A Customizable Implementation of RMI for High Performance Computing[⋆]

Fabian Breg and Dennis Gannon

Department of Computer Science, Indiana University

Abstract. This paper describes an implementation of Java's Remote Method Invocation (RMI) that is designed to run on top of the Globus high performance computing protocol. The primary contribution of this work is to illustrate how the object serialization mechanism used by RMI can be extended so that it becomes more configurable. This allows the implementation of object serialization protocols that are more efficient than the default or that are compatible with other distributed object models like HPC++, which is based on C++. Both issues are important when RMI is to be used in scientific computing.

1 Introduction

One of the main reasons of the popularity of the Java programming language is its support for distributed computing. Java's API for sockets, URLs and other networking facilities is much simpler than what is offered by other programming languages, like C and C++, which is why Java is likely to be adopted in scientific computing. These constructs, however, are still too tedious to use for generic distributed applications. Java's Remote Method Invocation (RMI) [18] was designed to make distributed application programming as simple as possible, by hiding the remoteness of distributed objects as much as possible.

RMI is a framework that allows objects to invoke methods on remote objects (i.e. objects residing in a different Java Virtual Machine, JVM for short) in the same way as methods of local objects (in the same JVM) are invoked. The syntax of a remote method invocation is similar to a method invocation on a local object, but the semantics for parameter passing are different. While remote objects are still passed by reference, non-remote objects are passed by copy instead (local method invocations pass all objects by reference). Also, references to remote objects must be obtained from a special entity called the registry. RMI does, however, provide a convenient framework for distributed computing.

As noted by others (see Sect. 2), Java RMI has a number of shortcomings when it comes to performance and flexibility. The first contribution of this work is to overcome these shortcomings improving both flexibility and performance of an important bottleneck of RMI: object serialization. To this end, we reimplemented RMI from scratch, retaining the original specification as much as possible. We

[⋆] This research has been supported by a contract with the Dept. of Energy DOE2000 project.

have only slightly altered the API to implement a flexible framework for object serialization. Since we only add methods to the API, our implementation can still handle most applications written for Java RMI.

The second contribution of this work is to investigate into the interoperability of RMI with another remote method invocation framework: HPC++ [2]. We will only briefly outline our approach to obtaining interoperability. A more in-depth study into the issues involved has been conducted in a previous article [3].

We are planning on using our implementation of RMI in future versions of our Distributed Problem Solving Environment Component Architecture Toolkit [5]. In this project, we will have a component architecture toolkit implemented in Java that communicates with distributed components written in HPC++.

The rest of this paper is organized as follows. In Sect. 2 we will point out some related research into RMI conducted by others. Section 3 gives a brief overview of the Java RMI framework. Section 4 gives an overview of some important design and implementation issues of our RMI implementation. Section 5 focuses on the object serialization and compares a number of object serialization protocols. Section 6 concludes this paper.

2 Related Work

One of the goals of the **Java Grande Forum** [8] is to investigate into methods to optimize Java RMI performance. In a short note presented to this forum, Philippsen and Haumacher [11] critiqued the performance of Java RMI and offered a number of suggestions to improve both its performance and flexibility. They conclude that a more open approach would encourage the widespread use of RMI in the scientific computing area.

The same is suggested by Thiruvathukal and others in [15]. They implemented a more open version of the Java RMI framework, which also provided some additional functionality, such as dynamically obtaining the interface of remote objects. Their API, however, is quite different from the original. Our goal was to make minimal changes to the official specification, while still providing some openness.

As part of the **Ninja** project [9] at the University of California, Berkeley, **NinjaRMI** [17] has been developed. This project adds some interesting features that are not (yet) available in Sun's RMI version. They do not address object serialization or interoperability with other languages.

Another implementation of RMI is provided by Raje and others. **ARMI** offers the possibility of performing asynchronous method invocations on remote objects. Their implementation is built on top of the original Java RMI and thus suffers from the same performance drawbacks.

In [16] Veldema and others implemented their own version of RMI for use in their **Albatross** [1] project. Their goal was to optimize RMI for homogeneous cluster computers and they offer a programming model based on **JavaParty** [12]. In addition, they translate Java applications to native code, thereby giving up the cross-platform portability. Nevertheless, their approach is interesting for scientific computing, because very low latency and high bandwidth is obtained with their implementation.

There are a number of distributed object frameworks like RMI available. **CORBA** [6] provides a distributed object model for interoperability between different platforms. Besides remote method invocation, **Voyager** [10] also provides the notion of active migratable entities, called agents. Hirano and others compared the performance of some of these different distributed object technologies in [7], including **HORB** [13], their own distributed object technology. They showed that indeed RMI exhibits worse performance than other distributed object technologies and adds significant overhead to the use of sockets.

3 Remote Method Invocation

Java **Remote Method Invocation (RMI)** [14] is a framework designed to simplify distributed object oriented computing in Java. An overview of the Java RMI model is shown in Fig. 1.

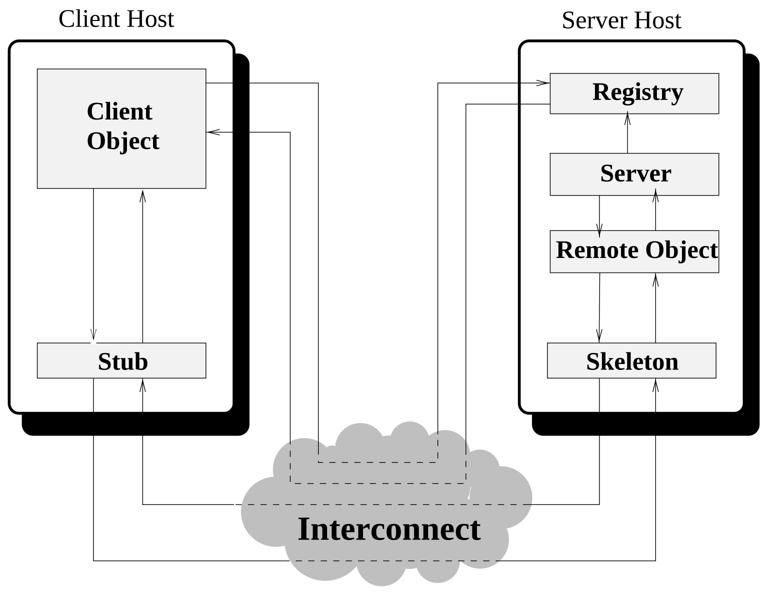


Fig. 1. RMI overview

RMI is essentially a client-server model in which the client talks to the remote object through a proxy called a **stub**. The stub itself talks to an active entity on the server side called a **skeleton**, which in turn calls the appropriate method on the remote object (which is local to the skeleton). The result is communicated back through the skeleton and the stub to the client. The stubs and skeleton are generated by a special compiler, `rmic`, and are completely hidden from the programmer.[1]

To obtain a reference to a remote object, the client contacts the **registry** on the remote host, providing it the name of the desired remote object. The remote object must previously have been bound in the registry by a server process. The remote reference that is passed from server via the registry to the client is actually the stub object itself.

To pass non-remote objects as a parameter of a remote method invocation, the notion of **object serialization** has been introduced in Java 1.1. Object serialization transforms an object or even a graph of objects into a sequence of bytes that can be sent over the network, allowing the other side to reconstruct the original object (graph). Note that the RMI parameter passing mechanism for non-remote objects is pass-by-copy, which is different from the parameter passing semantics for method invocations on local objects. From a programmer's point of view, remote objects are passed by reference. What is actually transfered is the stub object (which itself is passed by copy).

4 NexusRMI Implementation

We reimplemented Java RMI with two requirements in mind. The first requirement was to obtain interoperability with HPC++. For this, our stubs and skeletons have to use the Java port of Nexus to communicate with each other. Although the RMI implementation in the JDK 1.2 offers the possibility for the programmer to provide a custom socket implementation to use for communication, this is not flexible enough for our needs. The problem is that NexusJava is not a stream based protocol and therefore it does not fit a socket model. Our solution consists of redesigning the stubs and skeletons from scratch and reimplementing the `rmic` compiler.

The second requirement was to be able to experiment with object serialization to try to obtain better performance. Although Java RMI provides ways to customize object serialization, we needed to have more control over the entire process. In particular, we need to specify the actual byte representation of the data to be sent. Although HPC++ requires the programmer to supply the marshal and unmarshal methods for each object separately, the format in which primitive data is written is determined by Nexus. Since this format is dependent on the architecture on which the HPC++ object is running, we must deal with all possible formats, which is why we use the NexusJava framework for reading and writing primitives. Also, offering the possibility to specify the object serialization protocol allows us to implement various optimizations. Examples of this are described in Sect. 5.

[1] In an RMI application, the remote reference is declared to be of the remote interface type. The generated stub is defined to implement this interface, allowing the remote reference to be a reference to the stub.

This section describes the design and implementation of NexusRMI and how both requirements are met by it. In Sect. 5 we evaluate the performance of object serialization in our framework.

4.1 NexusRMI Transport Protocol

To understand the underlying communication protocol of NexusRMI, we will first introduce the NexusJava [4] model of communication. An overview of NexusJava is given in Fig. 2. In Nexus, a **node** or a host can contain multiple **contexts**, which in turn can contain multiple **threads**. A thread can communicate with an object in another context by issuing a **remote service request** on a **startpoint**. The startpoint will forward this request to the **endpoint** connected to it. The endpoint will then invoke the appropriate method on its associated **user object**.

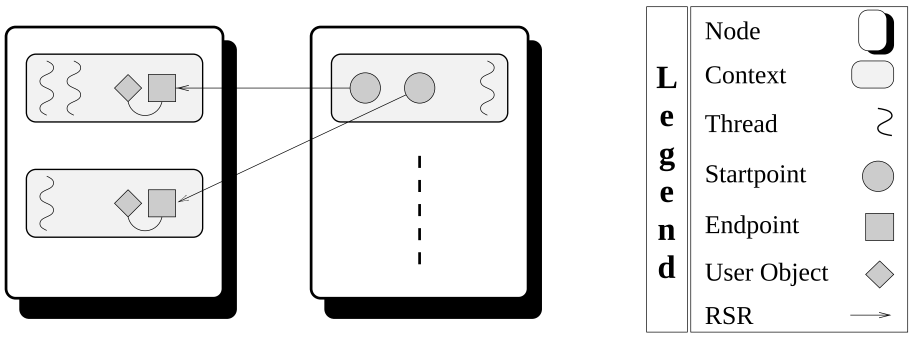


Fig. 2. NexusJava model

Figure 3 shows the different abstractions of NexusJava in our RMI implementation (refer to the NexusJava overview figure for a legend of the symbols used). Basically, a skeleton object contains an endpoint, with the actual remote object associated to it as a user object. The registry can be considered just another remote object and thus has a similar implementation. If the client invokes a remote method, the stub issues a remote service request on the startpoint connected to the appropriate skeleton endpoint. No reply message is implicit in a remote service request. The result value of the remote method is sent to client through a separate remote service request. To this end, the client itself contains an endpoint and it sends a startpoint to itself as part of the remote method invocation request message. A separate handler thread at the client extracts the result and passes it to the thread that initiated the remote method invocation.

When a remote object is exported, i.e. when the stub/skeleton pair is created, containing the necessary startpoints and endpoints. The stubs are registered in the registry when the server binds the remote object and are passed to the client

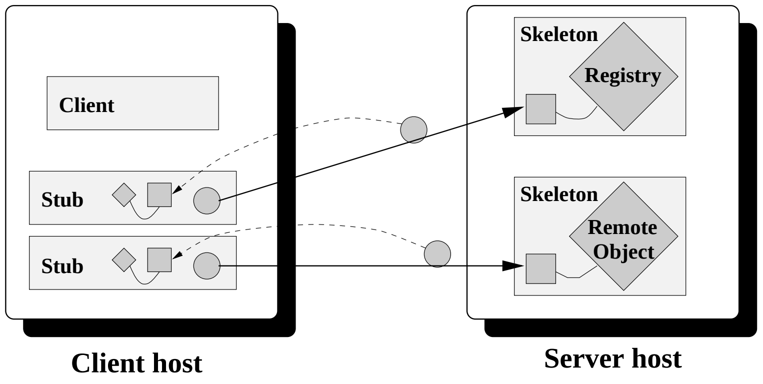


Fig. 3. NexusRMI communication system

when the client performs a lookup for the remote object. A startpoint to the registry is obtained through NexusJava's `attach()` method.

4.2 NexusRMI Object Serialization

Besides obtaining interoperability with HPC++, another purpose of our implementation was to experiment with object serialization in RMI to improve its performance. The intent was to provide the possibility to write a serialization protocol on a per remote object basis. The idea is that a remote object may know what kind of data to expect, and can thus make the best decision on the exact object serialization protocol.

To be able to specify the object serialization protocol on a per remote object basis, we had to alter the RMI specification slightly. When a remote object is exported, i.e. when the stubs and skeleton are created, we pass the serialization protocol with it, so that both the stubs and the skeleton know how to communicate with each other. A remote object can be exported by invoking `UnicastRemoteObject.exportObject()` method. If, however, the remote object extends `UnicastRemoteObject`, it is exported implicitly in the constructor. The NexusRMI interface of `UnicastRemoteObject` is shown in Fig. 4.[2]

The `Serialize` class is the superclass of objects implementing a serialization protocol. Its definition is shown in Fig. 5. Data is written into a `PutBuffer`[3] and read from a `GetBuffer`. The codebase is used to dynamically download classes from the specified site [14, sect 3.8]. Whenever a class has to be loaded that is not locally available the URL is passed to the `RMIClassLoader`, which than downloads the class. The `initWrite()` method initializes the object for

[2] Our implementation of RMI is mainly based on the JDK 1.1 version.

[3] The implementation of the `PutBuffer` class has been adapted for this project to allow it to dynamically expand as needed. This behavior is needed since we cannot simply determine the total buffer size requirements for an arbitrary graph of objects.

```
public class UnicastRemoteObject extends RemoteServer
{
  protected UnicastRemoteObject()              throws RemoteException;
  protected UnicastRemoteObject(Serialize ser) throws RemoteException;
  public Object clone()             throws CloneNotSupportedException;
  public static RemoteStub exportObject(Remote obj)
                                               throws RemoteException;
  public static RemoteStub exportObject(Remote obj, Serialize ser)
                                               throws RemoteException;
}
```

Fig. 4. `UnicastRemoteObject`

writing. The `initRead()` method initializes the object for reading. It is given the buffer to read from and the URL to dynamically load classes from. When all data has been written, the buffer with the serialized data can be obtained by invoking `getBuffer()`. The next six methods implement the reading and writing of HPC++ specific data, which are not to be altered by the custom serialization protocol. The rest of the methods implement the reading and writing of primitives and references. The actual implementations in this class are no-ops.

To implement a custom serialization protocol, the `Serialize` class has to be extended and the desired read and write methods need to be implemented. Only those methods actually needed have to be overridden. The next section will show some implementations of object serialization protocols that can be used in our framework.

Serialization of primitive types is fairly simple to implement since NexusJava provides methods to write primitives and arrays of primitives. To serialize non-array references, we need to be able to access the complete internal state of such references. Using the JDK 1.1, it is not possible to access the values of private and protected fields of an object. Furthermore, to reconstruct an object at the receiving side, we need a uniform way to create any object. Since the only way to construct an object is by invoking a constructor and the signature of a constructor is not guaranteed to be fixed, we need some other way of performing object serialization.[4]

Our approach is to add methods to a serializable class, which traverses all fields of an instance of the class and reads or writes these fields from or to a buffer. We implemented a compiler which automatically adds such methods to classes. If a parameterless constructor is not available in the class, the compiler will also add one for us so we can create instances of serializable objects in a uniform way.[5] Since we add our methods to the original Java source code, this

[4] In JDK 1.2, because of its more flexible security implementation, it is possible to access all fields of an object through reflection. However, we still have the problem of creating an uninitialized instance.

[5] This method is not completely safe, since an existing parameterless constructor may produce undesired side-effects.

```
public class Serialize
{
  protected Nexus nexus;
  protected PutBuffer outbuf;
  protected GetBuffer inbuf;
  protected URL codebase;

  public Serialize();
  public void initWrite();
  public void initRead(Object buffer, String codebase);
  public Object getBuffer();

  public final void writeHPCxxHeaderStartpoint(Startpoint sp);
  public final void writeHPCxxHeaderOffset(int off);
  public final void writeHPCxxHeaderHandlerid(int id);
  public final Startpoint readHPCxxHeaderStartpoint();
  public final int readHPCxxHeaderOffset();
  public final int readHPCxxHeaderHandlerid();

  public void writeboolean(boolean b);
  public void writebyte(byte b);
  public void writechar(char c);
  public void writeshort(short s);
  public void writeint(int i);
  public void writelong(long l);
  public void writefloat(float f);
  public void writedouble(double d);
  public void writeObject(Object o);
  public void writeFinalObject(Class type, Object o);

  public boolean readboolean();
  public byte readbyte();
  public char readchar();
  public short readshort();
  public int readint();
  public long readlong();
  public float readfloat();
  public double readdouble();
  public Object readObject();
  public Object readFinalObject(Class type);
}
```

Fig. 5. Serialize

method only works for serializable classes for which we have access to the source code.

5 Experiments with NexusRMI

In this section, we give some examples of object serialization protocol implementations that we have experimented with in our NexusRMI framework. We will show that, with the facilities offered in NexusRMI, a wide range of protocol semantics can be implemented, from semantics close to Java's own to semantics that can improve the performance of specific applications.

Section 5.1 describes the `DefaultSerialization` protocol. This protocol allows the serialization and deserialization of all primitive and reference types.[6] It offers the possibility of dynamically downloading classes from the server host and it will handle graphs of objects correctly, constructing a logically equivalent graph of objects on the receiving size. This protocol is close to the protocol used in Java RMI [7] and serves as the default serialization protocol in NexusRMI.

The `RelaxedSerialization` protocol described in Sect. 5.2 relaxes the default serialization semantics a little by removing the dynamic class loading facility and the check for previously serialized objects. This means that a graph of objects can still be passed as a parameter or return value, but only if it satisfies certain restrictions explained later. In that sense, this protocol is potentially unsafe, but may give better performance.

Finally, we present the `SpecificSerialization` protocol that can only handle objects very specific to the serialization protocol. This is of course a very restricted protocol, but this approach may further improve the performance for a specific application. This protocol is described in Sect. 5.3.

The benchmark that we will use to compare the performance of the various approaches is one that sends back and forth an array of Complex number objects, each containing two doubles. We experimented with different types of object serialization protocols as will be described in Sect. 5.4.

5.1 The `DefaultSerialization` Protocol

In this section we will describe a general serialization protocol that can handle any object given to it. Reading and writing primitive types is done by delegating this task directly to Nexus' buffer implementations. Reading and writing references to objects is done by invoking the methods added to these object by our compiler. For array references, a distinction is made between arrays of primitives and arrays of references. The serialization of the former is done by the Nexus library, the latter is serialized by serializing each reference separately. Some Objects that do not have the serialization methods added (because source code was not available) are serialized in an ad hoc manner.

[6] provided that source code is available.
[7] class versioning is not yet implemented.

By default, complete type information has to be included to be able to create an instance of the right class. We provide additional methods for (de)serializing parameters of a final class, since these do not need complete type information (the stub compiler can obtain the type information statically). Furthermore, care has to be taken to make sure that multiple references to the same object does not cause the creation of multiple objects at the receiver. Related to this is detecting cycles in the object graph so that the object serialization does not end up in an infinite loop.

5.2 The `RelaxedSerialization` Protocol

A majority of the data that we want to serialize has a 'flat' structure for which the check for multiple references to an object is not needed. By removing this check, we should obtain better performance in case many objects are serialized.

Another source of overhead may be the dynamic loading of classes from the server host. Even if classes are available locally, just invoking the RMIClass-Loader instead of loading classes directly by an invocation of `Class.forName()` may yield some overhead.

Both optimizations were incorporated in `RelaxedSerialization`, the second serialization protocol that we experiment with in this paper. Its implementation is similar to the `DefaultSerialization` protocol, except for the optimizations described above.

5.3 The `SpecificSerialization` Protocol

If parameters and return values for a particular remote method can only take values of a restricted set of types, it is possible to optimize object serialization significantly. Tests for types of objects that are never encountered could be removed completely. Also, the internal state of some objects could be accessed more efficiently than they are accessed in the `DefaultSerialization` and the `RelaxedSerialization` protocol.

To see how much the performance of RMI could be improved by using a more efficient serialization protocol, we performed some experiments with an array of Complex number objects, each containing two (primitive) double values that are publicly accessible. We wrote a little program that just sends these objects back and forth plus an accompanying object serialization protocol that can only handle these types of objects. The only methods that had to be provided were the `writeObject()` and `readObject()`. Since all fields of the object are public, these two methods directly access these fields to obtain higher performance.

The next section describes our experiments and the results obtained.

5.4 Object Serialization Performance

In this section we will compare the performance of the three serialization protocols described in the previous sections. These serialization protocol implementations all use the serialization primitives offered by NexusJava. In essence, object serialization in NexusRMI can be divided into three layers as shown in Fig. 6. The `DataConversion` layer takes care of writing primitive values in appropriate form in a byte array. The `PutBuffer/GetBuffer` layer offers facilities for managing outgoing and incoming buffers respectively. The `NexusRMI` layer is in fact any custom implemented serialization protocol. Only serialization protocols implemented in the `NexusRMI` layer can be 'plugged' into NexusRMI.

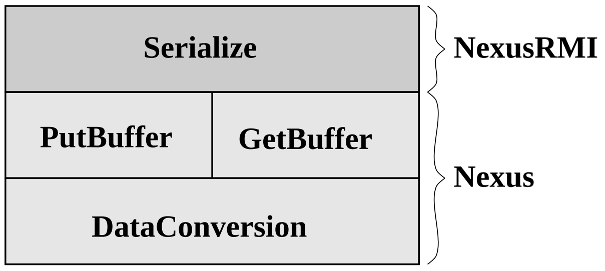


Fig. 6. Overview of serialization layers

The performance of our serialization protocols is measured by invoking the `writeObject()` and `readObject()` methods offered by the `NexusRMI` layer. To assess which parts of object serialization cause the most overhead, we will also measure object serialization performance by directly invoking the the methods from `GetBuffer` and `PutBuffer` and by directly invoking the methods from `DataConversion`. These methods cannot be 'plugged' into NexusRMI, however, since the interface required by NexusRMI is bypassed.

The first alternative method uses the `PutBuffer` implementation to write the simple internal state directly, and the `GetBuffer` implementation to read the internal state directly. Comparing the performance of this method with the performance of the specialized object serialization protocol of Sect. 5.3 gives us a good indication of the overhead incurred by the object serialization framework of NexusRMI.

In the other methods, we bypass the `PutBuffer` and `GetBuffer` implementations and use the `DataConversion` object implementation provided by Nexus-Java to directly write into a byte array. We first measure the serialization and deserialization performance, including the overhead caused by allocating the buffer when writing, and allocating the data structure (the Complex number object array and each complex number individually) when reading. Lastly, we measure the (de)serialization performance when using the `DataConversion` object, without including the memory allocation overhead.

We assessed the performance of object serialization by calculating the **write rate** and **read rate** separately. We measured the time T (in seconds) needed to serialize a particular object of size K (in Mbits). The write rate is than defined to be K/T (in Mbits per second or mbps). We also calculated the read rate by dividing the size of the object by the time it takes to deserialize it. The performance of the different object serialization protocols on a uniprocessor 300 MHz UltraSPARC II is summarized in Fig. 7.

As can be seen, the upper limit for object serialization for complex number objects is approximately 40 mbps (megabits per second). For lower number of objects, the resolution of the used timer is too low, which causes the measured time to be 0 msec, causing the object serialization performance to become infinity. The allocation of the buffer does not add significant overhead, while our implementation of the `PutBuffer` class does add significant overhead. The object serialization framework of our NexusRMI package does not add much overhead when using the special serialization protocol for complex numbers. Finally, adding generality to the object serialization performance causes the performance to drop some more.

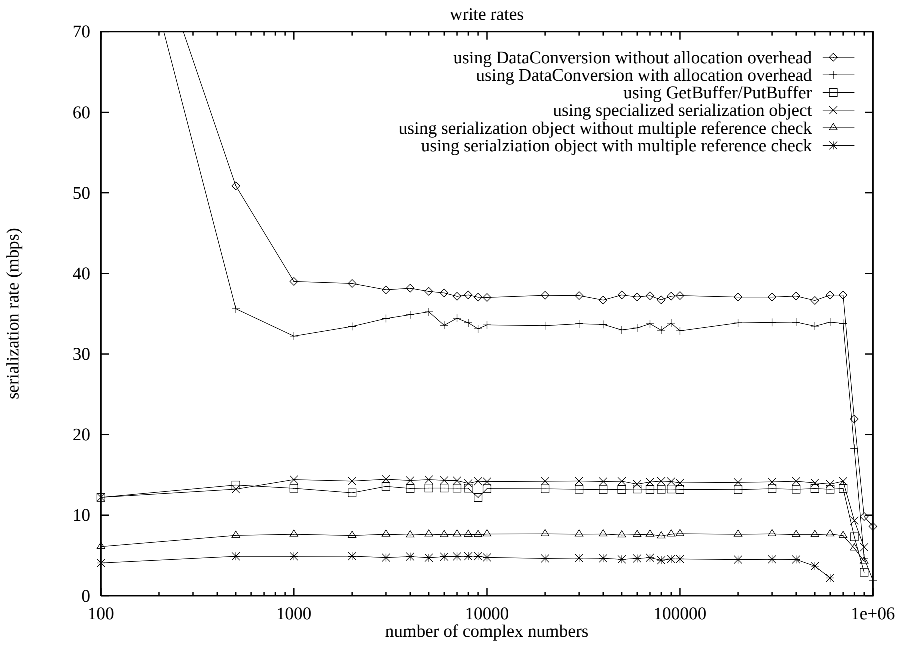
Object deserialization does suffer from overhead due to the allocation of the data. In this case, each complex number must be allocated on the heap separately. The implementation of the `GetBuffer` class does not add significant overhead, which was to be expected, since it is just a wrapper around byte-array. Contrast this with the `PutBuffer` implementation, which had to provide the dynamic expansion of the buffer. Also on the deserialization side, our object serialization framework does not add much overhead in itself, except when the protocol gets more general.

To compare our serialization protocols in the context of RMI we ran a simple RMI program that sends back and forth an array of complex number objects. We tested the three serialization protocols that are implemented according to the NexusRMI serialization framework as well as the default JavaRMI for comparison. We used a dual processor UltraSparc to perform the test in order to keep actual transfer time at a minimum, and still allow client and server to run in parallel. The results are shown in Fig. 8.

As can be seen, the default serialization protocol achieves the same order of performance as Java RMI for larger message sizes. Note, however, that Java RMI's object serialization semantics is still stronger than ours. Also, the implementation of the RMI system is different. Where we rely on compiler generated class traversal routines to access the internal state of an object, Java RMI uses reflection. On the other hand, Java RMI uses TCP to communicate, while we use the Java port of the Nexus library, which uses TCP as its underlying communication mechanism. As noted in [3], using NexusJava also inhibits the overlapping of computation and serialization, something that is done in Java RMI.

Relaxing protocol semantics improves RMI performance, although not dramatically, which could be expected after investigating the object serialization protocol performance in isolation as we did earlier.

The specialized serialization protocol improves performance even further. The performance thus obtained is, however, still disappointing. The reason for this poor performance is probably a combination of reasons also mentioned in [11], where the most important would be the fact that each array element has to be separately serialized. The serialization of the actual primitive data is done by



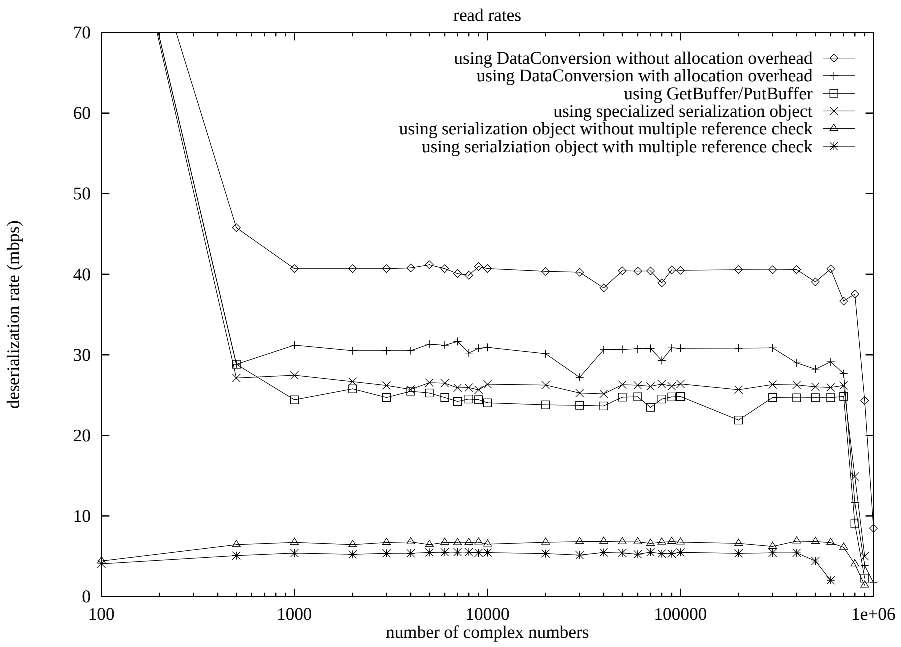


Fig. 7. Write and read rates on a uniprocessor UltraSPARC II

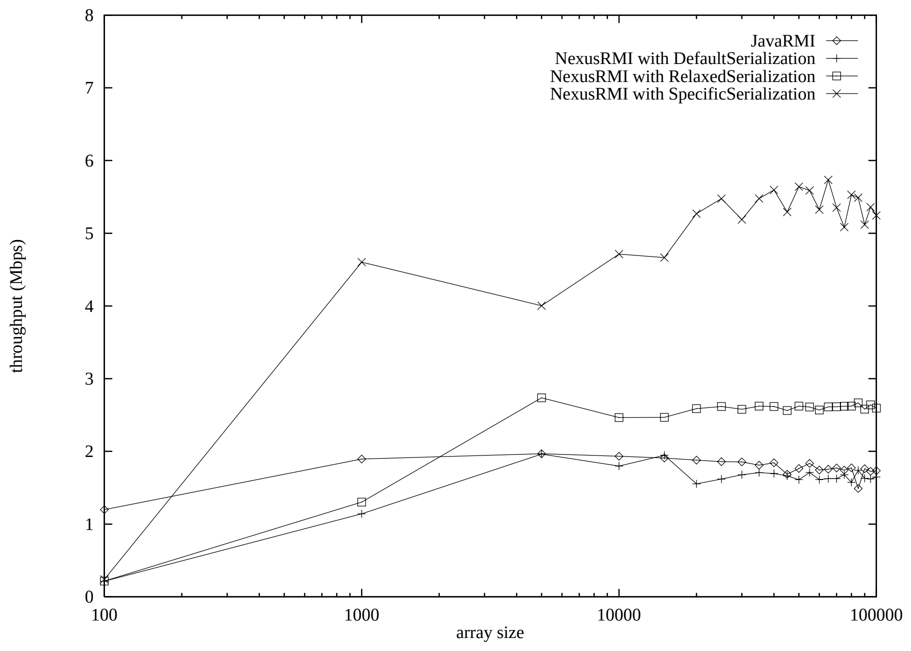


Fig. 8. Performance of RMI implementations

explicitly converting each data item to a byte representation and copying these bytes to the destination buffer.

6 Conclusions and Future Research

We have presented an implementation of the Java Remote Method Invocation framework that allows us to almost transparently communicate with remote object written in the HPC++ language. We have also shown how we let the implementor of remote objects exercise more control over the object serialization aspect of RMI, while making minimal changes to the RMI specification. Next we measured the performance of a number of object serialization protocols to identify those aspects of object serialization that prevent us from obtaining a high performance. These tests, however, also show that the performance can be improved by restricting the object serialization protocol.

In addition to specifying the object serialization protocol on a per remote object basis, one may also wish to specify the underlying network protocol on a per remote object basis. This would allow remote objects to easily communicate with objects that use a different transport layer. The issue here is to design a common transport interface that can be efficiently implemented by any other existing communication library.

Java RMI was designed as a very general framework for remote objects to communicate. Although this is very convenient for most application programmers, when dealing with performance demanding applications, it is generally useful to sacrifice some generality in favor of efficiency. We have shown that this is possible with RMI too, while retaining the convenience for applications that do not really need the performance.

References

1. Albatross project, 1998. http://www.cs.vu.nl/albatross.
2. P. Beckman, D. Gannon, and E. Johnson. Portable Parallel Programming in HPC++. 1996.
3. F. Breg, S. Diwan, J. Villacis, J. Balasubramanian, E. Akman, and D. Gannon. Java RMI Performance and Object Model Interoperability: Experiments with Java/HPC++. *Concurrency: Practice and Experience*, 10:(to appear), 1998.
4. I. Foster, G.K. Thiruvathukal, and S. Tuecke. Technologies for ubiquitous supercomputing: a Java interface to the Nexus communication system. *Concurrency: Practice and Experience*, 9(6):465–475, jun 1997.
5. D. Gannon, R. Bramley, T. Stuckey, J. Villacis, J. Balasubramanian, E. Akman, F. Breg, S. Diwan, and M. Govindaraju. Developing Component Architectures for Distributed Scientific Problem Solving. *IEEE Computational Science & Engineering*, 5(2):50–63, 1998.
6. Object Management Group. The Common Object Request Broker: Architecture and Specification, jul 1995.
7. S. Hirano and H. Yasu, Y. Igarashi. Performance Evaluation of Popular Distributed Object Technologies for Java. *Concurrency: Practice and Experience*, 10:(to appear), 1998.
8. Java Grande Forum. http://www.javagrande.org/.
9. Ninja project, 1998. http://ninja.cs.berkeley.edu/.
10. Objectspace. Objectspace Voyager Core Package Technical Overview, 1997.
11. M. Philippsen and B. Haumacher. Bandwidth, Latency, and other Problems of RMI and Serialization. 1998.
12. M. Philippsen and M. Zenger. JavaParty - Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, nov 1997.
13. H Satoshi. HORB: Distributed Execution of Java Programs. 1997.
14. Sun Microsystems. *Java(TM) Remote Method Invocation Specification*, oct 1997. revision 1.42 jdk1.2Beta1.
15. G.K. Thiruvathukal, L.S. Thomas, and A.T. Korczynski. Reflective Remote Method Invocation. *Concurrency: Practice and Experience*, 10:(to appear), 1998.
16. R. Veldema, R. van Nieuwpoort, J. Maassen, H.E. Bal, and A. Plaat. Efficient Remote Method Invocation. Technical Report IR-450, Vrije Universiteit, Amsterdam, sep 1998.
17. M. Welsh. Ninjarmi, 1998. http://www.cs.berkeley.edu/~mdw/proj/ninja/.
18. A. Wollrath, J. Waldo, and R. Riggs. Java-Centric Distributed Computing. *IEEE Micro*, 17(3):44–53, may/jun 1997.