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
Indianapolis, IN 46202, U.S.A.

Summary:
In recent past, Java has emerged as a powerful and easy to use language for net-centric computing. Simplicity, object oriented features, the presence of threads and architecture independence are main reasons for the popularity of Java. Despite having 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 implementation details of ARMI. Two experiments performed with ARMI along with performance comparisons to RMI are also explained.

1 Introduction

The computational model of Java is *write once and run anywhere* [1]. This basic model of Java does not provide the 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. The appearance of such a remote invocation is similar to a local (residing in the same address space) invocation. RMI uses an object serialization mechanism to handle the necessary marshaling/unmarshaling details [2, 3, 4, 5]. While RMI is easy to use, one drawback is that, it invokes the remote-methods synchronously, i.e., when the client invokes a method, it must wait for the server to respond with the result before it can perform any other actions. This is satisfactory in many applications where execution time is not critical, but it may be necessary in other cases to overlap remote computations with local computations. Such an overlap would promise 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 for an asynchronous communication, but to build upon the existing RMI architecture.

The rest of the paper is organized as: Section 2 provides a brief review of related work and the principles of ARMI. Section 3 presents experiments performed using ARMI along with their results. We conclude the paper with experiences learned and suggest possible extensions.

Throughout this paper, we will denote Sun's synchronous remote method invocation mechanism by the term RMI, and refer to the asynchronous RMI designed by us as ARMI.

2 Related Work and ARMI Principles

2.1 RPC and Asynchronous RPC

ObjectSystems has recently provided a communication system called Voyager [6]. Voyager provides facilities for remote-invocation, as well as the creation of mobile computing agents. 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.

There have been many efforts to introduce asynchronous facilities into programming languages and communication libraries. Examples of asynchronous RPCs may be found in [7], [8], [9]. Nexus Communication System, which is described in [10, 11], supports asynchronous invocations. It is intended as a target for parallel compilers. It is a flexible and a powerful framework, but unlike the ARMI programs, which are simple to write; programs written using Nexus are verbose and complex.

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

2.2 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 *future*. When the client calls an asynchronous remote-method, the method returns an instance of the class *Future*, created by a special stub object. When the method has finished its computation, the stub inserts the return value into the future. The client can query the future at any time in order to retrieve the return value. If it is not available, then the client will block. The future also provides an ability to determine whether the return value is available or not, without blocking the client.

2.3 Exception Handling

We now describe the mechanism to handle exceptions sent from the remote computation to the client, should they occur. RMI allows exceptions to be handled using standard Java exception handling techniques, because it does not change the semantics of a method call. In the presence of an asynchronous call, it is not possible to predict if the original method will be still running when the exception occurs, or even if the original client thread will remain alive. To effectively tackle this issue, we have designed following two mechanisms which allow the delivery of exceptions.

2.3.1 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 future in place of the normal return-value. When an attempt is made to retrieve the return-value, the future 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.

2.3.2 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 future. Whenever an exception is delivered to the future, it checks to see if the type of the exception matches the exception-type field of any of the pairs. If a match occurs, then the exception-handler is called with the exception as an argument so that appropriate actions can be carried out. 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.

2.4 Asynchronous Communication

When a server programmer uses RMI, the program *rmic* must be run on the classes which he/she wishes to make remotely accessible. When the programmer wishes to use ARMI, he/she instead calls *armic*. It generates stubs which handle the asynchronous communication.

These stubs provide asynchronous communication by wrapping each remote method invocation inside a thread. Since the server-side of the connection is already threaded, this allows the RMI to be handled without blocking, thus providing an asynchronous communication.

3 Experiments and Results

In order to test the performance of ARMI implementation, we performed two experiments. These were conducted over a network of Sun Sparc-4 workstations running Solaris operating system. We used JDK 1.1 as the Java development environment. The machines on which we performed our tests did not have any other applications simultaneously running on them.

3.1 Invocation Overhead

The first test was to determine the overhead which 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.

When RMI, ARMI(Wait) and ARMI(no wait) were used, it took 12, 14 and .878 milliseconds to perform the call, respectively.

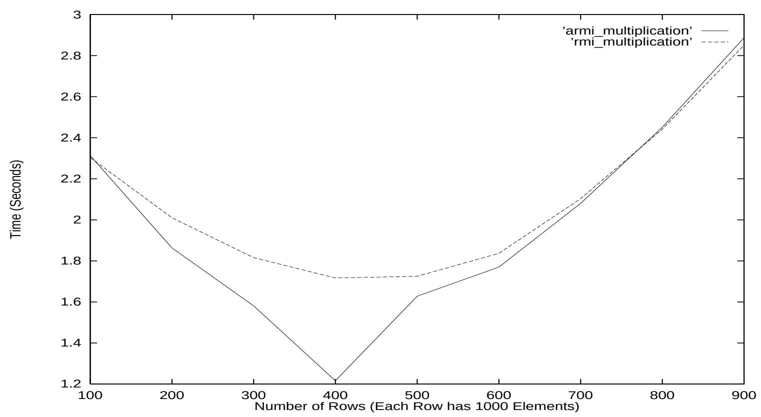

Figure 1: RMI and ARMI Performance for Matrix-Vector Multiplication

3.2 Numerical Computation

The next test was to implement an embarrassingly-parallel algorithm using ARMI and RMI to compute the speedup obtained by overlapping the local and remote 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 up-to 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 1. The y-axis in figure 1 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 1 indicates that when the ratio of computation 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 Future Work and Conclusion

Many exciting extensions are possible from our current work. 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 will benefit from the collection concept. Designing 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.

By concurrently executing the local and remote invocations, ARMI allows the users to reduce the overall computation time of their application. 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. Many distributed applications (which either have inherent parallelism and/or need to repeat overlapped actions) will benefit from ARMI.

References

[1] Hamilton, M. Java and the Shift to Net-Centric Computing. *IEEE Computer*, 29(8):31–39, August 1996.

[2] Sun Microsystems, Inc. *Java (TM) Object Serialization Specification*, 1996.

[3] Sun Microsystems, Inc. *Java (TM) Remote Method Invocation Specification*, 1996.

[4] Horstman, C. and Cornell, G. *Core Java – Second Edition*. Sunsoft Press, 1996.

[5] Orfali R, and Harkey, D. *Client/Server Programming with JAVA and CORBA*. John Wiley & Sons, Inc., 1997.

[6] 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.

[7] Coulouris, G., Dollimore, J., and Kindberg, T. *Distributed Systems – Concepts and Design*. Addison Wesley Publication Company, 1994.

[8] 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.

[9] 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.

[10] Foster, I., Kesselman, K. and Tuecke, S. The Nexus Task-parallel Runtime System. *Proceedings of 1st International Workshop on Parallel Processing*, 1994.

[11] 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.