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

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 over Asynchronous Transfer Mode (ATM)-based High Performance Distributed Computing (HPDC) environments. NCS capitalizes on thread-based programming model to overlap computations and communications, and develop a dynamic message-passing environment with separate data and control paths. This leads to a flexible and adaptive message-passing environment that can support multiple flow-control, error-control, and multicasting algorithms.

In this paper we provide an overview of the NCS architecture and present how NCS point-to-point communication services are implemented. We also analyze the overhead incurred by using multithreading and compare the performance of NCS point-to-point communication primitives with those of other message-passing systems such as p4, PVM, and MPI. Benchmarking results indicate that NCS shows comparable performance to other systems for small message sizes but outperforms other systems for large message sizes.

1 Introduction

Current advances in processor technology and the rapid development of high-speed networking technology such as Asynchronous Transfer Mode (ATM) [1] 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. HPDC applications require low-latency and high-throughput communication services. HPDC applications have different Quality of Service (QoS) requirements (e.g., bandwidth/delay requirements, flow/error-control algorithms, etc.) and even one single application has multiple QoS requirements during the course of its execution (e.g., interactive multimedia applications).

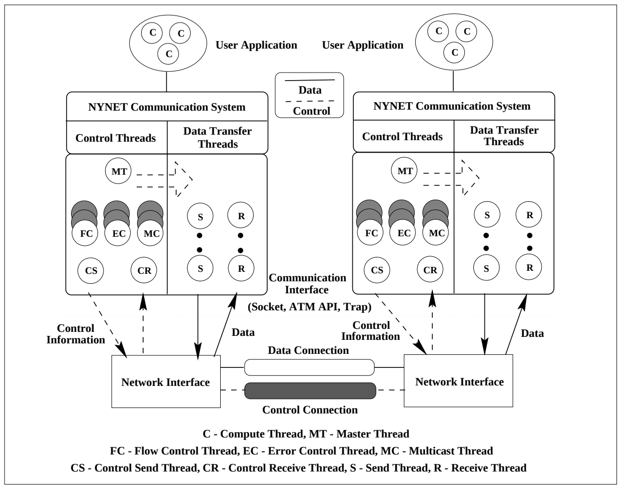
There have been several inter-process communication libraries such as p4 [5], Parallel Virtual Machine (PVM) [6], Message Passing Interface (MPI) [7], Express [8], and PARMACS [9] that simplify process management, inter-process communication, and program debugging in a parallel and distributed computing 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 Communication System (NCS [3] [4]) is a multithreaded message-passing system that provides high performance and flexible communication services over ATM-based HPDC environments. 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 different communication requirements. This paper provides an overview of the NCS architecture and presents how NCS point-to-point communication services are implemented in NCS.

The rest of the paper is organized as follows. Section 2 presents the general architecture of NCS. Section 3 discusses an approach to implementing 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 NCS Overview

In this section we present an overview of the NCS architecture. Additional details about NCS architecture can be found in [3] [4].



User Application
User Application
C
Data
Control
NYNET Communication System
Control Threads
Data Transfer Threads
MT
FC
EC
MC
CS
CR
S
R
Communication Interface
(Socket, ATM API, Trap)
Control Information
Data
Data Connection
Network Interface
Control Connection
C - Compute Thread, MT - Master Thread
FC - Flow Control Thread, EC - Error Control Thread, MC - Multicast Thread
CS - Control Send Thread, CR - Control Receive Thread, S - Send Thread, R - Receive Thread

Figure 1: NCS General Architecture

Figure 1 shows the general architecture of NCS. An NCS application consists of multiple *Compute_Thread*s that include programs to perform the computations of the application. NCS supports both the *host-node* programming model and the Single Program Multiple Data (SPMD) programming model. In both models processes are created at each node by using the *hostfile* that specifies the initial configurations of machines to run NCS applications. After each process is spawned, it creates multiple *Compute_Thread*s according to the computation requirements of the application. The advantage of using a 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.

NCS separates control and data functions by providing two planes (see Figure 1): a *control plane* and a *data plane*. The control plane consists of several threads that implement important control functions (e.g., connection management, flow control, error control) 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 thread*s). The *data transfer thread*s (*Send_Thread* and *Receive_Thread*) in the data plane are spawned on a per-connection basis by the *Master_Thread* to perform only the data transfers associated with a specific connection. Furthermore, 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 functions 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.

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.

NCS provides three application communication interfaces such as *socket* communication interface (SCI), ATM communication interface (ACI), and high-performance interface (HPI) in order to support HPDC applications with different communication requirements. 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. The HPI supports applications that demand low-latency and high-throughput communication services.

3 Point-to-Point Communications in NCS

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. Finally, we describe the NCS approach to managing connections and binding particular communication schemes into a given connection. 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). Each algorithm will be implemented as a thread and 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 2. In this example we assume that each connection is configured with the appropriate error-control algorithm, flow-control algorithm, and communication interface.

1. When *NCS_send()* is invoked at the source *Compute_Thread*, it activates the corresponding *Error_Control_Thread* associated with the sending connection.
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.

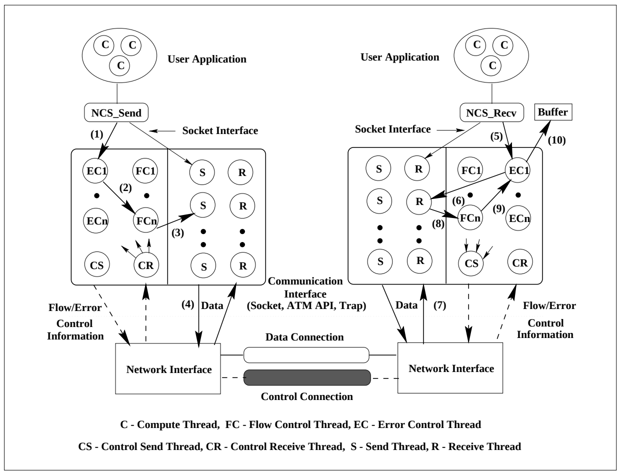

C
C
C
User Application
NCS_Send
Socket Interface
(1)
(2)
(3)
EC1
FC1
ECn
FCn
CS
CR
S
R
Communication Interface (Socket, ATM API, Trap)
Flow/Error Control Information
(4)
Data
Data Connection
Network Interface
Control Connection
User Application
NCS_Recv
Buffer
(5)
(10)
(6)
(8)
(9)
(7)
Data
Flow/Error Control Information
Network Interface
C - Compute Thread, FC - Flow Control Thread, EC - Error Control Thread
CS - Control Send Thread, CR - Control Receive Thread, S - Send Thread, R - Receive Thread

Figure 2: Point-to-Point Communication in the NCS Environment

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, 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

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

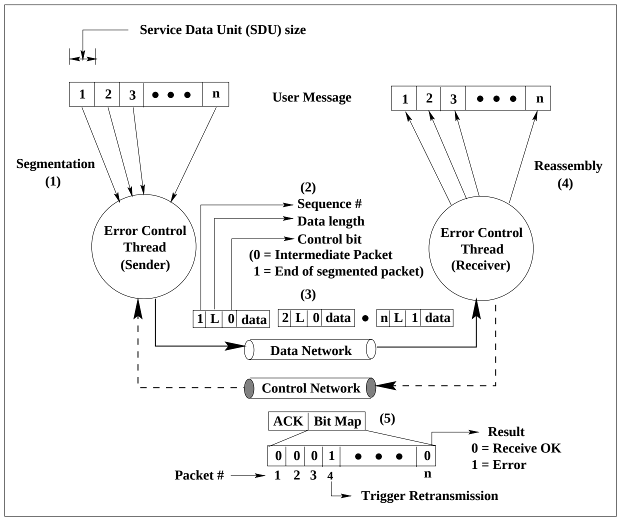


Figure 3: Selective Repeat Error-Control Scheme in the NCS Environment

The default error-control algorithm in NCS is based on *selective repeat* strategy [10], as shown in Figure 3. 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. In case of the last SDU, control bit is set to 1, which activates the *Error Control Thread* at the receiver side to send an Acknowledgment packet to the *Error Control Thread* at the sender side over the control connection. Otherwise, control bit is set to 0 so that the corresponding SDU is transmitted without triggering acknowledgement from the receiver.
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, otherwise, it is set to zero.
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 SDU size is the unit of error control and retransmission in NCS. The SDU size varies from 4 Kbytes to 64 Kbytes and corresponds to the single ATM Adaptation Layer 5 (AAL5) frame (Default SDU size is 4 Kbytes). The reason for this is that some ATM application programming interface (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 [2].

3.3 Flow Control

One of the drawbacks of existing protocols is that the flow-control algorithm is fixed and cannot accommodate a wide range of HPDC applications with different QoS requirements. This occurs because a flow control algorithm that is optimal for one application might not be optimal for another application.

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* flow-control algorithm. Figure 4 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

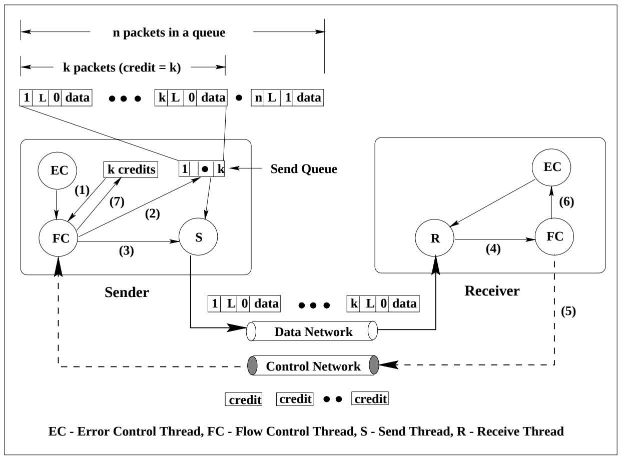


Figure 4: Credit-Based Flow-Control Scheme in the NCS Environment

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.

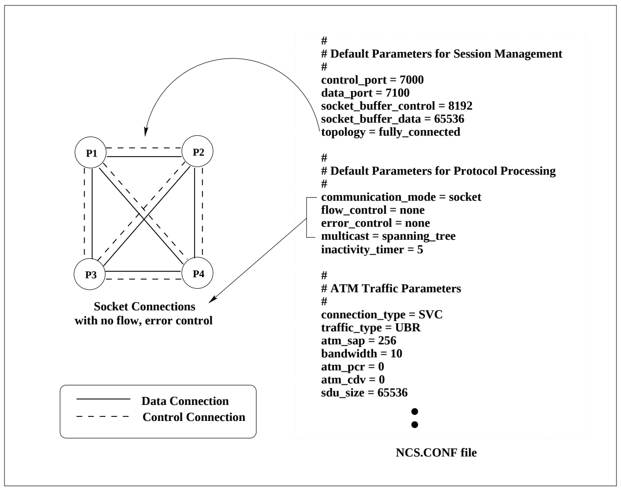
2. The *Flow_Control_Thread* puts the packets into the message queue maintained by the *Send_Thread* based on the number of *credit*s (e.g., in Figure 4, the *credit* is k). This *credit* is an indication of how many packets can be transmitted without any acknowledgment from the receiver.
3. The *Flow_Control_Thread* 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 *credit* for each connection is maintained dynamically. Initially, only small *credit*s are assigned to each connection. 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.

3.4 Connection Management and Algorithm Binding

The NCS provides two schemes to manage connections between processes, *static* management and *dynamic* management.

In the static management scheme users specify an appropriate connection topology between processes (e.g., fully connected, mesh, ring, tree, or random) in the *ncs.conf* file (see Figure 5). The connections between processes are established at initialization time by the *Master_Thread* according to the process topology provided by the users. In this scheme only one data session is established between any two processes. The *ncs.conf* file contains all default parameters used in the NCS environment such as session management parameters (e.g., control/data port number, connection topology, etc.), protocol processing parameters (e.g., communication interface, flow-control algorithm, error-control algorithm, multicasting algorithm, and timer value, etc.), and ATM traffic parameters (e.g., connection type, traffic type, and bandwidth, etc.). All default parameters (e.g., flow-control algorithm, error-control algorithm, multicasting algorithm, and communication interface) defined in the *ncs.conf* file are analyzed by the NCS *parser* program and applied to the static connections by default. Programmers are not allowed to change the default parameters bound to a particular static connection during program execution. This scheme is useful for synchronous parallel applications where communication patterns are predictable and can reduce the overhead associated with connection setup time by establishing all connections before data transfers.



```
#
# Default Parameters for Session Management
#
control_port = 7000
data_port = 7100
socket_buffer_control = 8192
socket_buffer_data = 65536
topology = fully_connected

#
# Default Parameters for Protocol Processing
#
communication_mode = socket
flow_control = none
error_control = none
multicast = spanning_tree
inactivity_timer = 5

#
# ATM Traffic Parameters
#
connection_type = SVC
traffic_type = UBR
atm_sap = 256
bandwidth = 10
atm_pcr = 0
atm_cdv = 0
sdu_size = 65536
        :
```

Figure 5: A Sample *ncs.conf* File

In the dynamic management scheme programmers can establish as many connections as possible between any two processes by using the *NCS_open_session()* primitive during the lifetime of the application. By passing arguments to this primitive, programmers can bind a particular communication scheme and a communication interface into the connection when a session is created. The application that runs over this session uses the selected flow-control algorithm, error-control algorithm, and communication interface when they transfer data using the *NCS_send()* and *NCS_recv()* primitives.

4 Benchmarking Results

This section analyzes the performance and overhead associated with using multithreads to implement NCS point-to-point communication services. First, 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. Next, 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 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 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 1 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 1, 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 6 depicts the overhead of NCS implementation relative to the native socket. It is clear from Figure 6 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.

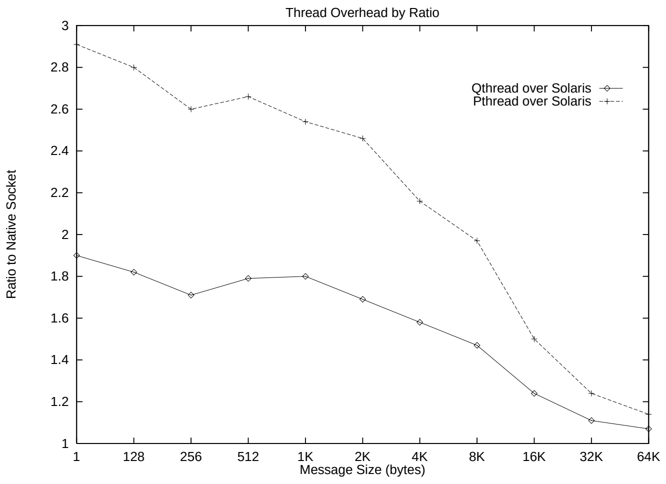


Figure 6: Overhead Ratio to Native Socket

From the experiment 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.

Table 1: 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 %

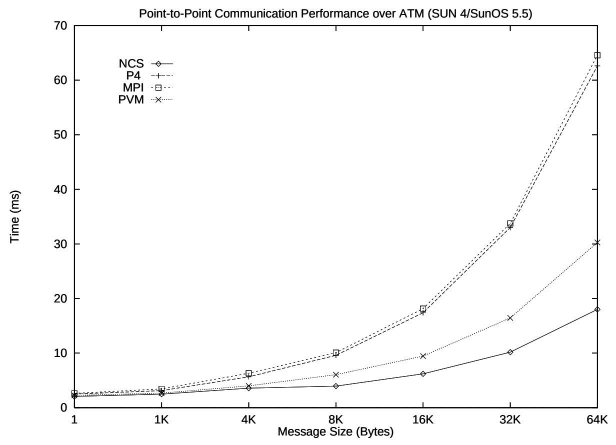


Figure 7: Point-to-Point Communication Performance Over Two SUN-4s

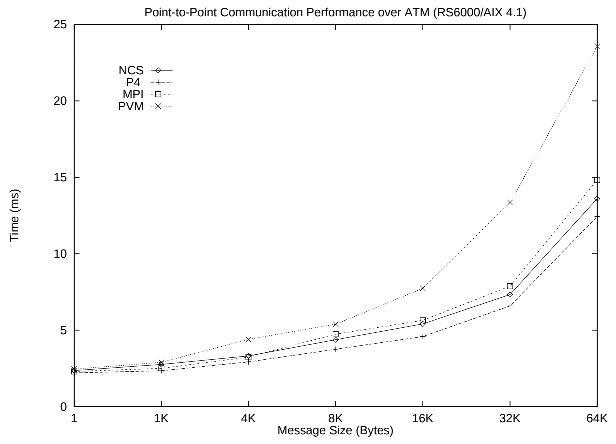


Figure 8: Point-to-Point Communication Performance Over Two IBM-RS6000s

4.2 Primitive Performance

In order to compare the performance of point-to-point communication primitives, the roundtrip latency is measured. Figures 7 and 8 show 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 Figures 7 and 8, 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 9 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-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 heterogeneous platforms. PVM shows worst performance on the IBM/RS6000 platform but shows comparable performance to NCS both on the SUN-4 platform and on

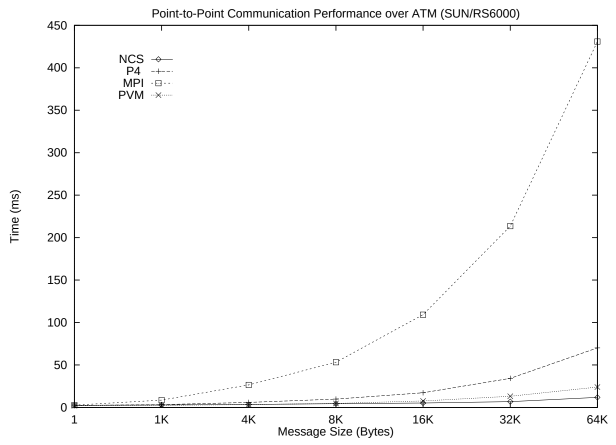


Figure 9: Point-to-Point Communication Performance Over ATM Using Heterogeneous Platform

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 algorithms. 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] 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.

[3] S. Y. Park, J. Lee, and S. Hariri, "A Multithreaded Communication System for ATM-Based High Performance Distributed Computing Environments", *Submitted to IEEE Transactions on Parallel and Distributed Systems*, 1997.

[4] 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.

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

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

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

[8] 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.

[9] 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.

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