which communicate through channels. The seal security model enforces hierarchical communication —a seal can only communicate with its direct parent and direct descendants in the seal hierarchy. The model also imposes strong restrictions on class loading to protect seals against code injection attacks —seals can only exchange instances of classes known to both communicating parties. This policy can be expressed by restricting invocations to parent-child isolate, and by throwing an exception if class loading is triggered as a side effect of isolate communication. The following class demonstrates how to interpose on invocations and loading requests.

```
class SealPolicy extends IsolateSecurityManager {
    void checkDefineClass(Isolate s, String n) {
        throw new SecurityException();
    }
    void checkInvokeFromIsolate(Isolate src, Method m) {
        if (!src.equals(getParent(getCurrent())) &&
           !getCurrent().equals(getParent(src)))
            throw new SecurityException();
}   }
```

`checkInvokeFromIsolate` ensures that the source of inter-isolate method invocation is either the parent of the current isolate, or that the current isolate is the parent of the source. Note that this security manager is generic, in that the same instance can be used for multiple isolates.

3.3.4 The Secure Object Spaces model

Bryce simplified the access model of JavaSeal in [5] by allowing any pair of isolates (or in the paper's terminology object spaces) to communicate provided appropriate access rights could be obtained. Access rights are hierarchical, thus a parent can grant and revoke the communication rights of its children. A similar approach was investigated in [13]. To enforce this model, the security manager plays the role of a reference monitor checking an access matrix for every invocation.

```
class SOSPolicy extends IsolateSecurityManager {
    void checkInvokeFromIsolate(Isolate src, Method m) {
        if (matrix(src, getCurrent()) != GRANT)
            throw new SecurityException();
}   }
```

While the authors have experience with these models, it is not clear how well the proposed isolate security model will coexist with the Isolate API. Experimental results presented in Section 5 were obtained without explicit security managers.

4. IMPLEMENTING XIMI ON MVM

The Multitasking Virtual Machine (MVM) [7] is a general-purpose environment for executing disjoint Java applications in a single operating system process. Internally, each isolate is supported by a *logical* instance of the JVM. The bulk of the runtime as well as the runtime representation of loaded classes, which includes bytecodes, meta-data describing the class itself (*e.g.*, its fields, methods and constants) and code produced by a dynamic compiler, are shared among all JVM instances. Creating a new JVM instance adds only a small set of data structures that captures the part of the execution context of a program that cannot be shared (for example, storage for static variables, class monitors, and initialization status of classes loaded by the program, storage for its threads, and so on). This separation of the JVM runtime components into shareable and non-shareable and careful elimination of other possible sources of interference among Java applications improves scalability as well as strengthens application isolation when compared to other currently available approaches and mechanisms such as class loaders, separate processes, or middleware containers.

MVM grants each isolate a share of the global heap. No programmatic means are given to obtain a reference to an object allocated by one isolate from another. This guarantees that objects allocated by different isolates belong to disjoint graphs of objects, and that each such graph has disjoint sets of garbage collection roots. Each live root is uniquely associated with one isolate. Reclaiming heap space used by an isolate consists of ignoring the isolate's roots in subsequent garbage collections.

When an isolate is terminated, either because it exits or because another, privileged isolate invokes `halt()`, the resources allocated to the isolate must be reclaimed. In particular, all threads must be rolled forward out of any critical sections protecting shared resources up to a termination-safe point where all the threads of that isolate can be stopped and their resources reclaimed. Reclamation of resources includes executing finalizers. Application-defined finalizers are not guaranteed to terminate (e.g., because of infinite loops in their code); resource control techniques, beyond the scope of the paper, address this problem.

The goal of the Incommunicado implementation in the context of MVM was to introduce the smallest possible number of new virtual machine primitives that would provide adequate performance and to implement the remaining functionality in user-level libraries while taking advantage of the existing RMI infrastructure. Our implementation is contained in the `ximi` package. The new MVM runtime entry point is `mvm_invoke()` which is used to implement actual XIMI method invocations.

4.1 XIMI references

What makes XIMI references interesting is that they are cross-isolate references and thus introduce sharing between distinct isolates. Yet since the illusion of disjoint object graphs must be maintained, JVM-level sharing must remain hidden to user code written in the Java programming language. We address this issue by introducing the `ximi.XIReference` class, which extends `java.lang.ref.WeakReference`. Use of `XIReference` is restricted to VM code to ensure that user code will not be able to gain a direct cross-isolate pointer. The target of a `XIReference` is thus strongly reachable only if there are references to it in its own isolate.

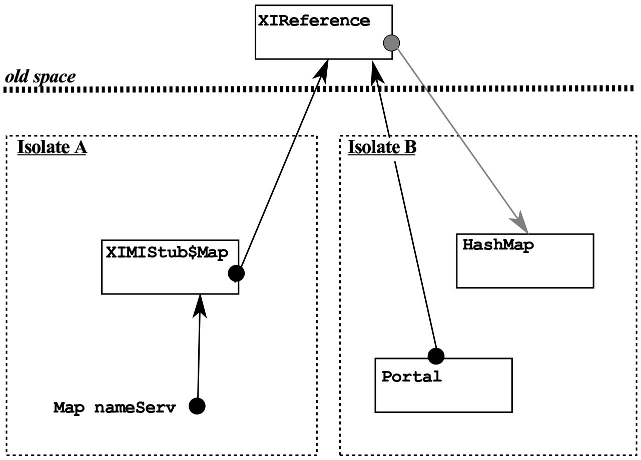

Figure 4: Cross-isolate references. A reference to a `Map` object in Isolate A points to stub which, in turn, points to the real object in Isolate B through a XIReference. The portal controlling the stub is located in Isolate B and has a reference to the XIReference.

Garbage collection is totally transparent in XIMI and does not need interfacing with user-level code. The functionality of `java.rmi.dgc` [9] is not needed for the proper functioning of XIMI, since global GC adequately fulfills that role. In MVM, the heap has generational organization. Each isolate has its own private, independently-collectable, new generation, while the old generation is shared physically (though not logically). `XIReference`s are allocated in old space to avoid being traversed during young generation collection and thus ensuring that the target object (in another isolate) is not traversed at young GC time. This scheme has the significant advantage that it does not entail changes to the GC algorithms. It is necessary to treat `XIReference`s as GC roots to prevent their target from being garbage collected during a young GC collection. Thus, new generation collections of one isolate never access new spaces of other isolates. This maintains the desirable property of isolate-local collection. Old space collections span all isolates and in effect perform an equivalent of a distributed garbage collection.

Each XIMI reference caches the identity hash code of its target so that `hashCode()` calls do not require cross-isolate accesses. The `equals()` method is implemented as a native call accessing the target and performing a pointer equality test (a native call is necessary because the target is a private field inherited from `Reference`). Also, synchronizing on a stub object will serialize accesses to that object but not to its target; this is consistent with the behavior of RMI and with MVM's properties of application isolation.

In order to ensure that targets of XIMI references are indeed unreachable to programmers, small changes to the JDK libraries were necessary. The Java programming language allows trusted user code to circumvent language protection mechanisms using reflection; hence it was necessary to introduce an extra check in the `setAccessible()` method of `java.lang.reflect.AccessibleObject` to prevent programmers from making the referent field accessible. This method is rarely used (mainly during serialization) and the extra check is not expensive.

4.2 Stubs and invocation

As mentioned earlier, `rmic` is not needed for creating stubs. Instead they are constructed on demand, as an instance of a dynamically generated subclass of `XIMIStub`, which extends `RemoteStub`. Note that instances of `RemoteStub` include a reference to an instance of `RemoteRef`. For `XIMIStub` this reference always points to an instance of `CrossIsolateReference`. Dynamic subclass generation is performed entirely in tentirelyprogramming language by the `XIMIStubGenerator` class similarly to the Proxy API introduced in JDK 1.3 [21].

When the generator constructs a stub, each method in the specified interface is translated to a method with the same signature as the original method. Arguments packaged as an array of objects are passed along with an instance of java.lang.reflect.Method identifying the remote method to be called, to the XIMI reference embedded in the stub instance. The reference object then calls its invoke() method which results in the mvm_invoke() VM call. The VM appropriately transmits the arguments over to the target isolate, resolves and calls the method in the target isolate, and transmits back the return value.

4.3 Serialization

The most significant win over RMI is due to optimized VM-local serialization and deserialization. For most objects serialization/deserialization can be replaced by a deep copy[5]. Special treatment is required for any object that must run user code during serialization. User code is invoked in the following cases:

- the class implements the Externalizable interface,
- the class (or its superclasses) has one of readObject(), readResolve(), writeObject(), writeReplace(), or
- the class is serializable but one of its parents is not.

Such classes will be referred to as *special* as opposed to classes which can be serialized entirely by the VM. Special classes require that the VM provide an interface to serialization that respects the standard protocol. Whenever the VM discovers that an instance of a special class has to be serialized, a pair of streams will be allocated and used for serialization.

The original serialization code attempts to serialize all objects reachable from the serialization root. Since in XIMI there are two serialization mechanisms working on the same object graph, one on the VM side and one in user code, there is a risk that an object could be copied twice. Moreover, when an object is deserialized the VM needs to know which object it is a copy of, to update all the fast-copied objects that pointed to the original. Unfortunately the information about an object's identity is lost during serialization. Therefore, the VM must be explicitly informed which objects are being written to the stream. We have created XIMIObjectOutputStream and XIMIObjectInputStream for this purpose. The replaceObject() method of ObjectOutputStream is overridden to always return the original object passed to it as an argument and to invoke an MVM callback which records the identity of the object being serialized. Similarly XIMIObjectInputStream overrides the resolveObject method to perform a MVM callback to set the identity of the object about to be deserialized.

For objects of normal classes, replaceObject() returns an instance of ximi.NormalObjectReplacement. This object has an integer field whose value is the address of a variable in the VM address space that, in turn, points to the actual object (a so-called global JNI handle). Such global JNI handles can be safely embedded in objects as integers because NormalObjectReplacement instances are never reachable from user code and even though the garbage collector may move objects to which JNI handles point to when serialization is being performed, the addresses of the handles themselves remain unaffected. The size of NormalObjectReplacement is most likely smaller than the size of the object it is replacing and since it has no reference fields, it will not cause the reflection code to serialize recursively all the objects the original object points to. Thanks to the callback in replaceObject() the VM is informed that the original object has to be copied on the VM side and its fields have to be processed. This technique ensures that pointers from special to normal objects are updated correctly.

To handle pointers from normal to special objects, instances of ximi.SpecialObjectRedirection are used. When a special object emerges from deserialization, the VM side does not know which special object it is a copy of, so it cannot update the fast-copies of normal objects to point to it. It is not guaranteed that during a given serialization the n-th call to resolveObject() will be invoked on the object returned by the n-th call to replaceObject(). Therefore one cannot assign serial numbers to special objects in the replaceObject() callback and identify them on deserialization. The following solution has been implemented. Each time a replaceObject() method is invoked on a special object, a SpecialObjectRedirection is created. Before the XIMIObjectOutputStream is closed, all the accumulated SpecialObjectRedirection objects are saved to it. These objects contain two fields, one is a reference to the original special object and the other is the same reference wrapped by a JNI handle. On deserialization, the native pointer retains its value while the original reference is updated to point to the deserialized copy of the special object. resolveObject() invokes a callback that informs MVM of the address of the copy of the special object so that references pointing to it can be correctly updated. Again, the memory overhead of those extra objects is relatively small (two fields plus serialization overhead) and proportional to the number of special objects.

Our solution has one problem: when an object contains a writeReplace method, the method is called and the resulting replacement object is given to the replaceObject() method. Therefore XIMIObjectOutputStream cannot see the original object and cannot register it with the VM. There might be references to the original object that the VM will not be able to update to refer to the replacement upon deserialization. To solve this problem, we changed the implementation of ObjectOutputStream to include a registerReplacement method called whenever a replacement is instantiated. The default implementation of this method does nothing, but XIMIObjectOutputStream overrides it and ensures that the SpecialObjectRedirection instance created for an object being serialized refers to the original object, not to its replacement. This is a small JDK change which can be easily maintained.

[5] The copy will only capture serializable objects and preserve the semantics of transient fields.

4.4 Threading

There are several choices regarding on which stack and in which thread in the target isolate remote method shall be invoked. XIMI currently employs an "impersonation" technique: whenever a call is performed on an object residing in another isolate the current thread temporarily changes its effective task ID, thus crossing over to the other isolate for the duration of the call or instantiation. To ensure preservation of isolation properties a new `Thread` object is allocated upon crossing the isolate boundary and associated with the VM-side thread object. When the remote call completes the original `Thread` object is restored (following stack discipline). Appropriate try-catch blocks are inserted to insulate the caller from a failure in the callee's code. It is likely that the same thread will call a remote method again on the same isolate, therefore for each isolate a weak reference to the thread most recently allocated by XIMI is maintained. This caching mechanism avoids repeated allocation of threads. Thread locals are nulled out when a thread crosses an isolate boundary and when a cached thread is reused.

4.5 Termination

An isolate can be terminated at any time, invalidating the referent fields of all `XIReference` pointing to objects of that isolate. These referent fields can only be accessed by special isolate-aware MVM downcalls or by the garbage collector. If an isolate is terminated while in JVM code, the computation will be aborted and an exception will be reported to the calling isolate. From the garbage collector's perspective all objects in the terminated isolate will be either unreachable or weakly reachable (since `XIReference` inherits from `WeakReference`). According to the language specification, once the garbage collector determines that an object is weakly reachable it will "atomically clear all weak references to that object and all weak references to any other weakly-reachable objects from which that object is reachable through a chain of strong and soft references" [21]. In the context of MVM this means that no dangling references will ever appear upon isolate termination: either the referent field of `XIReference` is `null` or it points to a valid object and the native downcall can determine that the isolate has been terminated. Once the referent fields are cleared all the objects belonging to the terminated isolate become available for reclamation. It is important that the new generation of the terminated isolate be actually scavenged (and not disposed of without examination), otherwise the `XIReferences` would not have their referent fields cleared, thus possibly misleading the global garbage collector during its future runs. An alternative technique not requiring new generation scavenging upon isolate termination would be to maintain a list of all XIMI references.

The current implementation limits immediate reclamation of resources upon isolate termination. If a remote call is in progress, the caller cannot be terminated because of the various VM-side resources allocated to the caller's thread. Termination is delayed until it returns from the inter-isolate call – which can take an arbitrarily long time. Lifting this limitation requires a mechanism for releasing VM-side resources from selected parts of the thread.

5. PERFORMANCE

To evaluate the efficiency of XIMI in comparison to RMI, we ran a set of microbenchmarks intended to reflect how RMI is used in real-world applications and how XIMI might be used for inter-isolate communication. In our baseline configuration both the client and the server were located in the same JVM and communicated via regular RMI. Therefore effects related to full serialization and use of operating system communication primitives are observed but not the network-related overhead. The experimental setup consisted of a Sun Enterprise^{TM} 3500 server with four UltraSPARC^{TM} II processors, with 4GB of main memory, running the Solaris^{TM} Operating Environment, version 2.8. The baseline configuration was executed using the Java HotSpot^{TM} virtual machine (referred to from now on as HSVM), version 1.3.1, with the JDK version 1.3.1. This is also the code basis for the MVM prototype used in our experiments, which allows for meaningful comparisons. Each remote method implementation in our benchmarks returns the value it has been passed as an argument. We organized our benchmarks in the following groups depending on the type of arguments passed to the remote call:

- **prims**: eight benchmarks in which values of various primitive types are passed, and one void call.
- **primarr**: eight benchmarks in which 100 element arrays of various primitive types are passed around.
- **smallobj**: one benchmark with balanced binary trees of depth 5.
- **bigobj**: one benchmark with binary trees of depth 5; each node in the tree contains the same fields as in `smallobj` and also one field for every primitive type and a `String` field
- **objarr**: one benchmark with 100-element arrays of the same type as nodes in `bigobj`; each array entry is a root of a balanced binary tree of depth 5
- **remote**: one benchmark with 100-element arrays of objects implementing the `Remote` interface

Figure 5 presents the execution time of XIMI-based invocations expressed as percentage of the execution time of the baseline configuration. For each type of argument we measured the time needed to execute 1000 remote method invocations. The data shows that XIMI is at least 8 times faster than RMI. The biggest difference is observed with `smallobj`, for which XIMI performs more than 70 times better than the baseline. This can be explained by XIMI's more efficient serialization, executed entirely within the VM in this benchmark. Other benchmarks do not experience similar improvements either because serialization is not as heavily used (e.g., `primarr` serializes a single object per invocation, `prims` serializes only one boxed primitive value per invocation), or because other costs are more prominent. In particular, the performance gains obtained from an optimized serialization decrease as the size of transferred objects increases, as indicated by the difference between `smallobj` and `bigobj`.

We believe that most actual uses of RMI resemble the `smallobj` benchmark, hence its good performance is most indicative of the advantages XIMI may have in the context of its practical, real-world applications. It is interesting to see that XIMI achieves better improvement on `primarr` than `primobj`. However, recall that in order to perform remote method invocation with a primitive argument, both XIMI and RMI wrap the primitive value with an object type during call marshalling and unwrap the result on method return. This is not required in case of `primarr` since arrays are objects already. The fixed cost of boxing primitives is incurred by both RMI and XIMI and hence the improvement achieved by XIMI in `prims` is lower than in `primarr`. In fact while for RMI `prims` is faster in absolute numbers than `primarr`, for XIMI it is `primarr` that is faster than `prims`.

Figure 6 shows the contributions of various stages in XIMI as the percentage of the total running time. The costs of a XIMI remote invocation break down into five components: byte-copying, reference patching, stub creation, isolate context switch, and "other" costs. The copying mechanism for passing arguments and returning a result proceeds in two phases. Phase 1 copies the objects into the target isolate, sets the transient fields in the copies to their default values, and establishes a mapping between original objects and their copies. Phase 2 uses this mapping to update all reference fields in the copies. Whereas performance of phase 1 depends on the size of the objects, phase 2 depends on the number of copied references. As result, the duration of phase 2 for both `prims` and `primarr`, where the passed objects do not include any references, is only a small fraction of the overall running time. Dynamic stub generation imposes a cost only for objects that are passed as stubs across isolates, as it is the case with `remote`. The generation of a stub class is a one-time event across all isolates, and only the first isolate to use a particular stub class pays the cost of full loading of that class. Subsequent isolates use fast loading. The costs of crossing isolate boundaries are negligible due to the thread impersonation technique described earlier.

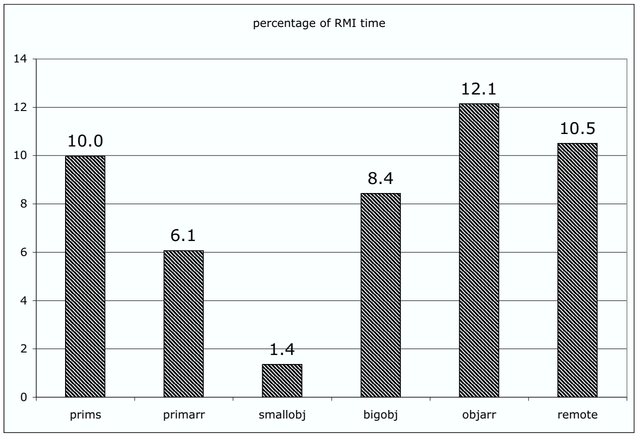

Figure 5: XIMI running time as percentage of RMI running time.

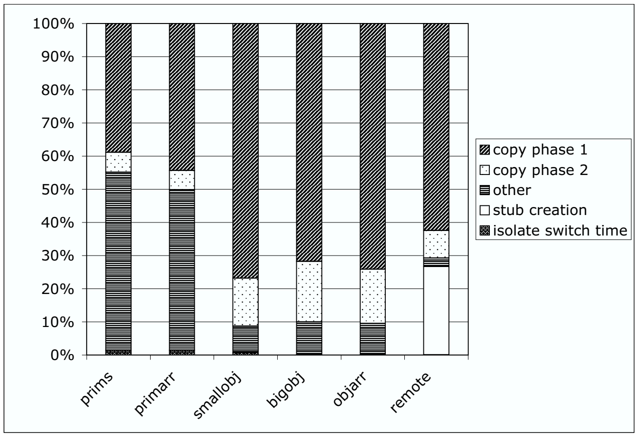

Figure 6: Remote invocation stages and their contribution to run-time overheads.

The above results do not reflect the costs of handling special objects during serialization. These costs are highly dependent on the number of special objects. Our experiments indicate that in the extreme case, when all serialized objects are special, XIMI is in fact slower than RMI. This is understandable since synchronizing two serialization mechanisms is costly as it involves serializing additional objects and communicating between the Java classes and the VM code. However, in many cases custom serialization yields exactly the same results as the deep copy performed by XIMI. For example, when implementing an array-based list the authors of the class might choose not to rely on the default serialization that would write the underlying array with possibly many `null` entries at the end but instead write only the non-`null` entries directly to the stream and recreate the array on deserialization. XIMI cannot in general discover that serialization and deserialization is semantically equivalent to a deep copy but code written in the Java programming language might indicate that this is the case by appropriately annotating classes with that property, e.g., using a marker interface. Standard serialization would ignore this interface but XIMI would recognize it as an indication not to treat instances of classes implementing this interface specially.

6. RELATED WORK

A substantial body of work exists on specializing RPC for the local case. Lightweight RPC [2] and doors [12] are good examples. In the context of the Java programming language and the JVM, related efforts revolve around (i) interfacing the JVM with fast, typically non-TPC/IP protocols, (ii) improving the speed of RMI, by either layering it on top of faster protocols or by redesigning and re-implementing it, and (iii) designing alternative communication mechanisms for computations executed in the same instance of the JVM. The notion of portals to regulate inter-domain communication was explored in the Seal calculus [23]. The interposition approach for the security manager is not novel, similar ideas have been studied in the security community [15, 10].

Exposing new communication primitives to the programmer is typically accomplished either via JNI or through extending the VM. In Javia [6] a user-level network interface is exposed to programmers as a set of send/receive methods. A major issue is buffer management. Applications can allocate pinned regions of memory and use them as arrays in the Java programming language. These arrays are objects and can be accessed directly but are not affected by garbage collection as long as they need to remain accessible by the network interface. Programmers manage buffers explicitly and safely by detaching their lifetime from the lifetime of their references. To maintain the safety properties of the language, the garbage collector was modified to change the scope of its collected heap dynamically. Buffers are not moved by the collector until their deallocation, which happens only after the application states there are no more references to the buffer and after the collector verifies this claim. Buffer management in Javia is analogous to some issues in XIMI: cross-isolate references effectively maintain communication buffers. One of the reasons Javia designers rely on programmer's explicit cooperation in identifying unused communication buffers is the lack of weak references in their version of the JDK.

Manta [19] and KaRMI [20] are efficient re-implementations of the RMI for high-performance parallel computing. While these systems faced some of the same difficulties as XIMI, their solutions are different. Manta layers RMI on top of a user-level communication subsystem. For efficiency Manta relies on compile-time analysis to avoid creating threads at the callee's site. The Manta RMI protocol cooperates with the garbage collector to keep track of references across machine boundaries. In KaRMI a major source of performance gains is slim encoding of type information: in the parallel computing settings, it is safe to assume that the bytecode of objects being exchanged is always available from the file system. Another serialization improvement is implementing buffering on the receiver side. KaRMI departs from RMI in that it cannot deal with code that explicitly uses socket factories or port numbers.

Object caching at the client side has been identified as a potential source of significant performance improvements to RMI [16]. Since most remote accesses to objects are typically read-only this approach can work well. However, it would not be very useful for XIMI as communication is fast.

Several projects aimed at safe and controlled direct sharing of objects among computations running in a single instance of the JVM. For instance the J-Kernel [13] adds protection domains to the Java programming language and makes a strong distinction between objects that can be shared between domains, and objects that are confined to a single domain. Inter-application communication is performed via deep object copies of method arguments and return values, similarly to XIMI. However, certain objects can be shared directly. They are wrapped inside capability objects, which support revocation. Accessing such objects is more expensive than using plain object references, since revocation status must be checked each time. Bryce and Vitek implemented JavaSeal, a mobile object platform in which disjoint computations, called seals, could coexist in the same Java virtual machine [25]. One of the differences with our approach is that JavaSeal was a purely user-level package which relied on user-defined class loaders to enforce disjointedness. Bryce later worked on secure object spaces in which bytecode rewriting was used to ensure isolation [5].

Direct object sharing has also been implemented through the modifications to the virtual machine. An example of this approach is KaffeOS [1], which supports the OS abstraction of a process in the JVM: each process executes as if it were run in its own JVM, and has a separately collectable heap. There is also a single kernel heap, maintained by trusted code and used to store objects related to user processes, for example, the objects that represent processes itself. KaffeOS processes can dynamically create a shared heap to communicate with other processes. Objects on the shared heap are not allowed to have pointers to objects on any user heap, because those pointers would prevent this user heap's full reclamation. Write barriers enforce this restriction; attempts to assign such pointers will result in an exception. As a write barrier is executed on every pointer write, even applications that never communicate incur performance overhead related to the cost of object sharing. We view this as an important limitation. XIMI's use of weak references effectively achieves object sharing at no cost for non-communicating programs.

7. CONCLUSIONS

Extending a multitasking-enabled virtual machine with a communication mechanism that is at the same time efficient, does not break any property of the virtual machine, and is exposed to programmers through a simple and flexible API poses various challenges. In this paper we presented a detailed analysis of the design and implementation of a particular realization of this goal: equipping MVM with an efficient inter-isolate communication mechanism. Incommunicado demonstrates that the goal of high-performance does not have to conflict with safety and isolation properties. Several techniques taking advantage of the properties of the Java platform and collocation of computations in the same instance of the JVM lead to a significant performance improvements – at least eight-fold when compared to the local invocation of RMI – while preserving the isolation, resource accounting and clean termination properties of MVM. Our investigation of isolate communication has been performed independently from the work of the JSR 121 expert group, in future work we plan to reimplement XIMI on top of the link interface.

Acknowledgments: Krzysztof Palacz implemented XIMI during a visit to Sun Microsystems Laboratories. The authors are grateful to David Holmes for comments, to Doug Lea for in depth discussions of the JSR 121 API, to Pete Soper for his help and comments as well as for taking the lead in the JSR 121 effort, to Miles Sabin for his comments on portals and security issues, and to Doug Simon and Glenn Skinner for last-minute corrections. Vitek is supported by NSF CAREER grant #0093282-CCR and CERIAS.

Trademarks: Sun, Sun Microsystems, Inc., Java, JVM, HotSpot, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc., in the United States and other countries. UltraSPARC is a trademarks or registered trademarks of SPARC International, Inc. in the United States and other countries. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.

8. REFERENCES

[1] G. Back, W. H. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in java. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI-00)*, Berkeley, October 2000.

[2] B. N. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[3] W. Binder, J Hulaas, and A. Villazon. Portable resource control in java: The j-seal2 approach. In *Proceedings of the 16th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-01)*, October 2001.

[4] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[5] C. Bryce and C. Razafimahefa. An approach to safe object sharing. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Application (OOPSLA-00)*, October 15–19 2000.

[6] C. Chang and T. von Eicken. Interfacing Java with the virtual interface architecture. In *ACM 1999 Java Grande Conference*, June 1999.

[7] G. Czajkowski and L. Daynès. Multitasking without compromise: a virtual machine evolution. In *Proceedings of the 16th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-01)*, October 1998.

[8] G. Czajkowski and T. von Eicken. JRes: A resource accounting interface for Java. In *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98)*, October 18–22 2001.

[9] T. Downing. *Java RMI*. IDG Books, 1998.

[10] T. Fraser, L. Badger, and M. Feldman. Hardening COTS software with generic software wrappers. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy (SSP '99)*, May 1999.

[11] L. Gong. *Inside Java 2 platform security: architecture, API design, and implementation*. Addison-Wesley, Reading, MA, 1999.

[12] G. Hamilton and P. Kougiouris. The Spring nucleus: a microkernel for objects. In *Summer USENIX Conference*, June 1993.

[13] C. Hawblitzel, C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing multiple protection domains in Java. In *Proceedings of the USENIX 1998 Annual Technical Conference*, New Orleans, LA, June 1998.

[14] Java Community Process. Application Isolation API Specification. http://jcp.org/jsr/detail/121.jsp, 2002.

[15] M. B. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Proceedings of the 14th Symposium on Operating Systems Principles*, December 1993.

[16] V. Krishnaswamy, D. Walther, S. Bhola, E. Bommaiah, G. Riley, B. Topol, and M. Ahamad. Efficient implementations of Java remote method invocation. In *Proceedings of the 4th Conference on Object-Oriented Technologies and Systems (COOTS-98)*, Berkeley, April 1998.

[17] D. Lea. *Concurrent Programming in Java*. Addison-Wesley, Reading, MA, 1997.

[18] H. Levy, editor. *Capability Based Computer Systems*. Digital Press, 1984.

[19] J. Massen, R. van Nieuwpoort, R. Veldema, H. Bal, and A Plaat. An efficient implementation of java's remote method invocation. In *ACM PPoPP*, Atlanta, GA, May 1999.

[20] C. Nester, M. Philippsen, and B. Haumacher. A more efficient RMI for Java. In *Java Grande Conference*, San Francisco, CA, June 1999.

[21] Sun Microsystems, Ind. Java 2 SDK, Standard Edition Documentation, 2001.

[22] A. Tanenbaum, S. Mullender, and R. van Renesse. Using Sparse Capabilities in a Distributed Operating System. In *The 6th International Conference on Distributed Computing Systems*, May 1986.

[23] J. Vitek. *The Seal Calculus of mobile computation*. PhD thesis, University of Geneva, 1999.

[24] J. Vitek and B. Bokowski. Confined types in Java. *Software Practice and Experience*, 31(6):507–532, 2001.

[25] J. Vitek and C. Bryce. The JavaSeal mobile agent kernel. *Autonomous Agents and Multi-Agent Systems*, 4, 2001.

[26] D. Wallach, D. Balfanz, D. Dean, and E. Felton. Extensible Security Architectures for Java. In *Proceedings of the 16th Symposium on Operating System Principles*, 1997.