Implementing Asynchronous Remote Method Invocation in Java

Article · August 1999

Source: CiteSeer

3 authors, including:

Katrina Falkner
University of Adelaide
91 PUBLICATIONS **472** CITATIONS
SEE PROFILE

Michael Oudshoorn
Wentworth Institute of Technology
90 PUBLICATIONS **407** CITATIONS
SEE PROFILE

Implementing Asynchronous Remote Method Invocation in Java

Katrina E. Kerry Falkner, Paul D. Coddington and Michael J. Oudshoorn
Distributed High Performance Computing Group
Department of Computer Science,
University of Adelaide,
Adelaide, SA, 5005, Australia
{katrina,paulc,michael}@cs.adelaide.edu.au

2 July 1999

Abstract

Java's Remote Method Invocation is an example of a synchronous communication mechanism with a well defined protocol. Many software systems require more flexibility in their communication mechanisms, including asynchronous communication and delayed referencing of objects (Futures). This paper introduces a novel mechanism allowing Java remote objects to use extended communication protocols without changes to the underlying wire or serialisation protocols. These extensions can be utilised by standard remote objects without additional coding changes and can be incorporated with standard Java clients.

This paper explores the possibilities of implementing client controlled versus server controlled asynchronous communication and dynamic selection of protocols through the use of a precompiler for the remote object classes. A discussion of the possibility of integrating Futures and Batched Futures, and any required programming abstractions, into this mechanism is conducted. It is proposed that this mechanism can be used in any object system that is based on Fragmented Objects, which use a stub or proxy to provide transparent access to remote server methods.

Keywords: Java, RMI, Asynchronous, Futures, Remote Method Invocation.

1 Introduction

Communication protocols in distributed object systems often do not explore the full range of possible optimisations due to the desire for simplicity and the maintenance of a consistent interface and underlying protocol. It is possible to create specialised implementations of these communications protocols, but generally these implementations are not completely compliant with the standard implementation. This leads to a reluctance on the part of the software developers to explore optimised or extended protocols.

Java's Remote Method Invocation (RMI) mechanism [5, 29] is an example of a standardised communications protocol that provides a limited set of optimisations. RMI provides a remote communications mechanism between Java clients and remote objects. Java clients obtain references to these remote objects through a third party registry service. These references allow transparent access to the remote object's methods by mirroring the remote interface. All method invocation is performed in a synchronous manner unless explicitly subverted by the client and server implementations.

Java RMI implementations are currently very slow on LAN networks, which is a problem for real-time and parallel distributed applications. Java RMI has been designed for Internet applications for which higher latencies are not as important as for real-time and parallel systems. Several parallel implementations of Java over RMI have isolated some of the problems with Java RMI, such as serialisation protocols and restrictive interfaces [13, 17].

Several alternative implementations have been developed that support extended protocols for RMI. These include JavaParty [21], Manta [17] and NinjaRMI [31]. However these implementations optimise the performance of RMI by changing the underlying protocols, such as, for example, the serialisation protocols [28]. Although dramatic performance increases can be achieved, it requires both parties in a remote communication to use the extended protocols and a non-standard RMI.

This paper proposes a mechanism for extending RMI to support asynchronous communication and delayed referencing of return objects (Futures [30]) through embedding threads into the communications protocol. This mechanism achieves significant performance increases while retaining compliance with the underlying RMI wire [29] and serialisation protocols.

Using this protocol extension mechanism a remote object server defines how its methods may be invoked. It defines which of its methods (on a per-method basis) may be invoked in a synchronous or asynchronous manner and if a remote object is non-void, whether its return object is a Future object. Delayed referencing, or Future objects, allow the client to obtain a reference to the object as soon as the asynchronous remote method returns, but delays the retrieval of the result from the server until the client explicitly accesses the value. This can result in optimisation in two ways: calculation of the result value can continue concurrently with client execution until the value is requested; and if the client is not interested in the return value of the method, but only in causing the method invocation itself, the Future value may never need to be retrieved.

In this paper, we discuss the implementation of such a mechanism in a distributed object system. Java RMI's use of a proxy or reference object on the client side to perform all communications is an example of the use of Fragmented Objects [18]. A fragmented object system, as seen in RPC [1], CORBA [24] and Java RMI, makes use of a stub object with identical method signatures to the server implementation. A fragmented object system is required to perform the extensions described here.

In Section 2, a more detailed discussion of the RMI mechanism and related work in the area of high performance RMI is presented. Section 3 presents an overview of our protocol extension mechanism and presents the architecture used in the system. Section 4 presents performance improvements possible using the system and a detailed discussion of any additional costs. In Section 5 a discussion of extensions to this work and conclusions is given.

2 Remote Method Invocation in Java

2.1 Java RMI

Java [5, 29] is an interpreted object-oriented language that supports distribution and distributed communication. Remote objects and Remote Method Invocation (RMI) [29] allow Java objects to perform distributed computation and communicate with each other. Remote objects are maintained through remote references, which take the form of a stub class. This stub class is responsible for protocol management and the actual communication required within the client/server model. Java provides additional distributed object services independent of RMI, such as the Messaging service, which provides asynchronous communication capabilities but is not compatible with Java RMI.

The Java Remote Method Invocation system is a three-tier communications protocol. Remote objects are defined through a remote interface, which can be exported to remote clients to abstractly define communications. A stub which defines how the communications protocol is to be used, in Java's case the RMI wire protocol, is generated statically from the implementation of the remote interface. A preprocessor, RMIC, is used to perform the stub generation after the completion of the interface and the server implementation.

Objects are passed between the client and server by marshaling the object into a serialised byte stream. This serialisation process can be performed dynamically using Java's reflective capabilities, or it can be tuned by the implementor for optimal results.

Communication between the client and remote server appears to be completely abstract. The client obtains a reference to the remote object through a remote registry lookup. It can then perform method invocations on the remote object as it would on a local object, since the stub or proxy that is returned appears to be exactly the same as the server implementation in its method signatures.

Figure 1 depicts the execution model for a Remote Method Invocation communication. A client must have access to, or dynamically download, the stub which will act as a reference to the remote server. The stub then controls how the method invocation will occur, i.e. which protocol is used.

Java implementations to date have suffered from performance problems due to their interpreted nature and unoptimised serialisation mechanism [6, 12]. Potential solutions, such as native compilation and Just-In-Time (JIT) compilation have been discussed in [15, 17]. These methods do provide performance improvements but do not tackle the fundamental performance problems and latency issues of Java RMI [21, 31].

To help alleviate the performance problems in RMI, a mechanism is proposed for introducing asynchronous communication and delayed referencing of return objects. To extend the supported protocols,

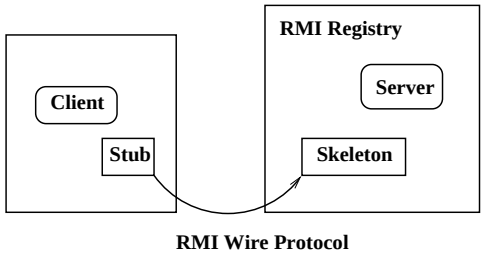

Figure 1: Execution Model for a Remote Method Invocation.

we extend the stub generation process to include support for the additional communication mechanisms. The new generation process is accessed in a similar way to the standard Java RMI generation process.

2.2 Related Work

There are several projects investigating fast implementations of RMI. The Manta project [17] is investigating a high performance implementation of RMI through native compilation and their own marshaling and serialization protocols. The Manta approach attempts to overcome the latency issues in Java RMI by performing as much compile time analysis of serialisation as possible in place of time consuming reflection. Manta also allows dynamic development of serialisation code for remote objects to support polymorphic operations. Manta has been able to achieve near RPC performance from their specialised RMI implementation, which is a significant cost improvement for Java RMI. Manta supports both standard Java RMI and its own serialisation protocols so, although they are not compliant at the underlying protocol level, it is possible to interoperate with standard Java RMI classes but with no performance increase.

The JavaParty project [21] adopts a similar aproach to Manta, using an efficient serialization protocol but without static native code compilation. The JavaParty implementation faces similar problems to that of the Manta project; these projects can be viewed as non-standard drop-in replacements for RMI.

Sampemane [27] and Java/DSM [25] describe implementations of JVMs and RMI on top of other distribution protocols. Sampemane describes an implementation of RMI above Fast Messages [26], while the Java/DSM project explores an implementation above Treadmarks [3]. Both of these methods maintain the flexiblity of RMI by retaining the standard API, however they share the interoperability problems of the Manta and JavaParty projects.

The NinjaRMI project [31], part of the UCSB Ninja [9] project, is a completely rebuilt version of RMI with extended features, including asynchronous communication. However, NinjaRMI is not wire compliant with standard Java RMI. NinjaRMI is not a high performance version of RMI but is intended to provide language extensions and exhibits performance at least as good as Sun's implementation.

The HORB project [11, 12] introduces a Java ORB with an alternative communications protocol to RMI that supports both synchronous and asynchronous communication. Using the HORB communications mechanisms, performance close to that of C sockets is achieved, however HORB supports a completely different wire protocol to standard Java RMI and hence can not be used in combination with standard Java RMI objects.

Implementations of CORBA [20, 23] over Java allow for one-way or asynchronous method invocations using IIOP as the underlying protocol. These costs have been investigated in [7, 8].

These improved serialisation and RMI protocols implemented in the systems described above may eventually be integrated into the Java RMI standard to improve performance, but currently they are not interoperable with clients or servers using standard Java. Our work differs from these projects in that we retain the underlying RMI wire protocol and conformance with standard Java RMI. Our approach uses methods for overcoming latency issues in communication, by allowing asynchronous communication and delayed referencing of invocation result values, such as is found in Futures [30], Promises [16] and DRAMFs [10].

Future objects delay any blocking on remote method invocation until the result is required by the Client. This is especially useful when a client is not interested in the result of a method invocation, but only in the invocation occuring, resulting in no unnecessary object transferral. An example code fragment that utilises a Future object as implemented in our system is given in Figure 2. The local client and remote server can both continue execution until the client explicitly requests that the Future object be returned.

Alternatively, greater optimisation can be achieved by batching delayed references. By bundling several requests together we can minimise the communication overhead required for the invocation and

```
public class Client {
          Server RMIFutureServer;
          ... Server is bound to Remote Server ...

          Future ResultValue = RMIFutureServer.Method1();
          ... Client continues with execution while Remote Server
             calculates Result value ...

          ResultValue.complete();
          ... Client blocks until Remote Server returns a value...
 }
```

Figure 2: Example code fragment utilising Future objects. The client can continue code execution until it explicitly blocks on the Future reference.

result retrieval. This is known as Batched Futures [2]. The dependencies required to utilise systems such as Futures and Batched Futures have been studied in [4]. We are exploring implementations of these further optimisations using the same techniques.

3 Implementation of Asynchronous RMI

We have extended the standard Java RMI implementation by extending the stub generation process to include additional communications protocols. To indicate which methods are to use asynchronous communication and Future objects we annotate a pseudo Java interface with the keywords `asynch` and `future`. This pseudo Java interface is then compiled with a preprocessor to convert pseudo Java to real Java code, producing an equivalent Java interface, stub and skeleton code and an outline of the server implementation, including method signatures. Through this process, the server implementor can define on a per-method basis which methods are to synchronous or asynchronous, and which result objects will be Future objects. An example of an interface written in this Java pseudo code is shown in Figure 3.

This process can be extended to support additional keywords to define properties about a server implementation. We are exploring this technique to indicate such as properties as mobility, fault tolerance rules and cloning requirements with the control code for these properties maintained in proxy code such as Java RMI stubs.

```
package ServerTests;
import Protocols.*;

public interface Test extends java.rmi.Remote {
      public void Synch1(String input) throws java.rmi.RemoteException;
      public String Synch2() throws java.rmi.RemoteException;
      asynch public void Asynch1(String input) throws java.rmi.RemoteException;
      asynch public future String Future1() throws java.rmi.RemoteException;
}
```

Figure 3: Example of interface definition using Java pseudo code with asynchronous extensions.

Client communication is performed by dynamic downloading of an appropriate stub from the server site. By replacing the vanilla stub with a stub that can understand protocol extensions the server defines how it wants to be communicated with with no additional processing or code required by the client.

3.1 System Architecture

Implementation of the asynchronous and Future object extensions is performed by embedding monitoring threads in the stub code. Java's support for multithreading [22] enables us to monitor the remote method invocation concurrently with client execution. A standard Java RMI stub performs the tasks of maintaining a socket connection with the remote server; marshaling or serialising the parameters for the invoked method; and performing (and blocking on) the remote invocation. Figure 4 shows a sample stub method invocation for a standard Java RMI synchronous method. By embedding the actual invocation code into a separate thread that will complete when the remote method completes and returns,

we can achieve an asynchronous invocation. The invocation thread will sleep until the remote method completes. Figure 5 shows a sample stub method for an extended asynchronous method. The invocation code required has been moved to a custom thread class, `Protocols.InvokeThread`.

```
public void Synch1  (String $_Param1) throws java.rmi.RemoteException  {
    int opnum = 0 ;
    java.rmi.server.RemoteRef sub = ref;
    java.rmi.server.RemoteCall call = sub.newCall(
                                (java.rmi.server.RemoteObject)this,
                                operations, opnum, interfaceHash);
    try {
        java.io.ObjectOutput out = call.getOutputStream();
        out.writeObject($_Param1);
    } catch (java.io.IOException ex) {
        throw new java.rmi.MarshalException("Error marshaling arguments", ex);
    };
    try {
        sub.invoke(call);
    } catch (java.rmi.RemoteException ex) {
        throw ex;
    } catch (java.lang.Exception ex) {
        throw new java.rmi.UnexpectedException("Unexpected exception", ex);
    };
    sub.done(call);
    return;
}
```

Figure 4: Sample stub method for a synchronous standard Java RMI method invocation, performing all marshaling and invocation in method.

```
public void Asynch1  (String $_Param1) throws java.rmi.RemoteException  {
    int opnum = 2 ;
    java.rmi.server.RemoteRef sub = ref;
    java.rmi.server.RemoteCall call = sub.newCall(
                                  (java.rmi.server.RemoteObject)this,
                                  operations, opnum, interfaceHash);
    try {
        java.io.ObjectOutput out = call.getOutputStream();
        out.writeObject($_Param1);
    } catch (java.io.IOException ex) {
        throw new java.rmi.MarshalException("Error marshaling arguments", ex);
    };
    try {
        Protocols.InvokeThread invoke = new Protocols.InvokeThread(call,sub);
        invoke.start();
    } catch (java.lang.Exception ex) {
        throw new java.rmi.UnexpectedException("Unexpected exception", ex);
    };
    return;
}
```

Figure 5: Sample stub method for an extended RMI asynchronous method invocation, performing invocation in a seperate thread.

A Future object is indicated by the keyword `future` in the pseudo Java interface. When compiled, the method signature will be changed to return a `Protocol.Future` object instead of the expected type. The stub code that is generated will perform any additional type checking to ensure that the correct return type is being returned. A Future object can be linked with the invocation thread, so that when the value of the Future object is requested (using a `Complete()` method), it will block until the invocation is

complete. Figure 6 shows the sample stub method for an extended asynchronous method with a Future result object. Future objects can be extended to provide partial blocking or pinging techniques to allow the client to perform checks on remote invocation completion.

```
public Future Future1  ()throws java.rmi.RemoteException  {
    int opnum = 3 ;
    java.rmi.server.RemoteRef sub = ref;
    java.rmi.server.RemoteCall call = sub.newCall(
                                        (java.rmi.server.RemoteObject)this,
                                        operations, opnum, interfaceHash);
    try {
        Protocols.InvokeThread invoke = new Protocols.InvokeThread(call,sub);
        invoke.start();
        Protocols.Future fut = new Protocols.Future(call,invoke,
                                                    ref,"java.lang.String");
        return fut;
    } catch (java.lang.Exception ex) {
        throw new java.rmi.UnexpectedException("Unexpected exception", ex);
    };
}
```

Figure 6: Sample stub method for an extended RMI asynchronous method invocation with a Future result value.

As the stubs are downloaded by the client, no additional code has to be stored at the client site to use the extended protocols. Multiple stubs can be supported by the server implementation to provide different communication semantics for different clients. The generation process for the extended stubs is shown in Figure 7. It can be reapplied during the development process of the server to keep the appropriate stubs up to date.

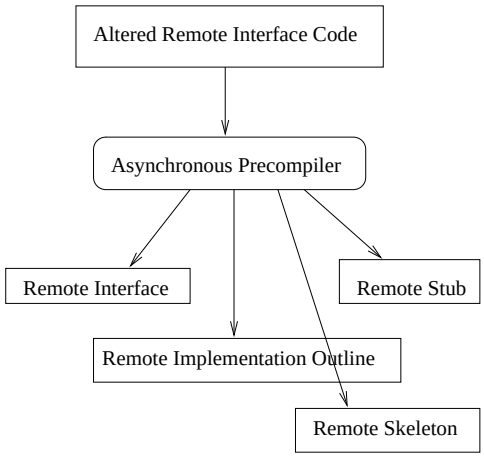

Figure 7: Generation Process for Extended RMI Stubs.

4 Evaluation

By adding protocol extensions that allow the client to avoid blocking on a communications request, we increase the time that a client has to perform useful work. What is interesting is the overhead incurred by the additional work in creating an asynchronous method and possibly a Future object on the client side. Obviously, asynchronous communication is going to be of benefit whenever the cost of the remote method is greater than the overhead cost.

The test were performed on null RMI methods to explore the different costs of standard RMI synchronous methods and our extended asynchronous methods with Future objects. Tests were performed between two Digital AlphaStation 255/300 MHz machines connected by 10MB Ethernet running JDK1.1.6-2 and JDK1.2B4; between two Sun Ultra2s (with 168 MHz UltraSPARC processors) connected by 10MB Ethernet running JDK1.1.6; and between a Sun E250 (with two 300MHz UltraSPARC processors) and a

	Synch.	Synch. w Result	Asynch.
Alpha JDK1.1.6-2	5.52	4.83	1.86
Alpha JDK1.2B4	7.65	6.98	4.22
SparcUltra2 JDK1.1.6	2.2	2.16	1.32
SparcE250/E450 JDK1.2.1	2.84	2.52	0.84

Table 1: Execution Results (in milliseconds) for synchronous and asynchronous RMI measured over different platforms and different JDKs. This table shows costs for executing a void synchronous method, a synchronous method with a simple return value and a void asynchronous method.

	Asynch, Future No Block	Asynch, Future Block	Synch
Alpha JDK1.1.6-2	7.38	28.78	15.92
Alpha JDK1.2B4	20.98	46.1	42.08
SparcUltra2 JDK1.1.6	5.04	7.08	7.44
SparcE250/E450 JDK1.2.1	3.64	5.28	7.04

Table 2: Total execution cost (in milliseconds) to obtain a result for a Future object after an asynchronous method invocation. Show is the time for a Future object access after remote method invocation has completed (no blocking), the time for a Future object access immediately after it was returned (blocking) and the equivalent cost for a synchronous standard Java RMI method returning the same Future object.

Sun E450 (with four 300MHz UltraSPARC processors) running JDK 1.2.1. Performance results in most cases were carried out in groups of 100 and averaged to produce the estimated cost. The costs for class loading shown in Table 4 were performed singly, as additional testing in the one JVM would produce a skewed result due to accessing cached copies of the class. Measurements were performed using the `System.currentTimeMillis()` method of Java.

Table 1 displays the execution costs of synchronous RMI calls and an extended asynchronous void RMI call. In these tests we used null RMI calls in order to explore the pure latency and blocking costs. Serialisation costs for remote methods are also a large cost component and have been discussed in [17, 21]. In Table 1 we can see results for three test series: a void synchronous method; a synchronous method with result; and a void asynchronous method. We can see that the additional costs for thread creation etc in the asynchronous method are greatly reduced in more advanced JDKs, i.e. JDK 1.2 for the Sparc platform.

Table 2 shows the execution costs for an asynchronous method with a return Future Object. The time shown in Table 2 is the total time to retrieve the value from the future object after an asynchronous method with the two options of waiting until the remote method has completed before access (no blocking) and immediate access (blocking). As expected, when Future objects are accessed immediately, the cost is much greater as the Client will block until remote method completion. The cost decreases in the more advanced JDKs due to optimised thread switching mechanisms and serialisation. Also shown in Table 2 is the cost for a synchronous standard RMI method with an equivalent return object. In all cases, even for a simple 'ping' equivalent method that just returns an object, the asynchronous cost plus delayed Future cost is less than that of a synchronous standard RMI method. In some cases, even the cost with blocking is smaller due to thread overlaying.

We can imagine that the additional costs for the asynchronous methods are incurred by the costs of creating additional threads of control. To explore this further we can analyse the costs on different platforms and using different JDKs for simple thread creation. We can see in Table 3 the costs for creating threads on the platforms used for the RMI tests. Threads were created for the `java.lang.Thread` class and for the custom thread class used in the asynchronous invocation, `Protocols.InvokeThread`. As can be seen, the thread creation cost for the base java.lang.Thread class and also the custom thread classes are very similar and not a major contribution to the additional cost for the asynchronous method overhead.

As we are introducing additional threads into the computation (one per asynchronous method invocation) we must explore the scalability costs for thread management in the JVM. There are three factors to be considered when analysing thread management: thread creation costs, thread running costs (whether

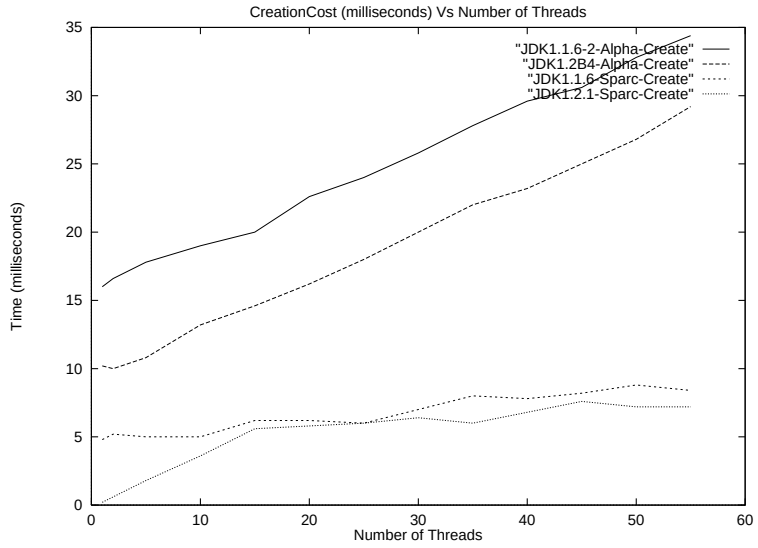

Figure 8: Cost of creating multiple threads on JDK1.1.6 and JDK1.2B4 (Alpha) and JDK1.1.6 and JDK1.2.1(Sparc).

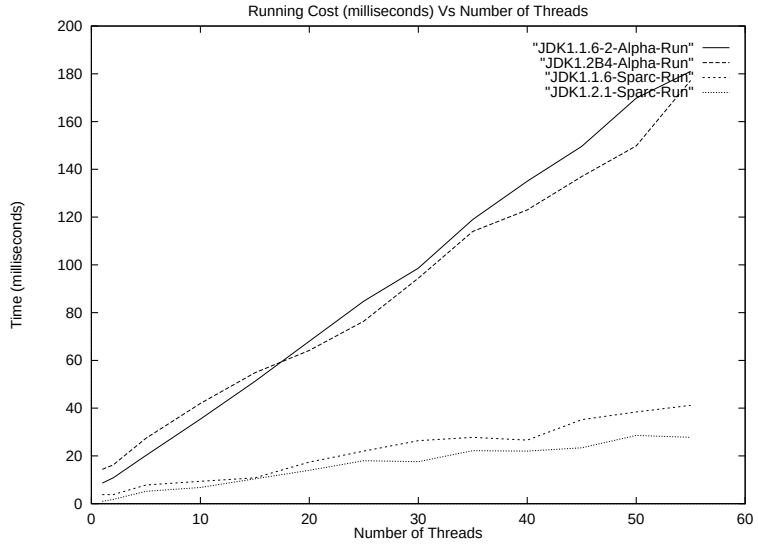

Figure 9: Completion cost of overlapping threads on JDK1.1.6 and JDK1.2B4 (Alpha) and JDK1.1.6 and JDK1.2.1(Sparc). Shown is the time (milliseconds) of completion for a number of threads running in the one JVM.

overlapping indeed even occurs); and thread switching costs. Figures 8, 9 and 10 show scalability results for these three factors for the different platforms and JDKs previously mentioned.

Figure 8 shows the cost of creating multiple threads with overhead increasing as more threads are required. Performance in recent JDKs scales very well. Figures 9 and 10 show an overhead increase as more threads are required, with improved scalability in later JDKs and on the Sparc platform. The cost of thread management is discussed in more detail in [14].

Another additional cost which must be considered is that of class loading. Specialised threads must be loaded by the client to perform the asynchronous invocation and the Future referencing. Table 4 shows that this is the area in which we find the most expense and also the most variation across platform. Interestingly, the more recent Sparc JDKs require more class loading time for custom classes, we can assume this is due to a more extensive verification process [19]. The additional costs of class loading will only be incurred the first time that a class is downloaded i.e. upon the first asynchronous method call per client. Any additional communications of this type will be able to access a cached class. Class loading costs for different JDKs is discussed in more detail in [19].

There is an obvious tradeoff between the benefit gained in using asynchronous RMI and the duration of the server method. The additional time taken by thread startup has to be exceeded by the remote

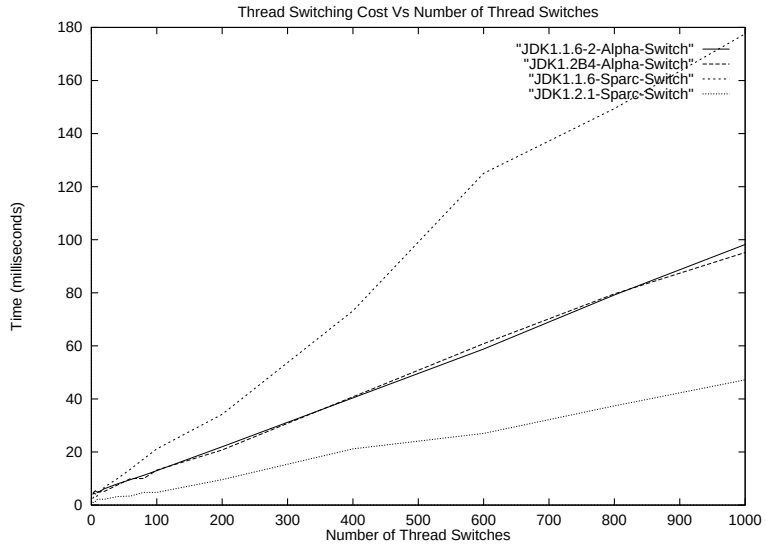

Figure 10: Cost of switching threads on JDK1.1.6 and JDK1.2B4 (Alpha) and JDK1.1.6 and JDK1.2.1 (Sparc). Shown is the time taken to perform a given number of thread switches. Each individual thread relinquished control immediately after access, so that the executing thread was switched immediately back to the main thread.

	java.lang.Thread	Thread Sub Class	Thread Sub class with Constructor
AlphaJDK 1.1.6-2	0.4854	0.589	1.0598
AlphaJDK 1.2B4	0.467	0.5176	0.7473
SparcUltra2JDK 1.1.6	0.0677	0.1096	0.3422
SparcUltra250JDK 1.2.1	0.0471	0.0441	0.0769

Table 3: Thread creation costs (in milliseconds) measured over different platforms and different JDKs. These tests were carried for the base java.lang.Thread class and two custom thread classes, the second of which contained a custom constructor with parameters.

	java.lang.Thread load	Asynchronous Thread class load
AlphaJDK1.1.6-2	0.26	14.56
AlphaJDK1.2B4	0.57	64.38
SparcUltra2JDK1.1.6	0.03	2.93
SparcUltra250JDK1.2.1	0.16	8.93

Table 4: Class loading costs (in milliseconds) for different platforms and JDKs. Shown are loading costs for the base `java.lang.Thread` class and the custom thread class used in the asynchronous communication mechanism.

method duration as well as any future blocking time incurred. This will of course be different depending on the client platform.

5 Conclusion

Several high performance or extended versions of Java's Remote Method Invocation mechanism exist, however they do not retain compliance with the standard underlying wire or serialisation protocols and are therefore not interoperable with clients or servers using standard JVMs. We propose a mechanism which extends Java RMI to support asynchronous communication and delayed referencing of returned objects (Futures). This mechanism involves embedding monitoring threads into the stub code with additional extensions to support Future objects. These extended stubs are produced using an alternative generation process to RMIC and are interoperable with standard Java RMI objects.

Servers define which methods will be invoked in a synchronous and asynchronous manner and which will use Future objects by defining a pseudo Java interface with additional `async` and `future` keywords.

Clients are essentially unaware of the communication mechanism used. They are aware, however, of the use of Future objects as explicit referencing is required. This work can be extended by increasing the number of protocols that are supported and further optimising the implementations of these protocols.

The extended stub mechanism we have described in this paper produces a dramatic performance increase by removing unnecessary delays caused by blocking on synchronous RMI invocations. As thread management and object creation costs decrease due to improvements in compilers and JVMs, the overhead of creating and switching between the monitoring threads and the client thread is decreasing to produce even better performance.

6 Acknowledgments

This work was carried out under the Distributed High Performance Computing Infrastructure Project (DHPC-I) of the On-Line Data Archives Program (OLDA) of the Advanced Computational Systems (ACSys) Cooperative Research Centre (CRC) and funded by the Research Data Networks (RDN) CRC. ACSys and RDN are funded by the Australian Commonwealth Government CRC Program.

Thanks to K.A. Hawick for useful discussion on the work presented here.

References

[1] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Trans. Computer Systems*, 2(1):39–59, February 1984.

[2] P. Bogle and B. Liskov. Reducing Cross Domain Call Overhead Using Batched Futures. In *Proc. OOPSLA'94, ACM SIGPLAN Notices*, volume 29, October 1994.

[3] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. *Supercomputing*, 1993.

[4] Henry Detmold and Michael J. Oudshoorn. Communication Constructs for High Performance Distributed Computing. In *Proceedings of the 19th Australasian Computer Science Conference, Melbourne, Australia*, January 31-February 2 1996.

[5] D. Flanagan. *Java in a Nutshell.* O'Reilly & Associates, Inc., 1996. ISBN 1-56592-183-6.

[6] Java Grande Forum. Available at http://www.javagrande.org/.

[7] A. Gokhale and D. Schmidt. Evaluating CORBA Latency and Scalability Over High-Speed ATM Networks. In *Proc. 17th International Conference on Distributed Computing Systems*, 1997.

[8] Aniruddha Gokhale and Douglas C. Schmidt. Principles for Optimising CORBA Internet Inter-ORB Protocol. In *Proc. HICSS Conference*, January 1998.

[9] Ian Goldberg, Steven D. Gribble, David Wagner, and Eric A. Brewer. The Ninja Jukebox. In *Proc. 2nd USENIX Symposium on Internet Technologies and Systems*, October 1999. To appear.

[10] K.A. Hawick, H.A. James, and J.A. Mathew. Remote Data Access in Distributed Object-Oriented Middleware. *Journal of Parallel and Distributed Computing Practices, Special Issue on Distributed Object Oriented Systems*, 1999. To appear.

[11] S. Hirano. HORB: Distributed Execution of Java Programs. In *Worldwide Computing And Its Applications, Lecture Notes in Computer Science*, volume 1274, 1997.

[12] S. Hirano, Y. Yasu, and H. Igarashi. Performance Evaluation of Popular Distributed Object Technologies for Java. In *Proc. ACM 1998 Workshop on Java for High-performance network computing*, February 1998.

[13] Matthew Izatt, Patrick Chan, and Tim Brecht. Ajents: Towards an Environment for Parallel, Distributed and Mobile Java Appilcations. In *Proc. ACM Symposium on Principles and Practice of Parallel Programming*, 1999.

[14] J.A.Mathew, P.D.Coddington, and K.A.Hawick. Analysis and Development of Java Grande Benchmarks. In *Proc. of the ACM 1999 Java Grande Conference*, June 1999.

[15] A. Krall and R. Grafl. CACAO - A 64 bit JavaVM Just-in-Time Compiler. *Concurrency: Practice and Experience*, November 1997. Available from http://www.complang.tuwien.ac.at/andi/.

[16] B. Liskov and L. Shrira. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *Proc. SIGPLAN'88 Conf. Programming Language Design and Implementation*, pages 260–267, June 1988.

[17] Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Henri E. Bal, and Aske Plaat. An Efficient Implementation of Java's Remote Method Invocation. In *Proc. ACM Symposium on Principles and Practice of Parallel Programming*, May 1999.

[18] Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Le Narzul, and Marc Shapiro. *Readings in Distributed Computing Systems*, chapter Fragmented Objects for Distributed Abstractions, pages 170–186. IEEE Computer Society Press, 1991.

[19] J.A. Mathew, A.J. Silis, and K.A. Hawick. Inter Server Transport of Java Byte Code in a Metacomputing Environment. In *Proc. of Technology of Object-Oriented Languages and Systems Pacific (TOOLS 28)*, November 1998.

[20] T. J. Mowbray and R. Zahavi. *The Essential CORBA: Systems Integration Using Distributed Objects*. Wiley, 1995. ISBN 0-471-10611-9.

[21] Christian Nester, Michael Philippsen, and Bernard Haumacher. A More Efficient RMI for Java. In *Proc. of ACM 1999 Java Grande Conference*, pages 152–157, June 1999.

[22] Scott Oaks and Henry Wong. *Java Threads*. Nutshell Handbook. O'Reilly & Associates, Inc., United States of America, 1st edition, 1997. ISBN 1-56592-216-6.

[23] Object Management Group (OMG). Common Object Services Specification, Volume 1. Available from http://www.omg.org/, March 1994.

[24] Object Management Group (OMG). The Common Object Request Broker: Architecture and Specification (Revision 2.0). Framingham, MA, July 1995.

[25] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Supercomputing '95*, December 1995.

[26] Scott Pakin, Vijay Karamcheti, and Andrew A. Chien. Fast Messages: Efficient, Portable Communication for Workstation Clusters and MPPs. *IEEE Concurrency*, 5(2), April/June 1997.

[27] Geetanjali Sampemane, Luis Rivera, Lynn Zhang, and Sudha Krishnamurthy. HP-RMI : High Performance Java RMI over FM. Available from http://www-csag.ucsd.edu/individual/achien/cs491-f97/projects/hprmi.html, 1999.

[28] Sun Microsystems, Inc. Java Object Serialization Specification. Available from http://www.sun.com/, November 1998.

[29] Sun Microsystems, Inc. Java Remote Method Invocation Specification. Available from http://www.sun.com/, October 1998.

[30] E. F. Walker, R. Floyd, and P. Neves. Asynchronous Remote Operation Execution In Distributed Systems. In *Proc. of the Tenth International Conference on Distributed Computing Systems*, May/June 1990.

[31] Matt Welsh. Ninja RMI : A Free Java RMI. Available from http://www.cs.berkeley.edu/~mdw/proj/ninja/ninjarmi.html, 1999.