CERIAS Tech Report 2002-40

INCOMMUNICADO: FAST COMMUNICATION FOR ISOLATES

by Krzysztof Palacz, Grzegorz Czajkowski,
Laurent Daynès, Jan Vitek

Center for Education and Research in
Information Assurance and Security,
Purdue University, West Lafayette, IN 47907

Incommunicado: Efficient Communication for Isolates

Krzysztof Palacz Grzegorz Czajkowski† Laurent Daynès† Jan Vitek

S^{3}Lab, 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 [**?**], and resource limits [**?**, **?**]. 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 [**?**].

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 [**?**], 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) [**?**]. 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 [**?**].
- **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.

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

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);
...
```

```
// 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 [**?**]). 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.

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

This section introduces the revised XIMI programming model. Implementation issues will be discussed in Section **??**. We start by contrasting XIMI with RMI.

3.1 Why not RMI?

The abstraction of remote procedure call (RPC) has proven to be versatile [**?**], 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 [**?**], 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

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 **??**.

`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 **??** 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 **??**. 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 **??** 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 **??** 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* [**?**]. 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 **??** 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 [**?**]. 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 [**?**] used in operating systems as well as some agent systems (e.g. [**?**] and [**?**]). 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 [**?**, **?**], 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 **??**, 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 [**?**] 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