MPI-2 Journal of Development

Message Passing Interface Forum

July 18, 1997

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

Abstract

This document is a supplement to the MPI-2 document. It describes topics that are part of the MPI-2 Journal of Development.

Contents

Chapter 1

Introduction to the MPI Journal of Development

During the course of the two years that the MPI Forum convened, many ideas were explored that are not included in the MPI-2 Standard. In some cases the ideas were not deemed finished enough to be considered formally. In others, much work was invested in refining the specifications, but in the end the Forum, after thorough consideration, voted the material out of the Standard. In a third case, an entire chapter was withdrawn from consideration for inclusion when it was realized that it could not be finished in time for the release of the Standard.

This document is not part of the MPI Standard in any way. It was thought, however, that the work invested in these topics should be preserved in some form. Those who wish to implement the functionality described here, as extensions to MPI, might find the trial specifications described here useful, but are not bound in any way by them. If at some future date the MPI Forum reconvenes and wishes to add some of this functionality as official extensions to the MPI specification, it is not bound by any "backward compatibility" restrictions to conform with this Journal of Development. If someone does implement the functionality described here, the `MPI_` prefix should *not* be used, since these functions are not part of the MPI Standard.

The chapters of the Journal of Development vary in the amount of consideration they were given by the Forum and the way they found their way into the Journal of Development:

The chapters in the JOD are

- Chapter 2, Spawning Independent Processes, includes some elements of dynamic process management, in particular management of processes with which the spawning processes do not intend to communicate, that the Forum discussed at length but ultimately decided not to include in the MPI Standard.
- Chapter 3, Threads and MPI, describes what it might look like to incorporate into MPI some functions normally associated with threads and shared variables.
- Chapter 4, Communicator ID, describes an approach to providing names for communicators. The MPI-1 Forum carefully defined communicators so that communicator creation need not always be a synchronizing operation. This makes the notion of a global identifier for each communicator problematic. An approach to the problem is discussed in this chapter.

- Chapter 5, Miscellany, discusses Miscellaneous topics in the MPI JOD, in particular single-copy routines for use in shared-memory environments and new datatype constructors.
- Chapter 6, Toward a Full Fortran 90 Interface, describes an approach to providing a more elaborate Fortran 90 interface.
- Chapter 7, Two-phase Collective Communication, describes a specification for certain non-blocking collective operations. The Forum spent considerable time discussing non-blocking collective operations, which present difficult implementation issues, particularly in the absence of threads. In the end, non-blocking collective operations were first replaced by the related "two-phase" collective operations, which offer many of the same advantages as non-blocking operations, but eventually decided not to include them in the Standard. The specifications are presented in this chapter.
- Chapter 8, Real Time, discusses MPI support for real time processing. Real-time requirements suggest both additions to, and subtractions from, the MPI-1 and MPI-2 specifications. The real-time subcommittee of the MPI Forum invested considerable work in this chapter, but eventually decided not to propose it for inclusion in the MPI-2 Standard. The work to produce a specification for a real-time message-passing library that will be strongly related to the MPI specification is ongoing.

Chapter 2

Spawning Independent Processes

These sections of the Dynamic Process chapter received thorough consideration and even received one vote of approval. They failed to receive the required second vote, however. The functions described here deal with spawning and managing processes with which the spawning processes do not intend to communicate. They are designed to be consistent with mpifuncMPI_SOMM_SPAWN.

2.1 Starting Independent Processes

This section describes how to spawn *independent* processes, that is, processes that do not communicate with their parent. These may be MPI processes (i.e., processes that call MPI_INIT) or not.

MPI_SPAWN_INDEPENDENT(command, argv, maxprocs, info, flag, group, array_of_errcodes)

IN	command	name of program to be started (string)
IN	argv	arguments to command (array of strings)
IN	maxprocs	maximum number of processes to start (integer)
IN	info	info object telling the runtime system where and how to start the processes (handle)
IN	flag	says whether processes are MPI processes or not (integer)
OUT	group	group of spawned processes (handle)
OUT	array_of_errcodes	one code per process (array of integer)

```
int MPI_Spawn_independent(char command[], char* argv[], int maxprocs,
             MPI_Info info, int flag, MPI_Group* group,
             int array_of_errcodes[])
```

```
MPI::Group::Spawn_independent(const char command[], const char* argv[],
             int maxprocs, const MPI::Info& info, int flag,
             int array_of_errcodes[])
```

```
MPI_SPAWN_INDEPENDENT(COMMAND, ARGV, MAXPROCS, INFO, FLAG, GROUP,
             ARRAY_OF_ERRCODES, IERROR)
    CHARACTER*(*) COMMAND, ARGV(*)
    INTEGER INFO, MAXPROCS, FLAG, GROUP, ARRAY_OF_ERRCODES(*), IERROR
```

This routine spawns a set of processes that do not establish communication with the parent processes. It therefore returns a group instead of an intercommunicator. It is not collective, and is called on a single process. The flag argument specifies whether or not the children are MPI processes, that is, whether or not they call MPI_INIT. In other respects, this routine is similar to MPI_SPAWN.

2.1.1 MPI processes

If flag has the value MPI_MPI then MPI_SPAWN_INDEPENDENT is identical to MPI_SPAWN except that it does not establish communication with the children, returning an MPI group rather than an intercommunicator, and it is not collective. Successful completion does not indicate that the processes have called MPI_INIT successfully, only that the processes were started successfully.

The child processes are *required* to call MPI_INIT when flag is MPI_MPI. All children spawned in a single call to MPI_SPAWN_INDEPENDENT have the same MPI_COMM_WORLD, which is separate from that of the parent. In the children, MPI_COMM_PARENT has an empty remote group.

2.1.2 Non-MPI processes

If flag has the value MPI_NONMPI then the child processes are assumed to be non-MPI processes. MPI_SPAWN_INDEPENDENT is not required to do any special setup to ensure that a call to MPI_INIT in the children will be able to establish communication among the children. The effect of calling MPI_INIT in the children is undefined (but see advice to implementors in Section 5.5.2 of the MPI-2 document).

2.2 Starting multiple independent processes

MPI_SPAWN_MULTIPLE_INDEPENDENT(count, array_of_commands, array_of_argv, array_of_maxprocs, array_of_info, flag, group, array_of_errcodes)

IN	count	Number of commands (integer, size of each of the following arrays)
IN	array_of_commands	programs to be executed (array of strings)
IN	array_of_argv	arguments for commands (array of array of strings)
IN	array_of_maxprocs	maximum number of processes for each command line to start (array of integers)
IN	array_of_info	info objects telling the runtime system where and how to start processes (array of handles)
IN	flag	says whether processes are MPI processes or not (integer)
OUT	group	group of spawned processes (handle)
OUT	array_of_errcodes	one error code per process (array of integer)

```
int MPI_Spawn_multiple_independent(int count, char* array_of_commands[],
             char** array_of_argv[], int array_of_maxprocs[],
             MPI_Info array_of_info[], int flag, MPI_Group* group,
             int array_of_errcodes[])
```

```
MPI::Group::Spawn_multiple_independent(int count,
             const char* array_of_commands[], const char** array_of_argv[],
             const int array_of_maxprocs[], const MPI::Info array_of_info[],
             int flag, int array_of_errcodes[])
```

```
MPI_SPAWN_MULTIPLE_INDEPENDENT(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV,
             ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, FLAG, GROUP,
             ARRAY_OF_ERRCODES, IERROR)
    INTEGER COUNT, ARRAY_OF_INFO, ARRAY_OF_MAXPROCS, FLAG, GROUP,
    ARRAY_OF_ERRCODES(*), IERROR
    CHARACTER*(*) ARRAY_OF_COMMANDS(*), ARRAY_OF_ARGV(COUNT, *)
```

MPI_SPAWN_MULTIPLE_INDEPENDENT is identical to MPI_SPAWN_INDEPENDENT except that there are multiple executable specifications. The first argument, count, gives the number of specifications. Each of the next four arguments are simply arrays of the corresponding arguments in MPI_SPAWN_INDEPENDENT. See the description of MPI_SPAWN_MULTIPLE (Section 5.3.3 of the MPI-2 document) for an explanation of array_of_argv in Fortran.

2.3 Nonblocking operations

MPI_ISPAWN_INDEPENDENT(command, argv, maxprocs, info, flag, group, array_of_errcodes, request)

IN	command	name of program to be started (string)
IN	argv	arguments to command (array of strings)
IN	maxprocs	maximum number of processes to start (integer)
IN	info	info object telling the runtime system where and how to start the processes (handle)
IN	flag	says whether processes are MPI processes or not (integer)
OUT	group	group of spawned processes (handle)
OUT	array_of_errcodes	one code per process (array of integer)
OUT	request	request object (handle)

```
int MPI_Ispawn_independent(char command[], char* argv[], int maxprocs,
             MPI_Info info, int flag, MPI_Group* group,
             int array_of_errcodes[], MPI_Request* request)
```

```
MPI::Group::Ispawn_independent(const char command[], const char* argv[],
             int maxprocs, const MPI::Info& info, int flag,
             int array_of_errcodes[], MPI::Request& request)
```

```
MPI_SPAWN_INDEPENDENT(COMMAND, ARGV, MAXPROCS, INFO, FLAG, GROUP,
             ARRAY_OF_ERRCODES, REQUEST, IERROR)
    CHARACTER*(*) COMMAND, ARGV(*)
    INTEGER INFO, MAXPROCS, FLAG, GROUP, ARRAY_OF_ERRCODES(*), REQUEST,
    IERROR
```

MPI_ISPAWN_MULTIPLE_INDEPENDENT(count, array_of_commands, array_of_argv, array_of_maxprocs, array_of_info, flag, group, array_of_errcodes, request)

IN	count	Number of commands (integer, size of each of the following arrays)
IN	array_of_commands	programs to be executed (array of strings)
IN	array_of_argv	arguments for commands (array of array of strings)
IN	array_of_maxprocs	maximum number of processes for each command line to start (array of integers)
IN	array_of_info	info objects telling the runtime system where and how to start processes (array of handles)
IN	flag	says whether processes are MPI processes or not (integer)
OUT	group	group of spawned processes (handle)
OUT	array_of_errcodes	one error code per process (array of integer)
OUT	request	request object (handle)

```
int MPI_Ispawn_multiple_independent(int count, char* array_of_commands[],
             char** array_of_argv[], int array_of_maxprocs[],
             MPI_Info array_of_info[], int flag, MPI_Group* group,
             int array_of_errcodes[], MPI_Request* request)
```

```
MPI::Group::Ispawn_multiple_independent(int count,
             const char* array_of_commands[], const char** array_of_argv[],
             const int array_of_maxprocs[], const MPI::Info array_of_info[],
             int flag, const int array_of_errcodes[],
             MPI::Request& request)
```

```
MPI_ISPAWN_MULTIPLE_INDEPENDENT(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV,
             ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, FLAG, GROUP,
             ARRAY_OF_ERRCODES, REQUEST, IERROR)
    INTEGER COUNT, ARRAY_OF_INFO, ARRAY_OF_MAXPROCS, FLAG,GROUP,
    ARRAY_OF_ERRCODES(*), REQUEST, IERROR
    CHARACTER*(*) ARRAY_OF_COMMANDS(*), ARRAY_OF_ARGV(COUNT, *)
```

2.4 Signalling Processes

It is not possible to send messages to a process represented only by a (group, rank) pair. For instance, it is not possible to send a message to an MPI process spawned by MPI_SPAWN_INDEPENDENT, since no communicator is available.

Here we describe simple mechanisms to manage such processes. MPI_SIGNAL sends a signal to a process, and MPI_PROCESS_MONITOR provides a mechanism to detect when a process dies or changes state.

MPI_SIGNAL(group, rank, signal)

IN	group	group containing process to signal (handle)
IN	rank	rank of process to signal (integer)
IN	signal	signal type (integer)

```
int MPI_Signal(MPI_Group group, int rank, int signal)
```

```
MPI::Group::Signal(int rank, int signal) const
```

```
MPI_SIGNAL(GROUP, RANK, SIGNAL, IERROR)
    INTEGER GROUP, RANK, SIGNAL, IERROR
```

MPI_SIGNAL sends a signal to a process represented by a (group, rank) pair. If rank is MPI_ANY_SOURCE, the signal is sent to every process in the group.

Where POSIX signals are supported, signal is a signal defined by POSIX. It is the responsibility of an implementation to translate between signals; in other words, a SIGINT that has value 3 on system A must be delivered as a SIGINT on system B, even if system B uses the value 5 for SIGINT. If there is no corresponding signal, the operation is erroneous and the result is undefined.

An MPI implementation must also support the MPI-defined signal type MPI_SIGNAL_KILL. Sending this "signal" will reliably kill a process and attempt to do the necessary system-level cleanup. In order to allow appropriate cleanup to occur, an implementation could, for instance, send a catchable signal (e.g., SIGINT) followed by a noncatchable signal (SIGKILL) if the process had not exited after a short time interval. MPI requires only that the process be reliably killed. Applications should not rely on a specific implementation.

Signals are delivered asynchronously. That is, when the MPI_SIGNAL function returns, the signal may not yet have been delivered. If a signal is undeliverable (e.g., the process has already exited), the signal is silently dropped.

If an MPI implementation can determine, at the time MPI_SIGNAL is called, that the process is already dead, it may return the error MPI_ERR_NOPROCESS. If MPI_ANY_SOURCE had been specified and some of the processes are known to be dead while others are not, no error is returned.

Advice to users. Signals do not provide any of the reliability or guarantees of regular MPI communication. A high quality implementation will deliver signals quickly and reliably, but applications should never depend on ordering. For example, a signal sent after a message may arrive before the message does. Since signals may not be queued, when multiple signals are sent, only one may get through. Some MPI implementations may use signals internally. If an application attempts to use these signals, unexpected behavior may result. Finally, it may not be possible at the implementation level to restrict the effects of a signal to a single MPI process, so that a side-effect of sending a signal to a process may be that other processes receive that signal or are killed. (*End of advice to users.*)

2.5 Notification of change in state of a process

In order to manage processes an MPI application must be able to be notified when a process exits.

MPI_PROCESS_MONITOR(group, rank, event, request)

IN	group	a group of processes (handle)
IN	rank	rank of process to be monitored (integer)
IN	event	a flag specifying what event to be notified about (integer)
OUT	request	a request object (handle)

```
int MPI_Process_monitor(MPI_Group group, int rank, int event,
              MPI_Request* request)
```

```
MPI::Group::Process_monitor(int rank, int event, MPI::Request& request)
```

```
MPI_PROCESS_MONITOR(GROUP, RANK, EVENT, REQUEST, IERROR)
    INTEGER GROUP, RANK, EVENT, REQUEST(*), IERROR
```

This function provides a general method for an MPI process to detect a change in state of another process. The request completes when the process changes state as indicated by event.

event is an integer specifying a change of state that the calling process wishes to detect. The most important state-change and the only one MPI currently defines for event is MPI_PROCESS_DIED. For this case, request completes when the process terminates. If a process has already terminated, the request completes immediately. Process termination is defined by the implementation, and may conceivably include cases where a process has become unreachable or whose status cannot be determined. If an application wants to be notified of several different events, it must call MPI_MONITOR once for each event. If multiple requests are active for a single type of event, all requests will complete when the event occurs.

A request generated by MPI_PROCESS_MONITOR must progress regardless of whether any other MPI functions are called. MPI_REQUEST_FREE on a monitor request deletes the request.

> *Advice to users.* As in other cases in MPI, freeing the request marks it for deletion but does not necessarily complete the request. Therefore a handler associated with a MPI_PROCESS_MONITOR request may be called after the request is freed. To prevent MPI from continuing to perform action on behalf of the request, it is necessary to call MPI_CANCEL. (*End of advice to users.*)

Future versions of the MPI standard may specify new types of events, and implementations may define new ones as well. If an application wants to be notified of several different events, it must call MPI_PROCSSS_MONITOR once for each event.

For a request generated by MPI_PROCESS_MONITOR, MPI_WAIT returns a status that contains no useful information (i.e., the status is undefined).

Rationale. Another possibility would be to get the exit code from the status. However, the process may not have exited - it may have become unreachable. It may also have been killed by a signal and what we'd want to know is what signal killed it. In the end, we might have to reimplement Unix wait(). (*End of rationale.*)

MPI_GROUP_MONITOR(group, event, array_of_requests)

IN	group	a group of processes (handle)
IN	event	a flag specifying what event to be notified about (integer)
OUT	array_of_requests	array of request objects (array of handles)

```
int MPI_Group_monitor(MPI_Group group, int event,
             MPI_Request array_of_requests[])
```

```
MPI::Group::Monitor(int event, MPI::Request array_of_requests[]) const
```

```
MPI_GROUP_MONITOR(GROUP, EVENT, ARRAY_OF_REQUESTS, IERROR)
    INTEGER GROUP, EVENT, ARRAY_OF_REQUESTS(*), IERROR
```

This convenience function is equivalent to calling MPI_PROCESS_MONITOR once for each process in group. The ith element (numbered from 0) of array_of_requests is associated with rank i in the group. array_of_requests must have a number of elements greater than or equal to the size of group.

Chapter 3

Threads and MPI

This chapter contains a proposed definition of what mutexes and conditional variables would look like if integrated into a threaded MPI library. It was not voted on.

3.1 Global thread functions

It is useful to extend thread coordination functions to work across an MPI group of processes, including processes not under the same kernel. Such extensions can be implemented portably using MPI communication (with handlers) and proxies at each node to support the functions globally (of course, much more efficient implementations will be possible on many systems).

These functions can also be used to synchronize threads within one process, using the communicator MPI_COMM_SELF.

We outline below some possible choices.

3.1.1 Mutexes

MPI_MUTEX_CREATE(comm, mutex_attr, mutex)

IN	comm	communicator for group sharing mutex
IN	mutex_attr	info object for mutex attributes
OUT	mutex	mutex object

```
int MPI_Mutex_create(MPI_Comm comm, MPI_Info mutex_attr, MPI_mutex mutex)
```

```
MPI_MUTEX_CREATE(COMM, MUTEX_ATTR, MUTEX, IERROR)
    INTEGER COMM, MUTEX_ATTR, MUTEX, IERROR
```

```
MPI::Mutex::Create(const MPI::Comm& comm, const MPI::Mutex_attr&
             mutex_attr)
```

Missing: Need to specify predefined mutex attributes. No avoid problems with priority inversion, may need to require that all processes in the group of comm have the same priority ceiling. If left implementation dependant, could be a problem in strongly typed languages such as C++.

Creates a mutex object that is shared by all processes in the group of comm. (One could, instead, return a newcomm and further overload communicators.) This is the extension of pthread_mutex_init

MPI_MUTEX_FREE(mutex)

INOUT	mutex	mutex object

```
int MPI_Mutex_free(MPI_Mutex *mutex)
```

```
MPI_MUTEX_FREE(MUTEX, IERROR)
    INTEGER MUTEX, IERROR
```

```
MPI::Mutex::Free(void)
```

Free the mutex object (should that be collective?) This is the extension to pthread_mutex_destroy (POSIX says that the behavior of a pthread_mutex_destroy call on a locked mutex is undefined. We can leave it this way, or have our usual, more lenient implementation of free, that waits until objects are not busy. The later is consistent both with POSIX and with MPI, but may not be consistent with current implementations of POSIX threads.)

MPI_MUTEX_LOCK(mutex)

INOUT	mutex	mutex to be locked

```
int MPI_Mutex_lock(MPI_Mutex mutex)
```

```
MPI_MUTEX_LOCK(MUTEX, IERROR)
    INTEGER MUTEX, IERROR
```

```
MPI::Mutex::Lock(void)
```

MPI_MUTEX_TRYLOCK(mutex, flag)

INOUT	mutex	mutex to be locked
OUT	flag	success flag (boolean)

```
int MPI_Mutex_trylock(MPI_Mutex mutex, int *flag)
```

```
MPI_MUTEX_TRYLOCK(MUTEX, FLAG, IERROR)
    INTEGER MUTEX, FLAG, IERROR
```

```
MPI::Mutex::Trylock(int& flag)
```

MPI_MUTEX_UNLOCK(mutex)

INOUT	mutex	mutex to be unlocked

```
int MPI_Mutex_unlock(MPI_Mutex mutex)

MPI_MUTEX_UNLOCK(MUTEX, IERROR)
    INTEGER MUTEX, IERROR

MPI::Mutex::Unlock(void)
```

These are the extensions of the POSIX calls pthread_mutex_lock, pthread_mutex_unlock and pthread_mutex_unlock. The lock call is blocking (like a wait), whereas the trylock function in nonblocking, like a test.

3.1.2 Condition variables

MPI_COND_CREATE(comm, cond_attr, cond)

IN	comm	communicator for group sharing condition variable
IN	cond_attr	info object for condition attributes
OUT	cond	condition object

```
int MPI_Cond_create(MPI_Comm comm, MPI_Info cond_attr, MPI_Cond *cond)

MPI_COND_CREATE(COMM, COND_ATTR, COND, IERROR)
    INTEGER COMM, COND_ATTR, COND, IERROR

MPI::Cond::Create(const MPI::Comm& comm, const MPI::Info& cond_attr)
```

Creates a condition variable that is shared by all processes in the group of comm. (We could, here, too, overload communicators, rather than having a new opaque object.)

MPI_COND_FREE(cond)

INOUT	cond	condition object

```
int MPI_Cond_free(MPI_Cond *cond)

MPI_COND_FREE(COND, IERROR)
    INTEGER COND, IERROR

MPI::Cond::Free(void)
```

Free the condition object.

These are the MPI extensions to pthread_cond_init and pthread_cond_destroy.

MPI_COND_SIGNAL(cond)

INOUT	cond	condition to be signaled

```
int MPI_Cond_signal(MPI_Cond cond)
```

```
MPI_COND_SIGNAL(COND, IERROR)
    INTEGER COND, IERROR
```

```
MPI::Cond::Signal(void)
```

MPI_COND_BROADCAST(cond)

INOUT	cond	condition to be signaled

```
int MPI_Cond_broadcast(MPI_Cond cond)
```

```
MPI_COND_BROADCAST(COND, IERROR)
    INTEGER COND, IERROR
```

```
MPI::Cond::Broadcast(void)
```

MPI_COND_WAIT(cond)

INOUT	cond	condition to wait on

```
int MPI_Cond_wait(MPI_Cond cond)
```

```
MPI_COND_WAIT(COND, IERROR)
    INTEGER COND, IERROR
```

```
MPI::Cond::Wait(void)
```

These are the MPI extensions of the POSIX calls pthread_cond_signal, pthread_cond_broadcast, and pthread_cond_wait. The cond_wait call is blocking. The cond_signal call unblocks at least one of the threads waiting on the condition (if any) and the cond_broadcast call unblocks all threads that are waiting on the condition.

Chapter 4

Communicator ID

This chapter contains a proposal to give communicators identifiers. The Forum voted, after lengthy discussion, not to include it in the standard.

Profiling libraries often want a communicator ID so they can associate calls that occur on the same communicator. This is also useful since processes can have the same rank in different communicators or the same process can be involved in different communicators. Thus, the following call is given:

MPI_COMMUNICATOR_ID(comm, id)

IN	comm	Communicator to return id of (handle)
OUT	id	id of comm (integer)

```
int MPI_Communicator_id(MPI_Comm comm, int *id)
```

```
MPI_COMMUNICATOR_ID(COMM, ID, IERROR)
    INTEGER COMM, ID, IERROR
```

```
MPI::Comm::Id(int& id) const
```

> *Advice to implementors.* The above function can be implemented as a macro in C and Fortran, and as an inline in C++. (*End of advice to implementors.*)

> *Rationale.* Since these functions are likely to be used in profiling libraries, minimizing the overhead is very important. Allowing the use of a macro or inline function helps to accomplish this. When using a macro, the only loss is the error code but this seemed acceptable. (*End of rationale.*)

To specify the uniqueness of ID returned, the key MPI_COMM_ID_IS_UNIQUE is available on MPI_COMM_WORLD. This attribute returns one of the following values:

MPI_COMM_ID_UNIQUE Means that the values returned by MPI_COMMUNICATOR_ID are unique for each communicator (on different processes) and for the entire MPI job. In the unlikely event that the number of communicators created exceeds the range of an int in C or C++ and INTEGER in Fortran, the implementation may then begin reusing values previously used. At that point, the implementation should not reuse a value another time until it has once again used up all the values that ID can hold.

MPI_COMM_ID_NOT_UNIQUE The value returned in ID does not meet the requirements to be MPI_COMM_ID_UNIQUE.

Rationale. Most libraries would prefer a globally unique communicator ID. However, MPI was carefully specified so as to avoid needing such a value in an implementation. This was done to reduce potential overheads in communicator creation. Requiring that a communicator ID be unique could add overhead to all communicators in some implementations. However, some implementations may already have a unique ID that can be given for no additional cost. This key allows for both possibilities.

The attribute key is placed on MPI_COMM_WORLD since it is felt that it is only really useful if it is true throughout the complete job. (*End of rationale.*)

Advice to users. Intercommunicators can represent a problem. This occurs since there are two groups which have overlapping ranks but the ID is only unique to the intercommunicator. The alternative is to make the ID unique to the group in an intercommunicator. However, this would make it more difficult to associate operations which work on the whole intercommunicator, e.g., MPI_BARRIER. Having different IDs for the different groups would make the semantics different between inter- and intra-communicators. For this reason an intercommunicator returns a unique value for all members. (*End of advice to users.*)

Discussion:

The above proposal has the following problem. Suppose you have two communicators which were created by separate MPI jobs. By using, for example, the server capabilities in the dynamic chapter, the two comms are joined into a single intercomm. The two original intracomms are now, in some regards, part of the same MPI job. They may have had the same comm_id before they were joined. It does not seem possible make sure they will be unique in this circumstance. One possible solution is to note this problem and say that the two original comms cannot communicate and aren't really part of one MPI job. Only the new intercomm is and this must be unique.

Chapter 5

Miscellany

This chapter contains the MPI Forum's work on a number of proposals. Two new datatype constructors were designed, but concern over the details of how they might be implemented prevented them from being voted in. In addition, there are proposals that take advantage of single-copy possibilities for message-passing on shared-memory machines. These were not discussed in detail and were not voted on.

5.1 New Datatype Constructors

5.1.1 Simple Struct Types

MPI_TYPE_SIMPLE_STRUCT(count, array_of_blocklengths, array_of_types, newtype)

IN	count	number of blocks (integer)
IN	array_of_blocklengths	number of elements in each block (array of integer)
IN	array_of_types	type of elements in each block (array of handle)
OUT	newtype	new datatype (handle)

This datatype constructor is present to allow portable struct datatypes to be constructed. A portable datatype is a basic datatype or a datatype constructed from portable datatypes using one of the constructors MPI_TYPE_CONTIGUOUS, MPI_TYPE_VECTOR, MPI_TYPE_INDEXED, or MPI_TYPE_SIMPLE_STRUCT.

This type constructor is similar to MPI_TYPE_STRUCT, except that the user does not supply an array of displacements. Instead, the successive blocks are assumed to be contiguous. Padding spaces are added, according to the default alignment rules for structure components. This facilitates the construction of datatypes for structures, as the user need not compute displacements.

This can probably be more elegantly expressed in terms of extents.

Example 5.1 The following code declares a structure type and creates a datatype for that structure type.

```
struct record {
   char name;
```

```
   double position[3];
   float  mass;
   }

MPI_Datatype record_type;
MPI_datatype types[3] = {MPI_CHAR, MPI_DOUBLE, MPI_FLOAT};
int          lengths[3] = {1, 3, 1};

MPI_Type_simple_struct( 3, lengths, types, record_type);
```

Rationale. The code above is much simpler and more natural than the current approach, that uses MPI_TYPE_STRUCT, and requires to compute displacements.

It also provides a definition of the record type that is architecture independent (assuming that compilers don't fiddle with the order of fields, which is true in C) – this facilitates transfer of records using RMA, an fits well with various proposals for third party transfers. (*End of rationale.*)

Advice to users. Datatypes constructed using MPI_TYPE_SIMPLE_STRUCT should be used only for structures that use the default alignment rules. They should not be used if alignment rules have been changed, using compiler options or compilation pragmas. (*End of advice to users.*)

Advice to implementors. The datatype constructor MPI_TYPE_SIMPLE_STRUCT is most easily implemented on systems that use *natural* data alignment rules. On such systems, each basic datatype has an extent and an alignment; the alignment of a compound type is the strictest alignment of a component. Padding is added so that each component is aligned at its natural alignment. We can then derive obvious definitions for the extent and alignment of portable MPI datatypes:
The extent and alignment of a basic datatype is the extent and alignment of the corresponding language type.
The alignment of a datatype built with one of the constructors MPI_TYPE_CONTIGUOUS, MPI_TYPE_VECTOR, and MPI_TYPE_INDEXED is the alignment of the component datatype; the extent of the compound datatype is as defined in MPI-1.
The alignment of a datatype built by a call to MPI_TYPE_SIMPLE_STRUCT(n, lens, types, newtype) is $\max_i$ alignment(types[i]); the extent is computed by laying out the components with minimal padding added so that each component starts at its alignment.

Some compilers treat the first component of a structure differently. E.g., the IBM compiler aligns doubles at word boundaries, but align a structure that starts with a double component at doubleword boundaries. The definition of the alignment of a datatype built with MPI_TYPE_SIMPLE_STRUCT is changed, accordingly. (*End of advice to implementors.*)

5.1.2 Contiguous Struct Types

Data structures may contain padding holes, added for alignment. The definition of an MPI datatype, done using the MPI_TYPE_STRUCT constructor, does not distinguish between holes that contain no significant data, and are there only for alignment purposes, and between holes that may contain significant data and were purposefully left out by a user that wishes not to overwrite some of the information in a structure. As a result, MPI cannot overwrite these holes, and this prevents the obvious optimization of shipping a data structure as a contiguous block of data, padding holes included. The type constructor MPI_TYPE_CONTIGUOUS_STRUCT marks any such holes as "don't-care" locations, allowing the implementation to overwrite them.

MPI_TYPE_CONTIGUOUS_STRUCT(count, array_of_blocklengths, array_of_displacements, array_of_types, newtype)

IN	count	number of blocks (integer)
IN	array_of_blocklengths	number of elements in each block (array of integer)
IN	array_of_displacements	byte displacement of each block (array of integer)
IN	array_of_types	type of elements in each block (array of handle)
OUT	newtype	new datatype (handle)

The semantics of MPI_TYPE_CONTIGUOUS_STRUCT are identical to MPI_TYPE_STRUCT, except that, if a receive call uses a datatype constructed with this constructor, then the internal holes in the structure can be overwritten. The same rule (namely, that holes between components can be overwritten) applies also to datatypes constructed using MPI_TYPE_SIMPLE_STRUCT.

Discussion: It is reasonable to assume that holes in layouts defined using MPI_TYPE_SIMPLE_STRUCT contain no significant information, since this datatype constructor is expected to be used for specifying the layout of structures. Users will be required to use MPI_TYPE_STRUCT when holes hold significant information.

Advice to users. Users should arrange the fields in a C struct type in descending order of their size. This arrangement has two advantages: firstly it may reduce the size of the struct by eliminating padding between elements with different alignments (this is generally true for C programs). Secondly, it may increase the number of cases for which the implementation can perform block copies by ensuring that the first element of the structure has the most stringent alignment requirements. (*End of advice to users.*)

Advice to implementors. An implementation may ignore the constructor MPI_TYPE_CONTIGUOUS_STRUCT, and handle it exactly as MPI_TYPE_STRUCT. Thus, there is (almost) no overhead in supporting this constructor, without taking advantage of it. One possible way of taking advantage of it, is to adopt a canonical "wire" representation for structures. Suppose that the compiler uses, by default, natural alignment rules for structures: each basic component is aligned at a natural boundary. Suppose that one adopts natural alignment rules for messages on the wire, in a homogeneous environment: messages are padded so that each basic element is aligned at a natural boundary. Then
(1) A receiver can always decode an incoming message and retrieve correctly the basic components. This because the basic datatype of the element indicates at what

boundary it is aligned.
(2) A sender can block copy a structure onto the "wire" (most likely, onto a message buffer), when the structure is naturally aligned, *relative to the start of the sending buffer*. This optimization does not require that the sending datatype be built using MPI_TYPE_CONTIGUOUS_STRUCT: The MPI library can scan datatypes, when they are committed, and recognize when a datatype (or a datatype fragment) is naturally aligned. Block copying can be used even if the holes in the datatype contain significant information.
(3) A receiver can block copy an incoming message onto the receive buffer when the receiving structure is naturally aligned *relative to the start of the receiving buffer* and holes can be ignored. I.e., when the receiving datatype is built using MPI_TYPE_CONTIGUOUS_STRUCT or MPI_TYPE_SIMPLE_STRUCT and is naturally aligned, relative to its first position.

The same wire representation may be used in a heterogenous environment, in cases where data conversion is done at the destination; the destination has to know, not only the basic datatypes used by the source, but also the alignment rules used by the source.

Example: Consider a C structure `struct {char a[2]; float b[2]; double c}`; assume that floats are 4 bytes and doubles are 8 bytes, and structures are naturally aligned. The memory layout is `[c][c]xx[ffff][ffff]xxxx[dddddddd]`, and the structure is aligned to an 8 byte boundary.

Case 1: The entire structure is communicated. Sender constructs a datatype for structure using MPI_TYPE_STRUCT, MPI_TYPE_SIMPLE_STRUCT or MPI_TYPE_CONTIGUOUS_STRUCT; receiver constructs a datatype for this structure, using MPI_CONTIGUOUS_STRUCT or MPI_TYPE_SIMPLE_STRUCT. Then the layouts in the sender memory, on the wire, and in the receiver memory are identical, and data can be simply copied.

Case 2: communication involves only the second of the two characters, and the remainder of the structure. Receiver constructs the datatype using MPI_CONTIGUOUS_STRUCT. The memory layout is `[c]xx[ffff][ffff]xxxx[dddddddd]`, and the wire layout is `[c]xxx[ffff][ffff]xxxx[dddddddd]`. The datatype is not naturally aligned, so that block copying cannot be used at the sender or the receiver. However, a smart implementation may realize that the fragment starting from the float is naturally aligned, relative to the start of the buffer, so that block copying can be used for this fragment.

Case 3: communication involves only the two floats and the double. The memory layout is `[ffff][ffff]xxxx[dddddddd]`. The layout on the wire is `[ffff][ffff][dddddddd]`. The datatype is not naturally aligned, relative to its start, so that block copying cannot be used. (*End of advice to implementors.*)

Alternatives:

1. No new function MPI_TYPE_CONTIGUOUS_STRUCT: instead, holes can be overwritten only for datatypes built with MPI_TYPE_SIMPLE_STRUCT. The argument is that, at least in the scenario I presented, hole overwriting is taken advantage of only for structures with a natural alignment. But, then, their layout can be specified using MPI_TYPE_SIMPLE_STRUCT.

2. Both sender and receiver are required to use MPI_TYPE_CONTIGUOUS_STRUCT, and the datatypes on both ends have to be structurally equivalent, i.e., defined by the same sequence of definitions. There are several possible variants for such proposal

   (a) This construct can be only used in a homogeneous environment. But, then, we lose portability to heterogeneous environment, and we achieve little that could not be achieved by using MPI_BYTE.

   (b) This construct can be used portably, with enhanced performance obtained in a homogeneous environment. E.g., a homogeneous implementation will use block copying, while a heterogeneous implementation will move element by element (at least, for heterogeneous communicators). This assumes that, in a homogeneous environment, different processes are compiled with the same data alignment options. Also, this runs counter the datatype matching rules in MPI, where only signature matching is required.

   (c) MPI_TYPE_CONTIGUOUS_STRUCT can be matched with other datatypes, as long as signatures match; improved performance is achieved when both ends use MPI_TYPE_CONTIGUOUS_STRUCT. But, then, the wire protocol must be so that individual elements can be extracted, in case the receiver does not use MPI_TYPE_CONTIGUOUS_STRUCT. In this case, asking the sender to use MPI_TYPE_CONTIGUOUS_STRUCT does not provide additional function over the main proposal. The difference is only whether the use asserts that the sending datatype has a natural layout (using MPI_TYPE_CONTIGUOUS_STRUCT), or whether MPI discovers that fact by itself.

5.2 Two Process "Shared" Buffers

It is not uncommon for a pair of processes to engage in frequent pre-determined communications. On example of such a code is a pipeline algorithm in which process A creates data that is subsequently used by process B. Another example is a finite difference approximation model when communicating boundary values (a.k.a. ghost points). On a machine with shared memory, it is possible to share a common buffer to use for exchanging such data. Ownership of the buffer must be established in order for a process to safely fill or empty the buffer. This can be expressed as a produce/comsume relationship. This functionality can be expressed in such a way that it should be no less efficient when using MPI Send/Recv on a distributed memory machine and is simply a pointer pass on shared memory machines.

Discussion: This proposal is prompted by s proposal titled: "Infinite asymptotic bandwidth for thread communication"

We have a similiar protocol inside our system when we work on shared memory machines. Important to us is not only the bandwidth and latency, but also lightweight flow-control. The proposal in its current form solves our problem in particular, but we feel it may prompt discussion that can shape this proposal and Tony's proposal into something that will provide a portable high bandwidth/low-latency mechanism on true SMPs as well as provide high performance in distributed memory machines.

Questions for discussion:

1) Is this really going to be faster on a shared memory machine with a "high-quality" implementation.

2) Does the user visible flow-control provide significant added value?

3) For both shared and distributed memory machines, does predefining the communication pattern provide options for optimization?

```
Example:

Process 0:                             Process 1:

! Make a shared buffer with process 0 as the source, process 1 as the dest
! sbh is the shared buffer handle and buff is some memory allocated
! by MAKE_SHARED_BUFFER
MAKE_SHARED_BUFFER (...,               MPI_MAKE_SHARED_BUFFER (...,
      0, 1, buff, sbh)                           0, 1, buff, sbh)
! Process 0 owns the buffer            ! Process 1 waits for buffer
 ...Fill buff with useful stuff...     ACQUIRE_SHARED_BUFFER (sbh)

! Process 1 releases the buffer
! communicating that it was
! modified (PRODUCEd)
RELEASE_SHARED_BUFFER (sbh, PRODUCE)
                                       ! At this point process 1 unblocks
! Wait for process 0 to use the        ! and has full access to buff
! Information produced before          ...Read buff and do something...
! creating more data
ACQUIRE_SHARED_BUFFER (sbh)            ! Process 1 releases the buffer, but
                                       ! has not modified it (CONSUMEd)
                                       RELEASE_SHARED_BUFFER (sbh, CONSUME)

! At this point process 0 unblocks
! and is free to modify buff
          ...                                ...
FREE_SHARED_BUFF(sbh)                  FREE_SHARED_BUFF(sbh)
```

MPI_MAKE_SHARED_BUFFER (dataType, count, comm, source, dest, buffer, sharedBuffHandle)

IN	dataType	MPI Data Type (contiguous)
IN	count	Number of dataTypes
IN	comm	Communicator
IN	source	First process in pair (initially owns buffer)
IN	dest	Second process in pair
IN	tag	Tag for use in communications and for uniqueness
OUT	buffer	Buffer creaated by make_shared_buffer
OUT	sharedBuffHandle	Handle to describe shared object

```
int MPI_Make_shared_buffer (MPI_Datatype dataType, int, count,
            MPI_Comm comm, int source, int dest,
            void *buffer, MPI_Shared_buff *sharedBuffHandle)
```

```
MPI_MAKE_SHARED_BUFFER (DATA_TYPE, COUNT, COMM, SOURCE, DEST,
                        BUFF_REF, BUFFER_IX, SHARED_BUFF_HANDLE, IERROR)
   INTEGER DATA_TYPE
   INTEGER COUNT
   INTEGER COMM
   INTEGER SOURCE
   INTEGER DEST
   INTEGER BUFF_REFF(*)
   INTEGER BUFFER_IX
   INTEGER SHARED_BUFF_HANDLE
   INTEGER IERROR
```

This routine creates a buffer shared between two processes. The process identified by source initially "owns" the buffer. Source may write to the buffer and manipulate the buffer in any way that makes sense with standard language access (assignment statements, functions calls, etc.). When source is done with the buffer, it must release the buffer for dest to acquire it. Dest (and only dest) can acquire the buffer once released by source. Dest then has the same priviledgs as source previously had. To access a released buffer is considered to be erroneous. This function needs to be viewed as "bi-collective".

Discussion: Do we want to have a version that allows the user to supply a buffer. In that event on a distributed memory machine this becomes a "safe" interface to ready send, on a shared memory machine it may introduce extra copies.

MPI_ACQUIRE_SHARED_BUFFER (sharedBuffHandle)

INOUT	sharedBuffHandle	A shared buffer handle

```
int MPI_Acquire_shared_buffer (MPI_Shared_buff *sharedBuffHandle)
```

```
MPI_ACQUIRE_SHARED_BUFFER (SHARED_BUFF_HANDLE, IERROR)
   INTEGER SHARED_BUFF_HANDLE
   INTEGER IERROR
```

This routine waits for the shared buffer to be released by the partner. It is a blocking call. When it returns, the buffer associated with the shared buffer handle can be fully accessed. A subsequent acquire without an intervening release is erroneous (this includes an acquire by the source immediately after the make_shared_buffer). An acquire will correctly return only when the partner process performs a release.

MPI_RELEASE_SHARED_BUFFER (sharedBuffHandle, releaseAction)

INOUT	sharedBuffHandle	A shared buffer handle
IN	releaseAction	Indicate whether a produce (MPI_PRODUCE) or consume (MPI_CONSUME) action is associated with the release

```
int MPI_Release_shared_buffer (MPI_Shared_buff *sharedBuffHandle,
          int releaseAction)

MPI_RELEASE_SHARED_BUFFER (SHARED_BUFF_HANDLE, RELEASE_ACTION, IERROR)
   INTEGER SHARED_BUFF_HANDLE
   INTEGER RELEASE_ACTION
   INTEGER IERROR
```

This routine releases a shared buffer so that it can be acquired by the partner process. If the process has only read the buffer associated with the shared buffer handler, it may use the releaseAction of MPI_CONSUME. If the process has written to the buffer associated with the shared buffer handler, it MUST use the releaseAction of MPI_PRODUCE to guarantee that the partner sees the changes.

MPI_FREE_SHARED_BUFFER (sharedBuffHandle)

INOUT	sharedBuffHandle	A shared buffer handle

```
int MPI_Free_shared_buffer (MPI_Shared_buff *sharedBuffHandle)

MPI_FREE_SHARED_BUFFER (SHARED_BUFF_HANDLE, IERROR)
   INTEGER SHARED_BUFF_HANDLE
   INTEGER IERROR
```

This function frees the buffer associated with the shared buffer handler, the handle itself, and satisfies any outstanding communications. This function needs to be viewed as "bi-collective".

Possible Send/Recv Implementation:

```
int MPI_Make_shared_buffer (MPI_Datatype dataType, int, count,
            MPI_Comm comm, int source, int dest,
            void *buffer, MPI_Shared_buff *sharedBuffHandle)
  {
  /* Ok, ok this isn't really C, but you get the idea) */
  sharedBuffHandle = malloc (sizeof(sharedBuffHandle));
  *buffer = malloc (sizeof(dataType) * count);
  sharedBuffHandle->buffer = buffer;
  sharedBuffHandle->count = count;
  sharedBuffHandle->dataType = dataType; /* Dup here? */
  sharedBuffHandle->tag = getSharedBuffTag(); /* Some magic? */
  /* At this point, a robust application might check with it's partner
     to see if the parameters are correct and the malloc worked. But it
     would take too much space (besides if programming text books
     can omit error checking, why can't I? :-) */
  if (myProc == src)
    {
    Recv (comm, sharedBuffHandle->tag, dest, (count=0, buff=NULL,...));
    sharedBuffHandle->partner = dest;
```

```
    }
  else
    {
    sharedBuffHandle->waitHandle = IRECV (comm, sharedBuffHandle->tag,
                                          source, buffer, count,...)
    Send (comm, sharedBuffHandle->tag, source, (count=0, buff=NULL,...));
    sharedBuffHandle->partner = source;
    }
  return;
  }

int MPI_Acquire_shared_buffer (MPI_Shared_buff *sharedBuffHandle)
  {
  wait (sharedBuffHandle->waitHandle);
  }

int MPI_Release_shared_buffer (MPI_Shared_buff *sharedBuffHandle,
          int releaseAction)
  {
  if (releaseAction == MPI_CONSUME)
    count = 0;
  else
    count = sharedBuffHandle->count;
  sharedBuffHandle->waitHandle =
     IRECV (comm, sharedBuffHandle->tag,
                  sharedBuffHandle->partner,
                  sharedBuffHandle->buffer, count,...);
  /* Note that a non-buffering send can be used here */
  /* Note that a "ready" protocol can also be used since we guarantee
     the receive is always posted before the send */
  localHandle = ISSEND (comm, sharedBuffHandle->tag,
                              sharedBuffHandle->partner,
                              sharedBuffHandle->buffer, count,...);
  freeHandle (localHandle);
  }

int MPI_Free_shared_buffer (MPI_Shared_buff *sharedBuffHandle)
  {
  /*
    Either cancel the outstanding receive or (if you don't believe
    cancel works) send to complete the receive.
  */
  /*
    Free all structures
  */
  }
```

Possible Shared Memory Implementation:

```
int MPI_Make_shared_buffer (MPI_Datatype dataType, int, count,
            MPI_Comm comm, int source, int dest,
            void *buffer, MPI_Shared_buff *sharedBuffHandle)
  {
  /* Ok, ok this isn't really C, but you get the idea) */
  sharedBuffHandle = malloc (sizeof(sharedBuffHandle));
  if (myProc == source)
    {
    sharedBuffHandle->semaphore =
              shared_malloc (sizeof(dataType) * count + cacheLineSize);
    }
  sharedBuffHandle->buffer = buffer;
  sharedBuffHandle->count = count;
  sharedBuffHandle->dataType = dataType; /* Dup here? */
  if (myProc == source)
    {
    i = myProc;
    memcpy (sharedBuffHandle->semaphore, &i, sizeof(int));
    sharedBuffHandle->partner = dest;
    Send (comm, sharedTag, dest, sharedBuffHandle->semaphore,
          MPI_Pointer, ...);
    }
  else
    {
    sharedBuffHandle->partner = source;
    Recv (comm, sharedTag, source, sharedBuffHandle->semaphore,
              MPI_Pointer, ...);
    }
  *buffer = ((char *) sharedBuffHandle->semaphore) + cacheLineSize;
  }

int MPI_Acquire_shared_buffer (MPI_Shared_buff *sharedBuffHandle)
  {
  i = myProc - 1;
  while (i != myProc)
    {
    memcpy (&i, sharedBuffHandle->semaphore, sizeof(int));
    /* A smart implementation would release it's timeslice
       after some number of times through this loop */
    }
  }

int MPI_Release_shared_buffer (MPI_Shared_buff *sharedBuffHandle,
          int releaseAction)
  {
  i = sharedBuffHandle->partner;
  memcpy (sharedBuffHandle->semaphore, &i, sizeof(int));
  }
```

```
int MPI_Free_shared_buffer (MPI_Shared_buff *sharedBuffHandle)
  {
  /*
    Free all structures
  */
  }
```

5.3 Cluster attributes

A cluster is given, if the communication inside the members is faster than between the members, and at least one member has more than one MPI processes.

MPI implementations that allow to run applications on a hardware with a hierarchical network (e.g. a cluster of MPPs, or one MPP with a hierarchical set of crossbars) may return the actual *cluster mapping* used for the MPI processes of a communicator by attaching all of the following attributes to communicators:

MPI_CLUSTER_SIZE returns as keyvalue the number of cluster's members.

MPI_CLUSTER_COLOR returns as keyvalue a number between 0 and MPI_CLUSTER_SIZE-1. MPI processes that are part of the same cluster's member return the same value. MPI processes that are part of different cluster's members return different values.

MPI_CLUSTER_CHANGEABLE returns as keyvalue the values
0 if the cluster mapping cannot change until MPI_FINALIZE,
MPI_CLUSTER_CHANGEABLE_BY_TOPO if the cluster mapping can be changed only if the application calls MPI topology functions to any communicator, and
MPI_CLUSTER_CHANGEABLE_ANY if the cluster mapping may change at any time.

Rationale. This interface gives applications the possibility to suit the communication between MPI processes to the cluster properties of the network. This interface is orthogonal to the topology interface, i.e. first an application should specify its topology and the MPI implementation should map the MPI processes as good as possible and then the application should examine the cluster attributes. (*End of rationale.*)

Advice to users. This interface returns no information about the significance of the difference between the communication *inside* and *between* cluster's members. This e.g. can be achieved by small application specific benchmarks as part of the application. the returned color can be used as input to MPI_COMM_SPLIT. Implementations are free to return *no cluster information available on that communicator* by not supporting these attributes or by returning a cluster size of 1. (*End of advice to users.*)

Advice to implementors. High quality implementations on clusters with a large difference in communication efficiency should return these attributes at least on MPI_COMM_WORLD. (*End of advice to implementors.*)

Discussion: This interface has a minimal overhead for all implementations that need not support this interface (only 5 constants in mpi.h, defined in a way that they returne flag=false in MPI_ATTR_GET). And it can give all needed information to applications on clusters.

It may be that the application must communicate the colors because it needs to have the full cluster information in each MPI process, and this may occur although the MPI implementation has this information available inside the local database in each MPI process. But this is the result of a compromise to have a very simple interface without an additional inquiry function and therefore a minimal overhead in implementations that do not support this interface.

It was noticed that during the topology discussions in MPI-1, we talked about similar functionality (though not for clusters). The final conclusion was if you give enough information it isn't portable and if you make it portable then it isn't sufficently precise to be truely meaningful.

But now this is an approach for a new problem, not discussed in MPI-1. This proposal supports systems from clusters of 1- to n-processor PCs until clusters of large SVPs, and also hierarchical clusters. This approach does not give all information that is needed, but it gives the topology information and all the rest the aplication can achieve e.g. by little benchmark tests.

5.4 Continuable Errors

I sent a previous message on MPI_IS_CONTINUABLE, but I realise that it may seem a little over the top for general errors. However, here is some rationale why it is important for all errors, especially in the context of the new functions being added in MPI-2.

The first aspect is that the continuability after errors is generally NOT associated with an error code, but with the communicator state. For example, consider a rejection "insufficient store" or "too many outstanding requests".

It will very often be possible to retry after this, provided that it was due to a temporary logjam. But it may not be, especially if the failure is due to a key server not responding. And identical errors that occurs under different circumstances may need different recovery strategies.

This will very often differ according to the communicator; for example, one process may be on a system that uses a fixed request pool (and so such a failure is fatal) but another may be on one which has a "busy - try later" return.

Furthermore, some such errors will affect only the individual operation, others will affect only the communicator and others will indicate that the local MPI environment is knotted.

So, that is why I suggested a function that would inquire the state of a communicator, and that the return should not be a simple yes or no, but an indication of what has to be done to continue.

Incidentally, speaking as a run-time system implementor, the function doesn't ask for any information that won't be held internally. All I am suggesting is that the programmer be told how the MPI implementation regards the current state.

The second aspect is that of context-dependent messages. The standard joke about Unix is that there are three: "can't", "shan't" and "didn't". I currently have a problem where there is a failure SOMEWHERE in the IP or UDP stack, but have no way of finding out anything more.

As dynamic processes provide PVM-like facilities, there will be an increasing need to provide an indication of WHERE the failure occurred. If you have an intercommunicator covering 6 vendors' systems, the failure "insufficient store" isn't exactly helpful.

MPI needs to provide some way that the implementation can pass arbitrary text back to the programmer, so that it can be written out and taken to the support staff or MPI implementors. There is clearly no way that MPI can specify what the information will say, but any decent implementation will at least indicate which processes were involved!

And please note that I am thinking as an implementor, because one of the main purposes of this information is to enable problems to be reported in a useful way. Error reports "I have got a request rejected message" aren't exactly helpful.

I have also been thinking about the interface, and believe that the call to clear the errors is unnecessary and can be dropped (it was there because I was thinking in C terms) and that the error indication and messages should be requested separately. So here is a minimal syntax:

```
MPI_ERROR_COMM_STATE (comm, code, severity, scope)
    IN      comm        Communicator
    OUT     code        Error code associated with the communicator
    OUT     severity    Severity of the error state
    OUT     scope       Scope of the error state

MPI_ERR_IGNORABLE          No special action is needed
MPI_ERR_RECOVERABLE        Specific action is needed
MPI_ERR_RESTARTABLE        All outstanding operations must be abandoned
MPI_ERR_CORRUPTED          This is beyond hope

MPI_ERR_ACTION             The failure affects only the operation
MPI_ERR_LOCAL              The failure affects only the local processor
MPI_ERR_GLOBAL             The failure affects the whole communicator
MPI_ERR_UNIVERSAL          The failure is not localised


MPI_ERROR_COMM_MESSAGE (comm, message, length)
    IN      comm        Communicator
    OUT     message     Context-dependent messages
    OUT     length      Length of messages returned
```

In the Forum discussion, it was decided to add `FILE` and `WIN` to `COMM` as objects that could be inquired about. But then the section was voted to the JOD.

Chapter 6

Towards a Full Fortran 90 Interface

This chapter contains ideas for elements of a "Full" Fortran 90 interface that tries to address the problems of the F77-based interface. Some of these ideas are compatible, and some are mutually exclusive.

6.1 A different way of handling buffer arguments

1. Define a constant MPI_AINT_KIND. An integer of this kind is enough to hold an address. This integer is conceptually similar to MPI_Aint and will be referred to in the rest of this message as an MPI_Aint.
2. The F90 version of F77 functions with choice argument take an MPI_Aint instead of the choice argument.
3. The MPI_ADDRESS function in Fortran 90 now returns an AINT.
4. Add the intrinsic function MPI_F90_DESCRIBE_OBJECT as described below, or one or more variants.
5. For backward compatibility with current MPI bindings, the following would work, but would be subject to all the caveats in the current F90 discussion:

   ```
   real a(100) ! or real a(:)
   MPI_ADDRESS(a, a_address)
   MPI_ISEND(a_address, 100, MPI_REAL, ...)
   ```

6. For convenience, we add a new set of functions:

   ```
   MPI_SEND_F90
   MPI_ISEND_F90
   MPI_BSEND_F90
   etc.
   ```

6.1.1 MPI_F90_DESCRIBE_OBJECT and relatives

```
MPI_F90_DESCRIBE_OBJECT(IN object, OUT MPI_Datatype)
```

This is a generic function that can be used with any Fortran 90 object, including array sections, derived types, etc. It creates a committed MPI datatype that describes the Fortran 90 object. The datatype must be used in conjunction with a buf argument of MPI_BOTTOM.

Discussion: Need to define the type signature of the new datatype, for matching rules.

It could be used in the following way

```
MPI_F90_DESCRIBE_OBJECT(a(10:1:-2), newtype)
MPI_ISEND(MPI_BOTTOM, 1, newtype, ...)
```

MPI_F90_DESCRIBE_OBJECT is an intrinsic function. Moreover, it does not have copy-in/copy-out semantics. The address and type information recorded in the output datatype describes the data in the original object, not the data in a temporary copy. It is not possible to implement this routine portably in Fortran 90. It must be closely integrated with the compiler. It is expected that this routine and other MPI routines would be part of an intrinsic module.

Rationale. In many applications, you send the same data many times. Creating the datatype once means you only pay the overhead once, instead of at each send call. (*End of rationale.*)

We could also consider a routine for which the created datatype is relative:

```
MPI_F90_DESCRIBE_OBJECT(a(10:1:-2), newtype)
MPI_ADDRESS(a, a_address)
MPI_ISEND(a_address, 1, newtype, ...)
```

[I don't propose this because I'm not sure we can say clearly when/if this works for a second array b, or even if you were to try:

```
MPI_ADDRESS(a(5), a5_address) ...
```

] In order for this to be guaranteed to work, it is probably necessary to make MPI_ADDRESS be an intrinsic function as well. It might be possible to allow both approaches through two calls:

```
MPI_F90_DESCRIBE_OBJECT(...)
MPI_F90_DESCRIBE_OBJECT_RELATIVE(...)
```

The following approach suggested in subcommittee discussion might provide a cleaner separation between MPI and the compiler.

```
MPI_F90_GET_DOPEVECTOR(a(10:2:-2), OUT dope)
MPI_IRECV(dope, 1, MPI_REAL)
```

This needs to be fleshed out.

6.2 Using a Fortran 90 - ANSI C interface

Some work is currently being done on an interface between Fortran 90 and ANSI C. MPI may be able to use this.

6.3 Typed functions

Here are some possibilities for using typed functions:

1. Require overloaded interfaces for all functions with choice arguments.
2. Add typed functions for "the most important" basic types, for instance MPI_SEND_REAL.
3. Add a "typed version" of all functions with choice arguments, e.g. MPI_SEND_T.
4. Have two versions of the MODULE MPI – one "light" version and one with full generic interfaces. Same name of the module, but mpi.mod files stored in different MODPATHs. Users can switch between safe+debug and fast mode by changing only one Makefile line: the search path for module files (library may contain code for both versions in one ar file...).

6.4 The possibility of optional arguments

For any routines with an explicit interface, it would be possible to have optional arguments. Clear candidates are STATUS and IERR arguments. Datatype arguments could also be omitted and the data derived from the type of buf.

This proposal could work in conjunction with one of the typed-function proposals above (but not with implicit F77-style interfaces, for which optional arguments are not allowed).

6.5 Derived types for MPI handles

Type-checking enabled by explicit interfaces might not be too useful because almost all of the arguments to MPI routines are integers in Fortran. It would be possible (losing all compatibility with Fortran 77) to introduce derived types for MPI objects, substantially improving the usefulness of type checking (though it is unclear that there would be any other benefits).

Chapter 7

Split Collective Communication

In some applications, better performance can be achieved by separating the initiation and completion of a collective operation. For example, in some numerical applications, better performance can be achieved by overlapping other work (both computation and communication) with an MPI_Allreduce. At the same time, the full generality of non-blocking collective operations, considered at length by the MPI Forum, is both not needed by some applications, and can be a burden for some MPI implementations. Instead, a very restricted set of collective operations that can be started and completed with separate MPI calls are proposed. These are called "split" collective communication routines.[1] For each of these operations, there is a separate begin and end routine. For example, for the split allreduce operation, the calls are

```
MPI_Allreduce_begin( sbuf, rbuf, count, datatype, op, comm );
...
other work and/or MPI calls
...
MPI_Allreduce_end( rbuf, comm );
```

The reason for the `rbuf` in the end call is described below.

These routines allow applications to separate the process of data transmission from synchronization, much as the non-blocking point-to-point operations (such as MPI_Isend and MPI_Irecv) do for point-to-point communication. The split versions can provide important performance advantages even on platforms with fast collective operations by allowing applications to avoid unnecessary synchronizations. For example, an MPI implementation with some shared memory may choose to implement these operations by using a small amount of shared memory attached to each communicator and a special algorithm for each split operation. An MPI implementation without shared memory but with a multicast or simple data-collection capability may use these features. Because the split operations are distinct from the other collective operations, the split version can be optimized, and there is no impact on the implementation or performance of the other collective operations.

[1] These were originally called two-phase collective communications. However, since two-phase has a different meaning in parallel I/O, and since the MPI-2 parallel I/O chapter adopted a varient of this approach and termed it split communication, this section adopts the term split communication.

7.0.1 General rules for all split collective communication

- Only one split collective communication may be active on any particular communicator and MPI process at any time. In other words, on any MPI process, each communicator may have a most one active split operation at any time. The inclusion of process here is intend to make clear that "at any time" applies to each process separately.

- Begin calls are collective over the communicator and follow the ordering rules for collective calls.

- End calls are collective over the communicator and follow the ordering rules for collective calls. End calls match the preceeding begin call for the same collective operation. When and "end" call is made, exactly one unmatched "begin" call for the same operation must preceed it.

- An implementation is free to implement any split collective operation using the corresponding blocking collective routine when either the begin (e.g., MPI_Allreduce_begin) or end call (e.g., MPI_Allreduce_end) is issued. The begin call is provided to allow the user and MPI implementation to optimize the collective operation.

- Split collective operations do not match the corresponding regular collective operation. For example, an MPI_Allreduce and an MPI_Allreduce_begin/end do not match.

- Split routines that have a destination buffer specify that buffer in both the begin and end routines. By specifying the buffer that receives data in the end routine, we can avoid many (though not all) of the problems described in Section 10.2.2 (A Problem with Register Optimization).

- An MPI communicator may only have one collective operation in progress. In other words, if a split collective operation is started on a communicator, no other collective operation may be begun on that communicator. For example, the following is illegal:

```
MPI_Allreduce_begin( ..., comm );
...
MPI_Barrier( comm );
...
MPI_Allreduce_end( rbuf, comm );
```

- In a multithreaded implementation, any split collective begin and end operation called by a process must be called from the same thread. This restriction is made to simplify the implementation in the multithreaded case. (Note that we have already disallowed having two threads start a split operation on the same communicator since only one split operation can be active on a communicator at any time.)

Specific routines

MPI_ALLREDUCE_BEGIN(sendbuf, recvbuf, count, datatype, op, comm)

IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	count	number of elements in send buffer (integer)
IN	datatype	data type of elements of send buffer (handle)
IN	op	operation (handle)
IN	comm	communicator (handle)

```
int MPI_Allreduce_begin(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

```
MPI_ALLREDUCE_BEGIN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, IERROR
```

MPI_ALLREDUCE_END(recvbuf, comm)

OUT	recvbuf	starting address of receive buffer (choice)
IN	comm	communicator (handle)

```
int MPI_Allreduce_end( void* recvbuf, MPI_Comm comm)
```

```
MPI_ALLREDUCE_END(RECVBUF, COMM, IERROR)
    <type> RECVBUF(*)
    INTEGER COMM, IERROR
```

The arguments have the same meaning as for MPI_Allreduce.

MPI_ALLREDUCE_BEGIN begins a split operation that, when completed with MPI_ALLREDUCE_END, produces the result as defined for MPI_ALLREDUCE.

> *Advice to users.* This is deliberately weaker than "same as MPI_ALLREDUCE," to allow for differing orders of operations which may produce different results. For example, when performing the common operation of MPI_SUM on MPI_DOUBLE data, the exact return value depends on the order of operation, since floating point arithmetic is not associative. (*End of advice to users.*)

MPI_BCAST_BEGIN(buffer, count, datatype, root, comm)

INOUT	buffer	starting address of buffer (choice)
IN	count	number of entries in buffer (integer)
IN	datatype	data type of buffer (handle)
IN	root	rank of broadcast root (integer)
IN	comm	communicator (handle)

```
int MPI_Bcast_begin(void* buffer, int count, MPI_Datatype datatype, int
              root, MPI_Comm comm )
```

```
MPI_BCAST_BEGIN(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
    <type> BUFFER(*)
    INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
```

MPI_BCAST_END(buffer, comm)

INOUT	buffer	starting address of buffer (choice)
IN	comm	communicator (handle)

```
int MPI_Bcast_end(void* buffer, MPI_Comm comm )
```

```
MPI_BCAST_END(BUFFER, COMM, IERROR)
    <type> BUFFER(*)
    INTEGER COMM, IERROR
```

The arguments have the same meaning as for MPI_Bcast.

MPI_BCAST_BEGIN begins a split operation that, when completed with MPI_BCAST_END, produces the result as defined for MPI_BCAST.

MPI_BARRIER_BEGIN(comm)

IN	comm	communicator (handle)

```
int MPI_Barrier_begin( MPI_Comm comm )
```

```
MPI_BARRIER_BEGIN(COMM, IERROR)
    <type> BUFFER(*)
    COMM, IERROR
```

MPI_BARRIER_END(comm)

IN	comm	communicator (handle)

```
int MPI_Barrier_end(MPI_Comm comm )
```

```
MPI_BARRIER_END(COMM, IERROR)
```

```
INTEGER COMM, IERROR
```

The arguments have the same meaning as for MPI_Barrier.

MPI_BARRIER_BEGIN begins a split operation that, when completed with MPI_BARRIER_END, produces the result as defined for MPI_BARRIER.

Advice to users. One use of the split barrier is in double buffering schemes, where the barrier is used to ensure that the buffers may be swapped. (*End of advice to users.*)

MPI_WIN_FENCE_BEGIN(assert, win)

IN	assert	program assertion (integer)
IN	win	window object (handle)

```
int MPI_Win_fence_begin( int assert, MPI_Win win)
```

```
MPI_WIN_FENCE_BEGIN(ASSERT, WIN, IERROR)
    INTEGER ASSERT, WIN, IERROR
```

```
MPI::Win::Fence_begin(int assert)
```

MPI_WIN_FENCE_END(win)

IN	win	window object (handle)

```
int MPI_Win_fence_end(MPI_Win win)
```

```
MPI_WIN_FENCE_END( WIN, IERROR)
    INTEGER WIN, IERROR
```

```
MPI::Win::Fence_end()
```

The arguments have the same meaning as for MPI_WIN_FENCE.

MPI_WIN_FENCE_BEGIN begins a split operation that, when completed with MPI_WIN_FENCE_END, produces the result as defined for MPI_WIN_FENCE.

For the MPI_WIN_FENCE_BEGIN and MPI_WIN_FENCE_END routines, the rules for split collective communication are modified to apply to a window (MPI_Win) instead of a communicator (MPI_Comm).

7.0.2 Examples using MPI_ALLREDUCE_BEGIN

The following example shows how a split collective operation is used to avoid unnecessary synchronization in a

```
      call MPI_ALLREDUCE_BEGIN(teilresl1, resl1, 1, MPI_REAL, MPI_SUM,
     .                           mpi_comm_world_a, info)
      call MPI_ALLREDUCE_BEGIN(teilresl0, resl0, 1, MPI_REAL, MPI_MAX,
     .                           mpi_comm_world_b, info)
```

```
c...solve the equation system
       call parallelsolver( rm000, rmf00, rmb00, rm0f0, rm0b0, rm00f,
     .                      rm00b, dq, rhs, nsubit, nhofhausit )

c...computation of the CFL-Number with resl1
       call MPI_ALLREDUCE_END(mpi_comm_world_a, resl1)
       ...
c...output of all residuals
       call MPI_ALLREDUCE_END(mpi_comm_world_b, resl0)
       ...
```

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.

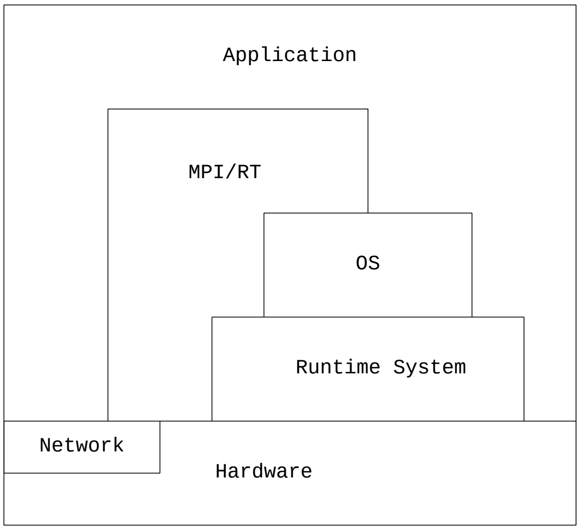

Figure 8.1: Model of Software Abstraction Layers for MPI/RT.

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.
- 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 parameter when global system clocks are synchronized. (*End of advice to implementors.*)

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.

8.5.4 MPI/RT Clock Attributes

In order for MPI/RT to allow portable applications across a wide variety of parallel real-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.

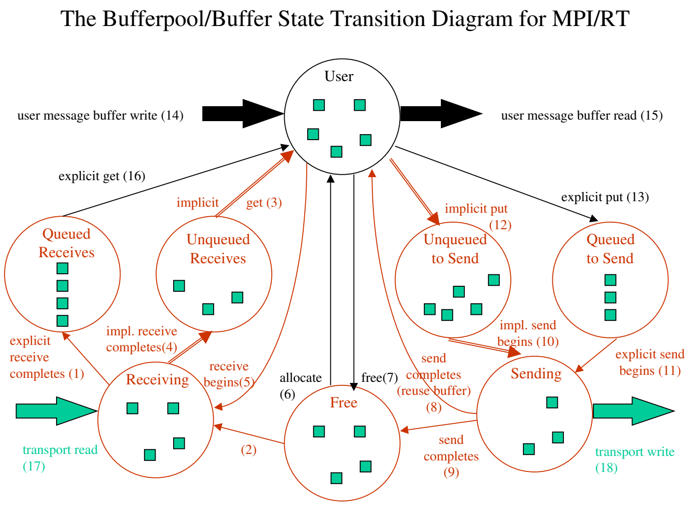

Figure 8.2: Buffer Pool State Transition Diagram. This diagram represents the semantic behavior of the buffer pool and the operations, but does not specify the implementation.

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
```

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.

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)
IN	comm	communicator (handle)
OUT	request	request object (persistent request)

```
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
```

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)

```
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
```

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.

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 specification (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 associated with that request have been called. Even if the process or another handler 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 .

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 provides the interface for control flow. The events can be both persistent and one time 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.

MPI/RT makes three types of events available: system events, communication events 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.

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 two relevant operations: MPI_SIGNAL and MPI_MONITOR that appear in JOD. However, only the 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 "generic" application activities.

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.

MPI/RT is responsible to deliver events and to trigger (start or stop) an activity if it 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

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.

MPI/RT Monitoring Management - Block of Code

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)

```
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
```

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
```

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.

MPIRT_MONITOR_RESET(info, monitor_handle, error)

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

```
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
```

MPI/RT Monitoring Management - Requests

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)

```
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
```

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
```

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)

```
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
```

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)

```
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
```

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
```

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)

```
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
```

8.14.3 MPI/RT Monitoring Control

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.

MPI/RT Start and Stop Monitoring

The ability to start and stop monitoring is accomplished with the following basic functions:

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
```

MPIRT_MONITOR_STOP(monitor_handle)

IN	monitor_handle	monitor handle (handle)

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

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

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.

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.

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.