5-1-2002

Zero-Sided Communication Challenges in Implementing Time-Based Channels using the MPI/RT Specification

Jothi P. Neelamegam

ZERO-SIDED COMMUNICATION: CHALLENGES IN IMPLEMENTING TIME-BASED CHANNELS USING THE MPI/RT SPECIFICATION

By

Jothi P. Neelamegam

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computer Science
in the Department of Computer Science

Mississippi State, Mississippi

May 2002

ZERO-SIDED COMMUNICATION: CHALLENGES IN IMPLEMENTING TIME-BASED CHANNELS USING THE MPI/RT SPECIFICATION

By

Jothi P. Neelamegam

Approved:

Anthony Skjellum Associate Professor of Computer Science (Major Professor)	Donna S. Reese Associate Professor of Computer Science (Committee Member)
Tomasz Haupt Research Associate Professor of Computer Science (Committee Member)	Julian E. Boggess Associate Professor of Computer Science and Graduate Coordinator of the Department of Computer Science

A. Wayne Bennett
Dean of the College of Engineering

Name: Jothi P. Neelamegam

Date of Degree: May 11, 2002

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Dr. Anthony Skjellum

Title of Study: ZERO-SIDED COMMUNICATION: CHALLENGES IN IMPLEMENTING TIME-BASED CHANNELS USING THE MPI/RT SPECIFICATION

Pages in Study: 86

Candidate for Degree of Master of Science

Distributed real-time applications require support from the underlying middleware to meet the strict requirements for jitter, latency, and bandwidth. While most existing middleware standards such as MPI do not support Quality of Service (QoS), the MPI/RT standard supports QoS in addition to striving for high performance. This thesis presents HARE, the first known implementation of a subset of the MPI/RT 1.1 standard with time-driven QoS support.

This thesis proves the following hypothesis: It is possible to achieve zero-sided communication (a model of communication characterized by the absence of any explicit per-message transfer calls by any of the participating sides) in a real-time environment using a QoS contract between an application and message-passing middleware. Furthermore, it is shown that the performance and predictability of a time-driven task using zero-sided communication is better than that of a best-effort task. The hypothesis is validated through compact MPI/RT application programs that achieve zero-sided communication.

DEDICATION

This thesis is dedicated to my Paatti (tamil, grandmother).

ACKNOWLEDGMENTS

I have been extremely fortunate to have had the opportunity of working under Dr. Anthony Skjellum in the High Performance Computing Lab. Dr. Skjellum has not only been offering quality advice, but has also been a great source of motivation and inspiration for me. This work would have never found the light at the end of the tunnel without his help and encouragement. I would also like to thank my committee members – Dr. Donna Reese and Dr. Tomasz Haupt. My sincere thanks are also due to my fellow researchers – Ecap, Manoj, and Yogi. Most of the ideas in this work have been the result of numerous discussions I have had with them. I would like to explicitly thank Yogi for allowing me to use Figures 3.1 and 3.2. These figures and the accompanying discussion was a collaborative effort with him. I would finally like to thank all my HPC lab mates for all the help they have extended to me over the past two years and for making the lab a great and fun place in which to work.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

LIST OF TABLES

TABLE Page

LIST OF FIGURES

CHAPTER I

INTRODUCTION

1.1 Background

Parallel and distributed applications on a network of workstations have become the natural choice for solving large scientific problems. These systems possess a better cost to performance ratio, higher availability, higher throughput and better scalability as compared to the traditional supercomputers (Baker and Buyya 1999). The trend towards this kind of arrangement is also fueled by the ever-increasing speeds of individual processors and by the improvements in the network technologies used to connect the workstations. The new networking technologies achieve low latency and high bandwidth by bypassing the operating system and removing critical communication overheads (Baker and Buyya 1999).

The message-passing programming model is the most common choice for developing applications in these clusters of workstations. These applications are typically of single program multiple data (SPMD) nature, where the different nodes communicate with each other by passing messages. The Message Passing Interface (MPI) (MPI Forum 1994) is the most common message-passing standard for messaging libraries in these environments. The MPI standard aims at providing a range of efficient functionality without compromising portability (Gropp, Lusk and Skjellum 1999) and has consequently become the de facto message-passing standard in recent times.

The move towards distributed computing has also influenced the real-time systems world. According to Tsai et al. (1996), distributed real-time systems such as command and control systems, space shuttle landing control systems, and nuclear plant control systems are becoming commonplace. By real-time systems, we refer to those systems whose correctness depends on both the logical results of computation and on the timeliness of those results (Stankovic and Ramamritham 1993). Real-time systems can be classified into hard real-time systems and soft real-time systems, depending on the consequences of timing constraint violations (Tsai et al. 1996). Hard real-time tasks are those tasks for which there is no or even negative value in executing the task after its deadline, and such deadline misses can often have disastrous consequences. Examples of hard real-time systems include flight control systems, and chemical-process control systems. Soft real-time tasks retain some diminished value even if they are executed after their deadline (Stankovic and Ramamritham 1990). Here the consequences of missed deadlines are not as disastrous as with the hard real-time systems. Examples of soft real-time systems include airline reservation systems (Tsai et al. 1996).

Real-time systems evidently may possess a mixture of such deadlines as well. Message passing with QoS is evidently needed to support distributed real-time systems. Most of the existing middleware standards such as MPI (MPI Forum 1994) offer little or no guarantees on the services they offer. They offer what are called as "best-effort" services. The absence of a widespread real-time messaging standard had developers of real-time applications to be overly involved with their target platform features to make a reasonable use of the communication features (MPI/RT Forum 2001). This makes porting of such real-time applications difficult. The MPI/RT standard (MPI/RT Forum 2001), the real-time extension of the MPI standard, has

been evolved to solve this problem . The MPI/RT standard tries to address real-time concerns by describing how the various system resources can be marshaled to develop distributed real-time applications around the MPI/RT service framework. MPI/RT supports the concept of early-deferred binding, where the applications specify their resource requirements in advance, but the middleware makes these resources available only when the real-time communication begins. This arrangement allows the middleware to optimize the use of its underlying resources. MPI/RT makes pairwise ordering assumptions through channel abstractions and quality of service (QoS) requirements define the ordering between channels (MPI/RT Forum 2001).

MPI/RT 1.1 (MPI/RT Forum 2001) supports three different real-time programming models – time-driven, event-driven, and priority-driven in addition to the best-effort model in which the application does not explicitly ask for any desired QoS. MPI/RT also supports three different communication patterns, namely two-sided communication, one-sided communication and zero-sided communication. Two-sided communication refers to the most common send-receive kind of communication where all the participating nodes issue explicit message transfer calls. The second model of communication is one-sided communication where only one of the participating sides issues a data transfer call. The last model of sidedness is the zero-sided communication model, characterized by the absence of explicit per-message data transfer calls by any of the participating nodes. The middleware schedules data transfers on behalf of the application based on the channel QoS specifications, which usually translates to transfers at pre-specified times or as responses to pre-specified events. A more complete characterization of the different "sidednesses" of communication is presented in Chapter 3. Zero-sided communication is both interesting to achieve and also has practical significance for

time-based hard real-time systems as outlined in the next section. This thesis is focussed on achieving zero-sided communication.

1.1.1 Advantages of Zero-Sided Communication

The zero-sided communication model appears to be a good choice for hard real-time systems that require complete control of message transfers by virtue of its ability to faciliatate the potential predictable message transfers. Predictability of message transfers is facilitated in this model by:

- Providing a shorter critical path from the state when the message is ready in the application to the state when it is actually transferred.
- Decoupling the scheduling of the application and the communication threads, once the message transfer schedule is established.
- Elimination of resource contention by the middleware by drawing up contention-free schedules at admission-control time.

Chapter 3 discusses zero-sided communication in more detail.

1.2 Hypothesis

The hypothesis for this thesis consists of two components. It is hypothesized that:

- It is possible to implement zero-sided communication in a real-time environment using a QoS contract between an application and message-passing middleware. Alternatively, it is possible to develop middleware that provides mechanisms and features to facilitate zero-sided communication by distributed applications.

- A time-driven task using the zero-sided communication model achieves better performance and predictability than a best-effort task.

The middleware with which this thesis is concerned is the MPI/RT library, which conforms to the MPI/RT 1.1 standard (MPI/RT Forum 2001) specification. Also for this thesis, latency will be used for measuring performance, latency jitters for predictability.

1.3 Motivation

The motivation for this thesis lies in the development of HARE, the first ever prototype implementation of the MPI/RT 1.1 standard with time-driven QoS. HARE is the final component in the PromisQoS project. PromisQoS is an envisioned time-based real-time operating system that supports a prototype implementation of MPI/RT with full QoS support. PromisQoS is being designed and implemented currently at the High Performance Computing Lab in Mississippi State University. The architectural diagram of the project is illustrated in Figure 1.1. TURTLE (Apte et al. 1999) is a real-time scheduler based on RT-Linux and BDM-RT (Chakravarthi 2000) is a low-level real-time messaging layer over Myrinet. Both TURTLE and BDM-RT have been developed by researchers at the High Performance Computing Lab in Mississipi State University and will be explained in the next chapter. HARE has been layered over TURTLE and BDM-RT and logically completes the PromisQoS project.

The motivation for this work stems from both the literature review and the initial experimentation with BDM-RT (Chakravarthi 2000) and TURTLE (Apte et al. 1999) to achieve zero-sided communication.

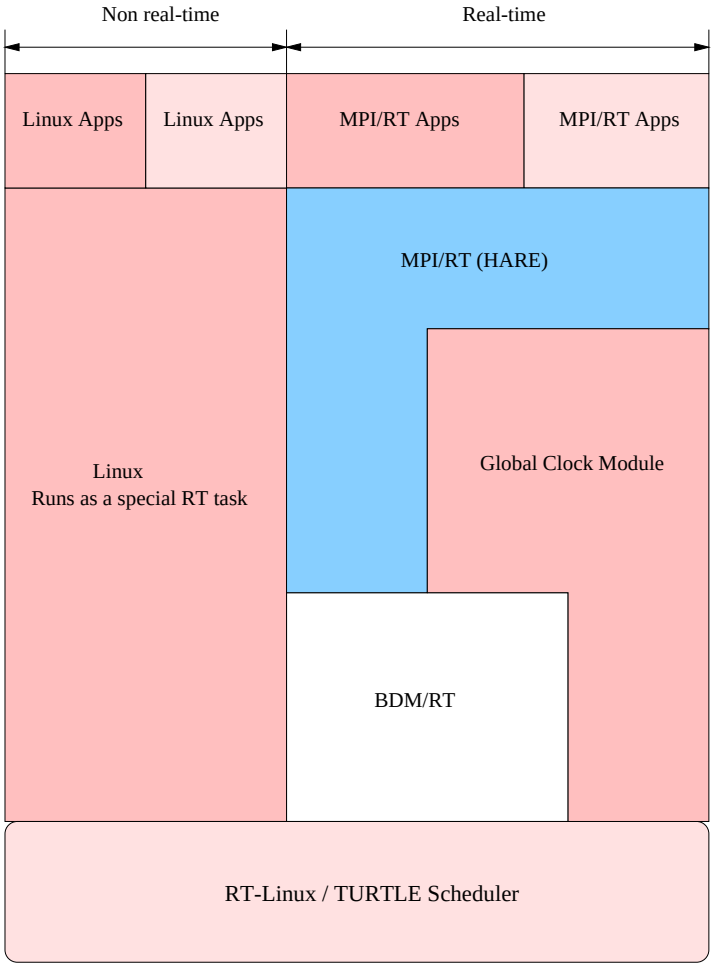


Figure 1.1: PromisQoS Architecture

1.3.1 Feasibility Study with BDM-RT and TURTLE

Based on the literature survey described in Chapter 2, an initial study of BDM-RT (Chakravarthi 2000) and TURTLE (Apte et al. 1999) was made to determine the feasibility of layering time-based MPI/RT channels on BDM-RT and TURTLE to achieve QoS guarantees. The initial study revealed that it is possible to develop and layer HARE over BDM-RT and TURTLE. The details of this study are given in Section 4.2.

1.3.2 Initial Experiments

Initial experiments were conducted in order to perform zero-sided communication using BDM-RT, TURTLE, and the global clock module. The complete details and results of these experiments are available in Apte et al. (2001). Experiments were conducted on a eight-node Linux cluster connected by Myrinet. Each node ran a send channel task and a receive channel task. Messages were transferred across the nodes in a ring fashion. The start times of the channel tasks in all nodes were synchronized to facilitate zero-sided communication. With a calculated stagger (the time interval between the start time of the send task on one node and the start time of the receive task on the previous node), zero-sided communication was achieved. This realization of zero-sided communication with BDM-RT and TURTLE served as an important underlying motivation for the current hypothesis.

1.4 Contributions

The contributions of this thesis are as follows:

1. This thesis establishes a proof-of-concept that zero-sided communication is possible to implement with suitable support from middleware.

2. This work evaluates the suitability of implementing MPI/RT time-based channels using the services of a real-time operating system and a low-level messaging layer.

3. This thesis developed the first known prototype implementation of the MPI/RT 1.1 standard (MPI/RT Forum 2001) with time-driven QoS.

4. This thesis proposes refinements to the MPI/RT QoS specification for possible future inclusion.

5. This thesis developed a global clock module to synchronize up to eight nodes with fine-grain accuracy (of the order of 5μs). This work was accomplished by enhancing the capability of the existing clock module (Chakravarthi et al. 2000).

1.5 Organization

The remainder of this work is organized as follows. Chapter 2 presents the literature review. Chapter 3 characterizes the different sidedness of communication and derives expressions for the message passing times for the different communication models. Chapter 4 discusses the research methodology adopted by this thesis. It gives a brief overview of the implementation and derives the theoretical value for latency in the PromisQoS architecture. Chapter 5 outlines the experiments that were conducted in order to validate the hypothesis and discusses their expected and actual results. Chapter 6 identifies some gray areas in the MPI/RT 1.1 standard and proposes some refinements to the standard. Chapter 7 offers conclusions and suggests future work.

CHAPTER II

LITERATURE REVIEW

As mentioned in the previous chapter, MPI/RT (MPI/RT Forum 2001) attempts to meet the dual, potentially conflicting challenges of achieving high performance and meeting QoS guarantees without compromising portability. In this context, this literature review comprises the study of a few low-level high performance messaging libraries over Myrinet (Boden et al. 1995), FM (Pakin, Lauria and Chien 1995), GM (Myricom 1998), and BDM (Henley et al. 1997), and their real-time counterparts FM-QoS (Connelly and Chien 1997), GM-RT (Zan 2000), and BDM-RT (Chakravarthi 2000). RT-Mach (Lee et al. 1996), a real-time system based on the resource reservation protocol, is also studied. This study reveals the tradeoffs the real-time messaging layers are required to make with regard to high performance when achieving real-time goals. Another focus of the literature review is to identify the low-level messaging layer on which it is most suitable to layer HARE.

The outline for the rest of this chapter is as follows: the following section gives an overview of the Myrinet network interface and the typical software architecture of messaging libraries based on Myrinet. This section is followed by a brief overview of each of the high performance messaging layers with a particular emphasis on identifying those features that enable them to achieve high bandwidth and low latency. A brief description of the real-time messaging layers is provided next. The features that enable these real-time messaging layers to provide guarantees on the services they offer is emphasized. The review then turns to a study of RT-Linux

(Barbanov and Yodaiken 1996) and TURTLE (Apte et al. 1999) and their design features that are likely to be exploited by the implementation of HARE. An overview of the MPI/RT API is then provided, and the literature review concludes with a brief mention of a few other known MPI/RT implementations.

2.1 Myrinet

2.1.1 Hardware Architecture

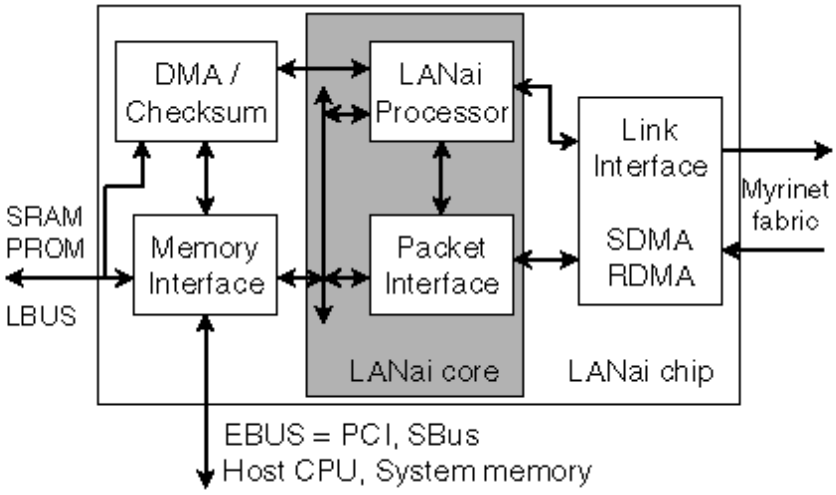


Figure 2.1: LANai Interface (Adopted from Myricom)

Myrinet (Boden et al. 1995) is a high performance System Area Network (SAN) technology often used to interconnect clusters of workstations. Myrinet supports full duplex 1.2 Gigabit/s links, interface ports, and cut-through, crossbar switches that attain latencies as low as 0.5μs. Myrinet provides error control on every link and consequently is reliable, with less than 1 bit error for every 10^{15} bits transmitted. The Myrinet switch networks are highly scalable (to tens of thousands of hosts)

and the more current ones achieve bisection rates in Terabits/s. The network interface comprises the on-board LANai 4.x processor, 1MB of SRAM and three DMA engines. The LANai processor runs a custom built Myrinet Control Program (MCP) that interacts directly with host processes (OS bypass) to achieve low latency communication. The DMA engine handles the following DMA transfers – DMA from the LANai memory to the network, DMA from the network to the LANai memory, and DMA between host and LANai memory.

2.1.2 Software Architecture

Myrinet messaging libraries normally have two components – a host component, running on the host processor, and the MCP, running on the LANai. These components interact with each other using shared memory on the LANai. The MCP monitors incoming and outgoing network data and initiates DMA transfers whenever required. The DMA transfer between the host and the LANai can be initiated by either of the two and this option is handled differently by the different messaging layers. Some of them use both the options, switching from host-initiated to LANai-initiated transfers depending on the message size. Messaging layers on Myrinet like FM, GM, and BDM provide API for data transfers (send/receives) over the network. The protocols used by these messaging layers are significantly lighter than the traditional TCP/IP due to the collapsing of the traditional layers in the OSI reference model. These messaging libraries therefore obtain high performance (high bandwidth and low latency) by using OS bypass techniques – the OS is not involved during DMA initiations by the LANai, ideally allowing overlap of communication and communication and by eliminating redundant copying of messages.

2.2 High-Performance Messaging Layers

2.2.1 FM (Fast Messages)

The FM library was developed by researchers at the University of Illionis, Urbana-Champaign (Pakin, Lauria and Chien 1995). FM was designed to be a low-level messaging library that can be used as a portable high performance building block for higher level messaging systems. One of the primary design goals of FM was to deliver high performance all the way to applications written over middleware that have been layered over FM. To achieve this goal, FM provides reliable delivery, preserved transaction order, overlapping communication and computation, and freedom from communication deadlock. These functions are usually provided by the higher level libraries and consequently these libraries incur penalties in performance as compared to the low-level messaging library they are layered on. By providing these functions in the low-level messaging layer itself, FM tries to propagate the performance gains all the way up to the user application. FM also provides flow control allowing for an efficient management of resources.

In FM, messages are passed as streams; they can be transmitted or received piecewise rather than as atomic units. Every message is a stream and FM streams honor message boundaries. This streaming interface facilitates easy adding or stripping of message headers.

FM achieves low latency by avoiding redundant copies. Because of the use of message streams, messages no longer need to be marshaled into contiguous buffers before transmission and un-marshaled during reception. Higher throughput is obtained by pipelining receives and sends. The receiver can start processing the data even before the completion of the send by the sender. FM uses polling for message

reception, as opposed to the interrupt-driven approach, to reduce context-switching overhead.

2.2.2 GM (Glenn's Messages)

GM (Myricom 1998) is a low-level message based communication system from Myricom that has been designed with the primary objectives of low CPU overhead, low latency, high bandwidth, and portability. Some of the key features of GM include the overlapping of communication and computation by providing user-level, OS-bypass network interface accesses; reliable ordered delivery of messages between hosts, even in the presence of network faults; support of two different priority levels for messages to ensure deadlock-free message delivery; and automatic mapping of Myrinet networks.

The GM communication model is one of "connectionless reliability"(from the application's perspective). The model is connectionless in the sense that GM applications are not required to establish connections between the communication endpoints (ports). GM achieves reliability by actually maintaining reliable connections between each pair of hosts in the network. It then multiplexes the traffic between the ports using these reliable connections.

One of the distinguishing features of GM from other high performance layers on Myrinet like FM and BDM is the way it manages internal resources during message transfers. Each GM application allocates "tokens" for sending or receiving messages of various sizes during program initialization time. During run-time, the application relinquishes a token to GM for every send or receive. GM uses these tokens to organize its internal resources like LANai memory. These tokens are implicitly handed back to the user application by GM on successful completion of sends and receives. These notifications for successful completions are usually in the form of

GM events, and the user program is expected to register a call-back to handle these events. Till recently, GM versions have required polling and have not been thread-safe.

2.2.3 BDM

Bull-Dog MCP (BDM) (Henley et al. 1997) from Mississippi State University is a multi-protocol messaging layer over Myrinet. It provides a variety of protocols spanning various levels of reliability and the order of delivery of messages. The primary design objective of BDM was to provide a simple but capable MCP that would not only meet the high bandwidth and low latency requirements, but can also be be used for experiments with different protocols and different levels of service in support of systems such as MPI (Henley et al. 1997). Providing robust reliability was another design goal.

In order to achieve high performance, BDM toggles between DMA and Programmed I/O depending on the architecture. For example, in the UltraSparc architecture, BDM uses CPU-based *memcpy* to copy messages. On the other hand, in Linux-based clusters, the PCI DMA is much faster than the CPU-based copy. Therefore BDM uses DMA in this architecture.

BDM uses simple queues to manage buffers for incoming and outgoing messages. All these queues are single-producer and single-consumer queues. The host-side buffers in BDM are all allocated as kernel pages and mapped to user memory space at initialization time in order to maximize performance. BDM does not support any multiplexing of data based on tag, leaving that job to the middleware libraries layered on it. This inability to multiplex seems consistent with its goal of achieving high performance. The MCP running on the LANai does all DMA initiations for transferring data between host and LANai memory. The messages are transferred

from the LANai memory to the host memory in a "best-effort" fashion. The aim is to reduce the latency by minimizing the time spent by the messages in LANai. BDM also uses polling to detect completion of message transfers to reduce latency.

Thus the design of BDM appears to be similar to those of the other low-level messaging libraries with regard to the fact that all features are intended to improve performance.

2.3 Real-time Messaging Layers

2.3.1 FM-QoS

The first known implementation of a real-time messaging layer over Myrinet is FM-QoS (Connelly and Chien 1997). FM-QoS tries to address the issue of switch contention in a multi-switch Myrinet network to solve the problem of unpredictability in communication performance. FM-QoS adopts a technique called Feedback-Based Synchronization (FBS), which exploits network flow control information and self-synchronizing schedules, to produce a global view of time. FM-QoS sends periodic messages which are expected to block one another at the Myrinet switch by virtue of sharing the same outgoing port. Based on the latency of the messages because of the blocking, it builds a global view of time. This is important; FM-QoS operates without a global clock. This global notion of time is then used to implement communication schedules with significantly reduced resource conflicts. However, this synchronization achieves control over LANai-to-LANai jitter and not the end-to-end host jitter. To achieve control of end-to-end host latency jitter, explicit control of the PCI bus is essential, and this is not done in FM-QoS.

The overheads for FBS are less than 1% of the total communication traffic and can help FM-QoS achieve latencies of the order of tens of microseconds. While

FM-QoS is the only known real-time messaging layer over Myrinet that can support multiple switches, the complexity of synchronization schedules increases significantly with increasing number of switches. The multi-switch networks will evidently require complex global resource schedule and reservation protocol involving all the switches. Also, the scheduling functionality of FBS is currently implemented in the LANai control program, and needs to be migrated to the hardware for better scalability of FM-QoS.

FM-QoS does not appear a good choice for layering HARE primarily because of its inability to provide guarantees on end-to-end host latency jitter. This capability is absolutely important for any MPI/RT implementation that aims to provide time-driven QoS.

2.3.2 GM-RT

GM-RT (Zan 2000) was an effort to add real-time features to GM by Zan at the High Performance Computating Laboratory at Mississippi State University. This work clearly proved that GM, in its existing form, is not a good choice for use in real-time systems. Though GM supports two different priority levels, it does not provide any isolation of real-time and non real-time tasks with regard to memory management and link scheduling.

GM uses the system calls *malloc* and *kmalloc* for memory allocation, based on user or kernel requirements. The use of *malloc* and *kmalloc* can cause undeterminstic delays and cannot be afforded by a real-time system. Similarly, GM has limited buffers in the LANai memory for its message transfer management. If these buffers are exhausted, GM updates the status of the message as "waiting." This mode of operation is unacceptable for real-time systems.

GM maintains queues for keeping track of connections with pending sends and acknowledgments. GM rotates these queues (both at the port level and at the connection level) for fairness. While this technique is definitely fair for best-effort tasks, it is QoS-insensitive.

GM-RT solves the memory problem mentioned above by implementing a RT-module over the GM device driver. This RT-module preallocates a set of buffers at initialization. The real-time tasks request and obtain memory from this RT-module. This technique eliminates the uncertainties the original GM module encountered. To remove the memory management problem at the NIC, GM-RT uses shadow buffers for the LANai. All DMA transfers are then restricted between the host memory and its corresponding shadow memory.

For increasing predictability with link scheduling, priority-based scheduling is adopted. Real-time tasks are always assigned a higher priority than non real-time tasks. This priority allocation scheme helps in isolation of real-time traffic and non-real-time traffic. FIFO is used for ordering traffic within real-time tasks as GM-RT assumes that real-time packets are already prioritized by the sending sides. GM-RT also maintains additional state information to keep track of the progress of RT transfers and help provide the necessary guarantees.

While GM-RT does provide guarantees on host-to-host latencies, it also seems unsuitable for layering HARE for two reasons. Firstly, GM-RT does not support a global clock. Presence of a global clock is essential for supporting time-driven QoS, a key aspect of this research. Secondly, the existing design of GM-RT does not allow for easy composition with real-time operating system like TURTLE (Apte et al. 1999). This implies that while the user application will be able to obtain bounded

latencies, it might not be able to explicitly specify the required QoS. For these two reasons, GM-RT was not considered further during the development of HARE.

2.3.3 RT-Mach

RT-Mach (Lee et al. 1996) is a real-time operating system developed by researchers at Carnegie Mellon University. RT-Mach is a micro-kernel-based OS, implying that only the most basic of functions are provided in the kernel and all other supporting functions are added on as modules. It uses the resource reservation model (Lee et al. 1996) to provide a predictable and reliable distributed real-time communication environment. RT-Mach also tries to guarantee temporal isolation for real-time tasks. Temporal isolation refers to the capability of the system to prevent one misbehaving task from affecting the other real-time tasks. RT-Mach recognizes that though CPU usage is the most commonly considered shared resource in real-time systems, other resources such as disks and network I/O also need to be scheduled among processes for meeting the QoS requirements of real-time applications. In RT-Mach, every application explicitly states its resource requirements to the resource kernel. The requirements are specified in the form of period P, deadline D and computation time C. The resource kernel decides whether to admit the application or not based on the current demands on the system and its ability to meet the requirements of the new application. Reservations are depleted when the application uses up its allocation C, but are replenished at the end of period P. This process of reservation achieves temporal isolation of processes in RT-Mach.

Though RT-Mach supports parameters similar to those of MPI/RT QoS (including parameters such as deadline and compute time) and seems to be a good choice for layering HARE, there is one drawback that has been highlighted by Chakravarthi (2000). It has been pointed out in this work that the resource

reservation model followed by RT-Mach is not valid in the presence of bus-master capable devices such as the LANai PCI DMA engine. Furthermore, RT-Mach is not as prevalent as Linux in the realm of distributed computing. As this work is aimed at a COTS cluster of workstations, RT-Mach was not considered further for layering HARE.

2.3.4 BDM-RT

BDM-RT (Chakravarthi 2000), like FM-QoS and GM-RT, is a low-level message passing system over Myrinet that has been designed to provide predictability and QoS support along with high bandwidth and low latency. BDM-RT also attempts to provide a good mapping between its messaging system software and the MPI/RT channel abstraction. BDM-RT is also the first known real-time messaging layer over Myrinet to support a fine-grained global clock. The design and development of BDM-RT was done by Chakravarthi at the High Performance Computing Lab in Mississippi State University.

Chakravarthi (2000) identifies the existence of a fundamental dichotomy in the design of high performance and real-time messaging layers and the tradeoffs required in the real-time messaging layers to make them more predictable. BDM-RT uses techniques like "blocking DMA" to achieve deterministic behavior with respect to communication delays by compromising performance. BDM-RT uses PCI DMA transfers for all message transfers, and the initiation and detection of completion of these transfers is done by the MCP program running on the LANai. BDM-RT strives to couple these transfers with the application's CPU schedules. For example, the BDMRT_send function returns only when the message is transferred from the host memory to the LANai memory. It holds the CPU for the duration for which this transfer is taking place. By holding the CPU, BDM-RT ensures that there is

no contention of the PCI bus and other network shared resources. Similarly, the transfer of the message from the receiver's LANai memory to the host memory is done during the call to BDMRT_Recv. Thus, by synchronizing local resource accesses and network protocol processing with local CPU schedules, BDM-RT minimizes contention for locally shared resources.

While predictability is the primary goal of BDM-RT, high performance is an important secondary goal. BDM-RT address this this goal as well, as is indicated by its choice of having the MCP initiate all DMA transfers. When the host is ready to initiate transfers, it simply sets a flag and lets MCP do the actual initiation. This arrangement increases latency because of the time delay between the setting of the flag and its detection by the MCP. However, this delay is shorter as compared to the alternative arrangement where the host initiates the data transfer. The reason is because of the fact that the host has to access DMA registers across the PCI bus, which is a relatively costly operation.

BDM-RT avoids priority inversion by providing tag-based demultiplexing of messages and by embedding protocol processing overhead in the CPU time. To support tag-based demultiplexing, BDM-RT maintains a linked list of received messages at the LANai and does message delivery based on the requested tag. This arrangement allows out-of-order receipts.

By providing bounded MCP response times and ensuring contention-free message transfers between the host and LANai, BDM-RT achieves predictable performances in end-to-end latency. As mentioned earlier, guaranteeing predictability is one of the most important requirements for implementing time-based MPI/RT channels. The other important requirement of time-based channels, the global clock, is also supported by BDM-RT. Thus BDM-RT, in accordance with

its design objectives, seems to be the ideal choice for layering HARE, the prototype implementation of the MPI/RT 1.1 standard subset.

2.4 RT-Linux

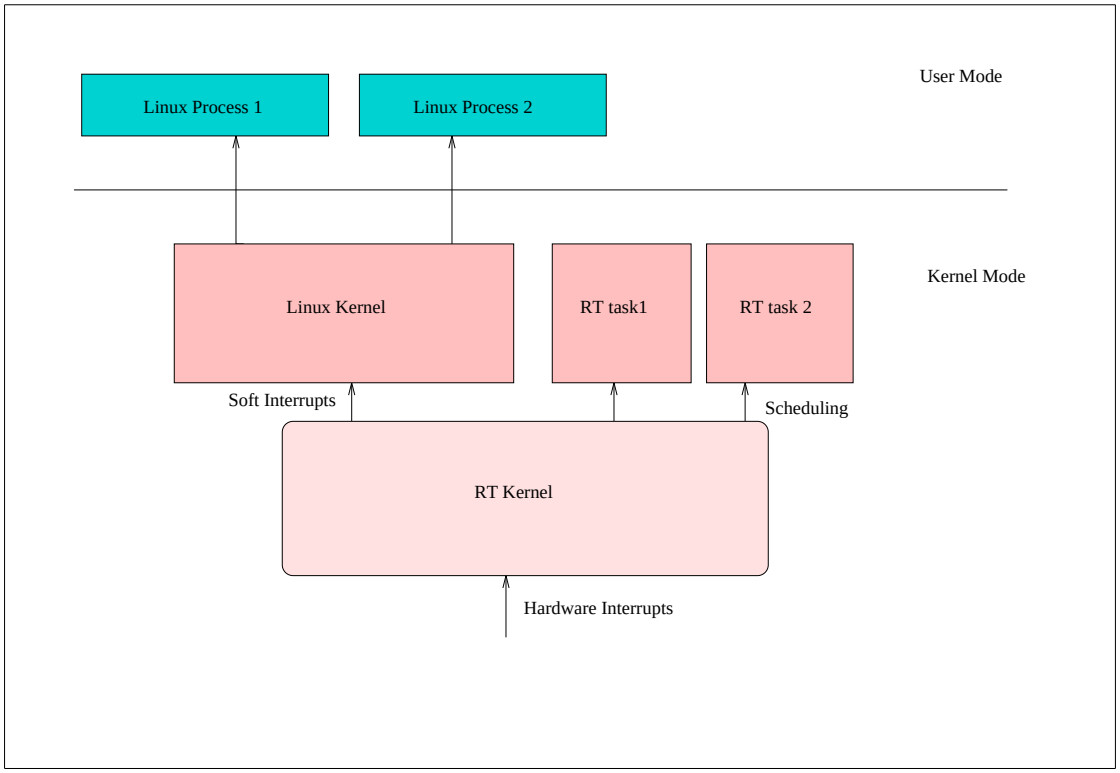


Figure 2.2: RT-Linux Architecture (Adopted from Barbanov and Yodaiken, 1996)

RT-Linux, from New Mexico Institure of Technology (Barbanov and Yodaiken 1996), is a project focused on adding real-time properties to Linux without making many changes to the kernel. Standard Linux as an operating system is unsuitable for real-time systems because of the following reasons:

- High overheads associated with process switching. Overheads on the order of 200μs (measured on our target hardware) are not acceptable in real-time systems.

- Non-preemptive nature of the Linux kernel. If a non real-time task makes a system call, it becomes non-preemptive and can potentially block higher priority real-time tasks.

- Handling of interrupts. The Linux kernel clears interrupts while dealing with critical sections in the code. As a direct consequence, the real-time interrupts cannot be delivered, regardless of their priority, until the kernel finishes its critical section. This results in unpredictable latency delays.

One way of making Linux real-time is to rewrite substantial portions of the code to make it preemptable and have a low interrupt-processing latency. RT-Linux avoids this tedious approach and adopts a novel technique to make Linux real-time OS.

RT-Linux introduces a real-time kernel that controls scheduling of real-time tasks while Linux itself is made a real-time task with the lowest priority. Linux is completely unaware of the existence of this RT-kernel. The architecture as reproduced from Barbanov and Yodaiken (1996) is shown in the Figure 2.2. This architecture solves each of the above mentioned problems:

- The RT-kernel is itself non-preemptable. However it is light weight and the process switches take less than 20μs (in our target hardware).

- The problem of interrupts is solved by simulating Linux interrupt-related routines by the RT-kernel. All interrupts are caught by the RT-kernel and handed over to Linux whenever Linux is scheduled to run. If Linux disables

interrupts, RT-kernel still intercepts interrupts, but retains them and hands them over to Linux only when Linux has re-enabled interrupts.

RT-Linux has its disadvantages, too. The foremost drawback is that the real-time tasks need to run at kernel-privilege level in order to gain direct access to hardware. There can be no user-level real-time tasks. Secondly, these real-time tasks cannot access Linux system calls as such calls could cause inconsistencies.

The real-time tasks of RT-Linux communicate with other Linux tasks using RT-FIFOs. RT-Linux supports two scheduling algorithms – fixed-priority-based scheduler and the Earliest-Deadline-First scheduler (EDF). TURTLE, described in the next section, is a variant of RT-Linux that supports an EDF like scheduler.

2.5 TURTLE

TURTLE (Apte et al. 1999) is a variant of RT-Linux developed by researchers at the High Performance Computing Lab of Mississippi State University. It adopts the reservation abstraction in scheduling policy and uses the "critical deadline first" algorithm (Apte et al. 1999) for its scheduling. In TURTLE, as in EDF, the periodic hard real-time tasks are represented by five parameters:

- C_i - The computation time for the task,
- P_i - The period of the task,
- D_i - The deadline of the task,
- S_i - The start time of the task, and
- E_i - The end time of the task.

The "critical deadline first" algorithm is superior to the EDF, as explained below. One of the problems faced by the EDF algorithm is its inability to curtail misbehaving and greedy tasks. Consider the case where there are two tasks, t_1 and t_2 with

deadlines D_1 and D_2, and computation times C_1 and C_2. If $D_1 < D_2$, EDF will schedule t_1. If after running for C_1, t_1 has still not completed its processing, EDF would continue to schedule t_1 as its deadline is still the earliest. As a direct consequence of the misbehavior on the part of t_1, t_2 might miss its deadline. In the TURTLE environment, all tasks register their five parameters with itself. Every time a task has run for its computation time C, TURTLE automatically updates its deadline to the next period. This is also called the critical deadline. TURTLE is in effect, an "earliest-critical-deadline-first" scheduler. By scheduling the earliest critical deadline first, the scheduler would schedule task t_2 in the above example after t_1 has completed executing for C_1. This kind of policing of misbehaving tasks is also called /emphtemporal isolation. While TURTLE allows for temporal isolation, it is flexible with jobs that declare themselves as greedy. When there is no contention for resources, the scheduler does not advance the critical deadline.

The other problem faced by EDF is its inability to determine deadline misses before the actual deadline. In other words, EDF knows that a task has missed its deadline only when it tries to schedule the task and finds that its deadline has passed. Most systems would want to detect deadline misses as early as possible so they can take any possible corrective action and minimize the damage caused by the missed deadline. TURTLE detects possible deadline misses before running the task by virtue of maintaining critical start times. Critical start time is the time at which the task has zero laxity. By monitoring the critical start times at every scheduling instant, TURTLE can recognize in advance all the tasks that would miss deadlines. Suitable handlers can then handle these situations. Critical start time is also used to avoid unnecessary context switches by giving a task exclusive CPU time whenever its critical start time is close to the current clock time (a heuristic).

TURTLE, like other deadline-driven scheduling algorithms, is optimal in the sense that if a set of tasks can be scheduled by any algorithm, it can be scheduled by the deadline driven scheduling algorithm (Liu and Layland 1973).

In TURTLE, Linux is registered as a greedy task that is guaranteed at least 1ms CPU time every 10ms of real-time. "Greedy task" implies that Linux runs on all other time instants when no other real-time task is ready to run on CPU. As mentioned earlier, TURTLE does not advance the critical deadlines of these greedy tasks in the absence of any resource contentions. In its current form, TURTLE allows Linux as the only greedy task. This means that all best-effort tasks should be Linux tasks and can be run only when Linux is scheduled to run by TURTLE. Linux itself schedules those using its Unix-style schedule (simple priority).

2.6 MPI/RT API

The MPI/RT API (MPI/RT Forum 2001) has been designed using object oriented principles. The standard does not restrict implementations to any particular set of languages, though it currently supports bindings only for C and C++. All classes inherit attributes and methods from their ancestors. The hierarchy is rooted at MPIRT_Object. Only the objects of leaf classes can be instantiated by the user program. The complete hierarchy, as reproduced from MPI/RT Forum (2001), is shown in Figure 2.3.

The MPI/RT objects that are descendents of the MPIRT_Commitable class reflect *deferred early binding*. These objects participate in the admission test in the MPIRT_Commit function. The attributes of these objects cannot be altered after the commit operation until they are decommissioned by the MPIRT_Uncommit function.

MPIRT_Cset and MPIRT_Cvector_base are two container classes used to store references to MPI/RT Objects. The Cset class stores unordered, variable-sized collection of references while the Cvector_base is an abstraction of an array of references to MPI/RT objects.

MPI/RT works to eliminate unpredictability resulting from dynamic memory allocation by defining objects for buffer management. MPIRT_Buffers are used as storage for messages and also as storage for other intermediate storage operations. All the standard datatypes are supported by the standard, including integer, floating point, double precision, logical, character, byte, and time. MPI/RT Buffer Iterators provide mechanisms to manage the order in which the buffers are accessed during message-passing operations. The common buffer iterator policies include unordered, FIFO, LIFO, and sorted user-selectable policies. Effectively, predefined finite state machines are supported at each endpoint for high performance and flexibility.

MPI/RT uses the channel abstraction for message transfers. Channels are defined by the MPI/RT Forum (2001) as "unidirectional, logical conduits through which data travels from one process to another with a particular QoS." The MPIRT_Ptchannel class represents the point-to-point channel. There are two buffer iterators associated with each channel endpoint – an input buffer iterator and an output buffer iterator. These iterators provide and accept the buffers for the channel communication. As mentioned earlier, MPI/RT channels facilitate three kinds of 'sidedness' in communication – two-sided, one-sided and zero-sided. For a send, the application inserts a buffer into the input buffer iterator and calls MPIRT_Start to initiate the transfer. For two-sided communication, all the participating nodes call MPIRT_Start. For one-sided communication, only one node calls the MPIRT_Start function; all the other nodes call MPIRT_Activate. In the activated state, the channel

consumes buffers from the input buffer iterator and transfers them to the output buffer iterator, based on the QoS specified for that channel.

According to MPI/RT Forum (2001), the MPI/RT QoS specifications are the most important distinction between MPI/RT and other high performance message-passing interfaces like MPI (MPI Forum 1994). The QoS specification also follows deferred early binding and the parameters are defined for the different MPI/RT objects. The QoS constraints and the QoS requirements make up the QoS specification. The former are the constraints for the start and finish of an operation and the latter specifies the requirements for an operation on the object. As mentioned earlier, the MPI/RT standard supports three different QoS models for channels: time-driven, event-driven, and priority-driven. In the time-driven model, the QoS parameters are the start time and deadline for the message transfer operation. Another important parameter is the period, the time interval between successive initiation of message transfer calls. The principal aim of the time-driven approach is to provide sufficient control over the environment to the user application so that it can explicitly schedule its message-passing activities and resource usage (MPI/RT Forum 2001). The event-driven model is aimed at providing user applications control over the runtime environment with scheduling of communication and computation activities and their resource usage (MPI/RT Forum 2001). An application specifies events with triggers and receptors to control resource usage. Process and message priorities (channels have fixed priority) are used to meet timing specifications in the the priority-driven model.

This thesis is primarily interested in providing the time-driven QoS functionality.

2.7 Other MPI/RT Implementations

Currently four other implementations of the MPI/RT standard are known to exist – MPIRT-RACE and MPIRT-CSPI, both developed by MPI Software Technology Inc (MSTI 2001). Sky Computers evidently also developed its own implementation. Currently, a fourth MPI/RT implementation, also by MPI Software Technology Inc., MPIRT-Radstone is in progress. The author of this thesis is a part of the development team involved in this implementation. However, none of these implementations support QoS and are best-effort implementations only.

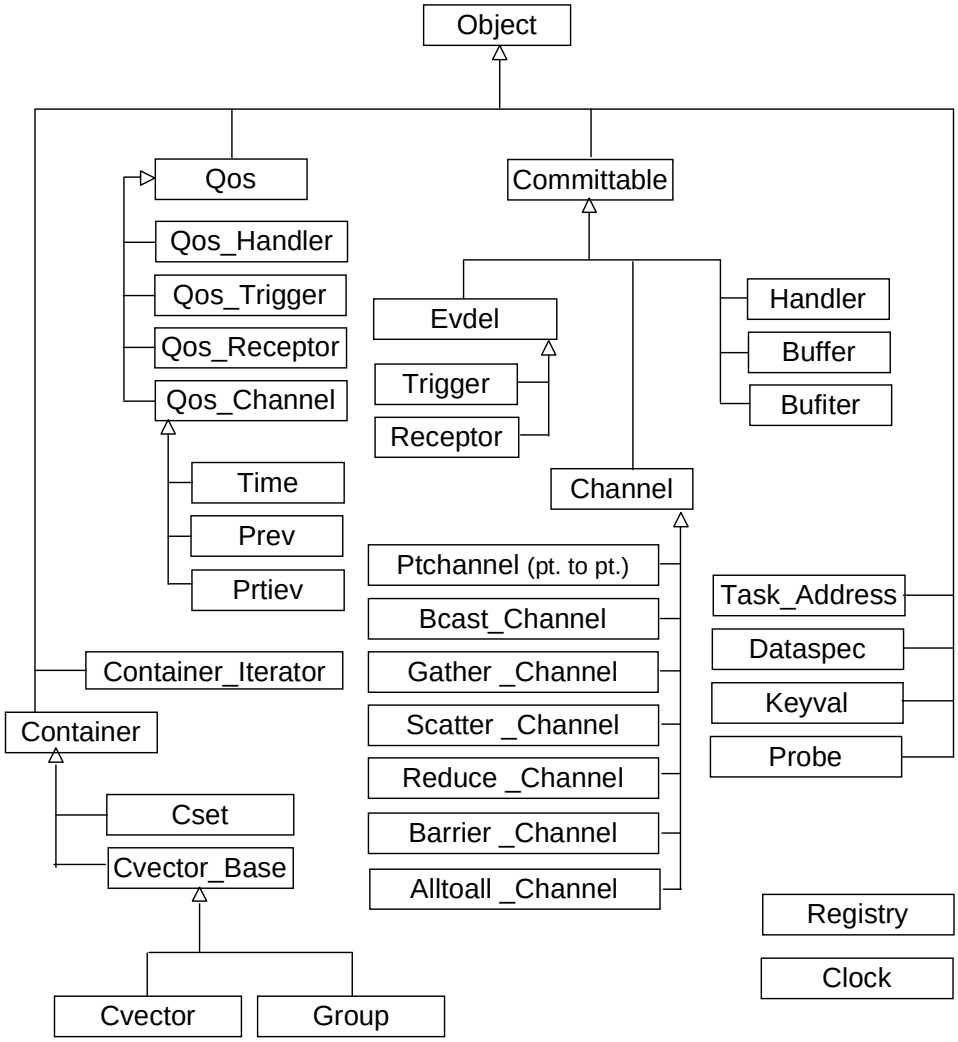


Figure 2.3: MPI/RT Hierarchy (Adopted from the MPI/RT 1.1 Standard)

CHAPTER III

SIDEDNESS IN COMMUNICATION

The MPI/RT standard (MPI/RT Forum 2001) identifies three types of 'sidedness' in application message passing – two-sided, one-sided and zero-sided communication. The MPI/RT standard refers sidedness as "the type of rendezvous (or lack thereof) between sending and receiving processes in a communication." This chapter will briefly discuss each of these sidedness and will strive to quantitatively derive the total time taken for a simple message transfer. This chapter will also discuss realizing these sidedness using the different real-time programming models.

3.1 Two-Sided Communication

Two-sided communication is the type of communication in which applications on both sides fo transfer issue explicit message transfer calls for each data exchange. For a collective operation, all the participating nodes issue message transfer calls for each exchange. This is the most commonly used type of message passing paradigm and is also known as the send-receive paradigm. There is an explicit rendezvous between processes at communication end points. Typically, two-sided communication involves a three-way handshake between the sender and receiver for flow control (atleast for large transfers). After the sender application makes the data transfer call, the sender middleware initiates the transfer mechanism by sending a request-to-send (RTS) packet to the receiver middleware. Among other things, RTS packets usually contain the requested message size. The receiver middleware, on receipt of this RTS

packet, makes the necessary arrangements for receiving the data. Arrangements might include allocating buffers and setting up control parameters. The allocated buffer can be either a middleware buffer or the receive application's buffer itself. Once the preparation for data reception is done, the receiver middleware sends the clear-to-send (CTS) packet back to the sender. If the data transfer is expected to occur through DMA, the address of the destination buffer might also be transferred as a part of the CTS. The final leg of the handshake occurs when the sender does the actual data send on receipt of the CTS. In the above discussion, "application" refers to any layer of level 4+ (transport layer and above) in the OSI reference model, while middleware refers to level 4- (transport layer and below). The exact delineation of transport duties is fuzzy, with some systems having applications do part of the work for flow control and/or reliability, and others are strictly adherent to the ISO OSI reference model.

3.1.1 Message Transfer Time

Figure 3.1 illustrates the components involved during the transfer using the two-sided communication model. Here, T_{MP} denotes the total time taken to transmit the message. T_{MP} is the end-to-end transmission time; it is the interval from when the sender application is ready to send the message to the time at which the receiver application is notified of the receipt. T_{WMLP} is the time difference between the instants when the send call is made by the application and the instant when the middleware begins processing of the message. Various parameters could form a part of this delay component – context switches, protocol processing times, and the waiting time for resources to become available. T_{RTS} and T_{CTS} are the time taken to transmit, receive, and process the control messages. T_{WSYNCH} is the difference in time from when the middleware at the receiving side has the message in its buffer to

when the middleware is able to copy the message to the application's buffer. T_{COPY} is the time to copy the message from the middleware's buffer to the application's buffer. T_{NOTIFY} is the time taken by the middleware to notify the application and T_X is the time for actual transmission of the message across the network. The aim is to minimize T_{MP}, the message-passing latency.

T_{MP} varies with the actual order of send and receive calls and with the middleware implementation. Four different variations are evidently possible. In the first scenario, the application calls send before the corresponding receive has been posted. In this scenario, the receive-side middleware will typically buffer the data in one of its temporary buffers. When the application calls receive, it simply copies the data from its temporary buffer to the user application buffer. In most real-world messaging libraries, short messages, those for which the T_{COPY} component is relatively small, are handled this way for better performance. Overall,

$$T_{MP} \doteq T_{WMLP} + T_{RTS} + T_{CTS} + T_{WSYNCH} + T_{COPY} + T_{NOTIFY}. \tag{3.1}$$

The second scenario is when the middleware is either unable or unwilling to buffer the messages. This scenario is typically the case for those messages where the T_{COPY} is a large fraction of T_{MP} in Equation 3.1. Here, the receive side middleware sends the CTS only after the receive application makes the receive call. Here,

$$T_{MP} \doteq T_{WMLP} + T_{RTS} + T_{CTS} + T'_{WSYNCH} + T_X + T_{NOTIFY}, \tag{3.2}$$

Where,

$$T_{COPY} \geq T_X \text{ and} \tag{3.3}$$

$$T'_{WSYNCH} \geq T_{WSYNCH} \tag{3.4}$$

$$\text{but } T'_{WSYNCH} + T_X \leq T_{WSYNCH} + T_{COPY}. \tag{3.5}$$

The third scenario occurs when the send and receive calls are synchronized so well that T_{WSYNCH} is essentially zero. In other words, the receiver does the preparation for data reception at the same time that the sender is doing its protocol processing and sending RTS. That is, T_{WSYNCH} is subsumed by $T_{WMLP} + T_{RTS}$. As soon as the RTS is received, the CTS is sent because all the preparations have been done beforehand. Eliminating T_{WSYNCH} from Equation 3.2 yields:

$$T_{MP} \doteq T_{WMLP} + T_{RTS} + T_{CTS} + T_X + T_{NOTIFY}. \tag{3.6}$$

The last scenario is when the receive application has already made the receive call before the sender application issues send. Here, the receiver middleware can proactively send its readiness information to the sender middleware, eliminating the need for a three-way-handshake. Here, Equation 3.6 further simplifies to the following:

$$T_{MP} \doteq T_{WMLP} + T_X + T_{NOTIFY}. \tag{3.7}$$

3.2 One-Sided Communication

In one-sided communication, only one side of the application is allowed to make an explicit data transfer call. The common models of one-sided communication

are the put and get (also called push/pull). For one-sided communication to be possible, the receiver application must ensure that the buffer for data reception is made available before the data transfer is initiated by the sender. This obviates the need for any explicit flow control. However, there might still be control message exchanges between the sender and receiver middleware or underlying hardware for the communication of buffer addresses. Also, for one-sided communication, the middleware should be equipped with suitable mechanisms to notify the application of completion of data transfers.

3.2.1 Message Transfer Time (Send-Centric)

The Figure 3.2 shows the time line for the send-centric one-sided communication. There are evidently two possible scenarios. The first one is where the middleware uses handshaking to establish the receiver's buffer address. Here,

$$T_{MP} \doteq T_{WMLP} + T_{RTS} + T_{CTS} + T_X + T_{NOTIFY}. \tag{3.8}$$

Alternatively, it is also possible that the send and receive applications perform an early binding with the middleware. The applications can specify a schedule of buffers to the middleware before the commencement of message passing operations. Given the fact that the address of destination is known beforehand, the send middleware no longer needs to have any explicit handshaking. In this scenario, the total time is given by

$$T_{MP} \doteq T_{WMLP} + T_X + T_{NOTIFY}. \tag{3.9}$$

3.3 Zero-Sided Communication

Zero-sided communication is characterized by the absence of data transfer operations at the user thread level by any of the participating sides (per operation). There is no rendezvous with processes at communication end points. It is the responsibility of the underlying middleware to schedule data transfer depending on the application's QoS, which translates to data transfer calls to/from prespecified application memories at prespecified times. Zero-sided communication necessitates early binding and hence eliminates the delay caused by handshaking, synchronization, and system calls.

3.3.1 Message Transfer Time

In the absence of any per-message synchronization, there is only one component making up the total message transfer time, the actual time taken by the message to travel across the network. Hence,

$$\mathrm{T_{MP}} \doteq \mathrm{T}_X. \tag{3.10}$$

3.4 Sidedness and Real-Time Programming Models

Two-sided communication is typically the message passing paradigm for best-effort tasks. Best-effort tasks are those tasks that do not specify any QoS requirements or constraints to the underlying middleware. The message transfer patterns of these tasks are modeled as unpredictable in middleware design and therefore require explicit synchronization between the communicating nodes. Best-effort tasks theoretically cannot realize one-sided or zero-sided communication. This is primarily because there is no way of notifying the middleware of the application status (or vice versa). The following mechanism can be used by the best-

effort tasks to achieve something close to one-sided and zero-sided communication. An application could register a list of buffers with the middleware and then produce/consume buffers from this list in a predetermined order. The buffers need to be as large as the largest message size since there is no prior knowledge of either the message initiation times or the message sizes. The middleware on the send side will keep polling a status flag (or equivalent) that is set by the application. As soon as the middleware becomes aware of the readiness of the application, it initiates the transfer operation directly to the receive application buffer, since this information is available to it beforehand. The notification for data completion (to the application) is again achieved by middleware setting a status flag (or equivalent). This, however, cannot be termed zero-sided because the application still needs to set and read flags for notification. While eliminating the synchronization times, this model actually is inefficient with respect to both buffer usage and CPU utilization – applications need to regularly poll the status flags. The performance of message passing itself depends on the frequency of such polling.

Applications that use the event-driven programming paradigm achieve two-sided communication. Here, the sender and receiver applications can raise events that trigger the actual data initiation and reception. The more interesting aspect of event-driven applications is that they can achieve one-sided communication as well. The handicap suffered by best-effort tasks, that is, the inability to notify the application by the middleware of data reception, is addressed by having the middleware raise events. However, the receive application does need to register its buffer with the middleware beforehand. If the application did such early binding, scenario two of Section 3.2 would be achieved. Message-based-scheduling techniques follow such a strategy and use one-sided communication. If the application

does not use early binding (case 1 of Section 3.2), what actually transpires is a two-sided communication at the middleware level, despite maintaining one-sided communication at the application level. Zero-sided communication is not possible with the event-driven programming model alone because the send application (in the send-centric scenario) needs to explicitly notify the middleware when it wants to initiate the data transfer.

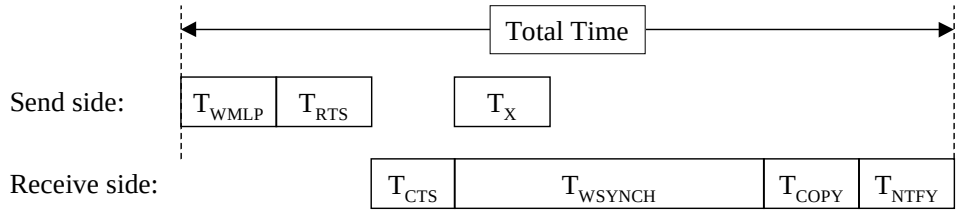
Applications that use the time-driven programming model can achieve all of the three sidednesses of communication. Since a time-driven task has complete prior knowledge of its message transfer patterns, the middleware can initiate message transfers to or from prespecified buffers and at prespecified times. Using two-sided or one-sided communication with the time-driven model does not make any sense because of the extra overheads. A time-driven application would be more efficient if it used zero-sided communication, provided it has either periodic or other attemptable temporal message transfer patterns.

3.5 Sidedness and Effect on Latency Jitter

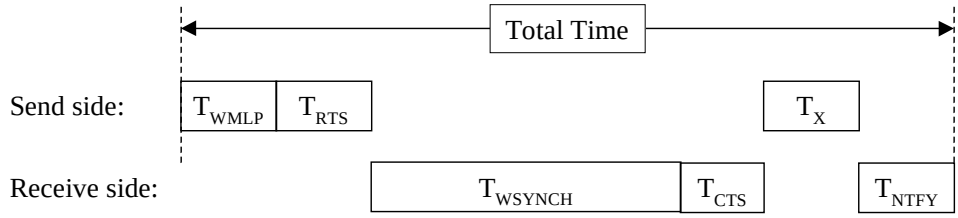
Zero-sided communication does not require any explicit per-message synchronization between the participating nodes. This implies that this model has a much shorter critical instruction path than the one-sided and two-sided approaches. The shorter the critical path, the lesser would be unpredictability in the instruction execution times – there would be fewer cache misses, and so on . Hence hard real-time systems would do well to follow the time-driven approach using zero-sided communication. Of course, programming may be much more onerous.

In time-based hard real-time systems, the message transfers are evidently either periodic or a function of time and are based on a globally parameterized schedule.

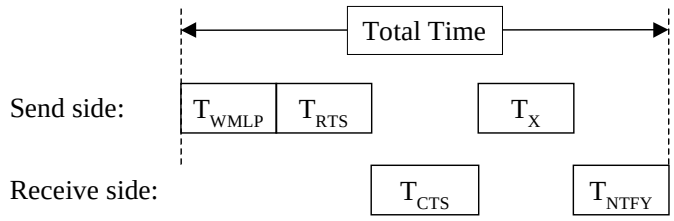
Using zero-sided communication in these systems allows for decoupling the scheduling of computation and communication threads, thereby achieving better predictability. This can help in jitter filtering and can prevent jitter cascading. For example, the jitter in the message ready time in sender is totally opaque to the receiver because the communication thread gets scheduled only at fixed time intervals (based on application-specified QoS).



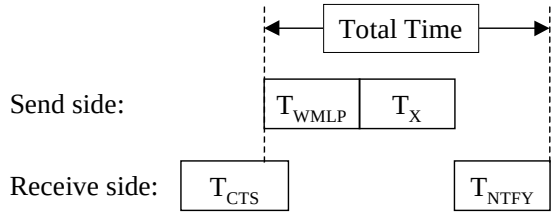
Case I: The application calls *send* before the corresponding *receive* and the middleware maintains a temporary buffer to store incoming messages.



Case II: The application calls *send* before the corresponding *receive* and the middleware waits for the *receive* before responding with a CTS.

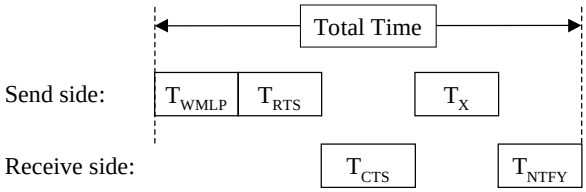


Case III: The application calls *send* and the corresponding *receive* simultaneously.

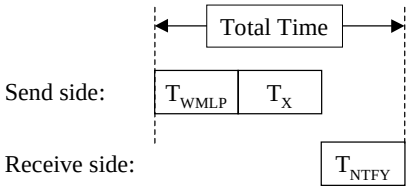


Case IV: The application posts a *receive* before calling the corresponding *send* (ready-mode communication).

Figure 3.1: Timeline for Two-Sided Communication



Case I: The middleware uses handshaking to send the receiving buffer's address to the sender in order to perform a remote DMA operation.



Case II: The middleware's "early binding" knowledge of the receiver's buffer obviates the need for any handshaking.

Figure 3.2: Timeline for One-Sided Communication

CHAPTER IV

APPROACH

The hypothesis was verified by developing HARE, a prototype implementation of a subset of the MPI/RT 1.1 standard (MPI/RT Forum 2001), that supports time-driven QoS, and by subsequently writing compact applications over HARE using these time-based channels to demonstrate zero-sided communication. This chapter describes the research methodology behind the design and development of HARE.

4.1 Research Methodology

The research methodology for this thesis can be broken down into four different activities:

1. Study of BDM-RT (Chakravarthi 2000) and TURTLE (Apte et al. 1999) and subsequent evaluation on the feasibility of implementing HARE using BDM-RT and TURTLE API's.

2. Enhancing the capability of the global clock module to support up to eight nodes.

3. Implementation of time-based MPI/RT channels using BDM-RT and TURTLE (Development of HARE).

4. Development of example application programs over HARE that perform and demonstrate zero-sided communication. The example programs have also been used to determine the performance and predictability of HARE. The start and

stop times of the different channel tasks have been recorded with TURTLE built-in profiler and used to verify the realization of zero-sided communication.

The first three activities are described in the subsequent sections, while the last is explained in chapter 5.

4.2 Feasibility Study

Based on the literature survey, an initial study was made to determine the feasibility of implementing HARE using BDM-RT (Chakravarthi 2000) and TURTLE (Apte et al. 1999) API's. The focus of the study was on determining the ease of layering HARE on BDM-RT and TURTLE, including the mapping of MPI/RT time-driven QoS specifications to that of TURTLE task parameters. This study also identified limitations in BDM-RT and TURTLE that had an impact on the development of HARE.

BDM-RT has been designed by Chakravarthi to support the development of HARE. Among other things, BDM-RT's support for global clock and PCI bus control (by using blocking DMA) are crucial for any MPI/RT implementation supporting time-driven QoS. However, BDM-RT has a few drawbacks as well. BDM-RT does not differentiate messages from different sources/nodes. Therefore, it is the responsibility of any higher layer above it to incorporate this functionality in their headers. However, BDM-RT does provide tag-based demultiplexing. HARE uses this tag to send in the local node information. Another constraint while using BDM-RT is the choice of buffers. In the current implementation, BDM-RT can send data using its own buffers only. This implies that HARE should do an additional *memcpy* from the user buffer to this buffer during every message transfer. This induces overhead and impacts latency, but is inevitable. Though BDM-RT does provide support for

global clock, the support was available for only two nodes. Additional features were added to BDM-RT as a part of this thesis to allow support of more than two nodes. This work is discussed in Section 4.3.

TURTLE, like BDM-RT, was also developed as a part of the PromisQoS project and provides features that facilitate the development of HARE. The MPI/RT time-based QoS parameters, period and deadline, directly map to TURTLE task parameters. However, TURTLE, a variant of RT-Linux, supports only kernel tasks. This implies that in the existing setup, both the MPI/RT library and the user programs using it need to be installed as kernel modules. Consequently, neither the library implementation nor the user programs can use the standard user C/C++ libraries that are available to normal user programs. Though this arrangement might be unacceptable to real-world systems, it is none the less sufficient for the current proof-of-concept implementation, HARE.

Admission control is an integral component of any model based on resource reservation (Lee et al. 1996). The same applies to MPI/RT. However, in the current MPI/RT implementation, admission control is not implemented, because of the current limitations of TURTLE. TURTLE admits all jobs but suspends them at runtime if it should be unable to honor any QoS requirements. TURTLE does not perform any feasibility checks while admitting the tasks.[1]

Lack of admission control, however, is not a deterrent for the current work. Empirical values for compute times for the channel tasks for different message sizes have been calculated using trial runs. TURTLE also provides mechanisms to record the compute times for its tasks and the MPI/RT library can pass this feature as a

[1]Development of admission control for MPI/RT programs is however an ongoing work of two other PhD dissertations at the High Performance Computing lab in the Department of Computer Science.

non-standard API up to the user application. The MPI/RT application can then use this as a reference while specifying its QoS specifications. The support for this new API is left as future work.

A concurrent activity with the feasibility study had been the experimentation on BDM-RT and TURTLE to perform zero-sided communication. These experiments have already been discussed in Section 1.3.2.

4.3 Global Clock

The availability of a global clock module is essential for any distributed real-time system that intends to support time-based QoS. The original global clock module (Chakravarthi 2000) on BDM/RT supports only two nodes. The clock module was modified as a part of this thesis to scale up to eight nodes. The limit of eight nodes is not because of the limitation of the proposed module; rather, it is the number of nodes in the current test-bed cluster. The scalability of the module is actually limited by the number of nodes that can be directly connected to a single Myrinet (Boden et al. 1995) switch. BDM-RT uses a naive mapping algorithm to map nodes to ports on Myrinet switches and this needs to be modified significantly if the global clock intends to synchronize nodes across multiple switches. This work is outside the scope of the current thesis and is left as future work.

4.4 Functionaltiy Supported by HARE

HARE implements only a subset of the MPI/RT 1.1 specification and therefore has a much reduced functionality compared to a full-fledged implementation of the specification. Some of these restrictions are as follows:

- Only point_to_point channels are supported, collective channels are not supported.
- Only time-driven QoS is supported.
- MPIRT_GROUP_WORLD, the default world group, is the only group supported.
- The algorithms for container sets and vectors trade off high performance for simplicity. Simple linked lists are used in favor of other optimal data structures.
- FIFO is the only buffer iterator policy that is supported.
- There is no admission control.

The complete list of the subset of API that has been implemented in HARE is provided in appendix A.

The primary goal of HARE is to guarantee predictability; performance, though an important goal, is only a secondary requirement, as with many real-time systems. This is the rationale in using simple data structures and algorithms for developing MPI/RT objects like container sets and vectors.

4.5 Implementation

HARE, as mentioned previously, is layered over TURTLE, BDM-RT and the global clock module. HARE, like the other subsystems, is also developed as a kernel module. The user programs using HARE are also kernel modules and execute their code in their *init_module* and *cleanup_module* functions.

The MPI/RT API's have been designed using object-oriented principles and a language like C++ would be an ideal choice to implement the library. However, using

C++ for developing kernel modules have been discouraged in Linux discussion forums because of of name clashes between C++ keywords and kernel symbols. Hence, HARE was developed using C. Inheritance and object-oriented concepts have been simulated using nested structures. All MPI/RT objects, as required by the standard, are handles (or pointers) to the actual structures. Simple linked lists have been used to develop container sets. All buffers are allocated dynamically using *kmalloc*.

Internally, HARE registers one TURTLE task for every time-based channel the user creates during commit time. This task is named the *communication thread* throughout the rest of this thesis document. The parameters for these communication threads are specified by the application as a part of the QoS specification during channel creation. This mechanism of handling time-based tasks binds HARE tightly to TURTLE.

For time-driven systems, the MPI/RT 1.1 standard (MPI/RT Forum 2001) specifies that it is the user's responsibility to produce and consume messages at specified instants. In the PromisQoS architecture, user programs also need to register themselves with TURTLE so that they get appropriately scheduled at required intervals. These TURTLE tasks will be referred to as *computation threads* from now on. Zero-sided communication can be achieved only by choosing the appropriate values for both the QoS of the channels and the scheduling of the computation threads.

To synchronize the scheduling of the communication and computation threads properly, it becomes important to know the start times of the the communication threads. The MPI/RT standard mentions that the periods of all channels start at the time of return of the MPIRT_Commit function. This return time is unavailable to the user program beforehand and so specifying the correct parameters for the

computation thread becomes a problem. HARE deviates from the standard in that MPIRT_Commit returns the time when the channel tasks begin their periods. This time is not the time when this function returns, but rather a time in the future, thereby allowing the computation threads to be spawned with correct parameters in the meanwhile. This and a few other gray areas in the MPI/RT standard have been listed in chapter 6.

4.5.1 Sample Example Program

Figure 4.1 shows a sample HARE application program. The program, in its *init_module*, calls the MPIRT_Init function to initialize the MPI/RT library and then goes on to creating the various objects. While creating the QoS objects for time-based channels, the release time is specified as a relative time from the period. The MPI/RT standard mandates that the time be relative whenever the period is specified. The deadline is another argument to Qos-Create API. HARE translates these parameters into TURTLE parameters inside the MPIRT_Commit funcion. As mentioned above, the MPIRT_Commit function itself returns the period-start-time, a time in future. This arrangement gives enough time for the application program to register its computation threads with TURTLE. In the above code, *ReceiveFunction* and *SendFunction* are the two computation threads. Then, the application calls MPIRT_Activate before returning from *init_module*. At this stage, TURTLE schedules the tasks at their specified periods and zero-sided communication takes place. When the computation is done, the tasks call MPIRT_Uncommit. MPIRT_Finalize is called in the *cleanup_module*.

4.6 Latency under PromisQoS

This section defines quantitatively the latency for applications using the zero-sided communication model with HARE in the PromisQoS framework. The example considered is a simple send-receive program. At the time when zero-sided communication takes place in this setup, there are two tasks at each node, one computation thread and one communication thread (channel task).

The timeline in messaging is shown in figure 4.2. The latency depends broadly on two components, the compute time of the application and channel tasks and the actual message latency (time taken by the message to travel across the network from the sender to the receiver).

In the figure, let δ_2 be the time required by the receive user thread for preparation to receive the message, δ_1 be the time by the send user thread for preparation to send the message, α be the scheduling overhead, β_2 be the deadline of receive channel task, β_1 be the deadline of send user task, μ be the network latency, γ_2 be the receive protocol processing time in HARE and γ_1 be the send protocol

processing time in HARE. The latency of HARE is as follows:

$$T_{Lat} = T_{10} - T_2 \tag{4.1}$$

$$T_2 = T_1 + \delta_1 \tag{4.2}$$

$$T_3 = T_1 + \beta_1 \tag{4.3}$$

$$T_4 = T_3 + \alpha \tag{4.4}$$

$$T_5 = T_4 + \gamma_1 \tag{4.5}$$

$$T_7 = T_5 + \mu \tag{4.6}$$

$$T_7 = T_6 + \gamma_2 \tag{4.7}$$

$$T_8 = T_6 + \beta_2 \tag{4.8}$$

$$T_9 = T_8 + \alpha \tag{4.9}$$

$$T_{10} = T_9 + \delta_2 \tag{4.10}$$

Substituting the values of T_i from the above equations,

$$T_{Lat} = \beta_2 + \beta_1 + 2\alpha + \delta_2 - \delta_1 + \gamma_2 - \gamma_1 + \mu. \tag{4.11}$$

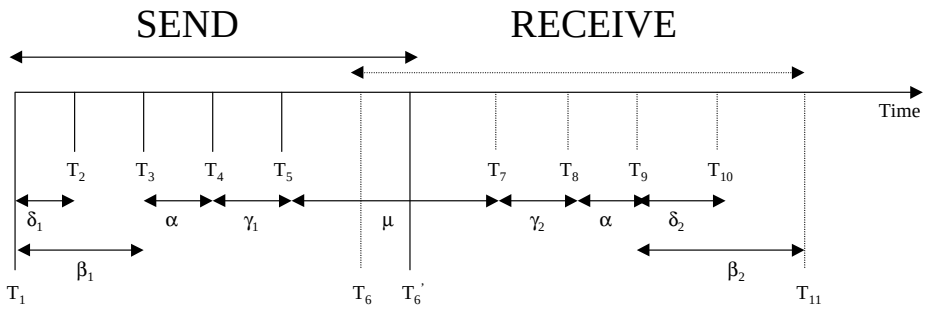
In the above equation, α is a constant and δ_1 and δ_2 are user-program-dependent values. A significant component of γ_2 and γ_1 is the time for making the extra copy from the user buffer to BDM-RT buffer (during send) or from the BDM-RT buffer and application buffer (during receive). Hence, these values are dependent on the message size. μ, the actual network latency, and β, the deadline for the tasks, also increase with increasing message sizes.

It should be mentioned here that the above timeline assumes an ideal synchronized scheduling of the receive task where by it receives the message at exactly

the same instant it is ready to process it. In practice, this is difficult to achieve, and there might be some additional delay because of this scheduling variance.

```
init_module()
{                                                          Uninitialized Phase
    Init()

    /* Resources specifed now, will be created during commit */

    Create Objects, Containers, Container sets

    /* Create QoS. Note that Release time are relative to the period.
    These will be translated to absolute times of TURTLE by HARE */
    Create Channel Qos1 (Release time S1, Period P, Deadline D1)      Non RT Phase

    Create Channel Qos2(Release time S2, Period P, Deadline D2)

    Create Channel1 (Qos1)

    Create Channel2 (Qos2)
    /* Do commit now */

    Period_Start_time = Commit()

    /* Resource are available now for RT communication */

    /* Now we have period start time, register with turtle the computation thread */

     TURTLE_Register(SendFunction, Period_Start_time + send_start_time,
          Period,myDeadline)

    TURTLE_Register(ReceiveFunction, Period_Start_time+ receive_start_time,
         Period,myDeadline)

   Activate(Channel1)
   Activate(Channel2)                                          RT Phase
}
 SendFunction
 {
    For every period
         Do computation
         Fill Buffer
   When done
        Deactivate(Channel1)
        Uncommit()
}
cleanuup_module                                            Non RT Phase
{
   Finalize()                                              Finalized Phase
}

ReceiveFunction
{
   For every period
         Read from buffer
         Do computation
   When done
        Deactivate(Channel)
        Uncommit()
}
```

Figure 4.1: Sample Application Program



T_1 - User app begins
T_2 - Message is ready
T_3- T_4 TURTLE does context switching
T_4 - MPI/RT channel Task begins
T_5 - Message actually sent across network
T_6 - Receive Channel Task begins
T_6' - Send Channel Task ends
T_7 - Message actually received
T_8 - Receive MPI/RT channel task ends
T_8-T_9 TURTLE does context switching
T_9 - User app begins
T_{10}- User app actually receives message
T_{11}- User app ends

Figure 4.2: Timeline in Messaging

CHAPTER V

EXPERIMENTS, RESULTS, AND ANALYSIS

This chapter presents the experimental methodology and the results obtained from these experiments. The aim of these experiments is to demonstrate the validity of the hypothesis.

Three main experiments were conducted as a part of this thesis. The first was a three-node ring application used to demonstrate zero-sided communication. The second experiment was an effort to measure the latency and latency jitter of HARE. The final experiment compared the performance of best-effort tasks with that of time-based tasks. An associated experiment was that of measuring the accuracy of the global clock module while supporting eight nodes.

The following section describes the software and hardware configuration of the experimental testbed. The subsequent sections describe each of the above-mentioned experiments along with their expected and actual results. The analysis of the results is also done in these sections.

5.1 Experimental Setup

This section describes the configuration in which the experiments were performed. This section clearly identifies the limitations and estimates the accuracy of the metrics gathered.

5.1.1 Hardware and Software Configuration

All experiments were carried out using two or more nodes of an eight node Linux cluster connected by Myrinet (Boden et al. 1995). BDM-RT (Chakravarthi 2000) was installed in each node as a kernel module. Both HARE and the example programs were also installed as kernel modules.

Each node consists of a PentiumPro 200MHz processor. All the network interfaces (Ethernet, Myrinet, and SCSI interfaces) share a single 32-bit 33MHz PCI bus. The Myrinet interface consists of the LANai 4.x processor with 1MB SRAM.

This configuration represents a typical commercial-off-the-shelf (COTS) desktop. The main reason for limiting the experiment to this configuration is the current state of development of TURTLE (Apte et al. 1999) and BDM-RT. However, it appears fair to conclude that the results achieved in this setup can be easily extrapolated to the latest Intel processors and Myrinet interfaces, and some porting is underway already apart from this thesis work.

5.1.2 Timers Used

The clock synchronization module provides a 64-bit nanosecond resolution virtual clock on all slave nodes. This clock is based on the 64-bit nanosecond resolution APIC time-stamp Counter (found in Pentium Chipsets) of the master node. All the test programs in the following experiments use global time-stamps to compute end-to-end latency and latency jitters.

5.1.3 Accuracy and Error Bounds

As mentioned in the previous section, the global clock was used to measure latency and latency jitters. The global clock has an error bound of ± 5 μs, as will be shown in Section 5.2. The latency values measured for the current experimental

setup also depend on the TURTLE scheduling overhead since message transfers in this configuration involve multiple context switches. The worst case scheduling overhead of TURTLE was measured to be about 25μs, but the average case was around 15μs. This variation in the scheduling overhead can potentially affect the latency jitter. The values for the compute times and start times for the different tasks were estimated empirically. This implies that the latency values obtained are not strict lower bounds and can be reduced through a more rigorous evaluation. This work is however not essential for proving the hypothesis. All the experimental results for latency and latency jitter were averaged over 5,000 runs. This number is large enough to provide a reasonable averaging of values. Also with 5,000 periods, these experiments span at least two clock resynchronization intervals and capture any effect this resynchronization procedure might have.

5.2 Global Clock

The aim of this experiment was to determine the accuracy of the global clock while supporting up to eight nodes. Since the global clock module uses a master-slave algorithm, the accuracy of the global clock can be obtained by measuring the variation between the master and the slave clock values at regular intervals. To capture the worst case skew, the deviation between the master and slave clocks was measured at every synchronization period. To make a reasonable study, the deviations were captured for about 2,500 synchronization intervals, with a resynchronization period of one second.

5.2.1 Expected Results

The accuracy of the global clock with two nodes was measured to be bounded by $\pm$5μs. The enhanced version of the global clock module was also expected to

retain that accuracy as long as all the nodes were connected by a single Myrinet switch. This accuracy can be maintained because of a novel technique used by the global-clock algorithm whereby the clock message has been time-stamped at different instants during its transit from the master node to the slave nodes, thereby reducing the error incurred because of the latency of the network.

5.2.2 Actual Results

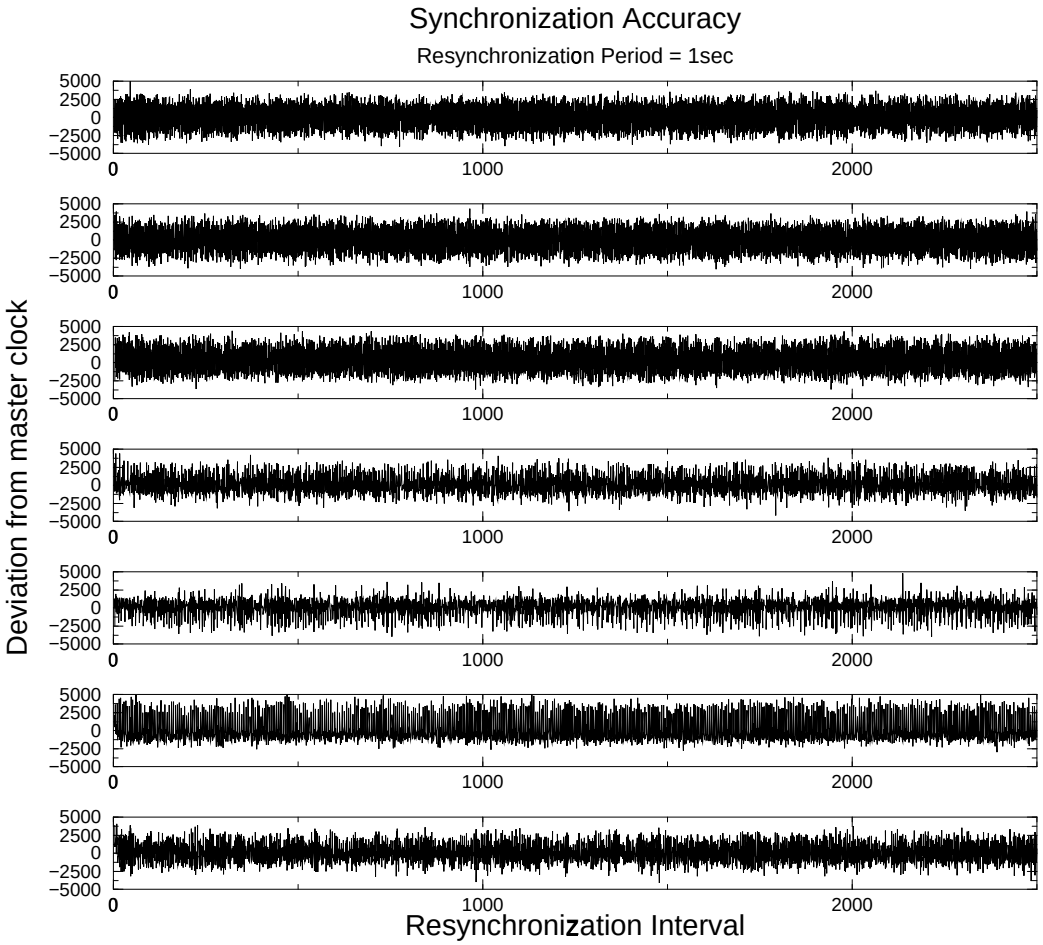


Figure 5.1: Global Clock : Deviation(ns) of the Seven Slave Modules from Master

The deviation of the slave nodes with respect to the master is given in Figure 5.1. In this figure, each row represents the deviation of one slave. As can be observed from the figure, the deviation of the slave nodes is within $\pm 5\mu$s of the master. This is the expected result, and the global clock module provides a good platform for developing HARE and other MPI/RT example programs. This fine-grained clock is also useful in evaluating the performance of HARE by providing the correct support for latency and latency jitter measurements.

A complete and a more rigorous treatment of the experimentation, results, and analysis of the global clock module is available at Chakravarthi et al. (2000).

5.3 Three-Node Ring Application

The primary goal of this experiment was to demonstrate zero-sided communication. The application used in this experiment was the ring application where messages were transferred across nodes in a ring fashion. As the name indicates, this experiment involved three nodes – A, B, and C. Each of the three nodes opened two MPI/RT time-based channels, one channel with each of the other two nodes. One of the channels was a receive channel and the other a send channel, which completes a ring. The arrangement is illustrated in Figure 5.2.

This experiment involved sending a designated number of messages (5,000 messages) in a ring. Node A was the originator of the ring and incremented the message number (which was copied to the first four bytes of the data payload) at the beginning of each round. The experiment was considered a failure if the receiver application of any node did not have its message whenever it was scheduled to run. The experiment was conducted for two arbitrary messages sizes – four bytes and 1k bytes.

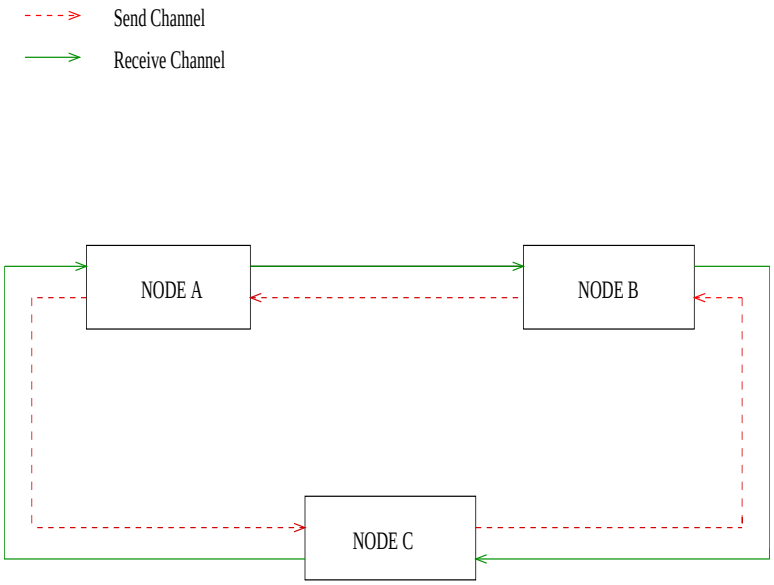


Figure 5.2: Three-Node Ring Application

In order to prove that dedicated nodes are not a necessary requirement for zero-sided communication, the above experiment was also repeated with an additional unrelated task scheduled in the system.

5.3.1 Expected Results

It was clearly mentioned in the previous chapter that zero-sided communication in PromisQoS architecture can be achieved only by a careful selection of values for both the channel parameters and for the computation-thread parameters. Once the right values are chosen, the receive component of the computation thread should receive the message whenever it is expecting it. In other words, there should not be any missed or delayed messages.

The presence of extraneous tasks should not affect the completion of the ring as long as their task parameters are chosen so as not to conflict with the existing TURTLE tasks in the system. That is, as long as there are no resource conflicts,

the subsystems comprising the PromisQoS architecture – TURTLE, BDM/RT, and HARE should guarantee zero-sided communication.

5.3.2 Actual Results

The three-node-ring experiment was conducted successfully and the ring application completed without any missed or delayed messages, demonstrating that zero-sided communication is indeed possible to achieve in practice given the right middleware layer support. Table 5.1 shows the choice of task parameters for all the three tasks in the system that enabled zero-sided communication.

Table 5.1: Task Parameters for Zero-sided Communication

Node	Task Name	Start Time	Period(ms)	ComputeTime(μs)	Deadline(μs)
A	User	0	5	50	75
A	Channel Send	75	5	50	75
B	Channel Recv	125	5	50	75
B	User	200	5	50	75
B	Channel Send	275	5	50	75
C	Channel Recv	325	5	50	75
C	User	400	5	50	75
C	Channel Send	475	5	50	75
A	Channel Recv	525	5	50	75

The task graph shown in Figure 5.3 illustrates the task scheduling for this configuration while achieving zero-sided communication. This figure is not to scale, it is just a convenient way for depicting how the tasks were planned for scheduling. The three nodes share the same notion of time and hence have been plotted in the

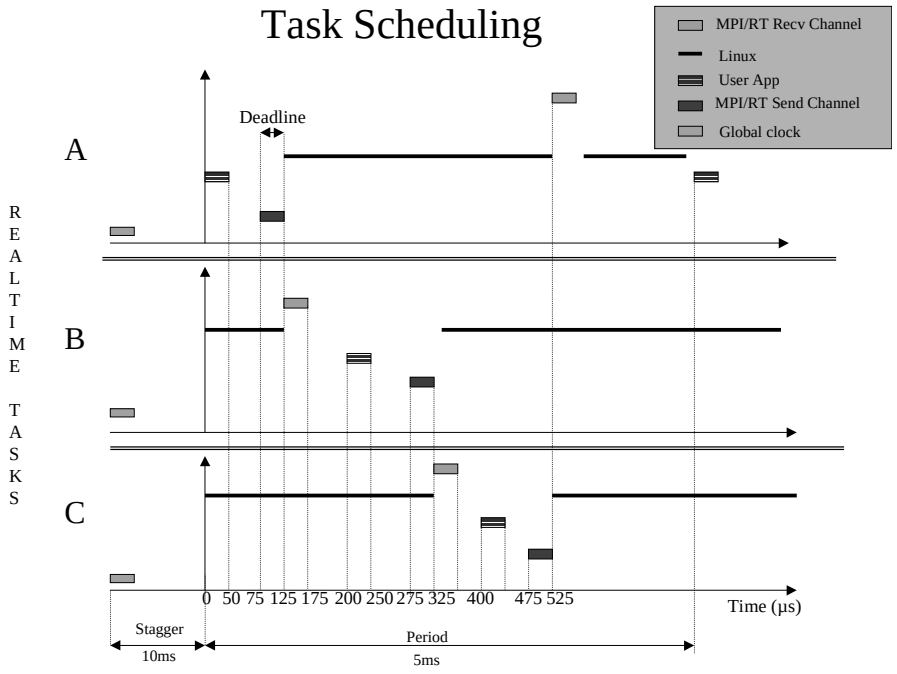


Figure 5.3: Task Scheduling for Zero-Sided Communication (Ring Application)

same graph. Linux is the greedy task that occupies the CPU whenever TURTLE has no other tasks scheduled to run.

The successful completion of the ring experiment was also verified by using the built-in profiler from TURTLE. Table 5.2 shows the start and stop times for the different tasks in the system for two periods when the ring experiment was conducted with the four byte message. As mentioned in Section 5.1, the times were measured from the global clock. The dummy task is the extraneous task in the system, introduced to emphasize that dedicated nodes are not mandatory for achieving zero-sided communication.

Table 5.2: Start and Stop times for TURTLE tasks

Task Id	Task Name	Period	Start Time	Stop Time
10	User	1	$1627\times2^{32}+1402277084$	$1627\times2^{32}+1402292244$
22	Channel Send	1	$1627\times2^{32}+1402293504$	$1627\times2^{32}+1402309634$
88	Dummy Task	1	$1627\times2^{32}+1406677140$	$1627\times2^{32}+1406692245$
31	Channel Recv	2	$1627\times2^{32}+1407023896$	$1627\times2^{32}+1407039980$
10	User	2	$1627\times2^{32}+1407276569$	$1627\times2^{32}+1407292284$
22	Channel Send	2	$1627\times2^{32}+1407292660$	$1627\times2^{32}+1407309045$
88	Dummy Task	2	$1627\times2^{32}+1411676301$	$1627\times2^{32}+1411692270$
31	Channel Recv	3	$1627\times2^{32}+1412030436$	$1627\times2^{32}+1412041506$

5.4 Latency and Latency Jitter

The primary goal of this experiment was to measure the latency and latency jitter of the time-based MPI/RT channels under PromisQoS. The latency values obtained from this experiment can be used as a reference by other real-world MPI/RT applications under PromisQoS for their QoS specifications.

This experiment was carried out using two nodes, a sender node and a receiver node. The sender node opened the send channel and the receiver opened the receive channel. Therefore, there were two TURTLE tasks in the system – the sender/receiver application task (computation thread) and the MPI/RT channel task (communication thread).

Latency was measured by recording the global time-stamp at the sender's node (by the computation thread) before inserting the buffer into the input buffer iterator and at the receiver's node just after the buffer has been extracted from the output buffer iterator. Since the global time-stamp is 64 bits, the minimum buffer size for

this experiment was fixed at eight bytes. The experiment was carried out for message sizes from eight bytes to 4K bytes. The upper limit was dictated by the the MTU of BDM-RT, which is 8K bytes. In each experiment, a stream of messages (5,000) was sent and the latency and latency jitter were calculated.

5.4.1 Latency - Expected Results

The latency of any real-time messaging layer is expected to be considerably high compared to other high performance messaging layers such as GM (Myricom 1998), FM (Pakin, Lauria and Chien 1995), and BDM (Henley et al. 1997) (Chakravarthi 2000). HARE, a layer on top of another low-level real-time library BDM-RT, is expected to have even higher latencies than these low-level real-time messaging libraries. In fact, HARE is expected to have considerably higher latency than BDM-RT. This higher latency is primarily because of two additional overheads suffered by HARE that are not present in BDM-RT. The first such overhead is the number of context switches involved in HARE. A simple send-receive requires two context switches, between user context to MPI/RT library context at both the sender and the receiver. The second overhead is because of the actual protocol processing by the MPI/RT library. An important component of this MPI/RT protocol processing is the additional *memcpy*. BDM-RT can perform message transfers only on specific buffers maintained by itself. Consequently, data from any user buffer needs to be copied by HARE to the BDM-RT buffer before a send or from BDM-RT buffer after a receive. This additional copy is expected to affect the latency as well.

Of these two overheads, the context switching overhead is a fixed cost and is independent of the message size. Therefore, the relative growth of latency for different message sizes should follow similar trends with that of BDM-RT.

The theoretical analysis of latency in HARE was provided in Section 4.6.

5.4.2 Latency - Actual Results

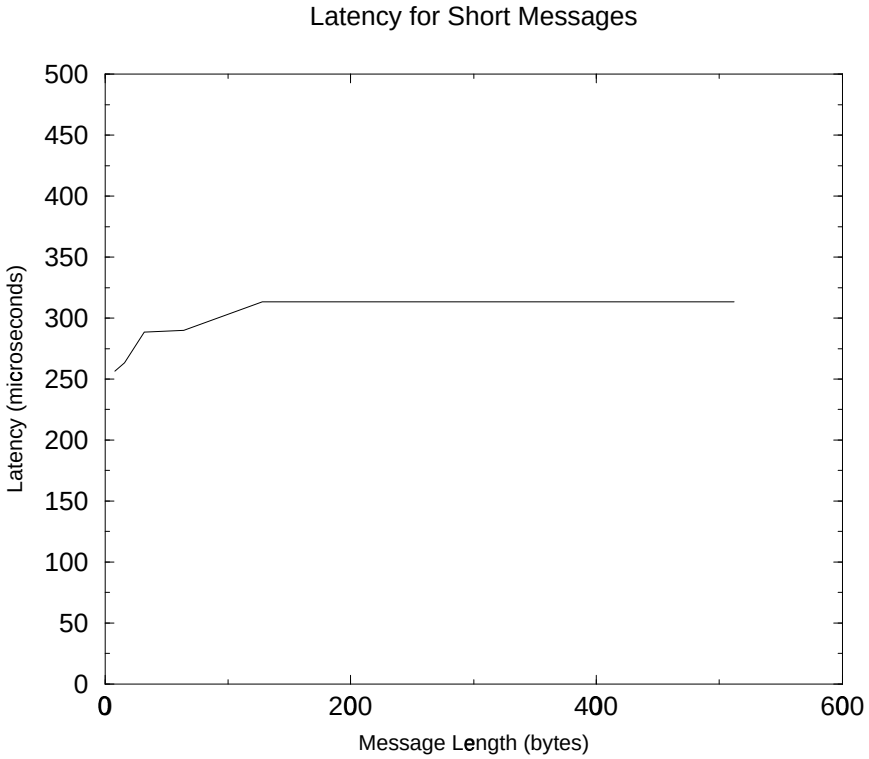


Figure 5.4: Latency for Small Messages

Figure 5.4 shows the latency values for short messages, and Figure 5.5 shows the latency values for large messages. As expected, HARE has a high latency compared to BDM-RT; the eight byte latency of BDM-RT is about 40μs compared to 250μs of HARE. In fact, this latency is higher than even the expected value, but can be explained as follows. The latency for messages in PromisQoS architecture from Section 4.6 is given by

$$T_{Lat} = \beta_2 + \beta_1 + 2\alpha + \delta_2 - \delta_1 + \gamma_2 - \gamma_1 + \mu. \tag{5.1}$$

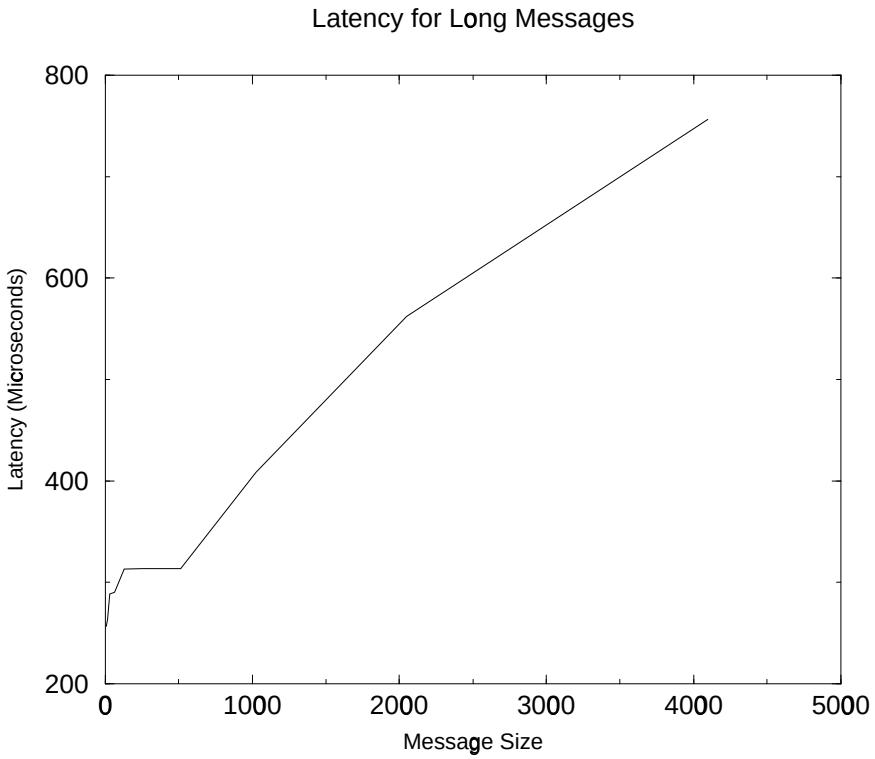


Figure 5.5: Latency for Large Messages

As δ_2 and δ_1 are user-program-dependent parameters, we can assume them to be equal for this experiment. Also, γ_1 and γ_2, the protocol processing times in the sender and receiver respectively, can be assumed to be equal. Plugging in $\beta_1 = \beta_2 = 50$, $\alpha = 25$, $\mu = 50$, gives $T_{Lat} = 200\mu$s. However, as mentioned in Section 4.6, the T_{Lat} calculated with the above equation assumes an ideal setting where the message arrives at the receiver node when the channel-receive task has been scheduled, and has finished all its preprocessing and is expecting the message. This arrangement is rarely achievable in practice and an delay of about 50μs here is the practical limit. This places the total latency at 250μs, which agrees closely with the actual result.

The caveat in this discussion is the fixing of β_1 at 50μs, which is considerably large. The reason is because of the existing state of TURTLE, which operates only at

that level of granularity. This minimum compute time encompasses certain hidden costs and is the operating constraint of PromisQos. Future schedulers on faster systems will reduce this factor.

Another interesting observation from Figure 5.4 is the flattening of latency curve for small messages. This can also be explained with precisely the same reasoning as above. The β in the above equation is the dominating parameter for short messages. Given the granularity of modification in β, messages of different sizes can be handled by same or similar values of βs. This flattening is a natural result of this characteristic.

The increasing behavior seen in the latency for large messages is typical of latency curves and needs no explanation.

5.4.3 Latency Jitter - Expected Results

The most important feature of real-time messaging libraries like HARE is their control of latency jitter. HARE, like BDM-RT, is expected to have low latency jitters compared to high performance messaging libraries. In fact, HARE is expected to have a marginally higher jitter than BDM-RT, because of the variance in the scheduling overhead. TURTLE scheduling overhead has a variance of about 10μs; HARE should be worse off by no more than 10μs as compared to BDM-RT. BDM-RT has an average latency jitter of 7μs. So, the jitter of HARE should still be small in absolute terms.

5.4.4 Latency Jitter - Actual Results

The latency jitter is shown in Figures 5.6 and 5.7. The maximum latency jitter is about 10μs, which is only marginally higher than that of BDM-RT. The reason for this behavior is that the scheduling jitter of TURTLE has been masked by a careful choice of parameters for the TURTLE tasks. The deadline offset for the experiment

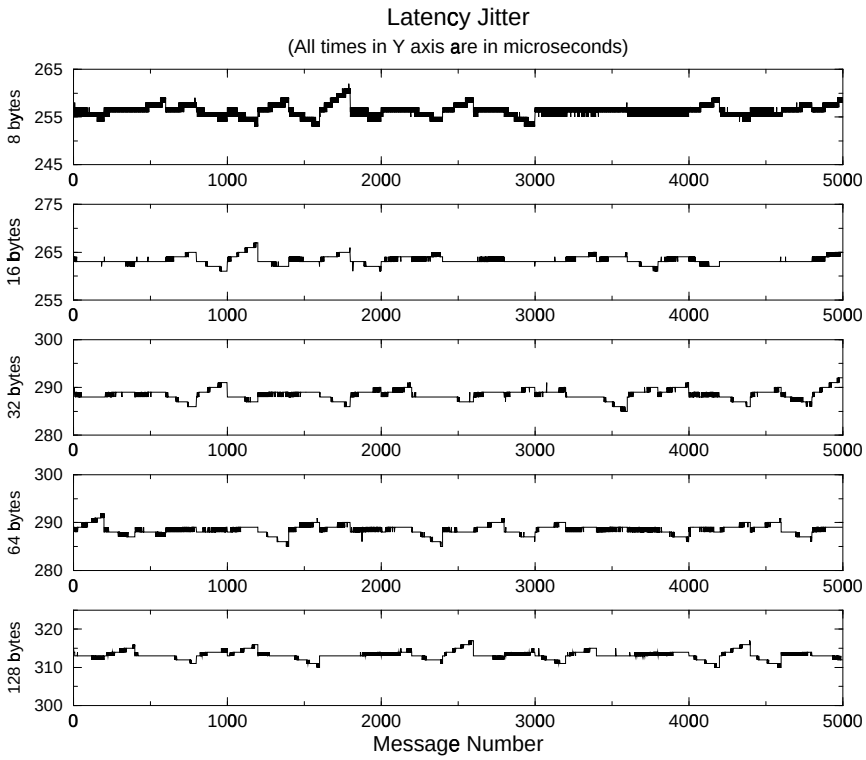


Figure 5.6: Latency Jitter for Small Messages

was chosen as 25μs, which is the same as the worst-case scheduling overhead of TURTLE.

5.5 Best-Effort and Time-based Tasks

The primary goal of this experiment was to compare the performance of time-based tasks with that of best-effort tasks. End-to-end latency and latency jitter values were measured to evaluate the performance.

The experimental configuration was similar to experiment 5.4; it also involved two nodes, the sender node A and the receiver node B. The setup involved opening two channels – a time-based channel, and a best-effort channel by both A and B. Node A also registered two other TURTLE tasks as its computation threads, one

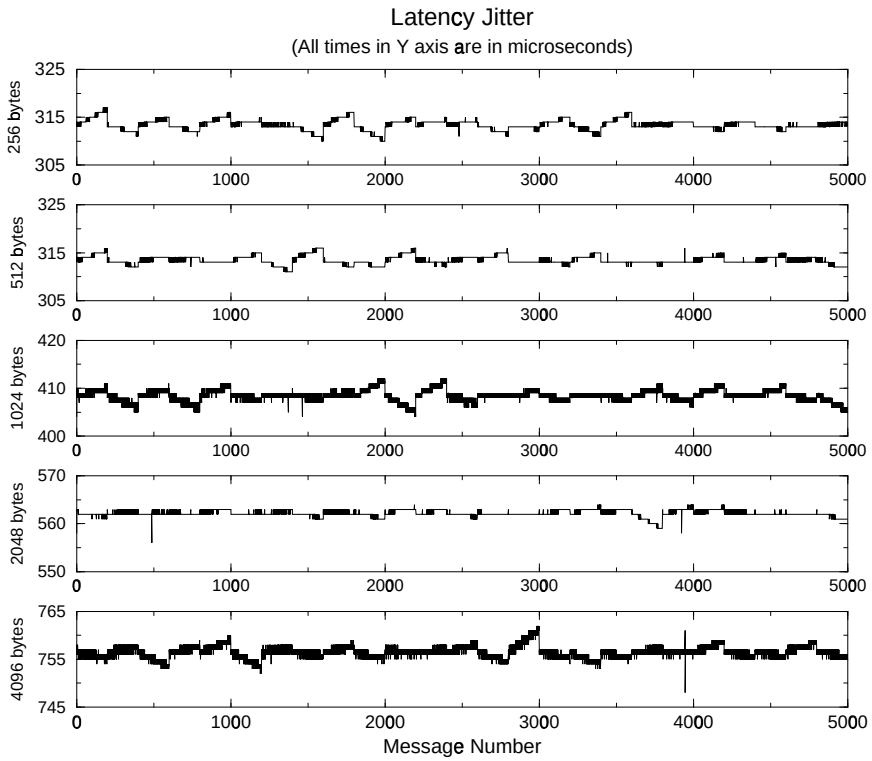


Figure 5.7: Latency Jitter for Large Messages

each for providing send messages (for each of the channels). Similarly, node B also had two user receive tasks registered with TURTLE.

Best-effort tasks, by definition, do not specify any requirements for the scheduling of either their computation thread or their communication thread. These tasks are inherently unpredictable (unlike the time-based tasks, which have clearly defined periods). In other words, the number of messages these tasks process in each period is variable. As a result, these tasks are usually greedy; they take up as much resources as are required subject to availability, they do not prespecify any QoS requirements. However, as mentioned in Section 2.5, TURTLE does not allow any greedy task other than Linux. Hence this experiment was carried out on simulated best-effort tasks.

The best-effort task was simulated in the PromisQoS environment as follows. The send computation thread was configured to generate a random number between zero and four for the number of messages to be sent during each period. The send communication thread (MPI/RT channel task) was configured to do a maximum of two sends (average of the number of sends by the send computation thread) during each period. The receive communication thread and the receive computation thread were configured to consume as many messages as available in every period, which will be a maximum of two in this setup. If the send computation thread did a send of more than two during a period, the extra messages were queued by the MPI/RT library channel send task to be sent during the next period. This experimental setup, though it might not capture the characteristics of a best-effort task completely, is a fairly representative setup because in this setup, the best-effort task does not need to do any of the synchronization a real-world best-effort task might be required to do.

For fairness of this comparison, the send computation thread of the time-based channel was configured to send two messages each period. This means that in the long run, the number of sends by both the best-effort task and the time-based task would be the same. The send communication thread, receive communication thread and computation threads have exactly the same configuration as that of the best-effort task, namely each handles two messages each period.

The actual experiment involved sending a stream of messages from node A to node B and calculating the latency for the transfer. The latency was calculated in exactly the same fashion as in experiment 5.4; time-stamps were taken just before insertion into the input buffer iterator by the sender thread and just after extraction from the output buffer iterator by the receiver thread. The difference between these

time-stamps gave the latency. The latency jitter, which is the variation in latency, is subsequently a straightforward calculation.

5.5.1 Expected Results

The time-driven tasks are expected to have lower latency than the best-effort tasks because of the fact that the time-based tasks have complete knowledge of their message transfer pattern and hence can reserve the resource requirements precisely beforehand (early binding). In this experiment, this translates to specifying the QoS requirements for the computation and communication threads appropriately so as to maximize performance. On the contrary, the best-effort tasks do not have complete knowledge of their message transfer pattern and hence cannot reserve resources a priori. These tasks need to make a tradeoff between efficient resource utilization and achieving higher performance. If these tasks reserve resources based on their worst-case behavior, then they can achieve performance comparable to the time-based tasks but will be wasting resources most of the time as the average case usually requires considerably less resources than the worst-case. In practice, efficient resource utilization is as important as performance and so these tasks are set up to handle the average case behavior.

In this experiment, the send computation thread generates a variable number of messages each period. Because the send communication thread is configured to send only the average number of messages each period, some messages will be delivered in subsequent periods. The latency of these messages, therefore, will vary by a multiple of the period. However, the time-based channels are guaranteed to have their message delivered during the same period itself. In other words, for time-based channels, the latency is independent of the period, whereas the latency is dependent on the period

for best-effort channels. For time-based,

$$\frac{dT_{Lat}}{d\rho} = 0 \tag{5.2}$$

and for best-effort,

$$\frac{dT_{Lat}}{d\rho} \neq 0 \tag{5.3}$$

where ρ is the period.

Though a real-world best-effort task will not suffer from latencies as bad as the period (actually they do not have any periods at all by definition), they are not expected to perform as well as the time-based channels. Instead of the period, the added latency component will be from the extra synchronization that the best-effort tasks with two-sided communication are likely to incur. Therefore, the emphasis of this experiment was not as much on the absolute value of the latency as it was with the relative latencies of the time-based and best-effort channels.

The latency jitter of the best-effort task will also be considerably higher than the time-based tasks for exactly the same reasons. Messages from the best-effort task are not guaranteed to be delivered during the same period and so the jitter itself will be multiples of the period. The latency jitter of the time-based channels is expected to be a few tens of microseconds and have been discussed in the previous section.

5.5.2 Actual Results

The actual results align themselves closely to the expected results. The latency for 10 messages (for both eight byte and 1Kbyte messages) is shown in Table 5.3. As can be seen, the latency of the best-effort task varies by multiples of period (5,000 μs), while the latency of the time-based task varies by a few microseconds only.

Table 5.3: Latencies of Best-effort and Time-Based Tasks

Message Number	*8 bytes*		*1K bytes*	
	Best-Effort (μs)	*Time-Based (μs)*	*Best-Effort (μs)*	*Time-Based (μs)*
1	423	255	889	610
2	428	262	5899	603
3	435	258	10893	607
4	5427	257	5890	612
5	5433	265	5885	608
6	10431	261	15881	605
7	15429	259	5886	610
8	5423	258	892	603
9	10433	262	10893	602
10	5426	260	5886	609

Consequently, the average latency for the best-effort task is considerably higher than the average latency for the time-based task. Again, the emphasis of this experiment was not on the absolute values of latencies, but rather on the fact that best-effort tasks inherently incur more latency than time-based tasks.

5.6 Summary of Results

All experiments yielded expected results and confirm both parts of the hypothesis. The three-node-ring experiment demonstrated that zero-sided communication is possible to achieve in practice given the right middleware support. The values from the TURTLE built-in profiler also confirm realization of zero-sided communication. Experiments on the performance metrics showed that HARE is able to achieve low latency jitter, though the latency is higher than BDM-RT. The experiment that compared the performance of time-based and best-effort tasks indicated that the latency and latency jitter of time-based channels are better than that of best-effort channels.

CHAPTER VI

OTHER OBSERVATIONS

This chapter identifies some gray areas in the existing MPI/RT 1.1 standard and offers some possible suggestions.

- The MPI/RT standard does not deal with scheduling of computation threads. This scheduling often needs to be done with support from the underlying OS and so is platform dependent. Also, the arrangement for scheduling the computation thread usually needs to be done inside the MPI/RT program. As a result, the program (in an as is state) is not portable and that only specific portions of the program are portable across architectures. Given that portability is an important design goal, this appears to be a drawback. If the standard can couple scheduling of the computation thread with the scheduling of communication threads, the MPI/RT programs can be made portable.

- MPIRT_Commit is a non-real-time function. The implementation is free to take as much time as required to complete the admission test. The standard also goes on to state that all periods will be synchronized at the end of commit (commit return time). This indeterminism on the start of the periods makes appropriate scheduling of the computation threads difficult. If MPIRT_Commit can return a time in future (epoch_start_time), when all periods will start, the computation thread can use this information to schedule itself appropriately.

- Currently, in the time-driven paradigm, the endpoints are required to call the MPIRT_Activate to transition the channel to a state where it can participate in the data transfer operation. There are two issues with respect to this kind of arrangement as follows:

    1. When performing zero-sided communication, it becomes important that both sides call MPIRT_Activate concurrently. Since these calls are made on different nodes, this synchronization is possible only if the nodes can enter into a sort of agreement about the calling time (even in the presence of a global clock). This information cannot be preplanned as MPIRT_Activate can be called only after MPIRT_Commit and MPIRT_Commit function return time is indeterminate. One solution to this problem could be to create channels just for control data and subsequently using the control channels to exchange data for MPIRT_Activate call times. Though possible, this solution could introduce many complexities. An alternative solution could be to make MPIRT_Activate a collective call, but this appears to be an acceptable solution only for periodic transfers where the one-time synchronization cost is amortized over a large number of message transfers. For a single transfer, using a collective MPIRT_Activate does not appear attractive.

    2. The behavior of how the implementation should handle the scenario when MPIRT_Activate is called in the middle of a period (when there is not sufficient time for data transfer during that period) is not defined by the standard.

- Currently, the periods for all the time-based tasks are assumed to start at the same time, which is the commit end time. The standard could have been more flexible if it allowed different period start times (epoch_start_times) for the different time-based tasks. The channel QoS can have an absolute start_time – it currently has a release time, but this has to be relative to the period if the period is specified.

CHAPTER VII

CONCLUSIONS

7.1 Summary

This thesis hypothesized that zero-sided communication is achievable in practice with the correct middleware support. This hypothesis also claimed that using such a zero-sided communication for time-based hard real-time tasks yields better performance in terms of latency and latency jitter as compared to best-effort tasks.

The motivation, hypothesis, and the basis for this thesis were presented in the first chapter. The primary motivation for this thesis was the development of HARE, the first known prototype implementation of the subset of MPI/RT 1.1 standard with time-based QoS. HARE was developed over other two in-house components – TURTLE (Apte et al. 1999), a real-time scheduling variant of RT-Linux, and BDM-RT (Chakravarthi 2000), a low-level real-time messaging layer over Myrinet. Together, TURTLE, BDM-RT, the global clock module, and HARE comprise the PromisQoS architecture. Successful achievement of zero-sided communication with BDM-RT, TURTLE and the global clock module provided the necessary basis for the thesis.

The literature review in chapter two identified the mechanisms used by the low-level messaging layers to achieve high performance and the real-time layers to achieve predictable performance. The review also focussed on identifying a suitable low-level real-time messaging layer that HARE could be layered. The review revealed the unsuitability of FM-QoS (Connelly and Chien 1997), GM-RT (Zan 2000), and RT-

Mach (Lee et al. 1996) for layering HARE. BDM-RT (Chakravarthi 2000), though with some small drawbacks, proved to be the successful candidate.

The third chapter characterized the different 'sidednesses' of communication as mentioned in the MPI/RT standard (MPI/RT Forum 2001). Two-sided communication requires explicit message transfers for flow control and usually uses three-way handshake for the data flow. Best-effort tasks map well with two-sided communication primarily because of the inherent unpredictability in their message passing patterns. One-sided communication might be accomplished either by applications using early binding or by having the middleware do a two-sided communication that is transparent to the application. Zero-sided communication necessitates early binding and consequently removes all delays resulting from handshaking, synchronization and system calls.

The message-passing time for each of the above was also quantitatively described in chapter three. It was shown that the message passing time is less for zero-sided communication than one-sided and two-sided communication models. It was also explained in chapter 3 that the latency jitter is theoretically much less for time-based zero-sided communication than the other two models.

Chapter 4 discussed the research methodology of the thesis and explained the implementation details of HARE. HARE was developed as a prototype implementation of a subset of the MPI/RT standard with time-driven QoS support. HARE spawns a TURTLE task for every time-driven channel created by the application. The QoS channel parameters provided by the application are translated into TURTLE task parameters at commit time by HARE. The latency of a simple send-receive program using HARE is a function of many parameters including the the scheduling overhead of TURTLE, the deadlines for the different user tasks, channel

tasks, protocol processing times, and the actual network latency. The actual value for theoretical latency was derived.

The validity of the hypothesis was demonstrated in chapter 5. Zero-sided communication was accomplished with the ring application by a careful choice of parameters for the computation and communication threads. The start and stop times recorded by the built-in TURTLE profiler corroborated the results. The latency and latency jitter for different message sizes were also measured and explained with the theoretical latency equation derived in chapter 4. The final experiment compared the performance of best-effort tasks with that of time-based tasks. Best-effort tasks were simulated in the PromisQoS environment by generating a random number of messages during each period. The results of this experiment clearly establishes that the latency and latency jitter for time-based tasks using zero-sided communication are considerably less than that of best-effort tasks, thereby validating the hypothesis.

Chapter 6 identified some gray areas in the MPI/RT 1.1 standard and also offered some possible modification for the QoS specifications of the standard.

7.2 Future Work

The PromisQoS components, HARE, BDM-RT, TURTLE and the global clock module provide together the necessary architecture for developing and executing hard real-time tasks with time-driven QoS. However, HARE is a proof of concept implementation and is in a preliminary stage. This section mentions some future work for this project.

The global clock module in its existing state of development can provide accuracies of ± 5 μs only if all the nodes are connected to a single switch. If the module needs to scale across multiple switches, additional functionalities will have

to be incorporated to the BDM-RT module. BDM-RT, unlike many other messaging layers over Myrinet, does not do an automatic mapping of nodes at startup time. Rather, the routes are hard-coded, which needs to be generalized. This needs to modified. Secondly, the time broadcast follows a star pattern. When the number of nodes are increased by an order, this star pattern might not scale well. Alternative approaches like a tree-based broadcast might have to be considered.

One of the important components of latency in HARE is the delay induced by the additional copy involved, from the user buffer to BDM-RT buffer and vice versa. Future versions of HARE should try and provide the BDM-RT buffers directly to the user application (when the user uses *system_mem_alloc* option). This feature would obviate the need for additional copy at least in some cases. A concurrent activity should be to modify BDM-RT to increase its queue size to provide more such buffers.

TURTLE, as mentioned earlier, supports Linux as the only greedy task. This makes implementation of non-time-based channels difficult in the PromisQoS architecture. The modification of TURTLE to allow other greedy tasks is a potentially rewarding future work.

Admission control is not implemented in TURTLE. This is an absolutely important requirement for any MPI/RT implementation supporting time-based QoS. This functionality needs to be added soon. However, it should be mentioned that admission control is already an ongoing research (as a part of dissertation of two researchers) in the High Performance Computing Laboratory.

One other practical problem faced during the development of HARE was the specification of the compute-times parameter for TURTLE. TURTLE seems to require a minimum compute time of 50μs, irrespective of the actual computational

requirement. This fine-tuning of TURTLE with regard to the specification of parameters is a pending activity for others to pursue.

An interesting functionality to add to HARE would be that of providing the actual compute times for the channel tasks back to the user, either as a return value of a non-standard API or through some shared memory. Providing such a functionality would be of immense help while arriving at the final QoS specifications for the application. Resource conscious, adaptive algorithms/applications could then be explored.

The existing PromisQoS architecture has been developed and tested with hardware that is far from the latest. It would be interesting to port the work to the latest hardware and calculate the latency and latency jitter. With processor speeds that are ten times faster than that of the processors in the test bed and the latest Myrinet NICs, there should be an appreciable improvement in the latency and latency jitter. Some porting is evidently already underway.

REFERENCES

Apte, M., S. Chakravarthi, J. Padmanabhan, and A. Skjellum. 2001. A synchronized real-time linux based myrinet cluster for deterministic high performance computing and MPI/RT. In *Proceedings on the workshop on Parallel and Distributed Real-Time Systems held in San Francisco, CA, April 2001.*

Apte, M., S. Chakravarthi, A. Pillai, A. Skjellum, and X. Zan. 1999. Time-based linux for real-time NOWs and MPI/RT. In *Proceedings of the Real-Time Systems Symposium held in Phoenix, Az, December 1999.*

Baker, M., and R. Buyya. 1999. Cluster computing at a glance. In *High Performance Cluster Computing*, edited by R. Buyya, 3–45. New Jersey: Prentice Hall.

Barbanov, M., and V. Yodaiken. 1996. Real-time linux. *Linux Journal* (March).

Boden, N. J., D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. 1995. A Gigabit-per-Second Local-Area Network. *IEEE-Micro* 15(1):29–36.

Chakravarthi, S. 2000. Predictability and performance factors influencing the design of real time messagine layers. Master's thesis, Mississippi State University.

Chakravarthi, S., A. Pillai, J. Padmanabhan, M. Apte, and A. Skjellum. 2000. *A fine-grain clock synchronization mechanism for QoS based communication on myrinet.* Mississippi State University. Technical Report 20001001.

Connelly, K., and A. Chien. 1997. Fm-qos: Real-time communication using self-synchronizing schedules. In *Proceedings of SuperComputing held in San Jose, CA, December 1997.*

Gropp, W., E. Lusk, and A. Skjellum. 1999. *Using MPI: Portable parallel programming with the massage passing interface.* London, England: MIT Press.

Henley, G., N. Doss, T. McMahon, and A. Skjellum. 1997. *BDM: A multiprotool myrinet control program an host application programmer interface.* Mississippi State University. Technical Report MSSU-EIRS-ERC-97-3.

Lee, C., K. Yoshida, C. Mercer, and R. Rajkumar. 1996. Predictable communication protocol processing in real-time mach. In *Proceedings of 2nd Real-Time Technology, and Applications Symposium, held in Japan, June 1996.*

Liu, C. L., and J. W. Layland. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM* 20(1):46–61.

MPI Forum. 1994. *MPI forum: Document for a standard message passing interface.*

MPI/RT Forum. 2001. *MPI/RT: Real-time message passing interface standard 1.0.* http://www.mpirt.org/ Last accessed November 23, 2001.

MSTI. 2001. *MPI software technology inc.* http://www.mpi-softtech.com Last Accessed: November 20, 2001.

Myricom. 1998. *GM.* http://www.myri.com/GM/doc/gm_toc.html Last Accessed: July 20, 1999.

Pakin, S., M. Lauria, and A. Chien. 1995. High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of Supercomputing held in San Diego, CA, December 1995.*

Stankovic, J. A., and K. Ramamritham. 1990. What is predictability for real-time systems? *Journal of Real-Time Systems* 2(December):247–254.

Stankovic, J. A., and K. Ramamritham. 1993. *Advances in real-time systems.* Los Alomos, CA: IEEE Computer Society Press.

Tsai, J., Y. Bi, S. Yand, and R. Smith. 1996. *Distributed real-time systems: Monitoring, visualization, debugging, and analysis.* New York, NY: Wiley-Interscience.

Zan, X. 2000. A real-time messaging layer over mrinet network. Master's thesis, Mississippi State University.

APPENDIX A

SUBSET OF MPI/RT API'S IMPLEMENTED

The subset of the MPI/RT 1.1 standard (MPI/RT Forum 2001) that has been implemented in HARE is given below:

1. Object
   (a) MPIRT_object_is_null
   (b) MPIRT_object_get_object_type
   (c) MPIRT_object_get_name
   (d) MPIRT_object_set_name
   (e) MPIRT_object_free
2. Container
   (a) MPIRT_container_access_count
   (b) MPIRT_container_get_name
   (c) MPIRT_container_set_name
   (d) MPIRT_container_free
3. Cset
   (a) MPIRT_Cset_create
   (b) MPIRT_Cset_insert
   (c) MPIRT_Cset_remove
   (d) MPIRT_Cset_get_name
   (e) MPIRT_Cset_set_name
   (f) MPIRT_Cset_free
4. Buffer
   (a) MPIRT_Buffer_create
   (b) MPIRT_Buffer_get_bufsize
   (c) MPIRT_Buffer_set_bufsize
   (d) MPIRT_Buffer_get_base
   (e) MPIRT_Buffer_set_base
   (f) MPIRT_Buffer_get_dataspec
   (g) MPIRT_Buffer_set_dataspec

(h) MPIRT_Buffer_get_name
(i) MPIRT_Buffer_set_name
(j) MPIRT_Buffer_free

5. Buffer Iterator

(a) MPIRT_Bufiter_create
(b) MPIRT_Bufiter_insert
(c) MPIRT_Bufiter_remove
(d) MPIRT_Bufiter_free

6. Channel

(a) MPIRT_Channel_get_inbufiter
(b) MPIRT_Channel_set_inbufiter
(c) MPIRT_Channel_get_outbufiter
(d) MPIRT_Channel_set_outbufiter
(e) MPIRT_Channel_get_qos
(f) MPIRT_Channel_set_qos
(g) MPIRT_Channel_start
(h) MPIRT_Channel_wait
(i) MPIRT_Channel_activate
(j) MPIRT_Channel_deactivate
(k) MPIRT_Ptchannel_create

7. Qos

(a) MPIRT_Qos_channel_time_create

8. Setup

(a) MPIRT_Init
(b) MPIRT_Finalize
(c) MPIRT_Commit
(d) MPIRT_Uncommit
(e) MPIRT_Is_initialized
(f) MPIRT_Is_finalized

APPENDIX B

BDM-RT AND TURTLE API'S USED BY HARE

The following BDM-RT and TURTLE API's were used for the implementation of HARE:

1. BDM-RT
   (a) BDM_Frame_malloc
   (b) BDM_Frame_Free
   (c) BDM_Frame_Blocking_send
   (d) BDM_Frame_Blocking_recv
   (e) BDM_Frame_send
   (f) BDM_Frame_recv
2. TURTLE
   (a) rt_task_wait
   (b) rt_task_init
   (c) rt_task_make_periodic