Proceedings of the IASTED International Conference
Parallel and Distributed Computing and Systems
November 3-6, 1999 in Cambridge Massachusetts, USA

Protocols Aboard Network Interface Cards

C. Beauduy R. Bettati

{*cbeauduy, bettati*}*@cs.tamu.edu*
Department of Computer Science
Texas A&M University
College Station, TX

Abstract

Traditional host-resident protocol stacks are burdensome and often fail to keep pace with today's high-speed network data movement. With the PANIC system (Protocols Aboard Network Interface Cards), we explore shifting all or part of the protocol processing to the network interface card (NIC). Our system allows us to deploy user-level protocols, or portions thereof, across a collection of machines. We have implemented a first prototype of PANIC over Myrinet, and experiments show the feasibility and efficiency of this approach.

Key Words: Network interface, protocol composition, networks-of-workstations, Myrinet.

1 Introduction

As the performance provided by networking technologies dramatically increases, solutions for high-performance fine-grained distributed computing start to emerge. Computing based on clusters, or on networks of workstations, greatly increases the performance of a variety of applications at low costs [1, 2, 3, 4].

The performance of such clusters relies heavily on low communication latency. For example, applications on clusters frequently rely on reliable multicast protocols to disseminate the state of the computation and to manage the state of the system. These protocols typically involve several rounds of message exchanges, and so are very sensitive to communication latency. For example, it has been shown that the effect of latency on the performance of Microsoft's Cluster Server is severe enough that, without low-latency communication, its scalability is limited to 8 nodes [5].

In the past, communication latency was mainly due to insufficient network bandwidth and excessive protocol overhead in the host. With the increases in available network bandwidth and host computing power, latency in current systems is mostly caused at the network-host interface. In particular, context switches between kernel and user-level applications are burdensome.

A number of user-level network interface protocols have been proposed [6, 7, 8, 9], which eliminate the kernel from the critical path between the application and the network interface card. While eliminating the kernel protocol stack from the message path on the host greatly reduces protocol processing latency, significant sources thereof remain. User-level threads on the receiving host still suffer from scheduling latencies and context switch overhead. If thread invocation is part of a receive-reply loop, the sender may be unduly delayed by the latencies first at the receiver and then back at the sender. Thus, many user-level protocols that rely on any form of request-reply cycle between sender and receiver incur such delays at the interface between network interface card and user-level threads. Such protocols are very common in distributed computing, for example, reliable transmission, directory management and token-based mechanisms in general. Very often, the amount of computation at the receiver is trivial, typically involving some form of simple table lookup.

In this work, we propose to reduce latencies for user-level protocol processing by moving portions of the protocol related computation into the network. The network infrastructure should support distributed applications by efficiently performing latency-critical portions of the protocol processing rather than placing the burden on the kernels or user-level applications. As the protocol requirements of distributed applications vary, the network support to such applications should be configurable. Components of protocol stacks should therefore dynamically be deployable into the network during application startup. The resulting paradigm is that of a *Network-Level Application Interface* for distributed applications: components within the network can be programmed at user-level to perform protocol-related computation on behalf of distributed user programs.

We investigate the feasibility of this approach in the project PANIC (Protocols Aboard Network Interface Cards), where we focus on network-level processing of user-level protocols at the boundary of the network. That is, the user-level protocol processing happens either in the host or in the network interface card (NIC). While NIC-level processing avoids the context-switch overheads in the host, it has other advantages as well: Many query or dissemination protocols (for example, directory look-ups or invalidation requests)

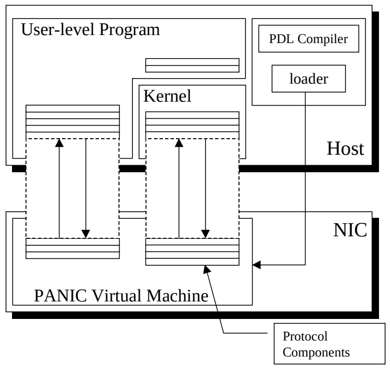

Figure 1: Protocol Stack Split between Host and NIC

rely on multicast communication schemes. These protocols cause unnecessary processing and interrupts at host level. Only one host in the multicast group may have the directory entry to be returned or invalidated, while all incur the overhead of passing the message to the application layer. Processing such requests within the network (that is, processing the directory management at the network level) not only significantly reduces the latency of directory look-ups, but reduces the number of context switches on the host as well, as requests are effectively handled at network level before a host interrupt is generated.

Similarly, authentication and authorization checks for incoming messages in a distributed security framework can be done at network level, that is, before the message reaches the host. In this way, hosts can be effectively shielded from many forms of denial-of-service attacks.

2 Abstraction

Traditionally, NICs contain some amount of protocol processing in their firmware. Additional protocol components can therefore easily be embedded. The drawback of this method is that the NIC control program must be recompiled and re-installed whenever a significant change in protocol behavior is desired, however.

The PANIC approach focuses on greater versatility. To achieve this, we incorporate a virtual machine (VM) for protocol processing into the NIC firmware. Protocol stacks, or portions thereof, can be dynamically deployed or recalled, or portions dynamically swapped at runtime. These operations can happen either on the local NIC or across the network on several NICs in an orchestrated fashion. A programming environment on the host allows us to define protocol components in a high-level language, PDL, which we specifically designed to support protocol processing. A compiler transforms PDL programs into loadable modules that are then interpreted by the virtual machine on the NIC. Protocol stacks are defined either as single PDL programs or as collections of inter-operating PDL programs.

PANIC is specifically designed to augment the existing protocol processing on the card. In this spirit, it does not interfere with existing packet delivery mechanisms. At no point do the operations on the VM affect the execution of the remainder of the NIC control program. The structured communication between PANIC programs allows them to form a dynamic protocol stack, or portions thereof, on the card. The complete protocol stack may reside entirely in the host, entirely in the card, or be split between the host and card as illustrated in Figure 1. The designer chooses how to best distribute the burden of protocol processing. For example, in an authentication subsystem, the table lookups and simple hash operations for incoming requests could be easily performed within the NIC, while more expensive encryption operations would remain on the host. Similarly, preprocessing of outgoing video data in a distributed multimedia system would be done on the host, while the shaping and policing of the outgoing traffic would be performed on the NIC.

3 Realization

As platform for implementation of PANIC we have chosen Myrinet. Myrinet is a good candidate because of its readily modifiable, modular firmware, real-time clock, very low latency, and high data transfer rate (1.2 Gbit/sec). Myrinet's high performance is particular interesting in this context, as it exposes the limitations of the processing messages on the host particularly well. Sustained transfer rates are constrained by the inability of the host to process data at these speeds, and the low latencies provided by the network are countered by the protocol processing overhead on the host.

3.1 The PANIC Virtual Machine

In its current realization, PANIC executes protocol components on the NIC within a virtual machine (VM). The first-generation VM is a simple stack machine that interprets code generated by the PDL compiler on the host. It has two main parts: a loader to manage the memory used to store PANIC programs, and an interpreter to execute those programs.

The VM is targeted towards simplicity rather than performance. Memory for PANIC programs is divided into instruction and data sections, both implemented as global arrays in the NIC SRAM. With few exceptions, memory in PANIC programs is static, and as such is allocated at load-time. The VM instruction set relies on relative addressing, making the code relocatable. Similarly, memory references are resolved relative to the beginning of the memory section allocated to the PANIC program.

We chose to develop a custom virtual machine rather than use a Java or other off-the-shelf VM for two reasons. First, the size of a minimal Java VM prohibits its integration onto our memory-challenged (256 kBytes) Myrinet NICs. Second, other VMs support instructions that have little or no use for our purpose, for example, string operations. Our instruction set is specifically tailored for use in the NIC environment. Special instructions support, for example, send and receive operations to and from the network, and host/card synchronization.

3.2 PANIC VM Interface to the Network

In order to quickly direct incoming packets for processing by the PANIC VM, they must be readily identifiable. In the Myrinet implementation we take advantage of a tagging mechanism used by the control program on the NIC to distinguish incoming control messages from data messages. We designate a new packet type, called PANIC packet, that coexists with other Myrinet packet types.

To cause minimal intrusion to normal packet processing on the host, we let PANIC packets be dispatched in the same fashion as all other incoming packets. Normal Myrinet behavior is unaffected except for the delay of executing the PANIC programs. For complex operations, the work is spread over two or more PANIC components, which each comprises a portion of the protocol stack. In this fashion, PDL programs temporarily relinquish control to the Myrinet firmware, and thus allow for some amount of multiprogramming on the card.

3.3 Host-PANIC Virtual Machine Interface

In order for applications on the host and their programs within PANIC VMs to effectively interact, we provide a shared memory space between host and PANIC VM and mechanisms for efficient synchronization between processes and threads on the host and programs on the PANIC VM. In addition, an interface is in place to control the loading and activation of PANIC programs.

Program Management: A library and suite of tools to load, unload, run, and send messages to PANIC programs is supplied. These operations can control programs locally or on remote PANIC VMs and currently use programmed I/O rather than DMA to transfer data because the program messages typically are small.

Memory Management: We provide a shared memory space between the PANIC VM and user programs within a DMA-able memory segment allocated by the device driver on the host. Applications acquire segments of this memory through `malloc`/`attach`-style calls, which reserve segments for use. The application and the PANIC VM attach to the same memory segment by sharing its segment identifier. The shared memory paradigm is not fully supported here: while user programs access memory directly, the PANIC VM has to access it through DMA operations. When applications exit, they free the memory to restore it for general use.

Synchronization: Synchronization between host and NIC is based on a lock/flag model. The host may set locks and clear flags; the NIC may clear locks and set flags. The locking structure is in Shared Memory and may be protected with mutexes from the host side. There are no interrupts involved. An application is expected to create a lightweight process to periodically check the flag that may be set by a PANIC program. Similarly, a PANIC program may check for a lock set by an application and take some appropriate action.

```
PROGRAM PING: # -- This program initiates, awaits reply from remote program.

VAR   timesent[100], timerec[100], i, msg[1024];  # -- Variable declarations
TRIGGERS 10;                                      # -- Trigger declarations

BEGIN # -- Program Initialization
    i = 0;
    timesent[i] = CLOCK;
    SEND msg[0] LEN 64 AS 12 TO 415 AT 0:0:0:96:221:127:255:140;
    AWAIT;  # -- Wait for first message to come in
END;

MESSAGE 10:   # -- Trigger block for trigger 10
    BEGIN
        timerec[i] = CLOCK; i = i + 1;
        if i < 100 THEN BEGIN
            Timesent[i] = CLOCK;
            SEND msg[0] LEN 64 AS 12 TO 415 AT 0:0:0:96:221:127:255:140;
            AWAIT;  # -- Wait for the next trigger to arrive
            # -- We never reach this point
        END;
        i = 0; # -- Reach here when i >= 100
        # -- Send timestamp information to buffer for reading by host
        WHILE i < 100 DO BEGIN
            DISPLAY timesent[i]; DISPLAY timerec[i]; i = i + 1;
        END;
        STOP;
    END; # -- End of Trigger block
END; # -- End of program
```

Figure 2: Sample PDL Program

3.4 The PDL Language

PDL is a general purpose programming language created for this research project. Through modifications to the compiler, the language evolves to meet ever-changing demands. Figure 2 shows a sample PDL program, the ”ping” program, which we use for latency measurements in Section 4.

4 Performance

We performed two experiments to assess the performance of our PANIC VM. First, we measured the message delivery latency between two VMs. For this, we performed a series of message round-trip measurements in a ping-pong setting. Second, we focused on the VM-host interface and measured the data transfer latency from a VM to the host memory.

4.1 Round-trip Latency

On a small network of Myrinet-connected hosts, we selected two, a Sparc-5 and Sparc T-1000 for the latency test. The hosts were connected by two Myricom 4-port switches (M2F-SW4) and Myricom LAN/SBus interfaces (M2F-SBus32A). Both hosts were executing no other user applications besides the test suites.

mcp2mcp: The applications are a pair of PANIC programs, see Figure 2, running entirely on the Myrinet NIC. Timestamps were captured on the card using the PDL CLOCK function and the NIC’s real-time clock register. Each CLOCK operation has an overhead of approximately 7 μsecs. The test applications include the timestamp overhead, but exclude the time required to increment and compare the counter that determines the number of packets transmitted. Pay-

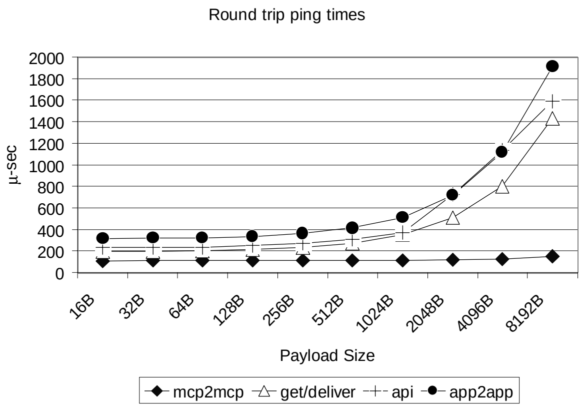

Figure 3: Round Trip Delays in PANIC

```
PROGRAM delivertest:  # DELIVER and trigger test program

    VAR      y, msg[50];
    TRIGGERS 10;

    BEGIN
        msg[0] = 2;
        msg[1] = 3;
        msg[2] = 4;
        AWAIT
    END;

    MESSAGE 10: BEGIN
        DELIVER msg[0] OFFSET 0 INTO 2 LEN 64;
        AWAIT
    END;
END;
```

Figure 4: PDL Program for Measuring Internal Time Consumption

load size excludes any header data, which is 16 bytes for all Myrinet packets and 16 bytes additionally for PANIC packets. For these tests, we send the uninitialized content of a PANIC variable to the second card, which returns that data.
GetDeliver: This experiment performs host-to-host DMA using two PANIC programs on two NICs, and measures the round-trip time: the NIC-resident applications move, via DMA, the packet payload from host memory into a PANIC variable and send that variable to the second card. The receiver DMAs the payload into a reserved location in host memory. Immediately after that delivery completes, the data is DMA'd into a PANIC variable for return to the first card. The host is not notified of the arrival or departure of the data.
api: This test uses Myricom's `api_latency` tool executing on the hosts to measure application-to-application round-trip times using a send-receive model that uses busy-polling to constantly check for an arrival packet. Although our PANIC mcp was installed, this test did not use PANIC.
app2app: In this most complicated of the tests, an application on the host and PANIC program on the NIC co-operate in the exchange. The PDL program is identical to that in the `GetDeliver` test, except that the host is notified of packet delivery and re-transmission does not occur until the host initiates the action. Like the `api_latency`, the user-application employs busy-polling to detect changes in a lock signaling packet arrival. Times are measured on the host.

The results of this series of measurements are illustrated in Figure 3, which depicts the packet round-trip latency as a function of exchange mechanism and packet size.

Our goal was not to improve Myrinet's overall performance. Other researchers have demonstrated that packet latencies can be reduced by as much as an order of magnitude over those inherent in a standard Myricom distribution [10, 11]. Rather, we show that our PANIC subsystem causes little, if any, degradation to the system in which it is embedded. Our performance benefits will come from selectively processing some packets on the card instead of speedily moving all packets to the host.

As was expected, the `mcp2mcp` pings show the promise of limiting communication to the card. The difference between the `GetDeliver` and `app2app` curves exposes excessive delay in our host-to-card collaboration. A small part of this latency is from the host reading and resetting the lock used for notification. A larger portion, up to 48 μsecs on each side, stems from an inefficient messaging interface between the host and the NIC. We have designed strategies to minimize the latter overhead.

4.2 Internal Latency

These experiments measured the time spent in different sections of our VM using the MCPs `timestamp()` function.

The PDL program in Figure 4 with a single trigger block was loaded on the card from a Sparc-5 host. That trigger block contained a `DELIVER` statement, which DMAs data from the PANIC VM memory to a previously allocated portion of memory on the host.

The PANIC VM first demultiplexes the incoming message to the correct PANIC program. Control then passes to the interpreter, which further demultiplexes the message to the correct program block, sets the program counter, ascertains the beginning of the memory segment for the program, and executes statements until a `AWAIT` is encountered. The two statements executed in the test program were `DELIVER` followed by an `AWAIT`. The `DELIVER` causes a DMA transfer from PANIC VM variable space into the specified segment in host memory. We break the `DELIVER` operation into two phases: time required for the VM to reach the `DELIVER` op code, and time required to complete the actual DMA. The small amount of time required to process the `AWAIT` is also included in the execution time of the interpreter.

The measurements of time spent in various components of the PANIC VM are shown in Figure 5. Most of the time spent in the interpreter is attributable to start-up costs. This cost cost is disproportionate to the work actually performed. Had we executed several statements, interpreter overhead would appear less burdensome.

The experiments reveal one obvious candidate for improvement - the interpreter - and a less obvious need for compiler modification. Execution of the interpreter is more costly than DMA movement of 512 bytes of data. Most of

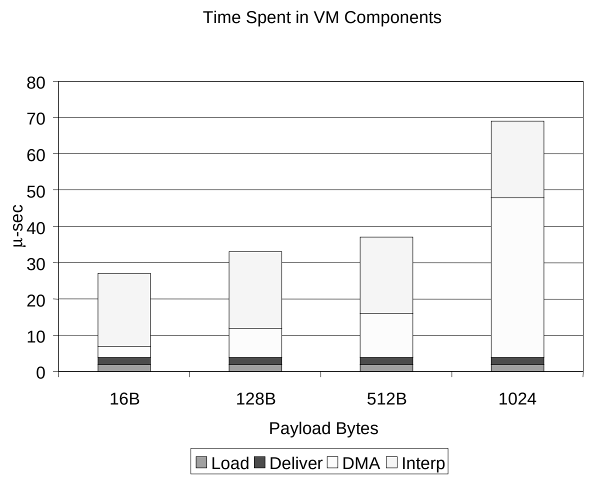

Figure 5: Proportion of Time Spent in VM Components

this latency is due to start-up costs that could be ameliorated over several statements; we are investigating ways to reduce this delay.

As the amount of data being moved via DMA increases, time required for the transfer increases linearly. For DMA movements of 1 kByte or more, time spent in the DMA machine dominates all other activities. In some circumstances the DMA must complete before program execution can continue. For example, if a program issues a `GET` command to load a local variable from host memory, then uses that variable in a computation, the DMA must finish before computation proceeds. In other situations, it would be possible to start the DMA and immediately resume execution of subsequent statements not dependent on the data transfer. We are exploring ways to modify the compiler to detect and generate code to effectively handle data dependencies.

5 Related Work

As noted, this work is the natural convergence of two distinct trends in network programming: Dynamic configuration of protocol stacks and user-level network interface protocols. Some of the salient work in each field is summarized below:

Researchers concur on the need to remove the OS from network messaging paths. A recent survey [14] reported eleven schemes to boost throughput and reduce latency. The common strategy is moving messages out of the network and into application memory with minimal OS intervention. These systems, loosely categorized as *User Level Network Interface Protocols*, report impressive delivery times. VIA, *Virtual Interface Architecture* [15], seems destined to become an industry standard.

Researchers also recognize the need to tailor protocol layers to application demands. Forcing every network message through a regimented kernel protocol stack is wasteful. As process requirements fluctuate, so too should messaging protocols. Advanced protocol design techniques include application-level framing, in which the protocol buffering is fully integrated with application-specific processing. Integrated layer processing, in which many protocol layers are collapsed into an efficient code path, is another alternative. Work in this area has led to *dynamic protocol configuration* in such well-known systems as X-Kernel [16], STREAMS [17], and DaCapo [18].

Historically, I/O interconnects have been much slower than the attached I/O devices. High-performance network cards can saturate the bus used for host-card communication. Evolution of the personal computer from a desktop tool to an enterprise server is made possible by a standardized, *intelligent I/O architecture* (I_2O) [19]. This architecture delegates much of the interrupt processing and hardware management to separate I/O processors (IOP). These IOPs may reside on the main board or the peripheral device itself. Each IOP runs its own embedded operating system tailored for the device. IOP performance is limited only by processor speed and memory availability. Both these limitations are artificial and will yield to decreasing hardware cost.

SPINE [20] is touted as a *Network Operating System*. The SPINE runtime resides in the host kernel while SPINE extension written in Modula-3 are downloaded to a Myrinet network interface. As proof that the concept is viable, two applications - video server and IP router - are implemented with impressive results. This work demonstrates that network interfaces may indeed perform work heretofore left to the host.

A more general framework for an *intelligent network device* is U-NET/SLE reported by UC-Berkeley [21]. That work extends the U-Net model by incorporating a scaled-down JAVA Virtual Machine on a Myrinet network interface. JAVA applets are downloaded to the network interface to serve as Active Message style handlers for incoming/outgoing packets. Much empirical data is provided to justify the choice of JIVE (Java Implementation for Vertical Extensions) as the virtual machine on the interface. Aside from reporting ping times between JAVA applets, the paper suggests future applications.

Perhaps closest to our PANIC system is *Active Messages* [6]. AM and its successor, AM-II [10] , are based on placing control information in message headers that will invoke a user level handler routine. The handler will efficiently extract the message from the network to the sender-specified address in host memory. The message is detected either by polling or interrupt. Polling is likely the best choice inasmuch as waking a sleeping thread on Solaris requires nearly 100 microseconds. The key difference is that the PANIC component is invoked on the NIC rather than host memory. Another difference is the variety of actions. A PANIC component may help direct the message to a specific user memory location, redistribute the message to other NIC's or hosts, or invoke another component in the protocol scheme.

VMMC [22] - *Virtual Memory-Mapped Communication* - is yet another system for optimizing message delivery. In a rather complex set-up procedure, receiving processes export areas of their address space wherein they agree to accept incoming data. Sending processes import these remote buffers they will use as destinations for transferred data. When a message arrives at a Myrinet NIC, it is automatically transferred into the memory of the receiving process. No ”receive” primitive is defined. Attaching notification to a message causes invocation of a user level handler function once the message is received. Incoming and outgoing page tables are kept in a specialized NIC control program. That control program maintains a virtual-to-physical two-way set associative TLB for each process using VMMC on a given node. One way latency of 9.8 microseconds is reported. As previously noted, the major cost is disabling of Myrinet’s self-mapping ability.

6 Conclusion

Our claim is that splitting protocol components across host and NIC is not only possible but yields performance benefits superior to those of an entirely host based system. In this paper we have shown that a virtual-machine based approach, as realized in PANIC, provides good performance. We are currently adapting a suite of higher-level protocols to PANIC to collect data on the performance gain that can be achieved by moving latency critical components into the NIC. We are working with protocols in two areas: reliable multicast and distributed shared memory.

Reliable multicast protocols often rely on token-passing mechanisms and simple queue management for message ordering in the receiver hosts. We can take advantage of mechanisms that are particularly well suited for very high-speed, low error-rate, networks, such as “buffering on the link” by returning messages to the sender for flow control [12].

To further illustrate the feasibility and the performance benefits of PANIC-style network-level protocols, we are currently porting a distributed shared memory libray (in our case Quarks [13]) to PANIC. Empirical evidence will be collected to compare the PANIC-based DSM system with a traditional host-based implementation.

References

[1] Anderson, Culler, Patterson, A Case for Networks of Workstations NOW, I.E.E.E. Micro, February 1995.

[2] Rodrigues, Andi, Culler, High Performance Local-Area Communication Using Fast Sockets, USENIX 97

[3] Lumetta, Culler, Managing Concurrent Access for shared Memory Active Messages, IPPS/SPDP 98, Orlando, Florida, March 1998

[4] Dusseau, Arpaci, Culler, Effective Distributed Scheduling of Parallel Workloads, SIGMETRICS ’96 Conference on Measurements and Modeling, 1996

[5] Vogels, W. et al., Scalability of the Microsoft Cluster Service, Proceedings of the Second Usenix Windows NT Symposium, USENIX Association, Berkeley, Ca, 1988, pp 11-29

[6] Prylli, L., and Tourancheau, Protocol design for high performance networking: a Myrinet experience, Technical Report 97-22, LIP-ENS Lyon, 69364 Lyon, France, 1997.

[7] Tennenhouse, D., and Wetherall, D., Towards an active Network Architecture, Computer Communication Review, Vol 26, No. 2, April 1996

[8] Welsh, Basu, and von Eicken, Incorporating Memory Management into User-Level Network Interfaces, Proceedings of Hot Interconnects V, August 1997.

[9] Von Eicken et al., U-Net: A User-level Network Interface for Parallel and Distributed Computing, Proc 15th Symp., Operating System Principles, ACM Press, New York, 1995, pp. 40-53.

[10] B. Chun, A. Mainwaring, and D. Culler, Virtual Network Transport Protocols for Myrinet, IEEE Micro, Jan 1998, pp 53-63.

[11] C. Dubnicki et al, Myrinet Communication, IEEE Micro, Jan 1998, pp. 50-52

[12] Verstoep, K., Langendoen, K, and Bal, H., Efficient Reliable Multicast on Myrinet,

[13] Copyright 1995, University of Utah and Comuter Systems Laboratory (CS).

[14] Bhoedjang, Ruhl, and Bal. User-Level Network Interface Protocols, IEEE November 1998, pp. 53-60

[15] von Eicken and Vogels, Evolution of the Virtual Interface Architecture, Computer, November 1998, pp 61.68

[16] Hutchison, N., and Peterson, L., The x-kernel: An Architecture for Implementing Network Protocols, I.E.E.E. Transactions on Software engineering, vol 17, pp. 64-76, January 1991.

[17] UNIX Software Operations, UNIX System V Release 4 Programmers Guide, STREAMS, Prentice Hall 1990.

[18] Plagemann, T., A Framework for Dynamic Protocol Configuration, to be published in European Transactions on Telecommunications (ETT), Special issue on Architecture, Protocols, and Quality of Service for the Internet of the Future, 1999.

[19] Wilner, D., I2O’s OS Evolves, Byte Magazine, April 1998, 47-48.

[20] Fiuczynski, M, and Bershad, B, SPINE - A safe programmable and integrated network environment, SOSP 16 Works in Progress, 1997.

[21] Oppenheimer, D. and Welsh, M., User Customization of Virtual Network Interfaces with U-Net/SLE, Technical Report, http://www.cs.berkeley.edu/ mdw/projects/unet-sle

[22] Dubnicki et al., Design and Implementation of Virtual Memory-Mapped Communication on Myrinet, Proc. Int’l Parallel Processing Symp., IEEE CS Press, Los Alamitos, CA., 1997, pp. 388-396