JToe: a Java* API for Object Exchange

Serge Chaumette [a], Pascal Grange[a], Benoît Métrot[a], and Pierre Vignéras[a]
email: {Serge.Chaumette, Pascal.Grange, Benoit.Metrot, Pierre.Vigneras}@labri.fr

[a] Laboratoire Bordelais Recherche en Informatique,
Domaine Universitaire,
351, cours de la Libération,
33405 Talence Cedex, France.

This paper presents JToe, an API dedicated to the exchange of Java objects in the context of high performance computing. Even though Java RMI provides a good framework for distributed objects in general, it is known to be quite inefficient, mainly due to the Java serialization process. Many projects have already improved RMI either by redesigning and reimplementing it or by reimplementing the serialization process. We claim that both approaches are missing a clear and high level API for the exchange of objects. JToe proposes a new simple API that focuses on the exchange of objects. This API is flexible enough to allow, for instance, direct copy of byte streams representation of JVM objects over a specialized transport layer such as Myrinet or lapi. Remote method invocation frameworks such as Java RMI can then be implemented over JToe with good performance enhancement perspectives.
Keywords : Java, RMI, Serialization, object exchange, high performance computing.

1. INTRODUCTION

Whereas remote procedure call and remote method invocation have given developers a good abstraction of the underlying transport layers for client/server communication, high performance computing has not adopted those programming models – message passing interface is still preferred – mainly because of the performance penalty they suffer from. It is acknowledged that the main drawback of RMI is its inefficiency for object serialization and de-serialization. A lot of developments exist that improve this serialization mechanism in the context of high performance computing.

Some of them consist in a whole RMI reimplementation [1,2,4]. In these cases, the problem of the exchange of objects over the network has to be addressed simultaneously with the problems of distributed garbage collection, remote method invocation, threads management, registration management, and so on. Our work consists in defining the notion of exchange of objects as simply as possible through the JToe API. This allows one to specifically address this problem separately from the other RMI challenges.

Other works (or parts of previously cited works) address only the serialization problem in the context of high performance computing [1,3,9]. From a technical point of view, they generally rely [1,3] on the already defined `ObjectOutputStream` and `ObjectInputStream` classes. We argue in section 2 that these APIs are low level ones. What we propose is a new high level API dedicated to the exchange of Java objects in the context of high performance computing.

The rest of this paper is organized as follows. In section 2 we discuss why we introduce a new API. We describe this API in section 3. Existing implementations and performances are presented in section 4.

2. A NEW API

We need an API that allows one to get all the benefits of an efficient object exchange layer that can be used to implement a distributed application or more general frameworks such as RMI. For this

*Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. The authors are independent of Sun Microsystems, Inc.

purpose, a widely used API already exists in Java: `Object(Output/Input)Stream` [5]. However we believe that, in the context of high performance computing, a new distinct API for the exchange of objects is needed.

First, it is acknowledged that standard Java serialization is not efficient. Faster implementations have been proposed [1–4]. Of course, none of them rely on, neither respect, the Java serialization specification: technically speaking, one can inherit `ObjectOutputStream` and still deeply break the compatibility with the standard serialization process as far as the corresponding `ObjectInputStream` class is provided. However, looking at those classes, some elements remind us that, when inheriting these classes, we may (or have to) respect the standard serialization process. For instance, how to deal with the `UseProtocolVersion method` when we do not respect the standard protocol to achieve efficiency? Moreover, these classes, `ObjectOutputStream` and `ObjectInputStream`, may evolve with the protocol, and they have in the past. Such evolutions may lead to incompatibilities with legacy sub-classes[2].

Second, we consider that `Object(Output/Input)Stream` is a low level API since it does not hide the stream management of object serialization. Such a stream oriented API is not well suited for transport layers that do not rely on stream based hardware or libraries [6,7]. Of course one can inherit – and this is the common approach – `ObjectOutputStream` and give a brand new implementation for non stream based hardware. Even if we believe this is unnatural and error-prone, it is not a technical issue. The real problem is that, in order to receive an object, one must use an instance of the `ObjectInputStream` class and especially the `readObject` method. This implies that a thread must be waiting on this blocking method for an object to be received. This prevents one to get benefits from using special architectures and/or libraries that allow, for instance, one-sided initiator data-transfers [6]. For all these reasons, we claim that a new non stream oriented API has to be defined. One may argue that RMI could already be implemented to take advantage of one-sided initiator data-transfers. Nevertheless, this implies to reimplement RMI what, in turn and as stated in section 1, implies to deal with a lot of other high level challenges instead of just focusing on the exchange of objects: distributed garbage collection, method invocation, registration management, etc. This is why an object exchange oriented API has to be defined, independently of RMI.

The problem of defining adapted APIs for the easy replacement of the transport layer or of the serialization process has already been addressed. However, it generally leads to an unfortunate dividing line between the serialization process and the transport layer. For instance in KaRMI [1] a notion of *technology* is defined. The serialization is performed by KaRMI and the result of this serialization is sent using a given transport *technology*. A new *technology* can be defined to enhance the network transfer or to provide a new type of network support. However, in some improvement approaches [3,9], the data to be transferred may depend on the remote/local JVM and/or on the available communication mechanisms: for instance the assumption that the two communicating computers are running similar JVMs allows to directly send memory regions. It may also be the case that the transport layer makes it possible not to buffer the data to be transferred. In such a case, separating the serialization process from the transport process prevents one from providing improvements based on these characteristics. This is why we believe that this separation is cumbersome.

3. JTOE: THE API

From the previous observations, we defined a simple powerful API: JToe. It is mainly composed of two interfaces: `Node` and `CopyListener` (see program 1). When one wants to send a copy of an object to a remote node, he has to get the `Node` object[3] representing the node he wants to communicate with and then invoke the `copy` method passing it the object to send as an argument. On the server side, the JToe layer will inform the application, by means of a call-back mechanism, that a new object has arrived using the `copied` method of the `CopyListener` interface. The action of copying an object to a remote node is a one-sided action. No user thread has to wait for an object to be received. This is the responsibility of the implementation to accept and receive any new object and then signal this

[2]The best example being the `writeObjectOverride` method that allows one to define a new serialization process. It only exists since jdk 1.2.

[3]The problem of retrieving a `Node` is not managed by JToe since it is not a transfer of object problem but more a registry one. General naming services may be used such as JNDI [8].

Program 1 The JToe API

```
public interface Node {
    void copy(Serializable object) throws JToeException;
}

public interface CopyListener {
    void copied(Serializable object);
}
```

arrival to the application using an event driven programming model through `CopyListener.copied`.

This approach is not only easy to understand and to use, but also allows to really take advantage of one-sided communication libraries [6]. Moreover, as shown in figure 1, there is no limitation on the way the `Node` interface can be implemented. The `copy` method is responsible for the whole process of copying an object to another address space that is: the construction of the data to send (serialization) and the transfer of these data through a transport layer. This way, no artificial dividing line between the serialization process and the transport layer is introduced. We claim that this allows any serialization improvement approach to be implemented.

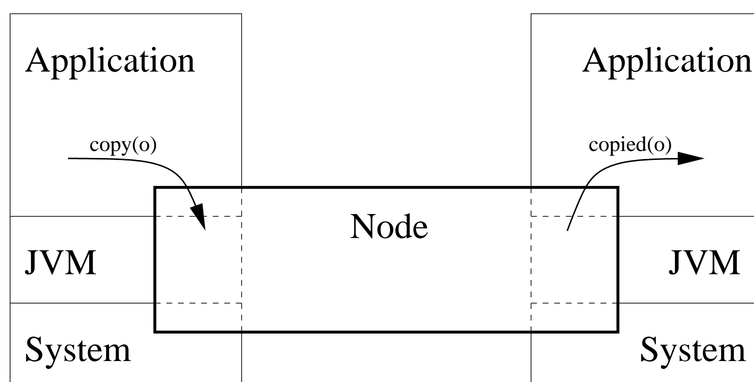
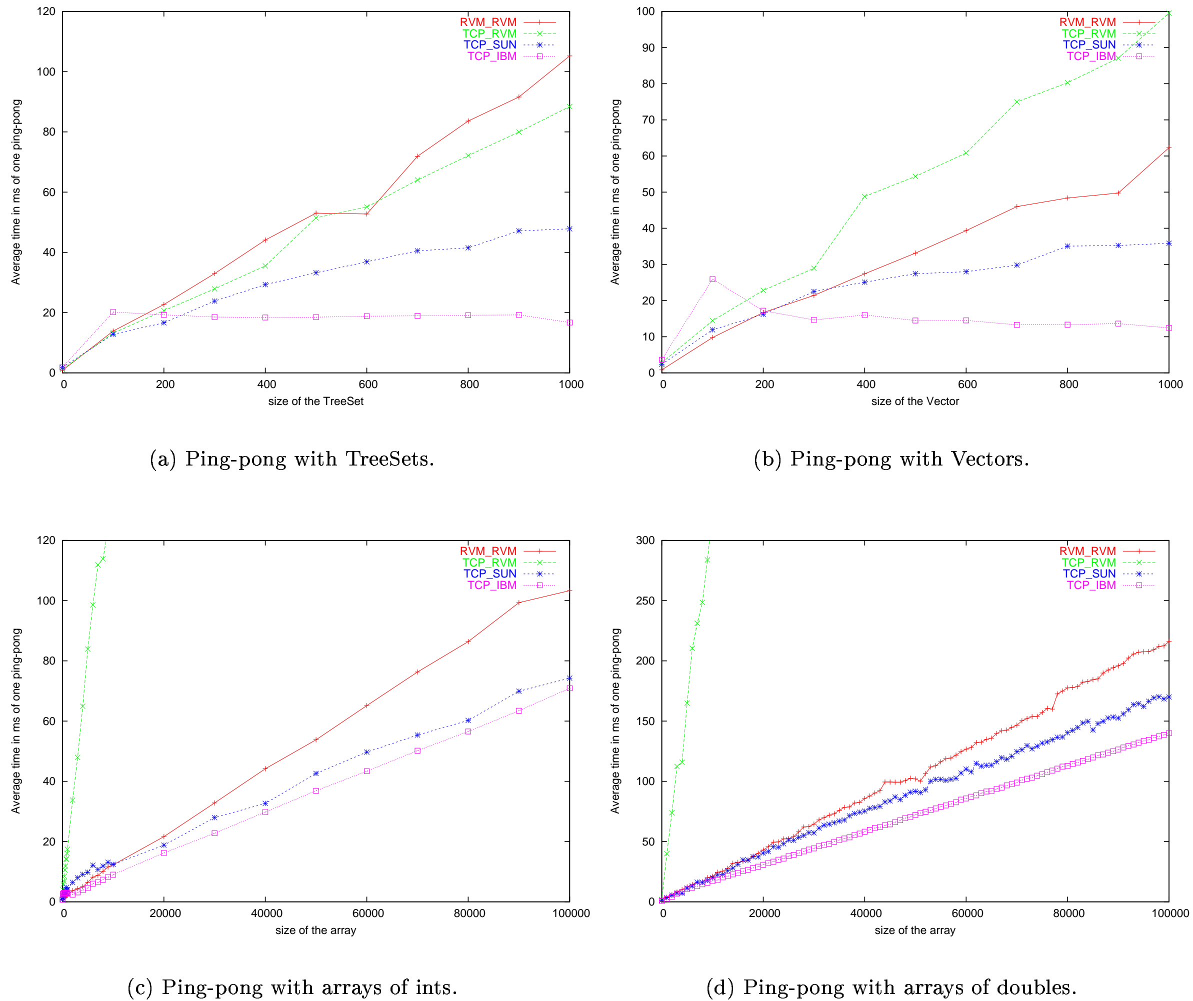


Figure 1. General JToe behavior.

4. JTOE IMPLEMENTATIONS

We have already developed three implementations of JToe [10]. Two of them are totally written in Java, one relying on Java RMI, the other using TCP sockets and the standard Java serialization. These allow any JToe application to be 100% Java compatible and portable. They also serve as reference implementations for regression tests.

The third is a JVM level implementation, i.e. at a level where we can directly access the memory representation of objects. The goal of this implementation of JToe is to provide high performance in clusters of homogeneous computers and homogeneous JVMs by directly sending memory data instead of going through the serialization process. This implementation uses the JikesRVM virtual machine [14]. JikesRVM allows, among other things, to directly access memory and interact with the garbage collector from Java code. It is an interesting experimentation platform and allowed us to implement a prototype in a reasonable time. In this implementation our concern is to perform



(a) Ping-pong with TreeSets.

(b) Ping-pong with Vectors.

(c) Ping-pong with arrays of ints.

(d) Ping-pong with arrays of doubles.

Figure 2. Ping-pong performances.

efficiently and interact well with the garbage collector. JikesRVM supports various garbage collection policies. Parts of our implementation are garbage collector dependent.

This implementation mainly behaves as follow: when an object is being copied, the corresponding graph of objects is computed. Then the data of the objects of the graph are sent with zero copy[4]. On the receiver side, memory is allocated in a garbage collector dependent space – mainly the nursery – and data are directly written into this area. The pointers are then updated to reflect the original structure.

Figure 2 shows the performances of the JikesRVM specific implementation of JToe compared to the 100% Java implementation. Both are using TCP as their communication layer. The values represented by the curves are the average time of one round-trip for the specified object in a ping-pong like application. The `RVM_RVM` curve shows the performances of the JikesRVM dedicated JToe implementation when run with JikesRVM – note that it cannot be run with another JVM. The TCP_RVM

[4]When we talk about zero copy we mean that our code does not perform any copy even if the actual communication layer will since we rely on TCP. We claim that future release with zero copy communication layer will actually achieve real zero copy.

curve shows the performances of the 100% Java implementation of JToe running with JikesRVM. The TCP_SUN and TCP_IBM curves are for the same implementation running on, respectively, the Sun Java virtual machine version 1.4.1_01-b01 and the IBM Java virtual machine version 1.4.0. All the experimentations were done using two Linux 2.4.18 computers with Intel 1.7GHz processors and 512MB of RAM connected with ethernet 100Mbps.

We can see that, generally speaking, JikesRVM does not perform as well as the two other virtual machines for our ping-pong application. However, for the ping-pong of vectors, we see that our JikesRVM specific JToe implementation outperforms the 100% Java one by 30% to 40% when both are running on JikesRVM. This is a really promising result since it allows one to think that the same sort of enhancement may lead to same performance enhancement with the other virtual machines. For the treesets, our implementation does not give interesting results, it performs even worse than the default serialization. Actually our code performs a suboptimal graph browsing to collect the objects to send where the adhoc serialization of the `java.util.TreeSet` class directly and linearly writes the objects it contains. We claim that this is the reason of this poor performance. Future releases of JToe for JikesRVM will improve the graph browsing algorithm by marking visited objects with JikesRVM specific techniques. Finally for arrays of int and double, our implementation totally outperforms the standard serialization on JikesRVM. The round-trip times are close to those obtained with the Sun or IBM virtual machine despite the poor performances of JikesRVM. This allow one to think that similarly optimized implementations of JToe for the Sun or IBM virtual machine would lead to comparable improvements.

The issue of fast transfer of objects has already been addressed in other works. Expresso [11] is a framework aimed at transferring Java objects efficiently using zero copy mechanisms. Expresso relies on the Kaffe [12] virtual machine. Expresso uses the notion of cluster to avoid graph exploration and pointer update. A cluster is a contiguous memory area where objects can be allocated. To transfer an object to a remote node, one actually transfers the cluster containing that object. All the other objects in the cluster are also transfered. With Expresso, objects must explicitly be allocated in the correct cluster to be transfered and the references between objects in different clusters are not conserved.

Ibis [13] is a grid computing environment for Java. It defines the IPL, an API over which higher level distributed environments such as RMI can be implemented. Ibis differs from our work since the IPL not only defines the way objects are sent between computers but also how to access topology information, monitoring data, etc. Moreover, object serialization optimization in Ibis relies on byte code rewriting, the aim of which is to add the serialization code to classes to avoid dynamic type inspection.

5. CONCLUSION

In this paper we have presented a new API which we call JToe that is dedicated to the exchange of objects between JVMs. This API makes it possible to take advantage of the knowledge we have of both the source and target JVMs and of the underlying network and the associated communication libraries. This leads to very efficient exchange of objects between JVMs.

We have three implementations running. Two are 100% written in Java (one over RMI and one over TCP), the third is a low level one dedicated to Jikes RVM.

This last implementation is still a work in progress. We are currently working to enhance the graph browsing algorithm using JikesRVM specific mechanisms. We also plan to release a lapi and a Myrinet version of JToe.

REFERENCES

[1] Christian Nester, Michael Philippsen and Bernhard Haumacher. A More Efficient RMI for Java. In Java Grande, pages 152–159, 1999.

[2] Jason Maassen, Rob Van Nieuwpoort, Ronald Veldema, Henri E. Bal, Thilo Kielmann, Ceriel J. H. Jacobs and Rutger F. H. Hofman. Efficient Java RMI for parallel programming. Programming Languages and Systems, 23(6):747–775, 2001.

[3] Fabian Breg and Constantine D. Polychronopoulos. Java virtual machine support for object se-

rialization. In Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande, pages 173–180. ACM Press, 2001.

[4] Fabian Breg and Dennis Gannon. A Customizable Implementation of RMI for High Performance Computing. In Proc. of Workshop on Java for Parallel and Distributed Computing of IPPS/SPDP99, pages 733–747, 1999.

[5] Sun Microsystem. Serialization specification. http://java.sun.com/

[6] Shah G., Nieplocha J., Mirza C.and Harrison R., Govindaraju R.K., Gildea K., DiNicola P. and Bender C. Performance and experience with LAPI: a new high-performance communication library for the IBM RS/6000 SP. In International Parallel Processing Symposium, pages 260–266, 1998.

[7] Myricom. Myrinet software. http://www.myri.com/scs/

[8] Sun Microsystem. Java Naming and Directory Interface. http://java.sun.com/products/jndi/

[9] K. Kono and T. Masuda. Efficient RMI: Dynamic Specialization of Object Serialization. In Proc. of IEEE Int'l Conf. on Distributed Computing Systems (ICDCS), pages 308–315, 2000.

[10] Pascal Grange and Pierre Vignéras. The JToe project. http://jtoe.sf.net

[11] L. Courtrai, Y. Mahéo and F. Raimbault. Expresso: a Library for Fast Java Objects Transfert. In Myrinet User Group Conference, Lyon, 2000.

[12] The Kaffe Java virtual machine. http://www.kaffe.org/

[13] Rob Van Nieuwpoort, Jason Maassen, Rutger Hofman, Thilo Kielmann, Henri E. Bal. Ibis: an Efficient Java-based Grid Programming Environment. Joint ACM Java Grande - ISCOPE 2002 Conference, pp 18–27, 2002.

[14] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen. Implementing Jalapeño in Java. In Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA 99, Denver, Colorado, November 1-5, 1999, volume 34(10) of ACM SIGPLAN Notices, pages 314 324. ACM Press, Oct. 1999.