Request Progression Interface (RPI) System Services Interface (SSI) Modules for LAM/MPI API Version 1.0.0 / SSI Version 1.0.0

Jeffrey M. Squyres
Brian Barrett
Andrew Lumsdaine
http://www.lam-mpi.org/

Open Systems Laboratory
Pervasive Technologies Labs
Indiana University
CS TR579

August 4, 2003

Contents

List of Figures

1 Overview

Before reading this document, readers are strongly encouraged to read the general LAM/MPI System Services Interface (SSI) overview ([2]). This document uses the terminology and structure defined in that document.

The rpi SSI type is used to perform point-to-point message passing between MPI processes. It accepts MPI messages from the MPI layer and passes them to the destination MPI process (including, potentially, itself). It also accepts messages from peer MPI processes and passes them up to the MPI layer when a matching receive request is found.

"rpi" stands for "Request Progression Interface." The rpi API was initially designed to follow the life cycle of point-to-point MPI requests. It has since been augmented to include other utility functions (such as "special" memory support, checkpoint/restart support, etc.), but the core of its functionality is still centered around the natural progression of an MPI point-to-point request.

Previous versions of the LAM/MPI rpi design made the distinction between the "lamd" and "C2C" rpi implementations. Since the entire rpi mechanism now falls under SSI, this distinction is no longer necessary – all rpi modules are effectively equal; the lamd rpi module is just the same as any other rpi module. One rpi module is chosen to be used at run time by the normal SSI module selection mechanism.

1.1 General Scheme

The rpi is the underlying mechanism that the MPI library uses to pass messages between MPI processes. The MPI library does this by passing MPI requests to the rpi module. The rpi module progresses these requests through their life-cycle. If there is a message associated with the request, it effectively causes the rpi to [attempt to] send or receive it, as appropriate.

Specifically, the MPI layer does not know (or care) how messages move from process A to process B – all it knows is that a request was created, progressed, and completed. This abstraction barrier gives the rpi module complete control over how it effects sending and receiving.

Currently, only one rpi module can be used within an MPI process, and all MPI processes must select the same module. Future versions of the rpi API will allow for multiple rpi modules to be used simultaneously, allowing the "best" rpi module to be used to communicate with a given MPI process peer (see Section 5.1 for more information). This behavior is called "multi-rpi", and is referred to in several places throughout this document because it will change certain current rpi behaviors.

An rpi module is intended to be targeted towards communications architectures and provide high performance direct communication between MPI processes. Although several helper functions are provided by the rpi SSI kind, each rpi module is responsible for all aspects of moving messages from source to destination. Put differently, each rpi SSI module has complete freedom of implementation in how to transport messages from source to destination.

The tasks that must be performed in the rpi module are essentially the initialization of the data transport connections between processes, transporting of messages across the connections, message synchronization, and cleanup. State information required by the MPI layer must be correctly maintained.

Higher level operations such as buffer packing/unpacking, handling of buffers for buffered sends, and message data conversion are handled by the MPI layer.

The LAM/MPI library maintains various data structures. The most important of these, as far as the rpi module is concerned, are the request list and the process list. The MPI layer of LAM/MPI handles the high-level initialization and manipulation of these structures. An rpi module can attach information to entries in these lists via handles.

For illustration, this document will often refer to the `tcp` rpi module provided with the LAM distribution. The `tcp` module uses Internet domain TCP sockets as the communication subsystem. It is implemented in the `share/ssi/rpi/tcp` directory.

This document should be read in conjunction with the header files `share/ssi/rpi/include/rpisys.h` and `share/include/mpisys.h`, in which some of the data structures referred to are defined.

1.2 The Request Lifecycle

As the name "Request Progression Interface" implies, the API described herein follows the life of an MPI request. All other details (such as how the bytes of a messages are moved from source to destination) are *not* within the scope of this API.

A request's life follows this cycle:

- **building**: storage is allocated and initialized with request data such as datatype, tag, etc. The request is placed in the init (LAM_RQSINIT) state. It is not to be progressed just yet, and is thus not linked into the request list.
- **starting**: the request is now made a candidate for progression and is linked into the request list.

  It is not necessary at this stage for any data transfer to be done by the rpi module, but this is not precluded. All that is required is that the request's progression state be correctly set. See Sections 4.8 and 4.9 for more details. Depending on the rpi module and the circumstances, the request will be put into the start (LAM_RQSSTART), active (LAM_RQSACTIVE), or done (LAM_RQSDONE) state.
- **progression**: the request is progressed in stages to the done state. The request is moved from the start state to the active state as soon as any data is transferred. It is is moved from the active state to the done state once all data is transferred and all required acknowledgments have been received or sent.
- **completion**: when completed, the request is either reset to the init state ready for restarting (if persistent) or destroyed (if non-persistent).

1.3 Forward Compatibility

Changes are being planned for this API. Although the final designs and implementation decisions have not yet been made, current rpi authors can make reasonable assumptions and protect themselves for forward compatibility. These assumptions are listed throughout this document and summarized in Section 5.

Three major changes are planned for the future of the LAM/MPI rpi API. The changes are listed below; each has a distinctive mark that is used in the rest of the document when referring to assumptions and potential effects that each change will cause.

- "Multi-rpi" $^{(MRPI)}$: Allowing more than one rpi module to be selected in an MPI process, effectively allowing multi-device point-to-point MPI message passing.
- Support for MPI_THREAD_MULTIPLE $^{(MT)}$: Allowing user MPI applications to have multiple threads simultaneously active in the MPI library.
- IMPI $^{(IMPI)}$: Interoperable MPI (IMPI) support is currently embedded within the MPI library, and has some inelegant effects on the rpi data structures. These will eventually be removed when $^{(MRPI)}$ becomes a reality and the IMPI support can become its own rpi module.

Additionally, one-sided MPI-2 functionality is currently implemented on top of point-to-point functionality. It is conceivable that one-sided functionality will turn into its own SSI type someday. As such, some of the hooks and extra functionality currently in the rpi may disappear. This should not present a problem for rpi authors; these hooks are not necessary for normal rpi usage. These items are marked with $^{(OSD)}$.

2 Threading

Although LAM/MPI does not yet *require* threads in order to be built properly, the default is to enable thread support. Hence, it is permissible for rpi modules to leave "progress threads" running to make progress on message passing even while the user program is not in MPI functions.

Note, however, that the MPI layer makes no guarantees and provides no protection for the re-entrance of top-level rpi functions. Although LAM does not currently support multiple threads running in the MPI layer, a high-quality rpi modules should assume this in order to be forward compatible (particularly if "progress threads" are used). Specifically: someday LAM/MPI *will* allow multiple threads in the MPI layer, any one (or more) of which may invoke an rpi function at any time. The only guarantee likely to be provided by MPI is that multiple threads will not attempt to make progress on the same MPI request at the same time (such condition is an error as defined by the MPI standard, and an rpi module's behavior is therefore undefined). An rpi module author can infer from this that even though the same rpi function may be simultaneously invoked multiple times by different threads, the argument list will be different.

It is expected that this section will be much more detailed in future rpi designs.$^{(MT)}$

3 Services Provided by the rpi SSI

Several services are provided by the rpi SSI that are available to all rpi modules.

3.1 Header Files

The following header files must be included (in order) in all module source files that want to use any of the common rpi SSI services described in this document:

```
#include <lam-ssi.h>
#include <lam-ssi-rpi.h>
```

Both of these files are included in the same location: `share/ssi/include`. If using GNU Automake and the `top_lam_srcdir` macro as recommended in [2], the following can be added to the `AM_CPPFLAGS` macro (remember that `LAM_BUILDING` must be defined to be 1):

```
AM_CPPFLAGS = \
              -I$(top_lam_builddir)/share/include \
              -I$(top_lam_srcdir)/share/include \
              -I$(top_lam_srcdir)/share/ssi/include
```

3.2 Module Selection Mechanism

The rpi SSI has a single scope over the life of an MPI process; a module is selected and initialized during MPI_INIT and later finalized during MPI_FINALIZE. The rpi selection process is divided into multiple parts:

1. The module's open function (if it exists). This is invoked exactly once when an MPI process is initialized. If this function exists and returns 0, the module will be ignored for the rest of the process.

2. The module's `lsr_query()` API function (see Section 4.3). This function is also invoked exactly once when the application is initialized and is used to query the module to find out its priority and what levels of thread support it provides. If this function returns a value other than 1, it will be closed and ignored for the rest of the process.

3. The rpi framework will then make the final module selection based upon the thread support levels and priority, and invoke `lsr_init` (see Section 4.4) on the selected module. All other modules that were queried will be closed and ignored for the rest of the process.

3.3 Types

Some types are used in different API calls and throughout the rpi SSI.

3.3.1 Process Location: `struct _gps`

This type is used to identify the location of a process in the LAM universe. It is typically used to fill in attributes required for the `nsend()` and `nrecv()` function calls (see their corresponding manual pages, section 2). See Figure 1.

```
struct _gps {
  int4 gps_node;
  int4 gps_pid;
  int4 gps_idx;
  int4 gps_grank;
};
```

Figure 1: `struct _gps`: Process location type.

The individual elements are:

- `gps_node`: The node ID in the LAM universe where the process is running. This will be an integer in $[0, N)$, where N is the number of nodes in the LAM universe.
- `gps_pid`: The POSIX PID of the process that invoked MPI_INIT.[1]
- `gps_idx`: The index of the process in the local LAM daemon's process table.
- `gps_grank`: The "global rank" of the process. This is the integer rank of this process in MPI_COMM_WORLD.

3.3.2 MPI Processes: `struct _proc`

This type is used to describe MPI processes. Each process in LAM/MPI maintains a linked list of all the processes that it knows about and can communicate with (including itself). See Figure 2.

[1] Note that because of Linux kernel 2.x POSIX thread semantics, this may or may not be the PID of the main thread. However, this *is* the PID of the process (thread) that invoked MPI_INIT (and therefore `kinit()`), and the PID that will be returned by `lam_getpid()`.

```
struct _proc {
  struct _gps p_gps;
  int p_ger_nsnd;
  int p_mode;
  int p_refcount;
  int p_num_buf_env;

  struct lam_ssi_rpi_proc *p_rpi;
};
```

Figure 2: `struct _proc`: MPI process information.

It is important to recognize that each of these elements are set from the perspective of the process on which the instance resides. An individual `_proc` instance indicates what process A knows about process B. Hence, it is possible for multiple processes to have different values for the individual elements, even if they are referring to the same target process.

The individual elements are:

- `p_gps`: The GPS for the process. See Section 3.3.1.
- `p_get_nsnd`: The number of messages sent to this process. Used for Guaranteed Envelope Resources (GER) purposes, for rpi modules that support them.
- `p_mode`: A bit-mapped flag indicating various things about the process. Valid flags are:
  - LAM_PFLAG: Generic flag used for marking. Care needs to be taken to ensure that two portions of code are not using/modifying LAM_PFLAG marking simultaneously.
  - LAM_PDEAD: Set if the process has died.
  - LAM_PRPIINIT: Set if the rpi module has been initialized on this process.
  - LAM_PCLIENT: Set if the process is in the "client" MPI_COMM_WORLD (e.g., if the process was spawned or connected to after MPI_INIT) in this process' original MPI_COMM_WORLD.
  - LAM_PHOMOG: Set if the process has the same endian as this process.
  - LAM_PRPICONNECT: Set if the rpi module has connected to this process or not.

  Note the distinction between whether the rpi module has initialized a process and whether it has connected to it. A process typically only needs to be initialized once, but may be connected multiple times throughout its life (e.g., checkpointing may force disconnects, or an rpi module may choose to only keep processes connected when they are actively communicating).

  The rpi module must leave the lower 8 bits of `p_mode` alone – they are reserved for the MPI layer.
- `p_refcout`: The reference count for this process. Every time a communicator or group includes this process, its reference count is incremented (and vice versa) by the MPI layer.
- `p_num_buf_env`: Number of buffered envelopes.

- `p_rpi`: This member is of type `(lam_ssi_rpi_proc *)`, which must be defined by each rpi module (but not necessarily before including the `rpisys.h` file, since the compiler can defer the exact definition of a pointer type until after its use). It is a mechanism for the rpi module to attach rpi-specific state information to a `_proc`. This information is typically state information for the connection to/from the process.

  For example, the `tcp` module stores (among others things):

  - File descriptor of the socket connection to the remote process
  - Pointers to requests (if any) that are currently being read from the socket
  - Pointers to requests (if any) that are currently being written to the socket

3.3.3 MPI Requests: `struct _req`

For each process, the LAM/MPI library maintains a linked list of all the requests that need to be progressed. The MPI layer keeps this progression list in order and also removes requests upon completion. rpi authors can thread other lists through the progression list via rpi-specific data. Several of the rpi functions deal with creating requests and moving them along to completion. See below.

A request is represented by a structure of type `(struct _req)`. The type `MPI_Request` is actually a typedef: `(struct _req *)`. Hence, when top-level MPI functions pass an `MPI_Request`, they are actually pointing to an underlying `struct _req`. This bears importance, particularly since it implies that the MPI layer must allocate and free individual `struct _req` instances (see the LAM_RQFDYNREQ mark, below). See Figure 3.

Each request is typically the result of a call to some kind of MPI send or receive function. The individual members are:

- `rq_name`: String name of the request. This is typically only used for debugging purposes. If not NULL, it needs to point to storage on the heap, and will be freed when the request is destroyed.
- `rq_type`: Flag indicating the type of action specified by this request. Valid values are:
  - LAM_RQISEND: Normal mode send
  - LAM_RQIBSEND: Buffered mode send
  - LAM_RQISSEND: Synchronous mode send
  - LAM_RQIRSEND: Ready mode send
  - LAM_RQIRECV: Receive
  - LAM_RQIPROBE: Probe
  - LAM_RQIFAKE: “Fake” request used to indicate a non-blocking buffered send. The rpi module should never see a request of this type – the MPI layer should handle it internally. This flag value is only mentioned here for the sake of completeness.

  Note that whether the request is blocking is not indicated by `rq_type` – it is indicated by `rq_flags`, described below.
- `rq_state`: The state of this request (see Section 1.2, page 6)). Valid values are:
  - LAM_RQSINIT: Init state (not active). No communication allowed.

```
typedef struct _req *MPI_Request;
struct _req {
  char *rq_name;
  int rq_type;
  int rq_state;
  int rq_marks;
  int rq_flags;

  char *rq_packbuf;
  int rq_packsize;

  int rq_count;
  void *rq_buf;
  MPI_Datatype rq_dtype;
  int rq_rank;
  int rq_tag;
  MPI_Comm rq_comm;

  int rq_cid;
  int rq_func;
  int rq_seq;
  int rq_f77handle;
  MPI_Status rq_status;
  struct _bsndhdr *rq_bsend;
  struct _proc *rq_proc;
  struct _req *rq_next;

  void *rq_extra;
  int (*rq_hdlr)();
  MPI_Request rq_shadow;

  struct lam_ssi_rpi_req *rq_rpi;
};
```

Figure 3: `struct _req`: Underlying structure for `MPI_Request`.

  - LAM_RQSSTART: Start state (active, but not yet done). Communication allowed, but not required.
  - LAM_RQSACTIVE: Active state (active, but not yet done). Communication allowed, but only required if the destination process is not this process.
  - LAM_RQSDONE: Done state (not active). No communication allowed.

  It is critical for rpi modules to update this member properly. For example, MPI semantics require that MPI_REQUEST_FREE may be invoked on a request before the communication associated with it has completed. In such a case, the LAM examines the state of the `rq_state` member to see if it is safe to actually destroy the request or not; the request will only be destroyed if the state is LAM_RQSINIT or LAM_RQSDONE. Otherwise, the request will be marked as an orphan (see `rq_flags`, below) and LAM will free it when it is actually finished.

- `rq_marks`: Bit-mapped persistent flags on a request. These flags will endure through the entire life of a request, regardless of its state (e.g., flags that only need to be set once on persistent requests). The rpi module should not modify these values. Valid values are:

  - LAM_RQFPERSIST: This request is persistent.
  - LAM_RQFDYNBUF: The buffer associated with this request is dynamic and will be automatically freed when the request is destroyed.
  - LAM_RQFDYNREQ: The request itself is dynamic and will be freed when the request is destroyed.
  - LAM_RQFSOURCE: A source request (i.e., indicates direction of message transfer).
  - LAM_RQFDEST: A destination request (i.e., indicates direction of message transfer).
  - LAM_RQFBLKTYPE: Indicates that the request is a blocking request.
  - LAM_RQFOSORIG: Origin of a one-sided request.
  - LAM_RQFOSTARG: Target of a one-sided request.
  - LAM_RQFMAND: A mandatory request. This mark is maintained by the MPI layer, and is really only intended for IMPI communications, and will (hopefully) someday be removed.$^{(IMPI)}$

- `rq_flags`: Bit-mapped active request flags. These will be reset for each iteration through the start state. The rpi module should only ever modify the LAM_RQFTRUNC value; all other values should not be modified by the rpimodule. Valid values are:

  - LAM_RQFCANCEL: The request has been canceled.
  - LAM_RQFBLOCK: The request is blocking.
  - LAM_RQFTRUNC: The request was (or is about to be) truncated. If set, this will cause an MPI_ERR_TRUNCATE error in the MPI layer.
  - LAM_RQFHDLDONE: The handler for this request has already been invoked.
  - LAM_RQFORPHAN: This request is an orphan and needs to be automatically freed when it is done.
  - LAM_RQFCHAR: Obsolete.
  - LAM_RQFINT: Obsolete.

  - LAM_RQFFLOAT: Obsolete.
  - LAM_RQFDOUBLE: Obsolete.
  - LAM_RQFSHORT: Obsolete.
  - LAM_RQFMARK: Arbitrary marking for requests. Care should be taken to ensure that two independent sections of code don't attempt to use/modify the LAM_RQFMARK flag at the same time.
  - LAM_RQFACKDONE: The ACK for this request has completed (e.g., for rendevouz protocols). This flag is only for IMPI support, and should not be used by rpi modules.$^{(IMPI)}$

- `rq_packbuf`: Pointer to start of contiguous buffer of message data to be sent or area where message data is to be received. Depending on the MPI datatype of data to be sent/received, this may or may not be the same as `rq_buf`, which is a pointer to the buffer given by the user. The MPI layer handles packing and unpacking of this buffer.
- `rq_packsize`: The size of the data to be sent/received in bytes. This is set by the MPI layer. This is how much message data the rpi module must send/receive for the request.
- `rq_count`: Parameter from the original MPI function call.
- `rq_buf`: Parameter from the original MPI function call.
- `rq_dtype`: Parameter from the original MPI function call.
- `rq_rank`: Parameter from the original MPI function call.
- `rq_tag`: Parameter from the original MPI function call.
- `rq_comm`: Parameter from the original MPI function call.
- `rq_cid`: The context ID to use in the message envelope. It corresponds to the communicator member `rq_comm`.
- `rq_func`: A flag indicating which top-level MPI function created this request. See the file `share/-include/blktype.h` for a list of valid values.
- `rq_seq`: The message sequence number. If the rpimodule is to work with LAM tracing, then this number must be sent with each message (it is set by the MPI layer) and then on the receiving side, the rpi module must extract it and set this field in the receiving request with its value.
- `rq_f77handle`: Handle index used by Fortran.
- `rq_status`: The status of a request. In the case of a receive request, the rpi module must fill the `MPI_SOURCE` field of this structure with the rank of the sender of the received message, the `MPI_TAG` field with the tag of the received message, and the `st_length` field with the number of bytes in the received message.
- `rq_bsend`: Pointer to the buffer header in the case of a buffered send. Used by MPI layer only.
- `rq_proc`: Pointer to the peer process. This is initially set by the MPI layer. In the case of a receive on MPI_ANY_SOURCE, it will be NULL. Once the actual source has been determined, the rpi module may set it to point to the peer process but is not required to do so.

- `rq_next`: Pointer to the next request in the list. Do not modify. If the rpi module needs to maintain its own request lists, it must do so through the rpi-specific information handle (`rq_rpi`).
- `rq_extra`: A general place to hang extra state off the request. Used for one-sided and IMPI communications; the rpimodule should not modify this field.$^{(IMPI)}$
- `rq_hdlr`: Function to invoke when a request has moved into the done state. Don't touch this; it is used exclusively in the MPI layer (mostly by one-sided communication).$^{(OSD)}$
- `rq_shadow`: "Shadow" requests used by IMPI. Don't touch these; they are handled exclusively in the MPI layer.$^{(IMPI)}$
- `rq_rpi`: rpi-specific data hanging off the request. Its type is `(lam_ssi_rpi_req *)`, and must be defined by the rpi module. For example, the `tcp` rpi module stores (among others things) the request envelope and a pointer into the data buffer indicating the current read/write position.

3.3.4 MPI Communicator: `struct _comm`

The `struct _comm` type is the underlying type for an `MPI_Comm`. The majority of members of this type are probably not useful to rpi authors; detailed definitions of this members are skipped for brevity. See Figure 4.

```
typedef struct _comm *MPI_Comm;
struct _comm {
  int c_flags;
  int c_contextid;
  int c_refcount;
  MPI_Group c_group;
  MPI_Group c_rgroup;
  HASH *c_keys;
  int c_cube_dim;
  int c_topo_type;
  int c_topo_nprocs;
  int c_topo_ndims;
  int c_topo_nedges;
  int *c_topo_dims;
  int *c_topo_coords;
  int *c_topo_index;
  int *c_topo_edges;
  int c_f77handle;
  MPI_Win c_window;
  MPI_Errhandler c_errhdl;
  char c_name[MPI_MAX_OBJECT_NAME];
  MPI_Comm c_shadow;
  long c_reserved[4];
};
```

Figure 4: `struct _com`: Underlying structure for `MPI_Comm`.

The individual members are:

- `c_flags`: Bit flags indicating various characteristics of the communicator. The defined flags on this field are:
  - LAM_CINTER: If set, this communicator is an inter-communicator. If clear, this communicator is an intra-communicator.
  - LAM_CLDEAD: At least one process in the local group is dead.
  - LAM_CRDEAD: At least one process in the remote group is dead.
  - LAM_CFAKE: This is a "fake" IMPI communicator. rpi modules should never see this flag.$^{(IMPI)}$[2]
  - LAM_CHOMOG: All the processes on this communicator are endian-homogeneous. This flags is merely a shortcut for traversing all the procs in a given communicator to see if they are endian-homogeneous or not.
- `c_contextid`: Integer context ID.
- `c_refcount`: Reference count – effectively how many communications are currently using this communicator (since it is possible to MPI_COMM_FREE a communicator before all non-blocking communication has completed).
- `c_group`: Local group. Will be a meaningful group for intra-communicators, and MPI_GROUP_NULL for inter-communicators,
- `c_rgroup`: Remote group. Will be MPI_GROUP_NULL for intra-communicators, and a meaningful group for inter-communicators.
- `c_keys`: MPI attribute key hash table. See [1] for more discussion of the HASH LAM type and accessor functions.
- `c_cube_dim`: Inscribed cube dimension of the communicator. Used for binomial trees in MPI collectives.
- `c_topo_type`: Topology type; either MPI_GRAPH, MPI_CART, or MPI_UNDEFINED.
- `c_topo_nprocs`: Number of processes in topology communicators.
- `c_topo_ndims`: Number of dimensions in Cartesian communicators.
- `c_topo_nedges`: Number of edges in graph communicators.
- `c_topo_dims`: Array of dimensions for Cartesian communicators.
- `c_topo_coords`: Array of coordinates for Cartesian communicators.
- `c_topo_index`: Array of indices for graph communicators.
- `c_topo_edges`: Array of edges for graph communicators.
- `c_f77handle`: Fortran integer handle for this communicator.

[2] This flag is ugly and will go away when true multi-rpi support is available, because IMPI will likely become its own rpi module at that time.$^{(IMPI)}$

- `c_window`: In LAM/MPI, windows for one-sided message passing are implemented on top of communicators. Abstractly, all windows "have" a communicator that they communicate on (even though it is implemented the other way around – communicators designated for one-sided message passing "have" a window).$^{(OSD)}$
- `c_errhdl`: Error handler for this communicator.
- `c_name`: A string name for the communicator.
- `c_shadow`: A "shadow" communicator that is used by IMPI.$^{(IMPI)}$[3]
- `c_reserved`: Reserved for future expansion.

3.3.5 MPI Group: `struct _group`

The `struct _group` type is the underlying type for an `MPI_Group`. This type is probably not useful to rpi authors, but it is included here for completeness. See Figure 5.

```
typedef struct _group *MPI_Group;
struct _group {
  int g_nprocs;
  int g_myrank;
  int g_refcount;
  int g_f77handle;
  struct _proc **g_procs;
};
```

Figure 5: `struct _group`: Underlying structure for `MPI_Group`.

The individual members are:

- `g_nprocs`: The size of the group, i.e., the size of the `g_procs` array.
- `g_myrank`: The index of this process in the `g_procs` array. If the process is not in the group, this will be MPI_UNDEFINED.
- `g_refcount`: The reference count of this variable. Reference counting is maintained by the MPI layer.
- `g_f77handle`: The Fortran integer handle of this group.
- `g_procs`: Pointer to an array of pointers to the processes in the group. The array is in order of rank in the group. Note that these are simply references to the real, underlying `_proc` instances that represent the peer MPI processes – they are not copies. Be very careful modifying what these pointers refer to.

3.3.6 Request Status: `struct _status`

The `struct _status` type is the underlying type for an `MPI_Status`. See Figure 6.

[3] This members is ugly and will go away when true multi-rpi support is available, because IMPI will likely become its own rpi module at that time.$^{(IMPI)}$

```
typedef struct _status {
  int MPI_SOURCE;
  int MPI_TAG;
  int MPI_ERROR;
  int st_length;
} MPI_Status;
```

Figure 6: `struct _status`: Underlying structure for `MPI_Status`.

Note that by definition in the MPI standard, the first three members listed above (`MPI_SOURCE`, `MPI_TAG`, and `MPI_ERROR`) are public variables.

The individual members are:

- `MPI_SOURCE`: As described by the MPI standard.
- `MPI_TAG`: As described by the MPI standard.
- `MPI_ERROR`: As described by the MPI standard.
- `st_length`: Private variable for use by the MPI and rpi layers. It is the length of the message in bytes.

3.3.7 Message Envelopes: `struct lam_ssi_rpi_envl`

The following type is provided as a prototype envelope that can be used by rpi modules for prefixing data messages across a communications channel. Although the use of this specific `struct` is not [yet] required, it is strongly encouraged because it will provide compatibility with TotalView debugging, unexpected queue support, and may become required for multi-rpi support. See Figure 7.

```
struct lam_ssi_rpi_envl {
  int4 ce_len;
  int4 ce_tag;
  int4 ce_flags;
  int4 ce_rank;
  int4 ce_cid;
  int4 ce_seq;
};
```

Figure 7: `struct _lam_ssi_rpi_envl`: General structure for envelopes.

The individual members are:

- `ce_len`: Message length (bytes).
- `ce_tag`: Tag of the message (16 bits max).
- `ce_flags`: Flags on this particular envelope. Valid values are:

  - C2CTINY: "Tiny" message protocol (usually indicates that the envelope and message payload are included in a single message (i.e., the message payload may have already been received by receiving the envelope).
  - C2CSHORT: "Short" message protocol (usually indicates that the envelope and message payload were sent in separate messages, or, more specifically, must be received in separate buffers, but the message payload directly follows the envelope).
  - C2CLONG: "Long" or "rendevouz" message protocol (usually indicates a three-way handshake to actually transfer the message). This is required for arbitrarily long messages where resources may need to be allocated on the receiver before they can be received properly.
  - C2CACK: During a long message protocol handshake, the receiver sends an envelope back to the sender with this bit set indicating that it is ready to receive the main body of the message.
  - C2C2ND: During a long message protocol handshake, the sender sends an envelope with this bit set, indicating that the message payload is immediately following.
  - C2CSSEND: Indicates a synchronous mode send, which requires the receiver to send back an envelope with C2CACK set before the sending request can complete.
  - C2CBOX: Long message using the postbox protocol.
  - C2CBUFFERED: The envelope has previously been buffered.

- `ce_rank`: Peer rank. This may be the source or the destination, depending on the context of the envelope.
- `ce_cid`: Context ID of the communicator in which this message is being sent/received.
- `ce_seq`: Sequence number.

3.3.8 Message Buffering: `struct lam_ssi_cbuf_msg`

This type may be used for unexpected message buffering. Its use is strongly recommended (see Section 3.5.2, page 20) in order to enable external access to unexpected message queues.

```
struct lam_ssi_rpi_cbuf_msg {
  struct _proc *cm_proc;
  struct lam_ssi_rpi_envl cm_env;
  char *cm_buf;
  int cm_dont_delete;
  MPI_Request cm_req;
};
```

Figure 8: `struct _lam_ssi_rpi_cbuf_msg`: Unexpected message bodies.

The individual members are as follows:

- `cm_proc`: Pointer to the source process.
- `cm_env`: Copy of the incoming envelope.

- `cm_buf`: Message data. This may or may not be NULL. For example, in the `tcp` rpi module, this pointer only ever points to short messages because long message data has not yet been received, since, by definition, an unexpected message receipt means that a corresponding long receive request has not yet been posted.
- `cm_dont_delete`: Flag to indicate whether the buffer should be freed or not when the request has completed.
- `cm_req`: The send request *only* if the sender is this process; NULL otherwise.

The use of this `struct` is explained more fully in Section 3.5.2.

3.4 Global Variables

Several global variables are available to all rpi modules. These variables are `extern`'ed in the rpi header file.

3.4.1 `struct _proc *lam_myproc`

A pointer to the `struct _proc` (described in Section 3.3.2) of this process. It is most commonly used to get the attributes and/or GPS (see Section 3.3.1) of this process.

This variable is `extern`'ed in `<mpisys.h>`.

3.4.2 `struct _proc **lam_procs`

An array of pointers to all the MPI processes. This array will be filled after MPI_INIT returns, and will be empty again after MPI_FINALIZE returns. This array is automatically updated whenever a process is added or removed, so it should be valid at all times. The length of the array is `lam_num_procs`, described in Section 3.4.3.

rpi authors are strongly discouraged from using this array. Instead, the arrays that are passed to the `lsr_init()` and `lsra_addprocs()` API calls (see Sections 4.4 and 4.5, respectively) should be used exclusively for the following reasons:

- The arrays passed to these API functions become the "property" of that rpimodule, and need not worry about multithread synchronization outside of the rpi module.
- The arrays passed to these API functions are the only `_proc`s that the rpi module is concerned with – which may be less than all `_proc`s in the current MPI job. The global array `lam_procs` is *all* MPI processes, not just the ones that a given rpi module has responsibility for.

This variable is `extern`'ed in `<mpisys.h>`.

3.4.3 `int lam_num_procs`

This variable is the length of the `lam_procs` array, as described in Section 3.4.2.

This variable is `extern`'ed in `<mpisys.h>`.

3.5 Functions

Several common functions are provided to all rpi SSI modules.

3.5.1 MPI Process Data Structures

A process list is maintained by each MPI process containing a list of `struct _proc` instances – one for each process that it knows about and can communicate with (including itself). The list is initially the ordered set processes in MPI_COMM_WORLD, but may be augmented after `MPI_INIT` if any of the MPI-2 dynamic process functions are invoked. `struct _proc` instances listed after the set of processes in MPI_COMM_WORLD are not guaranteed to be in any particular order.

The process list is opaque and can be traversed with the accessor functions `lam_topproc()` and `lam_nextproc()`. For example:

```
for (p = lam_topproc(); p != NULL; p = lam_nextproc()) {
  /* do what you want here with process p */
}
```

Note, however, that this may have unintended side-effects. Be aware that `lam_nextproc()` maintains state (i.e., the current process). So having nested loops that invoke `lam_topproc()` and/or `lam_nextproc()` will almost certainly not do what you intend.

Also note that `lam_topproc()` and `lam_nextproc()` traverse a list of *all* MPI processes involved in the parallel application. This may be a superset of the processes that a particular rpi module is responsible for. If the rpi module only needs to traverse the list of processes that it "owns", it should traverse its own array that is incrementally given to it by the `lsr_init()` and `lsra_addprocs()` API calls (see Sections 4.4 and 4.5, respectively).

3.5.2 Unexpected Message Buffering

It is strongly recommended that rpi modules use the `cbuf_*()` functions provided by LAM for unexpected message buffering for the following reasons:

- When the rpi design is evolved into multi-rpi, having a common buffering for unexpected messages will likely be required to handle unexpected messages in conjunction with MPI_ANY_SOURCE in communicators that span multiple rpi modules.
- LAM/MPI supports the Etnus TotalView parallel debugger which has the ability to display MPI message queues. LAM exports the unexpected message queues through the standard functions described in this section; if an rpi module uses the LAM-provided functions, TotalView will be able to see the unexpected message queue.

The LAM-provided functions for unexpected message buffering are:

- `lam_ssi_rpi_cbuf_init(void)`: This function is required to be called before any unexpected buffering can occur. It is invoked automatically by the rpi SSI startup glue after all rpi module `open()` functions are invoked, but *before* any rpi module `lsr_init()` functions (see Section 4.4, page 24) are invoked. This function is only mentioned here for completeness.
- `lam_ssi_rpi_cbuf_end(void)`: This function cleans up all the storage and state associated with unexpected message buffering. It is invoked automatically by the rpi SSI shutdown glue after all rpi module `lsra_finalize()` functions are invoked with `(proc == 0)` (see Section 4.6, page 25), but *before* the rpi module `close()` functions are invoked.

- `lam_ssi_rpi_cbuf_find(struct lam_ssi_rpi_envl *rqenv)`: Given a pointer to an envelope, find if there are any matching messages on the unexpected message queue. If there are no matching messages, NULL is returned. Otherwise, a pointer to the first matching `struct lam_ssi_rpi_cbuf_msg` is returned that contains information about the buffered message (see Section 3.3.8, page 18).

  Note that because this function may be invoked by a probe, it does *not* remove the message from the unexpected queue.

- `lam_ssi_rpi_cbuf_delete(struct lam_ssi_rpi_cbuf_msg *msg)`: Delete a specific message from the unexpected message queue. The argument is a pointer that was returned from either `lam_ssi_rpi_cbuf_find()` or `lam_ssi_rpi_cbuf_append()`.

- `lam_ssi_rpi_cbuf_append(struct lam_ssi_rpi_cbuf_msg *msg)`: Append a new unexpected message to the end of the queue. `*msg` is copied by value, so there's no need for `msg` to point to stable storage. See Section 3.3.8 (page 18) for an explanation of this type. This function returns a pointer to the buffered message upon success, or NULL on failure.

3.5.3 Utility Functions

- `lam_memcpy()`: While technically not an rpi-specific call, `lam_memcpy()` is important because some platforms have poor implementations of `memcpy()`. On these platforms, `lam_memcpy()` (particularly with mid- to large-sized copies) may significantly outperform the native `memcpy()`. On platforms with "good" implementations of `memcpy()`, `lam_memcpy()` will actually be a `#define` that maps to `memcpy()` in order to use the native function without any additional function call overhead. Hence, it is always safe to use `lam_memcpy()` instead of `memcpy()`, and ensure portable memory copying performance.

  The prototype for this function is the same as for `memcpy()`.

- `lam_ssi_rpi_base_alloc_mem()`: A wrapper around `malloc(3)`. This function is provided for rpi modules that do not wish to provide their own `lsra_alloc_mem()` functions.

  This function fulfills all the requirements (such as prototype) as the `lsra_alloc_mem()` API call needs. See Section 4.14 (page 30).

- `lam_ssi_rpi_base_free_mem()`: A wrapper around `free(3)`. This function is provided for rpi modules that do not wish to provide their own `lsra_free_mem()` functions.

  This function fulfills all the requirements (such as prototype) as the `lsra_free_mem()` API call needs. See Section 4.15 (page 30).

4 rpi SSI Module API

This is version 1.0.0 of the rpi SSI module API.

Each rpi SSI module must export a `struct lam_ssi_rpi_1_0_0` named `lam_ssi_rpi_<name>_module`. This type is defined in Figure 9. A second `struct` is used to hold the majority of function pointers and flags for the module. It is only used if the module is selected, and is shown in Figure 10.

The majority of the elements in Figures 9 and 10 are function pointer types; each is discussed in detail below. When describing the function prototypes, the parameters are marked in one of three ways:

- IN: The parameter is read – but not modified – by the function.

```
typedef struct lam_ssi_rpi_1_0_0 {
  lam_ssi_1_0_0_t lsr_meta_info;

  /* RPI API function pointers */

  lam_ssi_rpi_query_fn_t lsr_query;

  lam_ssi_rpi_init_fn_t lsr_init;
} lam_ssi_rpi_1_0_0_t;
```

Figure 9: `struct lam_ssi_rpi_1_0_0`: The rpi basic type for exporting the module meta information and initial query / initialization function pointers.

```
typedef struct lam_ssi_rpi_actions_1_0_0 {

  /* RPI API function pointers */

  lam_ssi_rpi_addprocs_fn_t lsra_addprocs;
  lam_ssi_rpi_finalize_fn_t lsra_finalize;

  lam_ssi_rpi_build_fn_t lsra_build;
  lam_ssi_rpi_start_fn_t lsra_start;
  lam_ssi_rpi_advance_fn_t lsra_advance;
  lam_ssi_rpi_destroy_fn_t lsra_destroy;

  lam_ssi_rpi_iprobe_fn_t lsra_iprobe;

  lam_ssi_rpi_fastrecv_fn_t lsra_fastrecv;
  lam_ssi_rpi_fastsend_fn_t lsra_fastsend;

  lam_ssi_rpi_alloc_mem_fn_t lsra_alloc_mem;
  lam_ssi_rpi_free_mem_fn_t lsra_free_mem;

  lam_ssi_rpi_checkpoint_fn_t lsra_checkpoint;
  lam_ssi_rpi_continue_fn_t lsra_continue;
  lam_ssi_rpi_restart_fn_t lsra_restart;

  /* Flags */

  int lsra_tv_queue_support;
} lam_ssi_rpi_actions_1_0_0_t;
```

Figure 10: `struct lam_ssi_rpi_actions_1_0_0`: The rpi type for exporting API function pointers.

- OUT: The parameter, or the element pointed to by the parameter may be modified by the function.
- IN/OUT: The parameter, or the element pointed to by the parameter is read by, and may be modified by the function.

rpi module writers looking for insight into how the API is used should also look at the source code in `share/mpi/lamreqs.c`. Most MPI functions that involve communication eventually call one or more of the functions in this file.

Unless specifically noted, none of the functions may block. Note that this may make single-threaded implementations arbitrarily complicated. For example, the state machine used in the `tcp` rpi module is extremely complicated for this very reason; in non-blocking mode, reads and writes on sockets may return partial completion which will require re-entering the same state at a later time.

4.1 Restrictions

It is illegal for any rpi API function to invoke top-level MPI functions.

4.2 Data Item: `lsr_meta_info`

`lsr_meta_info` is the SSI-mandated element contains meta-information about the module. See [2] for more information about this element.

4.3 Function Call: `lsr_query`

- Type: `lam_ssi_rpi_query_fn_t`

  typedef int (∗lam_ssi_rpi_query_fn_t)(**int** ∗priority, **int** ∗thread_min, **int** ∗thread_max);

- Arguments:
  - OUT: `priority` is the priority of this module, and is used to choose which module will be selected from the set of available modules at run time.
  - OUT: `thread_min` is the minimum MPI thread level that this module supports.
  - OUT: `thread_max` is the maximum MPI thread level that this module supports.
- Return value: 1 if the module wants to be considered for selection, 0 otherwise.
- Description: This function determines whether a module wants to be considered for selection or not. It can invoke whatever initialization functions that it needs to determine whether it can run or not. If this module is not selected, its `lsra_finalize()` function will be invoked shortly after this function.

  Additionally, the module must fill in `thread_min` and `thread_max` to be the minimum and maximum MPI thread levels that it supports. `thread_min` must be less than or equal to `thread_max`. See [2] for more details on the priority system and how modules are selected at run time.

  If the module does not want to be considered during the negotiation for this application, it should return 0 (the values in `priority`, `thread_min`, and `thread_max` are then ignored).

4.4 Function Call: `lsr_init`

- Type: `lam_ssi_rpi_init_fn_t`

  typedef lam_ssi_rpi_actions_t (∗lam_ssi_rpi_init_fn_t)(**struct** _proc ∗∗procs, **int** nprocs, **int** ∗maxtag, **int** ∗maxcid);

- Arguments:

  - IN: `procs` is an array of pointers to the initial set of `struct _proc` instances that this rpi module is responsible for.[4] The `procs` array will be freed after the call to `lsr_init()` completes; the rpi module is responsible saving its own copy.
  - IN: `nprocs` is the length of the `procs` array.
  - OUT: `maxtag` is the maximum MPI tag value that this rpi module can handle. `*maxtag` will be the current max tag value when this function is invoked. The rpi module may lower this value if necessary, but may *not* raise it!
  - OUT: Similar to `maxtag`, `maxcid` is the maximum number of communicators that this rpi module can handle (i.e., the maximum communicator CID). `*maxcid` will be the current maximum CID value when this function is invoked. The rpi module may lower this value, but it may *not* raise it!

- Return Value: A pointer to the `struct` shown in Figure 10. If the module returns NULL, an error will occur, because negotiation is over and this module has been selected.

- Description: Performs primary initialization of the rpi module (called from MPI_INIT) after the rpi module negotiation; it will only be called in the selected module.[5] This function typically performs once-only initialization of the communication sub-layer and initialize all processes with respect to the communication sub-layer. The latter may simply involve a call to `lsra_addprocs()` to initialize the initial set of "new" processes (even though this is the first set of processes that the rpi module will receive).

  The `tcp` rpi module, for example, initializes a hash table for message buffering and then calls `lsra_addprocs()` to save the `procs` array and set up the TCP socket connections between the initial processes.

  At the time of this call, the MPI process is also a LAM process, hence all LAM functionality is available to it. In particular the LAM message passing routines `nsend(2)` and `nrecv(2)` (see the LAM documentation and manual pages for more details) are available and can be used to pass out-of-band information between the MPI processes.[6] The `tcp` rpi module uses these functions to pass server socket port numbers to clients who must connect. See the function `connect_all()` in `share/ssi/rpi/tcp/src/ssi_rpi_tcp.c`.

  Finally, the rpi module may lower the current maximum MPI tag and CID values. The final values used will be the maximum over all rpi modules that are used in an MPI process (this will be more

[4] Passing an array of `(struct _proc *)` is intended to be among the first steps towards "multi-rpi". Although the current implementation of LAM will pass in the entire array of procs to the `lsr_init()` call, when multi-rpibecomes a reality, each rpiwill likely receive only a subset of all available procs. Hence, rpi modules should treat the list of procs that they receive via `lsr_init()` and `lsra_addprocs()` as the *only* procs that they are allowed to directly communicate with, and assume that any other procs that are not provided will be handled by other rpi modules.$^{(MRPI)}$

[5] Note that when multi-rpibecomes a reality, this function will likely be invoked on *all* the selected modules.$^{(MRPI)}$

[6] Remember that it is illegal for rpi modules to invoke MPI functions (e.g., MPI_SEND, MPI_RECV).

relevant when multi-rpibecomes a reality). Hence, an rpi module may *lower* these values, but the rpi module *may not increase them!*

4.5 Function Call: `lsra_addprocs`

- Type: `lam_ssi_rpi_addprocs_fn_t`

  typedef int (∗lam_ssi_rpi_addprocs_fn_t)(**struct** _proc ∗∗procs, **int** nprocs);

- Arguments:

  – `procs`: An array of pointers to *new* `struct _proc` instances that this rpi module is responsible for.[7] The `procs` array will be freed after the call to `lsr_init()` completes; the rpi module is responsible saving its own copy.

  – `nprocs`: Length of the `procs` array.

- Return Value: Zero on success, LAMERROR otherwise.

- Description: In LAM/MPI, a process can become aware of new processes with which it may communicate. For example, when it spawns MPI children. The MPI layer adds new process entries to the process list and then calls `lsra_addprocs()` to perform rpi module initialization with a list of *only* these new processes. The rpi module is responsible for augmenting its own internal list with the contents of the `proc` array.

  The `tcp` rpi module, for example, adds the contents of the `proc` array to its internal list and then sets up the TCP socket connections between the new processes and the old ones.

  This function is called from MPI_INTERCOM_CREATE, MPI_COMM_SPAWN, and MPI_COMM_SPAWN_MULTIPLE.

  It is important to allow `lsra_addprocs()` to fail gracefully; do not use network protocols during setup that may deadlock or "hang" in the event of a failure. If so, commands such as `mpirun` and functions such as `MPI_COMM_SPAWN` may never complete.

4.6 Function Call: `lsra_finalize`

- Type: `lam_ssi_rpi_finalize_fn_t`

  typedef int (∗lam_ssi_rpi_finalize_fn_t)(**struct** _proc ∗proc);

- Arguments:

  – IN/OUT: `proc` is the `_proc` to shut down, or NULL if the entire rpi module is to be shut down. This function should really only modify the flags in `p_mode` and nothing else in the non-rpi-specific portion of `proc`.

- Return Value: Zero on success, LAMERROR otherwise.

[7] See footnote 4 on page 24.

- Description: Performs final cleanup of a given `_proc` instance and/or the over rpi module (e.g., clean up all data structures, etc., created by the rpi module). This function is called from MPI_FINALIZE after all pending communication has completed. It is always called at least once, with `(proc == NULL)`.

  When `lsra_finalize()` is invoked with `(p != NULL)`, it is the rpi module's responsibility to never reference that process again, even when `lsra_finalize()` is invoked with `(p == NULL)`.

  If MPI-2 dynamic functions were invoked during the program's run, `lsra_finalize()` may be invoked multiple times with `(proc != NULL)` for the `_proc` instances that are not part of `MPI_COMM_WORLD`. Note that this may even happen *before* MPI_FINALIZE is invoked. For example, if processes are added by an MPI-2 dynamic function (e.g., MPI_COMM_SPAWN), but then later all communicators containing the spawned processes are freed via MPI_COMM_FREE, then `lsra_finalize()` will be invoked for each process that is no longer referenced.

  The last invocation of `lsra_finalize()` is always with `(proc == NULL)`, regardless of whether MPI-2 dynamic functions were used or not.

4.7 Function Call: `lsra_build`

- Type: `lam_ssi_rpi_build_fn_t`

typedef int (∗lam_ssi_rpi_build_fn_t)(MPI_Request req);

- Arguments:
  - IN/OUT: `req` is the request to build.
- Return Value: Zero on success, LAMERROR otherwise.
- Description: When the MPI layer creates a new request, it initializes general request information and then calls this function to build the rpi-specific portion of the request. Certain rpi module state, especially that which is unchanged over multiples invocations of a persistent operation, may be initialized here too. This function is called from `_mpi_req_build()`.

  This step is separated from the "start" phase in order to optimize persistent MPI communication – "build" only needs to occur once, while "start" may be invoked many times.

4.8 Function Call: `lsra_start`

- Type: `lam_ssi_rpi_start_fn_t`

typedef int (∗lam_ssi_rpi_start_fn_t)(MPI_Request req_top, MPI_Request req);

- Arguments:
  - IN: `req_top` is the top of the active list.
  - IN/OUT: `req` is the request to be started.
- Return Value: Zero on success, LAMERROR otherwise.

- Description: The MPI layer, after adding a request to the progression list, calls `_mpi_req_start()` to make it ready for subsequent progression. Among other things, it moves the request's state to the start state and then calls `lsra_start()` so that the rpi module can do any initialization it needs to make the request ready.

  This step is separated from the "build" phase in order to optimize persistent MPI communication – "build" only needs to occur once, while "start" may be invoked many times.

  This function may also perform some progression past the start state (this is really the only reason that `req_top` is passed in). For example, an rpi module needs to also handle the special case of a process sending to or receiving from itself here, and may thus actually advance a request all the way to the done state.[8]

  If any further progression is done, the request's state must be updated to reflect this. The possible states after the start state are:

  1. The active state: where the data transfer protocol is not yet finished but we have done some transfer and are past the point of cancellation, and
  2. The done state: where the data transfer protocol is finished and the request can be completed.

4.9 Function Call: `lsra_advance`

- Type: `lam_ssi_rpi_advance_fn_t`

  typedef int (∗lam_ssi_rpi_advance_fn_t)(MPI_Request req_top, **int** fl_block);

- Arguments:
  - IN/OUT: `req_top` is the first request that is in the active list.
  - IN: `fl_block` is 1 if `lsra_advance()` is allowed to block, or 0 if `lsra_advance()` must not block.

- Return Value: 1 if any requests' state has changed, 0 if none have changed state, or LAMERROR if an error occurred.

- Description: This is where most of the work gets done. Given a pointer to the top of the progression list, advance them where possible. The flag `fl_block` is true if it is permitted to block until progress is made on at least one request.

  The MPI layer knows and cares nothing about message transfer protocols and message buffering. This is solely the responsibility of the rpi module. The rpi module however must update the state of the request as it progresses from init, to start, to active, and finally to done, so that the MPI layer can do the Right Things.

  Note that a request may be moved from the start to the done state outside of the regular rpi progression by being canceled. The progression function `lsra_advance()` must take this into account. Currently, LAM does not allow the cancellation of requests which are in the active state.

  The rpi module must also update other information in requests where appropriate.

[8] Multi-rpibehavior will obsolete the need for "send-to-self" special cases because there will likely be a self rpi module that will be used for all such cases.$^{(MRPI)}$

4.10 Function Call: lsra_destroy

- Type: lam_ssi_rpi_destroy_fn_t

  typedef int (∗lam_ssi_rpi_destroy_fn_t)(MPI_Request req);

- Arguments:
  - IN/OUT: req is the request to destroy.
- Return Value: Zero on success, LAMERROR otherwise.
- Description: Destroys only the rpi portion of request. It is called from _mpi_req_destroy(). This function should free any dynamic storage created for this request by the rpi module and also perform any other necessary cleanup.

  Note: it is only necessary to clean up what was created/done in other parts of the rpi module. The rest of the request will be cleaned up by the MPI layer itself.

4.11 Function Call: lsra_iprobe

- Type: lam_ssi_rpi_iprobe_fn_t

  typedef int (∗lam_ssi_rpi_iprobe_fn_t)(MPI_Request req);

- Arguments:
  - IN/OUT: req is the request to check for.
- Return Value: 0 if no match was found, 1 if a match was found, or LAMERROR if an error occurred.
- Description: Implements the strange non-blocking probe beast. It is called from MPI_IPROBE and is passed a non-blocking probe request which has been built and started. This function should check for matches for the probe in a non-blocking fashion and then return a value of 0 if no match was found, 1 if a match was found or LAMERROR if an error occurred.

  In the case of a match, the MPI status in the request must also be updated as required by the definition of MPI_IPROBE.

  This is such a strange function that the generalized code in the tcp rpi may be sufficient for other rpi modules.

4.12 Function Call: lsra_fastrecv

- Type: lam_ssi_rpi_fastrecv_fn_t

  typedef int (∗lam_ssi_rpi_fastrecv_fn_t)(**char** ∗buf, **int** count, MPI_Datatype type, **int** src, **int** ∗tag, MPI_Comm comm, MPI_Status ∗status, **int** ∗seqnum);

- Arguments:

- IN: `buf` is a pointer to the buffer to receive the incoming message in. It corresponds to the buffer argument in the invoking MPI receive call.
- IN: `count` is the number of elements to receive. It corresponds to the count argument in invoking the MPI receive call.
- IN: `type` is the MPI datatype of the element(s) to receive. It corresponds to the datatype argument in the invoking MPI receive call.
- IN: `src` is the source rank to receive from. As described below, it will not be MPI_ANY_SOURCE. `src` corresponds to the source rank argument in the invoking MPI receive call.
- IN/OUT: `tag` is the tag to use. It corresponds to the tag argument in the invoking MPI receive call. Upon return, it must be set to the tag that was actually used. Note that the tag must be set in the case of MPI_ANY_TAG because the `status` argument may be MPI_STATUS_IGNORE, and the actual tag would otherwise be lost.
- IN: `comm` is the communicator to receive in. It corresponds to the communicator argument in the invoking MPI receive call.
- IN/OUT: `status` is the status that must be filled when this function returns, or the special constant MPI_STATUS_IGNORE. It corresponds to the status argument from the invoking MPI receive call.
- IN/OUT: `seqnum` is the sequence number of the incoming message from the sender's perspective. It is used for matching messages in trace files.

- Return Value: MPI_SUCCESS on success, or LAMERROR on error.
- Description: Like `lsra_fastsend()`, this function is intended to bypass the normal rpi progression mechanism, and is only called from MPI_RECV if there are no other active requests and the source of the message is neither MPI_ANY_SOURCE nor the destination. If a matching message has already arrived (and assumedly been buffered somewhere), it can just fill in the relevant values and return MPI_SUCCESS. If the message has not already arrived, it can block waiting for the message (since no other requests are active).

4.13 Function Call: `lsra_fastsend`

- Type: `lam_ssi_rpi_fastsend_fn_t`

```
typedef int (*lam_ssi_rpi_fastsend_fn_t)(char *buf, int count, MPI_Datatype type, int dest, int tag,
    MPI_Comm comm);
```

- Arguments:
  - IN: `buf` is a pointer to the buffer containing bytes to send. It corresponds to the buffer argument in the invoking MPI send call.
  - IN: `count` is the number of elements to send. It corresponds to the count argument in invoking the MPI send call.
  - IN: `type` is the MPI datatype of the element(s) to send. It corresponds to the datatype argument in the invoking MPI send call.
  - IN: `dest` is the destination rank to send to. It corresponds to the destination rank argument in the invoking MPI send call.

- IN: `tag` is the tag to use.
- IN: `comm` is the communicator to send in. It corresponds to the communicator argument in the invoking MPI send call.

- Return Value: MPI_SUCCESS on success, or LAMERROR on error.

- Description: This is a special case "short circuit" fast send. It was originally an experiment to optimize common sends and receives, but has proved to be a stable and efficient method of bypassing much of the request mechanism (and therefore, avoiding overhead).

  This function is a fast blocking send; it takes all the same arguments as MPI_SEND. It is only invoked from blocking sends when there are no active requests in the rpi module and the destination is not the same as the source. In this case, it is safe to bypass the normal rpi progression mechanism and send the message immediately. This function is allowed to block if necessary (since no other requests are active). No request is created, so the send must be completed (in terms of the MPI layer) when the function returns. It must return MPI_SUCCESS or an appropriate error code.

  Note that this function is not suitable for synchronous sends because it does not certify the the destination has posted a receive in MPI.

4.14 Function Call: `lsra_alloc_mem`

- Type: `lam_ssi_rpi_alloc_mem_fn_t`

```
typedef int (*lam_ssi_rpi_alloc_mem_fn_t)(MPI_Aint size, MPI_Info info, void *baseptr);
```

- Arguments:

  - IN: `size` is the number of bytes to be allocated.
  - IN: `info` is any "hint" information passed in from MPI_ALLOC_MEM.
  - OUT: `baseptr`, as described in MPI-2:4.11, this is an OUTvariable, but is `(void *)` for convenience. Hence, it is actually a pointer to the actual pointer that will be filled. You can think of it as a `(void **)`, even though it really isn't.

- Return Value: Zero on success, LAMERROR otherwise.

- Description: This function is used by the rpi module to allocate "special" memory that can be used for fast message passing (such as pinned memory for a Myrinet or VIA implementation). This function is invoked as the back-end of MPI_ALLOC_MEM.

  If the rpi module does not need "special" memory for any reason, the function `lam_ssi_rpi_base_alloc_mem()` can be used as the value of this pointer instead (see Section 3.5.3), which is mainly a wrapper around `malloc(3)`.

4.15 Function Call: `lsra_free_mem`

- Type: `lam_ssi_rpi_free_mem_fn_t`

```
typedef int (*lam_ssi_rpi_free_mem_fn_t)(void *baseptr);
```

- Arguments:
  - IN: `baseptr` is a pointer to the memory to be freed. It should be a value that was previously returned from `lsra_alloc_mem()`.
- Return Value: Zero on success, LAMERROR otherwise.
- Description: This function is used by the rpi module to deallocate "special" memory that was previously allocated by `lsra_alloc_mem()`. This function is invoked as the back-end of MPI_FREE_MEM.

  If the rpi module does not need "special" memory for any reason, the function `lam_ssi_rpi_base_free_mem()` can be used as the value of this pointer instead (see Section 3.5.3), which is mainly a wrapper around `free(3)`.

4.16 Function Call: `lsra_checkpoint`

- Type: `lam_ssi_rpi_checkpoint_fn_t`

```
typedef int (*lam_ssi_rpi_checkpoint_fn_t)(void);
```

- Arguments: None
- Return Value: Zero on success, LAMERROR otherwise.
- Description: This function is part of the checkpoint/restart functionality. It is invoked by the selected cr SSI module when a checkpoint is invoked.

  The purpose of this function is to do whatever is necessary for the rpi module to ready itself for checkpoint. For example, it may drain the network of any "in-flight" messages. This function may use any of the out-of-band communication mechanisms (such as `nsend(2)` and `nrecv(2)`) to ensure that all "in-flight" MPI messages are received before the function returns.

  Note that it may not be required to close all network connections. LAM's checkpointing model entails taking a checkpoint and then continuing the job. Hence, one possible model for the checkpoint function is to quiesce the network and then return – leaving all network connections intact. Upon continue, no special actions are required – the network connections are already in place, and normal MPI message passing progression can continue. Upon restart, however, all the network connections will be stale, and will need to be closed or discarded, and then re-opened.

  If the rpi module does not support checkpoint/restart functionality, it should provide NULL for this function pointer.

4.17 Function Call: `lsra_continue`

- Type: `lam_ssi_rpi_continue_fn_t`

```
typedef int (*lam_ssi_rpi_continue_fn_t)(void);
```

- Arguments: None
- Return Value: Zero on success, LAMERROR otherwise.

- Description: This function is part of the checkpoint/restart functionality. It is invoked by the selected cr SSI module when a checkpoint has finished and the parallel application is continuing afterward.

  The purpose of this function is to do whatever is necessary for the rpi module to be continue the job. Note that if the rpi module provides checkpoint/restart support, this function must be provided – even if it does nothing other than return 0.

  If the rpi module does not support checkpoint/restart functionality, it should provide NULL for this function pointer.

4.18 Function Call: `lsra_restart`

- Type: `lam_ssi_rpi_restart_fn_t`

  typedef int (∗lam_ssi_rpi_restart_fn_t)(**void**);

- Arguments: None
- Return Value: Zero on success, LAMERROR otherwise.
- Description: This function is part of the checkpoint/restart functionality. It is invoked by the selected C/R SSI module when a parallel process is restarted.

  The purpose of this function is to do whatever is necessary to restart the rpi module and ready it for MPI communications. Since the process has just restarted, it is likely to have stale network connections; it is typically safest to close/discard all network connections and re-initiate them.

  If the rpi module does not support checkpoint/restart functionality, it should provide NULL for this function pointer.

4.19 Data Member: `lsra_tv_queue_support`

- Type: `int`
- Description: This flag is used by LAM to determine if the rpi module supports TotalView queue debugging or not. Currently, this means that unexpected messages use the interface described in Section 3.5.2 (page 20).

  If the rpi module uses the Section 3.5.2 interface, it should set this flag to 1. Otherwise, it should set it to 0.

5 To Be Determined

Things that still need to be addressed:

- Currently, the rpi `close()` function is not invoked when we MPI_ABORT. What to do about this? Put an explicit call to the rpi close() in MPI_ABORT?

5.1 Multi-rpi Support

The current rpi design only allows one rpi module to be active at a time. It is not obvious that this is a problem because LAM's `tcp` rpi was cleverly designed such that its lower half can [effectively] be invoked by either of the shared memory modules (`sysv` and `usysv`). However, this is clearly not extensible to all other RPIs. More specifically, it is desirable to allow all "off-node" rpi modules to be able to share message passing responsibilities with either of the shared memory rpi modules for "on-node" message passing without the addition of "cleverly designed", multi-layer rpi modules.

Hence, not only will the the shared memory rpi modules be able to be used with any "off-node" rpi module, the "off-node" rpi modules will be able to be selectively chosen (at run time) depending on what kind of network connectivity exists between pairs of peer MPI processes.

For example, consider the MPI process layout in Figure 11. Process B_1 has TCP connectivity to process A, shmem connectivity to B_2 (because B_1 and B_2 are on the same node), and Myrinet connectivity to C. In this case, it would be best if B_1 can use the rpi module corresponding to the type of network connectivity that it has with each peer process. The current rpi design does not allow for this – the best that one do in situations like Figure 11 is to settle for a least common denominator (e.g., `tcp`).

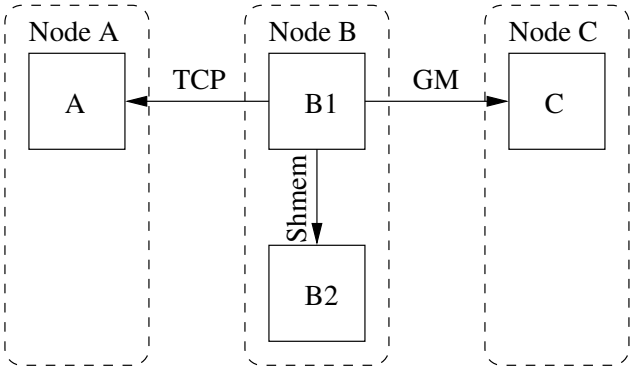

Figure 11: MPI process B_1 has different kinds of connectivity to each of its peer processes.

Future designs of the rpi will allow for this kind of "multi-rpi" behavior. Although no specifics are known about this design yet, the following assumptions are probably fairly safe:

- Each of the rpi modules will be given a specific set of MPI peer processes that it is responsible for communicating with.
- The MPI library will then manage the invocation of each rpi module methods as appropriate.
- Unexpected message handling will have to be standardized and pooled.
- Additional rpi API calls and/or abstractions may be required to properly handle MPI_ANY_SOURCE. Examples include:
  - Allowing multiple devices to cache their own module-specific information on the request.
  - Once an MPI_ANY_SOURCE request is matched by an rpi module, it will likely need to be removed from the "pending" queues on the other selected rpi modules.
- A "self" rpi module will be written that will handle all "send to self" and "receive from self" requests. This will allow all current rpi modules to avoid having to handle this case.

- The current shared memory rpi modules will have the TCP code removed and become pure shared memory modules.
- LAM_PRIINIT and LAM_RPICONNECT will likely be eliminated from the `struct _proc` structure, for the following reasons:
  - They are specific to each rpi module, and therefore not suitable for a shared structure.
  - With the possibility of coll modules using different progression engines than rpi modules, whether a process is "connected" or not is really a per-SSI-type issue, not a top-level/shared data structure issue.

  As such, these two flags seem like abstraction violations, and will be moved/eliminated.

5.2 MPI_THREAD_MULTIPLE Support

Additionally, threading will likely be a factor in future rpi designs. It is possible that it will be permissible to allow multiple threads to invoke rpi API calls simultaneously. Since, again, little is known about threading designs yet, the following recommendations / observations are noted here:

- rpi authors are encouraged to write thread-safe code. Using locks to protect shared variables is not yet recommended (this would needlessly incur a performance penalty), but at least leave comments and/or preprocessor symbols indicating where locks *would* need to obtained and released in multi-threaded situations.
- Future designs of the rpi will likely need some kind of atomic `find_and_remove()` function to account for race conditions when attempting to receive the same unexpected message in multiple threads.
- It is likely that the "fast" functions will have to go away when we have multi-threaded rpi modules. Or, more specifically, when MPI_THREAD_MULTIPLE is used, the "fast" functions will not be able to be used.

6 Acknowledgements

This work was supported by a grant from the Lily Endowment, National Science Foundation grant 0116050, and used computational facilities at the College of William and Mary which were enabled by grants from Sun Microsystems, the National Science Foundation, and Virginia's Commonwealth Technology Research Fund.

References

[1] Brian Barrett, Jeff Squyres, and Andrew Lumsdaine. *LAM/MPI Design Document*. Open Systems Laboratory, Pervasive Technology Labs, Indiana University, Bloomington, IN. See `http://www.lam-mpi.org/`.

[2] Jeffrey M. Squyres, Brian Barrett, and Andrew Lumsdaine. The system services interface (SSI) to LAM/MPI. Technical Report TR575, Indiana University, Computer Science Department, 2003.