Non-blocking Java Communications Support on Clusters

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 supporting efficient non-blocking Java communication on clusters. The communication performance is critical for the overall cluster performance. It is possible to use non-blocking communications to reduce the communication overhead. Previous efforts to efficiently support non-blocking communication in Java have led to the introduction of the Java NIO API. Although the Java NIO package addresses scalability issues by providing `select()` like functionality, it lacks support for high speed interconnects. To solve this issue, this paper introduces a non-blocking communication library to efficiently support specialized communication hardware. This library focuses on reducing the startup communication time, avoiding unnecessary copying, and overlapping computation with communication. This project provides the basis for a Java Message-passing library to be implemented on top of it. Towards the end, this paper evaluates the proposed approach on a Scalable Coherent Interface (SCI) and Gigabit Ethernet (GbE) testbed cluster. Experimental results show that the proposed library reduces the communication overhead and increases computation and communication overlapping.

1 Introduction

There is a growing interest shown by scientific and enterprise community in commodity clusters. The reason is that they deliver outstanding parallel performance at a competitive cost. A cluster consists of computing nodes connected together by a network fabric—usually a high-performance interconnect like SCI, Myrinet, or GbE. Scalability is a key factor to confront new challenges in cluster computing—it depends heavily not only on the network fabric, but also on the communication middleware.

This growing need of efficient communication middleware has led the community to devote significant efforts on this subject, although almost exclusively on native protocols. A thorough work focused on native protocols is that of Verstoep et al. [1], where several implementation issues are studied in order to obtain an efficient use of Myrinet. In this study, a non standard user level communication interface is implemented varying reliability protocols, maximum transfer unit, multicast protocols and studying Serial Direct Memory Access (SDMA)-based versus Processor Input/Output (PIO)-based message passing and remote-memory copy. The proposed approach inherits some optimizations from [1].

Despite the dominance of native protocol optimizations, the increasing interest in Java for high performance computing has recently increased the need for efficient Java communication middleware. This efficiency is of critical importance on clusters, especially on System Area Networks (SANs). In such environments, the overall performance is quite sensitive to the communication overhead [2]. As Java does not provide direct SAN protocols support, socket libraries and IP emulation layers have to be implemented on top of the high performance low-level SAN protocols. Moreover, communication is a major bottleneck in parallel Java applications. Thus, supporting efficient non-blocking communication on clusters, especially on SANs, appears to be a key objective to improve Java communication efficiency. As High Performance Cluster support has been traditionally focused on the blocking Java Remote Method Invocation (RMI) a follow-up aimed at supporting efficient non-blocking Java communications on clusters appears to be a promising research topic. This paper reports on the results obtained from the implementation of a non-blocking Java communication library with High Performance Cluster support.

1.1 Related Work

Previous efforts at obtaining non-blocking Java communications, NBIO (`http://www.eecs.harvard.edu/~mdw/proj/java-nbio/`) and Jaguar [3] have led to the introduction of some facilities in Java NIO to address scalability issues in server applications. Current efforts in non-blocking Java communications are more oriented to support communication for higher level libraries rather than constitute a messaging system *per se*. Therefore, their importance is centred around their projects. This is the case for `mpjdev` [4] used in HPJava [5] and of `xdev` used in a Java messaging-passing system, MPJ Express [6]. The `xdev` library is highly scalable due to the use of Java NIO and an efficient buffering scheme [7], supporting also Myrinet communications. mpiJava [8] is an object-oriented Java wrapper library to MPI implementations providing similar performance to native MPI implementations. Thus, non-blocking primitives present in native MPI implementations can be used efficiently in Java. Another Java Message-passing library that support non-blocking communication and Myrinet clusters is MPJ/Ibis [9].

2 Efficient Communication Libraries on Clusters

In the context of High Performance Cluster Computing the use of Network Interface Cards (NICs) is an attractive option as they offload communication processing from the host CPU. This helps in 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 usually only obtained by using their own efficient protocols. Figure 1 shows an overview of some protocols on SCI and GbE. Given components are colored in dark grey, whereas contributions presented in this paper are depicted in light grey.

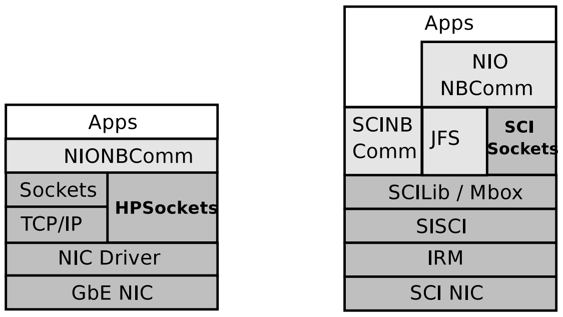

Fig. 1. Overview of communication libraries on popular cluster interconnects

Regarding SCI, the `IRM` driver interacts directly with the hardware, whereas `SISCI` provides resource management and a higher level API. This library implements basic mechanisms to share memory segments between nodes and to transfer data between them. `SCILib` is a communication protocol that offers unidirectional message queues. Depending on the message size `SCILib` presents three communication protocols: *inline*, *short* (both one-copy protocols) and *long* (zero-copy protocol). `Mbox` is a library that provides with remote interrupt mechanisms, so the target side can wait explicitly for an event or register a callback routine, whereas the initiator side triggers the event. SCI SOCKET [10] is a High Performance Socket implementation on SCI obtaining startup times as low as $4 \mu s$ on commodity clusters.

Regarding GbE, its socket implementations are usually not very efficient. Various projects tried to reduce the overhead of these protocols by means of High Performance Sockets implementations—much like SCI SOCKETS on SCI. These High Performance Sockets projects are usually lightweight communication protocols focused on reducing latency by removing buffering overheads and protocol processing. In this context, some efforts include FastSockets [11], SO-VIA [12], Sockets over GbE [13], and GAMMAsockets [14].

3 Designing Java Communication Libraries on Clusters

A non-blocking Java communication library, named `NBComm`, has been designed for efficient use of Java on clusters. This library abstracts the lower network layer and supports higher middleware libraries or runtime systems. As a result, such systems and libraries can be easily ported to different interconnects. This implementation constitutes the basis for a Java Message-passing library, as it

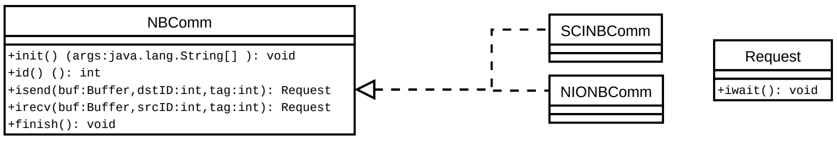
NBComm
+init() (args:java.lang.String[]): void
+id() (): int
+isend(buf:Buffer,dstID:int,tag:int): Request
+irecv(buf:Buffer,srcID:int,tag:int): Request
+finish(): void
SCINBComm
NIONBComm
Request
+iwait(): void

Fig. 2. `NBComm` API

provides a communication library with efficient non-blocking primitives along with good performance on different cluster interconnects.

This library is focused on reducing latency, avoiding unnecessary copying, and computation/communication overlapping. Figure 2 shows its object diagram, which consists of `NBComm`, the abstract communicator class that defines the general behaviour of the communication methods, and two implementation classes, `SCINBComm` and `NIONBComm`, for supporting different communication libraries. In this case, `SCINBComm` follows a native approach, implementing communications in native code over `SCILib` with a lightweight Java layer on top of it, whereas `NIONBComm` is a pure Java NIO-based solution. These classes implement the general behaviour in function of the underlying communication libraries: `init()` initialises the communicator object and `finish()` finalizes the communicator object; `id()` gets the identification for each process; `iwait()` waits for the completion of a communication; and `isend()` and `irecv()` perform communication using a *direct* `ByteBuffer` (a Java NIO buffer) which belongs to the class `Buffer`. These buffers can be accessed directly, and more efficiently, from native applications as they may reside outside of the normal garbage-collected heap. The `Buffer` class is similar to the Java NIO Buffer.

Listing 1.1. Non-blocking communications code example

```
public static void main(String args[]) throws Exception{
  int tag=10, size=10, capacity=40;
  int[] data = new int[size];
  NBComm nbComm = NBCommFactory.getNBComm("sci");
  nbComm.init(args);
  int myId = nbComm.id();
  int peer = 1-myId;
  Buffer buf = new Buffer(BufferFactory.getBuffer(capacity));
  if (myId==0){
        buf.write(data,0,data.length);
        Request req = nbComm.isend(buf,peer,tag);
        req.iwait();
  } else if (myId==1) {
        Request req = nbComm.irecv(buf,peer,tag);
        req.iwait();
        buf.read(data,0,data.length);
  }
  nbComm.finish();
}
```

Listing 1.1 shows a code example of a parallel application that uses `SCINBComm` (`getNBComm("sci")`). This application performs a non-blocking point-to-point communication. The `init()` and `finish()` functions serve as barrier because these methods do not return the control to the application until all processes involved in the parallel application have reached those points.

4 Implementing Efficient Non-blocking Communication

`NBComm` uses a dedicated thread for communication (`receptor_thread`) which is responsible for receiving messages. This thread is implemented in `SCINBComm` in native code whereas in pure Java for `NIONBComm`. Listing 1.2 shows its operation pseudocode.

There are two possible ways to implement message arrival notification depending on the implementation. The first, the native solution, is through a `callback()` function or through an event that is registered for being triggered every time a message arrives. The second, the pure Java solution, is checking arrival notification using Java NIO Selector. Each message is uniquely identified by <srcid,tag>, and `irecv()` requests posted and not actually received are in the `posted_messages` linked list.

Listing 1.2. Pseudocode of the `receptor_thread` operation

```
WHILE NBComm.finish() is not called
  IF pending_messages = 0 THEN
     wait until message arrival notification
  END IF
  receive message header
  check if this message irecv has been posted
  IF posted THEN
    receive message data in the irecv Buffer buf
    delete irecv post from posted_messages
  ELSE
    receive message data in temporal buffer
    add received post to posted_messages
  END IF
  notify the message reception to the waiting requests
END WHILE
```

A problem in this implementation is that `NBComm` subclasses replicate some code as it appears as Java code in `NIONBComm` and as native code in `SCINBComm`. Thus, this code can not be factorized in the superclass `NBComm` making it harder to maintain the source code. A proposed solution consists of moving the interconnection hardware support to a lower API level (Java sockets) and using `NIONBComm` over these low level libraries.

4.1 Java Sockets with SAN Support

In order to support high performance interconnection technologies on Java sockets, a High Performance Java socket implementation, called *Java Fast Sockets (JFS)*, has been developed. `JFS` aims to be efficient and portable by providing two alternative solutions using pure Java and JNI wrappers to low-level SAN protocols. In the presence of these SAN protocols, `JFS` uses the JNI approach. Otherwise, it uses the pure Java solution. Moreover, the use of the new Java NIO capabilities, such as new data containers (*direct* `ByteBuffer`), new I/O channels, selectors and selection keys, can optimize performance in `JFS`. Finally, by setting the default `SocketImplFactory` to a factory that returns `JFS` sockets, every socket operation in an application can transparently use `JFS`.

4.2 Native Java Communication Support

In the design of native support of `SCINBComm` and in `JFS`, both libraries use communication mechanisms implemented by the underlying libraries. High Performance Clusters usually provide several protocols depending on the message size, as communication performance depends on the trade-off between latency and protocol processing overhead. Thus, one-copy protocol trades off high CPU load for low latency, whereas zero-copy protocol cuts down system load (high bandwidth rates with low CPU loads). A sensible choice between protocols involves using one-copy protocol for latency sensitive applications, and zero-copy protocol for applications with high bandwidth requirements. On SCI, native libraries resort to `SCILib`, implementing the non-blocking semantic on top of this blocking layer by means of threads. The protocol choice can be configured by the user.

5 Performance Evaluation

In this section an evaluation of `NBComm` implementations is presented. Additionally, mpiJava non-blocking communication over MPICH on GbE has also been tested for comparison purposes. SCI-MPICH [15] is not supported by mpiJava in our testbed. In order to evaluate the performance, half of the round trip time of a ping-pong test (hereafter called latency) is measured. Moreover, two specific non-blocking communication benchmarks including a communication/computation overlapping test and an overlapping communications test are used.

5.1 Experiment Configuration

Our testbed consist 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 an 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 SCI SOCKETS/DIS 3.0.3. mpiJava version 1.2.5 runs on top of MPICH 1.2.5.

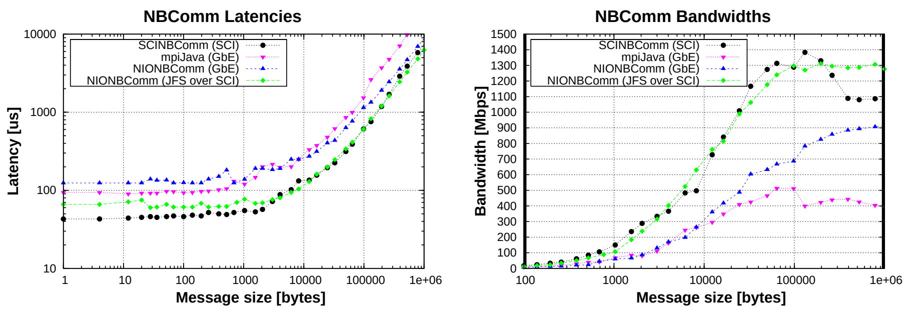

Fig. 3. Measured latencies and bandwidths of `NBComm` implementations

5.2 Performance Results

Figure 3 shows experimentally measured latencies and bandwidths of `NBComm` implementations on SCI and GbE as a function of the message length. The bandwidth graph (right side) is useful to compare long-message performance, whereas latency graph (left side) serves to compare short-message performance (note that their scale is logarithmic). In order to analyse the overhead imposed by `NBComm`, experimental results from `SCILib` (library used by `SCINBComm`), `JFS`, and Java sockets (libraries used by `NIONBComm`) are shown in Figure 4.

The two lower graphs of Figure 4 show the latency and bandwith of `NIONBComm` using GbE. In addition, the graphs also show the latency and bandwith of the raw Java sockets. The difference between the performance of `NIONBComm` and Java sockets shows the imposed overhead. This overhead is aproximately $60\mu s$ in terms of latency. As can be seen from the two upper graphs in Figure 4, `SCINBComm` obtains lower startup time than `NIONBComm` over `JFS` on SCI. The overheads in latency imposed by the `NBComm` layer are around $40\mu s$ and $58\mu s$ over `SCILib` and `JFS` respectively. Bandwidth performance is quite similar except for messages larger than 256KB where the pure Java implementation outperforms the native implementation. As expected, `SCINBComm` obtains better results in general than `NIONBComm` using `JFS` on SCI. However, the performance gain is due to the use of JNI. `JFS` has an asymptotic bandwidth similar to native sockets and startup times as low as $8\mu s$. Some experimentally measured examples of latency reduction have been observed: a 64Kb message in the SCI testbed where the reception is posted after receiving the message has $t_{isend} = 156\mu s$, $t_{send} = 308\mu s$, $t_{irecv} = 3\mu s$, $t_{recv} = 308\mu s$ and $t_{iwait} = 2\mu s$. The sender process obtains a time gain of $152\mu s$ (49%), apart from not having to wait to send, and the receiver process obtains a time gain of $303\mu s$.

The CPU overlap test determines the amount of software overhead involved in sending and receiving messages. The benchmark code consists of inserting gradually increasing computation between the calls that initiate and complete a non-blocking send or receive operation. By determining the maximum amount of computation that can be overlaped with communication the computation/-

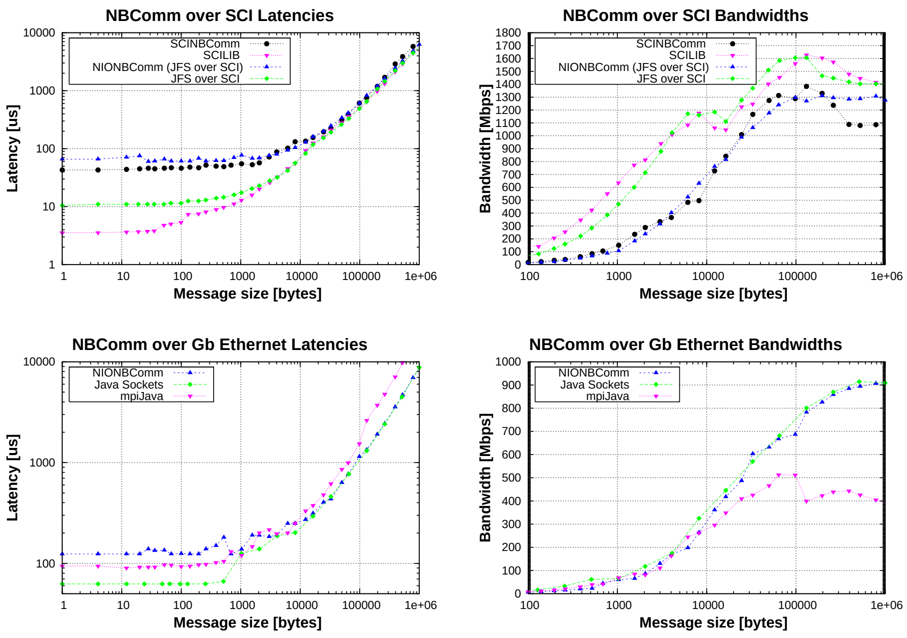

Fig. 4. Measured latencies and bandwidths of `NBComm` vs. underlying libraries

communication overlapping parameter can be obtained. This assumes that the computation cost does not affect the measured communication time. The results obtained by benchmarking 1Kb messages show that a 37% of the communication time can be overlapped with computation in `SCINBComm`. A 6% and a 40% performance improvement is obtained for `NIONBComm` and mpiJava respectively. Native-based solutions provides a higher degree of computation/communication overlapping.

The overlapping communications test benchmarks the overlap of communication with additional communication. Rather than filling idle CPU time with computation, as in the previous test, it can be used to send additional messages. It has been experimentally observed that sending 8 simultaneous 1Kb messages helps achieve a latency reduction of 44% in `SCINBComm`, a 33% in `NIONBComm`, and a 49% in mpiJava.

6 Conclusions

Communication performance is critical for the overall system cluster performance. In this scenario non-blocking communications can significantly reduce the communication overhead. Nevertheless, the definition of an efficient non-blocking Java communication library with cluster support poses an important number of implementation issues. These can be summarized in designing the solution for receiving messages, notify the arrival of messages, the study of the efficiency of data movements and the API definition. This Java communication library can use Java sockets implementations or native communication libraries specialized for SAN systems. This paper has presented a non-blocking Java communication library (`NBComm`) that resolves efficiently numerous design issues aforementioned and provides cluster support. This library aims at reducing the startup time of communications, avoiding unnecessary copying and overlapping computation and communication. In the design of the library a thread is devoted to receive messages (`receptor_thread`). The approach followed also ensures that unnecessary copying is avoided writing directly to a buffer of type direct `ByteBuffer` and DMA is used for messages longer than 8KB. This library implements different solutions depending on the underlying communication libraries—`SCINBComm` is implemented for using SCI native communication libraries and `NIONBComm` for using Java sockets. A High Performance Java socket implementation `JFS` can also be used as communication layer for `NIONBComm`, providing additionally access to SCI for this solution.

The use of non-blocking communication can gain significant improvements with respect to the use of blocking communication in parallel applications. It has been experimentally assessed that non-blocking communication is specially advantageous, obtaining latency reductions and overlapping computation with communication, yielding communication overhead reductions up to 50%.

Acknowledgments

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

References

1. 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.
2. 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 International Conference on Cluster Computing (CLUSTER'03)*, pages 118–126, Hong Kong, China, 2003.
3. M. Welsh and D. E. Culler. Jaguar: Enabling Efficient Communication and I/O in Java. *Concurrency: Practice and Experience*, 12(7):519–538, 2000.
4. S. B. Lim, B. Carpenter, B. Fox, and H.-K. Lee. A Low-Level Communication Library for Java HPC. In *Proc. 6th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP'05), LNCS 3719, Springer-Verlag*, pages 429–434, Melbourne, Australia, 2005.
5. H.-K. Lee, B. Carpenter, G. Fox, and S. B. Lim. HPJava: Programming Support for High-Performance Grid-Enabled Applications. *International Journal of Parallel Algorithms and Applications*, 19(2–3):175–193, 2004.

6. M. Baker, B. Carpenter, and A. Shafi. MPJ Express: Towards Thread Safe Java HPC. In *Proc. 8th IEEE International Conference on Cluster Computing (CLUSTER'06)*, Barcelona, Spain, 2006.
7. M. Baker, B. Carpenter, and A. Shafi. An Approach to Buffer Management in Java HPC Messaging. In *Proc. 6th International Conference on Computational Science (ICCS'06), LNCS 3992, Springer-Verlag*, pages 953–960, Reading, UK, 2006.
8. M. Baker, B. Carpenter, G. Fox, S. Ko, and S. Lim. mpiJava: an Object-Oriented Java Interface to MPI. In *Proc. 1st International Workshop on Java for Parallel and Distributed Computing (IPPS/SPDP'99), LNCS 1586, Springer-Verlag*, pages 748–762, San Juan, Puerto Rico, 1999.
9. M. Bornemann, R. V. van Nieuwpoort, and T. Kielmann. MPJ/Ibis: A Flexible and Efficient Message Passing Platform for Java. In *Proc. 12th European PVM/MPI Users' Group Meeting, (PVM/MPI'05), LNCS 3666, Springer-Verlag*, pages 217–224, Sorrento, Italy, 2005.
10. F. Seifert and H Kohmann. SCI SOCKETS - A Fast Socket Implementation over SCI. http://www.dolphinics.com/pdf/whitepapers/sci-socket.pdf. [Last visited: July 2006].
11. S. H. Rodrigues, T. E. Anderson, and D. E. Culler. High-Performance Local-Area Communication With Fast Sockets. In *Proc. Winter 1997 USENIX Symposium*, pages 257–274, Anaheim, CA, 1997.
12. J.-S. Kim, K. Kim, and S.-I. Jung. SOVIA: A User-level Sockets Layer Over Virtual Interface Architecture. In *Proc. 3rd IEEE International Conference on Cluster Computing (CLUSTER'01)*, pages 399–408, New Port Beach, CA, 2001.
13. P. Balaji, P. Shivan, P. Wyckoff, and D. K. Panda. High Performance User Level Sockets over Gigabit Ethernet. In *Proc. 4th IEEE International Conference on Cluster Computing (CLUSTER'02)*, pages 179–186, Chicago, IL, 2002.
14. S. Petri, L. Schneidenbach, and B. Schnor. Architecture and Implementation of a Socket Interface on top of GAMMA. In *Proc. 28th IEEE Conference on Local Computer Networks (LCN'03)*, pages 528–536, Bonn, Germany, 2003.
15. J. Worringen and T. Bemmerl. MPICH for SCI-connected Clusters. In *SCI Europe'99*, pages 3–11, Toulouse, France, 1999.