A Customizable Component for Low-Level Communication Software

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
Email: {robert / guto}@lisha.ufsc.br
http://www.lisha.ufsc.br/~{robert / guto}

Abstract

This paper discusses the design of a customizable component that encapsulates a raw baseline communication architecture and a set of lightweight protocols. These protocols can act in specific points of the baseline architecture's algorithm in order tailor communication or provide new services according to the requirements of specific classes of applications. An implementation of the described component is currently being used by the application-oriented parallel programming environment SNOW*, which provides specially tailored run-time systems for parallel applications running on dedicated clusters.*

1. Introduction

Recent advances in network hardware technology have triggered a new wave of research in the communication software field [10, 12]. Modern high-speed networks invalidate some of the assumptions of well-established, widely used protocols, making it necessary to redesign the architecture of next generation communication software. In order to provide applications with communication performance close to the limits imposed by the network hardware, modern run-time systems must be able to effectively exploit features provided by the communication hardware infrastructure, such as DMA engines, programmable NICs and wormhole routing. Different research projects have already proved that the design decision related to low-level communication software greatly impact high-level applications performance, especially in cluster environments where performance depends on efficient communication between the computing nodes [18].

The concept that communication software must be tailored according to application requirements is currently the subject of active research [1, 15]. *High-Performance Computing* is one of the fields that can take more advantage of application-oriented communication software since parallel applications vary drastically, imposing different communication patterns on the network and demanding custom support from the run-time system. It seems reasonable to assume that having the chance to tailor low-level communication software according to applications' needs can lead to a significant improvement in the overall performance of I/O intensive parallel applications and parallel programming environments, whose performance is greatly affected by the effectiveness of the underlying communication infrastructure.

This paper discusses the design of a configurable component that can be tailored in order to match the communication requirements of specific classes of applications. This component encapsulates a baseline communication architecture and a set of lightweight protocols that can act in specific points of the baseline architecture's algorithm in order to tailor its communication mechanisms or provide new services. The idea behind designing a set of lightweight protocols instead of a single, monolithic one is that applications can improve their communication performance by using the most appropriate protocols for their needs.

EPOS, an application-oriented, component-based operating system for dedicated applications, is the core of the SNOW project [5], which aims at delivering a high-performance, application-oriented parallel-programming environment for dedicated clusters. The component described in this paper is used by EPOS in order to customize its communication system according to the requirements of the parallel applications running on top of SNOW. The remainder of this paper is structured as follows: section 2 presents an overview of the customizable component just mentioned focusing on the design of the metaprogrammed mechanisms used to perform protocols selection and configuration. Section 3 gives an overview of EPOS communication system and explains how the described component fits in EPOS framework. Section 4 presents conclusions along with the directions for fu-

ture work.

2. Low-level lightweight communication protocols

Modern communication systems usually feature one or more lightweight protocols instead of the classic, much less efficient layered communication architecture. Lightweight protocols consist in a set of low-level communication mechanisms embedding a communication protocol and can be implemented as part of the host runtime system, hardwired in the network hardware or as a combination of host software and network interface firmware. Lightweight protocols can implement different communication services, such as reliable delivery, flow-control and multicasting, closer to the network hardware, which can be more efficient since special features of the communication hardware infrastructure can be used to improve performance and some of the tasks involved in providing these services can be offloaded to the network interface card.

It is probably impossible to design a single communication protocol that is optimal for all applications since different applications impose different traffic patterns on the underlying network and demand specific communication services. Besides, several different low-level communication protocols arise from the many different ways network hardware features can be used, and the fact that modern NICs can be programmed only expands the universe of possible protocol implementations. Instead of developing a single, highly complex, all encompassing protocol, it appears more feasible to construct a module that allows fine-grain selection and configuration of previously implemented lightweight protocols. This paradigm offers a number of potential advantages, including the ability to develop and deploy new network protocols and services quickly and allowing applications to experiment with different communication protocols, collecting metrics in order to identify the best one for their needs. Moreover, to structure communication software in such a modular fashion enhances maintainability and extensibility, and designing a set of low-level communication protocols instead of a monolithic one promotes flexibility since new protocols can be implemented specifically for a given application in order to satisfy its particular communication requirements.

In order to improve reusability and ease protocols implementation, we have chosen to design lightweight protocols in a strategy pattern [7] fashion: protocols will rely on a *baseline communication architecture*, a basic, optimized implementation of a low-level message passing API. Each active protocol will have the chance to act in specific *pointcuts* of the baseline architecture's communication algorithm in order to provide the services it implements. One of the constraints of this design decisions is the fact that a baseline communication architecture and the protocols it accepts become very tightly coupled. A good way to diminish that penalty is to design a baseline architecture that is simple and flexible enough not to hinder the design and implementation of specific protocols. Besides, the highest bandwidth and lowest latency possible are necessary since complex protocol implementations will definitely affect both of them.

A baseline communication architecture and the lightweight protocols is accepts must be grouped in a component with a defined interface, low-level enough so that different arrangements of runtime systems can be layered on top of it in an efficient way but also complete enough so that applications can use the component directly. In our approach, the baseline architecture dictates the component's interface as well as the methods that must be provided by the protocols it accepts. All lightweight protocols provide an uniform, inline interface that declares only the methods required by the baseline architecture.

Customization is an important requirement and it is addressed through a configuration repository that is accessible by all sub-components involved in communication. Customization points include the number and size of communication buffers in host and NIC memory, maximum transfer unit, low-level communication protocols to be used and so on. Protocol implementations that allow dynamic changes in the active protocols, MTU and other customization points can be combined with some kind of profiling provided by the run-time system to dynamically tailor communication according to application needs. This mechanism can also be extended to provide an active network architecture implemented as a lightweight protocol that is able to encapsulate other protocols or protocol activation flags in a frame that is then injected into the network. In the receiving side, this same protocol would evaluate the frames received from the network and use the encapsulated protocols or protocol activation flags in order to dynamically change the communication behavior.

It is desirable that lightweight protocols can be combined together at will, providing the complete set of communication services the combined protocols implement. The active lightweight protocols and the baseline architecture that supports them are combined together at compile-time using the metaprogrammed support described in next section.

2.1. Automated generation

Metaprogramming [3] is the key technology for developing adaptable systems and automating the assembly of components. Template metaprogramming is a form of metaprogramming limited to compile time: the C++ template mechanism allows us to write code that is executed by the C++ compiler during compilation. The fact that C++ static code

can be used to manipulate dynamic code is the basic principle of the component described in this section. In [17] the authors present an implementation of their broader view of aspects using metaprogramming techniques similar to the ones described in this section and the *Boost* metaprogramming library, an extensible compile-time framework of algorithms, sequences and metafunction classes that can be considered the equivalent of a large subset of the STL. In fact, the interaction between the baseline communication architecture and the lightweight protocols was inspired by some of the ideas of aspect-oriented programming and that is the reason the term *point-cut*, used to identify the control points at which aspect code fragments are to be inserted in aspect-oriented programs, was chosen to identify the points of interaction between baseline architecture and lightweight protocols.

At compile-time, the metaclass *protocol generator* selects and combines the active protocols taking as input the configuration knowledge and providing as output an abstract data-type that groups all the active lightweight communication protocols and the corresponding baseline communication architecture in a single component. It first goes through the flags in the configuration repository that trigger the activation of specific protocols. The corresponding protocol classes are then queued in a static metaprogrammed linked list in a predefined order and *dummy protocols* are used to fill the spot of the protocols that were not selected. A dummy protocol provides the interface associated with the baseline architecture in use, but all of its methods are empty blocks declared as inline in order to allow that the optimization process of the C++ compiler remove all the calls for any of its methods in the binary component.

The linked-list generated is used to parametrize a *composite protocol*, a metaclass that provides a common interface for a set of protocols. For each method in the uniform interface of the low-level protocols, the composite protocol uses a static metaprogrammed loop to run through the elements in the linked-list generating code for the corresponding method call of each one of the protocols on the list, in the predefined order. Method calls for dummy protocols, used to replace the protocols that were not selected, are wiped-off in a later optimization stage of the compilation process. The whole process of static selection and configuration of low-level lightweight communication protocols is summarized in Figure 1.

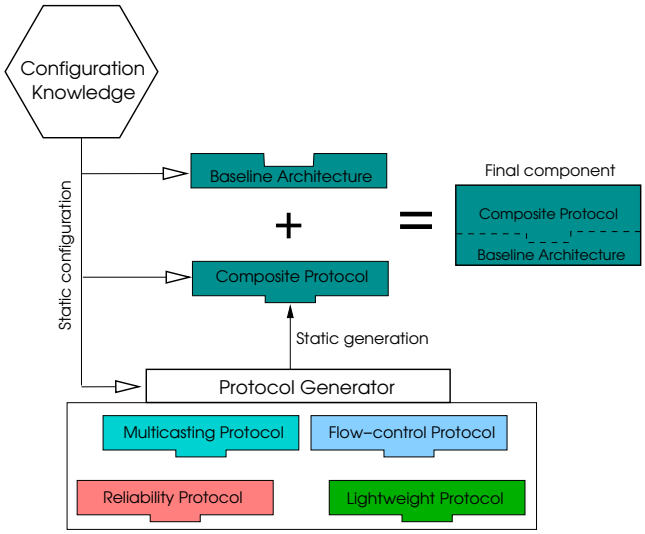

Figure 1. Configuration and generation of low-level lightweight protocols.

2.2. Protocols

In this section we will discuss the implementation of three communication services required by several applications that rely on a low-level communication interface as lightweight protocols: multicasting/broadcasting, reliability and flow-control. One of the advantages in implementing such services as low-level protocols is the fact that the closer to the hardware these communication services are implemented, taking advantage of modern network hardware features, the more efficient the implementations are.

In order to achieve this, we must define a low-level baseline communication architecture for an existing network technology. Therefore, a deep understanding of the network technology in question, from the network interface card architecture to the routing mechanisms involved, becomes necessary. Myricom's Myrinet is a switched, gigabit-per-second system area network (SAN). The ANSI standard Myrinet-on-VME Protocol Specification [19] defines packets of variable length that are wormhole-routed through a network of highly reliable links and crossbar switches. The Myrinet NIC is a programmable communication device, equipped with an instruction-interpreting on-board RISC processor, called LANai, a set of DMA engines and on-board fast SRAM. Figure 2 shows the architecture of a node in a Myrinet SAN. Each node in the SAN features a Myrinet NIC connected to its I/O bus, a typical organization for commodity hardware.

Myrinet requires that all incoming and outgoing packets go through NIC memory during communication, which makes the relatively small amount of SRAM a precious resource. The on-board memory holds all code and data for the LANai processor, the temporary buffers for the incoming and outgoing packets and other data used during communication. A host can access its network interface's memory using programmed I/O (PIO) and both the host and the NIC can use DMA to access data in each other's memory.

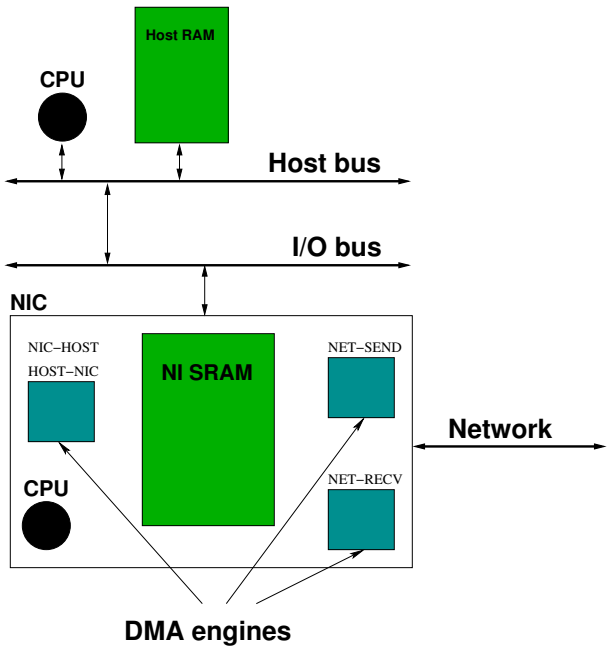

Figure 2. Architecture of a node in a Myrinet SAN.

The NIC allows at most two memory accesses per clock cycle, assigned based on the following priority order (highest to lowest): I/O bus, net-recv DMA, net-send DMA, and the LANai processor.

LANai gives protocol designers a great flexibility since part of the communication can be offloaded from the host CPU to the NIC processor. Besides, the NIC processor can be programmed to provide a variety of special communications services closer to the hardware. LANai is much slower than the host CPU, which yields a trade-off between performance and complexity of the communication tasks executed by it. Myrinet control programs (MCP) must be carefully designed since adding just a few instructions to the critical path of the MCP affects communication performance.

Three DMA engines are provided by the Myrinet NIC. They are responsible for injecting a frame into the network (Net-Send), for consuming a frame off of the network (Net-Recv) and for data transfers between host and NIC (Host/NIC). These DMA engines are designed to work fully in parallel in order to improve communication performance. The number of DMA operations that can be executed concurrently is limited by the restriction in the number of memory access per cycle imposed by the NIC. Myrinet networks provide other features that can be used in many different ways by communication protocols:

DMA queuing: Host/NIC data transfer requests can be queued together and executed asynchronously by the Host/NIC DMA engine. The engine uses up to four chains of DMA control blocks stored in LANai memory to trigger DMA-mastering operations. Each one of these chains can be activated by the host or the LANai. When a chain is activated, the Host/NIC DMA engine executes a host/NIC data transfer for each block in the chain until it finds a terminal block. The DMA engine processes chains in priority order. For example, if the DMA engine is executing the DMA operation described in a control block in the lowest priority chain when a new control block is queued in a higher priority chain, the DMA engine will finish the DMA operation being executed and move to that higher priority chain. A control block can be configured to make the Host/NIC DMA engine signal when it completes the DMA operation for that block.

Hardware doorbell mechanism: The Myrinet NIC implements a mechanism that allows the host to write anywhere in a specified region of the NIC I/O address space, the doorbell region, and to have that data stored into a FIFO queue in the LANai memory. The I/O address accessed by the host is also written in the FIFO queue.

Backpressure flow control: Hop-by-hop flow control scheme employed by Myrinet that stalls the sending NIC if the receiver is not available. One drawback of backpressure is the possibility of creating deadlock situations. There is a time limit in the backpressure mechanism that can be dynamically configured by software and is used to prevent deadlocks: if the receiver does not drain the network within the specified time limit the network resets the receiving host NIC or truncates the packet.

Wormhole Routing: In wormhole routing networks, each intermediate switch forwards a packet to the desired output port as soon as it enters the switch, without waiting for the entire packet to be assembled.

The high-performance Myrinet baseline architecture described in [14] follows the general concepts behind successful user-level communication systems for Myrinet. Figure 3 exhibits this architecture's dynamic data flow and host and NIC memory layout. Lightweight protocols will have the chance to define actions to be executed by the baseline architecture in the point-cuts defined for it, customizing the communication algorithm according to the services they will provide.

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 engines. 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 to signal for each other the arrival of new frames. The size of these buffers affects the communica-

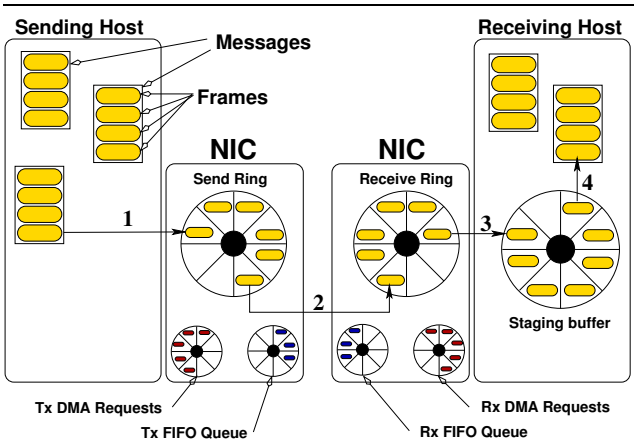

Figure 3. The Myrinet baseline architecture.

tion performance and reliability and they are statically configured at compile time according to the information in the configuration repository. The maximum transmission unit (MTU) is also read from the configuration repository but it can be changed dynamically by specific protocols in order to maximize communication performance with techniques such as the ones described in [6].

During communication, messages are split into frames of fixed size that receive a header with information related with the baseline architecture and the active protocols. These frames are pushed into the communication pipeline by the sender host processor that 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 (1) is carried out asynchronously by the Host/NIC DMA engine and the frame is sent (2) as soon as possible by LANai after the corresponding DMA finishes. 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 chain. The message is assembled asynchronously in a staging buffer in host memory (3). The receiving side is responsible for copying the whole message from the staging buffer before it is overwritten by other messages (4).

Much of the overhead observed in traditional protocol implementations is due to memory copies during communication. In Myrinet 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. In order to simplify the baseline architecture's design, a staging buffer was used to temporarily hold the frames received from the network until the receiving host is able to handle them. This adds some overhead but specific protocol implementations can avoid this extra copy acting in the point-cuts of the baseline architecture in order to provide services such as 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 another technique, achieving the optimal three copies.

The baseline architecture and many other Myrinet protocols assume that the Myrinet network is reliable and, for that reason, no re-transmission or time-out mechanism is needed. 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 can be addressed by specific protocols since different application domains may need different trade off between reliability and performance. An easy way to implement retransmission on top of the communication algorithm previously outlined would be designing a protocol that acts in the point-cut defined right after the Send operation making the baseline architecture wait for an acknowledgment from the receiving host for a previously configured amount of time. A retransmission would occur if the sending host didn't get the acknowledgment before timeout.

Backpressure, Myrinet's hardware link-level flow control mechanism, is used by the baseline architecture in order 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. Multicast and broadcast are desirable since they are fundamental components of collective communication operations but the described architecture supports only point-to-point messages. Lightweight protocols that provide these features could be easily implemented on top of point-to-point messages or using more efficient techniques [8].

Finally, the proposed baseline architecture provides no protection since there is a large number of parallel applications running on dedicated environments. Protection could be added using a VIA-like approach such as the one described in [9]. Protocols could be used to implement this mechanisms but that would require small changes in the described baseline architecture's design since the Myrinet's hardware doorbell would have to be used in order to implement the virtual interfaces.

3. EPOS Communication System

Configurable operating systems [11] [16] [2] are a recurrent theme in the operating system field. EPOS [4] is a statically configurable, application-oriented, component-based operating system for dedicated applications. EPOS is the core of SNOW, a dedicated parallel programming system for computational science applications that features a thin-layer implementation of the MPI standard [13].

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 adaptable framework in order to yield different arrangements of run-time systems. 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 and 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.

Components for low-level communication that follow the design described in the previous section are currently being used by EPOS in order to provide tailored communication support for SNOW and all the parallel applications running on top of it. 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 application programs. However, Communicators are not the only visible interface in the communication system since the component nature of EPOS enables individual elements of the communication system to be reused in isolation, even directly by applications. The second family of abstractions, Channel, features communication protocols, so that data fed into the communication system via a communicator gets delivered at the destination communicator accordingly. Examples of channels are Datagram and Stream. Network, the third family in EPOS communication system, is responsible for abstracting distinct network technologies through a common interface, thus keeping the communication system itself architecture-independent and allowing for flexible combinations of protocols and network architectures. Each member in the network family is allowed to extend the family's interface to account for advanced features.

The EPOS abstraction for the Myrinet network technology was implemented following the design outlined in the previous section. The high-performance baseline architecture outlined before is being used and a broad set of lightweight protocols is currently being implemented so SNOW can attend to the low-level communication needs of specific classes of parallel applications. The component was easily integrated into EPOS communication system since the baseline architecture defined for it was designed aimed at providing the interface required by the different Channel abstractions.

In EPOS, configuration knowledge is encapsulated in configuration repositories implemented as traits templates. There is one of these configuration repositories for each one of EPOS components. A given configuration repository will hold the values for the *Configurable Features* associated with the corresponding component. Configurable Features are being used by the metaclass Protocol Generator in order to trigger the selection of the set of low-level lightweight protocols that better satisfy parallel applications' communications requirements.

4. Conclusion

An important step towards an efficient, application-oriented communication system is to better understand the relation between the design decisions in low-level communication software and the performance of high-level applications. Several research projects discuss the performance of different implementations of low-level communication systems for a given network technology. The main problem in comparing these different communication systems is that all of them have made very different decisions in both the communication model and implementation and provide different interfaces. Using the component previously described combined with the application-oriented runtime system of SNOW will help us evaluate the impact that customization of low-level communication software has in the performance of parallel applications running on dedicated-clusters.

The architecture we propose should be flexible enough to allow that a broad range of design decisions related with low-level communication software for a given network platform be supported as lightweight protocols. The proposed lightweight communication protocols, along with the application-oriented run-time system provided by SNOW, will be used in order to evaluate how different low-level communication schemes impact on parallel applications' performance and the defined interface provided by the final component will ease performance comparison among different communication strategies.

References

[1] S. Bhattacharjee, K. L. Calvert, and E. W. Zegura. Reasoning about active network protocols. In *ICNP '98: Proceed-*

ings of the Sixth International Conference on Network Protocols, page 31. IEEE Computer Society, 1998.

[2] R. H. Campbell, N. Islam, and P. Madany. Choices, frameworks and refinement. In *Computing Systems*, pages 217–257, 1992.

[3] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., 2000.

[4] A. A. Frohlich. *Application-Oriented Operating Systems*. GMD Research Series. GMD - Forschungszentrum Informationstechnik, Sankt Augustin, Aug. 2001.

[5] A. A. Frohlich, P. O. A. Navaux, S. T. Kofuji, and W. Schroder-Preikschat. Snow: a parallel programming environment for clusters of workstations. In *Proceedings of the 7th German-Brazilian Workshop on Information Technology*, Maria Farinha, Brazil, Sept. 2000.

[6] A. A. Frohlich, G. P. Tientcheu, and W. 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] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[8] M. Gerla, P. Palnati, and S. Walton. Multicasting protocols for high-speed, wormhole-routing local area networks. *SIGCOMM*, pages 184–193, 1996.

[9] M.-S. Lee, J.-L. Yu, and S.-R. Maeng. Pipelined implementation of the virtual interface architecture on myrinet. 2001.

[10] S. S. Lumetta, A. M. Mainwaring, and D. E. Culler. Multi-protocol active messages on a cluster of smp's. In *Proceedings of Supercomputing'97*, San Jose, CA, Nov. 1997.

[11] A. B. Maccabe, P. G. Bridges, R. Brightwell, R. Riesen, and T. Hudson. Highly configurable operating systems for ultrascale systems. In *Proceedings of the First International Workshop on Operating Systems, Programming Environments and Management Tools for High-Performance Computing on Clusters (COSET-1)*, pages 33–40, Saint-Malo, France, June 2004.

[12] L. Prylli and B. Tourancheau. Bip: A new protocol designed for high performance networking on myrinet. In *IPPS/SPDP Workshops*, volume 1388 of *Lecture Notes in Computer Science*, pages 475–485, Orlando, FL, Mar. 1998.

[13] A. L. G. Sanches and A. A. Frohlich. Epos-mpi: a highly configurable run-time system for parallel applications. In *Proc. SBAC 2004*, Foz do Iguacu, Brazil, Oct. 2004.

[14] T. R. C. Santos and A. A. Frohlich. An application-oriented communication system for clusters of workstations. In *Proceedings of the First International Workshop on Operating Systems, Programming Environments and Management Tools for High-Performance Computing on Clusters (COSET-1)*, pages 15–24, Saint-Malo, France, June 2004.

[15] D. Schmidt, D. Box, and T. Suda. Adaptive: A flexible and adaptive transport system architecture to support lightweight protocols for multimedia applications on high-performance networks. 1992.

[16] W. Schroder-Preikschat. *The Logical Design of Parallel Operating Systems*. Prentice-Hall, Inc., Englewood Cliffs, USA, 1994.

[17] S. Sunder and D. R. Musser. A metaprogramming approach to aspect oriented programming in c++. Mar. 2001.

[18] K. Verstoep, R. A. F. Bhoedjang, T. Ruhl, H. E. Bal, and R. F. H. Hofman. Cluster communication protocols for parallel-programming systems. Technical Report 3, Vrije Universiteit, Amsterdam, The Netherlands, 1998.

[19] VITA Standards Organization. *Myrinet-on-VME Protocol Specification*, 1998. ANSI/VITA 26-1998.