High Performance Java Remote Method Invocation for Parallel Computing on Clusters

Guillermo L. Taboada, Carlos Teijeiro, Juan Touriño
Dept. of Electronics and Systems
University of A Coruña, Spain
{taboada,cteijeiro,juan}@udc.es

Abstract

This paper presents a more efficient Java Remote Method Invocation (RMI) implementation for high-speed clusters. The use of Java for parallel programming on clusters is limited by the lack of efficient communication middleware and high-speed cluster interconnect support. This implementation overcomes these limitations through a more efficient Java RMI protocol based on several basic assumptions on clusters. Moreover, the use of a high performance sockets library provides with direct high-speed interconnect support. The performance evaluation of this middleware on a Gigabit Ethernet (GbE) and a Scalable Coherent Interface (SCI) cluster shows experimental evidence of throughput increase. Moreover, qualitative aspects of the solution such as transparency to the user, interoperability with other systems and no need of source code modification can augment the performance of existing parallel Java codes and boost the development of new high performance Java RMI applications.

1. Introduction

Java, due to appealing characteristics such as platform independence, portability and increasing integration into existing applications, is gaining ground in environments where more traditional languages still have their predominance. One of these environments is parallel computing, where the performance is a key aspect. Regarding high performance parallel applications, the most common architecture is the cluster, as it delivers outstanding parallel performance at a reasonable price/performance ratio. Nevertheless, the use of Java parallel applications on clusters is still an emerging option, since the use of inefficient communication middleware has delayed its use. On clusters, efficient communication performance is key to deliver scalability to parallel applications, but Java lacks efficient communication middleware. Even if the cluster nodes were interconnected by a high-speed network, such as SCI, Myrinet, Infiniband and Giganet, Java would not take advantage of this mainly because these interconnection technologies are poorly supported. In fact, Java only fully supports these high-speed interconnects through TCP/IP protocol stack emulations. However, the TCP/IP protocol makes inefficient use of high-speed interconnects. Moreover, the emulation libraries do not take advantage of the high-speed interconnect capabilities to offload the host CPU from communication processing. Thus, the overhead of the TCP/IP emulation libraries is significant [15]. Examples of IP emulations are IP over MX and IP over GM [11] on Myrinet, LANE driver [7] over Giganet, IP over Infiniband (IPoIB) [6] and ScaIP [2] and SCIP [4] on SCI.

Besides the lack of efficient high-speed cluster support, the Java Virtual Machine (JVM) does not provide with efficient communication middleware for cluster computing. Some attempts have been made to develop efficient middleware for Java Distributed Shared Memory (DSM) implementations (e.g., CoJVM [9]) and for high performance Java message-passing libraries (e.g., MPJ Express [1] and MPJ/Ibis [3]). Nevertheless, these libraries do not optimize widely extended APIs and their integration into existing projects is reduced. Regarding the optimization of Remote Procedure Calls (RPCs), some previous efforts have been done in CORBA, especially optimizing high performance CORBA implementations [5]; and in Java RMI, developing several Java RMI implementations (see Section 2). It can also be found related projects on optimizing Java I/O for cluster computing, particularly on high performance Java parallel file systems, being jExpand [13] a good representative.

The goal of our work is to provide with a high performance Java RMI implementation with high-speed network support. This can be done by optimizing the Java RMI protocol for cluster communications under some basic assumptions for the target architecture, and using a high performance sockets library that copes with the requirements

of an RMI protocol for parallel computing on high-speed clusters. As Java RMI is a widely spread API, many Java parallel applications and communication libraries can benefit from this efficient Java RMI implementation. Moreover, the objective is to optimize this protocol with the minimum associated tradeoffs. Thus, the solution is transparent to the user, it does not modify the source code, and it is interoperable with other systems. The tradeoff is that this protocol is limited to clusters with a homogeneous configuration in terms of JVM and architecture and with a shared file system, although most of the high performance clusters are under these conditions.

2. Related Work

Different frameworks have been implemented with the efficiency of RMI communication on clusters as their goal. The most relevant ones are KaRMI [14], RMIX [8], Manta [10] and Ibis [12]. KaRMI is a drop-in replacement for the Java RMI framework that uses a completely different protocol and introduces new abstractions (such as "export points") to improve communications specifically for cluster environments. However, KaRMI suffers from performance losses when dealing with large data sets and its interoperability is limited to the cluster nodes. RMIX extends Java RMI functionality to cover a wide range of communication protocols, but the performance on high performance clusters is not satisfactory. The Manta project is a different approach for implementing RMI, based on Java to native code compilation. This approach allows for better optimization, avoids data serialization and class information processing at runtime, and uses a lightweight communication protocol. Finally, the Ibis framework is a Java solution that extends Java RMI to make it more suitable for grid computing. Looking for performance, Ibis supports some high performance networks and avoids the runtime type inspection.

3. Java RMI Optimization

The design and implementation of a high performance Java RMI library for parallel computing on clusters has been done bearing in mind: (1) the advantages/disadvantages of previous Java RMI optimization projects analyzed in the previous section; (2) the objectives of the proposed Java RMI implementation: the use of a standard API, increase communication efficiency transparently to the user, no source code modification, and interoperability with other systems; and (3) several basic assumptions about the target architecture, high performance computing clusters, such as the use of a shared file system from which the classes can be loaded, homogeneous architecture of the cluster and the use of a single JVM version. Out of these basic assumptions a more efficient communication is not guaranteed.

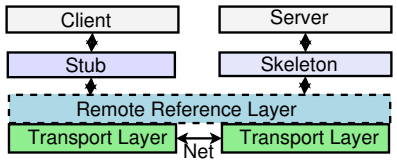

Figure 1. Java RMI layered architecture

Java RMI has been designed following a layered architecture approach. Figure 1 presents, from bottom to top, the transport layer, responsible for managing all communications, the remote reference layer, responsible for handling all references to objects, the stub/skeleton layer, in charge of the invocation and execution, respectively, of the methods exported by the objects; and the client and server layer, also known as service layer. The activation, registry and distributed garbage collection (DGC) services are also part of this service layer.

In order to optimize efficiently Java RMI, an analysis of the overhead of an RMI call has been accomplished. This overhead can be decomposed into four categories: (1) *Network* or transport handling, (2) RMI *Protocol* processing, mainly stub and skeleton operation, (3) *Serialization* and (4) *DGC*. Figure 2 shows a typical Java RMI call runtime's profile. It presents a 3KB Object RMI send on SCI using a high performance Java sockets implementation. Almost 84% of the overhead belongs to *Network*, 12.7% to the costly serialization process, 3.3% to *Protocol*, and a minimal 0.2% to *DGC*.

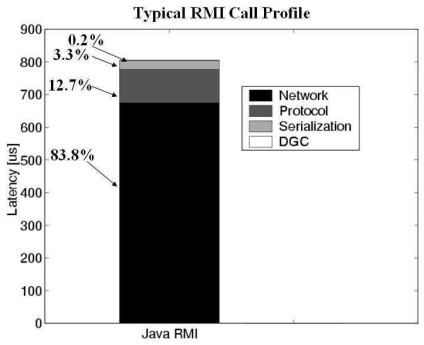

Figure 2. 3KB Object RMI send on SCI

The overhead incurred by the different phases of an RMI call has been considered in relative importance order to proceed with the optimization process. Thus, the proposed improvements are: (1) transport improvements, focused on the use of a high performance Java sockets implementation and

on managing data to reduce sockets delays and buffering, (2) serialization improvements, and (3) object manipulation improvements, changing the protocol to reduce the information about objects that Java RMI protocol includes in each communication, selecting the minimal data to successfully reconstruct a serialized object.

3.1. Transport Protocol Optimization

3.1.1 High Performance Sockets Support

The transport overhead can be reduced through the use of a high performance Java sockets implementation, named Java Fast Sockets (*JFS*) [16]. *JFS* provides with high performance network support on Java (currently SCI support) as it increases communication performance avoiding unnecessary copies and buffering and the cost of primitive data type array serialization, the process of transforming the arrays in stream bytes to send across the network. Most of these optimizations are based on the use of native methods as they obtain higher performance, but its use has associated tradeoffs: as they are more prone to failures and attacks they can compromise slightly the stability and security of the JVM.

JFS increases communication throughput looking for the most efficient underlying communication library in every situation. Moreover, it is portable because it implements a general "pure" Java solution over which *JFS* communications can rely on absence of native communication libraries. The "pure" Java approach obtains, in general, worse performance, but the stability and security of the application (associated tradeoffs for the higher performance of the native approach) is not compromised. The transparency to the user is achieved through Java reflection: the *Factory* for creating Sockets can be set at application startup to the *JFS SocketImplFactory*, and from then on, all sockets communications will use *JFS*. This feature allows Java RMI applications to use *JFS* transparently and without any source code modification. Nevertheless, if a Java RMI implementation wants to take most of the new *JFS* features, optimized communication protocols and native array serialization, it has to change its implementation to use these *JFS* capabilities.

3.1.2 Reduction of Block-data Information

By default, all primitive data that are serialized in a communication are inserted in a data block. Data blocks are used to differentiate data from different objects by setting delimitation marks. To create them, the Java RMI protocol uses a special write buffer, with some locks to help its management. The major goal of using this strategy for primitive data in serialization is to deal correctly with the versioning issue, but after removing some versioning information (improvement that will be described in Section 3.3.1) this block-data strategy is useless. Thus, this strategy has been disabled and the management of the buffer has been simplified, supporting only a minimal control to avoid serialization and deserialization incoherences.

3.2. Serialization Overhead Reduction

3.2.1 Native Array Serialization

In earlier versions of Java RMI, primitive data type arrays had to be serialized in an element-by-element approach. Thus, each byte of their elements (except for booleans) had to be processed using a pair of operations: a boolean *AND* and a right shift to process the next byte (except for the least significant byte). In the last versions, this inconvenience has been partially solved, implementing the native serialization of integer and double arrays, achieving a faster serialization. A generalized approach has been proposed for array serialization, which includes implementing a new generic, native method that can process arrays of any primitive data type as they were byte arrays. This method, in fact, has been implemented in *JFS* and used from the serialization method.

3.3. Object Manipulation Improvements

3.3.1 Versioning Information Reduction

For each object that is serialized, the Java RMI protocol serializes its description, including its type, version number and a whole, recursive description of its attributes; i.e., if an attribute is an object, all its attributes have to be described through versioning. This is a costly process, because the version number of an object has to be calculated using reflection to obtain information about the class. This versioning information is important to deserialize the object and reconstruct it in the receiving node, because sender and receiver can be running different versions of the JVM. Under the assumption of a shared file system and a single JVM, the proposed solution is to send only the name of the class to which the object belongs, and reconstruct the object basing only on the class description at the receiving side. As both sides use the same JVM the interoperability is not compromised.

3.3.2 Class Annotation Reduction

Class annotations are used to indicate the locations (as Java Strings) from which the remote class loaders have to get the serialized object classes. This involves the use of specific URL class loaders. In a high performance cluster environment with a shared file system and a single JVM, it

is useful to avoid annotating classes from the java.* packages, as they can be loaded by the default class loader that guarantees that serialized and loaded classes are the same. This change could also be applied to user classes, but the implementation has been restricted to java.* packages to preserve interoperability. In fact, the optimized RMI is interoperable applying the optimizations for intra-cluster communication, and relying on the default RMI implementation when communicating with a machine outside the cluster. Thus, the ability to use multiple class loaders is not compromised.

3.3.3 Array Processing Improvements

The Java RMI protocol processes arrays as objects, with the consequent useless type checks and reflection operations. The proposed solution is to create a specific serialization method to deal with arrays, hence avoiding that useless processing. Thus, an early array detection check is performed, and the array type is obtained through checking against a primitive data types list. This list has been empirically obtained from the frequency of primitive data type appearance in high performance Java applications. This list (*double, int, float, long, byte, Object, char, boolean*) optimizes the type casting compared to the default list (*Object, integer, byte, long, float, double, char, boolean*). If an object encapsulates a primitive data type array the proposed serialization method will handle this array when serializing the members of the object.

4. Performance Evaluation

4.1. Experimental Configuration

The testbed consists of two dual-processor nodes (PIV Xeon at 3.2 GHz with hyper-threading disabled and 2GB of memory) interconnected via SCI and Gigabit Ethernet (GbE). The SCI NIC is a D334 card plugged into a 64bits/66MHz PCI slot, whereas the GbE is a Marvell 88E8050 with an MTU of 1500 bytes. The OS is Linux CentOS 4.2 with compilers gcc 3.4.4 and Sun JDK 1.5.0_05. The SCI libraries are SCI Sockets 3.0.3, DIS 3.0.3 (SCILib and SISCI), ScaIP 1.0.0 and SCIP 1.2.0.

In order to benchmark communications, Java RMI and Java sockets versions of NetPIPE [17] have been developed (there is neither Java RMI nor Java sockets NetPIPE publicly available version). The results considered in this section are the half of the round trip time of a ping-pong test. Figure 3 shows the sequence diagram of the Java RMI ping and ping-pong tests. It has been taken into account that Java micro-benchmarking has some particularities. Thus, in order to obtain JVM Just in Time (JIT) results from running fully optimized native compiled bytecode, 10000 warm-up iterations have to be executed before the actual measurements. It has been measured the performance of byte and integer arrays as they are frequent communication patterns in Java parallel applications.

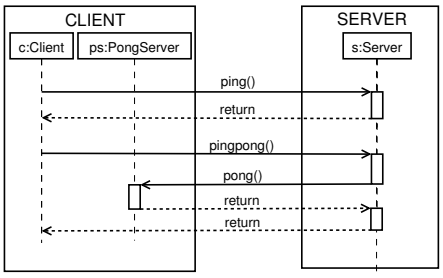

Figure 3. Ping and ping-pong RMI benchmark sequence diagram

Figure 4 shows an overview of the six-layered proposed architecture for high performance Java RMI parallel applications on GbE and SCI. Given components are depicted in dark gray, whereas the contribution presented in this paper, the optimized Java RMI (from now on "Opt RMI"), is colored in light gray. From bottom to top it can be seen the Network Interface Card (NIC) layer, NIC drivers, native sockets, Java sockets and the required IP emulation libraries, Java RMI implementations and high performance Java parallel applications.

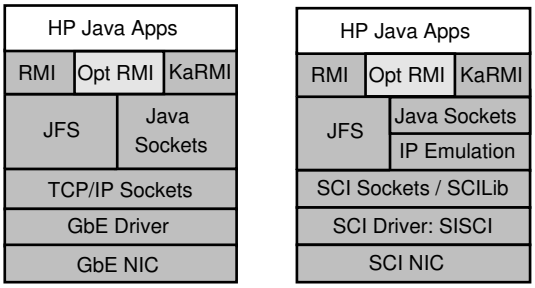

Figure 4. High performance Java RMI parallel applications overview

4.2. Java Sockets Performance Evaluation

Figure 5 shows experimentally measured latencies and bandwidths of the default Java sockets and *JFS* as a function of the message length, for byte and integer arrays on SCI. The bandwidth graph (right) is useful to compare long-message performance, whereas the latency graph (left) serves to compare short-message performance. It can be

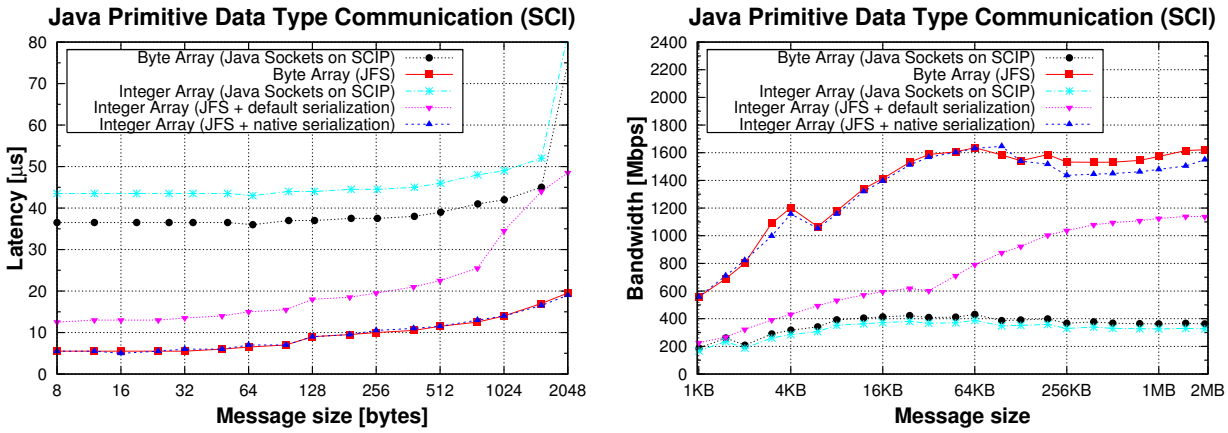

Figure 5. Java sockets primitive data type communication performance

seen that *JFS* clearly outperforms Java sockets, as *JFS* has direct SCI support and Java sockets use an emulation library (*SCIP*) that adds considerable overhead. Furthermore, it can be seen the influence on *JFS* of the different underlying native SCI protocols as their protocol boundaries can be appreciated at 128 bytes and at 8KB message sizes. Regarding integer array communication, the performance obtained when communicating using the native serialization is similar to the byte array communication.

4.3. Java RMI Experimental Results

Figure 6 compares the typical Java RMI runtime's profile (see Figure 2) with the Opt RMI one. In order to compare only at the RMI protocol level, both RMI implementations run on top of *JFS* on SCI. The measures presented are the mean of ten calls, showing a small variance. As it can be seen, *Network* and *Protocol* overheads decrease. The explanation for this behavior is that sending an Object with several attributes (both Objects and primitive data types) can be costlier in Java RMI than in Opt RMI because of the overhead, in terms of data payload, imposed by the versioning information and class annotation. In this case, sending this particular 3KB Object involves a payload almost 3 times larger in Java RMI than in Opt RMI. Nevertheless, the *Serialization* process takes longer in Opt RMI because the RMI server has to obtain the information on how to deserialize the object.

Figure 7 presents the results for RMI integer array communication using KaRMI [14], Java RMI and Opt RMI. Regarding the two upper graphs (GbE), KaRMI shows the lowest latency for short messages ($<$ 1KB), but the highest communication overhead for larger messages. The Opt RMI obtains slightly better results than Java RMI. Regarding SCI graphs, KaRMI and Java RMI on *SCIP* show the

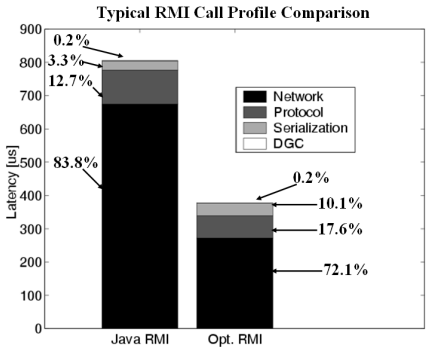

Figure 6. 3KB Object RMI send on SCI

poorest results. However, substituting Java sockets as transport protocol by *JFS* improves the results significantly. In this case, KaRMI presents slightly better performance than Java RMI, for all message sizes. Moreover, KaRMI shows better performance on SCI than on GbE, mainly for being designed to cope with high performance communication libraries, and it obtains poorer results with TCP/IP (GbE). Regarding the RMI bandwidth on SCI, it can be seen that Java RMI and KaRMI performance drops for large messages ($>$ 256KB) caused by a native communication protocol boundary. The Opt RMI presents slightly lower latencies than Java RMI and KaRMI for short messages. For longer messages its performance benefits increase significantly. Moreover, the Opt RMI obtains higher bandwidth optimization on SCI than on GbE as the interconnection network, something independent of the RMI protocol implementation, acts as the main performance bottleneck on GbE, whereas on SCI the major bottleneck is the protocol implementation itself.

Figure 8 presents results of communicating medium-size

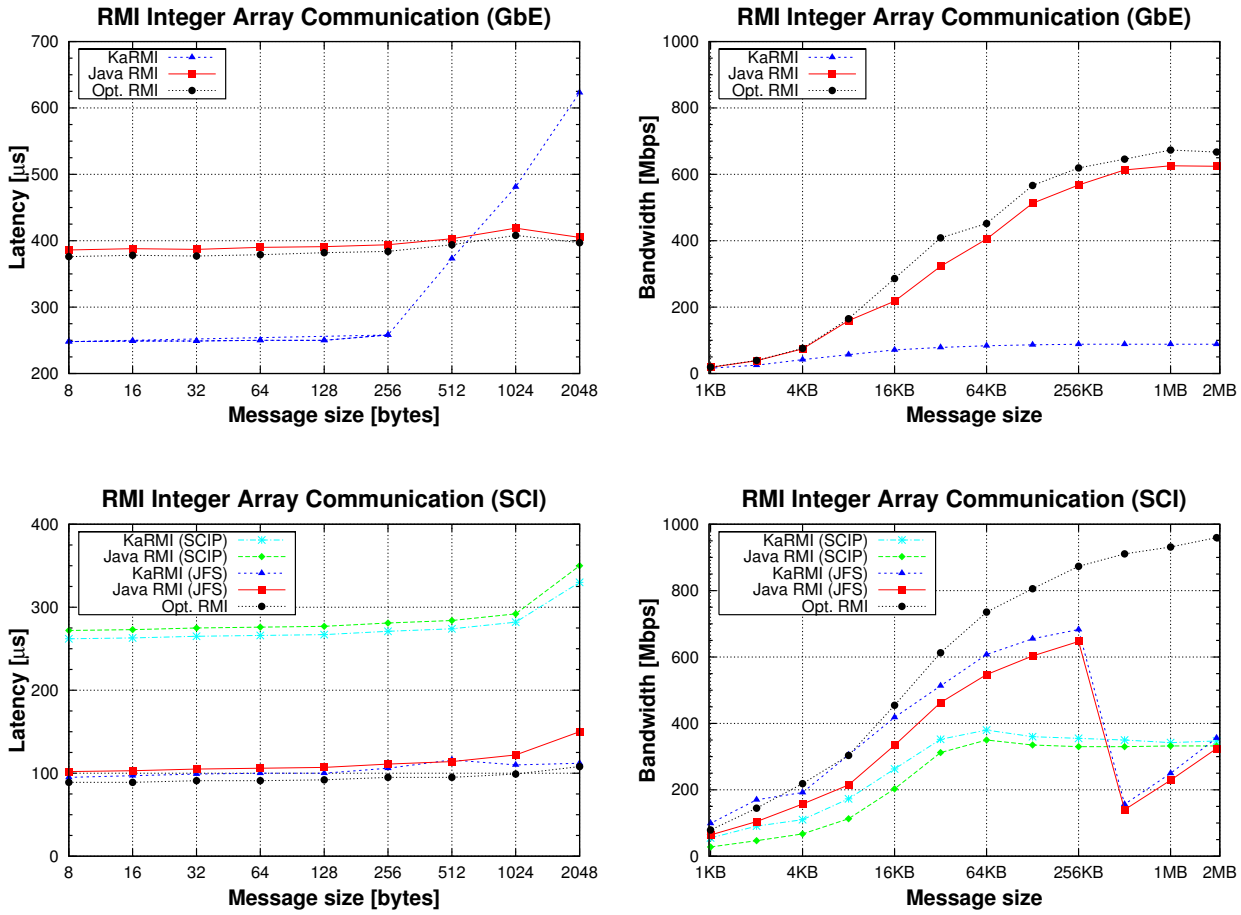

Figure 7. Java RMI integer array communication performance

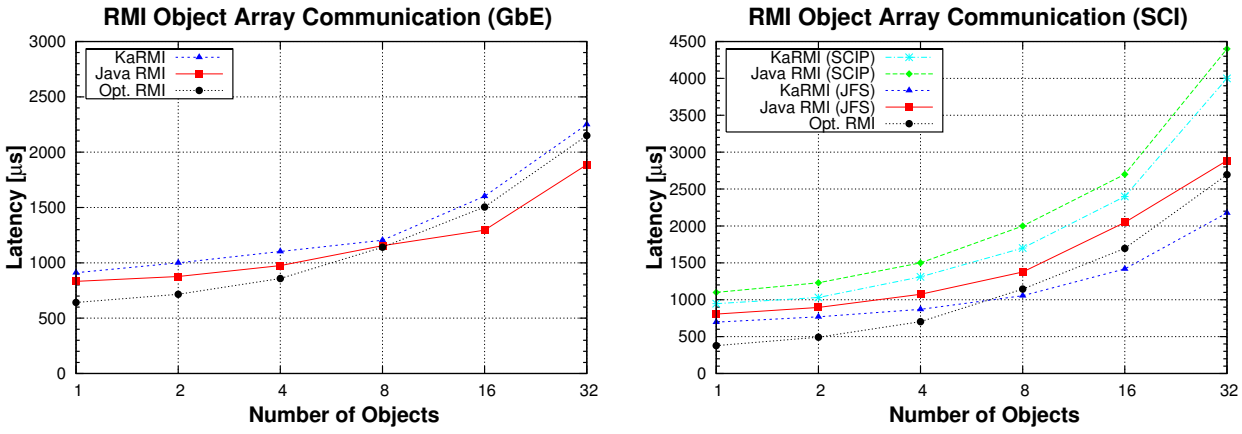

Figure 8. Java RMI object array communication performance

(3KB) Object arrays. The Opt RMI obtains the best performance for arrays of up to 8 objects. Although its latency for a single object is small, there is an important "extra" overhead for each object processed, bigger than for KaRMI and Java RMI overheads. Thus, for arrays from 8 objects Java RMI and KaRMI obtain better performance on GbE and SCI, respectively. The Opt RMI "extra" overhead is caused by its inability to detect if the object sent has changed since previous RMI calls. The object sent did not change in the tests performed.

5. Conclusions

A more efficient Java RMI implementation has been presented. This solution is transparent to the user, interoperable with other systems, it does not need source code modification and it offers a widely spread API. The RMI protocol optimizations have been focused on: (1) reducing block-data information, (2) the use of a high performance Java sockets library (*JFS*) as transport protocol, (3) performing native array serialization, (4) reducing versioning information, and (5) reducing class annotations.

Experimental results have shown that *JFS* greatly improves Java sockets performance, especially on a high-speed interconnect (SCI). Moreover, the RMI protocol optimizations reduce significantly the RMI call overhead, mainly on high-speed interconnection networks and for communication patterns frequently used in high performance parallel applications, especially for primitive data type arrays.

Acknowledgments

This work was funded by the Ministry of Education and Science of Spain under Project TIN2004-07797-C02 and by the Galician Government (Xunta de Galicia) under Project PGIDIT06PXIB105228PR.

References

[1] M. Baker, B. Carpenter, and A. Shafi. MPJ Express: Towards Thread Safe Java HPC. In *Proc. of 8th IEEE Intl. Conference on Cluster Computing (CLUSTER'06)*, pages 1–10, Barcelona, Spain, 2006.

[2] R. G. Börger, R. Butenuth, and H.-U. Hei. IP over SCI. In *Proc. 2nd IEEE Intl. Conf. on Cluster Computing (CLUSTER'00)*, pages 73–77, Chemnitz, Germany, 2000.

[3] M. Bornemann, R. V. van Nieuwpoort, and T. Kielmann. MPJ/Ibis: A Flexible and Efficient Message Passing Platform for Java. In *12th European PVM/MPI Users' Group Meeting, (PVM/MPI'05), LNCS 3666, Springer-Verlag*, pages 217–224, Sorrento, Italy, 2005.

[4] Dolphin Interconnect Solutions, Inc. IP over SCI. Dolphin ICS Website. http://www.dolphinics.com/products/software/sci_ip.html. [Last visited: April 2007].

[5] A. S. Gokhale and D. C. Schmidt. Measuring and Optimizing CORBA Latency and Scalability Over High-Speed Networks. *IEEE Transactions on Computers*, 47(4):391–413, 1998.

[6] IETF Draft. IP over IB. IETF Website. http://www.ietf.org/ids.by.wg/ipoib.html. [Last visited: April 2007].

[7] J.-S. Kim, K. Kim, and S.-I. Jung. SOVIA: A User-level Sockets Layer Over Virtual Interface Architecture. In *Proc. 3rd IEEE Intl. Conf. on Cluster Computing (CLUSTER'01)*, pages 399–408, New Port Beach, CA, 2001.

[8] D. Kurzyniec, T. Wrzosek, V. Sunderam, and A. Slominski. RMIX: A Multiprotocol RMI Framework for Java. In *Proc. 5th Intl. Workshop on Java for Parallel and Distributed Computing (JAVAPDC'03)*, page 140 (7 pages), Nice, France, 2003.

[9] M. Lobosco, A. F. Silva, O. Loques, and C. L. de Amorim. A New Distributed Java Virtual Machine for Cluster Computing. In *Proc. 9th Intl. Euro-Par Conf., (EuroPar'03), LNCS 2790, Springer-Verlag*, pages 1207–1215, Klagenfurt, Austria, 2003.

[10] J. Maassen, R. Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman. Efficient Java RMI for Parallel Programming. *ACM Transactions on Programming Languages and Systems*, 23(6):747–775, 2001.

[11] Myricom Inc. GM/Myrinet. http://www.myri.com. [Last visited: April 2007].

[12] R. V. v. Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H. E. Bal. Ibis: a Flexible and Efficient Java-based Grid Programming Environment. *Concurrency and Computation: Practice & Experience*, 17(7-8):1079–1107, 2005.

[13] J. M. Pérez, L. M. Sánchez, F. García, A. Calderón, and J. Carretero. High Performance Java Input/Output for Heterogeneous Distributed Computing. In *Proc. 10th IEEE Symposium on Computers and Communications (ISCC'05)*, pages 969–974, Cartagena, Spain, 2005.

[14] M. Philippsen, B. Haumacher, and C. Nester. More Efficient Serialization and RMI for Java. *Concurrency: Practice & Experience*, 12(7):495–518, 2000.

[15] G. L. Taboada, J. Touriño, and R. Doallo. Performance Analysis of Java Message-Passing Libraries on Fast Ethernet, Myrinet and SCI Clusters. In *Proc. 5th IEEE Intl. Conf. on Cluster Computing (CLUSTER'03)*, pages 118–126, Hong Kong, China, 2003.

[16] G. L. Taboada, J. Touriño, and R. Doallo. Efficient Java Communication Protocols on High-speed Cluster Interconnects. In *Proc. 31st IEEE Conf. on Local Computer Networks (LCN'06)*, pages 264–271, Tampa, FL, 2006.

[17] D. Turner and X. Chen. Protocol-Dependent Message-Passing Performance on Linux Clusters. In *Proc. 4th IEEE Intl. Conf. on Cluster Computing (CLUSTER'02)*, pages 187–194, Chicago, IL, 2002.