Non-Existent Foo & Bar Co.
Process-Change Proposal #28770

June 8, 2006

To: Imagine A. Manager
Director, Distributed-Applications

From: Suraj N. Kurapati
Analyst, Research & Development

Proposal: Use Java RMI to encapsulate low-level inter-process communication in our distributed-applications.

Start Date: 1st April 2005.

Duration: 6 weeks for evaluation.

Requested Funds: $300.00

Abstract

Presently, our distributed-applications employ a low-level approach to inter-process communication[1] (IPC), which burdens our programmers with protocol-design and -synchronization issues that are unimportant to the business logic they implement. To circumvent these issues in the future, the use of a high-level approach to IPC, such as Java RMI, is suggested; for such an approach can shorten development time, reduce development costs, and improve the reliability of our distributed-applications.

Contents

[1] A mechanism by which two or more running copies of a single program communicate with each other.

List of Algorithms

1 Introduction

This section, intended for non-specialist readers, introduces the purpose and significance of this proposal. Please refer to section 2 on page 7 for a more technical discussion.

1.1 Problem

Our distributed-applications, software that operates concurrently on one or more computers that are connected together through a common computer network, presently employ a complicated low-level method of IPC. This method facilities the physical transport of individual pieces of communication, or messages. Thus, in order to utilize this method, our developers have to custom-design messages, their appearance, meaning, and how they are processed.

Though not apparent at first, this method presents several drawbacks, discussed below, which distract our developers from implementing business logic and instead preoccupy them with low-level IPC programming.

1.1.1 Laborious

This method of IPC is inherently inflexible and soon becomes impractical as more functionality is added to a distributed-application. For example, when a new piece of functionality, such as retrieving the number of computers currently running a given distributed-application, is added to one of our distributed-applications, a torrent of new messages must be custom-designed. This process is immensely laborious in contrast to implementing the new functionality in a distributed-application's business logic—which simply involves writing new software subroutines.

1.1.2 Interoperability

IPC message designs vary between our distributed-applications, as each of them was designed to perform a wholly different set of business tasks. This variation makes it difficult, if not impractical, for our distributed-applications to inter-operate with each other. In addition, this inability to inter-operate at the IPC level is the primary obstacle keeping our company from marketing our distributed-applications in suites.

1.1.3 Complexity

Because this method only facilitates the transport of messages, our developers are forced to add extra logic to prevent miscommunication of obsolete or incorrect information, and to organize, delegate, and synchronize communication between groups of computers serving our distributed-applications.

1.2 Objectives

A better method of IPC, which reduces the aforementioned issues whilst enabling our developers to focus on implementing business logic, is essential. In particular, such a method must

- hide low-level details of IPC.
- provide an intuitive, consistent, high-level programming interface for IPC.
- facilitate IPC reliably—automatic or transparent message-synchronization, -serialization, -retransmission, and -processing.
- provide a clean, reusable, and extensible separation of interface and implementation to avoid an explosion of IPC messages upon adding new functionality.

At minimum, these conditions should be met in order to effectively circumvent the problems introduced by the present method of IPC.

1.3 Significance

The proposed approach encapsulates low-level IPC details and enables our developers to better focus on more important matters—implementing solid functionality that allows our customers to perform their business tasks quickly and reliably.

1.4 Risks

The amount of risk perceived depends upon the type of high-level approach chosen to implement IPC in our distributed-applications in the future. Nevertheless, the approval of this process-change proposal implies the following risks, which were inspired by, determined, and analyzed using techniques presented in [3].

Note that risks involving product failure exist presently and independently of the proposed change.

1.4.1 Human

Our distributed-applications' developers must

- be trained in how RMI works and how it can be used (see Section 2).

- rethink how their designs can function using the proposed high-level approach to IPC.

Otherwise, our developers may produce distributed-applications of poor quality because they lack expertise in the use of the proposed approach.

1.4.2 Operational

There will be a possibly lengthy pause in the development schedule of our distributed-applications whilst our developers are learning how to use the proposed approach to IPC. Consequently, losses may incur due to the delay in release of products currently in development.

1.4.3 Reputation

Our customers may be disappointed in the delay of our upcoming products. Thus, we may damage our company's reputation of being *the* company that sets industry standards in developing distributed-applications of the utmost low-level complexity. In addition, some of our top developers, who believe strongly in our commitment to developing unnecessarily complex distributed-applications, may lose confidence in our company's image, resign, and become employed by our company's competitors.

1.4.4 Procedural

Because the proposed approach encapsulates IPC in a manner transparent to our developers, failure in its internal mechanisms may cause our distributed-applications to fail. Thus, our company may be held, unjustly, accountable for losses if our products fail under said conditions. Furthermore, such an event could taint our reputation of engineering only the fastest, most reliable, egregious of distributed-applications.

1.4.5 Project

Our developers' possible lack of expertise in using the proposed approach to IPC may delay the development of current and future products; possibly leading to over-expenditure of budget, decreased reliability of our products, and increased invocation of customer-support—which may further drain our company's capital.

1.4.6 Financial

Possible delays in shipping our current products in development may negatively affect our presence in the stock market and incur heavy losses in our company's capital. At worst, our company could become bankrupt and render its brilliant employees unemployed.

1.4.7 Technical

A newer, faster, cleaner, more elegant approach to IPC may be more beneficial while introducing less risks than the proposed one. That is, the proposed approach to IPC may be a substandard, insufficient, impoverished, or obsolete approach that is condemned by industry leaders for its shortcomings.

1.4.8 Political

Some of our customers, being governmental agencies, may not approve of having our distributed-applications' IPC implemented using a a third-party technology. They may require distributed-applications to use a low-level approach to IPC because they find it to be more secure, reliable, and salvageable in case of failure; in which case, we may lose a significant portion of our present and potential customer base by implementing the proposed high-level approach to IPC.

1.5 Benefits

In spite of the possible risks, the reduced complexity brought about by the proposed approach to IPC, will enable our developers to focus on implementing business logic. Consequently, improved product quality, shortened development time, reduced development costs, and increased product sales may result.

In addition, our distributed-applications will be interoperable, due to their use of a common, high-level interface for IPC. Such functionality may enable our company to market our distributed-applications in suites to gain larger profits.

2 Background

This section, intended for technical readers, introduces the purpose and significance of this proposal. Please refer to section 1 on page 4 for a less technical discussion. Also, the phrase "Java RMI" will be used in referring to the Java RMI technology, whereas the word "RMI" will be used as an abbreviation for "Remote Method Invocation."

2.1 Problem

Our distributed-applications presently employ a custom application-level[2] protocol, which operates over the Internet Protocol (IP) [1], for IPC. This protocol operates by having two or more parties exchange sequences of bytes, known as messages. Thus, this protocol can be considered to be message-based [1].

2.1.1 Usage

Message-based protocols are primarily used in two ways: (1) to propagate changes in data and (2) to propagate knowledge of occurrences of events. The

[2]The top-most level of the OSI networking model [1].

former way is embodied in data-based messages, which specify a subject and its change whilst implying the event that brought about the change in the subject. Whereas the latter way is embodied in event-based messages, which specify an event that has occurred whilst implying an associated subject and its change.

For example, consider a situation where process[3] A is waiting for some change to be reported by process B. Suppose that process B has finished performing a computation and has stored its result. Now, process B can notify process A via (1) a data-based message, which would specify the old and new result of process B's computation whilst implying that process B has performed a computation; or (2) an event-based message, which would specify that process B has finished performing a computation whilst implying that process A can fetch the result of process B's computation if it so desires.

2.1.2 Design

In addition to implementing pathways—the aforementioned ways in which message-based protocols are used—for message exchanges, our developers must design messages and write logic to serialize, deserialize, and semantically process them. In particular, they must design

- how a message appears to the sender
- how a message appears to the recipient
- how a message appears while it is being transported
- how the recipient shall attempt to comprehend a message

and conjure strategies for

- how a message is transported to the recipient
- what happens if a message is lost during transport
- what happens if a message becomes distorted during transport
- what happens if portions of a message become distorted during transport
- what happens if a sequence of messages is delivered in the wrong order

2.1.3 Remarks

Depending on the type of tasks performed by our distributed-applications, choosing solely an event-based or data-based message exchange protocol may minimize the number of messages needed to be designed and implemented.

In practice, a combination of event-based and data-based messages is often used because event-based messages are more suitable for certain IPC and

[3] A single, running copy of a program.

data-based messages are more suitable for others. However, using both message schemes significantly bloats the source-code dealing with IPC. In addition, both parties of a message exchange must be programmed to handle messages in common—which duplicates message-handling code in many cases.

Furthermore, if the semantics of a message change, all parties which are programmed to handle that message must be updated accordingly. Similarly, if new functionality added, new messages and their respective handling procedures must be designed and implemented in excess of the anticipated extension of business logic. Naturally, such events consume huge amounts of development time, are tedious, and may enable a programmer, who does not fully understand the semantics of the message amongst various parties, to introduce defects.

Finally, it is often futile to avoid re-design and -implementation of messages by sharing messages between different distributed-applications because it results in tightly-coupled, brittle software. For example, if the semantics of a certain message changes in one application, all other applications which interact with the same message must also be changed accordingly. Furthermore, because each of our distributed-applications is designed to perform different sets of business tasks, one application's messages may not be relevant to, and unnecessarily bloat, another.

Thus, message-based protocols are inflexible, error-prone, and require an enormous amount of design and programming. Consequently, most of our distributed-applications' source-code is preoccupied with IPC, rather than business logic.

2.2 Approach

Java RMI can be thought of as an object-oriented version of traditional RPC (see Appendix 6.1 on page 17). However, it goes a step further from traditional RPC by facilitating transparent serialization of objects and entire trees of their references in addition to that of primitive[4] data-types [5]. This enables developers to pass complex data-structures as arguments to a RMI, regardless of whether they exist in the local Java Virtual Machine (JVM) or a remote one [5, 2].

Furthermore, Java RMI is *the* choice for IPC in (1) Jini technologies, which power dynamic service discovery in unreliable ad-hoc, mobile wireless networks, and (2) J2EE technologies, which power complex, mission-critical, multi-tiered, enterprise applications [2].

2.2.1 Utilization

To visualize how Java RMI is utilized, consider the following scenario.

Suppose that our director of Human Resources wishes to issue a monetary bonus of $100.00 to a few brilliant employees, whose identification numbers are 160, 230, and 986 respectively. Naturally, our director creates a new instance of the `HRDirector` object, invokes its `issueBonus` method whilst specifying the

[4] Data-types such as `int`, `char`, and `float` in the Java programming language.

employees' identification numbers as parameter, and examines the return value and standard output and error streams (see Algorithm 9). Surprisingly, that is all it takes to utilize a distributed-application which uses Java RMI for its IPC.

Behind the scenes, however, the process is a bit more complex, but nonetheless simpler than low-level IPC. Let us dissect and examine this process step by step.

First, The `rmiregistry` Daemon[5]

Our company invests in a dedicated RMI server whose primary role is to run the `rmiregistry` daemon. This daemon provides naming services for objects that have registered with it [6], much like a Yellow Pages [4] telephone directory provides telephone numbers of businesses that have registered to be listed within it. It enables any Java objects to discover and obtain **stubs** in order to invoke services provided by their associated **skeletons** [6].

Stubs are light-weight interfaces [6] which (1) act as representatives for and (2) provide a mechanism for interaction with a more heavy-weight associate—much like a telephone number (1) represents and (2) provides a mechanism for interaction with a business. These interfaces are always descendants of `java.rmi.Remote` [6] and all of their methods are marked as being able to throw a `java.rmi.RemoteException` [6]. This exception is thrown upon:

- "Communication failure (the remote server is unreachable or is refusing connections; the connection is closed by the server, etc.)" [6].
- "Failure during parameter or return value marshalling or unmarshalling" [6].
- "Protocol errors" [6].

When a stubs' method is invoked, it communicates with its associate skeleton in its originating JVM via Java RMI—which in turn utilizes low-level IPC [6]. However, these details are irrelevant to the method's invoker; a method invocation on the stub simply *feels* just like a method invocation on any Java object—the only difference being that remote methods always are capable of throwing a `java.rmi.RemoteException` [6].

Skeletons are heavy-weight objects [6] which (1) generally provide services for arbitrary objects and (2) are represented by a more light-weight associate through which their services can be invoked—much like a business (1) generally provides goods and services for arbitrary customers and (2) is represented by a telephone number through which its services can be invoked.

When a skeleton is passed as an argument to a RMI, the semantics of the RMI is altered depending upon the skeleton's super-class. That is, if the skeleton

[5] An ever-present process which provides services to other processes and persons.

is a descendant of `java.rmi.server.UnicastRemoteObject`, then it is casted as its associate stub and passed *by value*—also known as a **live reference** [6]. Otherwise, the skeleton is serialized and passed *by value*, just like a primitive data-type [6].

Because skeletons often contain their own non-remote methods in addition to their stubs' remote methods, a more complex situation arises when a remote method requires that a skeleton[6] be passed as its argument. In such a case, if the remote method invokes one of the passed skeleton's non-remote methods, a regular method invocation takes place where the passed skeleton's non-remote method executes within the JVM of the remote method to which the skeleton was passed [6]. Whereas if the remote method invokes one of the passed skeleton's remote methods, then a RMI will take place and the invoked method will execute in the passed skeleton's JVM [6].

Second, The RMI Naming Service

The `FooBarCompany` object (see Algorithm 1) initializes its `_registry` reference to our dedicated RMI server's naming services and facilitates other objects' access to these naming services via its `getRegistry` method. That is, objects can now register themselves with and discover already registered objects via the `java.rmi.Registry` object [6] returned by the `FooBarCompany.getRegistry` method (see constructors in Algorithms 5 and 7).

Third, Distributed-Object Design

Our Human Resources division and its Payroll department, respectively

1. write source-code for stubs for the services they wish to provide (see Algorithms 4 and 6)
2. write source-code for skeletons that implement their services (see Algorithms 5 and 7)
3. instantiate their skeletons
4. register their skeletons as using our company's RMI naming service (see constructors in Algorithms 5 and 7)

The skeleton registration method `java.rmi.registry.Registry.rebind` requires that a unique name (see variable `RMI_REGISTRY_NAME` in Algorithms 4 and 6) associated with the skeleton be specified alongside the skeleton itself [6]. This unique name is used by other objects when attempting to obtain stubs of registered skeletons [6]—just as a person uses the name of a business when attempting to obtain its telephone number in a telephone directory.

Also, note that when a skeleton is registered via the aforementioned registration method, the stub associated with the skeleton is registered in its place [6]. Thus, a skeleton is *only* accessible via its associated stub [6].

[6] Usually remote methods require that stubs be passed as their arguments instead of skeletons because they can in turn invoke remote methods belonging to the passed stubs.

Fourth, Obtaining Stubs

The `HRDirector` object obtains stubs for services provided by the `EmployeeDatabase` and `PayrollDatabase` skeletons for use by its `issueBonus` method (see constructor in Algorithm 8). These stubs are obtained by specifying the unique name with which their associate skeletons registered themselves via the RMI naming service [6].

Fifth, Utilizing Stubs

Our director instantiates the `HRDirector` object, invokes its `issueBonus` method (see Algorithm 8), and the fun begins.

First, each employee identification number specified by our director is verified with the `EmployeeDatabase`, via its `getEmployeeById` method, whilst retrieving the `EmployeeRemote` stub associated with the employee whose identification number is presently under scrutiny. Assuming that the `EmployeeDatabase` skeleton is running in a JVM other than that of our directors', the retrieved stub represents a *live* reference of its associate skeleton in a remote JVM [6].

Second, the `PayrollDatabase` is instructed to issue a bonus, via its `issueBonus` method, to the employee represented by the previously obtained `EmployeeRemote` stub. Notice that we pass *both* local (our director's `HRDirector` object and the bonus' amount) and remote (the previously obtained `EmployeeRemote` stub) objects as arguments to this RMI.

Third, the payroll database checks if the monetary mount specified for the bonus is valid, feasible, and issuable. If so, it issues a bonus to the specified employee accordingly. Otherwise, it throws a `InvalidBonusAmountException` object which, through Java RMI, is automatically serialized and passed *by value* to our director's `HRDirector` object [6].

Additionally, the payroll database may proceed to verify that the specified employee has not been already been given a bonus during the last work-week by invoking the `EmployeeRemote` stub's `getIdNumber` remote method. In this case, the RMI will effectively return a *copy* [6] of the skeleton's `_id` member variable (see method `getIdNumber` in Algorithm 3).

Finally, if any exceptions occurred during execution of our director's initial invocation of the `HRDirector.issueBonus` method, appropriate notifications will be printed to the standard error stream for further scrutiny by our director. Otherwise, our director will see that the invoked method has returned, with a `void` return value in this case, without having thrown any exceptions.

2.2.2 Remarks

Notice that in the previous example on Java RMI utilization, we were forced to separate the interface from the implementation of our services [6]. This important design feature enables skeletons to extend, improve, and evolve independently of their associate stubs whilst having their implementation completely hidden. For example, a business that is listed in a telephone directory may promote their most egregious employees, but such change is not reflected in

its telephone number. Similarly, an arbitrary customer seeking the business' services in a telephone directory is usually unconcerned about promotion of employees within the business.

Furthermore, the amount of maintenance required to modernize or enhance distributed-applications, whose IPC is encapsulated by Java RMI, is dramatically reduced when compared to those whose IPC is implemented using a low-level approach. That is, with Java RMI, there is no explosion of low-level IPC messages to be designed and implemented whenever new functionality is added to a distributed-application. Rather, a remote method corresponding to the new functionality is added to a stub, implemented in its associate skeleton, and ready for use. This process is astonishingly similar to the common practice of adding new functionality to regular, non-distributed-applications. Thus, with Java RMI, our developers need not waste their time thinking about low-level IPC and, consequently, they are better equipped to perform their primary duties—to design and implement pure business logic.

2.2.3 Alternatives

Because our present distributed-applications are written in the Java programming language, it would be counterproductive[7] to present alternative high-level approaches that do not operate over a network of heterogeneous processors. That is, with such approaches, our distributed-applications would no longer run on a broad array of computers architectures [5]. In which case, we would have to write a separate versions of our distributed-applications for each specific computer architecture—a tedious, costly, and obsolete approach. Thus, only those alternative high-level approaches which facilitate IPC between heterogeneous processors are considered here.

A more thorough treatment of the following alternatives to Java RMI, including unmentioned, orthogonal approaches such as MPI and DCOM, is given in [5].

CORBA has a concept of separating service interfaces from their implementation similar to Java RMI [5]. It is static, requiring manual configuration of IPC pathways, and not fault tolerant [5].

PVM provides a traditional RPC approach to IPC. It is dynamic, self-healing, and fault tolerant [5].

Both PVM and CORBA have an advantage of being able to operate over a heterogeneous network of processors *and* processes [5]. That is, portions of a distributed-application may be implemented in different programming languages. However, because they facilitate IPC between heterogeneous processes, they must perform low-level message translation every time a RPC is invoked between such processes [5]. This behavior makes IPC with CORBA or PVM much slower than IPC with Java RMI [5].

[7] Our present distributed-applications are able to operate over a network of heterogeneous processors because they are implemented in the Java programming language [5].

Furthermore, because both PVM and CORBA present a traditional RPC approach to IPC, complex data-structures such as Java objects, especially *live* references to local and remote objects, cannot be passed to remote procedures [5]. Likewise, only primitive data-types can be returned from a RPC [5].

In addition, if the use of CORBA is desired, Java RMI can be used in conjunction with CORBA via its Internet Inter-ORB Protocol (IIOP) bindings [7]. These bindings enable developers to use Java RMI for IPC—as if they were solely using Java RMI—whilst it transparently communicates with CORBA behind the scenes [7].

3 Description

This section, intended for decision makers, describes the purpose of and procedure by which the proposed approach will be implemented. Please refer to section 1 on page 4, or section 2 on page 7, for a more thorough discussion of the problem and the proposed approach.

3.1 Problem

Our distributed-applications presently employ a low-level approach to IPC, which burdens our programmers with protocol-design and -synchronization issues that are unimportant to the business logic they implement.

3.1.1 Assumptions

- Our distributed-applications are implemented using version 1.4 of the Java programming language.
- Our developers have implemented custom low-level IPC for our distributed-applications.
- Our developers have substantial experience in decomposing high-level IPC into low-level IPC.

3.2 Approach

> "If I have seen a little further it is by standing on the shoulders of Giants." —Sir Issac Newton [8]

Employ an existing, proved, reliable technology—Java RMI in this case—to simplify IPC so that our developers are better able to focus on implementing business logic instead of re-inventing low-level IPC for every distributed-application we create.

3.2.1 Appropriateness

Since our existing distributed-applications are implemented using the Java programming language, it is only natural to employ a high-level approach to IPC provided by the Java programming language itself. Java RMI seamlessly encapsulates low-level IPC and presents it as a regular method invocation [6]. Furthermore, Java RMI facilitates IPC for the J2EE multi-tiered, enterprise application framework and the highly dynamic Jini technology used in ad-hoc wireless and mobile networks [5].

3.2.2 Implementation

First, our developers must rethink how IPC functions in our distributed-applications in high-level terms. Once the syntax and semantics of interactions between distributed objects have been determined, our developers must write source-code for the stubs of these distributed objects. Next, the services defined by these stubs are implemented in the source-code of their associated skeletons. Finally, these skeletons are instantiated, registered with a central RMI naming service, and ready for use.

3.2.3 Evaluation

A distributed-application prototype utilizing Java RMI for its IPC will be constructed by our developers for evaluation purposes. Its reliability, development cost, and development time will be assessed, analyzed, and compared with those of our existing distributed-applications, which were implemented using custom, low-level IPC.

4 Plan

This section, intended for decision makers, describes plans for realizing[8] and evaluating the proposed approach. Please refer to section 3 on the previous page for a brief description of the proposed approach, its objectives, evaluation, and implementation.

4.1 Achieving Objectives

In order to utilize Java RMI masterfully, our developers will need to be trained in its use. Because our developers are highly skilled in implementing low-level IPC, they can readily learn and incorporate techniques from well written textbooks on Java RMI, without the assistance of an external consultant.

[8] Bringing into existence.

4.2 Evaluation

We can evaluate the effectiveness of Java RMI in reducing our distributed-applications' development time, and total monetary expenditure whilst improving its reliability by having our developers construct a distributed-application prototype which utilizes Java RMI for its IPC. The prototype's quality, reliability, and the time and cost required for its development will be assessed, analyzed, and compared with those of our past distributed-applications.

4.3 Schedule

Table 1 shows a possible schedule for implementation and evaluation of the aforementioned distributed-application prototype. The debugging phase of this schedule is allotted two weeks because our developers are not yet experienced in debugging a distributed-application which utilizes Java RMI for its IPC.

Table 1: Schedule for implementation and evaluation of a distributed-application prototype which utilizes Java RMI for its IPC.

Description	Weeks Required	Total Weeks
Learn Java RMI	1	1
Design prototype	1	2
Implement prototype	1	3
Debug prototype	2	5
Evaluate prototype	1	6

5 Necessities

This section, intended for decision makers, describes materials, monies, and persons necessary for realizing the plans outlined in section 4 on the previous page.

5.1 Equipment

Several copies of well written text-books on Java RMI are required. The requested funds on the first page of this process-change proposal gives a rough estimate of the cost of these text-books. Other than text-books, equipment in excess of that possessed by our developers, such as workstations, the Java 1.4 compiler, etc. are not required.

5.2 Personnel

Our developers will learn how to use Java RMI; understand its underlying theory, motivation, and design techniques; and gain first hand experience in its

utilization by creating the aforementioned distributed-application prototype. During evaluation of this prototype, senior IPC design architects or consultants, in addition to our present distributed-applications' developers, can be invoked for their years of experience and keen insight may prove well in technically evaluating the prototype's particular utilization of Java RMI.

5.3 Budget

To purchase two copies of several well written text-books on Java RMI, the amount requested on the first page of this proposal will be required. These text-books would prove to be a useful return on investment upon successful deployment of our future distributed-applications using Java RMI, for our company may see increased profits as it releases better products, faster than its competitors. Nevertheless, from an educational viewpoint, these books will serve to enrich the knowledge of developers and researchers throughout our company.

6 Appendix

6.1 Remote Procedure Call (RPC)

RPC enables one process to engage in IPC with another by calling a procedure (or invoking a method) on the remote process. In addition, a RPC accepts primitive data-types as arguments and automatically retrieves the return value of the remote procedure. For example, if process A wishes to have process B return the sum of two numbers, it would simply call the appropriate procedure on process B as shown in Algorithm 10.

References

[1] A. S. Tanenbaum, Computer Networks, 4^{th} ed., New Delhi, India: Prentice Hall of India, 2003.

[2] F. Mattern and P. Sturm, "From Distributed Systems to Ubiquitous Computing – The State of the Art, Trends, and Prospects of Future Networked Systems," presented at Kommunikation in Verteilten Systemen (KiVS), Leipzig, Germany, 2003.

[3] Mind Tools, "Project Risk Analysis Techniques," [Online document], 2005 Apr 5, [cited 9 Apr 2005], Available HTTP: `http://www.mindtools.com/pages/article/newTMC_07.htm`

[4] SBC Knowledge Ventures, L.P. "SBC California White and Yellow Pages," [Online document], 2005 Mar 14, [cited 2005 Mar 14], Available HTTP: `http://www.sbc.com/gen/general?pid=3911`

[5] S. N. Kurapati, "A Brief Survey of High-Level Approaches to Implementing Distributed Applications," [Online document], 2005 Feb 23, [cited 11 Mar 2005], Available HTTP: `http://people.ucsc.edu/~skurapat/ce185/survey_article.html`

[6] Sun Microsystems, Inc. "Java^{TM} Remote Method Invocation Specification," [Online document], 2003 Dec 11, [cited 6 Feb 2005], Available HTTP: `http://java.sun.com/j2se/1.5/pdf/rmi-spec-1.5.0.pdf`

[7] Sun Microsystems, Inc. "Java^{TM} RMI over IIOP," [Online document], 2004 Aug 12, [cited 13 Mar 2005], Available HTTP: `http://java.sun.com/j2se/1.5.0/docs/guide/rmi-iiop/index.html`

[8] Wikimedia Foundation, Inc. "Issac Newton," [Online document], 2005 Mar 12, [cited 2005 Mar 13], Available HTTP: `http://en.wikipedia.org/wiki/Isaac_Newton`

Algorithm 1 Java source-code for the `FooBarCompany` object, which serves as a gateway to our company's dedicated RMI server (the `rmiregistry` daemon). It creates and hosts a `java.rmi.registry.Registry` object, which represents the naming service provided by our company's dedicated RMI server.

```
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

final class FooBarCompany
{
    // network address or hostname of our company's dedicated RMI server
    static final String RMI_SERVER_ADDRESS = "127.0.0.1";

    private static Registry _registry = null;

    /**
        Initializes a reference to our company's dedicated RMI server.
    */
    static
    {
        try
        {
            _registry = LocateRegistry.getRegistry(RMI_SERVER_ADDRESS);
        }
        catch(RemoteException e)
        {
            System.err.println("Unable to initialize RMI naming
            service.");
        }
    }

    /**
        Returns a reference to our company's RMI naming service.
    */
    static Registry getRegistry()
    {
        return _registry;
    }

    private FooBarCompany() {}
}
```

Algorithm 2 Java source-code for the `EmployeeRemote` object, which is the stub associated with the `Employee` skeleton.

```
import java.rmi.Remote;
import java.rmi.RemoteException;

interface EmployeeRemote extends Remote
{
    /**
        Returns the identification number of this employee.
    */
    long getIdNumber() throws RemoteException;
}
```

Algorithm 3 Java source-code for the `Employee` object, which is the skeleton associated with the `EmployeeRemote` stub.

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

class Employee extends UnicastRemoteObject implements EmployeeRemote
{
    private long _idNumber;

    Employee() throws RemoteException
    {
        // ...
    }

    public long getIdNumber() throws RemoteException
    {
        return _idNumber;
    }
}
```

Algorithm 4 Java source-code for the `EmployeeDatabaseRemote` object, which is the stub associated with the `EmployeeDatabase` skeleton.

```
import java.rmi.Remote;
import java.rmi.RemoteException;

interface EmployeeDatabaseRemote extends Remote
{
    static final String RMI_REGISTRY_NAME =
    "/human_resources/EmployeeDatabase";

    /**
        Thrown when an employee is not found in the employee database.
    */
    class EmployeeNotFoundException extends Exception {}

    /**
        Returns a reference to an employee associated with the specified
        identification number.
    */
    EmployeeRemote getEmployeeById(long idNumber) throws RemoteException,
    EmployeeNotFoundException;
}
```

Algorithm 5 Java source-code for the `EmployeeDatabase` object, which is the skeleton associated with the `EmployeeDatabaseRemote` stub.

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

class EmployeeDatabase extends UnicastRemoteObject implements
EmployeeDatabaseRemote
{
    EmployeeDatabase() throws RemoteException
    {
        // register this database with our company's RMI naming service
        FooBarCompany.getRegistry().rebind(RMI_REGISTRY_NAME, this);
    }

    public EmployeeRemote getEmployeeById(long idNumber) throws
    RemoteException, EmployeeNotFoundException
    {
        // ...
    }
}
```

Algorithm 6 Java source-code for the `PayrollDatabaseRemote` object, which is the stub associated with the `PayrollDatabase` skeleton.

```
import java.rmi.Remote;
import java.rmi.RemoteException;

interface PayrollDatabaseRemote extends Remote
{
    static final String RMI_REGISTRY_NAME =
    "/human_resources/payroll/PayrollDatabase";

    /**
        Thrown when a monetary bonus specifies an invalid amount of money.
    **/
    class InvalidBonusAmountException extends Exception
    {
        InvalidBonusAmountException(String reason)
        {
            super(reason);
        }
    }

    /**
        Issues a monetary bonus of the specified amount to the specified
        employee. Also, the bonus is marked as being issued by the
        given director.
    */
    void issueBonus(HRDirector issuer, EmployeeRemote receiver, double
    amount) throws RemoteException, InvalidBonusAmountException;
}
```

Algorithm 7 Java source-code for the `PayrollDatabase` object, which is the skeleton associated with the `PayrollDatabaseRemote` stub.

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

class PayrollDatabase extends UnicastRemoteObject implements
PayrollDatabaseRemote
{
    PayrollDatabase() throws RemoteException
    {
        // register this database with our company's RMI naming service
        FooBarCompany.getRegistry().rebind(RMI_REGISTRY_NAME, this);
    }

    public void issueBonus(HRDirector issuer, EmployeeRemote receiver,
    double amount) throws RemoteException, InvalidBonusAmountException
    {
        if(amount < 0)
            throw new InvalidBonusAmountException("Amount is negative.");

        else if(Double.isNaN(amount))
            throw new InvalidBonusAmountException("Amount is not a number
            (NaN).");

        else if(Double.isInfinite(amount))
            throw new InvalidBonusAmountException("Amount is infinite.");

        // ...
    }
}
```

Algorithm 8 Java source-code for the `HRDirector` object, which serves our Human Resources director. Notice that Java RMI passes this object by reference rather than by value to the remote method `_payrollDb.issueBonus` because this object is a sub-class of `java.rmi.server.UnicastRemoteObject`.

```
import java.rmi.NotBoundException;
import java.rmi.RemoteException;

class HRDirector
{
    EmployeeDatabaseRemote _employeeDb = null;
    PayrollDatabaseRemote _payrollDb = null;

    HRDirector() throws RemoteException
    {
        try // fetch references to Employee and Payroll databases
        {
            _employeeDb = (EmployeeDatabaseRemote)
            FooBarCompany.getRegistry().lookup(EmployeeDatabaseRemote.RMI_REGISTRY_NAME);

            _payrollDb = (PayrollDatabaseRemote)
            FooBarCompany.getRegistry().lookup(PayrollDatabaseRemote.RMI_REGISTRY_NAME);
        }
        catch(NotBoundException e)
        {
            System.err.println("Unable to fetch references to Employee
            and Payroll databases because: "+ e);
        }
    }

    /**
        Issues a monetary bonus of the specified amount to employees
        associated with the specified identification numbers.
    */
    void issueBonus(double amount, long... employeeIdNumbers)
    {
        for(long id : employeeIdNumbers)
        {
            try
            {
                EmployeeRemote employee = _employeeDb.getEmployeeById(id);

                try
                {
                    _payrollDb.issueBonus(this, employee, amount);
                }
                catch(RemoteException e)
                {
                    System.err.println("Unable to issue bonus to "+
                    employee +" because: "+ e);
                }
            }
            catch(RemoteException e)
            {
                System.err.println("Unable to query the employee database
                because: "+ e);
            }
            catch(EmployeeDatabaseRemote.EmployeeNotFoundException e)
            {
                System.err.println("The employee with identification
                number "+ id +" was not found in the employee database
                because: "+ e);
            }
            catch(PayrollDatabaseRemote.InvalidBonusAmountException e)
            {
                System.err.println("The amount specified for the bonus
                is invalid because: "+ e);
            }
        }
    }
}
```

Algorithm 9 Java source-code which represents our director's actions in issuing a monetary bonus to several brilliant employees.

```
import java.rmi.RemoteException;

class Main
{
    public static void main(String args[])
    {
        try
        {
            HRDirector me = new HRDirector();
            me.issueBonus(100.00, 160, 230, 986);
        }
        catch(RemoteException e)
        {
            System.err.println("Unable to instantiate my HRDirector
            object because: "+ e);
        }
    }
}
```

Algorithm 10 An example of traditional RPC, written in pseudo-code akin to the Java programming language. Here, process A instructs process B to compute and return the sum of two specified numbers by calling process B's `addNumbers` procedure.

```
ProcessB b;
// ...
int number1 = 3;
int number2 = 5;
int sum = b.addNumbers(number1, number2);
```