1997

Standardization of a Communication Middleware for High-Performance Real-Time Systems

Arkady Kanevsky
The MITRE Corp., arkady@mitre.org

Anthony Skjellum
Mississippi State University, NSF ERC, tony@erc.msstate.edu

Jerrell Watts
Syracuse University, Electrical Engineering and Computer Science, jwatts@scp.syr.edu

Standardization of a Communication Middleware for High-Performance Real-Time Systems

Arkady Kanevsky[*]
The MITRE Corp.
202 Burlington Rd.
Bedford, MA 01730-1420
e-mail: arkady@mitre.org

Anthony Skjellum[†]
Mississippi State University
NSF ERC, 2 Research Blvd.
Starkville, MS 39759
e-mail: tony@erc.msstate.edu

Jerrell Watts
EECS Dept./2-120 CST,
Syracuse University
Syracuse, NY 13244-4100
e-mail: jwatts@scp.syr.edu

Abstract

*The last several years saw an emergence of standardization activities for real-time systems including standardization of operating systems (series of POSIX standards [1]), of communication for distributed (POSIX.21 [10]) and parallel systems (*MPI/RT *[5]) and real-time object management (realtime CORBA [9]).*

*This article describes the ongoing standardization work and implementation of communication middleware for high performance real-time computing. The real-time message passing interface (*MPI/RT*) advances the non-real-time high-performance communication standard Message Passing Interface Standard (*MPI*), emphasizing changes that enable and support real-time communication, and is targeted for embedded, fault-tolerant and other real-time systems.* MPI/RT *is the only communication middleware layer that provides guaranteed quality of service and timeliness for data transfers, is also targeted for real-time CORBA to replace RPC layer and for real-time and embedded JAVAs.*

1 Introduction

Over the past several years, many standards that address real-time issues have emerged. They address networking: SAFENET [3], Futurebus+ [11], and extensions to FDDI, ATM, Token Ring, Token Bus, and others [2]; communication: real-time message passing interface (MPI/RT) and realtime distributed system communication (POSIX.21); operating systems: realtime POSIX (POSIX.1b, POSIX.1c [1], POSIX.1d, and POSIX.1j); and realtime object management (realtime CORBA). This article presents MPI/RT, the real-time message passing interface, for high performance applications.

The approved MPI-1 standard provides point-to-point communication, collective operations, process groups and communication domains, process topologies, environment management and inquiry [7], formulated within language-independent specifications, together with C and FORTRAN API bindings. MPI-2, which was standardized and published in June 1997, provides additional functionality over MPI-1 in the areas of process creation and management, one-sided communication, collective operations, external interfaces and I/O. It also provides a C++ binding for MPI-1 and MPI-2 functionality.

By way of contrast, the main goal of MPI/RT is to provide message-passing functionality with quality of service (QoS) for development of real-time applications with performance portability. The parameters of QoS include a variety of fault-tolerant and real-time application requirements. Since many high-performance real-time applications would like to take advantage of MPI functionality but require timing guarantees from the message-passing layer, the MPI/RT working group was created with the objective of providing an appropriately designed application programming interface (API). MPI/RT follows MPI's underlying assumptions of reliable and ordered data transmission; programming assumptions, that are common to a majority of parallel environments and platforms that are targeted by MPI/RT. MPI/RT adds greater predictability and schedulability to message-passing programming, while modifying

[*] This work was supported in part by the U.S. Air Force Electronic Systems Center and performed under MITRE MOIE Project 03977450 of contract F19628-94-C-0001, managed by Rome Laboratory/C3CB.

[†] This work was supported in part by the U.S. Air Force Rome Laboratory under DARPA Order D350 and E339, contracts F30602-95-1-0036 and F30602-96-1-0329.

and extending the useful concepts embodied in the original standard.

The rest of the paper is organized as follows. Section 2 presents the underlying philosophy of MPI/RT. Section 3 presents the common underlying layer for all real-time models including buffer and queue management abstraction, channel management abstraction, and event handler abstraction. Section 4 presents communication paradigms and real-time models, and section 5 presents the current status of the MPI/RT standard and future plans.

2 MPI/RT Philosophy

Currently, application developers must become experts on a platform before they can take advantage of its message-passing facilities in order to achieve the desired performance. The challenges are even greater for developers of real-time applications that are required to satisfy timing constraints and proper interaction with the environment independent of the computing platform. The application design is often so dependent on the computing platform that it requires complete redesign when ported to a different platform or targeted for the next-generation platform.

This approach hinders the portability of an application to a different platform or upgrades on the currently used one. The current philosophy is that the platform provides the user with an API and places the burden on the application developers to satisfy timing and quality of service requirements. This philosophy is contradictory to the "portability viewpoint" and MPI/RT has consequently taken the opposite approach. Under MPI/RT, the user provides detailed information about timing constraints of application modules and the interactions between them including message-passing data and control message exchanges. The user's requests are analyzed by the platform, including middleware of which MPI/RT is a part, and either satisfies them with user required QoS or states that it cannot satisfy the user requested QoS. The denial of a request usually results from a lack of platform resources.

MPI/RT supports the view that middleware and platform designers have greater insight into how efficiently to provide QoS on the platform given enough information about the application. With this approach application programmers can concentrate on improving application code and let middleware providers concentrate on providing the best QoS available on the platform. Application programmers are not required to reveal all the information to MPI/RT and can take it upon themselves to provide some or all QoS. It is quite clear that the exact boundary of the responsibility for providing QoS for the user between the platform (including system software and middleware) and the application is still unknown, but the same trends that lead to the development of higher level languages, operating systems, and middleware, are pushing the development of MPI/RT.

In order to provide the quality of service guarantees for communication, an MPI/RT implementation may need to address a difficult scheduling problem. While there is a lot of work going on in CPU and network real-time scheduling, these results in many cases are insufficient to provide guarantees for communication. The number of resources that are involved in communication is rather large and is different from one platform to another. These resources can have their own schedulers that may use completely different techniques, like prioritities for CPUs, round robin for network switches, and interrupts and signals for network interface chips.

However, it is hard, if not impossible, for the application programmer to coordinate the use of these resources in order to establish user-required quality of service even with the complete knowledge of the application. Furthermore, even if it was done successfully on one platform, it cannot be ported to a different platform because of the differences between platform architectures. MPI/RT implementors have a better chance of meeting the user's quality of service requirements because of their knowledge of their platform, since for most cases they work closely with or are part of the same organization that designed and built the platform.

In order to improve the chance for satisfying user quality of service requests, MPI/RT recommends *early binding.* Many of the highly demanding, real-time parallel applications are characterized by the periodic nature of the environment outside the computing platform, and for these applications establishing communication channels with QoS (see section 3) promises the greatest benefits. Using application information about the communication patterns and QoS requirements, MPI/RT implementations can allocate resources using an algorithm and run-time scheduling criteria that are most suitable for the platform prior to the actual data transfers. This allows an implementation to minimize the critical execution path for message passing and the overhead of MPI/RT implementation, so the message passing performance using MPI/RT will come close to the platform native message passing performance and, hence, the so-called "price of portability" will be minimized.

3 Common Functionality

Due to the lack of space we just outline the supporting functionality without any details (for details see [5]). The supporting functionality contains a synchronized clock definition with detail specifications for resolution, drift, skew, accuracy and access time parameters; an instrumentation for MPI/RT and user functionality, and a fault handling.

3.1 Channels

In MPI/RT, persistent channels offer the functionality of a virtual channel [4, 8] within the framework of the MPI standard. Motivations for having virtual channels in MPI/RT include: ability to exploit persistent communications that are common for high performance real-time applications, deadlock and livelock avoidance, virtual channels guarantees for properties critical for timing correctness, and more efficient resource usage by the implementations.

MPI/RT, as a specification and programming notation, encourages early binding in order for the implementations to establish user-required quality of service, while providing both early and late bindings for data transfer operations. The initialization of the channels collectively provides MPI/RT with the big picture of application-desired, point-to-point channels and their respective QoSs. The early knowledge of all the point-to-point channels allows MPI/RT implementation to exploit potential flexibility in satisfying individual channels QoS rather than establishing each channel individually and making arbitrary decisions in the process, that may be detrimental to MPI/RT's ability to satisfy all channels QoSs. This approach is not required to be done prior to any data transfer operations, but is strongly encouraged to maximize MPI/RT's potential performance. The channel establishment operations as well as channel modifications and deletions, can be used at any time, but these operations are expensive and it is harder for the implementation to satisfy later requests and to optimize resource usage, especially if these requests are relatively frequent.

Following the MPI principle that all communications are done over a communicator (clique or bipartite group formulation), group-oriented MPI/RT channel initialization operations are done over a communicator. The same application process can participate in more than one communicator group and by default all processes are members of one communicator MPI_COMM_WORLD. Hence, a process can participate in channel initialization for more than one communicator. The MPI/RT standard is silent on how the above established channels are mapped on the network channels. This is left to the implementation and is highly dependent on platform architecture, network topologies, routing information, etc. The solutions that shared memory platforms would like to use, may not be applicable to the distributed memory platforms and vice versa.

While suppressing the entire syntax of the collective point-to-point channel initialization operations for brevity, we would like to stress several parameters that carry semantic information. First, the operations allow specification of information for all point-to-point channels over a single communicator the process would like to use. This includes several point-to-point channels between the same pair of processes. Using this specification, an application can establish any virtual topology between processes. The operation returns a request handle for each channel. Instead of providing separate operations for creation, modification and destruction of the channels, MPI/RT has a single operation that combines all channel management functionality into one atomic operation. This allows application not to destroy existing channels if new/modified channels cannot be established with the requested QoS, and hence, preserve existing channels and resources they are using.

MPI/RT also provides functionality to establish collective channels with quality of service. These play the same role for collective operations (like scatter, gather, broadcast, all-to-all scatter-gather) as point-to-point channels for individual send/receive operations. The specification of the quality of service, buffers and other data may differ from one collective operation to another.

Each channel is specified by quality of service parameters, message buffers, buffer iterators and handlers that can be used for QoS and other errors. In order to simplify the application specification of the channels information, MPI/RT adopted the object-oriented design methodology of cloning and composition. An application uses the hierarchy of the objects where an object include both an object descriptor and a handle to the "physical" object. Uncommitted objects only have an object description without a handle to the actual object; these uncommitted objects collect the channel information for all channels. Once the information is collected for all the channels over the same communicator into a channel set, a single construction operation creates all the channels and channel objects that include: channel buffer iterators, buffers, handler handles, channel handles, and a channel set handle. Object operations are also defined by the MPI/RT standard that allow user create objects, "shallow" duplicate committed and uncommitted objects, and to query and set individual parameters of uncommitted objects to simplify the job of channel specification definition. The same object methodology is used by MPI/RT for QoS objects, events objects, and handler objects for both user and error handling.

The channel QoS specify timing and triggering requirements of either one of the real-time models that for which user request system guarantees, or a "softer" quality of service that does not provide an absolute guarantee for each data transfer. The detail QoS parameters of each model are presented in the Paradigm and Models section 4. Since no guarantee can be absolute (hardware and software faults) the channel initialization operation allows users to specify error handlers that will be invoked by MPI/RT when the data transfer quality of service is not achieved. This is a part of the generic functionality MPI/RT provides for an application fault-handling.

3.2 Buffer and Queue Management

The buffer set and queue management specification allows an implementation to minimize message copying and more efficient use of memory by application and implementation. The main difficulty in buffer management specification comes from the requirement that the same specification should support both implicit (time-driven and event-driven) triggering of message transfers and explicit message transfers which are the most common communication paradigms today.

The buffer pool is just a collection of the memory pieces (buffers), where each buffer has the same length, the same datatypes and application view layout. The buffers can be allocated by the users prior to an establishment of the buffer pool or by the system at the request of the user at the channel creation time. The latter allows implementation to allocate buffers from memory that system uses for message transfers rather than just from user space.

For each end of a channel user specifies two iterators. One is *in*-iterator that specifies the ordered collection of buffers ready to receive a message from the channel or the user. Another is *out*-iterator that specifies the ordered collection of filled buffers ready to be delivered to the user or the channel. For the sending side of the channel the buffer circulates from in-iterator (initially), out-iterator (upon receiving message from the channel), to user (upon explicit application request, or implicitly upon time instance or prespecified event), and back to the in-iterator again (upon explicit application request, or implicitly upon time instance or prespecified event).

The buffer iterator is defined over a subset (or the full set) of a buffer pool. The buffers are managed by the implementation on behalf of the application. The buffer iterator specifies the maximum length of the queue and which buffers from the specified buffer pool should be put in the queue initially. Users can also assign a label to each chosen buffer. The labels allows users to group buffers in the iterator together so that any buffer with the same label can be used from the group.

The main parameter of the buffer iterator is policy. The iterator policy defines where the next buffer goes in or taking from the iterator. Currently, the standard specifies four policies:

- *MPIRT_BUFITER_FIFO* specifies a first-in, first-out policy. That is, buffers are taken from the iterator in the order they were put into it.
- *MPIRT_BUFITER_LIFO* specifies a last-in, first-out policy. That is, buffers are taken from the iterator in the opposite order as they were put into it.
- *MPIRT_BUFITER_SORTED* specifies that buffers are ordered from lowest label to highest label. Since users define the labels, they can achieve any order the choose. For example, a priority scheme can be defined by assigning labels in reverse order.
- *MPIRT_BUFITER_UNORDERED* specifies that buffers are not ordered.

The sharing iterators between multiple channels (point-to-point and collective) allows user to set up pipeline processing, data fusion between multiple channels, load balancing, and, in general, various ways users would like to share buffers between multiple channels. To support explicit operations for message transfers, two operations are defined that allow users to insert a buffer into a buffer iterator, and to remove a buffer from the buffer iterator.

3.3 Handlers

The *system event handlers* are a generic mechanism that allows users and implementors to handle events, errors and other conditions that arize during an application execution. The handlers are created by the users or the implementors and are waiting for local events. These events can be arrival of the message over a channel, completion of the data transfer, unfulfilled QoS guarantee, channel errors (hardware or software), buffer iterator overflow, buffer iterator underflow and other MPI/RT or platform-defined events.

The handler mechanism provides the functionality for a request handler and a local event that will be used by a MPI/RT implementation as a trigger to schedule the request. Request handlers are an ideal mechanism for implementing the event-driven model that can be used by both an MPI/RT implementation and an application. This functionality can be used with either MPI or MPI/RT operations' requests. To help users better manage resources, two events for the data transfer completions over a channel (point-to-point or collective) are introduced. One event specifies the local completion of the data transfer, that is when the message buffer can be reused, an event which is currently available on most platforms. The other specifies the global completion of the data transfer, meaning the channel resources can be reused.

The event handler can be persistently posted (*e.g.*, for a channel) or a one-time only posting, with the handler function to be called within the system-wide event triggering QoS (either time duration or priority provides by the operating system in the component where handler reside) after the given request reaches the event condition. This *system event handler* provides a mechanism for scheduling (with QoS) an application handler upon the completion of a data transfer operation (implicit polled delivery). When the handler is called, it is passed the object handle and the condition that causes the handler invocaction, and the input parameters for the handler. If the condition handler cannot be called within the specified QoS then the handler failure handler is called.

4 Paradigms and Models

MPI/RT provides support for three application message passing paradigms. The first is the most commonly used *two-sided* communication. It is characterized by the application issuing data transfer operations for two sides of the data exchange. This paradigm is commonly called *send-receive.* The second *one-sided* communication paradigm allows only one side of the application data exchange to issue data transfer operations. The most commonly used one-sided operations are *put* and *get.* The last data transfer paradigm is *zero-sided* communication. It is characterized by the absence of any data transfer operation by either side of the data exchange. It is the responsibility of a communication service to move the data from/to prespecified application memories, at prespecified times, or in response to certain events. With the pre-established early binding channels it is possible to exploit "no-sided" communication where the middleware (MPI/RT) does the data transfer operations on behalf of the application at the predetermined times [6] that are part of the channel QoS.

The three real-time models presented in this article span the most commonly used real-time programming models: time-driven, event-driven and priority-driven. The primary goal of all real-time MPI/RT programming models is to allow a real-time application sufficient control of the environment in which it is running so that it can explicitly or implicitly schedule its message-passing activities and resource usage. Since MPI, the underpinning of MPI/RT, is designed as a message-passing library, it cannot schedule by itself, but must depend upon the operating system and communication and network protocols to enforce specified schedules. While the time-driven and event-driven models specify explicit schedules, the priority-driven model specifies implicit ordering for message passing activities.

The specification of the real-time models either states an application QoS requirements for data transfer, or triggering mechanisms for data transfer, or both. The time-driven model parameters specify both triggering mechanism for data transfer (start and stop) and application QoS requirements (deadline). The priority-driven model parameters only specify only QoS requirement (priority), while event-driven model parameters only specify triggering events for data transfer (start and stop). Because of the lack of full data transfer specifications for the last two real-time models the mixture of the models are used. The two most common mixed specifications and event and time-driven one, and event and priority-driven one. In both mixtures, the full set of QoS and triggering mechanisms are now defined for a channel for persistent ot one at a time data transfers.

4.1 Time-Driven Paradigm

An application using time-driven MPI/RT QoS will be able to specify time intervals to bound the resource usage of communication operations using globally synchronized clock values.

The existing MPI message transfer operations lack two parameters that we consider critical for real-time applications. These are a starting time of the operation and a timeout for completion of the operation. The starting time of an operation and the timeout should be considered special cases of an event. While certain applications (especially embedded ones) prefer an even finer granularity of control, we sought to strike a balance between the feasibility of an implementation and what time-driven application designers want to use. For example, there is a hard lower bound for the starting time, but no hard upper bound on the starting time, in the current specification.

One distinctive characteristic of the time-driven approach to real-time message-passing is its lack of need for queues and system buffers. On many systems, this allows the removal of a hand-shake operation and results in improved performance. Since a parallel time-driven program must globally schedule all message transmissions, the message receiver always knows when to expect an incoming message. Thus, for reasons of efficiency and simplicity, a time-driven MPI/RT implementation should not do any handshaking (as many of the existing non-real-time implementations do). It is rather up to the application to specify times (for start and timeout) to ensure that the sender/receiver pairs are working in synchrony.

Another distinctive feature is a potentially more efficient way of using notifications, which can be more minimal (shorter critical instruction path) than with other approaches. A time-driven MPI/RT application does not need to be notified when a message is transmitted successfully and on time; instead it is notified only when an error occurs (*e.g.*, a timeout expires).

An activity interval, specified by a starting time and a timeout, is an input parameter for a scheduled message send. The purpose of this parameter is to ensure that the system resources required to satisfy this operation will not be used outside of a specified interval. These resources can be narrowly interpreted to refer to the interprocess communications network. A broader interpretation would include memory accesses, node busses, network interface cards, and so on. Again, while we prefer a finer granularity of control, we have tried to strike a balance between the feasibility of an implementation and what time-driven schedule designers want to use.

The starting time and timeout are somewhat symmetric. The starting time ensures that the resources needed for a data transfer operation will be available at the specified start time. The timeout parameter, in contrast, would ideally specify the time when all resources required by the message transfer operation are no longer in use. That is, after the time specified in the timeout, irrespective of whether the operation completed successfully or not, all system resources (physical network, network interface cards, node buses, message buffers, etc.) have been released and can be used for subsequent message-passing operations.

Unfortunately, in practice these guarantees cannot always be met. The MPI/RT timeout therefore specifies that the message transfer should be stopped and the calling application should be notified if the operation has not completed by the time specified by the timeout. Since the message may be progressing through a multi-stage network, a time-driven MPI/RT implementation may need to send a message from the receiver node to the sender to indicate that the timeout has occurred. The resulting error messages may not be received by the timeout deadline, and they may use resources after the timeout. Thus the application may need to reserve resources to handle such events. It should not be the responsibility of the MPI/RT implementation to provide this bound, since any guarantees that can be given from the perspective of a user-level message-passing library would be too naive to be useful. The application itself is in a much better position to know timing and performance details relevant to establishing such a bound, including details of the platform and knowledge of the run-time patterns of communication. Even for the application, it may be extremely difficult to establish such bounds, especially if the real-time performance characteristics of the operating system or the underlying runtime system are poorly known or highly variable. The situation is even more difficult for collective data transfer operations that use time-driven model.

The starting times and timeouts of the activity interval in time-driven MPI/RT data transfer operation calls are specified either as an *absolute* time instance of the synchronized clock, or a *relative* to the current synchronized clock value. In the latter case the actual scheduling time is derived by adding the relative value to the most recent reading of the global synchronized clock.

4.2 Event-Driven with Priority Paradigm

The *event-driven model* supports the specification of events that either trigger or stop an application or MPI/RT data transfer operations. The event-driven model provides a mechanism for scheduling any application activity with QoS, including an MPI/RT data transfer and an application function triggered by a system, an application, or an MPI/RT event. This model allows users to synchronize and manage MPI/RT, system, and user resources using events.

The event-driven model does not have an explicit quality of service the same way as deadline provides for time-driven model. Consequently, it is most commonly used in conjunction with the priority-driven model (that specifies an integer priority of the channel for the data transfer operations), or with the time-driven model (that specifies the deadline as an event relative to the start of the data transfer) models.

In MPI/RT, priorities are specified *per channel* as part of the the channel QoS. Because varying platforms may provide different levels of support for message priority at the OS level and below, MPI/RT specifies little about how message priorities are implemented. In addition to passing message priority information to the appropriate OS and hardware layers, a high-quality MPI/RT implementation will order operations internally according to priority information. For example, given the choice between performing two different communication operations (such as receiving one message or another), the higher priority communication should be performed first. If the high priority communication blocks or stalls, lower priority communication may be initiated. Notice that in the general case, this implies that communication may need to be preempted. MPI/RT makes no attempt to correlate process (or thread) and message priorities.

The only explicit QoS for the event-driven model is the bound required on starting time of the user activity that is guarded by an event, which is more a requirement for the operating system where the MPI/RT implementation is running. Additional specification is under consideration that allows users to provide the bound on the number of events over some time interval. This is similar to most specifications for aperiodic tasks [4].

In a nutshell, an application using event-driven MPI/RT will be able to specify intervals *guarded* by the specified events in order to bound the resource usage of communication and computation activities. Coordination is required between MPI, the operating system, and communication and network protocols to enforce the schedules.

Currently many applications "wait" on system events or user control messages to schedule a handler, that in turn schedules several application activities: functions, processes, threads, and data transfers. The model for the event-driven model presented in this section establishes the direct coupling between events and application activities without user handlers. Just as MPI provides the interface for data flow, the high level event-driven section provides the interface for control flow. The events can be both persistent and one-shot only. Three types of events are specifiable: system events, communication events, and user events. Each event is identified by name. A name is associated with a persistent event. The type of event indicates the type of the resource that generated the event. System events are generated by the platform environment, for example the operating system. Communication events are coupled with persistent channels and are generated by MPI/RT. User events are dedicated to the synchronization of the resource usage among different processes (nodes) on the platform, and are generated by the application.

Events are not necessarily local to the process or even a node. Each process registers the persistent event names with MPI/RT that it wants MPI/RT to "monitor" and the persistent event names that the process will generate.

All the communication events are associated with the MPI/RT channel usage. Currently, the specification document contains only two events associated with the channel: local and global communication completion. In order to match these events with the guarded activities properly, MPI/RT associates a persistent global name with a channel. The channel name can be either provided to an implementation by the application or the implementation will assign a name to a channel. Hence, there are two persistent event names associated with the channel. For a channel named α they are: α*_local_complete* and α*_global_complete*. The user can provide the channel names and MPI/RT will assign them to the channels, or the user can request MPI/RT to provide the channel names and return them as an out parameter. The names on both endpoints of the channel must match.

User events have meaning only to the application. MPI/RT is just a mechanism to match user events and responses as well as the mechanism for event delivery and response triggers. An application assigns a persistent name to a user event and notifies MPI/RT about which process generates this event. This is the only event type that is generated by the user. The events of two other event types are generated by MPI/RT and the system. MPI/RT delivers all the events to the processes that are registered for them and then triggers application functions or data transfers according to the events that guard the activity.

For any function or communication operation, an application can specify events that trigger its start and its termination if it is not finished. Events "guard" a liveliness interval within which the activity can use resources. The guards use two lists. The first one is the list of events whose conjuncture trigger the activity. The second one is the list of events, such that any event on the list stops the activity if it is not yet finished by itself. For completeness an empty list is defined. The well-known priority-driven model can be specified using empty guards and a priority for the channel.

The event-driven guards are analogous to the time-driven model where no resources will be used by an MPI/RT data transfer operation prior to its starting time of the operation time interval and, to the best of the MPI/RT implementation's ability, no resources will be used after timeout of the operation time interval. The time interval of time-driven MPI/RT contains two events that are specified by time stamps. From this perspective, the time-driven model is just a subset of the event-driven one. There is, however, one critical difference that lies in the ability of the application to schedule its non-MPI/RT activities. For the time-driven model, there are existing facilities to start non-MPI/RT activities using OS timers, spin-locks and others. These facilities and the synchronized clocks allow the application to coordinate all of its activities, MPI/RT and non-MPI/RT, both local and global. There are no analogous mechanisms for the event-driven model, and event delivery/monitoring across the entire platform requires application action and sufficient communication support. This is the place where MPI/RT can really help.

MPI/RT is responsible for delivering events and for triggering (start or stop) an activity if it is eligible. Each application process registers event names it wants MPI/RT to monitor and event names it will generate. Since an application can only generate user events, only user event names that application will generate need to be registered with MPI/RT. MPI/RT is already aware of where and how system and communication events are generated. The issue of how the events are delivered to the guarded activity is left to the implementation. The MPI/RT standard provides the functionality for an application to notify an MPI/RT implementation about application generated events.

5 Conclusions

The MPI/RT standard constitutes the first effort to provide a portable specification for real-time message passing user requirements. It allows the domain of portable message passing high performance parallel computation (MPI domain) to be enlarged to include embedded and time-critical applications. While still not in its final stage, MPI/RT clearly reveals the functionality missing from MPI and different application design approaches that real-time applications are using, and addresses these omissions.

The latest draft of the standard can be found in *http://www.mpirt.org* [5]. One can join the MPI/RT standard working group by sending a message *subscribe mpi-realtime* to *majordomo@mpirt.org*.

Acknowledgements

The authors would like to thank all the members of the MPI/RT working group without whom this work would not been achieved.

References

[1] ISO/IEC 9945-1. *Information technology - Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) [C Language]*, 1996. ANSI/IEEE Std 1003.1, second edition, 1996-07-12.

[2] C. M. Aras, J. P. Kurose, D. S. Reeves, and H. Schulzrinne. Real-time communication in packet-switched networks. *Proceedings of the IEEE*, January 1994. Special issue on Real-Time Systems.

[3] U.S. Department of Defense. *Survivable Adaptable Fiber Optic Embedded Network*, MIL-STD-2204A edition, January 1994.

[4] D. Ferrari. A new admission control method for real-time communication in an Internetwork. In D. Son, editor, *Advances in Real-Time Systems*. Prentice Hall, 1995. Chapter 5.

[5] Message Passing Interface Forum. Real-time message passing interface standard draft. http://www.mpirt.org, October 1997.

[6] Richard A. Games, Arkady Kanevsky, Peter C. Krupp, and Leonard G. Monk. Real-time communication scheduling for massively parallel processors (position paper). In *Real-Time Technology and Applications Symposium*, pages 76–85. IEEE, 1995.

[7] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1994.

[8] A. Mehra, A. Indiresan, and K. G. Shin. Resource management for real-time communication: Making theory meet practice. In *Proceedings of the 2nd IEEE Real-Time Technology and Applications*, pages 130–138, 1996.

[9] Object Management Group (OMG). Realtime distributed systems communication application program interface. ftp://ftp.omg.org/pub/docs/orbos/96-09-02.pdf, 1996.

[10] IEEE Information Technology-Portable Operating System Interface (POSIX). Realtime distributed systems communication application program interface. ftp://ftp.sei.cmu.edu/pub/posix, 1996.

[11] L. Sha, R. Rajkumar, and J. Lehoczky. Real-time computing using Futurebus+. *IEEE Micro*, June 1991.