An Efficient Group Communication Architecture over ATM Networks

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 tool developed at Syracuse University that provides low-latency and high-throughput communication services over Asynchronous Transfer Mode (ATM)-based high-performance distributed computing (HPDC) environments. NCS provides flexible and scalable group communication services based on dynamic grouping and tree-based multicasting. The NCS architecture, which separates the data and control functions, allows group operations to be implemented efficiently by utilizing the control connections when transferring status information (e.g., topology information, routing information). Furthermore, NCS provides several different algorithms for group communication and allows programmers to select an appropriate algorithm at runtime.

In this paper we overview the general architecture of NCS and present the multicasting services provided by NCS. We analyze and compare the performance of NCS with that of other message-passing tools such as p4, PVM, and MPI in terms of primitive performance and application performance. The benchmark results show that NCS outperforms other message-passing tools for both primitive performance and application performance.

1 Introduction

We are experiencing a rapid deployment of high-performance distributed systems (HPDS) that are typified by a heterogeneous collection of machines with widely differing performance characteristics and are connected by one or more high-speed networks. These systems combine workstations, shared-memory multiprocessors, and distributed-memory multicomputers. The high-speed network technologies used include Asynchronous Transfer Mode (ATM) [1], Myrinet [2], Gigabit Ethernet [3], High Performance Parallel Interface (HIPPI) [4], and wireless technologies. Consequently, the development of high-performance distributed computing (HPDC) applications is a non-trivial task that requires a thorough understanding of the application requirements and architecture, and the communication services provided.

HPDC applications require low-latency and high-throughput communication services comparable to that experienced in a bus-based parallel computer. HPDC applications have different Quality of Service (QoS) requirements and even one single application might have multiple QoS requirements during the course of its execution (e.g., interactive multimedia applications). Furthermore, a significant fraction of the traffic in HPDC applications is multi-point (e.g., video-conferencing, collaborative computing). In order to meet the requirements of a wide variety of HPDC applications, the parallel and distributed software systems should provide high performance and dynamic group communication services. The group communication services provided by traditional message-passing tools such as p4 [11], Parallel Virtual Machine (PVM) [12], Message-Passing Interface (MPI) [13], Express [15], and PARMACS [16] are fixed and thus can not be changed to meet the requirements of different HPDC applications. Furthermore, some message-passing tools such as PVM implement group communication operations by repeatedly calling send routines for each participant, which is computationally expensive and not scalable. There have been several distributed computing software tools specially designed to support group communication services such as Isis [18], Horus [19], Totem [20] and Transis [21]. However, most of them are designed to support spe-

cial functionalities (e.g., fault tolerance, message ordering, virtual synchrony, group partition) rather than to achieve high throughput.

NYNET Communication System (NCS) [7, 8, 9] is a multithreaded message-passing tool for an ATM-based HPDC environment that provides low-latency and high-throughput communication services. NCS capitalizes on a 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. This paper overviews the general architecture of NCS and presents the multicasting services provided by NCS. NCS multicasting services are based on dynamic grouping, where each process can dynamically create, join, or leave a group. NCS uses a binary tree to implement multicasting operations, which is more efficient and scalable than repetitive techniques especially when the number of groups is large. Furthermore, NCS group communication services can be implemented using different group communication algorithms. These algorithms can be selected by the application at runtime.

The rest of the paper is organized as follows. Section 2 outlines the general architecture of NCS. Section 3 discusses an approach to implement the NCS multicasting services. Section 4 analyzes and compares the multicasting performance of NCS with that of other message-passing tools such as p4, PVM, and MPI. Section 5 contains the summary and conclusion of the paper.

2 NCS Overview

In this section we present an overview of the NCS architecture. Additional details about NCS architecture can be found in [9].

Figure 1 shows the general architecture of NCS. An NCS application consists of multiple *Compute_Threads* 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_Threads* 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

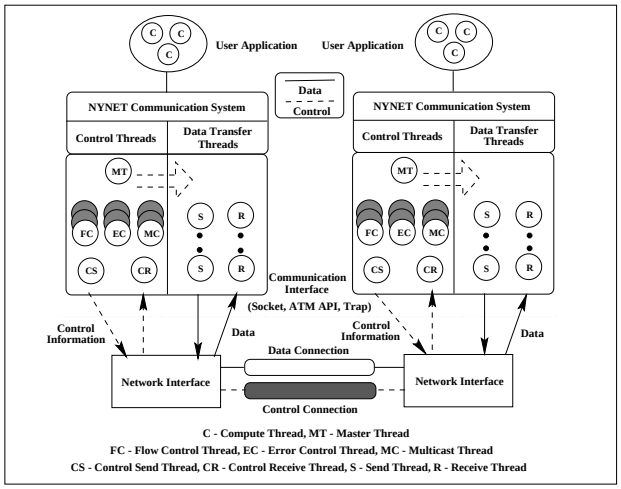

Figure 1: NCS General Architecture

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 threads*). The *data transfer threads* (*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 Multicasting Support in NCS

The implemented NCS multicasting algorithm is based on dynamic grouping, where each NCS process can dynamically create, join, or leave a group during the lifetime of the process. Within each group, there is a single group server that is responsible for intergroup communications and multicasting. The multicasting operation in NCS is implemented by using a binary tree. This approach is more efficient than repetitive techniques especially when the number of groups is large. In addition, the separation of control and data functions facilitates the development of efficient multicasting. For example, when the status of each process has been changed, it can be broadcast promptly to other processes without interfering with data traffic. This allows NCS to prepare most of the information needed to activate multicasting operations (e.g., tree information, group information) in advance before the actual multicasting operations are initiated. This reduces the set-up time (e.g., time to build a tree at runtime) of the multicasting operations and thus improves the performance of NCS group communication services. Other multicasting algorithms can be incorporated into NCS and activated at runtime by user applications without changing the NCS architecture and its supported group communication services.

In what follows we define the NCS group communication primitives and describe the NCS multicasting algorithm to implement these primitives.

3.1 NCS Group Communication Primitives

Figure 2 shows a set of NCS primitives that provide group communication services.

NCS multicasting primitive (*NCS_mcast()*) supports three classes of multicasting operations: (1) *global broadcast*, (2) *local broadcast*, and (3) *global multicast*. The global broadcast is used to transmit messages to all groups defined in the NCS applications. The local broadcast is used to transmit messages to all members within the same group. The global multicast is used to transmit messages to the specified groups. For all three operations, the destination endpoint is not the members, but the group servers. They can be invoked with either a reliable mode or an unreliable mode. The data-type of message (e.g., *char, int, float, double, etc*) and the message type can be specified by providing parameters to the *NCS_mcast()* primitive.

*int NCS_mcast(int mode, char *gname[], NCS_Dtype type, int tag, char *msg, int len);*
- Multicasts a message to the groups specified by gname[].

*int NCS_group_create(char *gname, int com_mode, int fc, int ec, int mc, struct QoS);*
- Creates a group named *gname*. Returns the group identifier.

*int NCS_group_join(char *gname);*
- Joins the group specified by *gname*.

*int NCS_group_destroy(char *gname);*
- Destroys the group specified by *gname*.

*int NCS_group_leave(char *gname);*
- Leaves the group specified by *gname*.

*int NCS_group_num_members(char *gname);*
- Returns the total number of members in the group.

Figure 2: NCS Group Communication Primitives

Users can create a new group by using the *NCS_group_create()* primitive. In this case a particular communication scheme (e.g., error-control algorithm, flow-control algorithm, multicasting algorithm), a particular communication interface (e.g., SCI, ACI, HPI), and ATM QoS parameters can be assigned to the group communication channel (e.g., binary tree). All the new processes that join this group by invoking *NCS_group_join()* primitive use the same communication scheme and communication interface when sending data over the group communication channel. The attributes assigned to this channel cannot be changed by the group members during program execution and they are released when the group is destroyed by using the *NCS_group_destroy()* primitive.

3.2 NCS Multicasting Algorithm

At program startup, a default NCS group called *NCS_GRP* is created, and each NCS process in the

hostfile joins this group automatically (see Figure 3). The *hostfile* is used to specify a list of machines to run NCS applications. The first process specified in the *hostfile* becomes a *master group server* (MGS). Each process that creates a new group becomes a *local group server* (LGS) of that group. The MGS represents all the LGSs and coordinates the group communication operations between these servers. The LGS is responsible for multicasting operations within the local group and maintains the membership information of the local group. A *global multicasting tree* (GMT) is built to connect all the LGSs rooted at the MGS. All the group members within the same group are connected by a *local multicasting tree* (LMT) rooted at the LGS of that group. The MGS and LGSs periodically exchange the status information of each group over the control connections.

Since three classes of multicasting operations (e.g., global broadcast, local broadcast, and global multicasting) are implemented using similar schemes, we will only describe the algorithm for global broadcast. The multicasting algorithm for global broadcast consists of six steps, as shown in Figure 4:

1. When the *Compute_Thread* of a process invokes the *NCS_mcast()* primitive, the *Multicast_Thread* of that process activates the corresponding *Send_Thread* to transmit an actual message to the MGS.

2. The MGS transmits the received message to the other LGSs using its GMT.

3. If the *NCS_mcast()* is invoked with reliable mode, each LGS that received the message sends an acknowledgment back to the MGS along the GMT.

4. An LGS maintains two buffers. The first buffer is used to assemble the messages, which are then transferred to the second buffer. The second buffer is used to retransmit the messages to the members that have not correctly received the messages.

5. Each LGS locally multicasts the message to its group members using its LMT.

6. If the *NCS_mcast()* is invoked with a reliable mode, each member that received the message sends an acknowledgment back to the LGS along the LMT. If there is any group member that has not received a message within the timeout period, the LGS of the group retransmits the message. This reduces the retransmission traffic from the source process.

The pseudo code for this algorithm is presented in Figure 5.

4 Benchmarking Results

In this section we analyze and compare the performance of NCS with that of other message-passing tools such as p4, PVM, and MPI using two levels of performance evaluation [10]: tool performance level (TPL) and application performance level (APL). In TPL we benchmark the performance of the broadcasting primitives provided by each message-passing tool, while in APL we compare the execution time of two applications (e.g., Back-Propagation Neural Network (BPNN) learning algorithm and static voting algorithm).

All experiments have been conducted over six SUN-4 workstations and four IBM RS/6000 workstations interconnected by an IBM 8260 ATM switch and a Cabletron MMAC-Plus ATM switch. In all measurements we used the NCS implementation over SCI. Consequently, the effect of error control and flow control is not considered in these experiments. The socket buffer size was set to 32 Kbytes and the TCP_NODELAY option was enabled. It is reported in [5] that setting those two options improves the socket throughput. For the PVM (Version 3.3.11) applications, we used the PVM Direct mode, where the direct TCP connection is made between two endpoints. The MPICH [14] (Version 1.0.13) was used to benchmark the MPI applications.

4.1 Tool Performance Level (TPL)

Figure 6 compares the performance of broadcasting primitives (e.g., *NCS_mcast()*, *pvm_mcast()*, *p4_broadcast()*, and *MPI_Bcast()*) of four message-passing tools over an ATM network when message sizes vary from 1 byte to 32 Kbytes. The group size varies from two to ten. Since ten heterogeneous workstations (six SUN-4 workstations and four IBM/RS6000 workstations) were used for measuring the timings, the results for the group size up to six represent the characteristics of broadcasting primitives over the SUN-4 platform, while the results for the group sizes of eight and ten represent the characteristics of broadcasting primitives over heterogeneous platforms.

As we can see from Figure 6, the execution time of each broadcasting primitive increases linearly for small message sizes up to 1 Kbytes, while it shows different patterns for large message sizes over 1 Kbytes.

NCS primitive (*NCS_mcast()*) achieved better performance (e.g., about five times faster than p4 and MPI) for various message sizes and group sizes. For

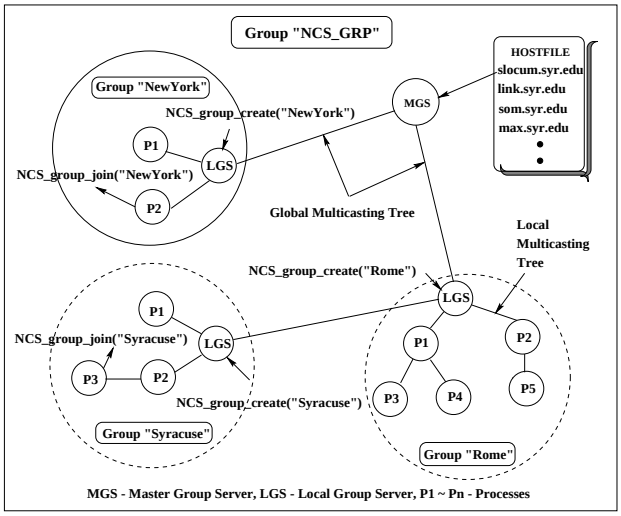

Figure 3: Group Structure in the NCS Environment

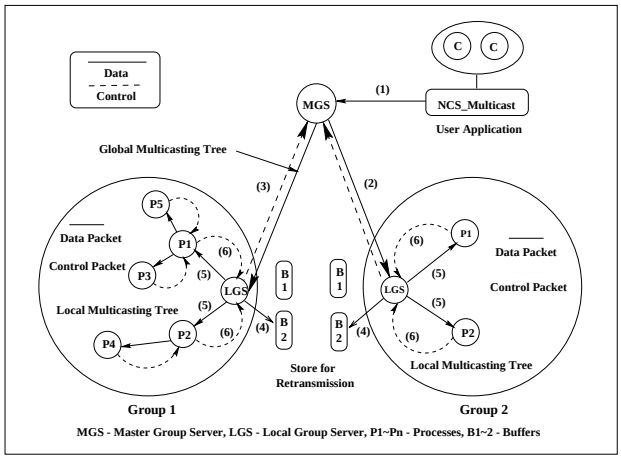

Figure 4: Multicasting in the NCS Environment

Thread *Master Group Server* (MGS)
  repeat Get the requests from other servers or members
    if group creation or destruction is requested **then**
      Update the GMT and send the information to the LGSs over the control path
    else if a *Global Broadcast* is requested
      Send the message to the LGSs along the GMT
      if a reliable multicast is requested **then**
        Check the ACKs from the LGSs and retransmit if necessary
      endif
    endif
  end

Thread *Local Group Server* (LGS)
  repeat Get the requests from other servers or members
    if group creation, destruction, join or leave are notified **then**
      Update the local database (or LMT) and send the information to all the members over the control path
    else if a message received for *Global Broadcast* or *Global Multicast* **then**
      Send the message to all the members along the LMT
      Route the message to other LGSs if necessary
      if a reliable multicast is requested **then**
        Merge the ACKs from the LGSs and send an ACK to its parent
        Check the ACKs from the members and retransmit if necessary
      endif
    else if a *Local Broadcast* is requested
      Send the message to all the members along the LMT
      if a reliable multicast is requested **then**
        Check the ACKs from the members and retransmit if necessary
      endif
    endif
  end

Thread *Multicasting Thread*
  if group creation, destruction, join or leave are notified **then**
    Update the local database for this information
  else if *Global Broadcasting* or *Local Broadcasting* are requested **then**
    Send the message to the MGS (*Global Broadcasting*) or LGS (*Local Broadcasting*)
  else if a *Global Multicasting* is requested **then**
    Setup a spanning tree at runtime and send the message to the LGSs along the new spanning tree
  endif

Figure 5: Pseudo code for the Multicasting Protocol

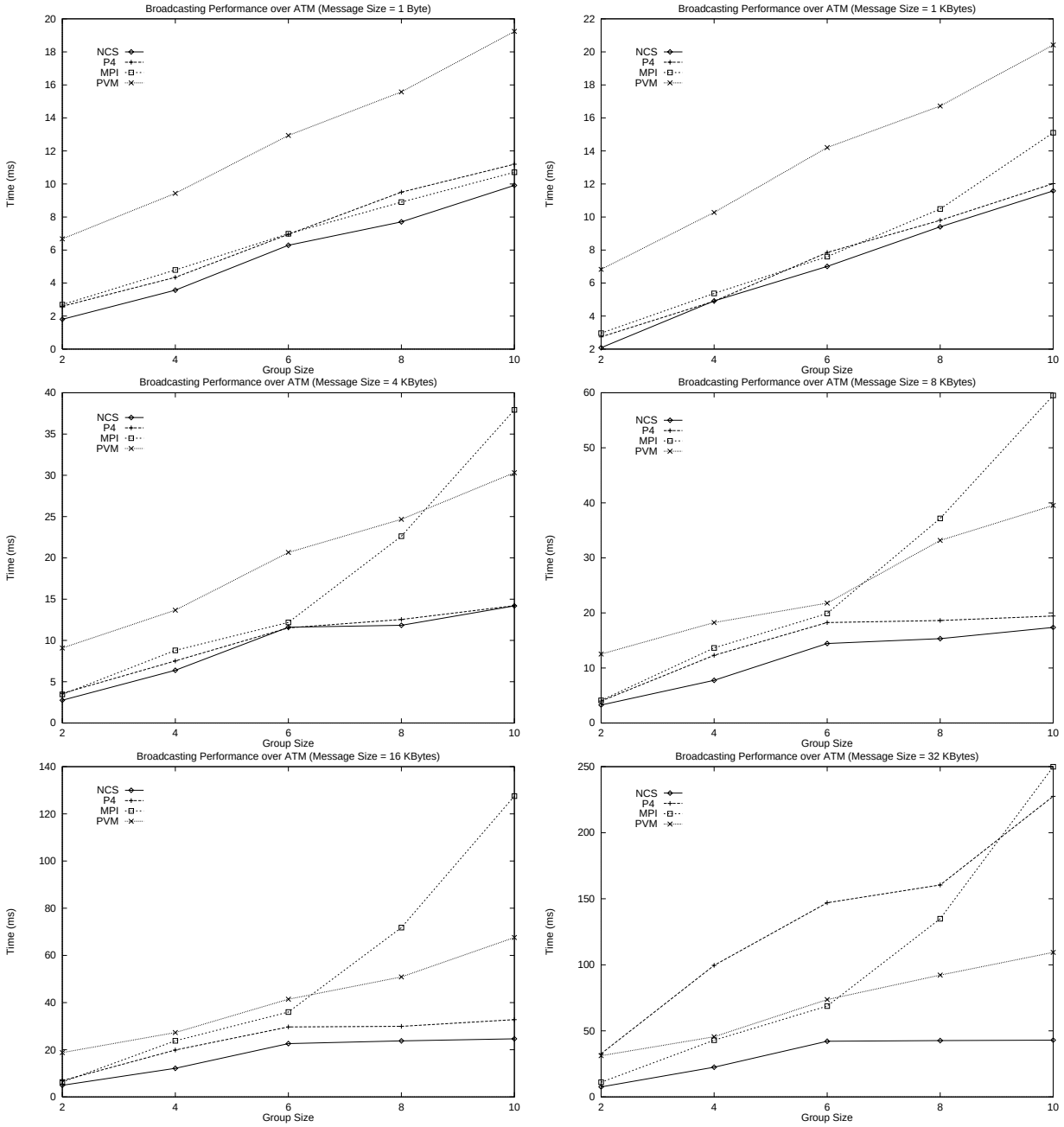

Figure 6: Comparison of Broadcasting Performance over ATM

example, given a message size of 32 Kbytes, the NCS broadcasting time is 42.966 *milli*seconds, while p4, PVM, and MPI took 227.568 *milli*seconds, 109.403 *milli*seconds, and 249.961 *milli*seconds, respectively. Furthermore, *NCS_mcast()* primitive shows almost similar performance for large group sizes as we increase the message size. For a message size of 16 Kbytes, the NCS broadcasting time using six members is 22.596 *milli*seconds and the broadcasting time using ten members is 24.623 *milli*seconds. In the *NCS_mcast()* primitive where most of the information for performing group communications (e.g., setup binary tree, setup routing information) is set up in advance by using the control connections, the start-up time for the broadcasting operations is very small. Also, the tree-based broadcasting scheme improves the performance as the group size gets larger. Consequently, the larger the message size and group size, the better is the performance of NCS when compared to that of other message-passing tools.

The performance of p4 primitive (*p4_broadcast()*) is comparably good except for large message sizes. For message size of 32 Kbytes, p4 performance gets worse rapidly as we increase the group size. One of the reasons for this is that p4 has also low performance for point-to-point communications with large message sizes, as shown in Figures 7 and 8.

The performance of PVM primitive (*pvm_mcast()*) is poor for small message sizes but as the message size and group size increase, its performance improves. In the *pvm_mcast()* where the broadcasting operation is implemented by repeatedly invoking a send primitive, the performance is expected to increase linearly as we increase the group size. Moreover, *pvm_mcast()* constructs a multicasting group internally for every invocation of the primitive, which results in a high start-up time when transmitting small messages as shown in Figure 6 (message size 1 byte and 1 Kbytes).

The MPI primitive (*MPI_Bcast()*) shows comparable performance to NCS and p4 for relatively small message sizes (e.g., up to 1 Kbyte) and small group sizes (e.g., up to 6 members) but its performance degrades drastically for message sizes larger than 4 Kbytes and large group sizes (e.g., over six members).

4.2 Application Performance Level (APL)

In this subsection we compare the performance of NCS with that of other message-passing tools by measuring the execution time of two applications (i.e., BPNN learning algorithm and static voting algorithm) that require intensive group communication services.

BPNN Learning Algorithm

Training BPNN for character recognition is one of the problems in Artificial Intelligence (AI) area that requires intensive group communications. We used a master/slave programming model to parallelize this application, as shown in Figure 9. In this algorithm the *master* process distributes the weight vectors between the input layer and the hidden layer to the *slave* processes. The *slave* processes receive weight vectors from the *master* process and compute the output values of the hidden nodes allocated to them, then transmit those output values back to the *master* process. After the *master* process receives the output values of the hidden nodes from the *slave* processes, it computes the output values of the output nodes, computes mean-squared error, and updates the weights vectors between input layer and hidden layer, and between hidden layer and output layer. These steps continue until the value of the mean-squared error falls under an appropriate value. This application intensively uses the broadcasting primitives when distributing the weight vectors to all the *slave* processes. The BPNN used in this experiment has 100 input nodes, 630 hidden nodes, and 4 output nodes to train 16 input vectors which represent the hexadecimal digits from 0x01 to 0x0F.

Static Voting Algorithm

Replicating data at different locations is a common approach to achieve fault tolerance in distributed computing systems. One well-known technique to manage replicated data is voting mechanisms. The algorithm used in this experiment is based on the static voting scheme proposed by Gifford [22]. In this algorithm (See Figure 10) we assume that there is a file server process in each node that handles *read* and *write* requests for a given file. Each file server process issues arbitrary *read* and *write* requests that were produced randomly using a random number generator. Whenever a server process issues a file access request, it sends a *Lock_Request* message for that file to the local lock manager and broadcasts a *Vote_Request* message to all other server processes. When the server process receives a *Vote_Request* message from other server processes, it sends a *Lock_Request* message for the requested file to the local lock manager. The server process then returns the version number of the replica and the number of votes assigned to the replica to the server process that initiated the *Vote_Request*. Based on the information returned from other server processes, the server process decides if the file access is granted and the file is the latest copy. If the local

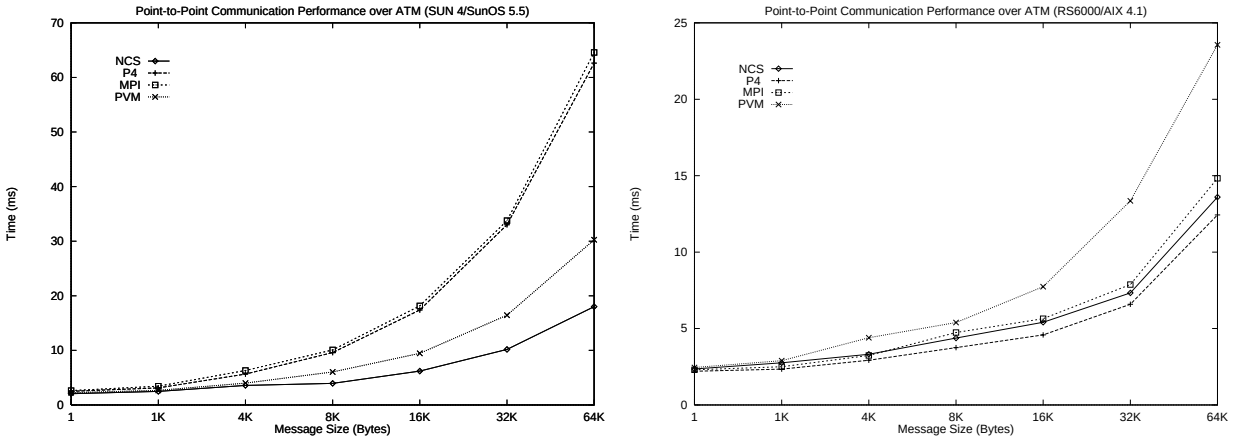

Figure 7: Point-to-Point Communication Performance over ATM Using Same Platform

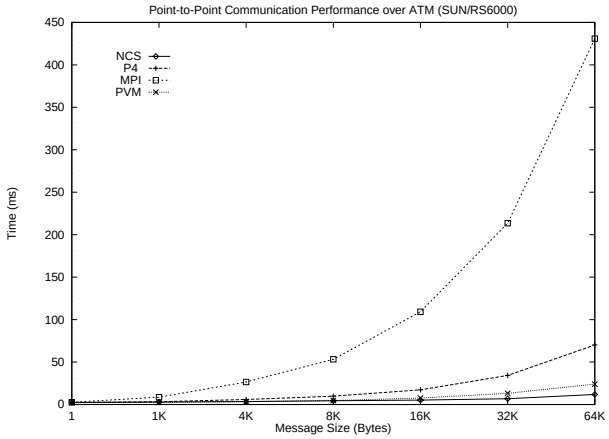

Figure 8: Point-to-Point Communication Performance over ATM Using Heterogeneous Platform

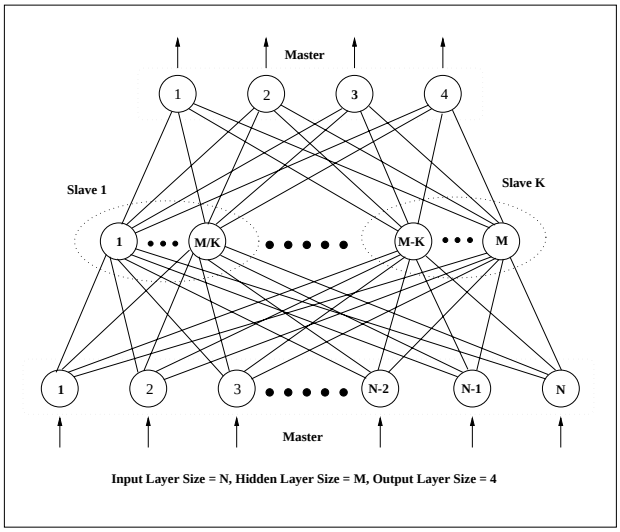

Figure 9: Back-Propagation Neural Network (BPNN) Learning Algorithm

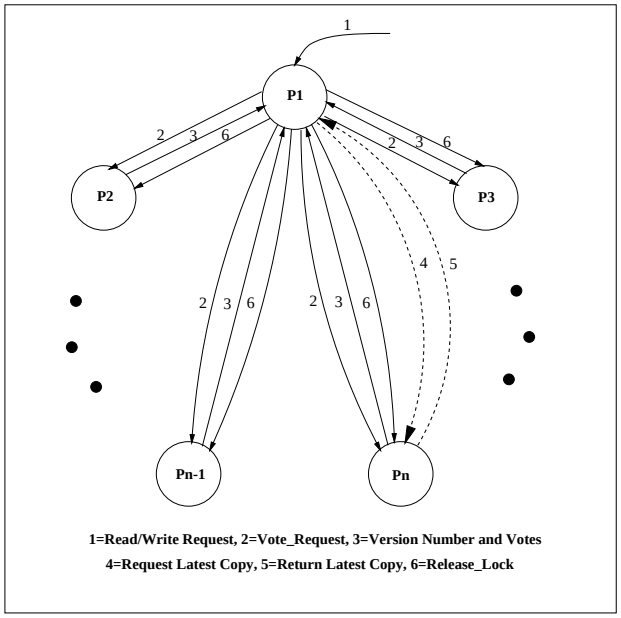

Figure 10: Static Voting Algorithm

copy is different from those replicated at other server processes, it gets the latest copy from other server processes. Finally, the file server process broadcasts a *Release_Lock* message to all other file servers if the file access is granted. In this experiment we assumed that there are 50 different files replicated at each node and each file server process generates 500 *read* or *write* requests for arbitrary files.

Performance Comparison

Figure 11 shows the performance of each message-passing tool to implement these two applications running over four homogeneous workstations (e.g., four SUN-4 workstations running SunOS 5.5 or four IBM/RS6000 workstations running AIX 4.1) and eight heterogeneous workstations (e.g., four SUN-4 workstations and four IBM/RS6000 workstations) interconnected by an ATM network. Due to the restrictions of the MPI broadcasting primitive (*MPI_Bcast()*), we couldn't implement the static voting algorithm using MPI. In MPI all messages broadcast using the *MPI_Bcast()* should be received by other processes using the *MPI_Bcast()* primitive instead of the receive primitive. Furthermore, one of the argument of this primitive represents the rank of the *root* process that initiated the broadcasting operation and this value should be identical on all processes that receive the message. Since the broadcasting operations in static voting algorithm are initiated randomly by different processes, it is difficult to obtain the *root* of the broadcasting operation. Consequently, implementing static voting algorithm using MPI is not straightforward.

As shown in Figure 11, the message-passing tool that has the best performance at TPL also has the best performance at APL. For example, NCS applications outperform other implementations regardless of the platform used. In the BPNN application using eight heterogeneous workstations, the execution time of NCS is 135 seconds, while p4, PVM, and MPI took 1088 seconds, 429 seconds, and 620 seconds, respectively. In the BPNN application where large messages are broadcast repeatedly, the performance improvement is noticeable and it improves further as we increase the group size. In the static voting application where the sizes of the broadcasting messages are small and the communications take place randomly, the performance of NCS is comparable to that of other message-passing tools for small size groups but the performance gap gets wider as we increase the group size. We believe that most of the improvements of NCS are due to overlapping of communications and computations and the use of tree-based broadcasting algorithm.

On the other hand, PVM implementations show better performance than MPI and p4 implementations in heterogeneous environment.

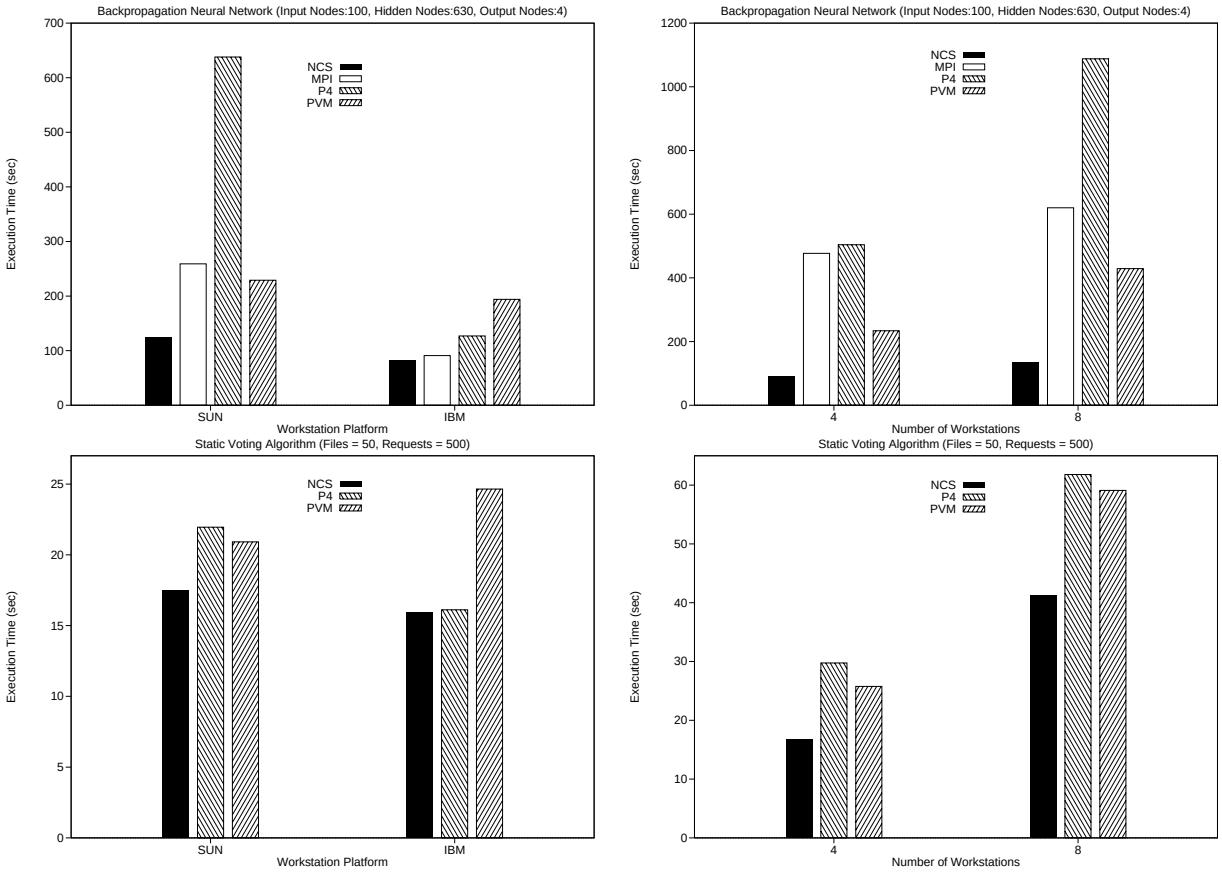

Figure 11: Comparison of Application Performance

5 Conclusion

In this paper we have presented NCS architecture that provides efficient and flexible group communication services over an ATM network. We have evaluated the performance of NCS group communication primitives and applications. The benchmark results showed that NCS outperforms other message-passing tools. It is clear that the NCS novel architecture, which separates the data and control functions and the use of tree-based multicasting scheme played an important role in improving the performance of the communication primitives and applications.

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] Gigabit Ethernet Alliance, "Gigabit Ethernet Overview", White Paper, September 1997.

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

[5] M. Lin, J. Hsieh, D. Du, and J. Thomas, "Distributed Network Computing over Local ATM Networks", *IEEE Journal on Selected Areas in Communications*, Vol. 13, No. 4, pp. 733–747, May 1995.

[6] S. Hariri, S. Y. Park, R. Reddy, M. Subramanyan, R. Yadav, and M. Parashar, "Software Tool Evaluation Methodology", *Proc. of the 15th International Conference on Distributed Computing Systems*, pp. 3–10, May 1995.

[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*, to appear.

[9] 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.

[10] S. Y. Park, J. Lee, and S. Hariri, "An Evaluation Methodology for Parallel/Distributed Software Tools", *Submitted to IEEE Transactions on Parallel and Distributed Systems*, 1997.

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

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

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

[14] B. Gropp, R. Lusk, T. Skjellum, and N. Doss, "Portable MPI Model Implementation", Argonne National Laboratory, July 1994.

[15] 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.

[16] 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.

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

[18] 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.

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

[20] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia and C. A. Lingley-Papadopoulos, "Totem: A Fault-Tolerant Multicast Group Communication System", *Communications of the ACM*, Vol. 39, No. 4, pp. 54-63, 1996.

[21] D. Dolev and D. Malki, "The Transis Approach to High Availability Cluster Communication", *Communications of the ACM*, Vol. 39, No. 4, pp. 64–70, 1996.

[22] D. K. Gifford, "Weighed Voting for Replicated Data", *Proc. of the 7th ACM Symposium on Operating System*, pp. 150–162, December, 1979.

Biographies

Sung-Yong Park received a BS degree in computer science from Sogang University, Korea, in 1987 and MS degree in computer science from Syracuse University, Syracuse, NY in 1994. He is currently working toward the PhD degree in computer science at Syracuse University. From 1987 to 1992, he has worked as a research engineer at LG Electronics (former Goldstar Telecommunication), Korea. From 1993 to 1996, he has worked at the Northeast Parallel Architectures Center (NPAC) at Syracuse University as a research assistant on ISDN and ATM networking. Since 1996, he has been with Computer Applications and Software Engineering Center (CASE) at Syracuse University. His research interests include high performance distributed systems, high speed networks, network computing, and multimedia.

Joohan Lee received BS and MS degrees in computer science from Sogang University, Korea, in 1993 and 1995 respectively, where he worked in artificial intelligence. He is currently pursuing a Ph.D. degree in computer science at Syracuse University. Since 1996, he has worked at High Performance Distributed Computing Laboratory at Computer Applications and Software Engineering Center (CASE) in Syracuse University. His research interests include high performance distributed computing and multimedia.

Salim Hariri is currently an Associate Professor in the Department of Electrical Engineering and Computer Science at Syracuse University. He is the director of the High Performance Distributed Computing Laboratory at Syracuse University (www.atm.syr.edu). He received his Ph.D. in computer engineering from University of Southern California in 1986and an MSc from the Ohio State university in 1982. His current research focuses on high performance distributed computing, high speed networks and protocols, network management, and performance.