Porting the LAM-MPI 6.3 Communication Layer

LAM Team
lam@mpi.nd.edu
March 8, 2000

1 Overview

The LAM-MPI library is written in two layers. The upper layer is portable and independent of the communication subsystem. It interfaces to the lower layer though the Request Progression Interface (RPI). The RPI consists of nine functions.

The portable upper layer interfaces to the lower layer in several places within the LAM-MPI library. In each case, this is done through a switch which invokes one of two versions of the RPI functions. One version of the RPI functions is itself portable and is intended to be present on every system. This version uses the LAM daemon for doing communication and will be referred to as the LAMD-RPI. Through the use of the LAM daemon, it provides excellent monitoring and debugging facilities at some cost in performance.

It is intended that the second version of the RPI functions, the client-to-client version, be targeted to specific communications architectures and provide high performance direct communication between MPI processes. It is this version that someone porting LAM to a new communication architecture must write. It will be referred to as the C2C-RPI.

There are currently three C2C RPIs that are provided with the LAM distribution: tcp, sysv, and usysv. The tcp C2C RPI uses internet domain sockets to communicate between clients. The sysv and usysv RPIs use shared memory to communicate between clients on the same machine, and internet domain sockets to communicate between clients on different hosts. The only difference between the sysv and usysv RPIs is that the sysv uses mutexes for locking, while usysv uses spin locks with a binary backoff.

The tasks that must be performed in the C2C-RPI 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 portable 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 portable layer.

The LAM-MPI library maintains various data structures. The most important of these, as far as the RPI is concerned, are the request list and the process list. The portable layer of LAM-MPI handles the high-level initialization and manipulation of these structures. The C2C-RPI can attach information to entries in these lists via handles.

For illustration, I will often refer to the tcp C2C-RPI provided with the LAM 6.3 distribution. As mentioned above, this version uses Internet domain TCP sockets as the communication subsystem and will be referred to as the TCP-RPI. It is implemented in the files rpi.tcp.h, rpi.tcp.c, tcp.low.c, and c2cbuf.c, which can be found in the share/h and share/mpi directories of the LAM 6.3 distribution.

It should be noted that the sysv and usysv RPIs also use all the same files, with the exception of rpi.tcp.c. Instead, they use rpi.sysv.c and rpi.usysv.c, respectively. The exact file that is chosen to be compiled is determined when the user runs configure.
This document should be read in conjunction with the header file `share/h/mpisys.h`, in which some of the data structures referred to are defined. See also Appendix A for a brief overview.

This document is in no way complete, but does cover the major areas of concern. Please send any comments, questions or suggestions, to lam@mpi.nd.edu.

2 Data Structures

2.1 Processes

An MPI process in LAM-MPI is represented by a structure of type “struct _proc”.

Each process in LAM-MPI maintains a linked list of all the processes that it knows about and can communicate with. The process itself appears in this list and the global variable `lam_myproc` is a pointer to it.

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

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

The process structure contains a field of type “void *” whereby C2C-RPI specific information can be attached to a process entry. This information is typically state information for the connection to/from the process. The TCP-RPI stores (among others things) the socket descriptor of the socket connection to the remote process and pointers to the requests (if any) currently reading from or writing to this socket.

2.2 Requests

For each process, the LAM-MPI library maintains a linked list of all the requests that need to be progressed. The portable layer keeps this progression list in order and also removes requests upon completion. C2C-RPI writers can thread other lists through the progression list via C2C-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 portable layer of the RPI stores information about the request in here, e.g., peer rank, message tag, message data-type. As with the process structure there is a handle for attaching C2C-RPI specific data. The TCP-RPI stores (among others things) the request envelope and a pointer into the data buffer indicating the current read/write position.

2.2.1 The Request Life-cycle

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 C2C-RPI, but this is not precluded. All that is required is that the request’s progression state be correctly set. See Sections 3.4 and 3.6 for more details. Depending on the C2C-RPI 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 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).

3 The RPI Calls

This section lists the seven C2C-RPI functions. It describes where they are called from and discusses briefly what the C2C-RPI specific layer is expected to do, and what the portable layer requires of it. All functions, apart from `rpi_c2c_advance()` and `rpi_c2c_iprobe()`, return 0 on success and the constant `LAMERROR` (−1) on error.

3.1 `int rpi_c2c_init(void)`

Performs primary initialization of the C2C-RPI sub-layer. It is called from MPI_Init(). In here, perform once-off initialization of the communication sub-layer and initialize all processes with respect to the communication sub-layer. The latter should usually just involve a call to `rpi_c2c_addprocs()` to initialize the initial set of “new” processes. TCP-RPI, for example, initializes a hash table for message buffering and then calls `rpi_c2c_addprocs()` to 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()` and `nrecv()` (see the LAM documentation and manual pages for more details) are available and can be used to pass information between the MPI processes. The TCP-RPI uses them to pass a server socket port number to clients who must connect. See the function `c2c_socketdance()` in share/mpi/rpi.c2c.c.

3.2 `int rpi_c2c_addprocs(void)`

In LAM-MPI, a process can become aware of new processes with which it may communicate. For example when it spawns MPI children. The portable layer adds new process entries to the process list and then calls `rpi_c2c_addprocs()` to perform C2C-RPI initialization. TCP-RPI, for example, sets up the TCP socket connections between the new processes and the old ones. This function is called from `MPI_Intercomm_create()`, `MPI_Comm_spawn()`, and `MPI_Comm_spawn_multiple()`.

3.3 `int rpi_c2c_build(MPI_Request req)`

When the portable layer creates a new request, it initializes portable request information and then calls this function to build the C2C-RPI portion of the request. Certain C2C-RPI 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()`.

3.4 `int rpi_c2c_start(MPI_Request req_top, MPI_Request req)`

The portable 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 `rpi_c2c_start()` so that the C2C-RPI can do any initialization it needs to make the request ready.
This function may also perform some progression past the start state. For example TCP-RPI handles 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.

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 cancelation, and
2. The done state where the data transfer protocol is finished and the request can be completed.

3.5 `int _rpi_c2c_destroy(MPI_Request req)`

Destroys the C2C-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 interface and also perform any other necessary cleanup.

**Note:** it is only necessary to clean up what was created/done in other parts of the C2C-RPI.

3.6 `int _rpi_c2c_advance(MPI_Request req_top, int flag_block)`

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 `flag_block` is true if it is permitted to block until some progress is made.

The portable layer knows and cares nothing about message transfer protocols and message buffering. This is solely the responsibility of the C2C-RPI. The C2C-RPI however must update the state of the request as it progresses from start, to active, to done, so that the portable layer can do the right thing.

Note that a request may be moved from the start to the done state outside of the regular RPI progression by being cancelled. The progression function `rpi_c2c_advance()` must take this into account. Currently we do not allow the cancelation of requests which are in the active state.

The C2C-RPI must also update other information in requests where appropriate. See Appendix A.2 for more details.

3.7 `int _rpi_c2c_iprobe(MPI_Request req)`

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 −1 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()`. I imagine that in most cases, a verbatim copy of the TCP-RPI version of this function should work.

3.8 `int _rpi_c2c_finalize(void)`

Performs final cleanup of the C2C-RPI subsystem. Clean up here all data structures etc. created by the C2C-RPI communication subsystem. This function is called from `MPI_Finalize()` after all pending communication has completed.
3.9 int _rpi_c2c_fastsend(char *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm)

This is a special case “short circuit” fast send that is invoked with the “-DSHORTCIRCUIT” flag while building LAM. Unless the user specifies “-without-shortcircuit” to configure, this flag is enabled automatically. 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 C2C mode is active, there are no active requests in the RPI, and the destination is not the same as the source. In this case, it is safe to bypass the normal RPI 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 LAM software) when the function returns. It must return MPI_SUCCESS or an 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.

3.10 _rpi_c2c_fastrecv(char *buf, int count, MPI_Datatype dtype, int src, int *tag, MPI_Comm comm, MPI_Status *stat, int *seqnum)

Like _rpi_c2c_fastsend, this function is intended to bypass the normal RPI, and is only called (from MPI_RECV) if there are no other active requests, and if the source of the message is neither MPI_ANY_SOURCE nor the destination. If the expected 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).
A Data Structures

In this section, the fields in the important data structures are discussed. In particular, a description is given, where appropriate, of when and with what they are initialized.

A.1 Process

```c
struct _proc {
    struct _gps p_gps; /* process GPS */
    void *p_rpi;       /* RPI specific data */
    int p_ger_nsnd;    /* #msgs sent there */
    int p_mode;        /* mode flags */
    #define LAM_PFLAG    0x001 /* used for marking */
    #define LAM_PDEAD    0x002 /* node died */
    #define LAM_PRPINIT  0x004 /* RPI initialized */
    #define LAM_PCLIENT  0x008 /* in client world */
    #define LAM_PHOMOG   0x010 /* homogeneous */
}
```

This holds the node identifier, PID, LAM kernel index, and the global rank of the process in the MPI_COMM_WORLD within which it was created.

Used by the LAM daemon RPI. Don’t touch.

A.2 Request

```c
struct _req {
    int rq_type;
    #define LAM_RQISEND  0
    #define LAM_RQIBSEND 1
    #define LAM_RQISSEND 2
    #define LAM_RQIRSEND 3
    #define LAM_RQIRECV  4
    #define LAM_RQIPROBE 5
    #define LAM_RQIFAKE  6
}
```

The type of request. They are classified in the order above as being either a send, buffered send, synchronous send, ready send, receive, probe or fake request. The fake request type is used only by the portable layer to handle the case of MPI_Ibsend() and the C2C-RPI will never see such a request of this type. Whether a request is of the blocking non-blocking variety is not indicated here but via the LAM_RQFBLKTYPE mark.
int rq_state;
#define LAM_RQSINIT 0 /* request initialized */
#define LAM_RQSSTART 1 /* active, nothing done yet */
#define LAM_RQSACTIVE 4 /* active, undone request */
#define LAM_RQSDONE 3 /* request done */

The request’s state of progression. See Section 2.2.1.

int rq_marks;      /* persistent flags */
#define LAM_RQFPERSIST 0x0001 /* persistent request */
#define LAM_RQFDYNBUF 0x0004 /* dynamic buffer */
#define LAM_RQFDYNREQ 0x0008 /* dynamic request */
#define LAM_RQFSOURCE 0x0200 /* source request */
#define LAM_RQFDEST 0x0400 /* destination request */
#define LAM_RQFBLKTYPE 0x0800 /* blocking request type */
#define LAM_RQFOSORIG 0x1000 /* origin 1-sided req. type */
#define LAM_RQFOSTARG 0x2000 /* target 1-sided req. type */

Flags (marks) marking persistent characteristics of the request. These characteristics do not change between successive invocations of a persistent request. These are set by the portable layer. The C2C-RPI should not modify them.

int rq_flags;      /* active req flags */
#define LAM_RQFCANCEL 0x0002 /* cancelled request */
#define LAM_RQFBLOCK 0x0010 /* blocking request */
#define LAM_RQFTRUNC 0x0040 /* truncated message */
#define LAM_RQFORPHAN 0x0100 /* destroy when done */
#define LAM_RQFCHAR 0x0100000 /* DRAWSG */
#define LAM_RQFINT 0x0200000 /* DINT4MSG */
#define LAM_RQFFLOAT 0x0400000 /* DFLT4MSG */
#define LAM_RQFDOUBLE 0x0800000 /* DFLT8MSG */
#define LAM_RQFSHORT 0x01000000 /* DINT2MSG */
#define LAM_RQFSHADOW 0x02000000 /* shadow req must finish */

Flags marking non-persistent characteristics of the request. The C2C-RPI must modify the LAM_RQFTRUNC flag when it is determined that the request is a receive that causes an MPI_ERR_TRUNCATE. Leave all other flags alone.

char *rq_packbuf;  /* pack buffer */

Pointer to start of message data to be sent or area where message data is to be received. Depending on the circumstances, this may or may not be the same as rq_buf which is a pointer to the buffer given by the user. The portable layer handles packing/unpacking etc. of this buffer.

int rq_packsize;   /* pack buffer size */

The size of the data to be sent/received in bytes. This is set by the portable layer. This is how much message data the C2C-RPI must send/receive for the request.
int rq_count; /* MPI request parameter */
void *rq_buf;  /* MPI request parameter */
MPI_Datatype rq_dtype; /* MPI request parameter */
int rq_rank;   /* MPI request parameter */
int rq_tag;    /* MPI request parameter */
MPI_Comm rq_comm;  /* MPI request parameter */

These record the arguments supplied by the user to the MPI function that created the request. Don’t modify.

int rq_cid;        /* context ID to use */

The context ID to use in the message envelope. It corresponds to the communicator rq_comm. See Appendix A.3.

int rq_func;       /* MPI func. which made req. */

The MPI function which created the request.

int rq_seq;        /* seq# of associated msg */

The message sequence number. If you want your RPI to work with LAM tracing, then this number must be sent with each message (it is set by the portable layer) and then on the receiving side, the C2C-RPI must extract it and set this field in the receiving request with its value.

int rq_f77handle;  /* f77 handle */

Handle used by fortran.

MPI_Status rq_status;   /* status info storage */

This holds the status of a request. See Appendix A.6. In the case of a receive request, the C2C-RPI 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.

struct _bsndhdr *rq_bsend;  /* ptr bsend header */

Pointer to the buffer header in the case of a buffered send. Used by portable layer only.

struct _proc *rq_proc;      /* ptr peer process */

Pointer to the peer process. This is initially set by the portable layer. In the case of a receive on MPI_ANY_SOURCE, it will be 0. Once the actual source has been determined, the C2C-RPI may set it to point to the peer process but is not required to do so.

struct _req *rq_next;       /* ptr next request */

Pointer to the next request in the list. Do not modify. If the C2C-RPI needs to maintain its own request lists it must do so through the C2C-RPI specific information handle.
void *rq_extra;    /* extra state */
A general place to hang extra state off the request.

int (*rq_handlr)();   /* handler function */
Function to invoke when a request has moved into the done state. Don’t touch this; it is used exclusively in
the portable layer (mostly by one-sided communication).

MPI_Request rq_shadow;    /* shadow IMPI requests */
“Shadow” requests used by IMPI. Don’t touch these; they are handled exclusively in the portable layer.

union rpi_req rq_rpi;    /* RPI specific stuff */
C2C-RPI-specific information on the request.

A.3 Communicator

This data structure is the internal representation of an MPI communicator object. Most of the fields are
self-explanatory and probably of no interest to C2C-RPI writers.

struct _comm {
    int c_flags;    /* properties */
#define LAM_CINTER 0x10    /* intercommunicator? */
#define LAM_CLDEAD 0x20    /* local group dead? */
#define LAM_CRDEAD 0x40    /* remote group dead? */

Flags indicating the status of the communicator.

LAM provides rudimentary mechanisms for fault tolerance. The C2C-RPI, upon detecting that a group
in a communicator is dead, can mark the communicator to indicate this. TCP-RPI does this, for example,
when it detects that a socket to a process it wants to communicate with is dead.

int c_contextid;    /* context ID */
Unique integer associated with the communicator. This is used in communication as the communicator
context. The portable layer sets the rq_cid field of the request structure with either a positive or negative
value of this according to whether point-to-point or collective operations are being done. Don’t use this field
directly, use rq_cid.

int cRefCount;    /* reference count */
MPI_Group c_group;    /* local group */
MPI_Group c_rgroup;    /* remote group */
HASH *c_keys;    /* keys cache hash table */
int c_cube_dim;    /* inscribing cube dim. */
int c_topo_type;    /* topology type */
int c_topo_nprocs;    /* # topo. processes */
int c_topo_ndims;    /* # cart. dimensions */
A.4 GPS

This data structure is used internally as the unique LAM system identification of a process.

```c
struct _gps {
    int gps_node;  /* node ID */
    int gps_pid;   /* process ID */
    int gps_idx;   /* process index */
    int gps_grank; /* glob. rank in loc. world */
}
```

The LAM nodeid of the node on which the process is running.

The process identifier on the node. This will usually be the UNIX PID.

The index of the process in its local LAM daemon’s process table. This is probably of no use to C2C-RPI writers.

The global rank of the process in the MPI_COMM_WORLD in which it was created.

A.5 Group

This data structure is the internal representation of an MPI group as maintained by an MPI process.

```c
struct _group {
    int g_nprocs;  /* # processes */
    int g_myrank;  /* my local rank */
}
```

The number of processes in the group.

The rank of the process within the group. If the process is not in the group, this will be MPI_UNDEFINED.
int g_refcount; /* reference count */
int g_f77handle; /* f77 handle */
struct _proc **g_procs; /* ptr to processes */

Pointer to an array of pointers to the processes in the group. The array is in order of rank in the group.

)

A.6 Status

This data structure is the internal representation of an MPI status object.

struct _status {
    int MPI_SOURCE;
    int MPI_TAG;
    int MPI_ERROR;

The transparent part of the MPI_Status object.

    int st_length; /* message length */

The amount of data (bytes) received by a receive. This must be set by the C2C-RPI once a receive is done.

)