A Multithreaded Message-Passing System for High Performance Distributed Computing Applications

Sung-Yong Park, Joohan Lee, and Salim Hariri

High Performance Distributed Computing (HPDC) Laboratory
Department of Electrical Engineering and Computer Science
Syracuse University
Syracuse, NY 13244
{sypark, jlee, hariri}@cat.syr.edu

Abstract

High Performance Distributed Computing (HPDC) applications require low-latency and high-throughput communication services and HPDC applications have different Quality of Service (QOS) requirements (e.g., bandwidth requirement, flow/error control algorithms, etc.). The communication services provided by traditional message-passing systems are fixed and thus can not be changed to meet the requirements of different HPDC applications.

NYNET (ATM wide area network testbed in New York state) Communication System (NCS) is a multithreaded message-passing system developed at Syracuse University that provides high-performance and flexible communication services. In this paper, we overview the general architecture of NCS and present how NCS communication services are implemented. NCS point-to-point communication is flexible in that users can configure efficient point-to-point primitives by selecting suitable flow control, errror control algorithms, and communication interfaces on a per-connection basis. Furthermore, NCS architecture separates the data transfer and control transfer functions that allows the control information to be transmitted over the control connections and thus improves the perforamnce of the point-to-point communication primitives.

We analyze the overhead incurred by using multithreading and compare the performance of NCS point-to-point communication with those of other message-passing systems such as p4, PVM, and MPI. Benchmarking results indicate that NCS shows comparable performance with other systems in transmitting small messages but outperforms other systems for large messages.

1 Introduction

Current advances in processor technology and the rapid development of high-speed networking technology (e.g., Asynchronous Transfer Mode (ATM) [1], Myrinet [2], and Fast Ethernet [3], High Performance Parallel Interface (HIPPI) [4]) have made network-based computing, whether it spans a local or a wide area, an attractive and cost-effective environment for large-scale High Performance Distributed Computing (HPDC) applications. The development of HPDC applications over such an environment is not a non-trivial task that requires a thorough understanding of the applications with widely differing performance characteristics. HPDC applications require low-latency and high-throughput communication services. HPDC applications have different Quality of Service (QOS) requirements and even one single application has multiple QOS requirements during the course of its execution (e.g., interactive multimedia applications).

There have been several inter-process communication libraries such as p4 [9], Parallel Virtual Machine (PVM) [10], Message Passing Interface (MPI) [11], Express [12], PARMACS [13], Linda [14], Isis [15], Horus [16], and Remote Procedure Call (RPC) [17] that simplify process management, inter-process communication, and program debugging in a parallel and distributed computing environment. However, the communication services provided by traditional communication systems are fixed and thus can not be changed to meet the requirements of different HPDC applications. In order to support HPDC applications efficiently, future communication systems should provide high performance and dynamic communication services to meet the requirements of a wide variety of HPDC applications.

NYNET (ATM wide area network testbed in New York state) Communication System (NCS [7] [8]) is a multithreaded message-passing system for an ATM-based HPDC Environment that provides low-latency and high-throughput communication services. NCS uses multithreading to provide efficient techniques to overlap computations and communications. By separating control and data activities, NCS eliminates unnecessary control transfers. This optimizes the data path and improves the performance. NCS supports several different communication schemes (multicasting algorithms, flow control algorithms, and error control algorithms) and allows the programmers to select at runtime the suitable communication schemes per-connection basis. NCS provides three application communication interfaces such as Socket Communication Interface (SCI), ATM Communication Interface (ACI), and High Performance Interface (HPI) to support various classes of applications with the appropriate communication services. The SCI is provided mainly for applications that must be portable to many different computing platforms. The ACI provides the services that are compatible with ATM connection-oriented services where each connection can be configured to meet the QOS requirements of that connection. This al-

lows the programmers to fully utilize the benefit of ATM networks. The HPI supports applications that demand low-latency and high-throughput communication services.

In this paper, we overview the general architecture of NCS and present how NCS communication services are implemented. NCS point-to-point communication is flexible in that users can configure efficient point-to-point primitives by selecting suitable flow control, error control algorithms, and communication interfaces on a per-connection basis. Furthermore, NCS architecture separates the data transfer and control transfer functions that allows the control information to be transmitted over the control connections and thus improves the performance of the point-to-point communication primitives.

The rest of the paper is organized as follows. Section 2 presents the general architecture of NCS. Section 3 discusses an approach to implement NCS point-to-point communication services over an ATM network. Section 4 analyzes and compares the performance of NCS point-to-point communication with those of several other message-passing systems such as p4, PVM, and MPI. Section 5 contains the summary and conclusion.

2 Overview of NCS Architecture

NCS is a multithreaded message-passing system that provides application programmers with multithreading (e.g., thread synchronization, thread management) and communication services (e.g., point-to-point communication, group communication, synchronization). NCS is architecturally compatible with the ATM technology where both control (e.g., signaling or management) and data transfers are separated and each connection can be configured to meet the QOS requirements of that connection. Consequently, the NCS architecture is designed to support various classes of applications by providing the following architectural supports (see Figure 1):

Thread-Based Programming Paradigm

NCS uses multiple threads to implement the computations of HPDC applications (we call them *Compute_Threads*). These threads use the NCS primitives to communicate and synchronize with other *Compute_Threads*. The advantage of using the thread-based programming paradigm is that it reduces the cost of context switching, provides efficient support for fine-grained applications, and allows the overlapping of computation and communication. Overlapping computation and communication is an important feature in network-based computing. In Wide Area Network (WAN)-based distributed computing, the propagation delay (limited by the speed of light) is several orders of magnitude greater than the time it takes to actually transmit the data [6]. For example, to transmit a 1-Kbyte file across

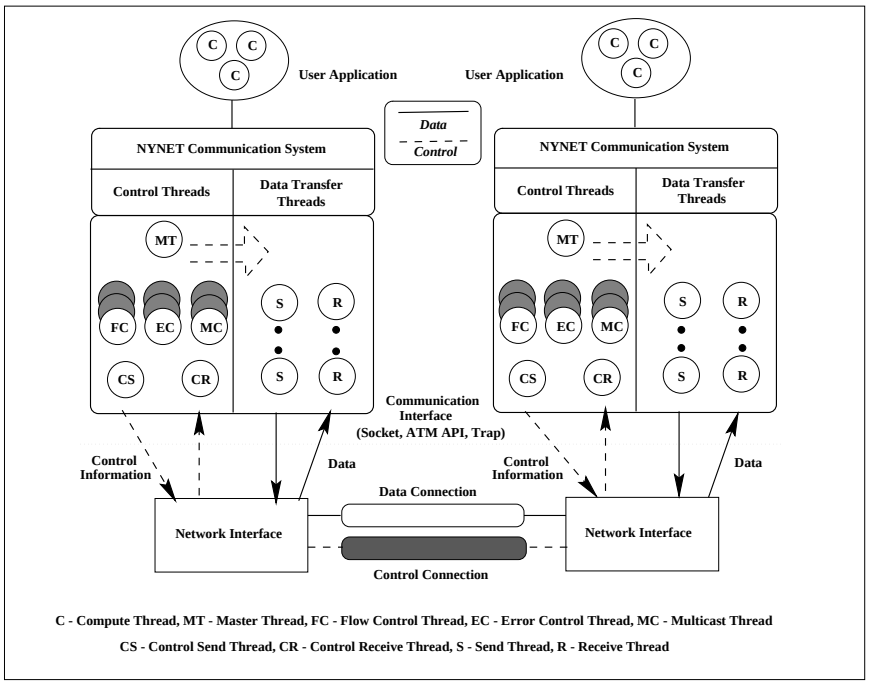

Figure 1: NCS General Architecture

the U.S at 1 Gbps takes only 8 *micro*seconds. However, the time it takes for the first bit to arrive at its destination (propagation delay) is 15 *milli*seconds. Consequently, the transmission time of this file is insignificant when compared to the propagation delay. To reduce the impact of the propagation delay requires that we modify the structure of computations such that they overlap communications.

Separation of Control and Data Functions

In high-speed networks, very little time is available to decode, process, and store incoming packets at gigabit rate. Also, the bandwidth provided by the high-speed networks is generally enough to be allocated to multiple connections. Therefore, the software architectures of communication systems for high-speed networks should be designed to fully exploit these requirements. The separation of control and data functions enables NCS to work efficiently in high-speed networks.

NCS separates control and data functions by providing two planes: *control plane* and *data plane*. The *control plane* consists of several threads that implement important control functions (e.g., connection management, flow control, error control, and configuration management) in an independent manner. These threads include *Master_Thread*, *Flow_Control_Thread*, *Error_Control_Thread*, *Multicast_Thread*, *Control_Send_Thread* and *Control_Receive_Thread* (we call them *control threads*). The *data transfer threads* in the

data plane (*Send_Thread* and *Receive_Thread*) are spawned based on a per-connection basis by the *Master_Thread* to perform only the data transfers associated with a specific connection. By having separate *data transfer thread*s per connection, the overhead associated with demultiplexing incoming packets is eliminated and each connection can provide different QOS requirements for its application. The separation of control and data functions increases flexibility by reducing control and data dependencies. This modular architecture allows easier modification and enhancement of NCS services by simply adding an NCS *control thread* for each new function or service.

In NCS, the control and data information from the two planes are transmitted on separate connections. All control information (e.g., flow control, error control, configuration information) is transferred over the control connections, while the data connections are used only for the data transfer functions. The separation of control and data connections eliminates the process of demultiplexing control and data packets within a single connection and allows the concurrent processing of control and data functions. This allows applications to utilize all available bandwidth for the data transfer functions and thus improves the performance.

Dynamic Support for Multiple Communication Algorithms

Each HPDC application requires different schemes for flow control, error control, and multicasting algorithms. One of the main goals of NCS is to provide a modular approach to support these requirements efficiently.

NCS supports multiple flow control (e.g., window-based, credit-based, or rate-based), error control (e.g., go-back N or selective repeat), and multicasting algorithms (e.g., repetitive send/receive or a multicast spanning tree) within the *control plane* to meet the QOS requirements of a wide range of HPDC applications. Each algorithm is implemented as a thread and programmers activate the appropriate thread when establishing a connection to meet the requirements of a given connection. This allows programmers to select for a given HPDC application the appropriate flow control, error control, and multicasting algorithms per-connection basis at runtime. For example, interactive multimedia applications (see Figure 2) use audio, video, and data media streams which have different QOS requirements. Voice and video streams need low latency and jitter but can tolerate moderate error rate, while data stream has no real-time constraint but requires error-free transfer. By using NCS, programmers can select no flow or error control for the audio and video connections, while they select the appropriate flow control or error control algorithms to achieve a reliable connection for data transfer. Consequently, the performance of these applications can be maximized by removing the overheads associated with flow control and error control

procedures in connections that do not need these capabilities.

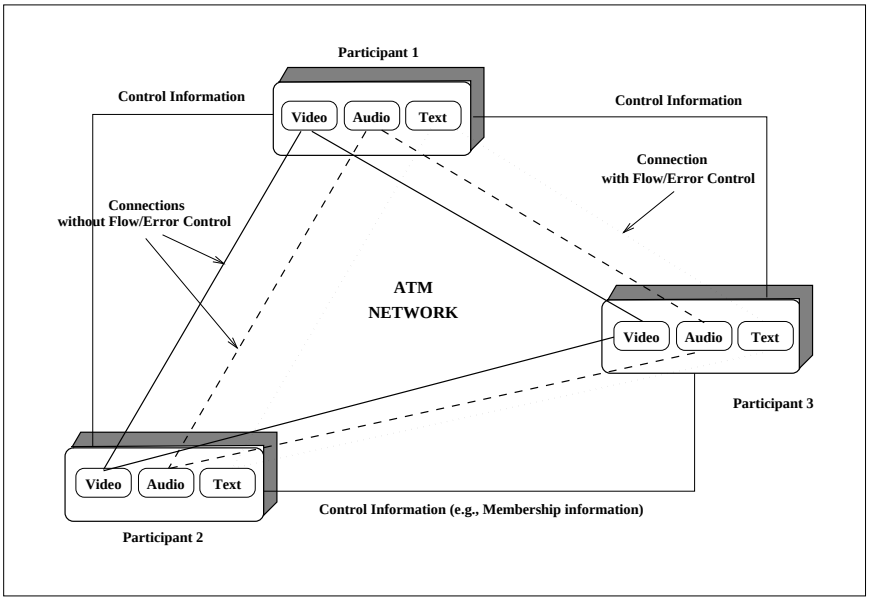

Figure 2: A Multimedia Application using NCS

Multiple Communication Interfaces

Some HPDC applications demand low-latency and high-throughput communication services to meet their QOS requirements, while others need portability across many computing platforms. Most of the message-passing systems cannot dynamically support a wide range of QOS requirements, because their protocol architectures and communication interfaces are fixed. NCS is designed to support these classes of applications by offering three application communication interfaces: 1) Socket Communication Interface (SCI), 2) ATM Communication Interface (ACI), and 3) High Performance Interface (HPI).

The SCI is provided mainly for achieving high portability over a network of computers (e.g., workstations, PCs, parallel computers). One of the disadvantages of using this interface is that we have to use the inherent flow control, error control algorithms in TCP/IP protocol and thus cannot fully exploit the features of NCS. However, some applications that require mainly the dynamic group communications and fault tolerance capability can exploit the NCS architectural feature from this interface due to the separation of control and data path. Moreover, considering that the Socket interface is supported on almost all UNIX-based workstations (even on the PCs with Winsock [20] [21]) and a number of ATM networks are running with TCP/IP protocol, supporting Socket interface is viable when implementing future communication systems.

The ACI is the application communication interface that provides applications with

more flexibility to fully utilize the benefits of NCS architecture when applied to ATM networks. Since ATM Application Programming Interface (API) does not define the flow control and error control schemes, programmers can select the appropriate communication services according to the QOS requirements of HPDC applications. The direct access to the ATM Adaptation Layer (AAL) allows us to use the inherent features of an ATM API (e.g., one-to-many connection). However, due to the lack of a standard ATM API and its poor performance, this interface has not been widely used. Recently, some research activities are underway to standardize the ATM API [21] and to develop high-performance ATM APIs [18] [19].

The HPI allows applications to achieve high-throughput and low-latency inter-process communications. This interface is usually built by modifying system software such as device driver or firmware code of the adapter card. Due to its dependencies on specific computing platforms (e.g., hardware, operating system), this interface is targeted at developing high-performance communication interfaces for the tightly-coupled cluster of homogeneous workstations.

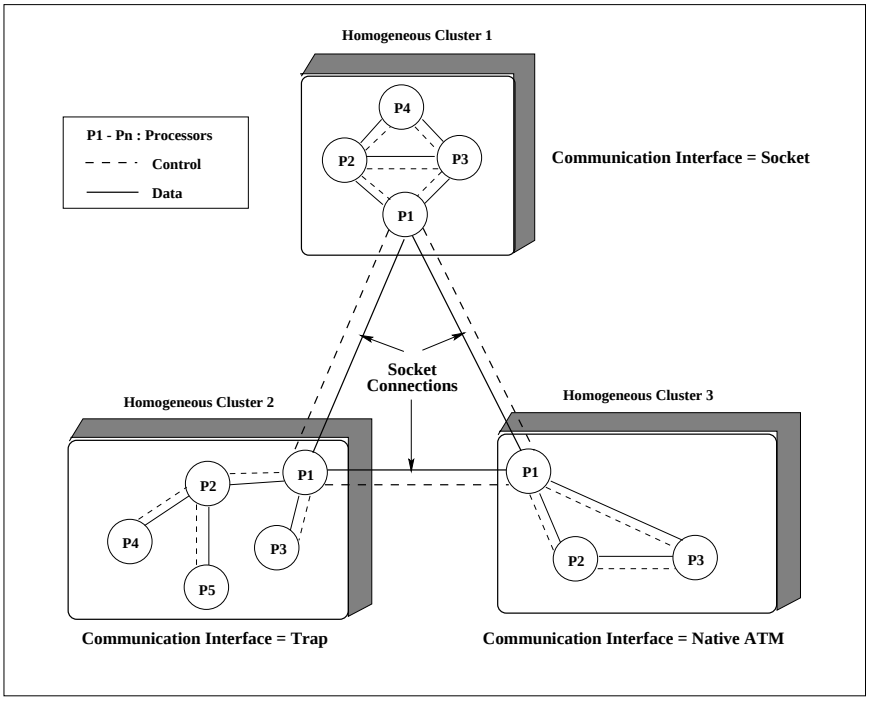

Figure 3: An Example Using Multiple Communication Interfaces in NCS

NCS architecture is flexible and can be used to build a large heterogeneous distributed computing environment that consists of several homogeneous clusters (see Figure 3). In the environment shown in Figure 3, each homogeneous cluster can be configured to use the

appropriate NCS application communication interface that is supported by the underlying computing platform and each cluster can be interconnected by using the SCI. This improves the performance of each cluster and thus improves the overall performance of applications running over this environment.

In what follows, we show how NCS architecture can be applied to provide point-to-point communication services. Similar approach is used to provide other NCS communication services. Additional details about NCS communication services can be found in [7] [8].

3 Point-to-Point Communications in NCS

In NCS point-to-point communication, it is assumed that both the sending *Compute_Thread* and the receiving *Compute_Thread* agree to communicate with each other and that the receiving *Compute_Thread* has explicitly invoked a *NCS_recv()* primitive to receive the message.

The NCS point-to-point communication is flexible. Users can configure efficient point-to-point communication primitives by selecting suitable flow control, error control algorithms, and communication interfaces on a per-connection basis. Those primitives may be reliable or unreliable, configured for achieving portability or for special requirements (e.g., low-latency for small messages). By transmitting control information over separate control connections, the performance of these primitives can be maximized. After a connection is established with appropriate QOS requirements (e.g., flow control algorithm, error control algorithm, communication interface), the underlying operations are transparent to users and they just need to invoke the same high-level abstractions (NCS primitives) to perform point-to-point communication independent of the selected configurations.

In what follows, we describe the communication flow when *NCS_send()* and *NCS_recv()* primitives are invoked at both ends. Next, we present algorithms to implement error control and flow control. Since NCS supports several different flow control and error control algorithms, the descriptions for these algorithms are focused on one specific implementation (e.g., default algorithms). Since each algorithm will be implemented as a thread, we can easily incorporate other advanced algorithms into the NCS architecture by activating the appropriate algorithms at runtime.

3.1 Communication Flow

NCS point-to-point communication can be described in terms of ten steps, as shown in Figure 4. In this example, we assume that each connection is configured with the appropriate error control algorithm, flow control algorithm, and communication interface.

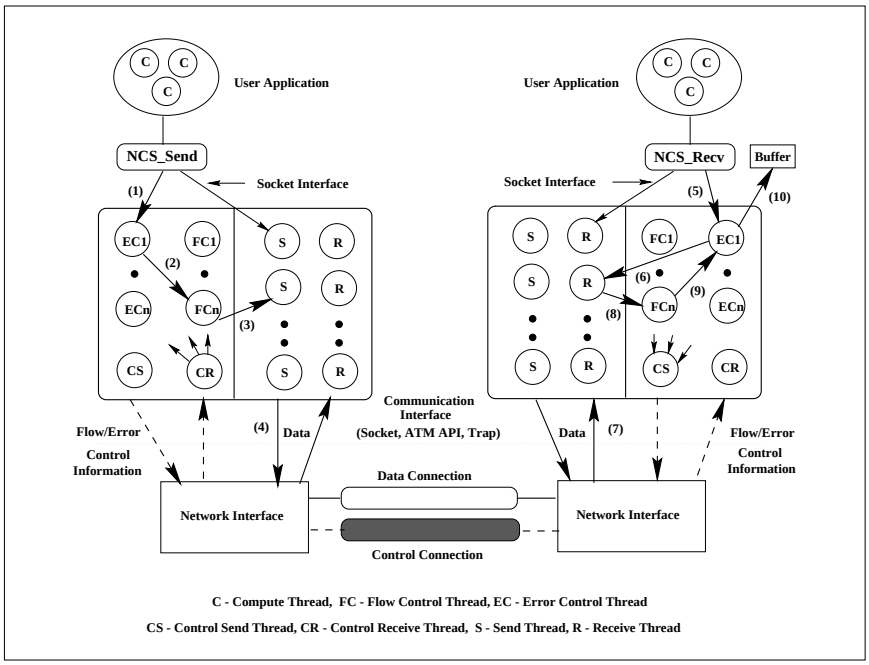

Figure 4: Point-to-Point Communication in the NCS Environment

1. When *NCS_send()* is invoked at the source *Compute_Thread*, it activates the corresponding *Error_Control_Thread* associated with the sending connection. Since each connection is bound to its own *Flow_Control_Thread* and *Error_Control_Thread*, the *Compute_Thread* should provide exact connection parameters (e.g., destination process id, destination thread id, session id) when calling the *NCS_send()* primitive.

2. The *Error_Control_Thread* in turn activates the corresponding *Flow_Control_Thread* after segmenting the user message into packets based on the Service Data Unit (SDU) size and attaching a header to each packet.

3. The *Flow_Control_Thread* then activates the corresponding *Send_Thread* based on the flow control information it is maintaining.

4. The *Send_Thread* transmits the requested packets over the data connection using the communication interface configured for this connection.

5. On the receiving side, the *Compute_Thread* invokes the *NCS_recv()* primitive and it activates the corresponding *Error_Control_Thread* associated with the receiving connection.

6. The *Error_Control_Thread* activates the corresponding *Receive_Thread* to receive the whole segmented packets.

7. The *Receive_Thread* receives a packet over the data connection using the communication interface configured for this connection.

8. The *Receive_Thread* activates the corresponding *Flow_Control_Thread* to check the flow control status.

9. The *Flow_Control_Thread* updates the flow control information and sends the information to the source *Flow_Control_Thread* over the control connection. On the other hand, it activates the corresponding *Error_Control_Thread*.

10. After receiving all segmented packets, the *Error_Control_Thread* reassembles the packets and puts them into the user buffer. The *Error_Control_Thread* also sends the error control information to the source *Flow_Control_Thread* over the control connection.

For environments where flow control and error control are not required (e.g., Socket Interface), the *NCS_send()* and *NCS_recv()* primitives bypass the *Flow_Control_Thread* and *Error_Control_Thread* by activating the corresponding *Send_Thread* and *Receive_Thread* directly.

3.2 Error Control

The error control procedures in NCS are designed to support reliable point-to-point data transfer by detecting errors and recovering from errors once they occur. Although the checksumming is done by the AAL5 layer to detect errors within the AAL5 frames, acknowledgment and retransmission procedures are required to guarantee the reliable delivery of user messages.

NCS supports several different error control algorithms, and users can select the appropriate error control algorithm according to the requirements of the applications. In applications that do not require error control procedure, users can deactivate it in NCS to reduce the overhead incurred by using an error control scheme.

The default error control algorithm in NCS is based on *selective repeat* strategy [22], as shown in Figure 5. This algorithm can be outlined in the following five steps:

1. **Segmentation:** The user message is segmented into packets based on the SDU size, which is defined by the user.

2. **Header Generation:** Each SDU has a sequence number and a control bit in the header that designates whether the SDU is the last SDU to be segmented. If the bit is set to 1, it activates the *Error Control Thread* at the receiver side to send

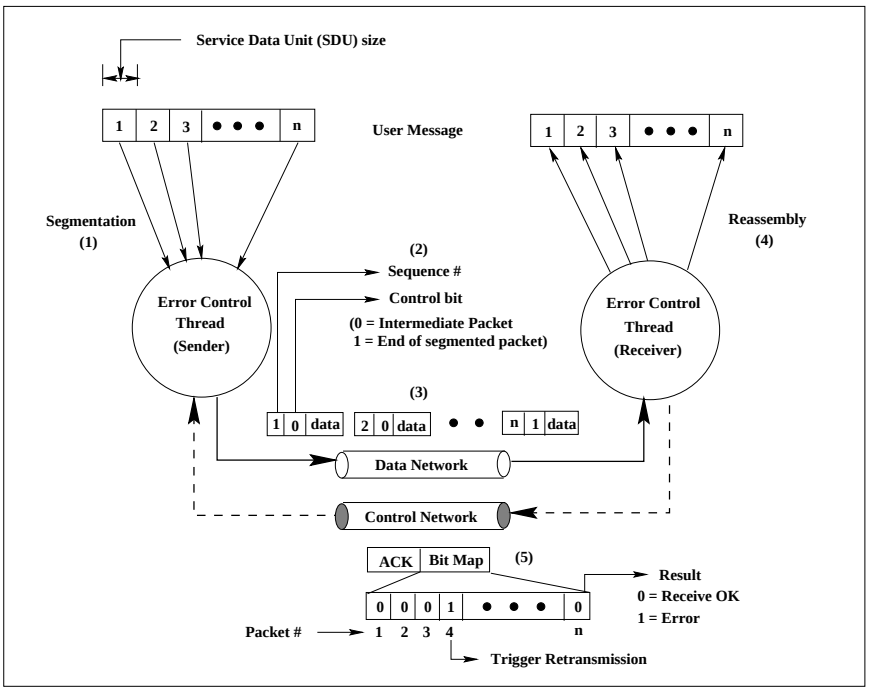

Figure 5: Selective Repeat Error Control Scheme in the NCS Environment

an Acknowledgment packet to the *Error Control Thread* at the sender side over the control connection.

3. **Data Transmission:** Each segmented SDU is delivered to the *Flow_Control_Thread* to be transmitted over the data connection by the *Send_Thread*.

4. **Reassembly:** At the receiver side, each segmented SDU is reassembled by the *Error Control Thread* if the SDU is received without errors. The *Error Control Thread* also updates bitmap information that represents the status of the received SDUs. Each bit in the bitmap corresponds to one SDU. If the SDU is received in error, the corresponding bit in the bitmap is set to 1.

5. **Acknowledgment:** If the control bit in the header of a received SDU is set to 1, the receiving *Error Control Thread* sends an Acknowledgment packet containing a bitmap that was updated in step 4 over the control connection. The *Error_Control_Thread* at the sender side retransmits the corresponding SDU if the bitmap in the Acknowledgment packet indicates that the SDU is received in error (e.g., if the corresponding bit in the bitmap is set to 1). If the *Error_Control_Thread* at the sender side does not receive an Acknowledgment packet within an appropriate interval (e.g., timeout), it retransmits the whole packets.

The pseudo code for this algorithm is presented in Figure 6. The SDU size is the unit of error control and retransmission in NCS. The SDU size is from 4 Kbytes to 64 Kbytes and corresponds to the single AAL5 frame (Default SDU size is 4 Kbytes). The reason for this is that some ATM API such as Fore Systems' ATM API restricts the size of the user message to less than 4 Kbytes and the single AAL5 frame is at most 64 Kbytes long. In general, a large SDU size generates high throughput, but results in high overhead by retransmission when the SDUs are lost. By keeping the size small, efficiency can be maximized but segmentation overheads (e.g., header and trailer) are introduced. Therefore, this size should be chosen for each environment to trade off per-fragment overhead, the connection's error characteristics, and the available timer resolution [5].

3.3 Flow Control

The flow control scheme is used to control the transmission rate to avoid receiver overrun or the network becoming congested. Several flow control algorithms have been proposed for high-speed networks such as rate-based, credit-based, and window-based algorithms. One of the drawbacks in existing protocols is that the flow control algorithm is fixed and cannot optimally control a wide range of HPDC applications with different QOS requirements. This occurs because an algorithm that is optimal in one environment may not necessarily be optimal in another environment. For example, the applications that control audio or video streams require minimum flow control for those streams, because the audio and video streams are usually error-resilient. However, data applications need a flow control scheme to guarantee reliable delivery of the data. Moreover, if both audio/video and data streams coexist in one application as in realtime interactive multimedia applications, we cannot optimally control these applications using the existing protocol architecture.

NCS supports several flow control algorithms and allows programmers to select the appropriate algorithm per-connection basis at runtime according to the needs of the application. The default flow control algorithm in NCS is the *credit-based* window flow control algorithm. Figure 7 shows the main steps of the NCS flow control algorithm and can be explained as follows:

1. When the *Flow_Control_Thread* is activated by the *Error_Control_Thread*, it first checks the *credit* buffer for the given connection and determines the appropriate number of packets to transmit. Each process maintains a separate queue and *credit* buffer for each connection.

2. The *Flow_Control_Thread* puts the packets into the message queue maintained by the *Send_Thread* based on the number of *credit*s (e.g., in Figure 7, the *credit* is k).

```
Thread Error-Control-Sender
    Get the user message from the NCS_send() routine
    Determine SDUsize (4K-64K)
    remsize ← messagesize, seqno ← 0
    while remsize > SDUsize do
        Segment the message into the packet with SDUsize
        Attach a seqno, Endbit ← 0
        Activate Flow Control Thread to transmit this packet
        remsize = remsize − SDUsize, seqno = seqno + 1
    endwhile
    Create last packet
    Attach a seqno, Endbit ← 1
    Activate Flow Control Thread to transmit this last packet
    Start Timer
    Wait for an Acknowledgment Packet from the Error-Control-Receiver
    if timeout then
        Go to Line 4 for retransmission
    else
        Stop Timer
        Check the received Bitmap
        if Bitmap > 0 then
            Selective Retransmission according to the Bitmap
        endif
    endif

Thread Error-Control-Receiver
    Get the request from the NCS_recv() routine
    Endbit ← 0, Bitmap ← −1
    while Endbit == 0 do
        Activate Receive Thread to receive a packet
        Extract seqno and Endbit from the packet
        Clear seqno position in the Bitmap
    endwhile
    Send an Acknowledgment Packet with Bitmap to the Sender
    if Bitmap > 0 then
        Endbit ← 0
        Go to line 4 for receiving retransmitted packet
    endif
    Reassemble the segmented packets into the user buffer
```

Figure 6: Pseudo Code for Selective Repeat Error Control Algorithm in NCS

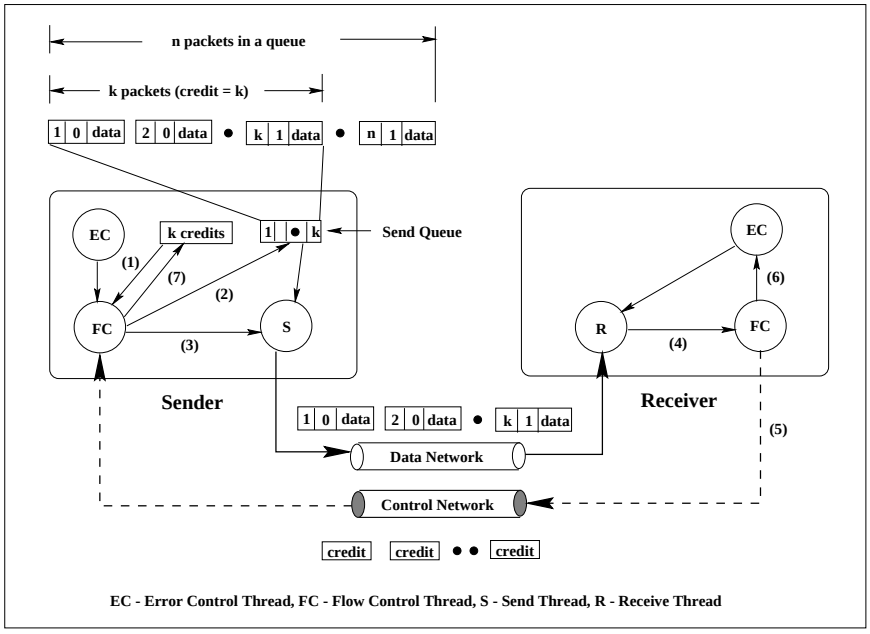

Figure 7: Credit-based Flow Control Scheme in the NCS Environment

This *credit* is an indication of how many packets can be transmitted without any acknowledgment from the receiver.

3. The *Flow_Control_Thread* activates the corresponding *Send_Thread* to transmit the packets over the data connection.

4. When the *Receive_Thread* receives a packet, it activates the *Flow_Control_Thread* associated with the given data connection.

5. The *Flow_Control_Thread* sends a *credit* to the sender over the control connection.

6. The *Flow_Control_Thread* activates the corresponding *Error_Control_Thread* to update the error control information and reassemble the original message.

7. After the *Flow_Control_Thread* at the sender side receives the *credit*, the *credit* information associated with that connection is updated.

The pseudo code for this algorithm is presented in Figure 8. From the implementation perspective, one *credit* corresponds to a free buffer allocated for receiving a packet. Since buffers are shared resources with other threads, it is not an efficient way to allocate fixed credits for every connection. In NCS, the *credit* for each connection is maintained dynamically. Only small *credit*s are assigned to each connection initially. The *Flow_Control_Thread* checks the data rate of each connection and adjusts accordingly the *credit* given to each

connection. As a result, active connections get more *credit*s, while inactive connections get only a fraction of the credits.

```
Thread Flow-Control-Sender
    if credit information received from Flow-Control-Receiver then
        Update the credit buffer for a specific connection
    endif
    Get the request to send packets from the Error Control Thread
    Read credit buffer and determine the max_num_packets
    k ← max_num_packets
    Put k transmit requests to Send Thread
    Activate Send Thread to transmit these k packets


Thread Flow-Control-Receiver
    if a packet received from Receive Thread then
        Update credit value
        Transmit credit information to Flow-Control-Sender
        Activate Error Control Thread to reassemble the packets
    endif
```

Figure 8: Pseudo Code for Credit-based Flow Control Algorithm in NCS

4 Benchmarking Results

This section analyzes the performance and overhead associated with multithreading to implement NCS point-to-point communication services. First, we quantify the effect of the thread package architecture (e.g., user-level thread or kernel-level thread) to implement a multithreaded message-passing system. Next, we measure the overhead incurred by using thread-based point-to-point communication instead of the point-to-point communication primitives provided by the underlying communication interface. Finally, we compare the performance of NCS point-to-point communication primitives with those of other message-passing systems such as p4, PVM, and MPI using the two homogeneous workstations (e.g., two SUN-4s running SunOS 5.5 or two IBM/RS6000s running AIX 4.1) or two heterogeneous workstations (e.g., SUN-4 and IBM/RS6000).

4.1 Thread Package Architecture

A user-level thread package is implemented as a user-level linkable library that includes all of the thread functions such as thread management and thread synchronization. Since the

entire functions are running completely within an application program's address space, the operating system is not aware of multiple threads. Therefore, the functions (e.g., creating threads, context switching between threads, synchronization between threads) of a user-level thread package is very fast and efficient. However, if one thread makes a blocking system call, the kernel blocks the whole process and thus eliminates the benefits of using multiple threads. In NCS implementation of user-level thread packages, all blocking primitives are implemented using non-blocking system calls to prevent the whole process from blocking. For example, when the *Receive_Thread* does not detect any incoming messages, it yields its control using the *NCS_thread_yield()* primitive, and thus other *Compute_Thread*s that are ready to run can continue their execution. This overlapping of communication and computation improves the performance of *NCS_recv()* operations. Due to its fast context switching and synchronization time between threads, the NCS implementations using user-level thread packages are suitable for applications where relatively small-size messages are exchanged frequently and a great many synchronization operations (e.g., mutex, semaphore etc.) are involved by the *Compute_Thread*s.

On the other hand, in a kernel-level thread package, an operating system directly manages threads and synchronization between threads. Therefore, it is slower to execute thread functions (e.g., creating threads, context switching and synchronization between threads) than in a user-level thread package. However, the kernel allows applications to use blocking system calls without blocking the entire process. In this case, the *NCS_recv()* primitive can be implemented using a blocking receive call to reduce the overhead of unnecessary context switching. Moreover, the kernel allows programmers to overlap computation and communication when a thread is blocked to wait for unavailable system resources. For example, if the data transmissions using BSD sockets exceed the buffering available in the socket send buffer maintained in the kernel, the kernel puts this thread to sleep and switches to another thread so that other available threads can continue their execution. To check the effect of overlapping in a kernel-level thread package, we measured the time required to send messages of various sizes (from 1 byte to 64 Kbytes) using the test program described in Figure 9. The socket buffer size was set to 32 Kbytes and some amount of computation was added after the *NCS_send()* operation in order to model the general behavior of the application programs. The time was averaged over 100 iterations.

Figure 10 shows the experimental results using two thread packages of different architecture (e.g., Pthread over Solaris, Quickthreads over Solaris). As we can see from Figure 10, the performance of a user-level thread package (Quickthreads) is better than that of a kernel-level thread package (Pthread) up to the 4-Kbyte message size. This indicates that the thread synchronization overhead in a kernel-level thread package is larger than that of

```
1  Thread Thread-Architecture-Comparison
2      start ← current_time
3      msgsize ← test_size
4      for 1 to 100 do
5          NCS_send(msgsize) /* Activate Send_Thread */
6          Computation (100 ms)
7      endfor
8      end ← current_time
9      avgtime ← (end − start)/100
```

Figure 9: Test Code to Check the Overlapping Effect in the Kernel-Level Thread

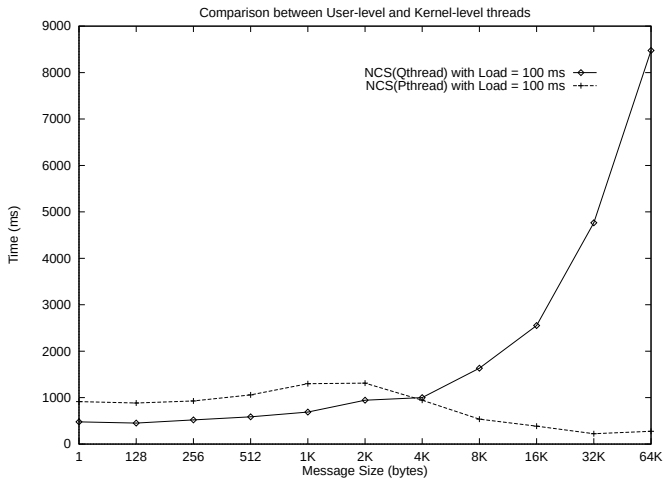

Figure 10: Comparison Between User-Level Thread and Kernel-Level Thread

a user-level thread package, and this dominates the total time used to send messages. However, for message sizes larger than 4 Kbytes, the sending time using a kernel-level thread package (Pthread) starts decreasing. This can be explained by the following observations. Since we have 32 Kbytes of socket buffer size and we repeatedly transmit large messages (e.g., larger than 4 Kbytes), the kernel finally runs out of the socket buffer and blocks the *Send_Thread* that has invoked *write()* system calls for this socket. In the user-level thread package, this results in blocking the whole process, while in the kernel-level thread package, the control is transfered to the original thread and this thread starts the computation while the blocked *Send_Thread* is waiting for the buffer to be released by the previous send operations. This overlapping of computation and communication makes NCS implementations of kernel-level thread packages suitable for applications that require exchanging large amount of data.

4.2 Thread Overhead

To evaluate the overhead incurred by using separate threads for transmitting and receiving operations, we measured the overhead involved in transmitting a 1-byte message using BSD Socket Interface. Since the main objective of this evaluation is to measure the thread overhead in terms of *NCS_send()* operations, we do not include the time for setting up the connection and assume that the connection is already set up before transmitting a message.

Table I: Cost of Sending 1-Byte Message via *Send_Thread* - QuickThreads Version

Activity	**Time** (*usec*)	**% of Total**
Session Overhead		
NCS_send() Function Entry/Exit	10	
Attaching a Message Header	4	
Queuing a Message Request	15	
Context Switch from NCS_send() to Send_Thread	27	
Dequeuing a Message Request	17	
Free a Message Request Buffer	10	
Context Switch from Send_Thread to NCS_send()	25	
Session Overhead Total	108	28 %
Data Transfer Overhead		
Transmitting a 1-Byte Message	274	72 %
Total	383	100 %

Table I shows the timing data with all overhead functions at the transmit side. The major

components of the overhead are: 1) Function call overhead (Entry/Exit) for *NCS_send()* primitive; 2) the overhead incurred by attaching a header for a request message; 3) queuing overhead for this request message; 4) context switching time from *NCS_send()* primitive to *Send_Thread*; 5) the overhead for dequeuing the request message in the *Send_Thread*; 6) message transmission time; 7) the time used for freeing the message structure; and 8) context switching time from *Send_Thread* to *NCS_send()* primitive. These overheads are largely divided into two categories: *session overhead* (1, 2, 3, 4, 5, 7, 8) and *data transfer overhead* (6).

The *session overhead* is the time spent for activities other than actual data transfer (in our case, the overhead incurred by using threads). The *data transfer overhead* is the time spent to transmit a message using the primitives provided by the underlying communication interface. The *session overhead* is constant, regardless of the message size, while the *data transfer overhead* is dependent upon the message size. This overhead involves a per-byte overhead such as data checksumming and data copying.

As we can see from Table I, the *session overhead* is 108 *micro*seconds, which is 28% of the total time to transmit a 1-byte message. Although the *session overhead* will be amortized as the message size increases, it dominates the overhead for small messages. For example, Figure 11 depicts the overhead of NCS implementation relative to the native socket. It is clear from Figure 11 that the overhead relative to the native socket is decreasing and finally becomes negligible as the message size increases. This concludes that the *session overhead* is not the major overhead factor in transmitting large messages, but that it dominates the overhead in transmitting small messages.

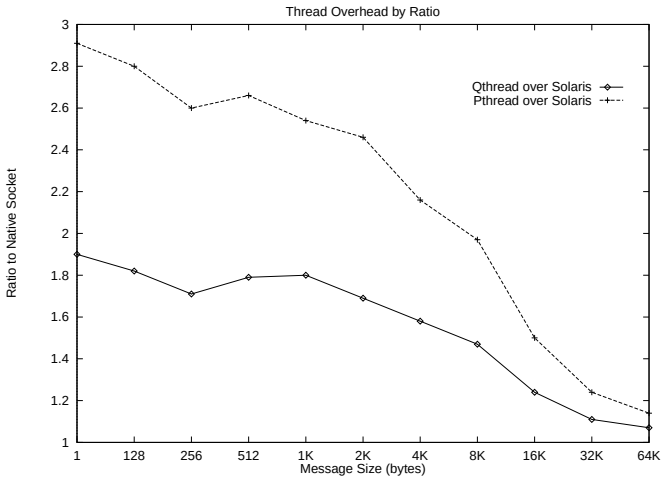

Figure 11: Overhead Ratio to Native Socket

From the experience described above, we decided to provide another version of *NCS_send()*

and *NCS_recv()* primitives, which bypasses all NCS threads (e.g., *control threads* and *data transfer threads*) and transmits or receives directly, using the primitives provided by the underlying communication interface. In this case, all threads can be replaced by procedures. These procedures include flow control, error control, multicasting algorithms, and low-level communication primitives.

4.3 Primitive Performance

In order to compare the performance of point-to-point communication primitives, the roundtrip performance is measured using an echo program. In this echo program, the client transmits a message of proper size which is transmitted back once it is received at the receiver side. The timer starts in the client code before transmitting a message and stops after receiving back the message. The difference in time is used to calculate the roundtrip time of the corresponding message size. The time was averaged over 100 iterations after discarding the best and worst timings.

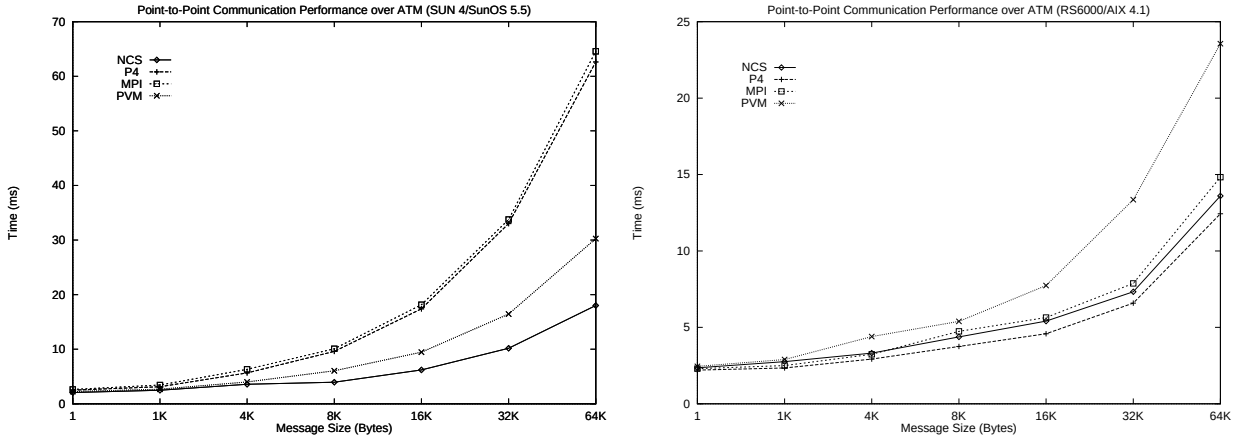

Figure 12: Point-to-Point Communication Performance Over ATM Using Same Platform

Figure 12 shows the performance of send/receive primitives of four message-passing systems for different message sizes up to 64 Kbytes when they are measured using the same computing platform (e.g., SUN-4 to SUN-4 or IBM/RS6000 to IBM/RS6000). As we can see from Figure 12, NCS has the best performance on the SUN-4 platform while p4 has the best performance on the IBM/RS6000 platform. For message sizes smaller than 1 Kbytes, the performance of all four message-passing systems is almost the same but the performance of MPI and p4 on the SUN-4 platform and the performance of PVM on the IBM/RS6000 get worse as the message size gets bigger.

Figure 13 shows the performance of corresponding primitives using the different computing platform (e.g., SUN-4 to IBM/RS6000). In this case, NCS outperforms other message-

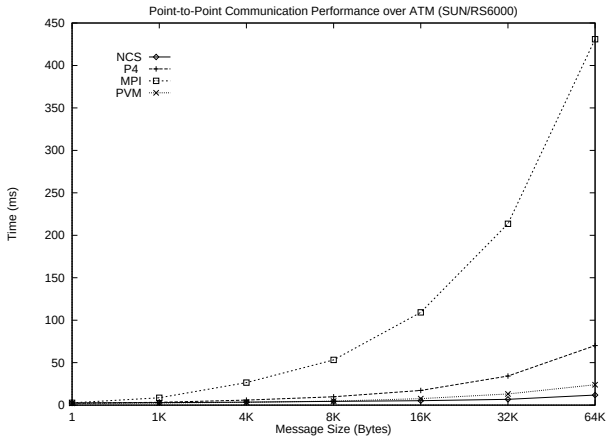

Figure 13: Point-to-Point Communication Performance Over ATM Using Heterogeneous Platform

passing systems. It is worthy to note that the MPI implementation performs very badly as the message size gets bigger and the p4 implementation does not perform well compared to PVM and NCS.

Consequently, it should be noted that the performance of send/receive primitives of each message-passing system varies according to the computing platforms (e.g., hardware or kernel architecture of the operating system) on which the message-passing systems are implemented. NCS shows good performance either on the same computing platform or on the different platform. PVM shows worst performance on the IBM/RS6000 platform but shows comparable performance to NCS both on the SUN-4 platform and on the heterogeneous platform. p4 and MPI show better performance on the IBM/RS6000 platform running AIX 4.1 than they are running both on the SUN-4 platform running SunOS 5.5 and on the heterogeneous platform. This implies that the performance of applications written by using these two message-passing systems over the SUN-4 platform and the heterogeneous environment will be worse than those of other message-passing systems.

5 Conclusion

In this paper, we have outlined the architecture of a high-performance and flexible multithreaded message-passing system that can meet the QOS requirements of a wide range of HPDC applications. Our approach capitalizes on thread-based programming model to overlap computation and communication, and develop a dynamic message-passing environment with separate data and control paths. This leads to a flexible, adaptive message-passing environment that can support multiple flow control, error control, and multicasting al-

gorithms. We also provided the implementation details of how NCS architecture can be applied to provide efficient and flexible point-to-point communication services.

We have evaluated the performance of NCS point-to-point communication primitives and compared that with those of other message-passing systems. The benchmarking results showed that NCS outperforms other message-passing systems.

References

[1] J. Y. Le Boudec, "The Asynchronous Transfer Mode: a tutorial", *Computer Networks and ISDN Systems*, Vol. 24, No. 4, pp. 279–309, 1992.

[2] N. J. Moden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su, "Myrinet: A Gigabit-per-second Local Area Network", *IEEE Micro*, Vol. 15, No. 1, pp. 29–36, February 1995.

[3] IEEE Std 802.3u, "Local and Metropolitan Area Networks: Media Access Control (MAC) Parameters, Physical Layer, Medium Attachment Units, and Repeater for 100 Mb/s Operation, Type 100BASE-T", 1995.

[4] D. Tolmie, and J. Renwick, "HIPPI: Simplicity Yields Success", *IEEE Network*, pp. 28–32, January 1993.

[5] R. Ahuja, S. Keshav, and H. Saran, "Design, Implementation, and Performance Measurement of a Native-Mode ATM Transport Layer (Extended Version)", *IEEE/ACM Transactions on Networking*, Vol. 4, No. 4, pp. 502–515, August 1996.

[6] L. Kleinrock, "The latency/bandwidth tradeoff in gigabit networks", *IEEE Communication Magazine*, Vol. 30, No. 4, pp. 36–40, April 1992.

[7] S. Y. Park, S. Hariri, Y. H. Kim, J. S. Harris and R. Yadav, "NYNET Communication System (NCS): A Multithreaded Message Passing Tool over ATM Network", *Proc. of the 5th International Symposium on High Performance Distributed Computing*, pp. 460–469, August 1996.

[8] S. Y. Park and S. Hariri, "A High Performance Message Passing System for Network of Workstations", *The Journal of Supercomputing*, Vol. 11, No. 2, 1997.

[9] R. Butler and E. Lusk, "Monitors, message, and clusters: The p4 parallel programming system", *Parallel Computing*, Vol. 20, pp. 547–564, April 1994.

[10] V. S. Sunderam, "PVM: A Framework for Parallel Distributed Computing", *Concurrency: Practice and Experience*, Vol. 2, No. 4, pp. 315–340, December 1990.

[11] MPI Forum, "MPI: A Message Passing Interface", *Proc. of Supercomputing '93*, pp. 878–883, November 1993.

[12] J. Flower, and A. Kolawa, "Express is not just a message passing system. Current and future directions in Express", *Journal of Parallel Computing*, Vol. 20, No. 4, pp. 597–614, April 1994.

[13] S. Gillich, and B. Ries, "Flexible, portable performance analysis for PARMACS and MPI", *Proc. of High Performance Computing and Networking: International Conference and Exhibition*, May, 1995.

[14] L. Dorrmann, and M. Herdieckerhoff, "Parallel Processing Performance in a Linda System", *International Conference on Parallel Processing*, pp. 151–158, 1989.

[15] K. P. Birman, R. Cooper, T. A. Joseph, K. P. Kane, F. Schmuck, and M. Wood, "Isis - A Distributed Programming Environment", User's Guide and Reference Manual, Cornell University, June 1990.

[16] R. Renesse, T. Hickey, and K. Birman, "Design and performance of Horus: A lightweight group communications system", Technical Report TR94-1442, Cornell University, 1994.

[17] B. J. Nelson, "Remote Procedure Call", Ph.D thesis, Carnegie-Mellon University, CMU-CS-81-119, 1981.

[18] Werner Almesberger, "Linux ATM API Draft, Version 0.4", EPFL, LRC Technical Document, July, 1996.

[19] W. J. Hymas, H. Stuttgen, D. Chang, S. Sharma, and S. Wise, "ATM Extensions to the Socket Programming Interface in AIX 4.2", http://www.rs6000.ibm.com/resource/technology/atmsocks/atmnew.html.

[20] M. Hall, M. Towfiq, G. Arnold, D. Treadwell, and H. Sanders, "Windows Sockets Version 1.1 Specification", 20 January, 1993.

[21] "Windows Sockets 2 Specification", 22 January, 1996.

[22] A. S. Tanenbaum, *Computer Networks*, Third Edition. Prentice Hall, 1996.