Java Objects Communication on a High Performance Network

Luc Courtrai, Yves Mahéo, Frédéric Raimbault
Orcade Project
VALORIA Laboratory
South Brittany University
Tohannic, rue Yves Mainguy – 56000 Vannes – France
{Luc.Courtrai,Yves.Maheo, Frederic.Raimbault}@univ-ubs.fr

Abstract

Local high performance networks availability already makes workstations clusters a serious alternative for parallel computing. However, a high level and effective programming language for such architecture is still missing. Recent works show the interest in Java for cluster programming. One of the main issues is to handle efficiently the communication of objects to really take advantage of the network speed. The paper presents an alternative to the standard serialization process through the proposal of a Java object communication library. Object allocation is controlled in such a way that the transfer of objects between two nodes comes to a direct memory to memory dump. We show how specific allocation mechanisms can cooperate with a Java Virtual Machine so that fast transfers of graphs of objects can be achieved. Experimental results are given for basic operations and for a genetic programming application; they demonstrate a dramatic change in the transfer speed.

1. Introduction

Local high performance networks availability already makes workstations clusters a serious alternative for parallel computing infrastructure. However, an effective high level programming language for such architecture is still missing. Recent works show a growing interest in Java for cluster programming [5].

One of the main issues in Java for high performance distributed applications remains the transfer speed of objects. On one side, the object paradigm implies that an object may have references to others objects, so that a complete graph of objects must be communicated. On the other side, communication libraries such as Unix sockets or PVM/ MPI API offer the transmission of a simple memory buffer. So the gap must be filled between the needed ability to send a whole graph of objects and the basic message passing functionality.

The most common way to handle object communication in Java is to apply a serialization on the sending side and an unserialization on the receiving side: the graph of references is explored and data representing each node of the graph is copied into a memory buffer that can be sent by the underlying communication primitive. A deep copy of the graph can be rebuild from the received buffer as far as enough type information is encapsulated in the message. This method can be applied in Java by using the standard interfaces [1] that allow an object to be serialized into a stream (so possibly an array of bytes). Marshalling relies on introspection mechanisms to automatically define the bytes array layout. Unfortunately, this shows very poor performance though the programmer may provide methods to specialize operations.

Firstly, serialization in Java has been designed in a very general way in terms of portability and heterogeneity. Much type information must be computed and put in the buffer. Besides, many method calls and object creations are involved when marshalling and unmarshalling the graph of objects. This makes the serialization a very time consuming operation. And serialization is especially disadvantageous in the context of high speed networks as demonstrated by the following experiment.

We have timed separately the marshalling process and the pure communication on a standard (10 Mb/s Ethernet) network and on a high-speed (1 Gb/s Myrinet) network[2]. Results for the transfer of a graph of objects (a binary tree of 1000 integers) are reported on table 1. Three available Java API have been tested. When using the most general one – the standard `JDK Serializable` API – the speedup

[1] The standard JDK API provides the `Serializable` and `Externalizable` interfaces in conjunction with the RMI mechanism for communication purpose.

[2] Technical characteristics of the experimental platform is given in section 3

API		Serializable	Externalizable	XSerializable
serialization (& unserialization) time in μs		$109,155$	$11,741$	$3,742$
communication time in μs	Ethernet	$39,400$	$19,420$	$19,420$
	Myrinet	797	598	593
total transfer time in μs	Ethernet	$148,455$	$31,161$	$23,162$
	Myrinet	$109,952$	$12,339$	$4,335$
speedup Ethernet → Myrinet		1.35	2.52	5.34
$\frac{\text{serialization time}}{\text{total transfert time}}$ in $\%$	Ethernet	73	37	16
	Myrinet	99	95	86

Table 1. Marshalling process costs on standard and high speed networks

obtained on the high speed network is only 1.35 while the communication speed is increased by 50! Explanation lies on the last row where it appears that the serialization and the unserialization process represent $73\,\%$ of the total transfer time. So the increase of the communication speed benefits only to a quarter of the transfer time. A common way of improvement is the use the `Externalizable` API. The marshalling method is provided by the programmer instead of relying on an introspection mechanism; it reduces the graph traversal time and the size of the message representing the object transferred. But the speedup obtained in this case grows only to 2.5 when migrating from the Ethernet to the Myrinet network while the communication speed is increased by 32. Things improve slightly because the part of the serialization process in the total transfer time remains high ($37\,\%$). Finally, we tested an optimized version of the `Externalizable` API (noted `XSerializable`) provided by the JavaParty team at Karlsruhe University [10]. It improves considerably the marshalling process and allows the speedup to reach 5.34. But, despite the various optimizations included in the `XSerializable` implementation, the programmer does not benefit much from the communication speed improvements: about 86% of the transfer time on the high speed network is spent in communication packaging and unpackaging.

We claim that the serialization mechanism should be reconsidered when one wants to take full advantage of the network technology improvement. In some cases where heterogeneity and portability are not the prior concerns, fast object communication could be achieved by avoiding serialization. In this article, we present an experimental library called *Expresso* that provides efficient communications of Java objects.

Our transfer scheme does not need any memory copy nor graph traversal; objects are managed in clusters in order to allow the straight communication of the whole graph of objects. Section 2 describes two kinds of object clustering: the ISO–address scheme which leads to best performance and the relocatable scheme which offers more flexibility. Of course these models imply that the data layout of the objects is accessible and so prevent us from developing a pure Java solution. We detail our implementation in section 3 and report some performances results. Current status and perspectives conclude this article.

Related Works

In the context of high performance networks, many researches on Java and message passing systems have been conducted in the last few years [5, 6]. Most of them want to stick to the Java standards by providing an efficient RMI (Remote Method Invocation) mechanism and consequently an efficient object serialization. For example, in [10], a pure Java solution is adopted that reduces the quantity of information that must be put in the buffer and optimizes buffer management. In [9], a native serializer, aware of objects layout, is automatically generated. Other works try to provide MPI facilities in the context of Java. If some early studies did not fully allow for graphs of objects, focus is now put on serialization by trying to extend MPI data-types with the `OBJECT` type [8, 3]. Several distributed shared objects services have been proposed on top of Java libraries, e.g. in [7, 11]. Performance issues such as clustering of objects we use in our implementation have been pointed out. However, to our knowledge, there is no proposal that questions the principle of serialization when communicating graphs of Java objects.

2. Expresso

Expresso is a Java library that aims at providing objects exchange capabilities to an object oriented language as MPI or PVM provides values exchange capabilities to procedural languages through the message passing paradigm. *Expresso* does not introduce a new object distributed model; it is inherited from the MPI library upon which *Expresso* is built: a N-node network is seen as a set of processes numbered from 0 to $N-1$ communicating on a fully connected

network through send and receive operations. So only one thread of each Java virtual machine (*JVM*) communicate with the others. Emphasis is on performance rather than on expressiveness: as mentioned in the introduction, very fast object transfer between two machines is achieved thanks to a direct memory to memory dump in replacement of the usual marshalling of structured values.

2.1. Object clustering

Simple and efficient communication is obtained through the clustering of objects. The allocation of the objects is controlled in order to maintain the entire representation of a graph of objects in a single memory region that can be easily transferred: the cluster. A set of clusters is managed in a specific memory area on each node. Virtual memory (not physical memory) is reserved for this cluster space. Enough space can be considered available given the size of the addressing space of a process (4 gigabytes on current 32-bit architectures).

A cluster is the unit of communication on the network and it may contain many objects. Firstly it allows communication granularity management. Without clusters, one would have to explore the complete graph of objects to know which memory words have to be transferred (this is the serialization process); or conversely it would be necessary to transfer the whole cluster space of the emitting node (transferring one big chunk of several megabytes is too expensive, even on a high speed network). So the cluster is an intermediate whose size is somehow tunable. Moreover, it makes sense to group the representations of certain objects as this grouping reflects the logical grouping that frequently already exists in applications, even if it is often implicit.

Only objects that may be exchanged are allocated in clusters; the others are left in the heap managed by the *JVM*. Clusters are exchanged between nodes with asynchronous send and receive operations. As a cluster may contain several linked objects, one object – called the root object – acts as a handle to access the others indirectly. Some objects stored in a cluster may be out of the graph that is to be communicated, thus incurring extra data transfer. However, in the context of high-bandwidth networks, this should not be too expensive.

It should be noticed that our objective is not to propose a new programming model; as a consequence, coherency of inter-cluster references is not treated. We focus here on the transfer of a graph of objects contained in a single cluster.

Actually, *Expresso* provides two types of clusters that are stored in separated areas and may be used simultaneously: ISO–address and relocatable clusters. ISO–address clusters lead to better performances whereas relocatable clusters offer a more classical communication model. Both are described in the following.

2.2. ISO–address clusters

Expresso runs on a set of workstations connected through a high speed network. Computing resources are supposed to be homogeneous, from the hardware and the operating system and the *JVM* point of view. The principle of the ISO–address transfer is to put a transferable object on the receiver at the same (virtual) memory address as on the sender. In this way, references inside an object remain valid and do not need to be updated. Therefore an object transfer becomes a simple memory block transfer analogous to a DMA operation. The same kind of idea has been applied to thread migration in [1].

The key point is to manage the memory between the nodes to ensure that an object can be copied at the same address on another node without a crash. Any transferable object created on one a node has to be allocated in an reserved range of addresses for this node, within the ISO–address cluster space. Figure 1 contains the ISO–address cluster space layout for a two-node network. In this case, the ISO–address space is divided into two parts: the higher one is devoted to the objects allocated on the first node; the lower one is devoted to the objects allocated on the second one.

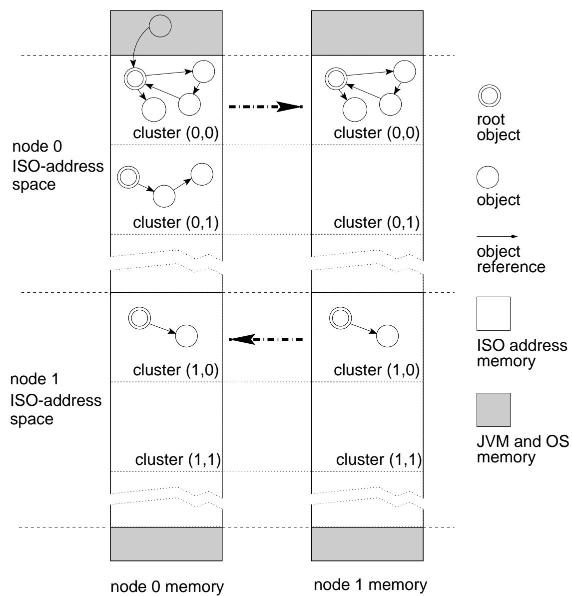

Figure 1. ISO–address memory for a two-node network

The size of clusters are identical and fixed at initialization time. Thus the exact memory location of a cluster can be deduced from its identity and duplicated at the same address on another node without any calculation nor any cluster overlapping problem. Two numbers identify a cluster:

its origin node – the node where it has been created – and its rank on this node.

Communications. ISO–cluster transfer is intended to be very efficient. On one side, the send operation transmits the entire content of a cluster to the destination node. On the receiving side, the cluster content is replaced by the incoming one. Therefore only one copy of a distant cluster is available at a time. Having received a cluster, a reference on the cluster root object can finally be obtained. Note that a receiving node is free to specify the node from which the cluster is awaited: it may or not be the origin node of the cluster; it may also be any node when the sender is not specified at all.

This simple protocol doesn't take care of coherency between multiple copies of clusters. This is left under the programmer's responsibility or to an upper layer library.

2.3. Relocatable clusters

Relocatable clusters can be used in addition or as an alternative to ISO–address clusters. In some cases, they offer more flexibility at the cost of a slight computing overhead.

Relocatable clusters are dynamically allocated within a specific memory space The size of the clusters is specified at creation time and may be different for each cluster. Freeing relocatable clusters is done explicitly.

The use of relocatable clusters is identical to the use of ISO–address clusters. Once created, a cluster can be set as the current cluster where new objects will be stored. A relocatable cluster can be transfered to another node by using the send and receive operations. No numbering of clusters is necessary: a copy of a cluster is sent and a new cluster is created upon receipt. In this respect, relocatable clusters offer the usual message passing facilities. There is no coherency problem due to cluster overwriting as with ISO–address clusters; still, inter-cluster references are not managed either.

Figure 2 depicts the layouts of the relocatable spaces in a two-node network after three cluster transfers. It can be seen that two copies of the second cluster on node 1 are received by node 0 after a possible modification. As mentioned earlier, this was not possible with ISO–address clusters.

2.4. Expresso API

ISO–address and relocatable clusters are implemented through a Java package. The main class of the package is the `Cluster` class. Two sub-classes (`ClusterISO` and `ClusterRELOC`) provide the implementation of the ISO–address clusters and the relocatable clusters.

We present in figure 3 a brief example of a Java program using *Expresso* classes.

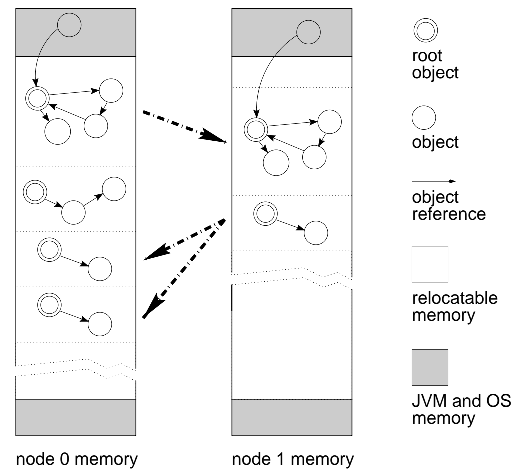

Figure 2. Relocatable memory for a two-node network

This program runs on a two-node network. A simple graph of objects (a car object referring to an engine object) is created on one side and sent to the other side where it is printed. As the same program runs on both nodes – this is the SPMD parallel programming model – a conditional tests the running node number to select the code to execute. The first node creates a cluster, allocates a car object inside it, fills it and sends it to the second node. The second node receives the cluster, extracts the car and prints it.

The class method `Cluster.newObject` must be used to create transferable objects in the current cluster. An object allocated this way is stored in the current cluster with the same structure as a Java object; the *JVM* can operate on it as on an ordinary object. All the others objects are allocated through the Java `new` operator.

A cluster can be sent to a node with the instance method `send` and received with the instance method `recv`. The destination of the message is a node number. When receiving a cluster, parameters are slightly different according to the type of clusters used. For ISO–address clusters, one should specify the cluster identity (*i.e.* the origin node and the rank) and optionally the number of the node from which the message is received if it is different from the origin node. For relocatable clusters, only the emitting node is required. In both cases, if the source of the message does not matter, `ANY` can be passed to the `recv` method in replacement of a node number. The `recv` method returns a handle to a newly created cluster – of the proper type – filled with the objects that were in the message.

```
import expresso.*;

public class TestCar {

  public static void main(String argv[]) {

    Cluster.init(argv); // initialize Expresso
    if (Cluster.myNode == 0) { // first node code
      // create a ISO cluster
      ClusterISO aCluster = new ClusterISO();
      // indicate which cluster will be filled
      // with future objects
      aCluster.setCurrent();
      // allocate and initialize a Car
      // inside the current cluster
      Engine anEngine =
          (Engine)Cluster.newObject(Engine.class);
      anEngine.init(50); // power
      Car aCar =(Car)Cluster.newObject(Car.class);
      aCar.init(2000,anEngine); // year and engine
      // set the cluster root object
      aCluster.setRootObject(aCar);
      // send the car to the second node
      aCluster.send(1);
    } else { // second node code
      // receive the 1st cluster of the 1st node
      ClusterISO aCluster = ClusterISO.recv(0,0);
      // extract the root object
      Car aCar = (Car)aCluster.getRootObject();
      // print it contents
      System.out.println(aCar);
    }
    Cluster.quit();// stop using Expresso
  }
}
```

Figure 3. Transfer of a simple graph of objects between two nodes with Expresso

3. Implementation

The main part of *Expresso* is written in *C* and is called by the interface through the *JNI* (Java Native Interface) mechanism. It is composed of a memory management module and a communication module. The implementation details of these modules are explained in the following subsections.

3.1. Memory management

Ordinary objects are normally allocated by the *JVM* in its heap located in the data segment of the Unix process. For *Expresso* needs, a memory space distinct from the *JVM* heap is required to host the ISO–address space and the relocatable space. The memory management module of *Expresso* asks the operating system a large zone within the part of the 2^{32} bytes of memory available to the process that is left free between the stack and the heap segment. Thus, at initialization time, a `brk` system call allocates two spaces. The first one is the ISO–address space; it is split up into as many chunks as the number of nodes. An extra chunk is also needed for the dispatch tables copies (see below). The second one is dedicated to the relocatable clusters. This last one is managed dynamically as a segmented memory.

The `newObject` method reserves a placeholder for an object in the current cluster. In the case of an ISO–address cluster, the allocation takes place within the pre-determined local part of the ISO–address space dedicated to the executing node. When the current cluster is a relocatable cluster, the object is added to the chained list of objects managed by the cluster. In both cases, the structure of the created object conforms to the running *JVM* machine implementation; so the *JVM* can apply any operation on it as on an object allocated by the `new` primitive. The current version of *Expresso* is bound to the *Kaffe JVM* implementation (version 1.0.5)[3] : figure 4 shows the layout of a *Kaffe* object. A header contains garbage collector and lock information, and a link to the object's dispatch table (for direct access to the object's methods). The object's fields reside in the words following the header.

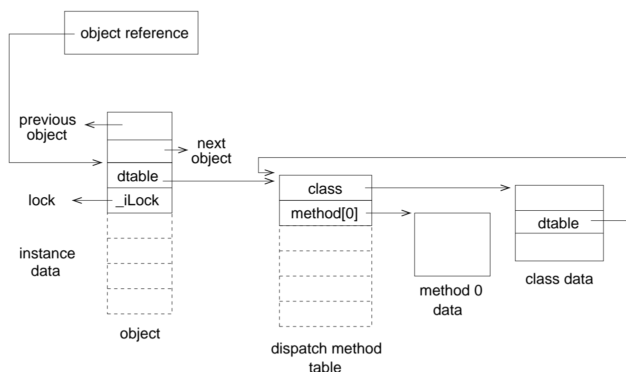

Figure 4. Object structure in the Kaffe virtual machine

The memory management module of *Expresso* takes all the information needed to size the object structure and to fill the dispatch table reference from the class parameter of `newObject`. Figure 5 exemplifies the links between the ISO–address memory space, the relocatable space and the *JVM* objects heap; it represents a situation where several objects allocated via *Expresso* and via the *JVM* share the same class data.

The classes of the program are loaded by a custom class loader during *Expresso* initialization. Class data are allocated normally within the JVM heap but copies of the dispatch tables are placed in a dedicated zone within the

[3] *Kaffe* is available from `http://www.kaffe.org`

ISO–address space. Every node load all the classes; consequently, the dispatch tables are allocated at the same address on all the nodes. So the `dtable` link between an object and its dispatch table is preserved during transfer and does not need any updating.

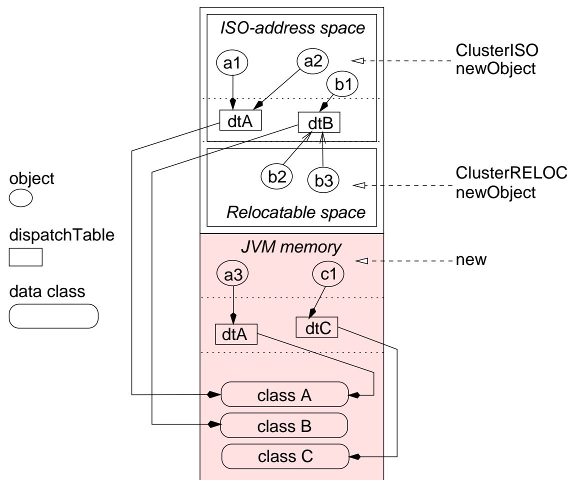

Figure 5. Links between memory spaces

3.2. Communication management

The communication management module of *Expresso* runs over a *Myrinet* high speed network [2], from the *Myricom* company. Very low level communications are managed by the proprietary library GM (version 1.01). A MPI implementation upon GM (*gm-mpich*) is also provided by *Myricom*. This communication layer offers all the message passing facilities needed to implement object transfer in *Expresso*.

Expresso methods `recv` and `send` are Java wrappers which call the *C* functions `expresso_recv()` and `expresso_send()` through the *JNI* mechanism. We have written these *C* functions using the underlying *gm-mpich* library.

The `send` method call on one node and its `recv` method counterpart on another node, act as a memory block dump between the corresponding machines.

When receiving an ISO–address Cluster, the message content is written at the same virtual address which is computed from the cluster number and the origin node number (see paragraph 2.2). No extra calculation is required.

A few operations are necessary when receiving a relocatable cluster: a new cluster is allocated according to the size of the incoming message before the actual receipt. After receipt, the references contained in the objects of the cluster are relocated: the list of objects is traversed so that the reference fields of each of them are modified by a constant value (the difference between the address of the cluster on the emitting node and its local address) if they are not null. In order to avoid class introspection, an offset table for references fields is built when the class is loaded. This table is accessed through the dispatch table link of the object.

Although the set of objects received must be examined, relocatable clusters management offers important advantages over the classical serialization-unserialization process: no treatment is done on the sending side and on the receiving side, cycles in the graph are not considered, no object allocation is needed and no class references are manipulated.

3.3. Performances

We have experimented the *Expresso* library on a cluster of Linux PCs (Pentium II 400Mhz) linked by a 1 Gb/s Myrinet network (processor Lanai 4, PCI 33 Mhz interface adapter). In the following, we show performance figures of basic *Expresso* primitives and of an application program.

Object creation and access. The `newObject` method is slightly faster than the Java `new` primitive as we do not have to deal with the garbage collector. The difference between the use of ISO–address clusters and relocatable clusters is negligible. Of course, the time for accessing an object created by `newObject` is strictly identical to the time for accessing an object created by the `new` primitive since the JVM is not aware of the difference.

Cluster communication. We have compared the global cluster transfer time using *Expresso* with a classical method that builds a buffer with the standard Java interfaces `Serializable` and `Externalizable` before sending it with MPI (in Java). The graph of objects transferred is a random graph representing a map (towns, crossroads and roads linked to each other). Each object has four references. The average number of non-null references per object is about 2.5. Table 2 shows times in μs obtained with the three versions. It is clear that avoiding serialization leads to a dramatic improvement. Because we use a high bandwidth network, global performances are not really affected by the fact that the message is larger (twice the size of the message generated by the serializable version) in the case of *Expresso*. Indeed, the message contains the complete memory representation of the object and not only its attributes. Besides, the overhead induced by relocatable clusters, compared to ISO–address clusters stays relatively low, and does not change the comparison between cluster-based transfer and serialization-based transfer.

Application Even if ISO–address clusters are not primarily intended to be used at the application level, we have

Number of objects	*Serializable*	*Externalizable*	*ClusterRELOC*	*ClusterISO*
29 (10 towns)	25,466	3,006	307	277
355 (100 towns)	145,525	24,969	1,075	808
1760 (500 towns)	1,383,576	210,249	4,302	2,825

Table 2. Performances of Java API and Expresso

experimented their use in a parallel genetic programming application. A simple parallelization method has been applied to the sequential Java code (a block data distribution is performed at each step of the algorithm with a scatter-gather phase) [4]. The objective was to compare the impact of the efficiency of objects communication in the overall parallel performance, not to obtain the peak performance by designing a parallel algorithm (namely load balancing issues are not really taken into account). The speedups obtained are presented in figure 6 for a 120-individual population. The efficiency of the code using serialization falls down rapidly when the number of nodes increases whereas it remains very correct with *Expresso*.

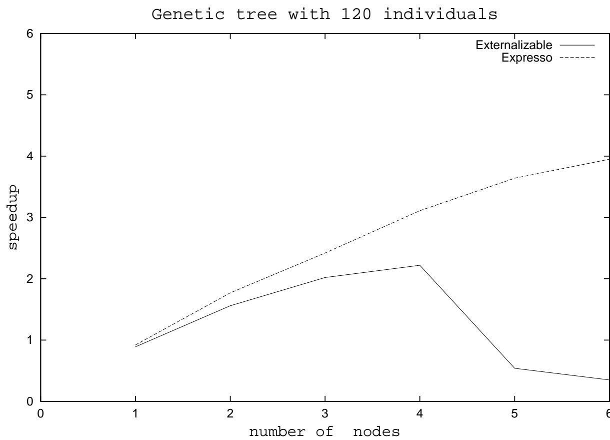

Figure 6. Speedup for a genetic programming application

4. Conclusion

We have presented in this paper *Expresso*, a library dedicated to the transfer of complex Java objects. The allocation of objects is controlled so that objects belonging to the same graph are allocated in a cluster that consists in contiguous memory regions. Two types of cluster have been implemented, ISO–address clusters and relocatable clusters so as to fulfill a wider range of application needs. These clusters can be efficiently communicated because no serialization is required. This enables the full exploitation of the capabilities of high speed networks.

Indeed, experimental results show that the transfer times for graphs of objects involving several types and containing cycles are very close to the times needed for a simple data transfer. These performance cannot be obtained with classical serialization whose relative costs can be considered prohibitive when using networks with low latencies and high bandwidth.

However, the communication library we proposed is at a very low level: the user has to handle the clustering of objects. It must be proposed a higher level programming paradigm that can take benefit of *Expresso* cluster exchanges. In this framework, we investigate how the RMI protocol can be built on top of *Expresso*. Another research direction is to integrate this kind of objects transfer in the JVM in order to develop object sharing between distant JVMs.

References

[1] G. Antoniu, L. Bougé, and R. Namyst. And efficient and transparent thread migration scheme in the PM2 runtime system. In *Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP'99)*, San Juan, Puerto Rico, April 1999.

[2] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W.-K. Su. Myrinet – A Gigabit-per-second Local-Area Network. *IEEE Micro*, February 1995. http://www.myri.com.

[3] B. Carpenter, G. Fox, S. Ko, and S. Lim. Object Serialization for Marshalling Data in a Java Interface to MPI. In *Proc. of the Java Grande Conference*, San Francisco, California, June 1999.

[4] L. Courtrai, Y. Mahéo, and F. Raimbault. Expresso : transfert d'objets Java sur réseau haut-débit. Internal report, Valoria, Université de Bretagne Sud, march 2000. http://www.univ-ubs.fr/valoria/rr.

[5] J. G. Forum. Java Grande Forum Report: Making Java work for high-end computing. Technical Report TR-01, Java Grande Forum, November 1998. http://www.javagrande.org.

[6] G. Fox. Editorial: Java for high performance network computing. *Concurrency: Practice & Experience*, 10(11-13), September-November 1998.

[7] D. Hagimont and D. Louvegnies. Javanaise: Distributed Shared Objetcs for Internet Cooperative Applications. In *Middleware '98*, The Lake District, England, September 1998.

[8] G. Judd, M. Clement, Q. Snell, and V. Getov. Design Issues for Efficient Implementation of MPI in Java. In *Proc.*

of the Java Grande Conference, San Francisco, California, June 1999.

[9] J. Maassen, R. van Nieuwpoort, R. Veldema, H. Bal, and A. Plaat. An Efficient Implementation of Java's Remote Method Invocation. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, pages 173–182, Atlanta, Georgia, May 1999.

[10] M. Philippsen and B. Haumacher. More efficient object serialization. In *Proc. International Workshop on Java for Parallel and Distributed Computing*, San Juan, Puerto Rico, April 1999.

[11] M. Philippsen and M. Zenger. Javaparty - transparent remote objects in java. *Concurrency: Practice & Experience*, 9(11):1225–1242, november 1997.