An Application-Oriented Communication System for Clusters of Workstations

Thiago Robert C. Santos and Antonio Augusto Frohlich

Laboratory for Software/Hardware Integration (LISHA)
Federal University of Santa Catarina (UFSC)
88049-900 Florianopolis - SC - Brazil
PO Box 476
Phone: +55 48 331-7552 Fax: +55 48 331-9770
{robert | guto}@lisha.ufsc.br
http://www.lisha.ufsc.br/~{robert | guto}

Abstract

The present paper proposes an application-oriented communication sub-system to be used in SNOW, a high-performance, application-oriented parallel-programming environment for dedicated clusters. The proposed communication sub-system is composed of a baseline architecture and a family of lightweight network interface protocols. Each one of these protocols is built on top of the baseline architecture and can be tailored in order to satisfy the needs of specific classes of parallel applications. The family of lightweight protocols, along with the baseline architecture that supports it, consists of a customizable component in EPOS, an application-oriented, component-based operating system that is the core of SNOW. The idea behind providing a set of low-level protocol implementations instead of a single monolithic protocol is that parallel applications running on dedicated clusters can improve their communication performance by using the most appropriate protocol for their needs.

Keywords: lightweight communication protocols, application-oriented operating systems, user-level communication.

1 Introduction

Clusters of commodity workstations are now commonplace in *high-performance computing*. In fact, commercial off-the-shelf processors and high-speed networks evolved so much in recent years that most of the hardware features once used to characterize *massively parallel processors* (MPP) are now available in clusters as well. Nonetheless, the majority of the clusters in use today rely on commodity run-time support systems (run-time libraries, operating systems, compilers, etc.) that have usually been designed in disregard of both parallel applications and hardware. In such systems, delivering a parallel API like MPI is usually achieved through a series of patches or middleware layers that invariably add on overhead for applications. Therefore, it seems logical to suppose that a run-time support system specially designed to support parallel applications on clusters of workstations could considerably improve on performance and also on other software quality metrics (e.g. usability, correctness, adaptability).

Our supposition that ordinary run-time support systems are inadequate to support high-performance computing is sustained by a number of research projects in the field focusing on the implementation of *message passing* [11, 3] and *shared memory* [1, 5, 12] middlewares and on *user-level communication* [15, 14, 17]. If ordinary operating systems could match with parallel application's needs, delivering adequate run-time support for the most traditional programming paradigms with minimum overhead, many of theses researches would be hard to justify outside the realm of operating systems. Indeed,

the way ordinary operating systems handle I/O is largely based on multitasking concepts such as domain protection and resource sharing. This impacts the way recurring operations like system calls, CPU scheduling, and application's data management are implemented, making little room for novel technological features [13]. Not surprisingly, system designers often have to push the operating system out of the way in order to implement efficient schedulers and communication systems for clusters.

Besides, commodity operating systems usually target reconfigurability at standard conformance and device support, failing to comply with applications' requirements. Clusters have been quenching the industry's thirst for low-end supercomputers for years as HPC service providers deploy cost-effective solutions based on cluster systems. There are all kinds of applications running on clusters today, ranging from communication-intensive distributed databases to CPU-hungry scientific applications. Having the chance to customize a cluster's run-time support system to satisfy particular applications' needs could improve the system's overall performance. Indeed, systems such as PEACE [16] and CHOICES [4] have already confirmed this hypothesis in the 90s.

In this paper, we discuss the use of dedicated run-time support system, or, more specifically, of *dedicated communication systems*, as effective alternatives to support communication-intensive parallel applications in clusters of workstations. The research that subsided this discussion was carried out in the scope of the SNOW project [8], which aims at developing a high-performance, application-oriented parallel-programming environment for dedicated clusters. Actually, SNOW's run-time system comes from another project, namely EPOS, which takes on a repository of software components, an adaptive component framework, and a set of tools to build application-oriented operating systems on demand [9].

The remainder of this paper is structured as follows. Section 2 gives an overview of EPOS. Section 3 presents a redesign of EPOS communication system aimed at enhancing support for *network interface protocols* and describes the baseline architecture that supports these protocols. Section 4 elaborates on related works. Conclusions are presented in Section 5, along with the directions for future work.

2 An Overview of EPOS

EPOS, the *Embedded Parallel Operating System*, is a highly customizable operating system developed using state-of-the-art software engineering techniques. EPOS consists of a collection of reusable and adaptable software components and a set of tools that support parallel application developers in "plugging" these components into an adaptive framework to produce a variety of run-time systems, including complete operating systems. Being fruit of *Application-Oriented System Design* [7], method that covers the development of application-oriented operating systems from domain analysis to implementation, EPOS can be customized to match the requirements of particular parallel applications. EPOS components, or scenario-independent system abstractions as they are called, are grouped in families and kept independent of execution scenario by deploying aspect separation and other factorization techniques during the domain engineering process (figure 1 illustrates this process). EPOS components can be adapted to be reused in a variety of execution scenarios. Usability is largely improved by hiding the details of a family of abstraction behind an hypothetical interface, called the family's *inflated interface*, and delegating the selection of proper family members to automatic configuration tools.

An application written based on inflated interfaces can be submitted to a tool that scans it searching for references to the interfaces, thus rendering the features of each family that are necessary to support the application at run-time. This tool, the `analyzer`, outputs a specification of requirements in the form of partial component interface declarations, including methods, types and constants that were used by the application.

The primary specification produced by the analyzer is subsequently fed into a second tool, the `configurator`, that consults a build-up database to further refine the specification. This database holds information about each component in the repository, as well as dependencies and composition rules that are used by the configurator to build a dependency tree. Additionally, each component in the repository is tagged with a "cost" estimation, so that the configurator will chose the "cheapest" option whenever two or more components satisfy a dependency. The output of the configurator consists of a set

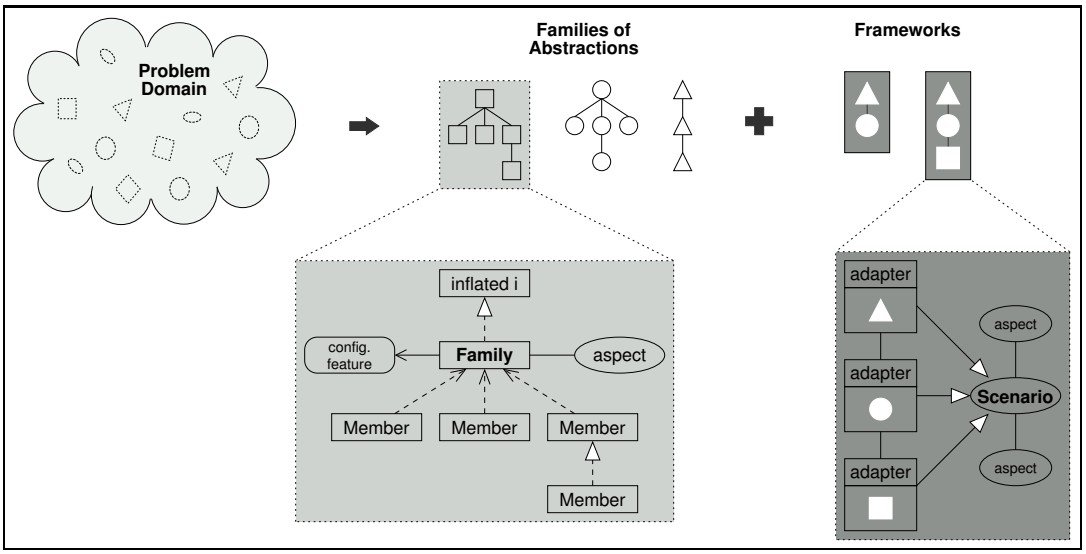

Figure 1: An overview of *Application-Oriented System Design*.

of keys that define the binding of inflated interfaces to abstractions and activate the scenario aspects and configurable features eventually identified as necessary to satisfy the constraints dictated by the target application or by the configured execution scenario.

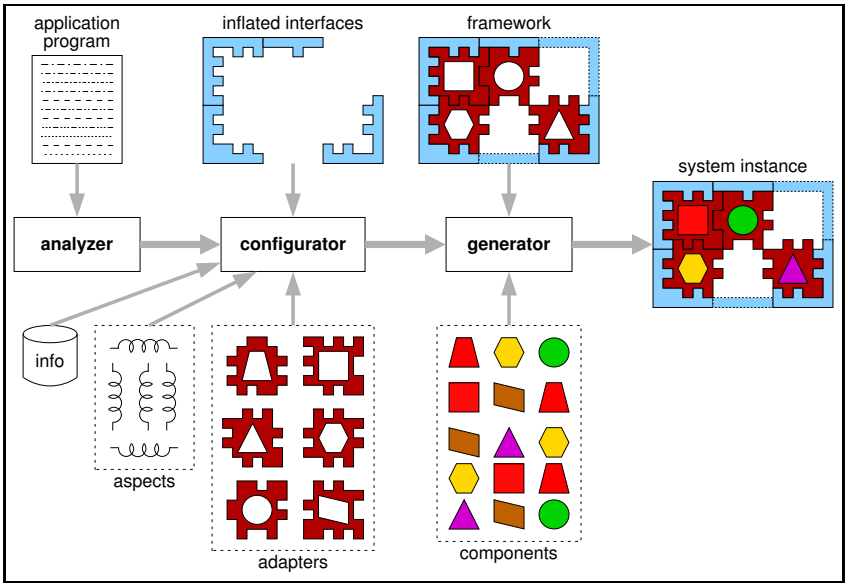

Figure 2: An overview of EPOS generation tools.

The last step in run-time systems generation process is accomplished by the `generator`. This tool translates the keys produced by the configurator into parameters for a statically meta programed component framework and triggers the compilation of a tailored system instance. Figure 2 brings an overview of the whole procedure.

3 EPOS Communication System

EPOS communication system is designed around three major families of abstractions: communicator, channel, and network. The `communicator` family encompasses communication end-points such as link, port, and mailbox, thus acting as the main interface between the communication system and appli-

cation programs[1]. The second family, `channel`, features communication protocols, so that application data fed into the communication system via a communicator gets delivered at the destination communicator accordingly. A channel implements a communication protocol that, according to the ISO/OSI-RM reference would be classified at level four (transport). The third family in EPOS communication system, `network`, is responsible for abstracting distinct network technologies through a common interface[2], thus keeping the communication system itself architecture-independent and allowing for flexible combinations of protocols and network architectures.

Figure 3: An overview of EPOS communication system.

Previous partial implementations of EPOS communication system for the *Myrinet* high-speed network architecture confirmed the lightness of its core, delivering unprecedented bandwidth and latency to parallel applications running on SNOW [10]. Nonetheless, EPOS communication system original designs makes it hard to split the implementation of *network interface protocols* [2] between the processors in the host machine and in the network adapter. Besides, it is very difficult to specify a single network interface protocol that is optimal for all parallel application, since different applications impose different traffic patterns on the underlying network. Instead of developing a single, highly complex, all encompassing protocol, it appears more feasible to construct an architecture that permits fine-grain selection and dynamic configuration of precisely specified *low-level lightweight protocol* mechanisms. In an application-oriented environment, this set of low-level protocols can be used to customize communication according to applications' needs.

EPOS design allows for the several network interface protocols that arise from the design decisions related to network features to be grouped into a software component with a defined interface, a *family*, that can be easily accessed by the communication system of the OS. EPOS' framework implements mechanisms for fine-grain selection of modules according to applications' needs. These same mechanisms can be used to select the low-level lightweight protocols that better satisfy the applications' communications requirements. Besides, an important step towards an efficient, application-oriented communication system for clusters is to better understand the relation between the design decisions in low-level communication software and the performance of high-level applications. Grouping the different low-level implementations of communication protocols in a component coupled with the communication system has the additional advantage of allowing an application to experiment with different communications schemes, collecting metrics in order to identify the best scheme for its needs. In addition to that, to structure communication in such a modular fashion enhances maintainability and extensibility.

3.1 Myrinet baseline architecture

The baseline communication architecture that supports the low-level lightweight protocols for the Myrinet networking technology must be simple and flexible enough not to hinder the design and implementation of specific protocols. The highest bandwidth and lowest latency possible are desired since complex protocol implementations will definitely affect both of them. User-level communication was the best answer from academia to the lack of efficient communication protocols for modern, high-performance networks. The baseline architecture described in this section follows the general concepts behind successful user-level communication systems for Myrinet. Figure 4 exhibits this architecture, highlighting the data flow during communication as well as host and NIC memory layout.

The NIC memory holds the six buffers that are used during communication. Send Ring and Receive Ring are circular buffers that hold the frames before they are accessed by the Network-DMA engine, the

[1]The component nature of EPOS enables individual elements of the communication system to be reused in isolation, even directly by applications. Therefore, the communicator is not the only visible interface in the communication system.

[2]Each member in the network family is allowed to extend this interface to account for advanced features.

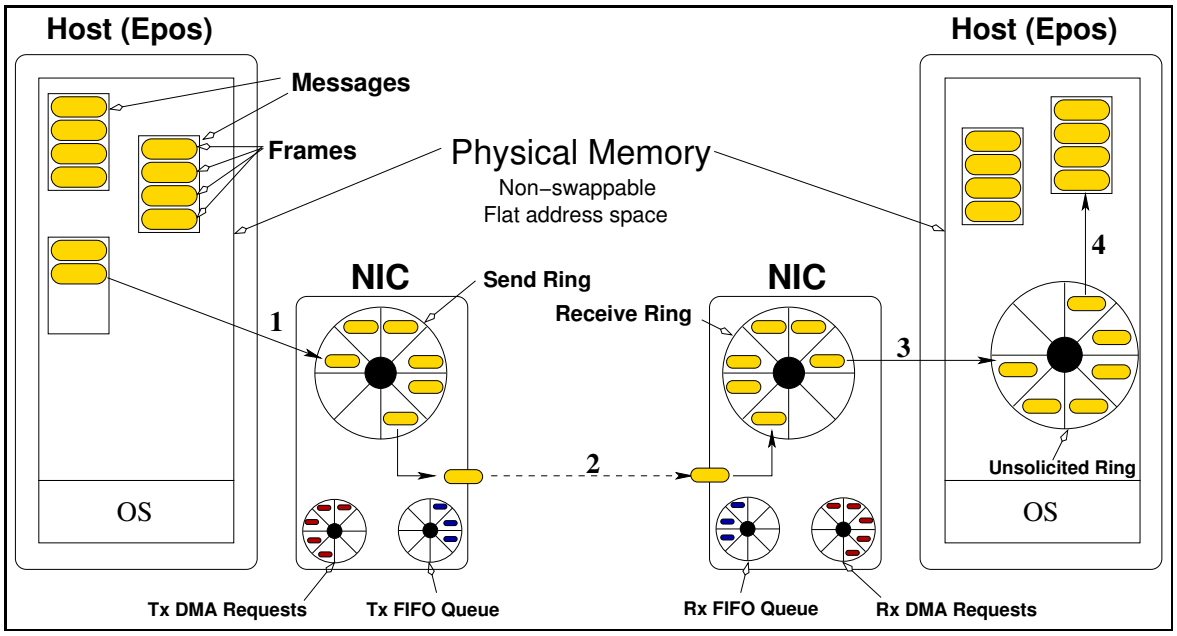

Figure 4: The Myrinet Family baseline architecture.

responsible for sending/receiving frames to/from the Myrinet network. Rx DMA Requests and Tx DMA Requests are circular chains of DMA control blocks, used by the Host-DMA engine for transferring frames between host and NIC memory. Rx FIFO Queue and Tx FIFO Queue are circular FIFO queues used by the host processor and LANai, Myrinet's network interface processor, to signal for each other the arrival of a new frame. The size of these buffers affects the communication performance and reliability and the choice of their sizes is influenced by the host's run-time system, memory space considerations, and hardware restrictions.

Much of the overhead observed in traditional protocol implementations is due to memory copies during communication. Some network technologies provide host-to-network and network-to-host data transfers, but Myrinet requires that all network traffic go through NIC memory. Therefore, at least three copies are required for each message: from host memory to NIC memory in the sending side, from NIC to NIC and from NIC memory to host memory in the receiving side. Write-combining, the DMA transfers start-up overhead and the fact that a DMA control block has to be written in NIC memory for each Host-DMA transaction make write PIO more efficient than DMA for small frames. The baseline architecture copies data from host to NIC using programmed I/O for small frames (less than 512 bytes) and Host-NIC DMA for large frames. Since reads over the I/O bus are much slower than writes, the baseline architecture uses DMA for all NIC-to-host transfers.

During communication, messages are split into frames of fixed size that are pushed into the communication pipeline. The frame size that minimizes the transmission time of the entire message in the pipelined data transfer is calculated [6] and the baseline architecture uses this value to fragment messages. Besides, the maximum frame size (MTU) is dynamically configurable. For each frame, the sender host processor uses write PIO to fill up an entry in the Rx DMA Requests (for large frames) or to copy (1) the frame directly to the Send Ring in NIC memory (for small frames). It then triggers a doorbell, creating a new entry in the Tx FIFO Queue and signaling for the LANai processor that a new frame must be sent. For large frames, the transmission of frames between host and NIC memory is carried out asynchronously by the Host/NIC DMA engine (1) and the frame is sent as soon as possible by LANai after the corresponding DMA finishes (2). Small frames are sent as soon as the doorbell is rung, since at this point the frame is already in NIC memory. A similar operation occurs in the receiving side: when a frame arrives from the network, LANai receives it and fills up an entry in the Rx DMA Requests DMA chain. The message is assembled asynchronously in the Unsolicited Ring circular buffer in host memory (3). The receiving side is responsible for copying the whole message from the Unsolicited Ring before

it is overwritten by other messages (4). Note that specific protocol implementations can avoid this last copy using rendezvous-style communication, where the receiver posts a receive request and provides a buffer before the message is sent, a credit scheme, where the sender is requested to have credits for the receiver before it sends a packet, or even some other technique, achieving the optimal three copies.

The host memory layout is defined by the operating system being used. Besides, the Myrinet NIC impose some constraints on the usage of its resources that must be addressed by the OS. The most critical one relates to the Host/NIC DMAs: the Host-DMA engine can only access contiguous pages pinned in physical memory. Most communication systems implementations for Myrinet address this issue by letting applications pin/unpin the pages that contain its buffers on-the-fly during communication or by using a pinned copy block. The problem with these approaches is that they add extra overhead since pinning/unpinning memory pages requires system calls, which implies in context saving and switching, and using a pinned copy block adds extra data copies in host memory. In EPOS, where swapping can be left out of the run-time system by mapping logical address spaces contiguously in physical memory, this issue does not affect the overall communication.

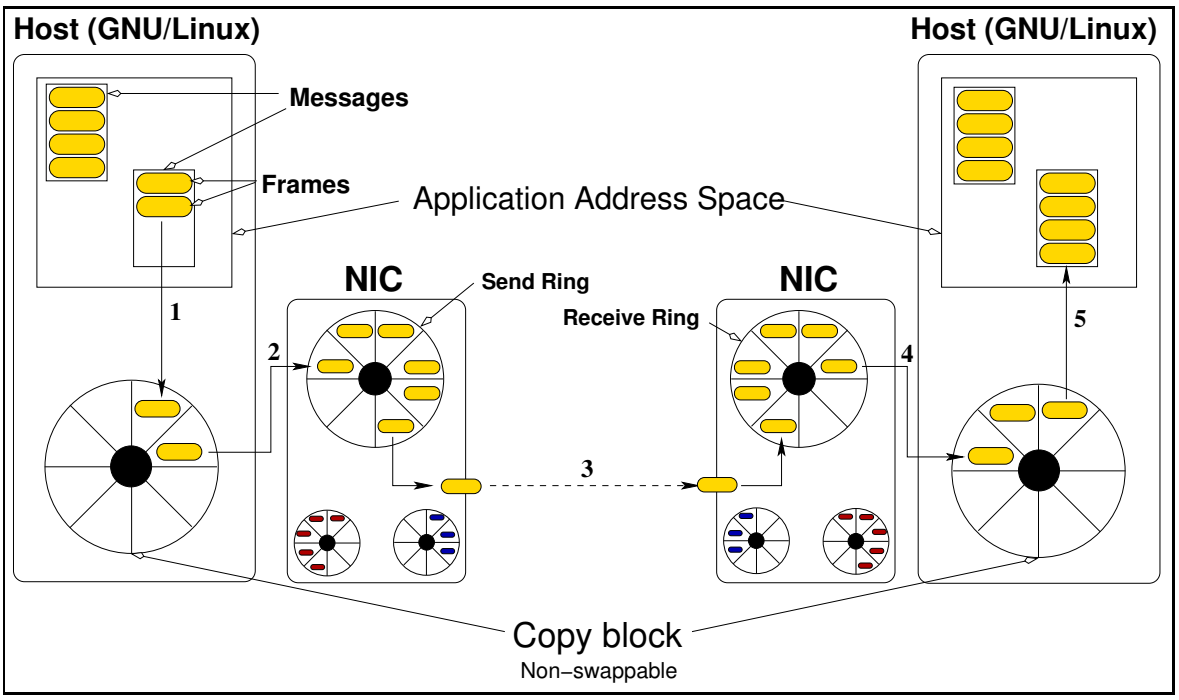

Figure 5: The Myrinet Family baseline architecture in a GNU/Linux host.

Figure 5 shows the memory layout and dynamic data flow of an implementation of the baseline architecture in a Myrinet GNU/Linux cluster. Issues such as address translation, kernel memory allocation and memory pinning had to be addressed in this implementation. Besides, a pinned copy block in kernel memory is used to interface Host/NIC DMA transfers, which adds one extra copy for each message in the sending side. Figure 6 exhibits a performance comparison between the GNU/Linux baseline architecture and the low-level driver GM (version 1.6.5), provided by the manufacturer of Myrinet. A round trip time test was performed in order to compare the two system's latency.

Many Myrinet protocols assume that the Myrinet network is reliable and, for that reason, no retransmission or time-out mechanism is needed. Indeed, the risk of a packet being lost or corrupted in a Myrinet network is so small that reliability mechanisms can be safely left out of the baseline architecture. Alternative implementations that assume unreliable network hardware and recover from lost, corrupted, and dropped frames by means of time-outs, retransmission, and hardware supported CRC checks are addressed by specific protocol implementations since different application domains may need different trade off between reliability and performance.

The presented architecture may drop frames because of insufficient buffer space. The baseline ar-

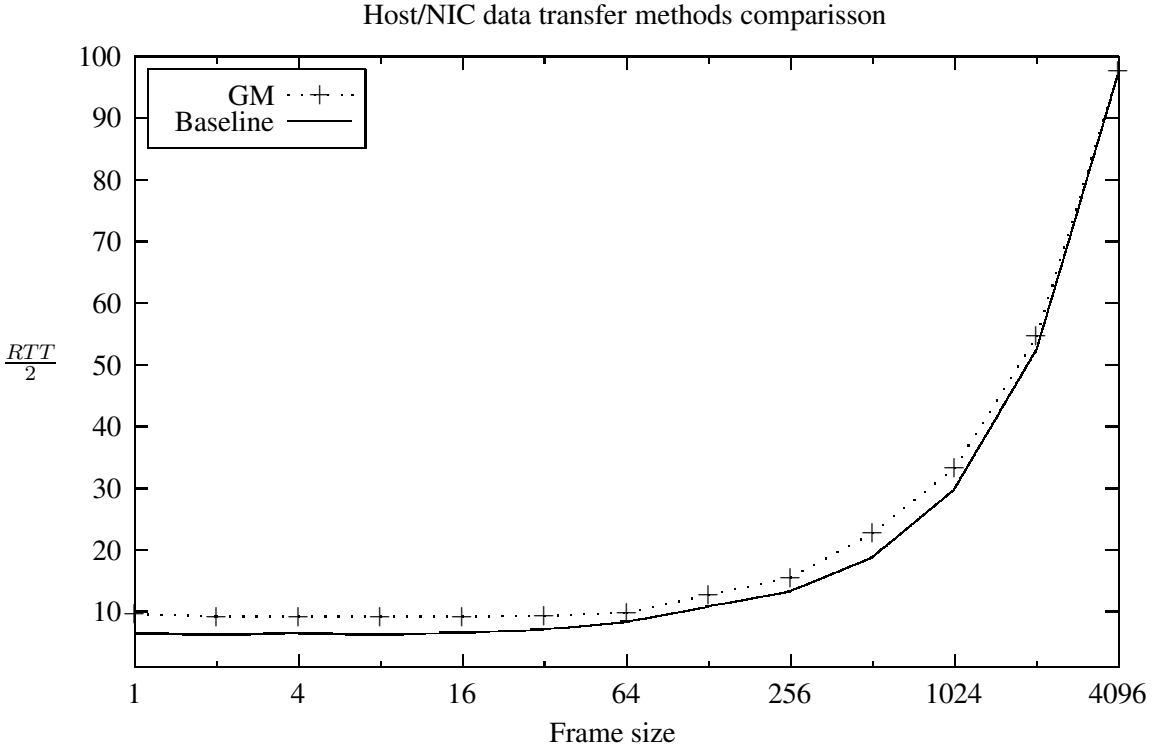

Figure 6: Comparison between the baseline architecture's and GM's latency (in microseconds) for different frame sizes (in bytes).

chitecture rests on a NIC hardware mechanism to partially solve this problem. Backpressure, Myrinet's hardware link-level flow control mechanism, is used to prevent overflow of network interface buffers, stalling the sender until the receiver is able to drain frames from the network. More sophisticated flow-control mechanisms must be provided by specific protocol implementations since specialized applications may only require limited flow-control from the network, performing some kind of control on their own.

3.2 Myrinet low-level lightweight protocols

While the baseline architecture is closely related to the underlying networking technology, low-level lightweight protocols are designed according to the communication requirements of specific classes of parallel applications. The lightweight protocols in the Myrinet family are divided into two categories: *Infrastructure* and *High-Performance* protocols.

Infrastructure protocols provide communication services that were left out of the baseline architecture: transparent multicasting, QoS, connection management, protection schemes, reliable delivery and flow control mechanisms, among others. In order to keep latencies low it would be desirable to efficiently execute the entire protocol stack, up to the transport layer, in hardware. Programmable network interfaces can be used to achieve that goal. LANai makes the design of communication protocols more flexible. Infrastructure protocols exploit the network processor to the maximum, using more elaborate Myrinet control programs in order to offload a broader range of communication tasks to the LANai. The communication performance is affected due to the trade-off between performance and MCP complexity but for some specific classes of applications this is a small price to pay for the communication services provided.

High-performance protocols deliver minimum latency and maximum bandwidth to the run-time system. They usually consist in minimal modifications in the baseline architecture that are required by

applications or in protocols that monitor the traffic patterns and dynamically modify the baseline architecture's customization points in order to address dynamic changes in application requirements.

4 Related Works

There are several communication systems implementations for Myrinet, such as AM, BIP, PM, and VMMC, to name a few. Although these communication systems share some common goals, performance being one of them, they have made very different decisions in both the communication model and implementation, consequently offering different levels of functionality and performance. From the several published comparison between these implementations one can conclude that there is no single best low-level communication protocol since the communication patterns of the whole run-time system (application and run-time support) influences the impact of the low-level implementation decisions of a given communication system on applications' performance. Besides, run-time system specifics greatly influences communication system implementations' functionality.

While the Myrinet communication systems mentioned above try to deliver a generic, all-purpose solution for low-level communication, the main goal of the presented research is customization of low-level communication software. The architecture we propose should be flexible enough to allow that a broad range of the implementation decisions behind each one of the several Myrinet communication systems be supported as a lightweight protocol.

Although our work has focused on Myrinet, there are some other networks for which the same concepts can be applied. Cluster interconnection technologies that are also implemented with a programmable NIC that can execute a variety of protocol tasks include DEC's Memory Channel, the interconnection network in the IBM SP series and Quadrics' QsNet.

5 Conclusions

The widespread of cluster systems brings up the necessity of improvements in the software environment used in cluster computing. Cluster system software must be redesigned to better exploit clusters' hardware resources and to keep up with applications' requirements. Parallel-programming environments need to be developed with both the cluster and applications efficiency in mind.

In this paper we outlined the design of an application-oriented communication sub-system based in low-level lightweight protocols for the Myrinet networking technology. The design decision related to these protocols' baseline communication architecture were discussed and a GNU/Linux implementation was presented. Experiments are being carried out to determine the best values for the architecture's customization points in different traffic pattern conditions.

In order to design an efficient, application-oriented communication system for clusters it is necessary to better understand the relation between the design decisions in low-level communication software and the performance of high-level applications. We intend to evaluate how different low-level communication schemes impact applications performance, using different low-level communication protocols and real parallel applications, since commonly used benchmarks fail in represent applications' communication patterns. The redesign of EPOS communication system was a first step towards that goal.

References

[1] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. User-level Interprocess Communication for Shared Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(2):175–198, May 1991.

[2] Raoul A. F. Bhoedjang, Tim Ruhl, and Henri E. Bal. User-level Network Interface Protocols. *IEEE Computer*, 31(11):53–60, November 1998.

[3] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of the Supercomputing Symposium*, pages 379–386, 1994.

[4] Roy H. Campbell, Nayeem Islam, and Peter Madany. Choices, Frameworks and Refinement. *Computing Systems*, 5(3):217–257, 1992.

[5] Jörg Cordsen. *Virtuell Gemeinsamer Speicher,*. PhD thesis, Technical University of Berlin, Berlin, Germany, 1996.

[6] Antonio Augusto Frohlich, Gilles Pokam Tientcheu, and Wolfgang Schroder-Preikschat. EPOS and Myrinet: Effective Communication Support for Parallel Applications Running on Clusters of Commodity Workstations. In *Proceedings of 8th International Conference on High Performance Computing and Networking*, pages 417–426, Amsterdam, The Netherlands, May 2000.

[7] Antônio Augusto Fröhlich. *Application-Oriented Operating Systems*. Number 17 in GMD Research Series. GMD - Forschungszentrum Informationstechnik, Sankt Augustin, August 2001.

[8] Antônio Augusto Fröhlich, Philippe Olivier Alexander Navaux, Sérgio Takeo Kofuji, and Wolfgang Schröder-Preikschat. Snow: a parallel programming environment for clusters of workstations. In *Proceedings of the 7th German-Brazilian Workshop on Information Technology*, Maria Farinha, Brazil, September 2000.

[9] Antônio Augusto Fröhlich and Wolfgang Schröder-Preikschat. High Performance Application-oriented Operating Systems – the EPOS Aproach. In *Proceedings of the 11th Symposium on Computer Architecture and High Performance Computing*, pages 3–9, Natal, Brazil, September 1999.

[10] Antônio Augusto Fröhlich and Wolfgang Schröder-Preikschat. On Component-Based Communication Systems for Clusters of Workstations. In *Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID 2001)*, pages 640–645, Brisbane, Australia, May 2001.

[11] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[12] H. Hellwagner, W. Karl, M. Leberecht, and H. Richter. SCI-Based Local-Area Shared-Memory Multiprocessor. In *Proceedings of the International Workshop on Advanced Parallel Processing Technologies - APPT'95*, Beijing, China, September 1995.

[13] IEEE Task Force on Cluster Computing. *Cluster Computing White Paper*, Mark Baker, editor, online edition, December 2000. [http://www.dcs.port.ac.uk/˜mab/tfcc/WhitePaper].

[14] Steven S. Lumetta, Alan M. Mainwaring, and David E. Culler. Multi-Protocol Active Messages on a Cluster of SMP's. In *Proceedings of Supercomputing'97*, Sao Jose, USA, November 1997.

[15] Loic Prylli and Bernard Tourancheau. BIP: a New Protocol Designed for High Performance Networking on Myrinet. In *Proceedings of the International Workshop on Personal Computer based Networks of Workstations*, Orlando, USA, April 1998.

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

[17] Hiroshi Tezuka, Atsushi Hori, Yutaka Ishikawa, and Mitsuhisa Sato. PM: An Operating System Coordinated High Performance Communication Library. In *High-Performance Computing and Networking*, volume 1225 of *Lecture Notes in Computer Science*, pages 708–717. Springer, April 1997.