

An Asynchronous Remote Method Invocation (ARMI) Mechanism for Java

Rajeev R. Raje* Joseph I. William†
Michael Boyles‡

Department of Computer and Information Science, Indiana University Purdue University Indianapolis, IN 46202.

Abstract

In recent past, Java has emerged as a powerful and an easy to use language for net-centric computing. Simplicity, object oriented features, a presence of threads and architecture independence are main reasons for the popularity of Java. Despite of the built-in threads in the language, the bare Java execution model does not support remote object invocations. RMI (Remote Method Invocation) is an interface specified by Sun Microsystems for the purpose of native Java-client and Java-server communication. Although, RMI is simple to use, it is not desirable in many applications due to its synchronous nature. In this paper, we describe ARMI (Asynchronous RMI), a mechanism which is built on top of RMI and allows concurrent execution of local and remote computations. This article presents the salient features and implementational details of ARMI. A few experiments performed with ARMI along with performance comparisons to RMI are also explained.

Keywords: RMI, RPC, Asynchronous, ARMI, Client, Server, Stub, Skeleton, Mailbox.

1 Introduction

In past two years, a new programming language called Java, has captured the attention of the Computer Industry. While Java was originally created for programming consumer electronics, it was soon realized by experts that it offered many features that would make it an ideal environment for web-centric computing[O'C95]. Some of these features are platform-independence, built-in security, object-orientation, and a rich set of APIs¹.

While all these features are a boon in any programming environment, Java's design principles have attracted the Scientific and High-performance communities to it. It has been suggested that the absence of pointers in Java, like Fortran, will allow a better compile-time analysis for the implementation of highly optimizing

*Email: rraje@cs.iupui.edu, Telephone: 317-274-5174.

†Email: jwilliam@cs.iupui.edu

‡Email: mboyles@cs.iupui.edu

¹Java offers toolkits for windowing, standard utility classes, networking, database management, and encryption. APIs for 2D and 3D graphics, electronic commerce, and extended multimedia facilities are currently being worked upon.

compilers. Java also has a built-in threading and synchronization model based upon Hoare's CSP/monitor ideas, that allows for an easy platform-independent specification of parallelism[Car96].

The computational model of Java is *write once and run anywhere* [Ham96]. The ability of Java to create applets, which can run anywhere, indicates its coarse-grain parallelism model. On the other extreme, the presence of built-in threads allows fine-grain parallelism to be incorporated in Java programs. However, the basic model of Java does not include an ability to initiate computations in a remote address space. In order to overcome this deficiency, Sun has released a mechanism, called RMI(Remote Method Invocation), which allows a programmer to invoke methods on specified remote-objects.² The appearance of such a remote invocation is similar to a local (residing in the same address space) invocation. RMI uses object serialization mechanism to handle the necessary marshaling/unmarshaling details [Sun96a, Sun96b, Hor96, Orf97]. RMI allows an easy implementation of client/server applications, which are prominent in business domains. While RMI is easy to use, one drawback is that, it invokes the remote-methods synchronously, i.e., when the client invokes the method, it must wait for the server to respond with the result before it can perform any other actions. While this is satisfactory in many applications where the execution time is not critical, in many other cases it is necessary to overlap remote computations with local computations. Such an overlap promises to reduce the overall computation time.

This paper describes an Asynchronous RMI (ARMI) paradigm for Java, in which, the execution of remote methods is overlapped with execution of local methods, thus, allowing the client to perform other work while the remote-object carries out the remote method invocation. The goal of ARMI is not to provide a completely new framework which provides asynchronous communication, but to build upon the existing RMI architecture. While this may restrict our design choice, it allows for a low learning curve, while providing the benefits of asynchronous communication.

The rest of the paper is organized as follows:

Section 2 provides a brief review of related work. In section 3, we describe the RMI model along with the principles and implementational details of ARMI. Section 4 presents various tests performed using ARMI along with the results of our experiments. We conclude the paper with experiences learned and suggest possible extensions. The detailed APIs of various ARMI components are presented at the end as an appendix.

Throughout this paper, we will refer to the program which invokes the remote method as the client, and the object which resides on the remote machine as the server. We will denote Sun's synchronous remote method invocation mechanism by the term RMI, and we refer to the asynchronous RMI designed by us as ARMI.

2 Related Work

2.1 RPC and Asynchronous RPC

RPC (Remote Procedure Call) offers an alternative to traditional socket communication in a distributed system. It provides an abstraction at the level of a procedure call. The remote call has a similar appearance of a local call. The basic RPC model is synchronous in nature. Various attempts have been made to add an asynchronous flavor to RPCs. In [Cou94], a brief overview of asynchronous RPC (Remote Procedure Call) is provided. They have described examples which illustrate the following benefits of asynchronous RPC paradigm:

- When a client does not expect a reply message, it can make the remote call and proceed without waiting for the server to complete the operation.

²RMI has become an integral part of JDK 1.1 onwards.

- When a reply is not required, several client messages can be buffered together and transmitted together.
- Even when a reply is necessary, a client may benefit if it is able to make the remote call that does not wait for the reply and the client claims the reply later.

Distributed applications such as X-11 window system incorporates asynchronous RPC mechanism [Dav92]. Many of the calls in the X-11 interface are asynchronous.

The Mercury communication system developed at MIT combines asynchronous and synchronous RPCs with and without replies in one single communication facility called a *call-stream* [Cou94, Lis88]. These different forms of the communication are treated in a similar way by the servers – they just receive invocations and return results. This approach allows the server design to be uninfluenced by the method of communication used by their clients. The clients can choose either the synchronous or the asynchronous mode of request to the servers. Three different forms of requests are available to the clients:

- Synchronous RPC
- Asynchronous RPCs without replies
- Asynchronous RPCs with replies

Liskov and Shriram [Lis88] have proposed a new data type called a *promise* to support asynchronous RPCs. Promises allow the clients to continue with other work during a call and subsequently to pick up results and exceptions from the call.

The Nexus Communication system, developed by Argonne National Laboratory [Fos94] supports asynchronous communication. The system is intended as a target for parallel compilers. As such, it provides a flexible framework, at the expense of verbosity. Nexus is based around two concepts, Global Pointers (GPs) and Remote Service Requests (RSRs). GPs represent communication end-points, and RSRs are a way of initiating a computation on these end-points. A RSR takes a GP, a procedure name, and data. It then transfers the data to the context of the given GP, and invokes the specified procedure within that context. Nexus provides the user the option of having a new thread being created within the GP, or using a pre-existing thread. RSRs are fully asynchronous, but provide no way to specify return-value. Nexus was originally written in C. The designers of Nexus have provided NexusJava which allows Java programs to communicate with other programs that use Nexus [Fos97].

ObjectSystems has recently provided a communication system called Voyager [Gla97]. Voyager provides facilities for remote-invocation, as well as the creation of mobile computing agents. Like Mercury, it provides several communication modes, allowing for Synchronous Invocation, Asynchronous Invocations with no reply (Oneway), and Asynchronous Invocation with a reply (Futures/Promise). The major difference between ARMI and Voyager is that Voyager aims at providing a completely new, more comprehensive system, thus making it incompatible with currently existing RMI technology.

2.2 RMI and ARMI

The RPC model, however, does not translate well into distributed object systems, where communication between program-level objects residing in different address spaces is needed [Sun96b]. In order to match the semantics of object invocation, distributed systems require RMI. In such systems, a local surrogate (stub) object manages the invocation on a remote object [Sun96b]. The principles of RMI are similar to those of RPC. However, the presence of inheritance, polymorphism and dynamic binding requires a different approach than the one used for RPCs. However, like RPC, RMI is also synchronous in nature.

As described at the beginning, ARMI provides a mechanism for overlapping the execution of local methods with the execution of remote methods. Such a concurrent execution, we believe, will reduce the overall time required to perform a task. Also, the advantages of the asynchronous RPC paradigm stated in previous subsection are inherent in ARMI. The design of ARMI is based upon RMI and hence, the user has the flexibility of selecting either of the two approaches for remote invocations. Also, both ARMI and RMI calls are handled in a transparent fashion by the server, thereby, allowing the server to be designed in a traditional manner.

It should be noted that, in this paper, we do not describe the RMI mechanism in detail, as many excellent sources such as [Hor96, Orf97, Sun96b] are available. However, as ARMI is built upon RMI, we will make brief references and comparisons to RMI as appropriate in following sections.

3 ARMI – Principles

In next sub-sections, we will discuss the important factors which effect the design of ARMI and our approaches to tackle these factors.

3.1 Return Values

In RMI, when a remote method is invoked, the client is suspended while the computation is carried out in a remote address space. When the computation has completed, the return value, if any, is immediately available to the caller for use. In the presence of an asynchronous method call, it is neither possible to predict when the result will be available nor when the caller will require it. These constraints necessitate another mechanism for returning values back to the client during an asynchronous method invocation.

We have used the concept of a *mailbox*. The client must register a mailbox with a remote object before any asynchronous RMIs can occur. The client is the *owner* of that mailbox. When the client calls an asynchronous remote-method, the method returns a *receipt*³ generated by the mailbox. When the method has finished its computation, ARMI inserts the return value into the mailbox, using the receipt as a key. The client can then query the mailbox at a later time to retrieve the value associated with the receipt. Figure 1 shows an overview of ARMI mechanism. Along with the client, the server and the mailbox, figure 1 also indicates a stub, a skeleton and a registry.⁴

The mailbox provides the ability for the client *to check if certain values are available, to retrieve a value associated with a specific receipt, to retrieve the next value inserted into the mailbox and to wait until a specific value is available.*⁵

3.2 Exception Handling

We now describe the mechanism to handle exceptions from the remote computation to the client, should they occur. RMI allows exceptions to be handled using standard Java exception handling techniques⁶, because it does not change the semantics of a method call. In the presence of an asynchronous call, it is not possible

³A receipt is an object of a class *Receipt*, which supports *wait* and *notify* methods. The interface of Receipt class is attached at the end.

⁴The stub, the skeleton and the registry achieve the same functions as their counterparts in RMI. The detailed functions of these components can be found in [Orf97, Hor96].

⁵The Mailbox API included in the appendix has the details of these methods.

⁶Java's exception handling mechanism is described in [Hor96, Fla96].

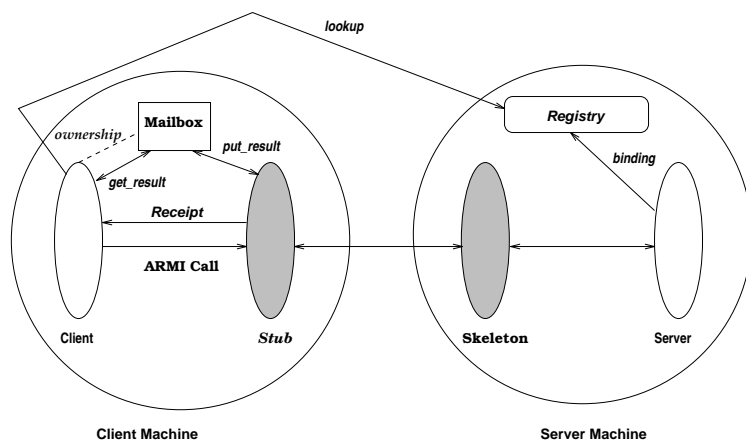


Figure 1: ARMI Mechanism

to predict if the original method will be still running when the exception occurs, or even if the original client thread will still be alive. To effectively tackle this issue, we have designed following two mechanisms which allow the delivery of exceptions.

3.2.1 Delayed Delivery Mechanism (DDM)

The first alternative, DDM, delivers exceptions upon the retrieval of the return-argument. When an exception occurs in the computation, the underlying ARMI system catches it and inserts it into the mailbox in place of the normal return-value. When an attempt is made to retrieve the return-value, the mailbox realizes an exception was stored and throws it. The advantages of using this approach is the simplicity in which it can be implemented and the use of familiar Java *try/catch* mechanism to handle the exceptions. A disadvantage of this approach is that the method which makes the original call may not want the method that gets the result to be concerned about possible exceptions that could happen during the computation.

3.2.2 Callback Mechanism (CM)

The second alternative provided by ARMI is a callback mechanism. The client can register an (*exception-type*, *exception-handler*) pair with the mailbox. Whenever an exception is delivered to the mailbox, it checks to see if the type of the exception matches the exception-type field of any of the pairs. If a match happens, then the exception-handler is called with the exception as an argument so the appropriate actions can occur. This solves the problem associated with DDM, by allowing the caller to specify an exception handler.

Both mechanisms suffer from one major disadvantage. Java forces all methods to either catch the exceptions that may occur in their body, or to explicitly state that they might throw an exception in the method header. These ensure that the programmer knows of all possible exceptions that may happen during the computation. The two alternatives described here both violate this behavior, making it possible for a method to throw an exception without the programmer knowing about it. While this is an area of further study, we predict the problem will not be solvable without appropriate modifications to the grammar/compiler.

3.3 Implementation

As ARMI adds the asynchronous flavor to RMI, it is logical that the implementation of ARMI be based upon the underlying communication facilities provided by RMI. In this sub-section, we discuss the modifications needed to support the asynchronous mode of communication.

3.3.1 RMI

If a programmer wishes to allow for instances of his/her class to be accessible from a remote machine, he/she must first create an interface for the class. The interface specifies which methods on the class will be available to a client. He/she must then run the program *rmic* on the bytecode of that class. Two new classes called the *stub* and *skeleton* are generated by *rmic*. These classes are described in the following paragraphs. The server code must also register itself with a *registry* object. The server tells the registry object that it is now available to offer its services under a given name.

When a client program wishes to interact with a remote server object, it must first *bind* to the remote server object. The binding involves contacting a Naming service (provided by Sun as a part of RMI) and providing the name of the remote server object, in the form of an URL. The naming service then contacts a registry object on the remote machine. The registry object ensures that the remote server object exists and then returns a stub object back to the client.

The client then assigns an instance of the server-defined interface class to this stub. It can then use this interface to invoke methods on the stub. The stub resides on the client's machine and forwards all incoming method invocations to the respective remote server object.

On the server's machine the RMI mechanism intercepts this remote method invocation. It creates a new thread, whose responsibility is to call the correct method on the skeleton object. The skeleton unmarshals all of the arguments, and then invokes the method on the server object. When this is completed, the return value is carried back to the client and computation is continued.

The flow of control during the execution of a remote-method call is:

Client \longleftrightarrow Interface \longleftrightarrow Stub \longleftrightarrow Skeleton \longleftrightarrow Server

More details about RMI can be found in [Orf97, Hor96].

3.3.2 Modifications Necessary for ARMI

As the server object is already threaded (through facilities provided by RMI), it can process multiple requests concurrently. Hence, in order to achieve asynchronous communication, we need to thread the client-side of the connection. We do this by replacing the RMI stub object by our specialized stub.

The server programmer runs our stub generator, *armic*, instead of *rmic* on the server class bytecode. The appropriate stub object and interfaces are generated by *armic*. This process is achieved by traversing the interface hierarchy of the server class, locating those interfaces which are descendents of the *Remote* interface. Once such interfaces are known, *armic* generates corresponding asynchronous interfaces. For each method defined in the original interface a corresponding asynchronous method is defined. These new methods have the same name as the original, with an 'a' prepended. When the client interacts with the remote object, it uses these asynchronous interfaces. These asynchronous interfaces inherit from the originals, so that both asynchronous and synchronous methods are available for the client. Figure 2 shows an example interface the server-class programmer might code, and the resulting asynchronous interface.

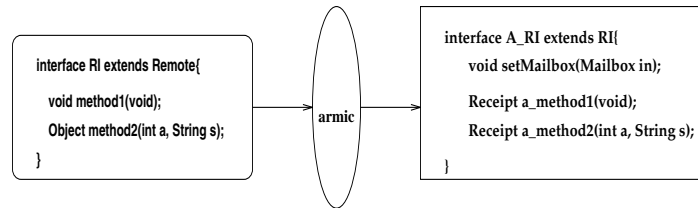


Figure 2: Example Interface and Resulting Asynchronous Interface

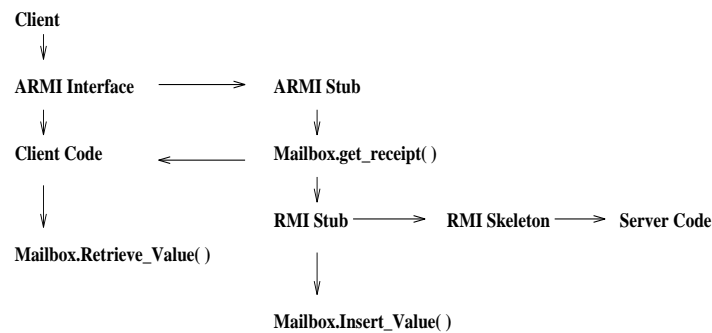


Figure 3: An Execution Sequence in ARMI.

Along with the interfaces, *armic* generates the stub class. It implements the interfaces mentioned in the previous paragraph. When a client makes a call to an asynchronous method, the method creates a thread on the client-side to handle the request, and then returns control back to the client. This thread is responsible for contacting the remote host, through RMI, and ensuring the return-value is inserted into the mailbox. Since both the client and server are threaded, the call is carried out in an asynchronous fashion. Figure 3 shows how the flow of control during an asynchronous method invocation. In figure 3 vertical lines show the sequence of execution within a thread and horizontal lines indicate an expansion of a method call.

Prior to making an asynchronous call, the client needs to contact our *AsyncNaming* class instead of the one provided by RMI. The *AsyncNaming* class first contacts the remote registry object and retrieves the RMI stub for the requested remote server object. *AsyncNaming* then loads the appropriate asynchronous stub, generated by *armic*, and returns the asynchronous stub back to the client. The RMI stub that was retrieved by *AsyncNaming* is used by the asynchronous stub to forward the methods invocations to the remote machine.

4 Experiments and Results

In order to test the performance of ARMI implementation, we performed three experiments. These were conducted over a network of Sun Sparc-4 workstations running Solaris operating system. We used JDK 1.1 as

	RMI	ARMI(Wait)	ARMI(No Wait)
Method-Invocation Times(Milliseconds)	8	10.5	3

Table 1: Amount of overhead incurred by using RMI, ARMI(with wait) and ARMI (with no wait)

the Java development environment. The machines on which we performed our tests, did not have any other applications running on them simultaneously.

4.1 Invocation Overhead

The first test was to determine how much overhead would be incurred by the use of ARMI. In this test, we performed 10000 calls to an empty-method sitting on a remote server and computed the average time it took, in seconds, for the call to complete. We performed the test using RMI, using ARMI with a wait for a confirmation that the call had completed and using ARMI with no wait for the confirmation. Table 1 contains the results.

The results are as expected. The increase in the cost of ARMI is due to two factors. The first is that at the beginning of the call a new thread must be created on the client side. Secondly, the retrieval and insertion of the return value into the mailbox involves the invocation of several synchronized methods, as well as the use of various hash-tables. This will naturally have more overhead than the use of the inbuilt stack when returning values.

4.2 Numerical Computation

The next test was to implement an embarrassingly-parallel algorithm using ARMI and RMI to see how much speedup could be gained by overlapping the computations. We chose a simple matrix-vector multiplication as an example, where the work was split up among two processors. The matrix was a square matrix of size 1000.

We performed the test multiple times, each time sending a sub-matrix (number of rows starting from 100 upto 900) of the original matrix to the remote machine. We also modified the program to ignore the initial overhead of transmitting the arrays. We measured the total time required to perform both the local and remote computations. The remote computations involve transmitting the vector, performing the computation, and returning the resulting vector. The results are displayed in figure 4. The y-axis in figure 4 indicates the total time (in seconds) required for the computation (local + remote) and the x-axis indicates the number of rows sent to the remote machine.

Figure 4 indicates that when the ratio of computation (proportional to the size of the local sub-matrix) to communication time (as decided by the size of the sub-matrix sent to the remote machine) is greater than or equal to 1, ARMI is able to benefit from the use of parallelism yielding increased speedups. At the optimal partition of 600 rows being computed locally, 400 rows computed remotely ARMI is able to achieve a 30% speedup over RMI. The cause for the similarity in times near the end-points is that if most of the computation is performed on a single processor (local or remote), very little computation can be overlapped.

4.3 D-SIFTER – Information Filtering Application

As the third test, we decided to use ARMI on an existing real-world application from the domain of information filtering. We, at Indiana University Purdue University Indianapolis, have developed a collaborative information

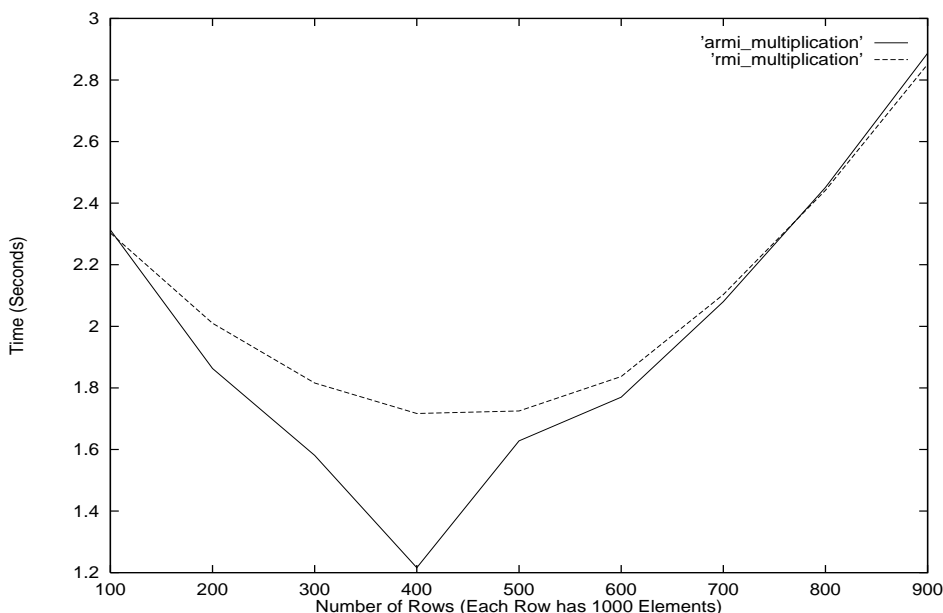


Figure 4: RMI and ARMI Performance for Matrix-Vector Multiplication

filtering environment called D-SIFTER. The original version of D-SIFTER was developed using Java and RMI. Hence, it was an obvious choice to create a new prototype based upon ARMI and compare the performance.

Information filtering is a technique used to classify, sort, and present documents to the user according to that particular user’s interests. Each user is serviced by an agent, which performs the tasks of classifying, sorting and presenting the incoming documents. However, the classifying ability of each agent is limited by its *knowledge-base*. This knowledge-base (stored as a thesaurus of keywords) is entered by the user and/or can be constantly updated. The agent makes an attempt to locally classify each incoming document based upon its thesaurus. If the document can be classified then it is presented to the user by the agent. If the document cannot be classified (due to limited size of the thesaurus), the agent initiates a collaborative action, by requesting other agents to help in the classification. The results (in form of remote classifications) are sent back to the original agent (one who initiated the collaboration) and then displayed to the user. More details about D-SIFTER are in [Raj97].

4.3.1 Experimental Setup

Although, D-SIFTER allows many agents to be involved in the collaboration, for this test, we used only two agents – one acting as a client and the other as a server. The client was made to classify 1064 documents from the domain of Computer Science. Based on its thesaurus, the client was able to classify 676 documents locally (not requiring any remote calls) and 388 documents had to be classified with the help of the server.

We conducted this experiment with two versions of D-SIFTER, one using RMI and the other using ARMI. A significant percentage of the documents (388 out of 1064) had to be classified remotely and hence, there were 388 remote calls. In the RMI version, the client was blocked during each of these 388 calls. On the other hand, in the ARMI version, the client could overlap these 388 remote classifications with 676 local classifications, thereby reducing the total time required for the classification of 1064 documents. The RMI version required 89.7 minutes to classify 1064 documents, while the ARMI version classified 1064 documents in 75.6 minutes, thus saving 13.9 minutes.

This performance improvement shown by the ARMI version of D-SIFTER is hardly a surprise. The information filtering application has inherent parallelism in it.⁷ Such an application will benefit from the the use of ARMI mechanism.

5 Future Work and Conclusion

5.1 Future Work

Various exciting extensions are possible from our current work. Several client messages can be buffered and transmitted together to the server. Also, if a task requires multiple agents to perform the same action on the same set of data, ARMI can be extended to communicate to a collection or a colony of similar agents. Collaborative tasks such as information filtering (where same document needs to be classified by multiple remote experts) can benefit from the collection concept. More comprehensive exception handling schemes is another area for future research. Use of ARMI to various scientific and engineering domains along with performance assessment are other future directions.

5.2 Conclusion

By concurrently executing the local and remote invocations, ARMI allows the users to reduce the overall computation time of their application. Many distributed applications (which either have inherent parallelism and/or need to repeat overlapped actions) will benefit from the ARMI. As ARMI is based upon RMI, it provides the user the flexibility of choosing either of the options and also achieves reuse of the low-level communication mechanism offered by RMI. Both ARMI and RMI present a similar appearance to the server, thereby simplifying its design. In this paper, we have described three such experiments, which emphasize the advantages of ARMI over RMI. Even though, our tests did not directly address high-performance computation applications, we believe that other scientific domains such as distributed simulations and visualizations will benefit from the use of the ARMI mechanism.

A API's

A.1 Callback API

```
interface RMI_Callback:
```

```
    Definition: This interface is used to specify a call-back function
                for the Mailbox. This function will be called when
                a particular piece of mail arrives.
```

```
void rmiCallback(Integer id, Object value):
    This function is called when the mail arrives. 'id' is the
    id of the message that arrived, and 'value' is the return-value.
```

⁷This observation is based on the fact that it is highly unlikely that the local thesaurus will be able to classify all the incoming documents!

A.2 Exception-handle API

interface RMI_ExceptionHandler:

Definition: This interface is used to define an exception handler that will be called when exceptions of a certain type are thrown during a remote-method invocation.

void handle(Exception e):

This function is called when an exception of type 'e' occurs.

A.3 Mailbox API

MailBox:

Definition: A mailbox is a way of storing (id, value) pairs, where 'id' is a unique message-ID, and 'value' is the return value of a RMI. The message-ID is an instance of the class Receipt.

Receipt getID():

Return a unique message-ID. This function is used mainly by the generated code and won't be used by the user. The generated server code will associate a message-ID with a particular result value, and return the id back to the caller.

Receipt waitForMail():

Wait until any return value arrives and then return. Returns the message-ID associated with the return value that arrived.

void waitForMessage(Receipt id):

Wait until the return-value associated with the message-ID 'id' arrives.

Object getMessage(Receipt id, boolean remove):

Return the value associated with the message-ID 'id'. If the value isn't available yet, block until one the message is available. If 'remove' is true, then the message will be removed from the mailbox and further attempts to retrieve it will be an error.

void setCallback(Receipt id, RMI_Callback f):

When the return-value associated the message-ID 'id' arrives, call the function 'f' to handle it. 'f' is passed the message-ID and the return-value of the call.

boolean removeCallback(Receipt id):

If a callback is currently registered to handle the return-value

with the message-ID 'id', remove the callback and return true.
If no callback is registered for that id, return false.

```
void setExceptionHandler(Class e, RMI_ExceptionHandler h):
```

If an exception derived from class 'e' is thrown by any remote-computation that is using this mailbox, call exception-handler 'h' with the exception as the argument.

```
boolean removeExceptionHandler(Class e):
```

If an exception handler is registered for the class 'e', remove the exception handler and return true. If no handler was is registered for that class, return false.

```
void insertMessage(Receipt id, Object value):
```

Associate the return-value 'value' with the pre-allocated message id 'id'. If an id equal to 'id' hasn't been assigned by getId(), signal an error.

```
boolean messageReady(Receipt id):
```

Return true of the return-value associated with the message-ID 'id' has arrived, false otherwise.

A.4 Interface for Receipt

```
class Receipt {
    synchronized void mWait();
        Block while waiting for a specific value associated with a receipt.

    synchronized void mNotifyAll();
        Unblock any clients waiting for a specific value of a receipt.
}
```

References

- [Car96] Carpenter, B., Chang, Y., Fox, G., Leskiw, D. and Li, X. Experiments with hpjava. Unpublished paper, <http://www.npac.syr.edu/users/dbc/HPJava/experiments/>, 1996.
- [Cou94] Coulouris, G., Dollimore, J., and Kindberg, T. *Distributed Systems - Concepts and Design*. Addison Wesley Publication Company, 1994.
- [Dav92] Davison, A., Drake, K., Roberts, W. and Slater, M. *Distributed Window Systems, a Practical Guide to X11 and OpenWindows*. Workingham: Addison Wesley Publication Company, 1992.
- [Fla96] Flanagan, D. *Java in a Nutshell*. O'Reilly & Associates, 1996.

- [Fos94] Foster, I., Kesselman, K. and Tuecke, S. The Nexus Task-parallel Runtime System. *Proceedings of 1st International Workshop on Parallel Processing*, 1994.
- [Fos97] Foster, I. and Tuecke, S. Enabling Technologies for Web-Based Ubiquitous Supercomputing. *To appear in Proceedings of 5th IEEE Symposium in High Performance Distributed Computing*
URL:- <http://www.mcs.anl.gov/nexus/nexusjava.html>, 1997.
- [Gla97] Glass, G. Voyager: The New Face of Distributed Computing. *Object Magazine*
URL:- <http://www.sigs.com/publications/docs/objm/9706/9706.glass.html>, June, 1997.
- [Ham96] Hamilton, M. Java and the Shift to Net-Centric Computing. *IEEE Computer*, 29(8):31–39, August 1996.
- [Hor96] Horstman, C. and Cornell, G. *Core Java – Second Edition*. Sunsoft Press, 1996.
- [Lis88] Liskov, B. and Shrira, L. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *Proceedings of SIGPLAN’88 Conference Programming Language Design and Implementation*, 1988.
- [O’C95] M O’Connel. Java: The inside story. *Sunworld Online*, July 1995.
- [Orf97] Orfali R, and Harkey, D. *Client/Server Programming with JAVA and CORBA*. John Wiley & Sons, Inc., 1997.
- [Raj97] Raje, R., Mukhopadhyay, S., Boyles M., Patel, N. and Mostafa J. D-SIFTER – An Intelligent Collaborative Information Filtering Environment. Unpublished paper, Soon to be come a Technical Report, Department of Computer and Information Science, Indiana University Purdue University Indianapolis, 1997.
- [Sun96a] Sun Microsystems, Inc. *Java (TM) Object Serialization Specification*, 1996.
- [Sun96b] Sun Microsystems, Inc. *Java (TM) Remote Method Invocation Specification*, 1996.