Expresso: a library for fast transfers of Java objects

Luc Courtrai, Yves Mahéo, Frédéric Raimbault

Expresso: a library for fast transfers of Java objects

Luc Courtrai Yves Mahéo Frédéric Raimbault
Orcade Project
VALORIA Laboratory
Université de Bretagne Sud
Tohannic, rue Yves Mainguy – 56000 Vannes – France
{Luc.Courtrai | Yves.Maheo | Frederic.Raimbault}@univ-ubs.fr

Abstract

Expresso is a Java library dedicated to cluster-based computing on a high speed network. It 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; the emphasis is on performance rather than on expressiveness: 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. We develop the ISO-address concept to manage the memory on the network nodes and implement it upon available communication libraries. Experimental results demonstrate a dramatic change in the objects transfer speed. Application programmers or higher level libraries are able to capitalize on such communication mechanism and to exploit the potential of high speed network even through an object language.

1 Introduction

Workstation clusters are emerging as the new parallel computing architectures. This is possible thanks to the relative low cost of local area high performance networks and the use of standard hardware for computing nodes. On the software side, the message passing paradigm has proven to be effective. It is widely adopted in the parallel an distributed programming community for its efficiency and the programmer is now provided with communication libraries that fully exploit the capabilities of high speed networks. However, widely used portable libraries such as MPI or PVM have been designed for procedural languages. *C* or Fortran programming is not sufficient for a range of novel high performance applications and object oriented languages offer a lot more comfortable environment for this kind of development. In particular, recent works show the interest in Java for cluster programming and the joint use of Java and message passing is the subject of numerous researches [6].

One of the main issues is to handle efficiently the communication of objects. Since an object may have references to others objects, a graph of objects that possibly contains cycles must be communicated. However, the gap must be filled between the needed ability to send a whole graph of objects and the basic message passing functionality that consists in the transmission of a simple memory buffer.

The most common way to handle object communication 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 simply applied in Java by using the standard interfaces 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. For instance, the class of the object put in the message may only have a compatible equivalent on the receiving end. This increases the size of type information inside 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. This is especially disadvantageous in the context of high speed networks. Measurements show that the time required to build the buffer can be more than an order of magnitude longer than the time of pure communication [4].

When designing communication protocols in low-level message passing libraries for high speed networks, efforts have been done in order to reduce computing overhead, namely through zero-copy strategies. We believe that these efforts must be extended to object communication in upper layer libraries. In some cases where heterogeneity and portability are not the prior concerns, fast object communication should be achieved by applying this zero-copy strategy. Of course, this implies that the data layout of the objects is accessible and so prevents from developing a pure Java solution.

In this article, we present an experimental library called *Expresso* that provides efficient communications of objects that do not rely on any serialization. The transfer scheme does not need any memory copy nor graph traversal. The allocation of objects is controlled in order to allow an ISO–address communication of the whole graph of objects: a simple memory transfer through the network is sufficient.

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 [6, 5]. Some of them want to stick to the Java standards by providing an efficient Remote Method Invocation mechanism and consequently an efficient object serialization. In [9], a pure Java solution is adopted that reduces the quantity of information that must be put in the buffer and optimizes buffer management. In [8], a native serializer, aware of objects layout, is automatically generated. Another objective is to provide MPI facilities in the context of Java. If some early work did not fully allow for graphs of objects, focus is again put on serialization by trying to extend MPI datatypes with the OBJECT type [7, 3]. 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 dedicated to cluster-based computing. It 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; the emphasis is on performance rather than on expressiveness: as explained 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.

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 Java virtual machine (*JVM*) point of view. More technical details are given in the next section (section 3). In this section we focus on the ISO–address principle and its use for the transfer of objects. Then the *Expresso* API is presented and exemplified.

2.1 ISO-address transfer

The principle of the ISO–address transfer is to put the transferable objects at the same (virtual) memory address on the receiver 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].

ISO-address memory

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. Each node has such a separate memory so the whole set of memory parts forms what we call an ISO–address memory space. Figure 1 contains the ISO–address memory 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.

It should be noticed that the ISO–address mechanism uses virtual memory reservation, not physical memory reservation. Besides, the size of the ISO–address space can be considered sufficient, given the size of the addressing space of a process (4 gigabytes on current 32 bits architectures).

Memory clusters

The ISO–address space is divided into consecutive chunks of memory called *ISO–clusters*. An *ISO–cluster* is the unit of communication on the network. It may contain many objects. The reason for the clustering is twofold:

1. 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 ISO–address of the sending 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.

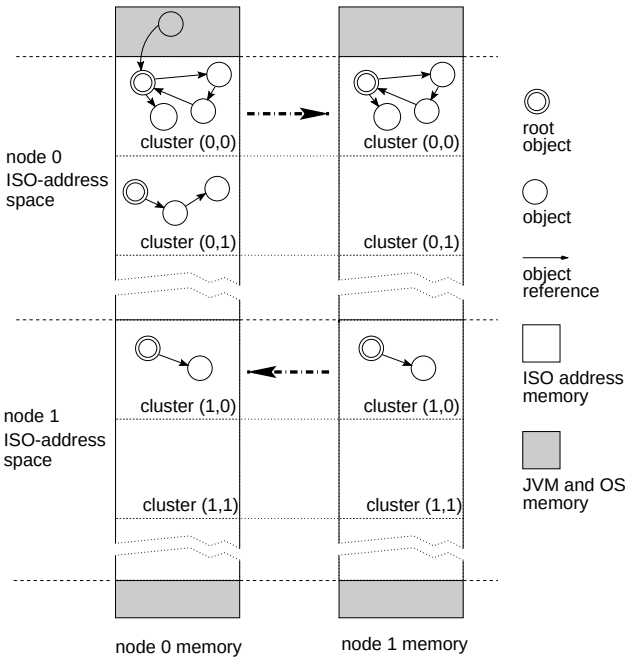

Figure 1. ISO-address memory for a two-node network

2. An another interest of clusters lies in the references between objects. References between two objects of different *ISO–cluster*s may be broken if one object is accessed before the second is transferred. Only pointers confined to a cluster are safeguard against transfer.

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.

Taking back to the previous example, the memory layout in figure 1 contains two clusters in the *ISO–cluster* space of node 0 and three clusters in the *ISO–cluster* space of node 1, numbered from $(0,0)$ to $(1,2)$. On node 0, four linked objects have been allocated in cluster $(0,0)$ and three objects in cluster $(0,1)$. On node 1, one cluster contains two linked objects, the others are empty. Node 1 memory also contains an object in cluster $(0,0)$ transferred from node 0. Similarly node 0 memory has received an object from node 1 in cluster $(1,0)$. The communication protocol is explained below.

As a cluster may contain several linked objects, one object – called the root object – acts as a handle to access the others indirectly.

Communications

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.

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. Once a cluster has been received, a reference to 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.

2.2 Expresso API

The ISO–address concept described above is implemented through a Java package. The main operations are located in the ClusterISO class. A brief overview and a simple example follow.

Class methods and attributes

The class method init makes all the necessary *Expresso* initializations and reserves memory from the operating system for the ISO–address space. Parameters may be given to adjust the size of the clusters and others details. Class method quit marks the end of the use of *Expresso*. It releases the ISO–address space memory. The class field myNode contains (after initialization) the current node number in the network.

Clusters creation

Clusters must be created to store transferable objects. Creation of a new cluster is obtained through the class constructor ClusterISO. Only one cluster can be accessible at a time. One chooses the cluster in which future objects will be allocated by calling the instance method setCurrent.

Transferable objects allocation

The class method newObject must be used to create transferable objects. This method takes the class of the object as parameter and returns a handle on a Java Object that should be cast to the desired class. 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. When an array of objects is communicated, it must also have been allocated with a similar method (newArray) from *ISO–cluster*. All the others objects should be allocated through the Javanew operator. Two complementary instance methods set (method setRootObject) and get (method getRootObject) the root object of a cluster.

Clusters exchange

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, 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. If the source of the message does not matter, ANY can be passed to the recv method in replacement of a node number.

Example

Figure 2 contains a simple illustration of the use of *Expresso*. This Java program runs on a two-node network. A binary tree is created on one side and sent to the other side where it is printed. As the same program runs on both nodes – 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 binary tree inside it, fills it and sends it to the second node. The second node receives the cluster, extracts the binary tree and prints it.

```
import expresso.*;              // Expresso package

public class TestISO {

  public static void main(String argv[]) {

    ClusterISO.init (argv); // initialize Expresso
    if ( ClusterISO.myNode == 0) { // first node code
      // create a cluster
      ClusterISO aCluster = new ClusterISO();
      // indicate which cluster will be filled
      ClusterISO.setCurrent(aCluster);
      // allocate a BTree inside the current cluster
      Btree tree =
          (Btree) ClusterISO.newObject(Btree.class);
      // fill the BTree
      tree.init (5);
      tree.addLeftChild(10);
      tree.addRightChild(25);
      // set the cluster root object
      aCluster.setRootObject(tree);
      // send the whole tree to the second node
      aCluster.send(1);
    } else { // second node code
      // receive the first cluster of the first node
      ClusterISO aCluster = ClusterISO.recv(0,0);
      // extract the root object
      Btree tree = ( Btree) aCluster.getRootObject();
      // print its contents
      System.out.println(tree);
    }
    ClusterISO.quit(); // stop using Expresso
  }
}
```

Figure 2. Binary tree transfer between two nodes with Expresso

3 Implementation

The ClusterISO class described in the previous section is the user interface of the *Expresso* library. This section contains a picture of how things are implemented. Performance figures are given afterwards.

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. A memory space distinct from the *JVM* heap is required to implement the concept of ISO–address. The memory management module of *Expresso* asks the operating system for this ISO–address space within the very large part of the 2^{32} bytes of memory available to the Unix process that is left free between the stack and the heap segment (see figure 3).

At initialization time, the local ISO–address space is allocated by a brk system call. It is managed as explained in the previous section. *Expresso* splits it up into as many chunks as the number of nodes: each of these chunks is numbered and is dedicated to objects created on a particular node. An extra chunk is also needed for the dispatch tables copies (see below).

The *Expresso* newObject method reserves a placeholder for an object in the local ISO–address space part dedicated to the executing node. The structure of this 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

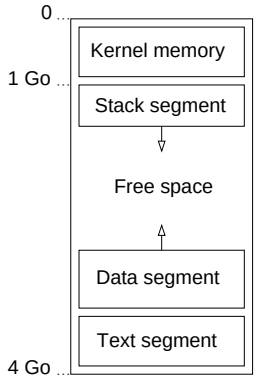

Figure 3. Address space of a Unix process

bound to the *Kaffe JVM* implementation (version 1.0.5)[1]: 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.

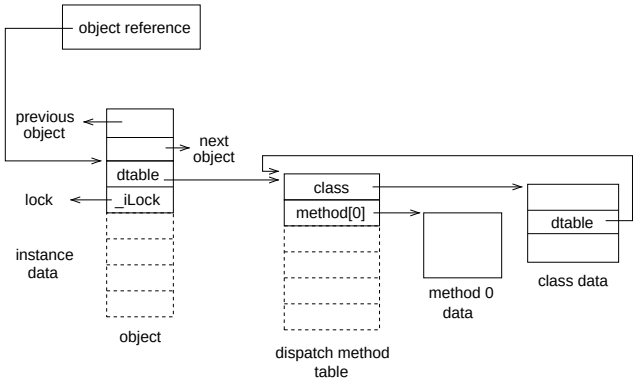

Figure 4. Object structure in the Kaffe virtual machine

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 and the *JVM* object heap; it represents a situation where several objects allocated in the ISO–address memory (a1, a2) and in the *JVM* memory (a3) share the same class data (data of class A). The classes of the program whose instances may be transferable are loaded by a custom class loader during *Expresso* initialization. For a given class, the class data (methods' code, attributes,...) are allocated normally within the *JVM* heap. Similarly, the dispatch table, which serves as an intermediate between the object reference and the methods' code, is loaded normally, within the *JVM* heap. At this stage, *Expresso* copies the dispatch table in a specific zone within the ISO–address space. Objects created with newObject contain a pointer to the copy of the dispatch table whereas objects created by the *JVM* contain a pointer to the original dispatch table. As every node load all the classes, the dispatch tables are allocated at the same addresses on all the nodes. So the dtable link between an *Expresso* object and its dispatch table is preserved during transfer and does not need any updating.

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

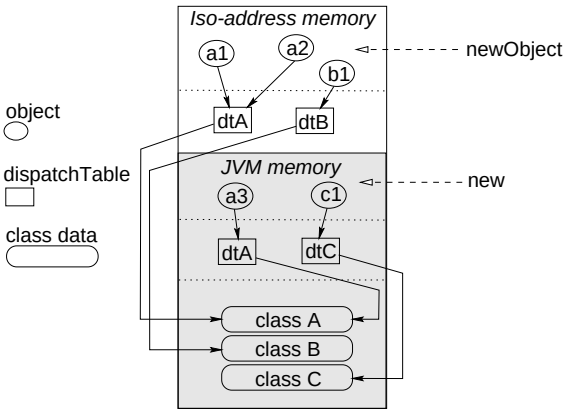

Figure 5. Links between memory spaces

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*. Figure 6 summarizes the communication layers stacking.

Expresso methods ClusterISO.recv and ClusterISO.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. On message receipt, words of memory are written at the same virtual address. which is computed from the cluster number and the origin node number (see paragraph 2.1).

Libraries	*Functions / Methods*
ClusterISO.class (Java class)	ClusterISO.recv() Cluster.send()
ClusterISO.so (JNI, C dynamic library)	expresso_recv() expresso_send()
modules from libmpich.a (C static library)	MPI_Init(), MPI_Send(), MPI_Recv(), MPI_Probe(), MPI_Finalize()
modules from libgm.a (C static library)	gm_open(), gm_send(), gm_receive(), gm_provide_receive_buffer()

Figure 6. Communication layers

3.3 Performances

We have experimented the *Expresso* library on a cluster of Linux Workstation (Pentium 400 Mhz, kernel 2.0.36). The nodes are linked by a Myrinet network with PCI 33 Mhz interface adapters (processor Lanai 4) and a 8-port switch. In the following, we show performance figures of basic *Expresso* primitives.

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. 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). With the first interface (serializable), serialization is automatically performed and relies on introspection mechanisms. With the externalizable interface, we had to implement the two methods writeObject() and readObject() that produces and exploits the serialized form of an object. The graph of objects transferred is a binary tree. Each node is composed of an integer and two references (to the left and right sub-trees). Table 1 shows times in μs obtained with the three versions. The last two columns give the gain factor of *Expresso*.

Times for *Expresso* objects transfer are very close to those of a simple message passing by MPI since there is nearly no extra work to perform. This is obviously not the case when using the standard Java interfaces.

4 Conclusion

We have presented in this paper *Expresso*, a library dedicated to the transfer of complex Java objects. It allows clusters of objects to be efficiently communicated between two nodes thanks to an ISO-address transfer of the memory representation of the objects. No costly serialization is thus needed, bringing up performances that make possible the full exploitation of the capabilities of high speed networks.

Although *Expresso* should mainly be used by upper level libraries, it can also be used to program distributed applications directly. This way, we have written with *Expresso* a genetic programming application [4]. The overall performances obtained show that a simple parallelization method can give good results as far as efficient object transfer is possible.

Our prototype can be improved in many ways. We should be able to provide object communication facilities at the thread level. This will be possible when underlying communication libraries fully support multi-threading. Another improvement would be to integrate the ISO-address space into the *JVM*.

More generally, the good performances of our prototype indicate that the use of a high performance network tends to question established methods tested for "slow" networks. Serialization for object transfer has been our prior concern. It appeared that the way objects are allocated is of importance. Similarly, other mechanisms should be examined to be adapted, regarding communication in a high performance network.

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. Rapport interne, Valoria, Univ. de Bretagne Sud, mars 2000. http://www.univ-ubs.fr/valoria/rapports/.

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

$Number\ of\ objects$	$Serialisable$	$Externalizable$	$Expresso$	$\frac{Serialisable}{Expresso}$	$\frac{Externalisable}{Expresso}$
10	11,700	8,900	210	55	42
100	21,300	10,800	470	45	22
1000	143,100	35,300	1,700	84	20
10000	5,083,000	1,208,000	16,200	313	74

Table 1. Performances of Java API and Expresso

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

[7] 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.

[8] 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.

[9] 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.