Design and Implementation of EPOS Communication System for Fast Ethernet

Fernando Barreto and Antônio Augusto Fröhlich
UFSC/CTC/LISHA
PO Box 476
88049-900 Florianópolis - SC, Brazil
{fbarreto|guto}@lisha.ufsc.br
http://www.lisha.ufsc.br/∼{fbarreto|guto}

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 ordinary Fast Ethernet networks, completing a previous article that described the implementation for the Myrinet high-speed network.*

1 Introduction

The parallel computing community has been using clusters of commodity workstations as an alternative to expensive parallel machines for several years by now. The results obtained meanwhile, both positive and negative, often lead to the same point: inter-node communication. Consequently, much effort has been dedicated to improve communication performance in these clusters: from the hardware point of view, high-speed networks and fast buses provide for low-latency and high-bandwidth; while from the software point of view, *user-level communication* [2] enables applications to access the network without operating system intervention, significantly reducing the software overhead on communication.

Nevertheless, good communication performance is hard to obtain when dealing with anything but the test applications supplied by the developers of the communication package. Real applications, not seldom, present disappointing performance figures [8]. We believe the origin of this shortcoming to be in the attempt of delivering generic communication solutions. Most high-performance communication systems are engaged in a "the best" solution for a certain architecture. However, a definitive best solution, independently of how well tuned to the underlying architecture it is, cannot exist, since parallel applications 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. Once more, there cannot be a best minimal basis for all possible communication strategies. This contradiction between generic and optimal is consequently discussed in [10].

If applications communicate in distinct ways, we have to deliver each one a tailored communication system that satisfies its requirements (and nothing but its requirements). Of course we cannot implement a new communication system for each application, what we can do is to design the communication system in such a way that it becomes possible to tailor it to any given application. In the *Embedded Parallel Operating System* (EPOS) project [5], we developed a novel design method that is able to accomplish this duty. EPOS consists of a collection of components, a component framework, and tools to support the automatic construction of a variety of run-time systems, including complete operating systems.

The particular focus of this paper is the implementation of EPOS communication system for ordinary FAST ETHERNET networks. In the next sections, EPOS communication system design will be discussed. The implementation of this communication system will be discussed later, including a preliminary performance evaluation. The paper is closed with authors' conclusions.

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* [7]. 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.

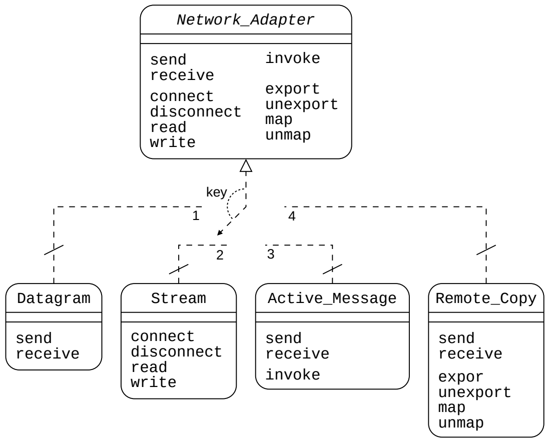

Figure 1. The Network_Adapter inflated interface and its partial realizations.

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.

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 Fast Ethernet

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 μ-kernel. The Linux-guest system is implemented by a library and a kernel loadable module. EPOS communication system implementation is detailed next.

To provide a Fast Ethernet communication suport to the EPOS system in a parallel cluster, came the ideia of create a communication mechanism. This mechanism may be considered a new protocol that should use the network speed technology at its maximum to execute the message exchanging. To reach this objective, the protocol must implement only the minimum requirements so that the system can transmit and receive a network package.

The architecture of this communication protocol, comparing with the OSI protocol stack (ISO, 1981), is located only at physic, data link, and application layers. This protocol is located only at these layers because parallel aplications in a dedicated cluster architecture do not need of the resources of the other layers. With these characteristics, the implementation of these layers is avoided and also the resulting overhead caused by them.

In the context of the EPOS object oriented system, these layers would be abstracted among members of three families. One member is the interface with user application, representing in the form of a mailbox variation from Communicators family, and its called cluster_com. The other member stores the methods and essential attributes in the communication mechanism, represented as a variation of Network family, named cluster_net. There is a member represented as a Interrupt Handler family's member named cluster_int. This way the cluster_net and cluster_int members abstract the physic, data link layer and the cluster_com member with the application layer. The Communicator's attribute channel is a "neutral" attibute, it is only used to directly reference the network family from Communicators Family.

The cluster_com member acts in the handling of two modules calls (Send,Receive) that directly reference the cluster_net's methods by means of channel attribute. In other words, it acts as an interface to obtain the passed parameters, make the analisys, fill attributes end call the cluster_net's methods.

So that the user's application can use the resources of this communication protocol, the system EPOS supplies the inflated interface of Communicator's family. That allows to user's application to know how to use the parameters passed and how to execute the transmission or the reception. The same happens with the Network family, that supplies an inflated interface, allowing the Communicators family to reference Network's methods.

The cluster_net member adds, as variations of Network family, control attributes, an object queue and trasmission and reception methods that work directly with the methods of object attribute Device. These control attributes are filled by the information obtained from the parameters in the methods called from cluster_com, as much in transmission as in reception, being used to identify the destination or origin of a message, and execute it's handling. The members are showned in figure 2.

In this paper, the cluster_net members were projected to make use only of Device attribute, inherited from network family, of Fast Ethernet type to communication with the cluster.

The member cluster_int is responsable by handle all the packages incomming from NIC Ethernet in the system. The handling is made in the analisys of the package and its contents, taking measures according with the result of the analisys. The measures taken by the cluster_int member are signaling and filling the package cluster_net reception queue. The picture 2 presents the main responsabilities of cluster_int in the EPOS system.

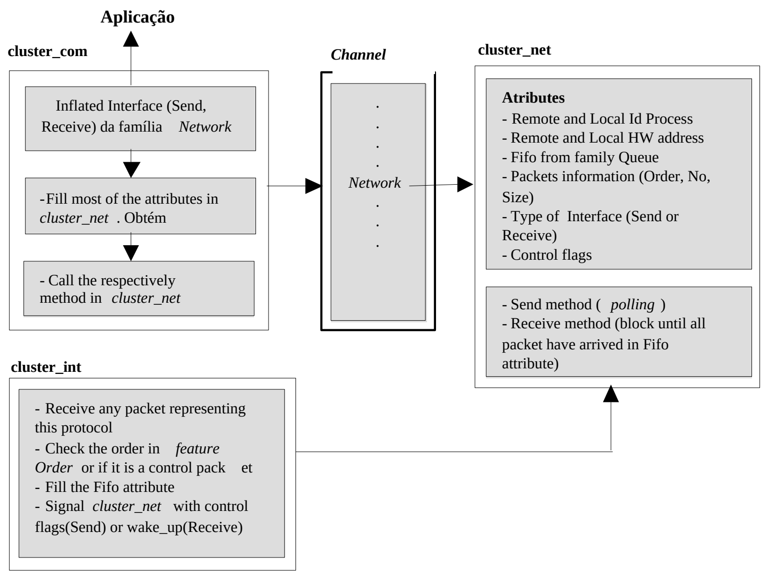
Aplicação
cluster_com
Inflated Interface (Send, Receive) da família *Network*
-Fill most of the attributes in *cluster_net* . Obtém
- Call the respectively method in *cluster_net*
Channel
Network
cluster_net
Atributes
- Remote and Local Id Process
- Remote and Local HW address
- Fifo from family Queue
- Packets information (Order, No, Size)
- Type of Interface (Send or Receive)
- Control flags
- Send method (*polling*)
- Receive method (block until all packet have arrived in Fifo attribute)
cluster_int
- Receive any packet representing this protocol
- Check the order in *feature Order* or if it is a control pack et
- Fill the Fifo attribute
- Signal *cluster_net* with control flags(Send) or wake_up(Receive)

Figure 2. EPOS **implementation**

The routines of transmission and reception of the cluster_net's communication mechanism are described in section 3.1 and 3.2

3.1. Transmission

In the transmission, the transmiting method called from cluster_com creates Ethernet Frames based in the information contained in the control attributes. The message is placed in the Ethernet package, beggining at memory region of the user process. In the case that the message is bigger then the data area of the Ethernet package including their headers, there is the need of the fragmentation. The number of fragments are generated only a quantity necessary to transmit a given message of any size.

The fragments will be sent as fast as possible by polling in the object attribute's method instanciated from family Device. This way , the implementation try to leave the NIC as used as possible by using the polling technic. This technic does not generate problems, because only one process is being executed in the host.

The transmission process is influenced by the cluster_int member, that identify the control packages comming from the receiever, signalizes cluster_net, reporting a broken sequence of packages or a final ACK. Both permiting that the Send method performs the measures according with the contents of the control package.

3.2. Reception

In the reception it's composed by a method of cluster_net called by cluster_com and by a cluster_int routine. The method called by cluster_com depends almost entirely from cluster_int routine.

The method in cluster_net, effects the filling of the control attributes of cluster_net, also puts the process in wait state until the message is ready to be trasfered from the kernel to the user space.

The cluster_int routines, represented as interruption handler, analises the header of the package that arrived with the existing control attributes in the object instantiated from cluster_net. The analisys tries to identify wich application is waiting for a message. After identify the application, the package is enqueued in the cluster_net's reception queue, that is an attribute instantiated from queue family of a fifo type when instantiating a cluster_net object.

The feature Order is used to check if there was a sequence break in the arriving packages. If was, a package is sent to the sender host reporting a sequence break. As the sender checks the break, it sends back to the receiver from the lost package.

The method in cluster_net called from cluster_com transfers the data from the receiver queue to the user space as soon as cluster_int signals cluster_net that the entire message was received. A control acknowledgement package is sent to the sender reporting that the message was successfully received.

4. EPOS guest level in Linux

To validate this communication protocol, a Linux module was created to provide support of this protocol to applications. The interface created by cluster_com is simulated in Linux as a additional code in IPC syscall. The cluster_net methods and attributes are declared in the module as a control structure to each application and function calls in the syscall. The member cluster_int is represented by a softirq and the standard routines of a NIC device driver. The figures 3 and 4 describes in details the kernel Linux data flow.

This protocol architecture in Linux is described in the following subsections.

4.1. Transmission

In transmission, the system call fills the control structure data with the syscall parameters. With this informations, the transmission function makes Ethernet frames filling the frame data region with the process user memory. With a created package, its transmission is realized enqueueing it in the transmission queue of Fast Ethernet device driver. This queue packages are processed by the network driver until the queue get empty or the network driver overloaded. If a overload occur, the package transmission is interrupted and reactivated when a network hardware became ready to process the queue again. This reactivation occurs by a network hardware interruption that release the queue to be processed again and set the transmission softirq in Linux.

The softirqs exist in Linux since kernel 2.3.43. They can be considered system routines that are executed at interruption level. Their main purpose is execute most of the tasks of the interruption handler. There are two network softirqs, the transmissor (NET_TX_SOFTIRQ) and the receiver (NET_RX_SOFTIRQ).

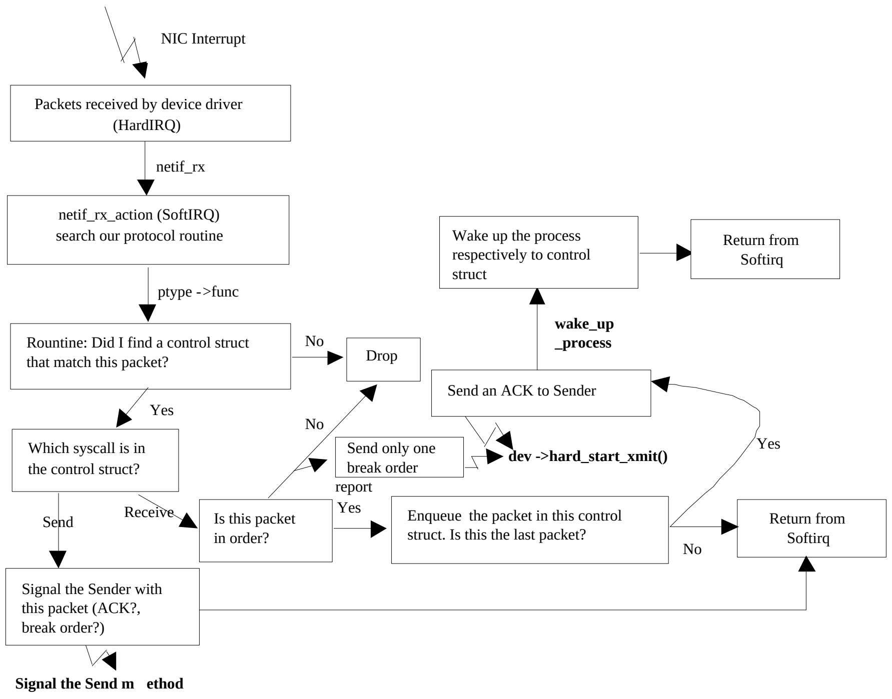
NIC Interrupt
Packets received by device driver (HardIRQ)
netif_rx
netif_rx_action (SoftIRQ) search our protocol routine
ptype ->func
Rountine: Did I find a control struct that match this packet?
No
Drop
Yes
Which syscall is in the control struct?
No
Send only one break order report
dev ->hard_start_xmit()
Send an ACK to Sender
wake_up _process
Wake up the process respectively to control struct
Return from Softirq
Yes
Send
Receive
Is this packet in order?
Yes
Enqueue the packet in this control struct. Is this the last packet?
No
Return from Softirq
Signal the Sender with this packet (ACK?, break order?)
Signal the Send m ethod

Figure 3. Package receiver softirq

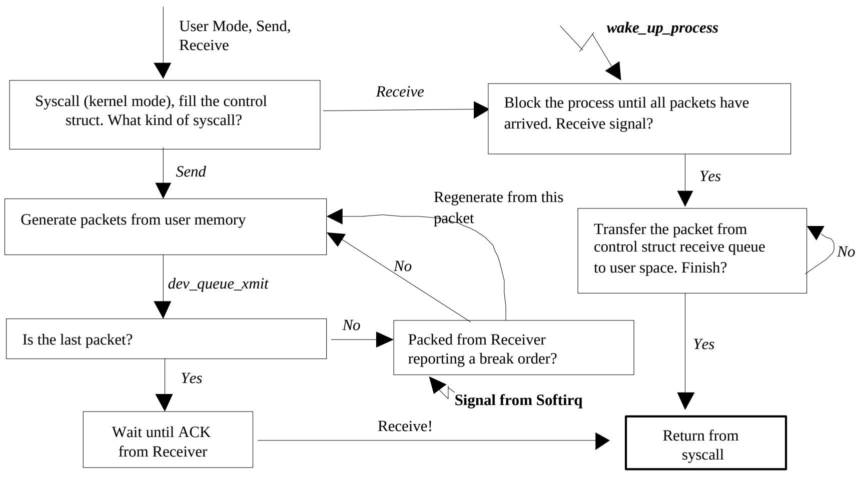
User Mode, Send, Receive
wake_up_process
Syscall (kernel mode), fill the control struct. What kind of syscall?
Receive
Block the process until all packets have arrived. Receive signal?
Send
Yes
Generate packets from user memory
Regenerate from this packet
Transfer the packet from control struct receive queue to user space. Finish?
No
No
dev_queue_xmit
Is the last packet?
No
Packed from Receiver reporting a break order?
Yes
Yes
Signal from Softirq
Wait until ACK from Receiver
Receive!
Return from syscall

Figure 4. Syscall user interface

4.2. Reception

In the reception process, the module establishes itself like a system call and as a receiver softirq routine.

Besides this system call fills the control structure data, it puts the process in wait state until the message become ready to be transfered from kernel to user space.

The softirq routine become responsable for context analisys of the Ethernet packages that arrive from the network driver.

When a package of this protocol arrives, it's handled by the driver interruption routine, then the receive softirq routine analises the package header and execute the receive routine related to the header identification. This routine locate among control structures the one that fits the package identification and puts it in the reception queue of this structure. When the routine checks that the entire message was already received, the softirq change the process state from waiting to ready.

When the system call runs again, it'll process the reception queue of its control structure transfering data from kernel to the process designated memory region referenced by the control structure. As soon as this task is completed, it returns to the user context.

As the package receive softirq routine realize that it's the last package being analised, it sends a control package to the sender host reporting that all packages were successfully received. This kind of sequence control simulate the feature order's use in network_adapter family.

The figure 5 ilustrate how the common flow of this communication mechanism works.

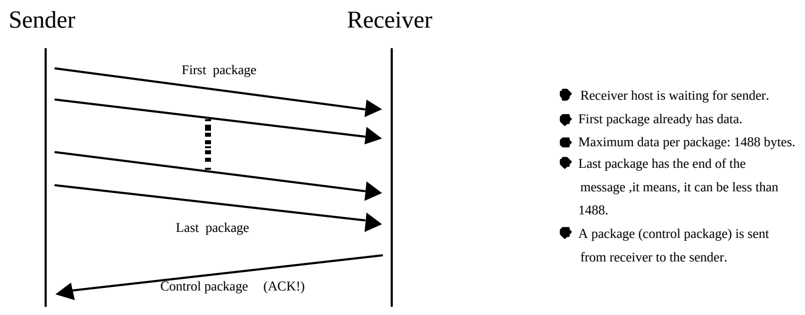
Sender
Receiver
First package
Last package
Control package (ACK!)
Receiver host is waiting for sender.
First package already has data.
Maximum data per package: 1488 bytes.
Last package has the end of the message ,it means, it can be less than 1488.
A package (control package) is sent from receiver to the sender.

Figure 5. Normal communication flow

As the receiver is always waiting for the message before the sender starts to transmit, the sincronization between them is dispensable. It enables the sender to begin the message transmission as soon as possible by polling technic.

Like the project in EPOS, this implementation has the following otimizations: less number of layers and consequently less header per package, routing and checksum inexistence and there is a simple control sequence.

The ideal communication,it means, without package loose, it'll be sent as many packages as it needs to mount the message. Just one control acknowledge package from receiver to sender is used to report that all fragments successfully arrived, as show in figure 5.

5. Results

In effects of comparission and a better look of this protocol, protocols GAMMA (CHIOLA, 2002) and TCP/IP were used in a cluster environment with Linux kernel 2.4.17.

The experiments realized with this protocol ran at AMDk6 64MBytes of RAM memory, Intel EtherExpress 100 network cards interconnected by a 3Com switch 3C16734B 100Mbits/s.

The GAMMA protocol, out of the context of this paper, has as its main advantage an use of the network bandwidth almost at maximum, because its data transfer occurs at device driver interruption level. This kind of implementation breaks the rules of Linux network system. This architecture receiver buffers are directly allocated in user space without Linux network system usage. However, important changes must be made in the network driver to manipulate these buffers. This protocol is completely dependent from the network hardwares that it can operate, it means it doesn't support any network hardware. The user process that uses this protocol should have its memory region forbbiden to be paginated, this avoid that a page fault at interruption level occurs. To avoid a memory region to be paginated, privileged permissions from the operating system are needed.

The figure 6 shows a comparative chart of the maximum throughputs reached in the experiments with the protocols GAMMA, TCP/IP and this one. All of them are compared to the ideal maximum allowed by physical environment of 12,5MB/s (BUYYA, 1999).

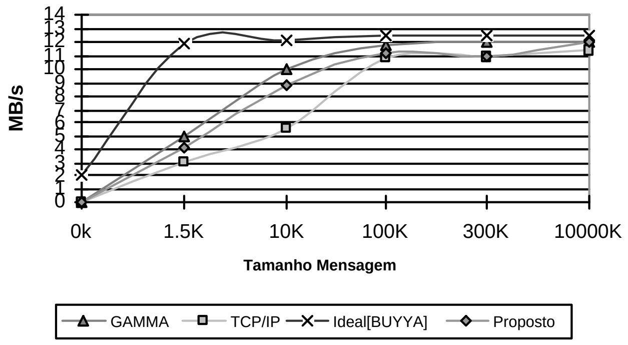

Figure 6. Communication performance

The chart data was obtained using tools of each one of the protocols to get maximum throughput. The GAMMA protocol used the "ping-pong" tool. TCP/IP used the "ttcp" tool. This protocol used a measuration in receiver system at system call level that evaluate the time to transfer data from sender host to memory user in receiver host. Both tools display the communication maximum throughput. Many tests battery were made and the higher throughput average were collected.

Over the chart analisys shown in figure 5, packages with small size (0 to 100KBytes) can be observed, this protocol showed itself more efficiently than TCP/IP. As most of the messages exchanged among in paralell cluster are of small sizes, this protocol can efficiently approach these types of message reaching its objective.

References

[1] M. Bar. *Linux Internals*. Osborne McGraw-Hill, 2000.

[2] R. A. F. Bhoedjang, T. Rühl, and H. E. Bal. User-level Network Interface Protocols. *IEEE Computer*, 31(11):53–60, Nov. 1998.

[3] R. Buyya, editor. *High Performance Cluster Computing: Architectures and Systems*, volume 1. Prentice Hall, 1999.

[4] G. Chiola and G. Ciaccio. Gamma: a low-cost network of workstations based on active messages. In *Proceedings of the 5th EUROMICRO Workshop on Parallel and Distributed Processing*, London, United Kingdom, Jan. 1997.

[5] A. A. Fröhlich and W. Schröder-Preikschat. Tailor-made Operating Systems for Embedded Parallel Applications. In *Proceedings of the 4th IPPS/SPDP Workshop on Embedded HPC Systems and Apllications*, volume 1586 of *Lecture Notes in Computer Science*, pages 1361–1373, San Juan, Puerto Rico, Apr. 1999. Springer.

[6] International Organization for Standardization. *Open Systems Interconnection - Basic Reference Model*, Aug. 1981. ISO/TC 97/SC 16 N 719.

[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-oriented Programming'97*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, June 1997. Springer.

[8] Numerical Aerospace Simulation Systems Division. *The NAS Parallel Benchmarks*, online edition, Nov. 1997. [http://www.nas.nasa.gov/Software/NPB/NPB2Results/index.html].

[9] A. Rubini and J. Corbet. *Linux Device Drivers*. O'Reilly, 2 edition, 2001.

[10] W. Schröder-Preikschat. *The Logical Design of Parallel Operating Systems*. Prentice-Hall, Englewood Cliffs, U.S.A., 1994.