Efficient Java Communication Protocols on High-speed Cluster Interconnects

Guillermo L. Taboada, Juan Touriño and Ramón Doallo
Department of Electronics and Systems
University of A Coruña, Spain
{taboada,juan,doallo}@udc.es

Abstract

This paper presents communication strategies for achieving efficient parallel and distributed Java applications on clusters with high-speed interconnects. Communication performance is critical for the overall cluster performance. Previous efforts at obtaining efficient Java communications have a limited applicability on high-speed interconnects as they are focused on high level APIs like RMI, ignoring the particularities of these systems and their native high performance communication protocols. By relying on a custom Java socket implementation higher degrees of performance can be achieved exploiting high-speed interconnect facilities. Several protocol definitions are presented, looking for obtaining high performance Java communications. Moreover, the quality of the protocol implementations and their design decisions has been thoroughly evaluated on a Scalable Coherent Interface (SCI) and Gigabit Ethernet (GbE) testbed cluster. The results of this analysis have demonstrated that these Java protocols obtain similar results to native communications.

1 Introduction

There is a growing interest of both scientific and enterprise environments in high-speed clusters as they deliver outstanding parallel performance at a competitive cost. High-speed clusters consist of computing nodes connected together by a high-performance interconnect. SCI, Myrinet and GbE are three types of high-speed interconnection technologies commonly used for clusters. Scalability is a key factor to confront new challenges in cluster computing, and it depends heavily not only on the network fabric, but also on the communication middleware. This growing need of efficient communication middleware has led to devote important efforts on this subject, although almost exclusively on native protocols. A thorough work focused on native protocols is that of Verstoep et al. [26], where several implementation issues are studied in order to obtain an efficient use of Myrinet. In that work, a non standard user level communication interface is implemented varying reliability protocols, the Maximum Transfer Unit (MTU), multicast protocols and studying Serial Direct Memory Access (SDMA)-based versus Processor Input/Output (PIO)-based message passing and remote-memory copy. Our proposed approach inherits some optimizations from [26]. Most of these topics are also covered in the related literature, where works are focused on evaluating multicast performance of user level multicast libraries on SCI and Myrinet [16], on obtaining lightweight transport protocols (RMPP [19]) and also on RDMA-based efficient message passing implementations [10]. Nevertheless, despite the predominance of native protocol optimizations, the increasing interest in Java for high performance computing has recently heightened the need for efficient Java communication middleware. This efficiency is of critical importance on high-speed clusters, where the overall performance is quite sensitive to the effect of communication latency and overhead, as Java does not provide direct high-speed interconnect protocols support and has to resort to the TCP/IP protocol stack.

Current efforts at obtaining efficient Java communication middleware are focused on RMI optimizations. RMI startup is significant (around $0.5ms$ in our SCI testbed cluster), being inadvisable its use on high-speed networks. In order to alleviate this drawback, several RMI projects have been undertaken to optimize RMI calls on the Myrinet high-speed interconnect: the Manta compiler [13], the KaRMI library [17] and RMIX [9]. Manta is a Java to native code compiler with an optimized protocol implementation; KaRMI reduces the access latency to a remote object by means of a more efficient object serialization and an optimized protocol implementation and RMIX supports non-IP-based transports via generalized socket factories. Nevertheless, these projects use libraries which are difficult to update or maintain. Another approach is to perform general RMI protocol optimizations increasing RMI efficiency with transformations based on compile time analysis [25].

Different approaches have also been followed in order to optimize Java communications in another context, in Java Distributed Shared Memory (DSM) implementations on clusters. Thus, the CoJVM [12] runtime system relies on the use of the Virtual Interface Architecture (VIA) as its communication protocol aiming to improve application performance. Another project which is worth mentioning is JESSICA2 [27] which supports transparent Java thread migration in a JIT compilation environment. Whereas the previous projects use modified Java Virtual Machines (JVMs), the JavaSplit [5] bytecode rewriting compiler uses unmodified JVMs. The multithreaded Java bytecode is rewritten into parallel bytecode, allowing each JVM to locally optimize the performance. The latter project's target platform is a homogeneous cluster. Nevertheless, Java programming environments are designed to cope with heterogeneous and dynamically changing groups of compute nodes. This assumption reduces significantly communication performance on homogeneous high-speed clusters, in exchange for functionality not always required.

Current efforts in Java communications performance optimization have attained their objectives only partially. Moreover, the lack of high-speed cluster support is decisive for the lack of performance of Java communications on these systems, as a previous work [23] has shown up. From that work, where analytical performance models for Java communications on high-speed clusters are presented, it can be stated that the major drawback is the use of RMI in high-speed clusters, although the use of IP emulation libraries, used due to the lack of native high-speed interconnection protocol support in Java, adds a significant overhead too. For performance improvement the RMI replacement by Java sockets as the base of communication libraries has already been done in gMP [21], Ibis [15] and MPJava [18]. Nevertheless, the native high-speed interconnection protocol support in Java is still under development.

This paper reports on the results obtained from the implementation of a Java communication protocol designed for efficient use of Java on high-speed clusters. After analyzing performance bottlenecks in existing protocol implementations, Java sockets have taken an outstanding role in the protocol design, being the API provided. Thus, the use of this well-known API guarantees the success of the approximation taken.

2 Efficient Communication Protocols

The JVM is composed of both Java and native code and performs its communications through calls to the underlying native libraries, specifically calling the widely spread TCP/IP protocol stack. Unfortunately, TCP/IP places a very heavy load on host CPUs when dealing with high-speed communications. In this context, a high-speed interconnection is a good alternative: Network Interface Cards (NICs) offload communication processing from the host CPU, freeing up valuable CPU cycles for application processing. Moreover, higher performance in terms of both latency and bandwidths can be reached with these network fabrics, although this performance is obtained with their own efficient libraries and protocols. Nevertheless, there is a huge number of applications that use the TCP/IP protocol stack. Rewriting an application to use a hardware specific API is usually not possible. Thus, TCP/IP emulation over high-speed interconnections is the preferred choice. Examples of IP emulations on high-speed interconnections are IP over GM from Myricom over Myrinet, LANE driver [8] over Giganet, IP over Infiniband (IPoIB) [7], ScaIP [3] from Scali over SCI, and SCIP [4] from Dolphin also over SCI. Nevertheless, the performance obtained by IP emulations has not been the expected. In fact, this has been the motivation for researchers to provide socket implementations on top of low level protocols, trying to avoid inefficiencies of the TCP/IP protocol stack.

2.1 High Performance Sockets

The first High Performance socket implementation was FastSockets [20], a stream socket implementation on top of ActiveMessages, a lightweight communication protocol that reduces latency by removing buffering overheads and providing user level access to the network hardware. Afterwards SOVIA [8], a user level socket layer on top of VIA, and Sockets over GbE [2] were introduced. A recent initiative is the Offload Sockets Framework (OSF) whose representative project is the Socket Direct Protocol (SDP) for Infiniband. However, the two projects worthier of deep consideration are Sockets-MX/Sockets-GM [6] and SCISOCKETS [22], because they are directly based on the native low latency protocols provided by the network hardware manufacturers.

Sockets-MX/Sockets-GM are low latency socket implementations over MX and GM, two low level Myrinet libraries. These libraries overcome IP emulations over Myrinet, which involve high system load, by bypassing the TCP/IP protocol stack, which takes a big percentage (up to 50%) of the time spent in communication. Moreover, these projects operate in user level mode. Sockets-GM provides both buffered communication and a zero-copy protocol. In buffered communication one copy of the data is copied into pre-registered buffers and the message delivery is handled by the Myrinet NIC. In the zero-copy protocol data is exchanged directly from application to application buffers using GM RDMA functions. Thus, one-copy protocol trades off low CPU load for low latency whereas zero-copy protocol cuts down sys-

tem load (high bandwidth rates can be exchanged with low CPU loads). A sensible choice between protocols involves using the one-copy protocol for latency sensitive applications, and the zero-copy protocol for high-bandwidth driven applications. This protocol choice, as well as the polling/blocking receiving strategy, can be set dynamically.

SCISOCKETS provides SCI with a socket implementation on top of SISCI and GENIF low level interfaces. SISCI [1] is the user level interface for SCI which implements basic mechanisms to share memory segments between nodes and to transfer data between them. GENIF is the kernel level interface for SCI. SCISOCKETS' transparent access semantics to remote memory in combination with CRC checksums greatly reduces overhead for very small messages. In fact, this implementation obtains much lower latencies ($4\mu s$) than other socket projects using similar hardware: Sockets-GM ($21\mu s$) and SOVIA on Giganet ($10\mu s$). Differences in performance could be attributed to both the network fabric and to the socket implementation, but from evaluations of native protocols performance between SCI and Myrinet [14], and between Myrinet and Giganet [11], it can be seen that the differences are mainly explained by the socket implementation. Aside from startup times, special emphasis has been put on allowing fully transparent integration with existing applications. By using the preloading mechanism of the dynamic library loader it is possible to make applications use SCISOCKETS implementation as the default socket implementation. This transparency to the user in such a convenient way (without changes to the code and even without recompilation or relinking) provides immediate speedups for both user and kernel space applications.

3 Java Fast Sockets

In order to support high performance interconnection technologies on Java a High Performance Java Socket implementation, named Java Fast Sockets (`JFS`), has been developed. Java applications can not run on top of native high performance sockets as the layers involved in Java communications (Java classes in `java.net` package, native functions in `libnet.so` library and JVM native functions) perform calls that usually are not supported by high performance socket libraries. Thus, `JFS` has to implement communication mechanisms in order to perform communications over these underlying protocols.

The implementation of `JFS` has posed several issues that can be classified into Java- and native-related issues. Java-based issues consist of the trade-off between portability and performance of the solution proposed, the use of the Java NIO facilities, the use of a lightweight communication layer and the transparency to the user of the solution proposed. Native-related issues are the convenience of resorting to native methods in order to improve performance, the interplay between native and Java code, the choice between user or kernel level of the underlying library, changes in the underlying protocols boundaries and communication strategies (e.g., the activation of Nagle's algorithm).

In order to obtain an efficient and portable solution the strategy to follow is the Ibis-based approach. Thus, an efficient pure Java solution is implemented together with native solutions to access low-level native protocols through JNI. At establishing connections, `JFS` will look for a native-based protocol. If any, it will take over the communications. Otherwise, `JFS` will resort to the pure Java efficient socket implementation. This design allows new protocols to be plugged in.

The use of `JFS` in an application is achieved replacing the default `SocketFactory`, in charge of creating sockets using the default socket implementation (`PlainSocketImpl`), by `JFSFactory`, which creates sockets using the JFS implementation (`JFSImpl`).

The lightness of the solution without lose of functionality is key to deliver the low latency and high bandwidth of the high-speed interconnection to applications. The transparency to the user is achieved by means of a small Java application that allows through Java's reflection to invoke the `main()` method of the main class of the application after replacing the default sockets implementation by `JFS`. The most interesting code appears in the following box:

Listing 1. Replacing default Sockets by `JFS`

```
SocketImplFactory factory = new JFSFactory();
Socket.setSocketImplFactory(factory);
ServerSocket.setSocketFactory(factory);

Class cl = Class.forName(className);
Method method = cl.getMethod("main", argsTypes);
method.invoke(null, parameters);
```

In order to extend `java.net.SocketImpl` and be fully compatible with Java sockets there is a minimum number of classes that must be implemented in the Java sockets implementation. The specific behavior of the library must be contained in these four classes:

Listing 2. Base classes implemented in `JFS`

```
jfs.JFSSocketImplFactory
jfs.JFSSocketImpl
jfs.SocketInputStream
jfs.SocketOutputStream
```

3.1 Efficient Java NIO

The use of the Java NIO classes, such as data containers (buffers), I/O channels, selectors and selection keys is an important issue in JFS. For instance, the use of direct byte buffers is key in achieving the zero-copy protocol, avoiding the memory copy overhead. This is because the contents of direct byte buffers reside outside of the normal garbage-collected heap. Thus, JVM can perform native I/O operations directly upon direct byte buffers, avoiding the copy of buffer's content using an intermediate buffer whenever an invocation of one of the underlying operating system's native I/O operations is done. Nevertheless, direct buffers usually have higher allocation and deallocation costs than non-direct buffers. It is therefore recommended that direct buffers be allocated primarily for large, long-lived buffers that are subject to the underlying system's native I/O operations.

Another remarkable characteristic of byte buffers is the definition of methods for reading and writing values of all other primitive types, except boolean. This type translation depends on the buffer's current byte order, which initially is always BIG_ENDIAN, and must be the same as the underlying architecture. It has been experimentally observed that mismatching byte orders between buffers and the system imposes important penalties ($> 100\%$ handling overhead increase). For access to sequences of values of the same type a view of a given byte buffer can be created. A view buffer is another buffer whose content is backed by a byte buffer, keeping its original direct or non direct properties. Most Java communications systems use the standard serialization method which needs intermediate buffers, but view buffers remove this need and even the serialization process. Therefore, the use of communicating direct buffers in JFS is expected to yield measurable performance gains. Another important feature of NIO is the use of selectors and selection keys, which together with selectable channels define a multiplexed, non-blocking I/O facility. These are worth characteristics for highly scalable efficient server technology.

Once the importance of NIO has been established, it is time to analyze its participation in the solution. The pure Java solution is based on Java NIO SocketChannels and direct byte buffers, whereas the optimized native solution is based on an extension of the I/O streams of the java.io package, which will deal with direct byte buffers through native methods. These choices have revealed encouraging experimental results.

3.2 Native Protocol Integration in JFS

In order to achieve higher degrees of optimization JFS has to resort to native protocols. In this matter, the selection of a native protocol poses several questions and implementation challenges. From the analysis of the SCI native protocols, SCILib [22] has been selected as the resorting layer for JFS over SCI.

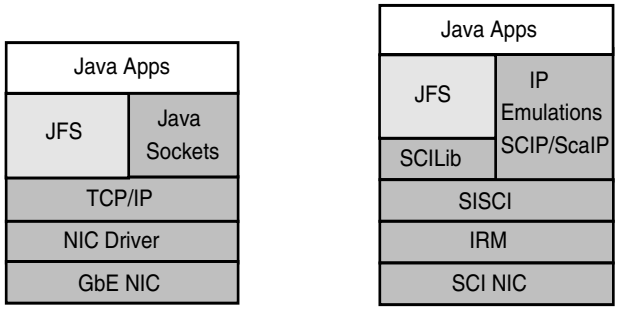

Figure 1. Overview of JFS architecture

A schema of the architecture of the solution proposed for SCI and GbE is shown in Figure 1. SCILib is a communication protocol on top of SISCI that offers unidirectional message queues. SISCI is a quite low level API, whereas SCISOCKETS, which is based on SCILib, provides functionality not required by JFS and imposes a higher communication overhead. The original SCILib has been extended in order to meet JFS requirements. The change in the underlying protocol boundaries has obtained performance gains only in the Short/Long protocol boundary. The Inline/Short protocol boundary, 116 bytes, is the longest message that the SCI hardware can send in a physical transaction. SCILib can operate both in user and kernel level mode, focusing on lower startups and higher transfer ratios respectively. The native protocol under JFS takes advantage of this by changing at runtime the value of the environment variable LD_PRELOAD, which points to the user or kernel level dynamic libraries. Thus, for messages shorter than a configuration variable, which is by default 8KB, the native protocol uses user level libraries to obtain lower startups. For messages longer than 8 KB, kernel level protocols obtains better transfer ratios.

The overhead of running these native protocols in kernel level is around $1\mu s$, a quite important value taking into account that the SCILib startup time is around $3\mu s$. The SCILib user level is based on SISCI, whereas the kernel level version is based on GENIF. Moreover, the use of efficient data movement techniques through the Java Native Interface (JNI) is of crucial importance.

Furthermore, the socket method implementation in JFS of setPerformancePreferences() and setTcpNoDelay() require to pass their parameters to the native library. With setPerformancePreferences() the importance of latency with respect to bandwidth can be established in order to increase the performance of the

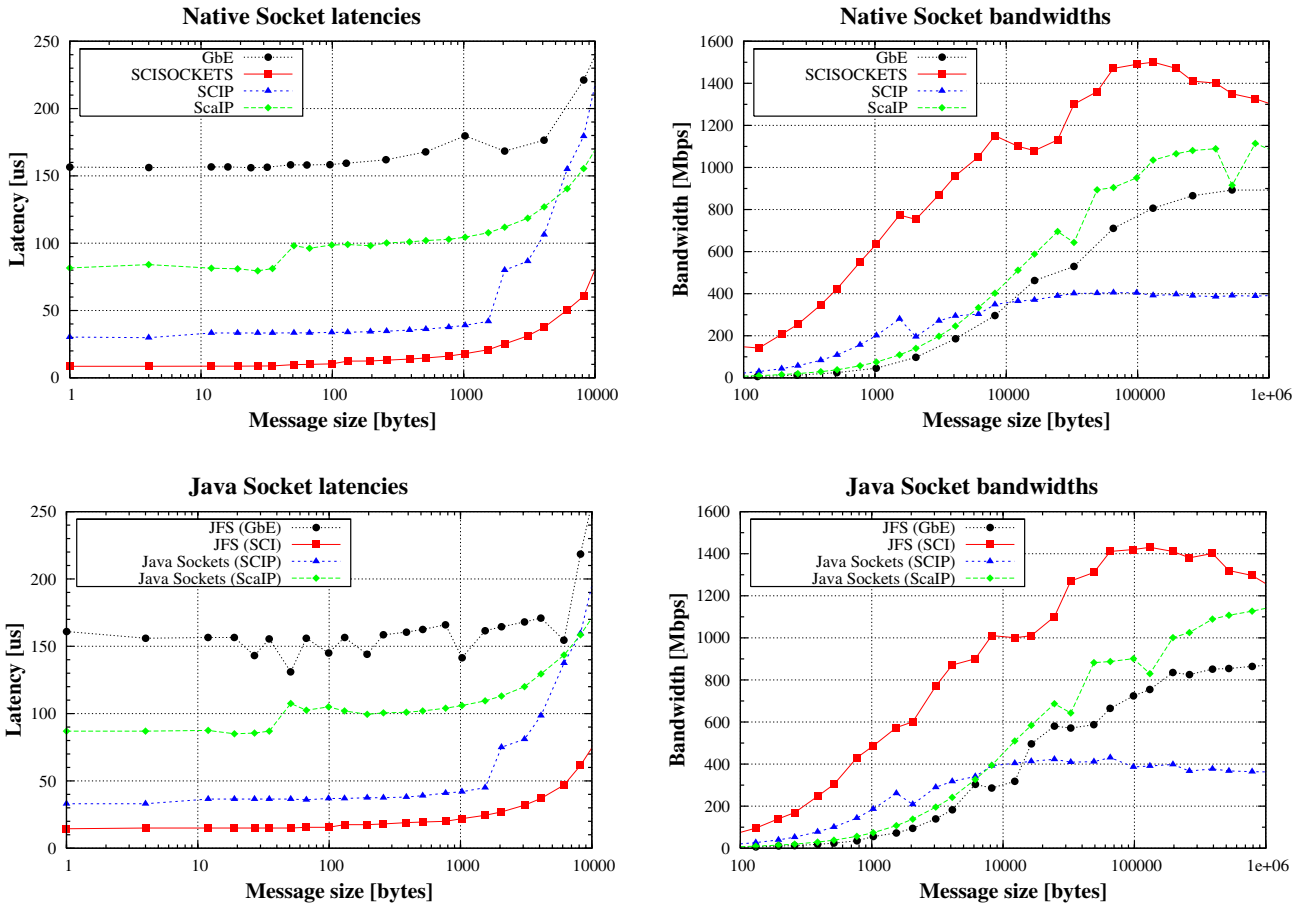

Figure 2. Measured socket latencies and bandwidths

preferred choice. The method `setTcpNoDelay()` enables Nagle's algorithm.

4 Experimental Evaluation

In this section it has been accomplished a performance evaluation of `JFS` and of both current Java and C TCP/IP high performance communication strategies.

4.1 Experiment Configuration

Our testbed consists of two dual-processor nodes (PIV Xeon at 2.8 GHz with hyper-threading disabled and 2GB of memory) interconnected via SCI and GbE. The SCI NIC is a D334 card plugged into a 64bits/66MHz PCI, whereas the GbE is a Marvell 88E8050 with a MTU of 1500 bytes. The OS is Linux Centos 4.2 with kernel 2.6.9 and compilers gcc 3.4.4 and Sun JDK 1.5.0_05. The SCI libraries are SCISOCKETS 3.0.3 and DIS 3.0.3 (IRM, SISCI, SCILib and Mbox). The IP emulation libraries used have been the IP emulation from Scali, ScaIP 1.0.0, and the IP emulation from Dolphin, SCIP 1.2.0. This latter library needs a kernel 2.4, so the testbed has also a 2.4.32 kernel.

In order to benchmark communications, NetPIPE [24] (Network Protocol Independent Performance Evaluator) has been used, more specifically the NetPIPE TCP module for C benchmarks and our own version of the Java TCP module (there is no public version of Java NetPIPE). These benchmark modules provide an easy way to measure the performance of TCP point-to-point communications while varying the socket buffer sizes. The results considered are the half of the round trip time of a ping-pong test.

4.2 Performance Results

Figure 2 shows experimentally measured latencies and bandwidths of native and Java sockets as a function of the message length, for the different communication protocols. Bandwidth graphs (right side of the figure) are useful to compare long-message performance, whereas latency graphs (left side of the figure) serve to compare short-message performance. The upper graphs show native communications, whereas the lower graphs show the obtained measurements for Java communications.

4.3 Analysis of Experimental Results

In order to analyze the goodness of the JFS implementation it has been performed a performance evaluation of native protocols over SCI and GbE. Results obtained from JFS over GbE have been quite similar to the results obtained by Java sockets over this interconnect (results not shown for clarity purposes). The main difference between using JFS or not lies on the support of options not implemented in Java sockets like the new options `setPerformancePreferences()`.

The IP emulation results are shown to strengthen what has been previously asserted, that IP emulations in general, and over SCI in particular, show poor performance. Nevertheless, several differences exist between the two IP emulations that deserve to be discussed and can be observed from the two upper graphs in Figure 2. First, ScaIP default configuration (MTU = $32KB$) achieves better results, whereas SCIP needs changes in the default configuration in order to obtain optimal results. It has been experimentally checked that the SCIP optimal configuration in our testbed needs the allocation of 25 $2KB$ communication buffers. With smaller and larger buffers the latency increases significantly. The default configuration, 10 $32KB$ buffers, shows poorer performance. The optimal configuration leads to increase up to five times the bandwidth of messages above 2KB. Secondly, SCIP obtains lower startup times than ScaIP, $30\mu s$ vs. $81\mu s$, respectively. Nevertheless, ScaIP achieves higher asymptotic transfer rates than the optimal SCIP configuration, $1108Mbps$ vs. $391Mbps$. The lower latency of SCIP and the higher bandwidth of ScaIP make SCIP more suitable for short messages (under 5KB) and ScaIP for larger messages. This behavior is explained by the fact that ScaIP is a kernel level library and SCIP has been tested using the user level version. Usually kernel level libraries show higher startups and transfer ratios than their user level counterparts.

Results obtained from a high performance socket implementation (SCISOCKETS) are depicted to show the improvement that these libraries can provide compared to IP emulations.

In the two lower graphs of Figure 2 Java sockets measurements can be observed. The purpose of showing these results is to acquire at a first glance an idea of the overhead imposed by the JVM over the native sockets communications. This overhead can be clearly observed in short messages, ranging from 6 to $10\mu s$. Nevertheless, with large messages the runtime differences between native and Java benchmarks are relatively less important: Java decreases transfer ratios around $1-4\%$. Nevertheless, the most interesing aspect is that in these graphs it can also be seen the performance that Java sockets obtained over SCI, prior to our work, when Java applications had to resort to IP emulations. With JFS this performance significantly improves, reducing start-up communication latencies up to 7 times (with respect to ScaIP) and improving 4 times the SCIP bandwidth for long messages. Indeed, the performance shown by JFS is quite similar to the performance achieved by the native high performance sockets.

For obtaining these results JFS can be configured in three different ways: changing the buffer size (the default value is 160 KB), using Nagle's algorithm and configuring the polling/interruption strategy on the receiving side. It has been observed that changing the buffer size to a value of 512KB increases the asymptotic transfer rate around a 4% without side effects. Nagle's algorithm has not been activated because of the latency increment for small messages (in our system $3\mu s$) and the unperceived latency reduction for larger messages. Related to the third option, it must be said that this library follows a hybrid approximation between polling and interruptions. JFS polls for a message during a certain period of time and after that, if it has not received anything, waits for an interruption. The default polling time is $80\mu s$ and it has been experimentally checked that it is the optimal value in the testbed; less polling time increases the latency for short messages without benefits, whereas longer polling decreases slightly the bandwidth. The measurements shown in Figure 2 have been obtained with the optimized JFS configuration.

5 Conclusions

A high performance Java sockets communication protocol named Java Fast Sockets (JFS) has been presented. The novelty of this approach is that the Java sockets API can now take advantage transparently and efficiently of high-speed cluster networks. The implementation of high performance native communication protocols on high-speed clusters allows JFS to resort to them in presence of this hardware. New communication protocols can be plugged into JFS transparently. The benefits of its implementation decisions, such as resorting to native protocols and the use of JNI and Java NIO facilities have been shown. Thus, the use of Java NIO direct buffers for

communications has yielded measurable performance gains. Experimental results have shown that short messages benefit specially from these optimizations. In fact, the higher optimization occurs at startup time, where a JFS implementation over a high-speed interconnect (SCI) has reduced latencies up to 7 times (with respect to ScaIP) and has increased 4 times the bandwidth (with respect to SCIP). Indeed, the performance shown by JFS is quite similar to the performance achieved by the native high performance sockets SCISOCKETS. JFS obtains startup latencies as low as $11\mu s$ on our SCI testbed.

To sum up, it may be said that advances in both native and Java-related design issues can deliver performance competitive with native codes on clusters. Java is not only a good choice as a common platform in heterogeneous network environments, but also in homogeneous systems, where portability issues are less important and the emphasis is on performance. This reduction in portability requirements enables the use of highly optimized communication libraries.

Our current work involves tackling interoperability issues between native libraries (SCILib and High Performance Sockets) and the JVM, and a native implementation of data transfers of primitive types other than bytes. Moreover, a key objective is to substitute the current design with a design based on Selectable Channels SPI. Another ongoing work is the management of basic data types other than bytes, non-direct NIO buffers and the implementation of some basic collective message-passing primitives.

Acknowledgments

This work was funded by the Ministry of Education and Science of Spain under Project TIN2004-07797-C02 and under an FPU grant AP2004-5984.

References

[1] ESPRIT Project 23174 — Software Infrastructure for SCI (SISCI), 1998.

[2] P. Balaji, P. Shivan, P. Wyckoff, and D. K. Panda. High Performance User Level Sockets over Gigabit Ethernet. In *Proc. 4th IEEE Intl. Conf. on Cluster Computing (CLUSTER'02)*, pages 179–186, Chicago, IL, 2002.

[3] R. 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.

[4] Dolphin Interconnect Solutions, Inc. IP over SCI. Dolphin ICS Website.

[5] M. Factor, A. Schuster, and K. Shagin. JavaSplit: a Runtime for Execution of Monolithic Java Programs on Heterogenous Collections of Commodity Workstations. In *Proc. 5th IEEE Intl. Conf. on Cluster Computing (CLUSTER'03)*, pages 110–117, Hong Kong, China.

[6] M. Fischer. Sockets-GM. Overview and Performance. Myricom website.

[7] IETF Draft. IP over IB. IETF Website.

[8] 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)*, New Port Beach, CA, 2001.

[9] D. Kurzyniec, T. Wrzosek, V. Sunderam, and A. Slominski. RMIX: A Multiprotocol RMI Framework for Java. In *Proc. 5th Int. Workshop on Java for Parallel and Distributed Computing (JAVAPDC'03), 17th Int. Parallel & Distributed Processing Symp. (IPDPS'03)*, page 140 (7 pages), Nice, France, 2003.

[10] J. Liu, J. Wu, and D. K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. In *Proc. of the 17th Intl. Conf. on Supercomputing (ICS'03)*, pages 295 – 304, San Francisco, CA, 2003.

[11] M. Lobosco, V. S. Costa, and C. L. de Amorim. Performance Evaluation of Fast Ethernet, Giganet, and Myrinet on a Cluster. In *Proc. 2nd Int. Conf. on Computational Science (ICCS'02), LNCS 2329, Springer-Verlag*, pages 296–305, Amsterdam, The Netherlands, 2002.

[12] 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)*, pages 1207–1215, Klagenfurt, Austria, 2003.

[13] 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.

[14] S. Millich, A. George, and S. Oral. A Comparative Throughput Analysis of Scalable Coherent Interface and Myrinet. In *Proc. of the Workshop on High-Speed Local Networks (HSLN) 27th IEEE Conf. on Local Computer Networks (LCN'02)*, pages 691–702, Tampa, FL, November 2002.

[15] R. V. v. Nieuwpoort, J. Maassen, R. Hofman, T. Kielmann, and H. E. Bal. Ibis: an Efficient Java-based Grid Programming Environment. In *ACM Java Grande - ISCOPE 2002 Conf., Seattle (WA)*, pages 18–27, 2002.

[16] S. Oral and A. George. A User-level Multicast Performance Comparison of Scalable Coherent Interface and Myrinet Interconnects. In *Proc. of the Workshop on High-Speed Local Networks (HSLN) 28th IEEE Conf. on Local Computer Networks (LCN'03)*, pages 110–117, Bonn, Germany, 2003.

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

[18] B. Pugh and J. Spacco. MPJava: High-Performance Message Passing in Java using Java.nio. In *Proc. 16th Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC'03)*, pages 323–339, College Station, TX, 2003.

[19] R. Riesen and A. B. Maccabe. Simple, Scalable Protocols for High-Performance Local Networks. In *Proc. of the Workshop on High-Speed Local Networks (HSLN) 28th IEEE Conf. on*

Local Computer Networks (LCN'03), pages 640–641, Bonn, Germany, 2003.

[20] S. H. Rodrigues, T. E. Anderson, and D. E. Culler. High-Performance Local-Area Communication With Fast Sockets. In *Proc. of Winter 1997 USENIX Symposium*, pages 257–274, Anaheim, CA, 1997.

[21] B. Schmidt, L. Feng, A. Laud, and Y. Santoso. Development of Distributed Bioinformatics Applications with GMP. *Concurrency: Practice & Experience*, 16(9):945–959, 2004.

[22] F. Seifert and H. Kohmann. SCI SOCKETS - A Fast Socket Implementation over SCI. Dolphin ICS Website.

[23] 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.

[24] 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.

[25] R. Veldema and M. Philippsen. Compiler Optimized Remote Method Invocation. In *Proc. 5th IEEE Intl. Conf. on Cluster Computing (CLUSTER'03)*, pages 127–136, Hong Kong, China, 2003.

[26] K. Verstoep, R. Bhoedjang, T. Rühl, H. Bal, and R. Hofman. Cluster Communication Protocols for Parallel-programming Systems. *ACM Transactions on Computer Systems*, 22(3):281–325, 2004.

[27] W. Zhu, C.-L. Wang, and F. C. M. Lau. JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support. In *Proc. 4th IEEE Intl. Conf. on Cluster Computing (CLUSTER'02)*, pages 381–388, Chicago, IL, 2002.