D R A F T

Document for a Standard Message-Passing Interface

Message Passing Interface Forum

May 30, 1997

This work was supported in part by NSF and ARPA under NSF contract CDA-9115428 and Esprit under project HPC Standards (21111).

This is the result of a LaTeX run of a draft of a single chapter of the MPIF Final Report document.

Chapter 8

Real-Time MPI

Real-Time proposals presented here constitute work in progress. The real-time working group unanimously voted to move the chapter into the MPI-2 Journal of Development in order to gain more time for preliminary implementations and further improvements since the work here standardize an area that does not have common existing practice. The MPI Forum had not voted on any parts of this chapter.

The work contained here is being updated and developed further in public meetings comparable to the MPI Forum, beginning after the conclusion of the MPI-2 Forum's, and scheduled to meet regularly in six week intervals for at least one year. For information, and to participate, see `http://www.cs.msstate.edu/mpirt`, the homepage for MPI/RT.

8.1 Introduction

The goal of real-time MPI (MPI/RT) is to provide the middleware for programmers to create real-time applications with performance portability. MPI/RT is to provide a consistent set of extensions and, in some cases, restrictions to MPI. MPI/RT adds greater predictability and schedulability to message-passing programming.

MPI/RT is intended to provide several core features for the most demanding applications while allowing flexibility for use in a broad variety of applications. The core features are:

1. Message-passing performance guarantees. MPI/RT will provide quality of service (QOS) by allocating resources and providing bounds on the delivery of messages which will allow the use of MPI/RT in time-critical applications.
2. Minimizing the critical path for message passing in a portable environment. Many applications take advantage of machine-specific message passing that provide minimal overhead. In order to be usable for tightly integrated equipment applications, MPI/RT will minimize the overhead relative to MPI-1 and MPI-2 as much as practical.

   (Note: MPI/RT will have the greatest chance of success if it can be designed to provide as little performance penalty as possible while achieving portability.)
3. Early binding. Facilities will be provided to allow MPI/RT resources to be scheduled. Scheduling will allow efficient use of resources. Early binding will allow message overhead in the critical path to be reduced and support such paradigms as "no-sided" communication.

Many applications will only require a subset of the core features or may not be as demanding in their timing requirements. Many features of MPI/RT are provided to allow its use in a wide variety of these applications while not compromising the core features.

MPI/RT is designed to provide high performance along with portability. Portability in the MPI/RT is defined as follows:

- Ability to move a parallel application from one platform to another, or from one configuration to another, and compile successfully without changing the syntax of the parallel program.
- The qualities of service achieved by the real-time application may not be guaranteed in a new configuration or port. By fine tuning the quality of service parameters, the program may be made to run in the new configuration or platform. These changes ensure timing correctness, provided that sufficient resources are present to support the application requirements.
- Upgrading from one generation of a platform to the next is generally expected to work seamlessly, even though resource utilization could be lower on the upgraded platform.

The design philosophy of MPI/RT is as follows:

- MPI/RT will *not* determine implementation policy, but will instead provide middleware to support for real-time paradigms such as:
  - Time-driven,
  - Event-driven,
  - Priority-driven,
  - Best-effort (aka, soft real-time), and
  - Resource constrained.
- MPI/RT should have analogous functions to MPI-1 and MPI-2 that guarantee message passing in a timely fashion.
- MPI/RT will make minimal changes to MPI-1 and MPI-2, so that MPI/RT programs can benefit from existing MPI libraries, at least for non-time-critical parts of real-time applications.
- Efforts will be made to make existing MPI libraries work as seamlessly as possible within these real-time profiles.
- MPI/RT must allow not only code portability but also performance portability, insofar as possible.
- MPI/RT will not replace the native runtime system or scheduler, but will provide a portable means to communicate with these systems.
- For resource-constrained users, the advantages of a layered approach indicate that "subset profiles" bear consideration as part of this effort. The smallest subsets should be minimal to allow the widest possible embedding of MPI-1.2 with and without specific real-time features. A feature-driven rather than call-driven approach to such profiles is indicated. Some real-time, embedded kernels might choose to add the smallest set of features to the kernel, with layering of additional features for less constrained situations (See section 8.3).

- The results of the MPI/RT effort will be a single set of real-time extensions, restrictions, and recommendations, suitable for realization as an MPI/RT implementation. It is expected that users will select operations in order to support the real-time paradigms of their choosing.

- The placement of this chapter in the Journal of Development (JOD) indicates that MPI/RT will be adopted separately from the MPI-2 standard but is related to it wherever appropriate. That is, not all MPI-2 implementations will need to support the real-time and resource-constrained features described here.

Figure 8.1 illustrates the model of software abstraction layers for MPI/RT.

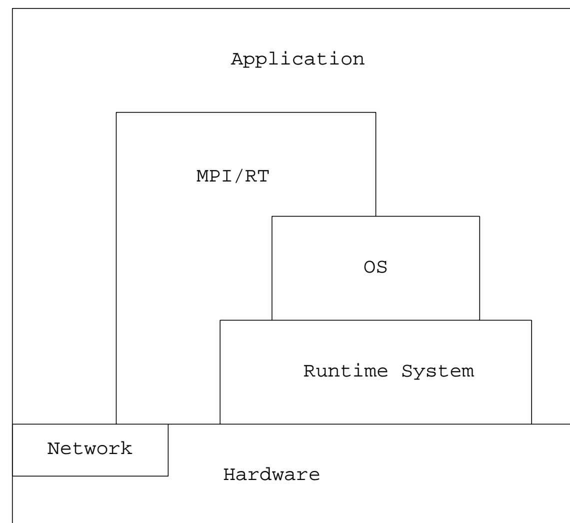

Figure 8.1: Model of Software Abstraction Layers for MPI/RT.

MPI/RT will serve as an open standard for a wide variety of high-performance, real-time, embedded, and heterogeneous parallel computing systems encompassing a diverse mix of computational paradigms. We intend that MPI/RT will make message passing programming relevant to the real-time community as well as enable development of vendor-independent real-time applications. It should also assist in bridging the gap between theory and implementation in parallel real-time computing and communications.

8.2 Real-time Message-Passing Requirements

Although developers have an intuitive sense of what they mean by a real-time system, definitions vary widely. The distinction between real-time computer systems and general-purpose computer systems lies not in their performance specifications, but in the relative importance of system timing considerations. In real-time computing, the correctness of a computation depends not only on the results of a computation, but also on the time at which the results of the computation are generated.

The measures of merit in a real-time system include:

- Timing correctness as well as program correctness. A real-time program can proceed, in a meaningful way, only if all previous steps are time-correct.

- Deterministic operation, even at the cost of performance,
- Predictable response to urgent events,
- High degree of schedulability,
- Stability under transient load: When the system is overloaded by events and meeting all deadlines is impossible, the deadlines of selected critical tasks must still be guaranteed.
- Dependability,
- Portability, with minimal impact on performance.

8.3 RT Profiles

There are several profiles of the real-time MPI. These profiles are independent of the programming paradigms that this chapter provides. All implementations are required to provide all profiles, though implementors may concentrate their efforts on the profile or profiles most relevant to their systems. This will provide for wide portability, and allow for appropriate investment in particular system niches.

Advice to implementors. Such profiles may be handled at compile/link time. We have an expectation that "smart linking" will be used to minimize the overhead associated with unused functions. This is compatible with the MPI-1 expectation of profiling libraries, that essentially puts one function per source file in most implementations. (*End of advice to implementors.*)

The hierarchy of profiles to be recognized by MPI/RT are as follows:

- Resource constrained MPI-1.2, defined below,
- Resource constrained MPI-1.2, plus RT features described in this chapter,
- MPI-1.2 (classic MPI), plus RT features,
- MPI-1.2 (classic MPI), resource constrained MPI-2 (to be defined), plus RT features.
- MPI-1.2 (classic MPI), MPI-2, plus RT features.

8.3.1 Resource Constrained MPI-1.2

This profile is to include all of the functionality of the revised MPI-1 reference document, with the following exceptions and restrictions:

- Derived datatypes are omitted, but built-in datatypes are preserved.
- Virtual topologies are eliminated.
- All buffered send modes are eliminated (*e.g.,* MPI_Bsend).
- Explicit operations on groups are eliminated.

- Users are recommended to assume zero buffering in writing their programs.

  Advice to users. MPI is not mandated to assume buffering, but most non-real-time implementations provide it. This is a strong advice to users and implementors of the resource-constrained versions to expect programs that use any buffering to deadlock or otherwise fail. (*End of advice to users.*)

Advice to implementors. The expectation is that all resource-constrained implementations will utilize smart linking to avoid code bloat, and will take advantage of simplifications that arise from the restriction to built-in datatypes. (*End of advice to implementors.*)

8.3.2 Other profiles

The profiles, defined above, are essentially self-explanatory. The details of MPI-2 are to be revisited upon its completion. Furthermore, there is an expectation that the meaning of "RT features" will be broadened to include all relevant RT versions of the MPI-2 functionality rather than MPI-1 functionality only. For instance, I/O will require a real-time analog.

8.4 RT Initialization/Termination

Discussion:

- In a mixed environment, with one "world," some processes may need RT features while others may not.
- It is desired to have MPI_INIT start all the MPI processes, and have some exploit real-time behavior according to the way they were linked and/or executed. ⊤ (Fin1) ⊥ (Fin1)
- Further arguments are needed about how to achieve the functionality of starting the real-time mode of operation, while allowing other processes to avoid expensive synchronization. It was pointed out that admission tests, with some processes uninvolved in real-time, might be specious at best.

8.5 Clocks

Discussion: It was stated that it will be difficult for implementations to provide appropriate bounds with respect to Accuracy and Access Time of clocks. Various caveats will be added to allow implementors to provide finite bounds, of some residual value to users.

8.5.1 Synchronization of Clocks

Most platforms that support real-time applications provide tightly synchronized system clocks, that are dependent upon special hardware support. There are several compelling reasons for having highly synchronized clocks [9]:

1. Fine-grained, accurate instrumentation is needed for all approaches to real-time message-passing systems, and even for performance measurements in non real-time systems.

2. Real-time applications require precise timing correctness. These systems also require demonstrations of this correctness, and often require delicate tuning for optimal performance. Well-synchronized clocks are necessary to support these requirements in a parallel environment.

3. Applications should be able to adjust scheduled times for portability, based upon the quality of the synchronized clocks. For example, time padding will be needed to adapt scheduled message-passing operations on heterogeneous processing nodes.

4. The primary goal of time-driven real-time MPI is to support application specification of resource usage. For time-driven MPI/RT, all resources that are used for communication need to be scheduled in order to achieve predictable behavior. These scheduled resources can include:

   (a) Distributed Memory,

   (b) Shared Memory,

   (c) Communication Bandwidth,

   (d) Communication Fabric

   (e) Computation.

   as well as others, such as platform specific resources related to inter-process communication.

8.5.2 Description of the Clocks

The processing nodes (which may contain one or more CPUs) that comprise a real-time parallel processing system may have access to several clocks. We designate one of the clocks as a globally synchronized clock. For each process in an MPI/RT program, this clock will the be one read by the MPI_WTIME call. There is an underlying assumption that each process is associated with a fixed processing node. The process accesses its globally synchronized clock through its associated node.

For most platforms, each node has a local clock, and this is periodically corrected in order for it to serve as a synchronized clock for all MPI/RT processes active at that node. However, other alternatives are not precluded. For example, there may even be a single clock at one of the nodes or even completely outside of the participating nodes, which all the processing nodes access over a network (which may also be used for regular data transfer operations) to get a synchronized clock value.

A fundamental assumption is made that the system clocks will be monotonically non-decreasing. We also assume that the underlying operating system will hide any artifacts resulting from the overflow of system clock counters.

8.5.3 Clock Synchronization Parameters

In this section, we present several parameters that describe the synchronized clocks, all of which must be accessible at both run time and compile time. The values of all these parameters are expected to be double-precision floating-point values measured in seconds, with exception of drift, which is dimensionless.

Resolution (Tick) Resolution represents the time between two successive clock ticks. The resolution of the various synchronized clocks in a heterogeneous system may differ.

Advice to implementors. We have proposed a subtle modification to MPI_WTICK, so that the call will return the resolution of the synchronized clock of the processing node associated with the calling process. In order to support tightly synchronized system clocks, we expect clock resolution to be at most one millisecond. (*End of advice to implementors.*)

Drift Drift indicates, for each synchronized clock, a guaranteed upper bound on the error in the rate of the clock. That is, if the drift is δ, then the clock rate (measured in seconds per actually elapsed second) is guaranteed to be between $1 - \delta$ and $1 + \delta$. Drift is, as stated above, dimensionless.

Drift can be effectively used to bound the accuracy of measurement of small time intervals, when they are measured by the difference between two readings of the same synchronized clock.

Advice to implementors. An implementation of MPI/RT should reset the drift ⊤ (Fin1) parameter when global system clocks are synchronized. (*End of advice to implementors.*) ⊥ (Fin1)

Skew Skew is a maximum bound on the absolute value of the difference between simultaneous values of the synchronized clocks in distinct nodes. Note that this refers to ideal values, not the result of any real reading operations.

Accuracy Accuracy is a maximum bound on the absolute value of the difference between simultaneous values of the synchronized clock and an ideal clock started at the synchronized clock hypothetical starting time (some critical instant in time).

If synchronized clocks are periodically corrected in order to deal with drifts and other inaccuracies, the calculation of the interval accuracy based upon drift will be too pessimistic for large intervals.

This value can also be used for cross-platform synchronization.

Access Time Access Time is a maximum bound on the time to execute a call of MPI_WTIME. This, of course, assumes that the execution of the call is not interrupted by the operating system. This undesirable and ambiguous caveat is necessary with current operating systems (especially those with virtual memory) because an absolute guarantee would be so long as to be practically unusable.

Implementations that are based on accessing the same global clock instead of a local synchronized one will have the same resolution for all nodes, with zero skew, no drift, and identical accuracy (subject to the propagation delay of the clock signal) for all processing nodes. However, the access times could still vary at each processing node and across all the processing nodes. A bound on this variability is needed. ⊤ (Fin1)

8.5.4 MPI/RT Clock Attributes

⊥ (Fin1)

In order for MPI/RT to allow portable applications across a wide variety of parallel real- ⊤ (Fin1)

⊥ (Fin1)

time systems, it is necessary to encapsulate the clock synchronization parameters of section 8.5.3 in a system-independent way. MPI provides a caching facility that allows an application to attach arbitrary pieces of information, called **attributes**, to both intra- and intercommunicators [13]. This information is retrieved by referencing a *key*. A set of attributes that describe the execution environment is attached to the communicator MPI_COMM_WORLD when MPI is initialized. The value of these attributes can be inquired by using the function MPI_ATTR_GET. At initialization time, MPI/RT adds the following keys to
MPI_COMM_WORLD:

- MPIRT_WTIME_DRIFT (DOUBLE)
- MPIRT_WTIME_SKEW (DOUBLE)
- MPIRT_WTIME_ACCURACY (DOUBLE)
- MPIRT_WTIME_ACCESS_TIME (DOUBLE)

Discussion: It was proposed that we also add MPI_WTICK to that list of attributes. Currently, MPI_WTICK is a function call in MPI-1. It was also considered that users might have the ability to change tick in some systems. This implies that this attribute is not static and may change or be changed at run time.

MPI-2 considered changes to the implementation requirements for MPI_WTIME and MPI_WTICK. We will have to consider this change in relationship to MPI/RT.

The format of above parameters match POSIX Specification.

The values of the listed parameters do not depend on what applications are running but rather on the parallel environment. These parameters represent constraints on the changes to the environment. For example, the skew should not be increased when new processes are added while an application is running.

Discussion: That suggests that dynamic process management functionality of MPI-2 are impacted by timing correctness requirements.

8.6 Real-Time Buffer Pools

This section proposes an interface for buffer management, where buffers are organized into buffer pools, that are associated with real-time channels.

Discussion: A proposal for a method to hook application specific modules to do queue management both for implementation and user sides is needed.

8.6.1 Buffer Pool Object Creation and Destruction

Buffer pools are created via single constructor that specifies their entire contents. The format of the participating buffers can not be modified after creation, however a new buffer pool that can reuse the same memory can be associated with the same channel (see section 8.7). The buffer pool handle is used in the creation of a real-time channel, following a

specific constructor (MPIRT_CHANNELS_INIT). The buffer state transition diagram, illustrated in Figure 8.2, shows the relationship of the buffers to data transfer operations and channels.

All the buffers in a buffer pool are the same length and contain the same datatype elements. The datatypes are relative, while addresses for individual buffers are absolute. The buffer pool can be shared between several channels but all channels must use the same queuing strategy.

MPIRT_BUFFER_POOL_CREATE(count, datatype, system_queue_strategy, bufcount, bases, bufpool)

IN	count	the number of elements of datatype in a buffer (nonnegative integer)
IN	datatype	datatype of each element (handle)
IN	system_queue_strategy	MPI's designated approach to managing the buffer pool when it is used in an association with a channel (integer)
IN	bufcount	the number of buffers in the buffer pool (nonnegative integer)
INOUT	bases	array of length bufcount with the beginning addresses of each buffer in the pool (choice)
OUT	bufpool	buffer pool object (handle)

```
int MPIRT_Buffer_pool_create(int count, MPI_Datatype datatype, int
              system_queue_strategy, int bufcount, void *bases[],
              MPIRT_Bufpool *bufpool)
```

```
MPIRT_BUFFER_POOL_CREATE(COUNT, DATATYPE, SYSTEM_QUEUE_STRATEGY, BUFCOUNT,
              BASES, BUFPOOL, IERROR)
    INTEGER COUNT, DATATYPE, SYSTEM_QUEUE_STRATEGY, BUFCOUNT, BASES(*),
    BUFPOOL, IERROR
```

The system_queue_strategy argument indicates the desired buffer management strategy. Two specific values are currently under consideration: MPIRT_BUFFER_CIRCULAR_WAIT and MPIRT_BUFFER_CIRCULAR_NOWAIT These two values specify that the system should traverse the buffers in a circular order when trying to locate an available buffer.

A unique index is defined with each buffer. When the MPI creates all buffers it creates an index between 0 and bufcount-1 for each buffer. These numbers are assigned in accordance with the order in the array bases above.

Discussion: The zero length buffers are allowed. The users can use them to do their own data flow control and the number of these buffers make the difference for the application users. The meaning of other parameters including datatype, queue strategy, and bases have to be considered for this case. Can we specify the same base for all buffers? Does queue strategy have any effect? However, the wait no wait (overwrite or not) option still makes sense. Should the value of datatype be ignored for zero length buffer?

The following function destroys a buffer pool handle:

MPIRT_BUFFER_POOL_HANDLE_FREE(bufpool)

INOUT	bufpool	buffer pool object (handle)

```
int MPIRT_Buffer_pool_handle_free(MPIRT_Bufpool *bufpool)
```

```
MPIRT_BUFFER_POOL_HANDLE_FREE(BUFPOOL, IERROR)
    INTEGER BUFPOOL, IERROR
```

8.6.2 User Buffer Access

In considering the consumption and release of buffers as a channel's state evolves under message transmission, both the system's strategy, and the user's emphasis on message age come into play. The desired system strategy for handling the buffer pools upon creation has been specified by MPIRT_BUFFER_POOL_CREATE. There is also the need for an application to specify "newer," "older" or "intermediate" age data, as a function of the semantics of the meaning of such data to the application.

Buffers in the buffer pool are either available or used. Buffers become used as the user consumes them. Used buffers can be made available to the system again by the following function:

MPIRT_BUFFER_MAKE_AVAIL(index, bufpool)

IN	index	buffer index (integer)
INOUT	bufpool	buffer pool object (handle)

```
int MPIRT_Buffer_make_avail(int index, MPIRT_Bufpool *bufpool)
```

```
MPIRT_BUFFER_MAKE_AVAIL(INDEX, BUFFER, IERROR)
    INTEGER INDEX, BUFPOOL, IERROR
```

The special index value MPIRT_ALL_BUFFER indicates that all buffers in the buffer pool are available for reuse by MPI/RT.

The following operation allows the user to get access to the buffers for a message receive or a future send operation:

MPIRT_BUFFER_GET(bufpool, user_strategy, count, index, equest)

IN	bufpool	buffer pool object (handle)
IN	user_strategy	user strategy for buffer management (integer)
INOUT	count	count of the buffer element (integer)
OUT	index	the index of the buffer (integer)
OUT	request	a copy of the request of the channel on which the message arrived (handle)

```
int MPIRT_Buffer_get(MPIRT_Bufpool bufpool, int user_strategy, int *count,
              int *index, MPI_Request *request)

MPIRT_BUFFER_GET(BUFPOOL, USER_STRATEGY, COUNT, INDEX, REQUEST, IERROR)
    INTEGER BUFPOOL, USER_STRATEGY, COUNT, INDEX, REQUEST, IERROR)
```

Three specific values of the **user_strategy** are currently defined: MPIRT_BUFFER_NEWEST, MPIRT_BUFFER_OLDEST, and MPIRT_BUFFER_NEXTAVAIL. Other options may be added. For example, MPIRT_BUFFER_GET(**bufpool**, NEWEST, 1, **index**, **request**) returns the index of the buffer with the last received message. To obtain a buffer suitable for use on the sending side MPIRT_BUFFER_NEXTAVAIL is defined.

Discussion: A frozen user view of the buffer queue may be required for some applications (atomic access to the buffer pool queue). For example, the user may need to access three consecutive receive messages. Simple repetition of MPIRT_BUFFER_GET(bufpool, NEWEST, 1, index, request) is not sufficient, since new messages may arrive between these three operations and the queue will be modified. Hence, the following operations are proposed: MPIRT_QUEUE_LOCK, MPIRT_QUEUE_UNLOCK. An implementation can have a shadow queue so it can continue to receive and send messages, while maintaining the frozen buffer queue snapshot for the user. Since additional resources may be required, a flag can be added to the API for the buffer pool at creation time to notify implementation. This flag provides information for establishing QOS for a channel.

Discussion: An example is needed here, that will be added later.

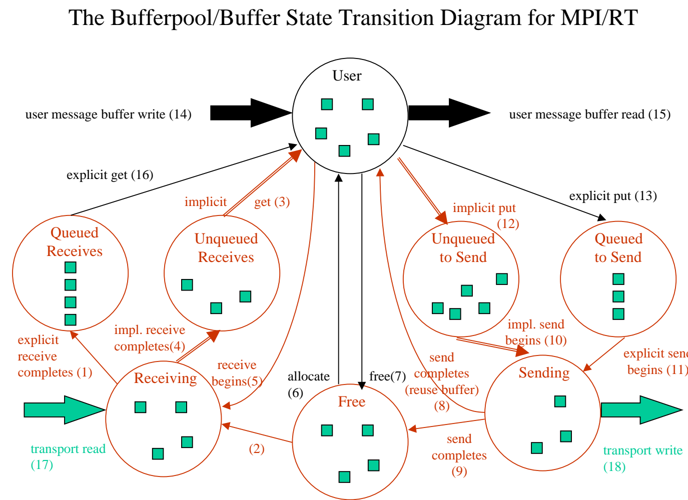

Figure 8.2: Buffer Pool State Transition Diagram. This diagram represents the semantic behavior of the buffer pool and the operations, but does not specify the implementation.

8.7 Channels

Persistent point-to-point communications can provide scheduled sends and receives. MPI-1 persistent communications allow any receive to match a persistent send, and any send to match a persistent receive [13]. These persistent communication calls operate as pre-negotiated communication endpoints. MPI-2 provides a simple way to bind such endpoints into a point-to-point channel. MPI/RT chooses a more general strategy, defining point-to-point collective channel.

In MPI/RT, persistent channels offer the functionality of a virtual channel [4, 6, 10] within the framework of the MPI standard. The motivations for having virtual channels in MPI/RT are as follows:

- Ability to exploit persistent communication
- Deadlock avoidance
- Once established, virtual channels should guarantee properties critical for timing correctness such as:
  - Bounds on end-to-end delay,
  - Jitter control,
  - Minimum bandwidth,
  - Buffer space.

8.7.1 Point-to-point Channels

Discussion: This draft contains two versions of the channel management: separate operations for creating, deleting and modifying channels, and a single combined operation one.

The combined unified initialize, modify, and delete channel operation is introduced and is written.

How to pass information about the endpoint of a channel is also under discussion. The suggestions range from several arrays, each defining one parameter of an endpoint, to two separate arrays one for HEADS and one for TAILS, to an array structure where each element of the structure defines single endpoint.

As part of this proposal, it was recommend that MPI/RT will provide no other buffering for a channel other than that explicitly provided by the user via the buffer pool. (For example, if all of the receive buffers have been exhausted, any further requests at the send side will not complete until one of the receive buffers is freed.) By eliminating extra buffering in an implementation, tighter real-time constraints can be met; especially the needs of resource-constrained systems.

Collective operations for point-to-point channels negotiation are as follows:

MPIRT_CHANNELS_INIT(bufpools, nchannels, flags, ranks, qoss, fns, names, comm, requests, errors)

IN	bufpools	array of transfer buffer pool addresses (handle array of length nchannels)
IN	nchannels	length of all arrays used locally (number of channels being opened) (integer)
IN	flags	array indicating endedness of ith channel (integer array of length nchannels)
IN	ranks	the ranks of the remote channel endpoints (integer array of length nchannels)
INOUT	qoss	quality of service parameters per channel (handle array of length nchannels)
IN	fns	array of function names for handlers on channels (array of user-defined function of type MPIRT_QOS_ERROR_FN
INOUT	names	array of names of the channels that are used in event lists (string array of length nchannels)
IN	comm	communicator (handle)
OUT	requests	array of request objects, one per channel (persistent requests array of length nchannels)
OUT	errors	specifies success, non-success, for each channel (integer array of length nchannels)

```
int MPIRT_Channels_init(MPIRT_Bufpool bufpools[], int nchannels,
              int flags[], int ranks[], MPIRT_QOS *qoss[], MPIRT_QOS_ERROR_FN
              fns[], char **names[], MPI_Comm comm, MPI_Request *requests[],
              int *errors[])
```

```
MPIRT_CHANNELS_INIT(BUFPOOLS, NCHANNELS, FLAGS, RANKS, QOSS, FNS, NAMES,
              COMM, REQUESTS, ERRORS, IERROR)
    INTEGER BUFPOOLS(*), NCHANNELS, FLAGS(*), RANKS(*), QOSS(*), FNS(*),
        CHARACTER*(*) NAMES(*),
            INTEGER COMM, REQUESTS(*), ERRORS(*), IERROR
```

From the perspective of a single process calling the function above, for each of nchannel entries, a single channel is specified, which must have a corresponding entry in exactly one other process calling the same constructor, with appropriate arguments. The ith channel's connectivity is specified completely by:

- ranks[i] – the name of the other end of the channel (integer rank in communicator group). The ranks are the names of those processes contained in the group of comm.
- flags[i] – the direction of the transfers on the channel, where each process specifies itself as sender with a constant HEAD (1), or else receiver with a constant TAIL (-1),

This operation yields as many channels as specified by the user, with the potential for duplicate ranks both for sending and receiving as well as multiple channels involving the same pair. The self-channels are allowed and the matching rule applies to them too.

These individual requests may be used to transmit point-to-point data between processes, independently. The initialization call is collective in order to establish required quality of service in the face of shared resources. This call does not initiate any communication, but only sets up such communication.

The quality of service parameters (per channel) include specification of hard vs. best-effort requirement for meeting the QOS request made. By hard, we mean that if the QOS cannot be offered exactly as specified, then an error is returned, and the channel is not established. Best effort constitutes the nearest level of service a system can provide [8, 9]. It is less than the application requested as specified in the QOS In-parameter, is returned in the QOS as an Out-parameter.

Both ends of each channel must specify the same quality of service requirements, or the call is erroneous. The error function handler array specifies functions that are called in case of QOS failures, but not for such a specification error. These error functions are needed to support one-sided and no-sided (where MPI/RT does data transfer without application having any per-transfer data transfer operations) communication uses of the channels, as there is no MPI/RT call to bind a function to an operation on a per-use (such as timeout) basis. This is especially critical for the periodic uses. We may create more than one error handle for a channel. One is needed to handle QOS failure situations, especially for no-sided communication. More may be needed to handle other types, like buffer pool overflow, communicator errors and others, that some applications may want to handle prior to a timeout expiration. The error condition array specifies MPI_SUCCESS for successful channel creation, and offers specific error codes, to be determined, for non-successful channel creation operations.

The names is used for application event-driven paradigm (see section 8.10). The user can provide the channel names and MPI/RT will assign them to the channels, or the user can specify MPIRT_IGNORABLE and MPI/RT will provide the channel names and return them as an Out-parameter in names. The names on both endpoints of the channel must match.

Discussion: *Proposal:* The complicated mappings between error codes and error handlers should be addressed in the future. Masking errors, providing mechanisms to define and find out about "fatal" errors should be address at the same time. This is still based on assumption that most of errors for this section, including, communicators, buffer pools, QOS and so on, will be associated with the channels.

Discussion: *Proposal:* Currently, the QOS error handler is specified by the name. For the future we should consider wrapping this error handler into MPI_REQUEST using persistent handlers, analogous to the lower-level or application event-driven paradigms. That method is more generic and does not require introduction of the new object MPIRT_QOS_ERROR_FN.

Advice to users. The same buffer pool may be used for more than one channel. While for some paradigms (event-driven, time-driven) the user can easily manage to avoid problems with the shared buffers, in general it is a dangerous scenario that can lead to various errors and create problems for maintaining QOS for the channels that

share the buffer pool. Hence shared buffer pool can be a source of a performance optimization, but it should be used with caution. (*End of advice to users.*)

MPIRT_CHANNELS_DELETE(comm, flag, nchannels, requests)

IN	comm	communicator (handle)
IN	flag	"abrupt" versus "close" type channel semantics (integer)
IN	nchannels	number of channels to be deleted (integer)
INOUT	requests	array of request objects, one per channel (persistent request array of length nchannels)

```
int MPIRT_Channels_delete(MPI_Comm comm, int flag, int nchannels,
             MPI_Request *requests[])
```

```
MPIRT_CHANNELS_DELETE(COMM, FLAG, NCHANNELS, REQUESTS, IERROR)
    INTEGER COMM,
        LOGICAL FLAG,
            INTEGER NCHANNELS, REQUESTS(*), IERROR
```

This operation frees the specified requests previously allocated, and is a collective operation over the group of the communicator. This may be all the channels allocated in a single or multiple calls, or just a subset thereof, according to user preference.

Any pending operations are cancelled, if the flag is set to MPIRT_DELETE, and are completed, if the flag is set to MPIRT_CLOSE.

In order to allow changes of channels parameters for application mode changes, a modification capability is offered:

MPIRT_CHANNELS_MODIFY(bufpools, nchannels, flags, qoss, fns, comm, requests, errors)

IN	bufpools	array of transfer buffer pool handles (handle array of length nchannels)
IN	nchannels	number of channels to be modified (integer)
IN	flags	if MPIRT_FLAGS_IGNORE, channel is left unchanged, otherwise it is MPIRT_MODIFY and specific parameters are renegotiated for this channel (integer)
INOUT	qoss	array of quality of service parameters per channel (handle array of length nchannels)
IN	fns	QOS error handler functions (array of user-defined functions of type MPIRT_QOS_ERROR_FN of length nchannels)
IN	comm	communicator (handle)
INOUT	requests	request objects, one per channel (persistent request array of length nchannels)
OUT	errors	specifies success, non-success, for each channel (integer array of length nchannels)

```
int MPIRT_Channels_modify(MPIRT_Bufpool bufpools[], int nchannels,
            int flags[], MPIRT_QOS *qoss[], MPIRT_QOS_ERROR_FN fns[],
            MPI_Comm comm, MPI_Request *requests[], int *errors[])
```

```
MPIRT_CHANNELS_MODIFY(BUFPOOLS, NCHANNELS, FLAGS, QOSS, FNS, COMM,
            REQUESTS, ERRORS, IERROR)
    INTEGER BUFPOOLS(*), NCHANNELS,
        LOGICAL FLAGS(*),
            INTEGER QOSS(*), FNS(*), COMM, REQUESTS(*), ERRORS(*), IERROR
```

When a flag is set to MPIRT_MODIFY, then the remaining arguments become significant. On both ends of a channel, the flags must be set consistently or the program is erroneous. For each channel to be renegotiated, each parameter is checked. A special value of "ignore" appropriate to the argument type of a given field will be defined for each of these, to keep the currently set value. If not "ignored," then the quantity will be modified:

- bufpool: MPIRT_BUFPOOL_IGNORE,
- qos parameter: MPIRT_QOS_IGNORE,
- error fn parameter: MPIRT_HANDLER_IGNORE,

It should be noted that the in-out parameter requests[] contain sufficient information so that it is not necessary to reiterate the ranks[] of the communicator involved, which are in any event not modifiable under this call. That is, channels may not be redirected.

Discussion: What parameters can be modified and what the meaning of modification for these parameters is still under discussion. The only one parameter that everybody agree on is QOS.

If an error occurs upon a channel renegotiation, the original channel set up continues to persist. Setting channels to have negligible usefulness of quality of service does not in any event cause a channel to disappear, and some system resources mat continue to be reserved for it.

Discussion: *Proposal:* There is no easy way to change any parameters of a buffer pool. We can only disconnect the old buffer pool with a channel and connect a new one. Maybe we should add another modify operation, to modify buffers in the existing buffer pool, while they are attached to a channel.

The most general real-time programs will want to transition channel resources in a combined create/modify/delete call. Others may choose to use these calls individually. In general, MPI/RT will be able to better utilize resources when presented with an atomic transition. The operation below is a combined operation that allows create, delete and modify channels in atomic fashion.

MPIRT_CHANNELS_TRANSIT(comm, ncchannels, cbufpools, cranks, cflags, cqoss, cfns, cnames, cchannels, nmchannels, mbufpools, mqoss, mfns, mchannels, ndchannels, dchannels)

IN	comm	communicator (handle)
IN	ncchannels	length of arrays for channels to be created, (integer)
IN	cbufpools	array of buffer pool addresses (handle array of length ncchannels)
IN	cranks	array of ranks of the remote channel endpoints (integer array of length ncchannels)
IN	cflags	array indicating the channel directions (integer array of length ncchannels)
IN	cqoss	array of quality of service parameters (handle array of length ncchannels)
IN	cfns	array of function names for qos error handlers on channels (array of user-defined function of type MPIRT_QOS_ERROR_FN of length ncchannels)
INOUT	cnames	array of names of the channels that are used in event lists (string array of length ncchannels)
OUT	cchannels	array of request objects for channels to be created (array of persistent requests of length ncchannels)
IN	nmchannels	length of arrays for channels to be modified (integer)
IN	mbufpools	array of buffer pool addresses (handle array of length nmchannels)
IN	mqoss	array of new quality of service parameters (handle array of length nmchannels)
IN	mfns	array of new function names for qos error handlers (array of user-defined function of type MPIRT_QOS_ERROR_FN of length nmchannels)
INOUT	mchannels	array of request objects for channels to be modified (persistent requests array of length nmchannels)
IN	ndchannels	length of arrays for channels to be deleted (integer)
INOUT	dchannels	array of request objects for channels to be deleted (persistent requests array of length ndchannels)

```
int MPIRT_Channels_transit(MPI_Comm comm, int ncchannels,
             MPIRT_Bufpool cbufpools[], int cranks[], int cflags[],
             MPIRT_QOS *cqoss[], MPIRT_QOS_ERROR_FN cfns[], char **cnames[],
             MPI_Request *cchannels[], int nmchannels,
             MPIRT_Bufpool mbufpools[], MPIRT_QOS mqoss[],
             MPIRT_QOS_ERROR_FN mfns[], MPI_Request *mchannels[],
             int ndchannels, MPI_Request *dchannels[] )
```

```
MPIRT_CHANNELS_TRANSIT(COMM, NCCHANNELS, CBUFPOOLS, CRANKS, CFLAGS, CQOSS,
             CFNS, CNAMES, CCHANNELS, NMCHANNELS, MBUFPOOLS, MQOSS, MFNS,
             MCHANNELS, NDCHANNELS, DCHANNELS, IERROR)
```

```
    INTEGER COMM, NCCHANNELS, CBUFPOOLS(*), CRANKS(*), CFLAGS(*), CQOSS(*),
    CFNS(*),
        CHARACTER*(*) NAMES(*),
            INTEGER CCHANNELS(*), NMCHANNELS, MBUFPOOLS(*), MQOSS(*),
            MFNS(*), MCHANNELS(*), NDCHANNELS, DCHANNELS(*), IERROR
```

⊥ (Fin1)

All the parameters of the above operations are identical to the parameters for the separate operations. The only exceptions are the errors. Since this operation is atomic and no channel handles can be created, modified, or deleted unless all the requests can be satisfied, there is no reason to return individual channel errors. These errors do not provide any information to a user beyond the fact that there are not enough resources to satisfy all the requests. A user need more information beyond what channels an implementation can create/delete/modify in order for him/her to be able to split this operation into a series of requests for a smaller number of channels if it possible at all.

This operation is atomic. Release of resources should not occur if all channels cannot be allocated. Modifying any channel requests is not permitted if any of the original channels passed in can not be modified. The call is collective in order to establish required quality of service in the face of shared resources. This call does not initiate any communication, but only sets up such communication.

8.7.2 Collective Operations with Quality of Service

For each operation in MPI-1 and MPI-2 and collective operations (i.e., non-blocking) that should be in the journal of development (future MPI-3), we have defined a persistent variant, that is initiated with asynchronous (MPI_START) and synchronous (MPI_DO) mechanisms. For each such operation, the real-time variant is specified as follows: A quality of service specification is added as an additional parameter, directly before the communicator parameter. Currently, we have both the MPI-1 and MPI-2 style functionality (illustrated for broadcast):

```
MPI_Bcast(bufpool, root, comm);
MPI_Bcast_init(bufpool, root, comm, &request);
```

For each collective operation MPI/RT provides an additional form. ⊤ (Fin1)

⊥ (Fin1)

MPIRT_BCAST_INIT(bufpool, root, qos, fn, comm, request)

IN	bufpool	buffer pool (handle)
IN	root	rank of broadcast root in a group (integer)
INOUT	qos	quality of service parameters per channel (handle)
IN	fn	function name for QOS error handler for the channel (user-defined function of type MPIRT_QOS_ERROR_FN) ⊤ (Fin1)
IN	comm	communicator (handle) ⊥ (Fin1)
OUT	request	request object (persistent request)

⊤ (Fin1)

```
int MPIRT_Bcast_init(MPIRT_Bufpool bufpool, int root, MPIRT_QOS *qos,
             MPIRT_QOS_ERROR_FN fn, MPI_Comm comm, MPI_Request *request)

MPIRT_BCAST_INIT(BUFPOOL, ROOT, QOS, FN, COMM, REQUEST, IERROR)
    INTEGER BUFPOOL, ROOT, QOS, FN, COMM, REQUEST, IERROR
```

⊥ (Fin1)

The quality of service parameter applies to the collective operation. The hard vs. best effort nature of the quality of service specification is as for point-to-point channels. The specification of quality of service may differ from that used for point-to-point channels, and may also differ for various collective operations. The collective operations can have a timeout version. For modification (late binding), the following type of service is to be provided for each collective operation, illustrated here for broadcast:

MPIRT_BCAST_MODIFY(bufpool, root, qos, fn, comm, request)

IN	bufpool	new buffer pool address or "ignore" (handle)
IN	root	new rank of broadcast root in a group or "ignore" (integer)
INOUT	qos	quality of service parameters per channel or "ignore" (handle)
IN	fn	function name for QOS error handler for the channel or "ignore" (user-defined function of type MPIRT_QOS_ERROR_FN)
IN	comm	communicator (handle)
INOUT	request	request object (persistent request)

⊤ (Fin1) ⊥ (Fin1)

⊤ (Fin1)

```
int MPIRT_Bcast_modify(MPIRT_Bufpool bufpool, int root, MPIRT_QOS *qos,
             MPIRT_QOS_ERROR_FN fn, MPI_Comm comm, MPI_Request *request)

MPIRT_BCAST_MODIFY(BUFPOOL, ROOT, QOS, FN, COMM, REQUEST, IERROR)
    INTEGER BUFPOOL, ROOT, QOS, FN, COMM, REQUEST, IERROR
```

⊥ (Fin1)

Each parameter will either be "ignorable" or specify a new value. The modify is itself a collective operation over the group of the communicator. Quality of services may be either increased or decreased. For deletion of a collective persistent operation in the Collective Extensions chapter, it is assumed that MPI_REQUEST_FREE is sufficient. Here, we add a specific collective destructor:

Discussion: *Proposal:* This operation may be removed provided that the existing MPI-2 channel deletion is sufficient. This functionality is currently in the collective chapter in the JOD.

MPIRT_CHANNEL_COLL_DELETE(comm, flag, request)

IN	comm	communicator (handle)
IN	flag	"abrupt" versus "close" type channel semantics (integer)
INOUT	request	request object representing real-time collective operation (persistent request)

```
int MPIRT_Channel_coll_delete(MPI_Comm comm, int flag, MPI_Request *request)

MPIRT_CHANNEL_COLL_DELETE(COMM, FLAG, REQUEST, IERROR)
    INTEGER COMM,
        LOGICAL FLAG,
            INTEGER REQUEST, IERROR
```

This operation frees all request previously allocated, and is a collective operation over the group of the communicator. Any pending operations are cancelled, if the flag is set to MPIRT_DELETE, and are completed, if the flag is set to MPIRT_CLOSE.

Discussion: An equivalent, alternative model is to have two calls, and no flag.

Advice to implementors. If real-time collective operations are layered on top of point-to-point channels, then a set of channels used to create a collective operation could be built up with the point-to-point channels, and deleted using the point-to-point channel deletion defined above.

However, the quality of service parameter to the collective operation, as specified by the user, will have to be translated to appropriate channel quality of service parameters by the implementation. The implementation is also free to select one of several algorithms (poly-algorithm [12]) to accomplish a given operation. We will consider in the future a single operation to allow creation, deletion, and modification of all channels (point-to-point and collective) over a single communicator. (*End of advice to implementors.*)

8.8 Quality of Service

The quality of service parameters span point-to-point persistent channels, collective persistent channels and all paradigms of real-time discussed in this chapter. The QOS parameters are the most important distinction between regular MPI and real-time MPI.

Only paradigm-specific QOS parameters are part of MPI/RT for point-to-point channels and collective channels, and that any lower-level QOS parameters are only discussed as advice to implementors. See also [4, 10].

Discussion: Some of the lower-level quality of service may be revisited for multiple real-time paradigms mixed in the same application, but that is beyond the current scope of this draft. This will also recur when we consider implementation interoperability.

For the four paradigms, here are the current proposed contents of the quality of service parameters:

Time-Driven The QOS parameters for time-driven paradigm include:

- the time interval for message transmission,
- the period, for periodic message transmission, including the relationship between the time interval and the period, usually defined by a release time and a deadline,
- the starting time within the period (transmission need not start at the beginning of the period, or end within the period).

Event-Driven The currently expected QOS for the event-driven paradigm is the bound required on the activation time of an event handler for a delivered event.

Low-level event-driven paradigm will put bounds on local event trigger, whereas high-level event-driven will put bounds on global event delivery and trigger.

Priority-Driven The QOS parameters for the priority-driven paradigm include:

- Preemptability flag - boolean (yes is true or no is false),
- Integer Priority of the persistent channel for message transmission (integer),
- Priority class of the persistent channel (integer) (provides second level of priority)
- Some measure of the preemption quantum desired in bytes (allows a user to calculate timing bounds for guaranteed delivery)

Discussion: Parameters still in flax. Only the second bullet is agreed upon.

soft QOS **Discussion:** *Proposal:* For some applications the hard real-time guarantees are too restrictive, sometime at the expense of the performance. The working group voted to add a QOS version that does not require hard guarantees. The details of the "softer" quality are still under discussion.

Currently, there are no deadline associated with the message transmission for the priority-driven or event-driven paradigms.

Note that the aggregate size of the transmission is needed as part of quality of service, but this is provided separately through the (buffer, datatype, and count) parameters of the standard form of the channel initialization, and alternatively as part of the buffer pool objects.

Discussion: It was agreed that it is appropriate for an application to specify event-driven quality of service with either time-driven or priority-driven quality of service parameters but not both. It is quite often the case that an application uses event-driven with other paradigms.

It is still under discussion what the meaning of the mixture of time-driven and priority-driven paradigms with their associate quality of services might be. The mixture of soft and hard QOS paradigms will also be considered later.

⊤ (Fin1)

8.9 Time-driven MPI/RT

The primary goal of the time-driven approach to MPI/RT is to allow the real-time application sufficient control of the environment in which it is running so that it can explicitly schedule its message-passing activities and resource usage. Since MPI 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.

An application using time-driven MPI/RT will be able to specify time intervals to bound the resource usage of communication operations using globally synchronized clock values, and the implementation of time-driven MPI/RT will fulfill these requirements with minimal changes to the MPI standard.

For practicality, the work of our efforts has been to minimize the number of new functions introduced into MPI/RT profiles to provide these real-time features. MPI supports both blocking and non-blocking sends and receives, as well buffered, synchronous, and ready versions of each. We decided it was important not to add new versions of each send and receive mode, increasing the number of MPI calls multiplicatively. Rather, we sought solutions that would involve additions to the number of MPI functions.

8.9.1 Scheduling Message Transfers

The existing MPI message transfer operations lack two parameters that we consider critical for real-time applications, especially for the time-driven programming paradigm. These are a starting time of the operation and a timeout for completion of the operation. The starting time of an operation should be considered as a special case of an event. While certain applications (especially embedded ones) prefer an even finer granularity of control, we tried 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.

One distinctive characteristic of the time-driven approach to real-time message-passing is its lack of need for queues and system buffers. Applications use *ready mode* message-passing implicitly. A ready-mode send may be started *only* if the matching receive has already been posted [5]. 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 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 up to the application to specify times (for start and timeout) to ensure that the sender/receiver (local/remote) 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). A matter of significant discussion in the MPI/RT group concerns precisely what should happen to messages left on the network when a timeout expires.

8.9.2 Schedulable Time Intervals

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 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 often cannot 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.

8.9.3 MPI/RT Time Handle

The starting times and timeouts of the activity interval in time-driven MPI/RT data transfer operation calls are specified by a structure called a MPIRT_TIME_OBJECT (one instance of the structure is used for each). A MPIRT_TIME_OBJECT has two fields, MPIRT_TIME_OBJECT_TYPE and MPIRT_TIME_OBJECT_TIME. MPIRT_TIME_OBJECT_TYPE must have one of two values, ABSOLUTE, or RELATIVE. When a starting time is replaced by MPIRT_TIME_IGNORE then there is no hard constraint on when the operation should start. Implicitly, it should start as soon as possible, just as with the current MPI calls. Similarly, when a timeout is given by a MPIRT_TIME_IGNORE there is to be no hard constraint on when the operation should end. Furthermore, the second field of a MPIRT_TIME_OBJECT is not significant if the first field is set to MPIRT_TIME_IGNORE.

For a MPIRT_TIME_OBJECT whose first field is ABSOLUTE or RELATIVE, the second field (MPIRT_TIME_OBJECT_TIME) should be a field containing a double precision floating point number. In either case (ABSOLUTE or RELATIVE) the MPIRT_TIME_OBJECT_TIME refers to the global synchronized clock, but in the relative case, an actual constraint is to be derived at run-time by adding MPIRT_TIME_OBJECT_TIME to the value returned by as fresh as possible a read of the global synchronized clock. Note that even in the ABSOLUTE form, an actual time requirement cannot necessarily be constructed until the value of the

time object at the time of the execution of the call is known.

Thus, the activity interval for time-driven MPI/RT message transfers is specified by two parameters: starting time and timeout, each specified in turn by a time handle of the form MPIRT_TIME_OBJECT.

When using a late binding call, MPIRT_TIME_NOOVERRIDE should be used to retain built-in QOS specification, whereas MPIRT_TIME_IGNORE would replace the specified times with an "ignore," thereby making the call work as if no time interval were required.

In C/C++, the MPIRT_TIME_OBJECT is semi-opaque, like the MPI_STATUS. For Fortran, a strategy will be worked out similar to the MPI_STATUS solution as well.

8.9.4 Time-Driven Channel Calls

Using persistent channels to schedule a time-driven message transaction involves the addition of an MPIRT_TIME_OBJECT parameters to the MPI_START() call, which activates a persistent communications handle. The two MPIRT_TIME_OBJECT parameters defines the time interval for data transmission. The users can set these parameters to MPIRT_TIME_IGNORE in order to use the time interval specified in MPI_CHANNELS_INIT. Analogously, users can either specify the period or reuse the one specified by MPI_CHANNELS_INIT. If period is specified by MPIRT_START_TIME or by MPI_CHANNELS_INIT then this starts the persistent channel and MPI/RT implementation is responsible for moving the data between buffers at the end of the channel within the defined time interval of the period.

MPIRT_START_TIME(request, start, timeout, period, fn)

IN	request	persistent channel request object (persistent request)
IN	start	message start timing parameter (MPIRT_TIME_OBJECT)
IN	timeout	message stop timing parameter (MPIRT_TIME_OBJECT)
IN	period	optional periodic re-invocation (MPIRT_TIME_OBJECT)
IN	fn	void function to call on QOS failure (MPIRT_QOS_ERROR_FN)

```
int MPIRT_Start_time(MPI_Request request, MPIRT_TIME_OBJECT start,
              MPIRT_TIME_OBJECT timeout, MPIRT_TIME_OBJECT period,
              MPIRT_QOS_ERROR_FN fn)
```

```
MPIRT_START_TIME(REQUEST, START, TIMEOUT, PERIOD, FN, IERROR)
    INTEGER REQUEST, START, TIMEOUT, PERIOD, FN, IERROR
```

The meanings of start and timeout are defined above. The meaning of period is either: non-periodic, if MPIRT_TIME_IGNORE is specified, or else the length of the period between automatic restarts of the request specified. When period is specified, both start and timeout are taken as relative to multiples of the period. The buffer pool strategy specified for the buffer pool creation will tell which buffer to use for each message send.

For time-driven MPI/RT, it is sufficient to map a request into a function call specifi-

cation (per invocation), so that no status is formed, and no special action is taken, unless a timeout occurs. Other violations could occur, in the channel situation, and an analogous mechanism to the timeout event must be provided.

8.10 Event-Driven MPI/RT

The local and application event-driven real-time MPI paradigms provide functionality for local and global events, respectively. The local event-driven paradigm provides a mechanism for scheduling with QOS an application handler upon the completion of a data transfer operation. The application event-driven paradigm provides a mechanism for scheduling with QOS any application activity, including MPI data transfer, application functions triggered from a system event, application event or MPI event. Both paradigms allow users to manage MPI, system, and user resources using events.

Discussion: A discussion of the relationship between the communication and OS scheduler prompted a proposal to add a caveat with respect to the granularity of the OS scheduler for event driven QOS. Also a bound on the number of events over a time interval may be needed to guarantee QOS. This may be more critical for application event-driven MPI/RT.

8.10.1 Local Event-Driven Real-Time MPI

Request handlers are an ideal mechanism for implementing the event-driven paradigm. Handlers were introduced in MPI-2. The functionality of this paradigm can be used with either MPI or MPI/RT operation's requests. To help users better manage resources, two events for the data transfer completions are introduced. One event specifies the local completion of the data transfer, that is 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, that is the channel resources can be reused.

Request handlers

The local mechanism for event-driven MPI/RT is the request completion handler, shown below. (NOTE: This mechanism is based on the MPIRT_POST_HANDLER routine currently in the External Interfaces chapter.)

MPIRT_REQUEST_POST_HANDLER(request, request_cond, cond_handler_fn, failure_fn, extra_state, qos)

INOUT	request	MPI request (handle)
IN	request_cond	request condition (integer)
IN	cond_handler_fn	request condition handler (user-defined function MPIRT_function)
IN	failure_fn	QOS failure handler (user-define function MPIRT_QOS_ERROR_FN)
IN	extra_state	user supplied state (choice)
IN	qos	event-driven QOS (handle)

```
int MPIRT_Request_post_handle(MPI_request *request, int request_cond,
                MPIRT_function cond_handler_fn, MPIRT_QOS_ERROR_FN failure_fn,
                void extra_state, MPIRT_QOS qos)
```

```
MPIRT_REQUEST_POST_HANDLER(REQUEST, REQUEST_COND, COND_HANDLER_FN,
                FAILURE_FN, EXTRA_STATE, QOS, IERROR)
    INTEGER REQUEST, REQUEST_COND, COND_HANDLER_FN, FAILURE_FN, EXTRA_STATE,
    QOS, IERROR
```

The request condition and QOS failure handlers are both of the form:

MPIRT_HANDLER_FN(request, status, extra_state)

INOUT	request	user-supplied MPI request (handle)
INOUT	status	status of the request (handle)
INOUT	extra_state	user-supplied state (choice)

Once MPIRT_REQUEST_POST_HANDLER has been called, the handler function cond_handler_fn is to be called within the event-driven QOS after the given request reaches the condition specified by the request_cond argument. Recall, that the event-driven QOS specifies the bound required for the activation time of an event handler after the request_cond is reached. When the handler is called, it is passed the request, the status of the request, and the extra_state. If the condition handler cannot be called within QOS specified (the event-driven allotted QOS time in the case of either absolute or relative times, or the specified time has already passed in the case of absolute times only), then the failure handler failure_fn is called. Like the request handler, the failure routine is passed the request argument, that request's status, and the extra_state argument.

Discussion: The definition of MPIRT_REQUEST_POST_HANDLER is as yet imprecise. The exact meaning of *the handler function to be called* need to be defined. One definition may specify that the function should be in the ready queue of the operating system, while another may specify that the function should start its execution. These and other choices need to be discussed and voted.

There are two exceptional cases for the time argument: First, if qos is a relative time and its value is zero, then the handler is to be called "as soon as possible." (How soon is an implementation quality issue. The desired goal is an interrupt-like functionality.) If the time is MPIRT_TIME_IGNORE, the request handler will be called at some later time but not necessarily "immediately." Since an instantaneous response time is not practically achievable in the first case and since the response time is unspecified in the second case, the failure handler will never be called for either case.

If the request has reached the specified condition when the MPIRT_REQUEST_POST_HANDLER call is made, the handler is scheduled for execution (unless the qos specified time is absolute and has already passed, in which case the failure routine is called). Notice that for normal nonblocking calls, it may often be the case that the request has already completed. In such circumstances the user may wish to use a persistent version of the call generating the request, if it is available. This would allow the handler to be specified before the request is started. (The lack of late binding poses something of a problem, however. See the example given below.)

Note that a request can have only one handler for each of its conditions. If the user wishes to have a callback list for each condition, this must be implemented manually by having a high-level handler that calls the individual handlers one-by-one. If MPIRT_REQUEST_POST_HANDLER is called for a request and a condition for which a handler has already been specified and the handler has not yet been invoked or the request is a persistent one, then the old handler is replaced by the new handler. If the new handler is a null pointer, then a handler will no longer be called for the specified condition on that request.

The request conditions currently specified are as follows:

- MPIRT_REQUEST_COMPLETE The associated handler is called when the request in question has been marked complete. For example, if the handler is associated with a nonblocking or persistent send, then the handler is called after the send buffer is available for reuse. (Note that the handler may run concurrently with the process if the process was blocked on an MPI_WAIT on the same request at the time the handler was invoked. If the user wishes to avoid this, he/she must provide explicit synchronization.)
- MPIRT_REQUEST_RELEASE The associated handler is called when all resources associated with the last execution of the request are free. Continuing the above example, if the handler is associated with a nonblocking or persistent send, then this handler is called when all local buffers and network resources have been released. (The overall semantics of this condition are admittedly fuzzy. The condition is necessary, however, in order to guarantee real-time performance in certain circumstances. For example, in the case of the above example, one might want the handler to initiate another send request, guaranteeing that the two sends do not contend for system resources.)

Discussion: As an alternative to the MPIRT_REQUEST_RELEASE condition, we may wish to strengthen the notion of request completion for real-time systems to include the release of all system resources–not just the user buffer.

The request handler is assumed to be "full-weight." That is, it can execute any MPI call or system-specific synchronization call and may run for an indeterminate amount of

time. (I.e., it is not restricted like a signal handler.) Also, handlers do not implicitly "consume" their request(s). The request passed to a handler can still be waited on or freed by the process before or after the handler is called, unless the handler itself explicitly frees the request. A request is not actually freed by MPI/RT until all of the handlers⊤ (Fin1) associated with that request have been called. Even if the process or another handler⊥ (Fin1) has called MPI_REQUEST_FREE on the request prior to the execution of the handler, the request is still valid and can be queried using the appropriate calls. One complication is MPI_CANCEL: If a request is cancelled prior to the execution of the handlers, the handlers for each condition are called in turn may note the fact that the request has been cancelled via MPI_TEST_CANCELLED.

Discussion: A suggestion was made to include a priority parameter in MPIRT_REQUEST_POST_HANDLER . ⊤ (Fin1)

⊥ (Fin1)

8.10.2 Application Event-Driven Real-Time MPI

Introduction

This section provides limited functionality that supports only some of many existing event-driven models currently in use by real-time and embedded systems.

The main goal of the event-driven approach of real-time MPI is to help the application to control the run-time environment in which it is running with explicit scheduling of MPI, computation activities and their resource usage. Coordination is required between MPI, the operating system, and communication and network protocols to enforce the schedules.

In a nutshell, an application using event-driven MPI will be able to specify intervals *guarded* by specified events in order to bound the resource usage of communication and computation activities.

The limited functionality presented here contains:

1. MPIRT_REQUEST_GUARDED that allows the application to bind a guarded activity associated with the request with an interval of time specified by events,
2. MPIRT_EVENTNAMES_REGISTER and MPIRT_EVENTNAMES_DEREGISTER that allow the application to manipulate event lists,
3. MPIRT_EVENT_GENERATED that allows an application to generate user events.

Currently many applications "wait" on system events or user control messages to schedule a handler, which, in turn, may schedule several application activities: functions, processes, threads, and data transfers. The model for the application event-driven paradigm presented in this section establishes the direct coupling between events and application activities without user created handlers.

Events

Just as MPI provides the interface for data flow, the application event-driven MPI/RT⊥ (Fin1) provides the interface for control flow. The events can be both persistent and one time⊥ (Fin1) only. Three issues need to be addressed for the events. First, who generates an event. Second, who is aware of the event. And finally, how do we distinguish events. ⊤ (Fin1)

MPI/RT makes three types of events available: system events, communication events⊥ (Fin1)

and user events. The type of the event indicates who generated the event and consequently what resources are involved. System events are generated by the platform environment, for example operating system. User is unaware of these events but would like to be able to schedule an activity based upon them. An example of this would be fault-tolerance. An MPI/RT implementation is aware or can be made aware of these events. Due to a preliminary nature of this proposal no system events are presented or defined in this section.

Discussion: In JOD MPI_PROCESS_DIED is defined as the only system event.

In this preliminary version all communication events are associated with the user MPI/RT channels. MPI/RT generates and is aware of all communication events. This proposal currently contains two events associated with the channel that are introduced in the local event-driven section: MPIRT_REQUEST_COMPLETE and MPIRT_HANDLER_COMPLETE. They represent local and global completion of the MPI/RT channel data transfer.

Each event is identified with a name. For the application event-driven paradigm events are not necessarily local to the process or even a node. Each process registers with MPI/RT both the persistent event names that it wants MPI/RT to "monitor" and the persistent event names that the process will generate.

In order to properly match communication events with the guarded activities, 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 example, if the channel is named α then they are: α_local_complete for a local data transfer completion, and α_global_complete for a global data transfer.

The MPIRT_CHANNELS_INIT and MPIRT_CHANNELS_TRANSIT operations specify the channel name 8.7.

User events are dedicated to the synchronization of the resource usage among different processes (nodes) on the platform, and are generated by the application. The user events has 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 other two 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.

Guards and Guarded Activities

For any activity an application can specify events that trigger its start and that trigger its termination if it is not finished yet.

Discussion: *Proposal for merging event-driven and time-driven paradigms.* The main purpose of the events is to *guard* the interval when the activity may use resources. This is analogous to the time-driven paradigm 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 effort, no resources will be used after timeout of the operation time interval.

The time interval of the time-driven real-time MPI/RT contains two events that are specified

by the time stamps. From this perspective time-driven paradigm is just a subset of the event-driven one. There is, however, one critical difference that lie in the ability of the application to schedule its non-MPI activities. For the time-driven paradigm there are existing facilities to start non-MPI activities using OS timers, spin-locks and others. That and the synchronized clocks allows the application to coordinate all of its activities, MPI and non MPI, local and global. There are no analogous mechanisms for event-driven paradigm, 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. ⊤ (Fin1)
⊥ (Fin1)

Discussion: *Relationship to the existing* MPI *functionality.* Currently, in the JOD (chapter 2) there is already a notion of the event; there are also event handlers in External Interfaces (ref to section 9.8). The only event specified there is MPI_PROCESS_DIED, which is a system-level event that an application can monitor. There are ⊤ (Fin1) two relevant operations: MPI_SIGNAL and MPI_MONITOR that appear in JOD. However, only the ⊥ (Fin1) monitor operation makes use of events.

Events "guard" a liveliness interval within which the activity can use resources. While many different activities are of interest for real-time and embedded system for this proposal we concentrated on two types of activities: MPI/RT data transfers over channels and ⊤ (Fin1) "generic" application activities. ⊥ (Fin1)

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 finished by itself yet. If we need more comprehensive arithmetic for event actions or a different one we can add it later. For completeness we may add action *IGNORABLE* for the empty action list.

Discussion: *Proposal for adding time specification for events.* For merging event-driven and time-driven paradigms special event types called time-instances should be allowed. The time-instance will take a specification of the MPIRT_TIME_OBJECT used by the time-driven paradigm. That will allow to specify the time for the beginning and/or timeout for the activity as well as mixed time/event guards.

⊤ (Fin1)
MPI/RT is responsible to deliver events and to trigger (start or stop) an activity if it ⊥ (Fin1) is eligible. The API for application guarded request is

MPIRT_REQUEST_GUARDED(request, start_list, stop_list, start_qos, stop_qos, failure_fn, extra_state)

IN	request	request (persistent handle)
IN	start_list	list of the events that trigger the start of the function (array of strings)
IN	stop_list	list of the events that trigger the stop of the function (array of strings)
IN	start_qos	event-driven qos for the delivery of starting events (MPIRT_QOS)
IN	stop_qos	event-driven qos for the delivery of terminating events (MPIRT_QOS)
IN	failure_fn	qos failure handler (user-defined function MPIRT_QOS_ERROR_FN)
IN	extra_state	user supplied state (choice)

```
int MPIRT_Request_Guarded(MPI_Request request, char **start_list,
            char **stop_list, MPIRT_QOS start_qos, MPIRT_QOS stop_qos,
            MPIRT_QOS_ERROR_FN failure_fn, void extra_state)

MPIRT_REQUEST_GUARDED(REQUEST, START_LIST, STOP_LIST, START_QOS, STOP_QOS,
            FAILURE_FN, EXTRA_STATE, IERROR)
    INTEGER REQUEST,
        CHARACTER*(*) START_LIST, STOP_LIST,
            INTEGER START_QOS, STOP_QOS, FAILURE_FN, EXTRA_STATE, IERROR
```

Recall that MPIRT_CHANNELS_INIT returns a request handle for each channel, and analogous collective channel initialization operations return a request handle for a collective channel. The persistent generalized requests that allow to encapsulate any application activity within a request is introduced in MPI-2 and is now in JOD.

The two arguments start_list and stop_list define the liveliness interval for the guarded request. The start_qos and stop_qos specifies the event QOS, that provides the bounds for the start time of the guarded function after a triggering event happened and stop time after a timeout event happens, regardless if it happened locally or remotely. Separate qualities of service for starting and terminating events allows specification of different guarantees for stopping and stopping an activity. The start_qos and stop_qos used standard event-driven QOS format.

The extra_state provides the list of the input arguments for the guarded activity. If the activity is a communication one, like data transfer over the channel then this parameter will be ignored. For simplicity user can always specify IGNORABLE. For the application computation activities, like functions, processes, threads, extra_state provides input values.

Notice that for communication data transfer this functionality is only needed for the late binding since the early binding can be done using MPIRT_CHANNELS_INIT and by specifying start and stop event lists with their QOS in the event-driven QOS attribute. This MPIRT_REQUEST_GUARDED should still be used to start the channel use.

Discussion: *Extensions.* The issue of periodicity and its effect on event names will need to be discussed further. The existing MPI_SIGNAL can be used in an MPI implementation to send an

event to a proper place. Alternatively MPI_MONITOR can be enhanced to include events discussed above. Notice also, that while action list specified only by the "sending" side, both sending and receiving sides may be aware of the action. For example, the function failure_fn of the receiving side will be triggered if the finishing event is triggered.

Discussion: There is still a few unaddressed issues. First, what functionality is allowed in the guarded activities. If an application is allowed to create new processes, communicators, register for events and generate new events and so on then an implementation may not be able to guarantee quality of service for new as well as existing processes. For this first version of application event-driven paradigm it may be sufficient to restrict application guarded activities functionality. We may not allow any MPI/RT and MPI functionality within the application guarded activities. An application can only use persistent channel requests for data transfers in the application event-driven paradigm.

The second issue addresses the reentry properties of the guarded activities. It is quite clear that guarded persistent channels are reentrant. So should be application guarded activities. However for the guarded channel only one instance of it at a time can be asked to run by the application. We should put the same restriction on all guarded activities.

Finally, there is an issue of event queue. How many events can the system deliver and what is the quality of service it can provide?

Queues are an implementation issue. Once we add missing QOS parameters that specify the event interarrival time or more generally the maximum number of event needed to be delivered within a time interval, and the scheduling quantum, it is up to the implementation to define the order in which events should be delivered, queue length, priorities of delivery of the events and other related issues. Queues are nothing more than an implementation mechanism of dealing with events. The issue of what should be "a time interval" for QOS specification and should it only specify local view of global view still needs clarification. But "a time interval" may be application specific since it is highly dependent on the timing granularity of the application.

Event Registration

Each application process registers event names it wants MPI/RT to monitor and event names that the process generates. Recall that an application can only generate user events, hence user event names that the application generates need to be registered with MPI/RT. MPI/RT is already aware where and how system and communication events are generated.

Discussion: All the event names that the process registers for appear in the start or stop list that guards an activity within the process and the process generated events that are used in MPIRT_EVENT_GENERATED. So in reality there is no need for registration, a compiler can pick the event names itself. Currently, the full registration of all events that are used to guard a process activity is presented.

The following operations register and deregister event names:

MPIRT_Eventnames_Register(events, generated_events, comm)

IN	events	list of the events (array of strings)
IN	generated_events	list of the events generated by the process (array of strings)
IN	comm	communicator (handle)

```
int MPIRT_Eventnames_register(char **events,char **generated_events,
            MPI_Comm comm)

MPIRT_EVENTNAMES_REGISTER(EVENTS, GENERATED_EVENTS, COMM, IERROR)
    CHARACTER*(*) EVENTS, GENERATED_EVENTS,
        INTEGER COMM, IERROR
```

MPIRT_Eventnames_Deregister(events, comm)

IN	events	list of the events (array of strings)
IN	comm	communicator (handle)

```
int MPIRT_Eventnames_deregister(char **events, MPI_Comm comm)

MPIRT_EVENTNAMES_DEREGISTER(EVENTS, COMM, IERROR)
    CHARACTER*(*) EVENTS,
        INTEGER COMM, IERROR
```

The events list for MPIRT_EVENTNAMES_REGISTER specifies all events that this process wants MPI/RT to handle and generated_events specifies all events that this process can generate. For MPIRT_EVENTNAMES_DEREGISTER the events list specifies the list of events that process no longer wants MPI/RT to monitor or the process will generate. Since all the names are unique at least over the scope of the communicator, MPI/RT implementation can always distinguish between monitored and generated events. The comm does not mean that events delivery should be done over this communicator. The communicator just defines the scope for the events the process wants MPI/RT to monitor as well as the scope for delivery of the events that the process generates.

Discussion: The use of communicator for the scope of event deliveries does not allow MPI/RT to see all the events for the process and hence may hinder potential optimization. This may require an application to use several communicators for event specification. The older version of event specification without a communicator allows the maximum flexibility both for an application and an implementation. It also left the scope of the events open. This meant that when a new process is created it can register to receive events that are already generated by existing processes of the application. However, application can always use only MPI_COMM_WORLD to get the same global application specification view for static applications. The dynamic version is a lot harder. The proposal was just trying to completely separate event delivery "network" from message/data delivery one. This separation was not physical but mental. That would still allow implementation a complete flexibility to use any network for delivery of global events. (see advice to implementors below.)

Event Delivery

The issue of how the events are delivered to the guarded activity is left to the implementation.

Advice to implementors. There were several discussions on the topic of the event delivery. The following methods were suggested:

1. Between all the nodes of the communicator establish point-to-point channels. MPI/RT will use these channels to delivery events. Using the registration operations together with the system and MPI/RT generated events, MPI/RT creates matching tables for all registered events. Upon receiving messages on these channels, MPI/RT checks if the event is sufficient to activate/deactivate an activity that is guarded by the received event(s).

   Similarly, during the registration phase an implementation can map each event type to a tag. When an event is generated or when MPI/RT become aware of an event broadcast a zero size message with the event tag. Locally for each event type (message tag) MPI/RT has a handler that is waiting for this message tag and invokes appropriate waiting application activity.
2. Similar to the above but uses signals (MPI/RT or system) for event delivery.
3. Hardware assisted event delivery.

(*End of advice to implementors.*)

User Generated Event Notification

The following operation is provided for the application to notify MPI/RT about a user generated event.

MPIRT_EVENT_GENERATED(event, comm)

IN	event	event name (string)
IN	comm	communicator (handle)

```
int MPIRT_Event_generated(char event, MPI_Comm comm)
```

```
MPIRT_EVENT_GENERATED(EVENT, COMM, IERROR)
    CHARACTER* EVENT,
        INTEGER COMM, IERROR
```

The event name in event must be in the list generated_events of the MPIRT_EVENTNAMES_REGISTER.

8.11 Priority-Driven MPI/RT

Discussion: Recommendations were made for the following proposals:

- Errors for priorities must be defined.
- Priority models be defined only with respect to a start time.

- Two categories of priorites: Emergency and Other.
- A line of demarcation for events and priorities.
- The limitations of mixing paradigms must be defined.
- Priority proposal discussions should be cross-referenced to the discussions on adhoc models.

In the priority-based real-time programming paradigm, process and message priorities are used to meet timing specifications. However, because process priorities can be handled in a variety of ways, it is extremely difficult to provide portable mechanisms to specify them, and MPI/RT does not directly address this issue.

8.11.1 Message Priority

In MPI/RT priorities are specified and fixed per channel by a field in the QOS argument to the channel creation calls. As with other QOS parameters, the processes at the input and output end of the channel must provide the same priority or an error occurs. (NOTE: We still need to work out the range of process priorities as well as the mechanism for setting the process priority field of the QOS block.)

Because varying platforms may provide different levels of support for message priority at the OS level and below, MPI/RT specifies very 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. For example, if the user initiates a low-priority nonblocking send, the begins a high-priority send, the low-priority send would be stalled in favor of the high-priority send.

8.11.2 Process Priority

As stated above, MPI/RT makes no attempt to correlate process and message priorities, and indeed, has nothing to do with process priorities whatsoever (with the possible exception of calls in the dynamic process chapter, which may be passed priority information via the "info" argument, and the case of request handlers, mentioned below). Such functionality should instead be provided by domain-specific middleware. Such middleware could, for example, set the priority of a process to the priority of the messages it receives. In the absence of middleware, the application developer must manage process priorities explicitly. For example, if the programmer wishes for processes receiving high priority messages to have a high process priority, the process priority must be set explicitly. Similarly, in the case of request handlers associated with a prioritized message, the user of MPI/RT must explicitly correlate the priority of the handler to that of the message by creating a thread, for example, that has a priority corresponding to that of the message, and allowing that thread to handle the message. Alternatively, the request handler mechanism could be used to alter the priority of the receiving process, providing an implementation of the middleware mechanism described above. Notice that this implies that request handlers must almost certainly be run at a higher priority that any MPI/RT process or thread. Thus, while an

MPI/RT *implementation* may need to concern itself with process priorities, the *interface* itself does not.

8.12 Best Effort MPI/RT

Discussion: The issues that affect soft real-time are related to best efforts of hard real time QOS. Discussions on soft real-time, priority models, adhoc real-time and mixed paradigms should be cross-referenced as some of the issues are related.

Some real-time systems have no explicit deadlines and may be interactive. Interactive systems try to achieve adequate response times. Interactive parallel visualization is an example of a non hard real-time application. These interactive systems have hard and soft deadlines, with interactive behavior with the user. A mechanism for handling this mixture of deadlines as well as providing QOS for the interactive portion must be available.

Discussion: Interactive real-time systems can take advantage of the notion of priority preferences. By assigning priorities and weighting the priorities by preferences, interactive QOS may be achieved. More discussion is required.

The correctness of real-time systems depends on the logical result of the computation and the time at which results are produced. For soft real-time systems, the user has flexibility with respect to hard deadlines. If certain deadlines are missed, the system continues to function correctly. Any system that tolerates intermittent delays may be considered soft. The degree that the system is allowed to miss deadlines is the key factor in defining the soft real-time system behavior.

Discussion: Proposal QOS: Quality of service parameters to define the deviation from the hard deadline could be considered. This gives flexibility to the system, where specific deadlines are hard and others are soft, the hard deadlines may be a reference point for defining the offset for the soft deadlines. One mechanism to achieve QOS is a guarantee to tolerate a percentage of timeouts within a period. Another mechanism is to have priorities with preferences.

8.13 Issues in Resource Constrained Systems

For resource-constrained systems, the following issues become more important than in other uses of MPI:

- Small amounts of program space (code bloat unacceptable),
- Small amount of data space (buffering limited, maximum message sizes may be limited),
- Simplified programming environments as compared to full-blown OS's,
- Static loading environments more nearly compatible with MPI-2's view of the world.

Such systems may also have real-time requirements.

MPI is already migrating into resource-constrained prototype systems; hence it is interesting to offer suggestions, and possibly additional profile language, to address this particular space of applications and systems. One particular approach will be to offer a set of subset implementation profiles for resource-constrained systems, so that if subsetting should occur, it can be done according to a systematic convention documented in the MPI-2 standard.

8.13.1 Resource Constrained MPI

There are users within the resource-constrained computing arena that would like to use MPI, but are constrained by extreme limitations on local memory and storage space. These users often run code stored in non-volatile memory, such as FLASH, or ROM. While the use of an efficient linker can dramatically reduce the size of MPI libraries, the resulting binaries are still far too large to be incorporated into firmware. There are three particular areas of concern.

- Large executable size due to the many functions in MPI and their interdependence in many cases. (i.e., resource-constrained MPI/RT may only be able to support a subset of MPI/RT functionality.)
- The amount of buffer space on the receiver side. (i.e., the illusion of infinite slack may be particularly untenable.)
- The amount of buffer space on the sender side. (i.e., the ability to pack derived datatypes may be restricted.)

As a result, we recommend that only a core set of MPI/RT functionality be required for resource-constrained MPI/RT. (Exactly which routines should be preserved remains to be discussed.) In fact, because buffer space may be limited or nonexistent, the user should only expect that the synchronous and ready send operations are available (these routines require no explicit buffer space at the receiver) and that only the built-in datatypes are provided (neither requires buffer space at the sender or the receiver). The combination of these two restrictions would seem to completely eliminate the need for buffer space (except, perhaps, for the case of collective communication, which may require space for intermediate values). Alternatively, the user could be expected to provide buffer space explicitly via routines such as MPI_BUFFER_ATTACH and be restricted to using buffered send operations (as well as synchronous and ready ones). Also, direct memory transfer between source and destination buffers may be implementable only when the source of a message is explicitly specified in a receive call, so the use of MPI_ANY_SOURCE may be precluded.

8.14 Instrumentation

Instrumentation is an essential aspect in providing application developers with the metrics needed to monitor quality of service assurances and fine tune specific platform configurations [7]. These metrics support performance portability, maintainability and fault tolerance. Real-time instrumentation includes, but is not restricted to monitoring application performance and monitoring MPI/RT performance. Other performance monitoring directly related to the overhead of MPI/RT operations will be implementation dependent.

An important benefit from performance monitoring is the ability to capture global and local resource utilization information.

Currently, there is little information available to the user concerning internal MPI events. In some circumstances where timing is critical, an application could benefit from information about times of resources used by internal MPI (and native) communication. Real-time instrumentation must be sensitive to any impact on the timing behavior of an application and its communication. To minimize this impact, MPI/RT instrumentation will include interfaces to external monitoring and/or event loggers. This design promotes implementation independence. Real-time instruments will not duplicate efforts provided in profiling tools, although some of the information collected may be similar. The distinction between profiling and instrumentation will be defined by global and local resource requirements and impact to timing requirements. Implementors may choose to use "profiling" hooks when applicable if performance can be achieved.

Discussion: *A proposal to extend performance monitoring:* A proposal was made to extend MPI/RT performance monitoring to accommodate layered libraries. Most real-time applications have performance instruments to monitor application resource usage associated with computations such as FFT's. A proposal was made to include instrumentation for layered libraries as part of a solution to integrate resource usage monitoring into MPI/RT. Monitors for layered libraries may be created, deleted, reset, started and stopped. The ability to start and stop must include specific start, mark and record functions. These functions mark the start time and end time with a provision for counting the number of monitoring accesses for this monitor. This is a method for marking and recording an interval of statistics.

8.14.1 MPI/RT Monitoring

Run time instruments support performance monitoring, decision analysis and fault tolerance. The MPI monitoring API is designed to monitor and output metrics. Performance monitoring for both application specific information (layered libraries), and MPI/RT communication metrics are accommodated. Monitoring a block of code, monitoring channel(s), monitoring over a communicator and monitoring layered libraries are supported. For systems that interface to an external performance monitoring capability, MPI/RT monitoring also provides an interface.

The user may define the parameters to be monitored and specify the implementation-dependent information (handle) required to manage resources. The monitor info (handle) allows the user to specify implementation-dependent information for managing resources. MPI info objects (handles) are described in MPI-2 in Chapter 4, Process Creation and Management. These handles may be created, set, deleted and a status taken.

MPI/RT monitoring supports quality of service assurances. Metrics that are obtained from performance monitoring also provide decision analysis criteria for conditional heuristics. These heuristics support fault tolerance policies and support load balancing schemes.

Runtime instruments are created, deleted and reset. Activation of real-time instrument monitoring is defined by a start and end function to capture a snapshot of any size. Some inaccuracies may occur when MPI monitoring is turned off and some communications have not completed. More than one monitor may execute simultaneously. The execution of some monitors may be mutually exclusive and the responsibility of non-conflicting monitors is currently left to the user.

8.14.2 MPI/RT Monitoring Management

⊥ (Fin1)

The creation and destruction of monitoring is designed to decouple the burden of overhead for initialization, reset and finalization from the ability to start and stop monitoring respectively. Initialization is performed during monitor create.

⊤ (Fin1)

MPI/RT Monitoring Management - Block of Code

⊥ (Fin1)

This monitoring is over an application snapshot or block of code and is designed to monitor MPI communication.

MPIRT_MONITOR_CREATE(monitor_info, monitor_handle, errcode)

IN	monitor_info	monitor information (info)
OUT	monitor_handle	monitor handle (handle)
OUT	errcode	indicates reason for failure (integer)

⊤ (Fin1)

```
int MPIRT_Monitor_create(MPI_Info Monitor_info,
              MPIRT_Monitor_handle *monitor_handle, int *errcode)
```

```
MPIRT_MONITOR_CREATE(MONITOR_INFO, MONITOR_HANDLE, ERRCODE, IERROR)
    INTEGER MONITOR_INFO, MONITOR_HANDLE, ERRCODE, IERROR
```

⊥ (Fin1)

⊤ (Fin1)

MPIRT_MONITOR_DELETE(monitor_handle)

INOUT	monitor_handle	monitor handle (handle)

```
int MPIRT_Monitor_delete( MPIRT_Monitor_handle *monitor_handle)
```

```
MPIRT_MONITOR_DELETE(MONITOR_HANDLE, IERROR)
    INTEGER MONITOR_HANDLE, IERROR
```

⊥ (Fin1)

Discussion: *Proposal to reset a monitor:* It was proposed that a monitor be reset and that initialization be coupled to create. The function MPIRT_MONITOR_INIT was deleted and the function MPIRT_MONITOR_RESET was added.

⊤ (Fin1)

⊥ (Fin1)

MPIRT_MONITOR_RESET(info, monitor_handle, error)

IN	info	monitor information (info)
IN	monitor_handle	monitor handle (handle)
OUT	error	specifies error (integer)

⊤ (Fin1)

```
int MPIRT_Monitor_reset( MPI_Info info, MPIRT_Monitor_handle monitor_handle,
             int *error)
```

```
MPIRT_MONITOR_RESET(INFO, MONITOR_HANDLE, ERROR, IERROR)
    INTEGER INFO, MONITOR_HANDLE, ERROR, IERROR
```

⊥ (Fin1)

⊤ (Fin1)

MPI/RT Monitoring Management - Requests

⊥ (Fin1)

The following allow monitoring of requests. An example is a snapshot over channels. A unique monitoring handle is associated with a channel request. These monitors are created, deleted and reset.

MPIRT_MONITOR_REQUEST_CREATE(request, info, count, monitor_handle, errors)

IN	request	array of requests of length count (array of requests)
IN	info	array of monitor information of length count (array of info)
IN	count	number of requests (non-negative integer)
OUT	monitor_handle	handle array of length count (array of monitor handle)
OUT	errors	specifies error for each request(integer array)

⊤ (Fin1)

```
int MPIRT_Monitor_request_create(MPI_Request request[], MPI_Info info[],
             int count, MPIRT_Monitor_handle *monitor_handle[],
             int *errors[])
```

```
MPIRT_MONITOR_REQUEST_CREATE(REQUEST, INFO, COUNT, MONITOR_HANDLE, ERRORS,
             IERROR)
    INTEGER REQUEST(*), INFO(*), COUNT, MONITOR_HANDLE(*), ERRORS(*),
    IERROR
```

⊥ (Fin1)

⊤ (Fin1)

MPIRT_MONITOR_REQUEST_DELETE(count, monitor_handle, errors)

IN	count	number of handles (non-negative integer)
INOUT	monitor_handle	handle array of length count (array of monitor handle)
OUT	errors	specifies error for each monitor handle(integer array)

```
int MPIRT_Monitor_request_delete(int count,
             MPIRT_Monitor_handle *monitor_handle[], int *errors[])
```

```
MPIRT_MONITOR_DELETE(COUNT, MONITOR_HANDLE, ERRORS, IERROR)
    INTEGER COUNT, MONITOR_HANDLE(*), ERRORS(*), IERROR
```

⊥ (Fin1)

MPIRT_MONITOR_REQUEST_RESET(info, monitor_handle, count, errors)

IN	info	array of monitor information of length count (info)
IN	monitor_handle	handle array of length count (handle)
IN	count	number of handles (non-negative integer)
OUT	errors	specifies error for each monitor handle(array of integers)

⊤ (Fin1)

```
int MPIRT_Monitor_request_reset( MPI_Info info[],
            MPIRT_Monitor_handle monitor_handle[], int count,
            int *errors[])
```

```
MPIRT_MONITOR_REQUEST_RESET(INFO, MONITOR_HANDLE, COUNT, ERRORS, IERROR)
    INTEGER INFO(*), MONITOR_HANDLE(*), COUNT, ERRORS(*), IERROR
```

⊥ (Fin1)

MPI/RT Monitoring Management - Communicator

The following three functions provide monitoring over a communicator.

MPIRT_MONITOR_COMM_CREATE(comm, monitor_info, monitor_handle, errcode)

IN	comm	communicator to monitor (handle)
IN	monitor_info	monitor information (info)
OUT	monitor_handle	monitor handle (handle)
OUT	errcode	indicates reason for failure (integer)

⊤ (Fin1)

```
int MPIRT_Monitor_comm_create(MPI_Comm comm, MPI_Info Monitor_info,
            MPIRT_Monitor_handle *monitor_handle, int *errcode)
```

```
MPIRT_MONITOR_COMM_CREATE(COMM, MONITOR_INFO, MONITOR_HANDLE, ERRCODE,
            IERROR)
    INTEGER COMM, MONITOR_INFO, MONITOR_HANDLE, ERRCODE, IERROR
```

⊥ (Fin1)

⊤ (Fin1)

MPIRT_MONITOR_COMM_DELETE(comm, monitor_handle)

IN	comm	communicator to monitor(handle)
INOUT	monitor_handle	monitor handle (handle)

```
int MPIRT_Monitor_comm_delete(MPI_Comm comm,
            MPIRT_Monitor_handle *monitor_handle)
```

```
MPIRT_MONITOR_COMM_DELETE(COMM, MONITOR_HANDLE, IERROR)
    INTEGER COMM, MONITOR_HANDLE, IERROR
```

⊥ (Fin1)

MPIRT_MONITOR_COMM_RESET(comm, info, monitor_handle, errcode)

IN	comm	communicator to monitor (handle)
IN	info	monitor information (info)
IN	monitor_handle	monitor handle (handle)
OUT	errcode	indicates reason for failure(integer)

⊤ (Fin1)

```
int MPIRT_Monitor_comm_reset(MPI_Comm comm, MPI_Info info,
            MPIRT_Monitor_handle monitor_handle, int *errcode)
```

```
MPIRT_MONITOR_COMM_RESET(COMM, INFO, MONITOR_HANDLE, ERRCODE, IERROR)
    INTEGER COMM, INFO, MONITOR_HANDLE, ERRCODE, IERROR
```

⊥ (Fin1)

⊤ (Fin1)

8.14.3 MPI/RT Monitoring Control

⊥ (Fin1)

Monitoring control includes starting and stopping monitoring. Monitoring may be started and stopped at any time after MPI_MONITOR_CREATE has been called and before MPI_MONITOR_DELETE is completed.

When monitoring requests, monitoring may be started and stopped at any time after MPI_MONITOR_REQUEST_CREATE has been called and before MPI_MONITOR_REQUEST_DELETE is completed.

When monitoring over a communicator, monitoring may be started and stopped at any time after MPI_MONITOR_COMM_CREATE has been called and before MPI_MONITOR_COMM_DELETE is completed.

When monitoring is stopped, all flags are set to the default settings. When monitoring is started, the user may specify which specific elements (options) are to be monitored during the session, by defining the parameters. ⊤ (Fin1)

MPI/RT Start and Stop Monitoring

⊥ (Fin1)

The ability to start and stop monitoring is accomplished with the following basic functions: ⊤ (Fin1)

MPIRT_MONITOR_START(monitor_handle, monitor_params, errcode)

IN	monitor_handle	monitor handle (handle)
IN	monitor_params	parameters for monitoring (choice)
OUT	errcode	indicates reason for failure (integer)

```
int MPIRT_Monitor_start(MPIRT_Monitor_handle monitor_handle,
             void monitor_params, int *errcode)
```

```
MPIRT_MONITOR_START( MONITOR_HANDLE, MONITOR_PARAMS, ERRCODE, IERROR)
    INTEGER MONITOR_HANDLE, MONITOR_PARAMS, ERRCODE, IERROR
```

⊥ (Fin1)

MPIRT_MONITOR_STOP(monitor_handle)

IN	monitor_handle	monitor handle (handle)

⊤ (Fin1)

```
int MPIRT_Monitor_stop(MPIRT_Monitor_handle monitor_handle)
```

```
MPIRT_MONITOR_STOP( MONITOR_HANDLE, IERROR)
    INTEGER MONITOR_HANDLE, IERROR
```

⊥ (Fin1)

Since specific monitoring may be mutually exclusive in specific systems, error codes are a convenient mechanism for monitoring to report conflicts; if the implementation handles this level of discrimination.

Discussion: *Proposal to Start and Stop Monitoring for Layered Libraries:* A proposal was made to have more than one type of start and stop monitoring functions to accommodate layered libraries which require an additional level of granularity.

8.14.4 MPI/RT Interface to External Monitor

Discussion: A proposal was made to add an interface to accommodate independent third party monitoring capabilities. Some of this capability may be achieved with info handles which are implementation dependent.

8.14.5 MPI/RT Metrics

Real-time instrumentation for MPI/RT provides metrics that will help the user determine QOS parameters for an application request. These metrics will include as a minimum set: number of timeouts (collective operations, channel operations); frequency and periods of timeouts.

Advice to implementors. Examples of information of interest for MPI/RT performance measures may include execution times, timeouts, counts and workloads.

1. Execution times

   Total MPI/RT execution time is time spent executing services for MPI/RT from a defined start point to a defined end point.

   ```
   execution time = (end time - start time)
   ```

   Some general categories for MPI/RT execution times may include time spent in a channel operation, a collective operation or a communicator group.

2. Counts

   The number of processes, number of times an MPI/RT communication is completed and the number of timeouts can be correlated with execution times to measure performance. This is a measure of quality of service.

   Timeout Counts are confined to timeouts of MPI/RT communications, and are extensible to collective operations and communicator groups.

   Collective operations counts may include a count of the number of processes participating and the number of MPI/RT communications completed.

   Communicator group counts may include a count of the number of MPI/RT communications completed, total number MPI/RT communication timeouts and the number of MPI/RT collective operations completed

3. Workload

   Workload is a measure of the message sizes and traffic load over time and may defined per communication group. To develop metrics for workload, the following information may be collected per communicator:

   - message sizes (large small)
   - frequency large number of large messages

- frequency small number of small messages
- monitoring frequency of recordings taken

Where message size definition (large and small) is recommended by the implementor.

Time is determined by the start and end time in the snapshot created by the MPI/RT start and stop monitoring bindings.

(*End of advice to implementors.*)

8.14.6 MPI/RT Misc Monitor Operations

The ability to use monitoring information as heuristics for conditional branching decisions in the system may be achieved with existing MPI/RT functionality.

Discussion: *Proposal to Trigger Event in Response To Monitor:* The ability to associate a special function with a monitor may be accomplished by using the MPI/RT guarded functions and event registration. These functions are described in JOD 8.10). The purpose of this activity would be to process the current results of a monitoring activity. These results would be available to the user and could provide heuristics.

8.14.7 MPI/RT Output Monitoring Results

This function is provided for systems that do not offer a mechanism for outputting the performance information (metrics) automatically and for support of decision analytic heuristics.

MPIRT_MONITOR_RESULTS(request, monitor_handle, fn, params)

IN	request	request to output monitoring (handle)
IN	monitor_handle	handle for monitoring (handle)
IN	fn	function to execute (MPIRT_Function)
IN	params	parameters for function (choice)

```
int MPIRT_Monitor_results(MPI_Request request,
            MPIRT_Monitor_handle monitor_handle, MPIRT_Function fn,
            void params)
```

```
MPIRT_MONITOR_RESULTS(REQUEST, MONITOR_HANDLE, FN, PARAMS, IERROR)
    INTEGER REQUEST, MONITOR_HANDLE, FN, PARAMS, IERROR
```

The monitoring information is returned locally to the user. This function may not be called before MPIRT_MONITOR_STOP and must be called before MPIRT_MONITOR_DELETE is called.

No provisions are planned to allow cancellation of monitoring.

8.14.8 Monitoring Other Software Services

In addition to the MPI/RT instruments described in this section, an option for distinguishing other services related to MPI/RT is available.

MPI calls are supported with software services. For each MPI implementation, there is a software layer between the hardware and the MPI application layer bindings. This layer may differ for each architecture and may consist of operating system services, vendor specific MPI services and/or other services. The overhead from software services with respect to an MPI call may be of interest to the user and/or implementor. Some systems are not capable of accurately collecting timing information for these services. Implementations that can not support this option, return a warning message when the option is selected.

This option can provide a level of granularity to the MPI/RT instruments defined in this section. Typical information derived from monitoring other services might be the time spent in operating system calls. These operating system calls would be confined to those that support MPI/RT.

8.15 Fault Handling

A fault tolerant system can continue the correct performance of its specified tasks in the presence of hardware and/or software faults [11]. Faults can result in errors and errors can lead to failures. Fault handling includes the ability to assess operability, detect faults and recover from failures. Operability assessment is preventive, while fault detection, fault recovery and reconfiguration are generally late response indicators.

Faults, errors and failures occur in the physical, informational and user universes respectively. The detection, location, and containment of faults contributes to successful fault recovery. Faults differ in duration ranging from transient to monotonic. The source of faults may be hardware and/or software. Faults originating in the physical universe may propagate errors into information space. Eventually, user space will become aware of these errors, and failures are reported [11]. The ability to minimize system degradation from the time a fault is detected through recovery is an important aspect of fault tolerance.

Operability assessment is an early response indicator for prevention of system degradation in the presence of faults. Using metrics provided in MPI/RT instrumentation and heuristic algorithms, MPI/RT communication faults may be detected early and errors contained before error propagation can occur.

Multiprocessor fault tolerance is characterized by replication and replacement policies [11]. Replication is designed to prevent data loss by providing full redundancy. Replacement policies are vulnerable to data loss, if the replacement policy is in response to a late indicator. Replacement for MPI/RT will include recoverable communicators and fully redundant communicators with reconfiguration.

Real-time systems can have integrated support for fault tolerance at all levels in the system design including the operating system, hardware, software, and application programs. The effects of real-time system recovery and reconfiguration on MPI/RT influences the ability of MPI/RT to meet its quality of service requirements.

Discussion: Issues in fault tolerance. In general, error containment is more easily controlled in distributed systems. Recovery and process distribution is more easily implemented in shared memory systems [11]. Distributed shared memory systems are a hybrid of the two. This is of interest to implementors. Consequently, the MPI/RT standard should attempt to compensate for

natural biases imposed by any fault tolerance policy or offer implementation advice.

Another important issue in the MPI/RT fault tolerance design is at what level(s) is fault handling provided. An assumption is made that fault handling will include early warning error detection, and recovery (redundancy and reconfiguration).

Object redundancy and quality of service extensions are proposed to implement fault tolerance for all MPI communication. This is invisible to the user, but some of the associated fault handling must be defined by the implementor and user. Instrumentation monitoring provides metrics to support real-time system fault monitoring. The metrics are visible to the user, but the algorithmic implementations that use the metrics can be hidden from the user. In addition, there are explicit MPI/RT fault handling functions that are visible to the user.

- Redundancy of channels
- Reconfiguration by recreating a communicator
- Reconfiguration of a redundant communicator

These MPI/RT functions must be designed and implemented to consider allocation of resources, fault handling, error handling and restoration of services.

8.15.1 Early Indicators

Operability assessment is an early warning indicator for error detection and supports error containment. Real-time systems can integrate a global fault tolerance for early error detection and containment. The ability to support these global fault handling schemes will be provided with MPI/RT instrumentation and monitoring. MPI/RT will not define heuristic algorithms, voting algorithms or software reliability algorithms for MPI. Instead, the MPI/RT monitoring section will define metrics that are specific to MPI/RT parameters of execution that can support real-time system fault tolerance. Consequently, no new functions are expected in this section.

Discussion: *Proposal:* The definition of and indicators for an MPI/RT fault must be defined before metrics can be identified. This section is TBD until the functions to collect metrics are defined in the MPI/RT monitoring section. A discussion of the origins of these metrics will be provided in this section.

8.15.2 Replication and Redundancy

Replication in a real-time system includes spatial and temporal techniques implementing forward, backward and rollback redundancy. A system provides redundancy in the form of replicated processes or physical nodes [2]. When failure occurs, duplicate process(es) or physical node(s) may already be executing to prevent data loss. Characteristically, the redundant node or process does not output data, but instead dumps data until conscripted into service. Redundant units are often referred to as online-standby.

Redundant Channels

For MPI/RT, some replication is provided with channel redundancy. By allowing redundant channels, data loss is minimized as a tradeoff to performance. The user may choose redundancy for MPI/RT channels with a potential sacrifice to performance portability.

Discussion: The following function is proposed:

MPIRT_CHANNELS_REPLICA(bufs, errors)

IN	bufs	array of send or receive buffers(array)
OUT	errors	specifies success/non-success(integer)

All channel requests for these buffers will be subject to redundancy after this function is called. Redundancy can not be cancelled for the duration of the application after it is defined.

The ability to create redundant channels could be handled by the MPI implementation in the structure that controls object fault tolerance. The design of this function assumes that MPI/RT fault tolerance for objects is implemented and that a strategy for alternate fault tolerance mechanisms is part of the schema. This is a very controversial function, since it assumes that specific fault handling is available in MPI/RT and implementable. It also duplicates resource allocation for each channel created for these buffers. This makes it a very restrictive function.

Discussion: *Proposal:* Some mechanism to associate the redundant channels (standby online) to the on-line channels is required. When a channel fault occurs, the on-line standby channel will be conscripted into service. This can be accomplished with an MPI fault handler. An alternative is to have the MPI implementation be responsible. Under these conditions, there must be a structure that couples the two sets of channels, and the structure must be accessible to the MPI implementation, but hidden from the user.

If redundant channels are designed into a structure as an alternate control mechanism for fault tolerance, then when a request to allocate an object is made, an attempt is made to satisfy the temporal redundancy level for each request. If the temporal redundancy level requirement includes a redundant channel, then it may be more difficult to satisfy this request.

The proposal on MPI object fault tolerance attempts to address some of these issues.

Redundant Real-Time System Components

Redundancy can be integrated into various components of the real-time system in support of both static and dynamic fault handling.

Discussion: Real-time system redundancy approaches are independent of but also influence MPI/RT. For example, rollback recovery schemes may be unique to shared memory, distributed memory, distributed shared memory and database systems. Special hardware is often required to support these recovery schemes, and software algorithms may also be implemented. For techniques such as rollback recovery with checkpoints, complications arise for MPI/RT implementations.

This complex set of recovery schemes for various system components will require a simple interface to MPI/RT. Performance monitoring in the MPI/RT instruments section will provide some of this interface.

8.15.3 Reconfiguration

The ability to reconfigure MPI/RT communication groups in response to both early warning and late response indicators supports fault tolerance recovery. The difficulty in meeting real-time quality of service commitments and implementing reconfiguration is a non-trivial problem.

Discussion: *Proposal: Basis for Reconfiguration.* There are two perspectives for supporting reconfiguration. A communication group can be created as a subset of an existing group (recoverable communicator) or may be replaced by a fully redundant communicator group.

MPI/RT can guarantee no data loss only to the level of assurance provided by the system implementation.

Channels are not reconfigurable. This will undoubtly create severe confusion in the real-time application domain.

Recoverable Communicator

Seamless replacement without degradation is an optimal goal of fault tolerance policies. When a fault is detected, a new comm group may be created by the application using the following MPI-2 functionality MPI_COMM_CREATE and/or MPI_COMM_SPLIT.

The user responds to the fault by adding a group of processors to to the new communicator MPI_COMM_CREATE. This is the simplest approach and is effective only if there is a transient non-fatal fault and the user continues to monitor the fault. This puts responsibility on the user.

If the faulty communicator can be split and new communicators created, then fault tolerance can also be achieved. This requires using MPI_COMM_SPLIT followed by MPI_COMM_CREATE. The user can pull the faulty group and build a new communicator from the split communicator.

These techniques imply a high risk of data loss and are probably more effective as a recovery mechanism to early indicators. In addition, critical sections of the applications will probably not be able to tolerate the timing demands imposed by these strategies.

Redundant Reconfigurable Communicators

Redundant communicators have origins in both redundancy and reconfiguration policies. No data loss occurs, since the redundant communicator is designed to concurrently execute with the on-line communicator. If the on-line communicator fails, then the recovery policy transitions to a reconfiguration policy. This is difficult to implement, since the failed communicator must eventually be restored and made available again. This operates on an optimistic policy that errors are detectable, recoverable and infrequent.

Discussion: *Proposal for a redundant communicator function.*

MPIRT_COMM_CREATE_REDUNDANT(comm, dupcomm, status)

IN	comm	communicator to be duplicated (handle)
IN	dupcomm	redundant communicator (handle)

When the redundant communicator is placed on-line, the user will be responsible for assigning the new redundant communicator to back it up. To avoid maintenance problems, faulty communicators must be maintained by the user. The alternative is to force the MPI/RT implementation to use MPI_COMM_SPLIT and/or MPI_COMM_CREATE to keep track of faulty groups and create new ones. This is not practical.

The following is an example of an MPI/RT redundant, reconfigurable communicator:

- In response to a fault, the user places the redundant communicator on-line. This is accomplished by the MPI/RT function MPIRT_RECONFIGURE. The implementation keeps track

of redundancy relationships. If redundancy is not specified after reconfiguration, then it is automatically cancelled.

- The user may assign a new redundant communicator for on-line standby using MPIRT_COMM_CREATE_REDUNDANT.
- The faulty communicator is reconstructed into new communicators using MPI_COMM_SPLIT followed by MPI_COMM_CREATE.

  This is the responsibility of the user and placed on a user-defined list. This is similar to a free-list.

The user must keep track of these redundant communicators and not misuse the communicators. When a on-line standby communicator transitions to on-line the implementation must effectively transition the output behavior associated with the new communicator. This could be a formidable task for the implementation. Consequently, when the new redundant unit is defined, the new mappings between the on-line communicator and the redundant must also be made by the implementation.

8.15.4 MPI Communication Fault Handling

If undetected, communication faults can lead to system degradation, data loss and non-recoverable errors [2, 3].

Timeouts for Traditional MPI Functionality

In this section, the proposal to add timeouts to MPI-1.1 and MPI-2 functions is a step towards an underlying fault tolerance prevention schema for MPI.

An example of an MPI function with a timeout is TBD.

MPI Objects

Discussion: *Proposal: Fault Handling and* MPI *Objects.* To implement realistic global real-time MPI fault handling, the design must accommodate fault detection, recovery, and reconfiguration. To support MPI executing in real-time, timeouts were added to all MPI functions. Real-time system fault tolerance can take advantage of these timeouts for object redundancy schemes. Redundancy of executing objects promotes safety and reliability in the presence of faults. The major disadvantage is resource consumption. The following is an example of how redundant objects may be designed and implemented.

A joint is a dedicated pointer that extends the current MPI object model and utilizes timeouts to support fault handling. A joint is a bridge between accessed objects and contexts (handles) that attempt to access the objects. A joint has the following attributes [1]:

- context independent pointer to object body.
- object owner/user justification - No users can be linked to an owner object that is to be deleted.
- resource requirements
- protection scheme - Authorization determined before binding to user's context. Authorization refers to action required, not the path to find the object. Identification refers to path.
- time constraint for executable object - Supported by the proposal in this section to add timeouts to all MPI functions.

- replica/alternate control mechanism for fault tolerance scheme

By implementing an object model for MPI opaque objects, a global MPI fault handling could be achieved. The MPI opaque object model separates opaque objects managed by system memory (not directly accessible by user) and provides handles in user space to manage the objects. When each request to allocate an object is made, a temporal redundancy level must be satisfied. When the temporal redundancy level can not be satisfied, the fault handling alternate control mechanism in the joint is consulted.

This fault tolerance, designed to handle monotonic and transient faults, is based on an allocation algorithm. If a system constructed from objects and resources is a computation, then when allocation is initiated, the following are specified: fault tolerance objectives; alternatives for carrying out the computation; computation timing constraints. The computational alternatives must satisfy physical and temporal redundancy as defined by the user [1].

Quality of Service Extensions

Discussion: *Proposal:* Fault handling and Extensions to QOS MPI quality of service guarantees that all messages are delivered to all destinations. If MPI/RT extends this requirement to all messages delivered correctly to all destinations or NO messages are delivered to any destination, then some level of fault tolerance can be achieved. The addition of timeouts to all MPI functions supports these criteria. The line of demarcation between guaranteeing correct delivery and not delivering any messages is left to the implementor.

It is conceivable that MPI/RT should also support the following:

- All receives will receive the messages in the same order sent [1].

This is a more difficult requirement to satisfy, although it is implied in the real-time quality of service parameter.

8.15.5 Fault Handling Definitions

This section will define an MPI/RT fault and the mechanism for handling a fault including:

- What is an MPI/RT fault?
- How is an error detected?
- How is an error reported?
- How do system faults affect MPI/RT and MPI?
- What are the responses to a fault?

8.16 Communicator Semantics with Channels

One of the most important features of MPI is its support of safe communication spaces. The introduction of persistent channels that refer back to communicators opens up issues of inheritance of such channels during an MPI_COMM_DUP.

Because channels are not explicitly attached to collective operations, but rather quality of service, there is no specific issue here.

By default, a duplicated communicator would have no real-time properties, and the channels of the parent communicator would not be available for use by the child communicator. This is clearly suboptimal in certain circumstances.

Qualities of service come in two forms: intensive and extensive. Extensive quality of service includes bandwidths, rates, and so on, and would have to be shared, or used exclusively when communicators are duplicated. Intensive properties, of which priority is the prime example, do not need to be subdivided but can be shared without loss of generality.

Thus, the following discussion is needed:

- To support passage of channels through a duplicated communicator
- To provide a means to duplicate a persistent collective operation, whether real-time or regular (Collective Chapter),
- To distinguish intensive and extensive quality of service, so that some may be shared, and others used duplicatively.

8.17 Errors and Error Codes

Discussion: Error codes will be discussed in this section.

8.18 APPENDIX A: Deadlock Avoidance/Recovery

This section contains old discussions saved for later reference on fault tolerance. It is important to be able to prevent, avoid or in the worst case detect and recover from deadlocks. The problem of deadlocks is difficult in distributed real-time systems and it is exacerbated with prioritization.

Two related issues are as follows:

- Livelock — it happens when two interacting tasks miss deadlines due to one doing some secondary activity when the other tries to rendezvous.
- Orphan processes — in the presence of spawn capability (a process creating other processes) and failures.

⊤ (Fin1)
⊥ (Fin1)

MPI/RT will utilize timeouts to help applications deal with deadlock in the face of possible remote faults or remote resource exhaustion. Both sender- and receiver-based timeouts are to be considered.

Specifically, we expect non-blocking, persistent communication to be used to recover from deadlock. The following allows one to asynchronously start a receive, and then wait immediately for its completion, with a defined timeout.

```
MPI_Status status;
MPI_Request request;
MPI_Receive_init(buf, count, datatype, src, tag, comm, &request);
...
for(;;)
```

```
{
   MPI_Start(request);
   MPI_Wait_timeout(request, &status);
}
```

Alternatively, a single call,

```
MPI_Start_and_wait(request, status, timeout);
```

could be considered, and might have value in that it is atomic. To be complete, this proposal should address "all" modes, as well as single persistent operations.

A similar sequence of calls would be used to achieve send with timeout. This approach makes only an additive addition to the number of calls in MPI. ⊤ (Fin1)

⊥ (Fin1)

Bibliography

[1] A. K. Agrawala and S. T. Levi. *Real-Time System Design.* McGraw-Hill, 1990.

[2] K. P. Birman and R. Van Renesse. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Press, 1993.

[3] T. D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the impossibility of group membership. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 322–330, 1996.

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

[6] Rainer Händel and Manfred N. Huber. *Integrated Broadband Networks: An Introduction to ATM-Based Networks.* Addison-Wesley, 1991.

[7] F. Jananian. Run-time monitoring of real-time systems. In D. Son, editor, *Advances in Real-Time Systems.* Prentice Hall, 1995. Chapter 18.

[8] E. D. Jensen. Asynchronous decentralized realtiem computer systems. In W. A. Halang and A. D. Stoyenko, editors, *Real-Time Computing*, NATO ASI Series, Series F: Computer and System Sciences, vol. 127, pages 347–372. Springer-Verlag, 1992.

[9] Hermann Kopetz and Paulo Verissimo. Realtime and dependability concepts. In Sape Mullender, editor, *Distributed Systems, 2nd Edition*, chapter 16, pages 411–446. Addisson-Wesley, 1993.

[10] 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.

[11] D. K. Pradhan. *Fault Tolerant Computer System Design.* Prentice Hall, 1996.

[12] J.R. Rice and S. Rosen. Numerical analysis problem solving system. In *Proc. 21st ACM Nat. Conf.*, pages 51–56. ACM Publications, 1966.

[13] Mark Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference.* MIT Press, 1996.