# EPOS and Myrinet: Effective Communication Support for Parallel Applications Running on Clusters of Commodity Workstations*

Antônio Augusto Fröhlich[1], Gilles Pokam Tientcheu[1], and Wolfgang Schröder-Preikschat[2]

[1] GMD FIRST
Kekulésraße 7
D-12489 Berlin, Germany
{guto|pokam}@first.gmd.de
http://www.first.gmd.de/~guto
[2] University of Magdeburg
Universitätsplatz 2
D-39106 Magdeburg, Germany
wosch@cs.uni-magdeburg.de
http://ivs.cs.uni-magdeburg.de/~wosch

**Abstract** This paper presents the EPOS approach to deliver parallel applications a high performance communication system. EPOS is not an operating system, but a collection of components that can be arranged together to yield a variety of run-time systems, including complete operating systems. This paper focuses on the communication subsystem of EPOS, which is comprised by the *network adapter* and *communicator* scenario-independent system abstractions. Like other EPOS abstractions, they are adapted to specific execution scenarios by means of scenario adapters and are exported to application programmers via inflated interfaces. The paper also covers the implementation of the *network adapter* system abstraction for the Myrinet high-speed network. This implementation is based on a carefully designed communication pipeline and achieved unprecedented performance.

## 1 Introduction

Undoubtedly, one of the most critical points to support parallel applications in distributed memory systems is communication. The challenge of enhancing communication performance, especially in cluster of commodity workstations, has motivated numerous well succeeded initiatives. At the hardware side, high-speed network architectures and fast buses counts for low-latency, high-bandwidth inter-node communication. At the software side, perhaps the most

* This research has been partially supported by the Federal University of Santa Catarina, by CAPES Foundation grant no. BEX 1083/96-1 and by Deutsche Forschungsgemeinschaft grant no. SCHR 603/1-1.

significant advance has been to move the operating system out of the communication pathway. In this context, several forms of active messages, asynchronous remote copy, distributed shared memory, and even optimized versions of the traditional send/receive paradigm, have been proposed. Combined, all these initiatives left the giga-bit-per-second, application-to-application bandwidth barrier behind [7].

Nevertheless, good communication performance is hard to obtain when dealing with anything but the test applications supplied by the communication package developers. Real applications, not seldom, present disappointing performance. We believe many performance losses to have a common root: the attempt to deliver a generic, all-purpose solution. Most research projects on high performance communication are looking for "the best" solution for a given architecture. However, a definitive best solution, independently of how fine-tuned to the underlying architecture it is, does not exist, whereas parallel applications simply communicate in quite different ways. Aware of this, many communication packages claim to be "minimal basis", upon which application-oriented abstractions can (have to) be implemented. One more time, there cannot be a best minimal base for all possible communication strategies. This contradiction between generic and optimal is presented in details in [8], and serves as motivation for the project EPOS.

In EPOS [5], we intend to give each application its own run-time support system, specifically constructed to satisfy its requirements (and nothing but its requirements). EPOS is not an operating system, but a collection of components that can be arranged together in a framework to yield a variety of run-time systems, including complete operating systems. Besides application-orientation, the project aims on high performance and scalability to support parallel computing on clusters of commodity workstations. The following sections describe EPOS communication system design, its implementation, and its performance.

## 2 Communication System Design

EPOS has been conceived following the guidelines of traditional object-oriented design. However, scalability and performance constrains impelled us to define some EPOS specific design elements. These design elements will be described next in the realm of the communication system.

### 2.1 Scenario-independent System Abstractions

Granularity plays a decisive role in any component-based system, since the decision about how fine or coarse components should be have serious implications. A system made up of a large amount of fine components will certainly achieve better performance than one made up of a couple of coarse components, since less unneeded functionality incurs less overhead. Nevertheless, a large set of fine components is more complex to configure and maintain.

In EPOS, visible components have their granularity defined by the smallest-yet-application-ready rule. That is, each component made available to application programmers implements an abstract data type that is plausible in the application's run-time system domain. Each of these visible components, called *system abstractions*, may in turn be implemented by simpler, non application-ready components.

In any run-time system, there are several aspects that are orthogonal to abstractions. For instance, a set of abstractions made SMP safe will very likely show a common pattern of synchronization primitives. In this way, we propose EPOS system abstractions to be implemented as independent from execution scenario aspects as possible. These adaptable, scenario-independent system abstractions can then be put together with the aid of a *scenario adapter*.

Communication is handled in EPOS by two sets of system abstractions: *network adapters* and *communicators*. The first set regards the abstraction of the physical network as a logical device able to handle one of the following strategies: datagram, stream, active message, or asynchronous remote copy. The second set of system abstractions deals with communication end-points, such as links, ports, mailboxes, distributed shared memory segments and remote object invocations. Since system abstractions are to be independent from execution scenarios, aspects such as reliability, sharing, and access control do not take part in their realizations; they are "decorations" that can be added by scenario adapters.

For most of EPOS system abstractions, architectural aspects are also seen as part of the execution scenario, however, network architectures vary drastically, and implementing unique portable abstractions would compromise performance. As an example, consider the architectural differences between Myrinet and SCI: a portable active message abstraction would waste Myrinet resources, while a portable asynchronous remote copy would waste SCI resources. Therefore, realizations for the *network adapter* system abstraction shall exist for several network architectures. Some abstractions that are not directly supported by the network will be emulated, because we believe that, if the application really needs (or wants) them, it is better to emulate them close to the hardware.

### 2.2 Scenario Adapters

EPOS system abstractions are adapted to specific execution scenarios by means of *scenario adapters*. Currently, EPOS scenario adapters are classes that wrap system abstractions, so that invocations of their methods are enclosed by the `enter` and `leave` pair of scenario primitives. These primitives are usually inlined, so that nested calls are not generated. Besides enforcing scenario specific semantics, scenario adapters can also be used to "decorate" system abstractions, i.e., to extend their state and behavior. For instance, all abstractions in a scenario may be tagged with a capability to accomplish access control.

In general, aspects such as application/operating system boundary crossing, synchronization, remote object invocation, debugging and profiling can easily be modeled with the aid of scenario adapters, thus making system abstractions, even if not completely, independent from execution scenarios.

The approach of writing pieces of software that are independent from certain aspects and later adapting them to a given scenario is usually referred to as *Aspect-Oriented Programming* [2]. We refrain from using this expression, however, because much of AOP regards the development of languages to describe aspects and tools to automatically adapt components (*weavers*). If ever used in EPOS, AOP will give means but not goals.

### 2.3 Inflated Interfaces

Another important decision in a component-based system is how to export the component repository to application programmers. Every system with a reasonable number of components is challenged to answer this question. Visual and feature-based selection tools are helpless if the number of components exceeds a certain limit —depending on the user expertise about the system, in our case the parallel application programmer expertise on operating systems. Tools can make the selection process user-friendlier, but certainly do not solve the user doubt about which selections to make. Moreover, users can usually point out what they want, but not how it should be implemented. That is, it is perhaps straightforward for a programmer to choose a mailbox as a communication end-point of a datagram oriented network, but perhaps not to decide whether features like multiplexing and dynamic buffer management should be added to the system.

The approach of EPOS to export the component (system abstraction) repository is to present the user a restricted set of components. The adoption of scenario adapters already hides many components, since instead of a set of scenario specific realizations of an abstraction, only one abstraction and one scenario adapter are exported. Nevertheless, EPOS goes further on hiding components during the system configuration process. Instead of exporting individual interfaces for each flavor of an abstraction, EPOS exports all of its flavors with a single *inflated interface*. For example, the datagram, stream, active message, and asynchronous remote copy *network adapters* are exported by a single `Network_Adapter` inflated interface as depicted in figure 1.

An inflated interface is associated to the classes that realize it through the *selective, partial realize* relationship. This relationship is partial because only part of the inflated interface is realized, and it is selective because only one of the realizations can be bound to the inflated interface at a time. Each selective realize relationship is tagged with a key, so that defining a value for this key selects a realization for the corresponding interface. The way this relationship is implemented enables EPOS to be configured by editing a single key table, and makes conditional compilations and "makefile" customizations unnecessary.

The process of binding an inflated interface to one of its realizations can be automated if we are able to clearly distinguish one realization from another. In EPOS, we identify abstraction realizations by the signatures of their methods. In this way, an automatic tool can collect signatures from the application and select adequate realizations for the corresponding inflated interfaces. Nevertheless, if two realizations have the same set of signatures, they must be exported by different interfaces.

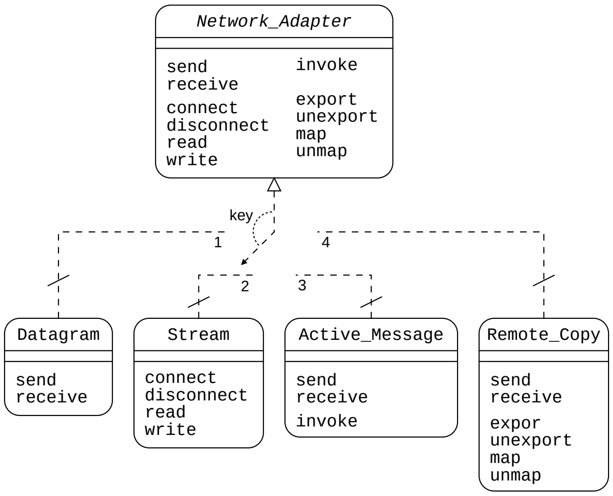


**Figure1.** The `Network_Adapter` inflated interface and its partial realizations.

The combination of *system abstractions*, *scenario adapters* and *inflated interfaces*, effectively reduces the number of decisions the user has to take, since the visual selection tool will present a very restricted number of components, of which most have been preconfigured by the automatic binding tool. Besides, they enable application programmers to express their expectations concerning the run-time system simply by writing down well-known system object invocations.

## 3 Communication System Implementation for Myrinet

EPOS is coded in C++ and is currently being developed to run either as a native ix86 system, or at guest-level on Linux. The ix86-native system can be configured either to be embedded in the application, or as $\mu$-kernel. The Linux-guest system is implemented by a library and a kernel loadable module. Both versions support the Myrinet high-speed network. EPOS communication system implementation is detailed next.

### 3.1 Platform Overview

EPOS is currently being implemented for a PC cluster available at GMD-FIRST. This cluster consists nowadays of a server and 16 work nodes, each with an AMD Athlon processor running at 550 MHz, 128 MB of memory on a 100 MHz bus, and a 32 bits/33 MHz PCI bus in which a Fast-Ethernet and a Myrinet network adapter are plugged. This platform and some performance figures have been introduced in [4], however, it is important to recall some of its characteristics in order to justify implementation decisions.

The Myrinet network adapter present in each node of our cluster has a processor, namely a LANai 4.1, 1 MB of memory and three DMA engines, respectively for transferring data between main memory and the memory on the network adapter, to send data to the network, and to receive data from the network. These DMA controllers can operate in parallel and perform two memory accesses per processor cycle. The memory on the Myrinet adapter is used to store the LANai control program and as communication buffer as well; it is also mapped into the main processor's address space, thus enabling data transfers without DMA assistance (programmed I/O).

A simple message exchange can be accomplished by using programmed I/O or DMA to write the message into the memory on the Myrinet adapter, and then signaling to the control program, by writing a shared flag, that a message of a given size is available in a certain memory location. The control program can then generate a message header with routing information and configure the send DMA controller to push the message into the network. The receiver side can be accomplished in a similar way, just adding a signal to the main processor to notify that a message has arrived. This can be done either by a shared flag polled by the main processor or via interruptions.

If the memory management scheme adopted on the node uses logical address spaces that are not contiguously mapped into memory, additional steps have to be included in order to support DMA. Epos can be configured to support either a single task (the typical case for MPI applications running on single processor nodes) or several tasks per node. The ix86-native, single-task version does not need any additional step, since logical and physical address spaces do match. The multi-tasking and Linux-guest version, however, allocate a contiguous buffer, of which the physical address is known, and give programmers two alternatives: write messages directly into the allocated buffer; or have messages copied into it.

Figure 2 depicts a message exchange between two applications (including the additional copies). The data transfer rate for each stage has been obtained and is approximately the following: 140 MB/s for the copy stages 1 and 5; 130 MB/s for the host/Myrinet DMA stages 2 and 4; and 160 MB/s for the send and receive DMA stages 3.1 and 3.2. Therefore, the total data transfer rate is limited to 130 MB/s by the host/Myrinet DMA stages.

### 3.2 Communication Pipeline

In order to deliver applications a communication bandwidth close to the 130 MB/s hardware limit the software overhead must be reduced to an insignificant level. Fortunately, a careful implementation and several optimization can help to get close to this limit. To begin with, the DMA controllers in the Myrinet adapter are able to operate in parallel, so that stages 2 and 3.1 of figure 2, as well as stages 4 and 3.2, can be overlapped. However, these stages are not intrinsically synchronous, i.e., there is no guarantee that starting stage 3.1 just after starting stage 2 will preserve message integrity. Therefore, overlapping will only be possible for different messages or, what is more interesting, different

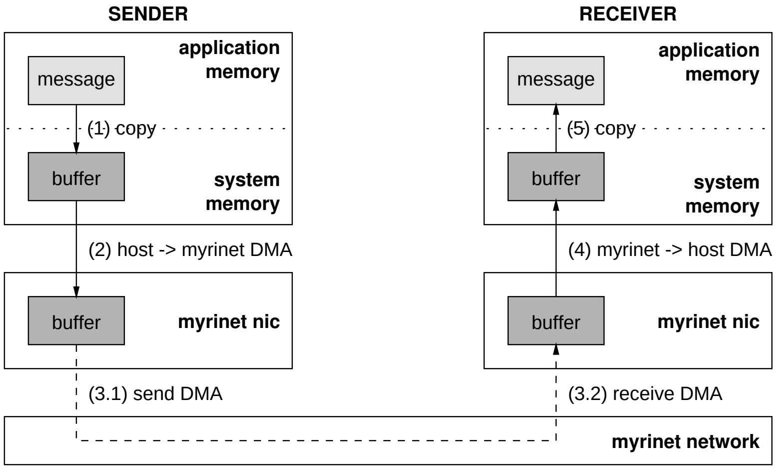


**Figure2.** Steps involved in a message exchange.

pieces of a message. We took advantage of this architectural feature to implement a communication pipeline for EPOS.

EPOS communication pipeline for Myrinet has been designed considering the time messages of different sizes spend at each stage of figure 2. This time includes the overhead for the stage (per-packet cost) and its effective data transfer rate (per-byte cost). It is important to notice that the overhead includes synchronization operations and the waiting time for the next stage to become available. According to Myrinet documentation, the delay between stages 3.1 and 3.2 is of 0.5 $\mu$s per switch hop. As this latency is much smaller then any other in the pipeline, we will consider stages 3.1 and 3.2 to completely overlap each other, thus yielding a single pipeline stage 3.

A message sent through the network is now split in small packets that move through the stages of the pipeline. In order to sustain a transfer rate close to the maximum, at least two requirements must be fulfilled: first, the number of packets must be at least equal to the depth of the pipeline (five in our case), and second, the packet length must be such as to minimize the total message transmission time. To determine the optimal packet length for the pipeline, we modeled it according to the following rules:

- As soon as a packet $n$ leaves a pipeline stage $i$, it enters the next stage $i+1$;
- A packet $n$ enters stage $i$ of the pipeline as soon as packet $n-1$ leaves it.

In this way, we can compute the optimal packet length $P_{optimal}$ by looking for a maximum of the function $T(L,P)$, which represents the time a message of size $L$, split in packets of length $P$, takes to move through the pipeline [3]. This function can be expressed as the sum of the time a packet spends on each of the pipeline stages. Using a linear cost model [1], the time spent by a packet $n$ in a stage $i$ can be expressed as:

$$T_i^{(n)} = \beta_i + P\tau_i \qquad (1)$$

where $\beta_i$ is the constant per-packet cost assigned to stage $i$, $P$ is the packet size, and $\tau_i$ the inverse of stage $i$ bandwidth.

Taking in consideration the characteristics of the platform described earlier, we can assume the cost of the copy stages 1 and 5 to be approximately the same, since they involve the same hardware resources, i.e., processor and memory. For the same reason we can consider the cost of stage 2 to be approximately the same of stage 4. Therefore, $\beta_1 \cong \beta_5$ and $\tau_1 \cong \tau_5$, while $\beta_2 \cong \beta_4$ and $\tau_2 \cong \tau_4$. Since the bandwidth of the Myrinet network (160 MB/s) is greater than that of main memory copy (140 MB/s), which in turn is greater than that of the host/Myrinet DMA (130 MB/s), we can write the following inequality:

$$\tau_2 \cong \tau_4 > \tau_1 \cong \tau_5 > \tau_3 \qquad (2)$$

From the inequality 2 above, we can deduce an upper bound for the delay experienced by a packet traveling through the stage 3 of the pipeline (inequalities 3 and 4); and also the delay experienced by the last packet of the message at stage 4 (inequality 5)[1].

$$\beta_1 + \beta_2 + \beta_3 + P\tau_2 > \beta_3 + P\tau_3 \qquad (3)$$

$$\beta_1 + \beta_2 + \beta_3 + P\tau_2 > \beta_1 + P\tau_1 \qquad (4)$$

$$\beta_1 + \beta_2 + P\tau_2 > \beta_1 + P\tau_1 \qquad (5)$$

Assuming that the message size $L$ is an integer multiple of $P$, the function $T(L,P)$ could be expressed as:

$$T(L,P) = \sum_{k=1}^{2} T_k^{(1)} + \sum_{i=1}^{\lceil \frac{L}{P} \rceil} T_3^{(i)} + \sum_{j=4}^{5} T_j^{(\lceil \frac{L}{P} \rceil)} \qquad (6)$$

From the inequalities above, and from the approximations suggested earlier, we can express $T(L,P)$ as:

$$T(L,P) \cong 2(2\beta_1 + \beta_2) + 2(\tau_1 + \tau_2)P + \lceil \frac{L}{P} \rceil (\beta_1 + \beta_2 + \beta_3 + P\tau_2) \qquad (7)$$

and the optimal packet size can be computed by solving the following equation:

$$\frac{dT(L,P)}{dP} = 0 \Rightarrow P_{optimal} = \sqrt{\frac{L(\beta_1 + \beta_2 + \beta_3)}{2(\tau_1 + \tau 2)}} \qquad (8)$$

[1] The overhead of upstream pipeline stages has to be considered because there is no analytical way to determine a relation among them. Even measurements are not possible, since execution time is not predictable in our CPUs.

By applying a linear regression approximation [6] to each stage of the pipeline on the set of predictor variables, which has been obtained from the non-pipelined transfer of messages of various sizes, we computed the values for $\beta$ and $\tau$ to be approximately the ones given in table 1.

**Table1.** Computed values for the per-packet cost $\beta$ and inverse bandwidth $\tau$ of each pipeline stage.

| stage | $\beta_i$(in $\mu s$) | $\tau_i$(in $\frac{\mu s}{KB}$) |
|---|---|---|
| 1, 5 | 3.0 | 5.9 |
| 2, 4 | 0.6 | 7.6 |
| 3 | 1.3 | 7.5 |

As an example, the calculated optimal packet length for a 64 KB message is 3490 bytes. In practice, 4 KB packets are to be used. We solved equation 8 for several message lengths in order to determine the optimal packet length for ranges of message lengths. With this information in hand, we implemented an adaptive pipeline that automatically selects the appropriate packet length according to the message length, thus minimizing the message transfer latency.

### 3.3 Short Messages

Although the pipeline described above has a very low intrinsic overhead, programming DMA controllers and synchronizing pipeline stages may demand more time than it is necessary to send a short message via programmed I/O. In order to optimize the transfer of short messages using programmed I/O, which usually has a mediocre performance on PCs, we instructed our processors to collect individual write transactions that would traverse the PCI bridge to form 32 bytes chunks. Each chunk is then transferred in a burst transaction. This feature is enabled by selecting a "combine" cache policy for the pages that map the memory on the Myrinet adapter into the address space of the process. For the current implementation, messages shorter than 256 bytes are transferred in this way.

### 3.4 Performance Evaluation

We evaluate the performance of Epos *network adapter* system abstraction, by measuring the latency and the bandwidth experienced at the application level. A single-task, ix86-native and a Linux-guest version have been considered. The one-way tests have been executed in the platform previously described and consist of one node sending messages of different sizes to an adjacent node, i.e., one connected to the same Myrinet switch. The results are presented in figure 3, and, when compared to the 130 MB/s limit, give an efficiency rate of 85% for 64 KB messages in the Linux-guest version and 92% in the native version.

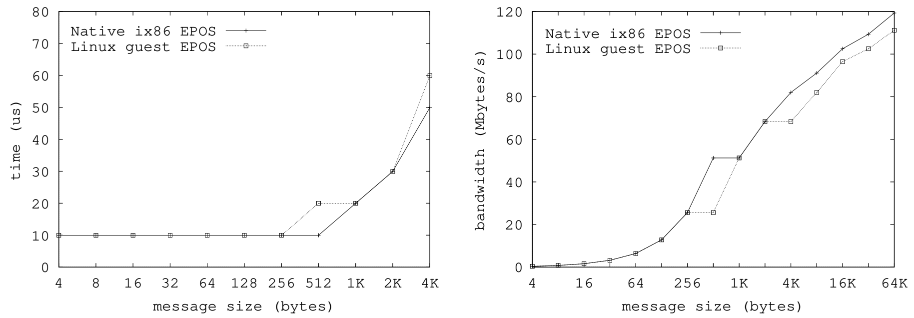


**Figure3.** Epos *network adapter* one-way latency (left) and bandwidth (right).

## 4 Conclusion

In this paper we presented the Epos approach to deliver a high performance communication system to parallel applications running on clusters of commodity workstations. We demonstrated how *system abstractions*, *scenario adapters* and *inflated interfaces* can simplify the process of run-time system configuration, mainly by reducing the number of decisions the user has to take. We also describe the *network adapter* system abstraction implementation for the Myrinet high-speed network that interconnects the nodes in our PC cluster.

The results obtained so far are highly positive and help to corroborate Epos design decisions. The evaluation of Epos *network adapter* abstraction revealed performance figures that, as far as we are concerned, have no precedents in the Myrinet interconnected PC cluster history. However, Epos is a long term, open project that aims to deliver application-oriented run-time systems. Many system abstractions, scenario adapters, and tools are still to be implemented in order to support a considerable set of applications.

## References


[1] J. de Rumeur. *Communications dans les Réseaux de Processeurs*. Masson, 1994.

[2] G. Kiczales et al. Aspect-Oriented Programming. In *Proc. ECOOP'97*, Springer-Verlag, 1997.

[3] R. Wang et al. Modeling Communication Pipeline Latency. In *Proc. SIGMETRICS'98*, June 1998.

[4] A. Fröhlich and Schröder-Preikschat. SMP PCs: A Case Study on Cluster Computing. In *Proc. First Euromicro Workshop on Network Computing*, August 1998.

[5] A. Fröhlich and Schröder-Preikschat. High Performance Application-Oriented Operating Systems – the EPOS Aproach. In *Proc. 11th SBAC-PAD*, September 1999.

[6] R. Jain. *The Art of Computer Performance Analysis*. John Willey & Sons, 1991.

[7] L. Prylli and B. Tourancheau. BIP: a New Protocol Designed for High Performance Networking on Myrinet. In *Proc. PC-NOW'98*, April 1998.

[8] W. Schröder-Preikschat. *The Logical Design of Parallel Operating Systems*. Prentice-Hall, 1994.