Thoughts on the structure of an MPJ reference implementation

Mark Baker
Syracuse University

Bryan Carpenter
Syracuse University, Northeast Parallel Architectures Center, dbc@npac.syr.edu

Thoughts on the structure of an MPJ reference implementation

Mark Baker and Bryan Carpenter

CRPC-TR99803
October 1999

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Submitted November 1999

Thoughts on the structure of an MPJ reference implementation.

Mark Baker*and Bryan Carpenter
NPAC at Syracuse University
Syracuse, NY 13244
Mark.Baker@port.ac.uk,dbc@npac.syr.edu

October 28, 1999

Abstract

We sketch a proposed reference implementation for *MPJ*, the Java Grande Forum's MPI-like message-passing API [9, 3]. The proposal relies heavily on RMI and Jini for finding computational resources, creating slave processes, and handling failures. User-level communication is implemented efficiently directly on top of Java sockets.

*Current address: University of Portsmouth, UK

Contents

1 Introduction

The Message-Passing Working Group of the Java Grande Forum was formed about a year ago as a response to the appearance of several prototype Java bindings for MPI-like libraries. An initial draft for a common API specification was distributed at Supercomputing '98. Since then the working group has met in San Francisco and Syracuse. The nascent API is now called MPJ.

Presently there is no complete implementation of the draft specification. Our own Java message-passing interface, mpiJava, is moving towards the "standard". The new version 1.2 of the software supports direct communication of objects via object serialization, which is an important step towards implementing the specification in [3]. Once a few remaining open questions about the specification have been resolved, we will release a version 1.3 of mpiJava, implementing the new API. Most likely this will be the first "reference implementation" for that API, although some other groups have related efforts.

The mpiJava wrappers rely on the availability of platform-dependent native MPI implementation for the target computer. While this is a reasonable basis in many cases, the approach has some disadvantages

- The two-stage installation procedure—get and build native MPI then install and match Java wrappers—is tedious and probably off-putting to new users. Systems like MPICH made considerable strides in terms of ease of installation on diverse platforms, but historically software for parallel computing has been relatively hard to install and configure for different platforms. Presumably this has not facilitated its wider uptake. In any case, the "wrapper" approach to implementing MPJ aggravates matters because it adds one more step to this process.

- On several occasions in the development of mpiJava we saw conflicts between the JVM environment and the native MPI runtime behaviour. The situation has improved, and mpiJava now runs on various combinations of JVM and MPI implementation. Some problems remain. A persistent one relates to concurrent operations involving the same Java array. These *ought* to be allowed if the concurrent operations refer to disjoint sections of the array. But the way the Java Native Interface mechanism interacts with the data in Java arrays means that unexpected results may occur.

- Finally, this strategy simply conflicts with the ethos of Java, where pure-Java, write-once-run-anywhere software is the order of the day.

Ideally, the first two problems would be addressed by the providers of the original native MPI package. We envisage that they could provide a Java interface bundled with their C and Fortran bindings, avoiding the the headache of separately installing the native software and Java wrapper. Also they are presumably in the best position to iron-out low-level conflicts between the MPI library and the Java runtime. Hence we can only encourage vendors and other providers of MPI software to consider releasing Java wrapper software (which could be based on the public domain mpiJava, for example) along with their core software. Ultimately, such packages would probably represent the fastest, industrial-strength implementations of MPJ.

Meanwhile, to address the last shortcoming listed above, this report considers production of a *pure-Java* reference implementation for MPJ. Design goals are that the system should be as easy to install on distributed systems as we can reasonably make it, and that it be sufficiently robust enough to be useable in an Internet environment[1]. Ease of installation and use are special concerns to us. We want a package that will be useable not only by experienced researchers and engineers, but also in, say, an educational context. A corollary of easy installation is that the software should only depend on other systems that are widely installed. A minimum requirement is a Java development environment, including RMI. Beyond this we assume the installation of some Jini software. This technology is relatively new, but it seems likely that it will become pervasive in Java-aware environments.

We are by no means the first people to consider implementing MPI-like functionality in pure Java, and working systems have already been reported in [8, 6], for example. The goal here is to build on the some lessons learnt in those earlier systems, and produce software that is standalone, easy-to-use, robust, and fully implements the specification of [3].

This report

Section 2 reviews our design goals, and describes some decisions followed from these goals. Section 3 reviews the proposed architecture. Various distributed programming issues posed by computing in an unreliable environment are discussed in Section 4, which covers basic process creation and monitoring. This section assumes free use of RMI and Jini. Implementation of the message-passing primitives on top of Java sockets and threads is covered in 5.

Acknowledgements

We are extremely grateful to Glenn Judd and Kivanc Dincer, who freely made the sources of their Java MPI systems available to us. Various ideas from those systems have been adopted in the architecture described here, and the final implementation is likely borrow directly from those earlier systems.

2 Some design decisions

As noted above, an MPJ "reference implementation" can be implemented as Java wrappers to a native MPI implementation, or it can be implemented in pure Java. It could also be implemented principally in Java with a few simple native

[1] A particularly strong requirement is that in no circumstances must the software leave resource-wasting orphan processes lurking after an untidy termination. This unfortunate behaviour has plagued some implementations of MPI in the past.

methods to optimize operations (like marshalling arrays of primitive elements) that are difficult to do efficiently in Java. In this note we will focus on the latter possibilities—essentially pure Java, although experience with DOGMA and other systems strongly suggests that optional native support for marshalling will be desirable. The aim is to provide an implementation of MPJ that is maximally portable.

We envisage that a user will download a jar-file of MPJ library classes onto machines that may host parallel jobs. Some installation "script" (preferably a parameterless script) is run on the potential host machines. This script installs a daemon on those machines (probably by registering a persistent activatable object with an existing `rmid` daemon). Parallel java codes are compiled on any host. An `mpjrun` program invoked on that host transparently loads all the user's class files into JVMs created on remote hosts by the MPJ daemons, and the parallel job starts. The only required parameters for the `mpjrun` program should be the class name for the application and the number of processors the application is to run on. These seem to be an irreducible minimum set of steps; a conscious goal is that the user need do no more than is absolutely necessary before parallel jobs can be compiled and run.

In light of this goal one can sensibly ask if the step of installing a daemon on each host is essential. On networks of UNIX workstations—an important target for us—packages like MPICH avoid the need for special daemons by using the `rsh` command and its associated system daemon. Dispensing with the need for special installation procedures on target hosts would be a significant gain in simplicity, so this option needs serious consideration. In the end we decided this is probably not the best approach for us. Important targets, notably networks of NT systems, do not provide `rsh` as standard, and often on UNIX systems the use of `rsh` is complicated by security considerations. Although neither RMI or Jini provide any magic mechanism for conjuring a process out of nothing on a remote host, RMI does provide a daemon called `rmid` for restarting *activatable objects*. These need only be installed on a host once, and can be configured to survive reboots of the host. We propose to use this Java-centric mechanism, on the assumption that `rmid` will become as widely run across Java-aware platforms as `rshd` is on current UNIX systems.

An implementation ought to be fault-tolerent in at least the following senses. If a remote host is lost during execution, either because a network connection breaks or the host system goes down, or if the JVM running the remote MPJ task halts for some other reason (eg, occurrence of a Java exception), or if the process that initiated the MPJ job is killed—in any of these circumstances—*all* processes associated with the particular MPJ job must shut down within some (preferably short) interval of time. On the other hand, unless it is explicitly killed or its host system goes down altogether, the MPJ *daemon* on a remote host should survive unexpected termination of any particular MPJ job. Concurrent tasks associated with *other* MPJ jobs should be unaffected, even if they were initiated by the same daemon. These requirements likely put some restrictions on the portability of the daemon. They probably imply at least the ability to create a new JVM on demand, for example by using `Runtime.exec` to execute the `java` command. This facility is available in the major operating systems we target (UNIX and NT).

In the initial reference implementation we will probably use Jini technology[1, 7] to facilitate location of remote MPJ daemons and to provide a framework for the required fault-tolerance. This choice rests on our guess that in the medium-to-long-term Jini will become a ubiquitous component in Java installations. Hence using Jini paradigms from the start should eventually promote interoperability and compatibility between our software and other systems[2]. In terms of our aim to simplify *using* the system, Jini multicast discovery relieves the user of the need to create a "hosts" file defining where each process of a parallel job should be run. If the user actually *wants* to restrict the hosts, unicast discovery is available. Of course it has not escaped our attention that eventually Jini discovery may provide a basis for much more dynamic access to parallel computing resources.

Less fundamental assumptions bearing on the organization of the MPJ daemon are that standard output (and standard error) streams from all tasks in an MPJ job are merged non-deterministically and copied to the standard output of the process that initiates the job. No guarantees are made about other IO operations—for now these are system-dependent. Rudimentary support for global checkpointing and restarting of interrupted jobs would be useful, although we doubt that checkpointing would happen without explicit invocation in the user-level code, or that restarting would happen automatically[3].

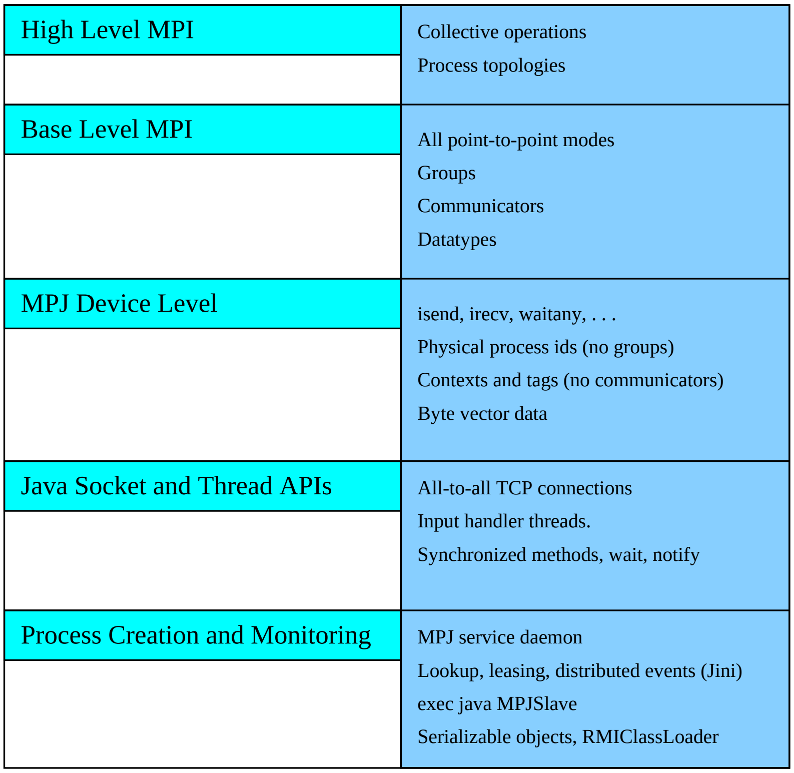
The main role of the MPJ daemons and their associated infrastructure is thus to provide an environment consisting of a group of processes with the user-code loaded and running, and running in a *reliable* way. As indicated above, the process group is reliable in the sense that no *partial failures* should be visible to higher levels of the MPJ implementation or the user code. As discussed above, partial failure is the situation where some members of a group of cooperating processes is unable to continue because other members of the group have crashed, or the network connection between members of the group has failed. To quote [11]: *partial failure is a central reality of distributed computing.* No software technology can guarantee the absence of *total* failures, in which the whole MPJ job dies at essentially the same time (and all resources allocated by the MPJ system to support the user's job are released). But total failure should be the *only* failure mode visible to the higher levels. Thus, to reiterate, a principal role of the base layer is to detect partial failures and cleanly abort the whole parallel program when they occur[4].

Once a reliable cocoon of user processes has been created through negotiation with the daemons, we have to establish connectivity. In the reference

[2] In the short-to-medium-term—before Jini software is widely installed—we might have to provide a "lite" version of MPJ that is unbundled from Jini. Designing for Jini protocols should, nevertheless, have a beneficial influence on overall robustness and maintainability. Use of Jini implies use of RMI for various management functions.

[3] Perhaps one could exploit the two-phase commit of the Jini transaction model to make checkpointing truly fault-tolerant...

[4] We notice that an MPJ job *as a whole* has some characteristics of a single Jini transaction. While interesting, this analogy is not clearly useful.



High Level MPI	Collective operations Process topologies
Base Level MPI	All point-to-point modes Groups Communicators Datatypes
MPJ Device Level	isend, irecv, waitany, . . . Physical process ids (no groups) Contexts and tags (no communicators) Byte vector data
Java Socket and Thread APIs	All-to-all TCP connections Input handler threads. Synchronized methods, wait, notify
Process Creation and Monitoring	MPJ service daemon Lookup, leasing, distributed events (Jini) exec java MPJSlave Serializable objects, RMIClassLoader

Figure 1: Layers of an MPJ reference implementation

implementation this will be based on Java sockets. Recently there has been interest in producing Java bindings to VIA [4, 12]. Eventually this may provide a better platform on which to implement MPI, but for now sockets are the only realistic, portable option. Between the socket API and the MPJ API there will be an intermediate "MPJ device" level. This is modelled on the abstract device interface of MPICH [10]. Although the role is slightly different here—we don't really anticipate a need for multiple device-specific implementations—this still seems like a good layer of abstraction to have in our design. The API is actually not modelled in detail on the MPICH device, but the level of operations is similar.

3 Overview of the Architecture

A possible architecture is sketched in Figure 1.

The bottom level, process creation and monitoring, incorporates initial negotiation with the MPJ daemon, and low-level services provided by this daemon, including clean termination and routing of output streams. The daemon invokes the MPJSlave class in a new JVM. MPJSlave is responsible for downloading the user's application and starting that application. It may also directly invoke routines to initialize the message-passing layer. Overall, what this bottom layer provides to the next layer is a reliable group of processes with user code installed. It may also provide some mechanisms—presumably RMI-based (we assume that the whole of the bottom layer is built on RMI)—for global synchronization and broadcasting simple information like server port numbers.

The next layer manages low-level socket connections. It establishes all-to-all TCP socket connections between the hosts.

The idea of an "MPJ device" level is modelled on the abstract device interface of MPICH. A minimal API includes non-blocking standard-mode send and receive operations (analogous to MPI_ISEND and MPI_IRECV, and various wait operations—at least operations equivalent to MPI_WAITANY and MPI_TESTANY). All other point-to-point communication modes can be implemented correctly and with reasonable efficiency on top of this minimal set. Unlike the MPICH device level, we do not incorporate direct support for groups, communicators and datatypes at this level (but we do assume support for message contexts). Message buffers will be byte arrays. The device level is intended to be implemented on socket send and recv operations, using standard Java threads and synchronization methods to achieve its richer semantics.

The next layer is base-level MPJ, which includes point-to-point communications, communicators, groups, datatypes and environmental management. On top of this are higher-level MPJ operations including the collective operations. We anticipate that much of this code can be implemented by fairly direct transcription of the src subdirectories in the MPICH release—the parts of the MPICH implementation above the abstract device level.

4 Process creation and monitoring

We assume that an MPJ program will be written as a class that extends MPJApplication. To simplify downloading we assume that the user class also implements the Serializable interface. The main program will be implemented as the an *instance* method main:

```
class MyMPJApp extends MPJApplication {
  public void main(String [] args, Comm world) {...}
}
```

The default communicator is passed as an argument to main. Note there is no equivalent of MPI_INIT or MPI_FINALIZE. Their functionality is absorbed into code executed before and after the user's main method is called[5].

In a perfect world we might execute MyMPJApp by a command like

[5] This is a change to the API of mpiJava [2], for example, where the main method is static and the default communicator is a class variable. The approach here (which follows more closely DOGMA [8] or JMPI [5]) seems to fit more naturally with RMI, and allows for the possibility of running several MPJ processes as threads in a single JVM, although probably that won't be supported in the initial reference implementation.

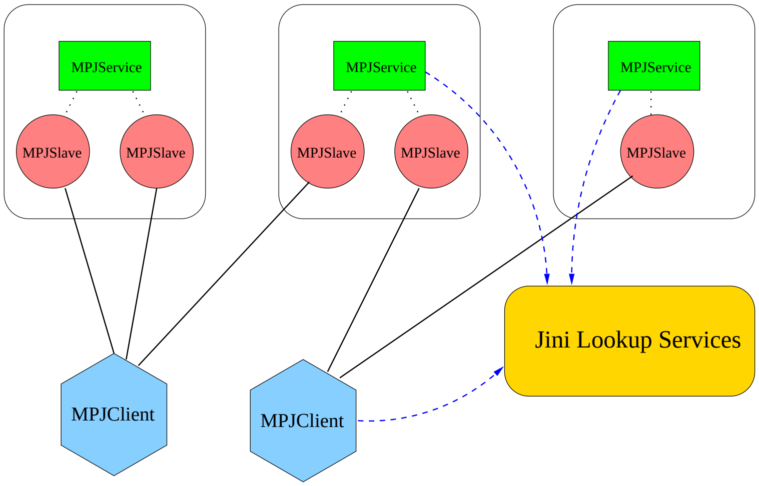


Figure 2: Independent clients may find `MPJService` daemons through the Jini lookup service. Each daemon may spawn several slaves.

```
  java MyMPJApp -np 8
```

where the `-np` option specifies the number of processors on which the program is to execute. This isn't quite practical, because there is no obvious way for a generic *static* `main` method (defined in the base class `MPJApplication`) to discover the actual subclass that the `java` command was started with[6]. So it cannot dispatch instances of `MyMPJApp` to remote machines. Probably we have to settle instead for

```
  java MPJClient MyMPJApp -np 8
```

where now `MPJClient` is a separate library class that is responsible for starting instances of the `MyMPJApp` on 8 remote machines.

4.1 The MPJ daemon

The MPJ daemon must be installed on any machine that can host an MPJ process. It will be realized as an instance of the class `MPJService`. It is likely to be an activatable remote object registered with a system `rmid` daemon[7]. The MPJ daemon executes the Jini discovery protocols and registers itself with available Jini lookup services, which we assume are accessible as part of the standard system environment (Figure 2).

The API of `MPJService` includes a `createSlave` remote method call, along the lines:

[6] The `args` array passed to `main` only holds command-line arguments *after* `MyMPJApp`.

[7] Using an activatable object is not essential, but it can reduce resources consumed by a daemon that is not in use, and provides an automatic way for the daemon to be restarted after crashes of the host system.

```
class MPJService extends Remote {
  public MPJSlave createSlave(MPJClient client, ...)
                                throws RemoteException {...}
}
```

In normal operation, a call to `createSlave` will behave essentially as:

```
int slaveID = SlaveTable.allocateID() ;

String cmd = "java MPJSlaveImpl " + slaveID + " " + registryPort ;
Process child = Runtime.getRuntime().exec(cmd) ;

... fork a monitor thread

SlaveTable.waitFor(slaveID) ;  // Wait for call-back from slave.

return SlaveTable.getSlaveObject(slaveID) ;
```

The data structure `SlaveTable` is a table of slave processes currently managed by the daemon. The daemon passes the id of the new slave into the `java` command that starts the slave running. We assume the daemon is running an RMI registry, in which it publishes itself. The port of this registry is passed to the slave as a second argument. The first actions of the slave object are to look up its master in the registry, then call back to the master and install a remote reference to itself (the slave) in the master's slave table[8]. The monitor thread in the daemon behaves essentially as:

```
DataInputStream stdout = new DataInputStream(child.getInputStream()) ;

// Forward standard output from child
String line ;
while ((line = stdout.readLine()) != null)
  client.println(line) ;

child.waitFor() ;
```

Output is multiplexed to the client by calling a remote `println` method on the client.

The net effect is that the client receives a remote reference to a new slave object running in a private JVM. In practise a remote `destroySlave` method that invokes the `Process.destroy` method will probably be needed as well.

4.2 The MPJ slave

The implementation class associated with the `MPJSlave` interface normally behaves as follows (schematically):

```
public class MPJSlaveImpl extends UnicastRemoteObject {
  public static void main(String args []) {
```

[8] *Not* its RMI registry!

```
    int slaveID       = Integer.parseInt(args [0]) ;
    String masterPort = args [1] ;

    MPJService master =
            (MPJService) Naming.lookup("rmi://localhost:" + args [1] +
                                       "/MPJService") ;

    master.addSlave(int slaveID, int new MPJSlaveImpl()) ;
  }

  public runTask(MPJApplication task, String [] args, ...)
                               throws RemoteException {

    ... create default communicator, ‘world’

    task.main(world, args) ;
  }
}
```

The `main` method creates a remote object and “registers” it with its daemon by calling a remote method `addSlave` on the master. Later the client calls back with the `runTask` method, passing an instance of the actual user class. Because this is a serializable object it is passed by value to the remote `runTask` method. Importantly, the byte code for the user class will be loaded by `RMIClassLoader` from the code-base specified in the serialized object. As discussed below, this will be the URL of a process serving a (typically very much stripped-down) subset of the HTTP protocol.

Hence, using the dynamic class-loading mechanisms provided as standard in RMI, we ensure that all user code is automatically available to the remote host.

4.3 The MPJ client

In pseudocode, the normal behaviour of the client is:

```
create an ‘MPJClient’ remote object for call-back by slaves

discover Jini lookup services and create table, ‘daemons’,
    of P remote references to suitable ‘MPJService’ objects

for i = 0..P-1 do {
  slaves [i] = daemons [i].createSlave(clientObject, ...) ;
}

create an instance, ‘task’, of user’s ‘MPJApplication’ class

for i = 0..P-1 in parallel threads do {
  slaves [i].runTask(task, args) ;
}

destroy slaves
```

The client must arrange for any byte code on the current CLASSPATH to be available via HTTP from a URL specified in the `rmi.server.code.base` property of the client JVM. In the usual way, this URL will be embedded in the serialized `task` object passed to the slave. A likely arrangement is for the client process itself to serve the necessary parts of the HTTP protocol.

In the normal case, the P threads terminate when the remote `runTask` methods all complete. The MPJ client process then terminates. As mentioned earlier, the client object provides a remote `println` method, which simply copies its argument to `System.out`.

4.4 Handling MPJ aborts—Jini events

If any slave JVM terminates unexpectedly while the `runTask` method is in progress, a `RemoteException` will be passed to the thread that started the remote call. The thread should catch the exception, and generate an `MPJAbort` event. This is a Jini remote event—a subclass of `RemoteEvent`. Early in the process of creating a slave, the MPJ daemons will have registered themselves with the client as handlers for `MPJAbort` events. Their `notify` method will apply the `destroy` method to the appropriate slave `Process` object.

Hence if any slave aborts (while the network connection stays good), all remaining slave processes associated with the job are immediately destroyed.

4.5 Other failures—Jini leasing

The distributed event mechanism can rapidly clean up processes in the case where some slaves disappear unexpectedly, but it cannot generally reclaim resources in the case where the client process is killed during execution of an MPJ job, or the daemon process is killed while it has some active slaves, or in the case of network failures that don't directly affect the client. There is a danger that orphaned slave processes will be left running in the network.

The solution is to use the Jini leasing paradigm. The client leases the services of each daemon for some interval, and continues renewing leases until all slaves terminate, at which point it cancels its leases. If the client process is killed (or it connection to the slave machine fails), its leases will expire. If a client's lease expires the daemon applies the `destroy` method to the appropriate slave `Process` object.

If a user program deadlocks, it is assumed that the user eventually notices this fact and kills the client process. Soon after, the client's leases expire, and the orphaned slaves are destroyed. We anticipate that lease periods will be relatively short by Jini standards—perhaps on the order of 60 seconds.

This doesn't deal with the (presumably less common) case where a daemon is killed while it is servicing some MPJ job, but the slave continues to run. To deal with this case a daemon may lease the service of its own slave processes immediately after creating them. Should the daemon die, its leases on its slaves expire, and the slaves self-destruct.

5 Sketch of a "Device-Level" API for MPJ

In this section we turn to the issue of how to implement MPJ once a reliable group of processes has been established. Whereas the previous section was concerned with true *distributed programming* where partial failure an overriding concern, this section is, properly speaking, concerned with *concurrent programming* within a single JVM, and (to a lesser extent, as it happens) message-passing *parallel programming* in a reliable environment. These are three somewhat distinct software disciplines.

We assume that the MPJ user-level API will be implemented on top of a "device-level" API, roughly corresponding to the MPID layer in MPICH. The following properties are considered to be desirable for the device-level API:

1. It should be implementable on the standard Java API for TCP sockets. In the absence of `select`, this essentially forces introduction of at least one receive thread for each input socket connection.

2. It should be efficiently implementable (and probably will be implemented) with precisely this minimum required number of threads.

3. It should be efficiently implementable with at least two protocols:

   a) The naive eager-send protocol, assuming receiver threads have unlimited buffering.

   b) A ready-to-send/ready-to-receive/rendezvous protocol requiring receiver threads only have enough buffering to queue unserviced "ready" messages.

4. The basic operations will include `isend`, `irecv` and `waitany` (plus some other "wait" and "test" operations). These suffice to build legal implementations of all the MPI communication modes. Optimized entry points for the other modes can be added later.

5. (Probably) all handling of groups and communicators will be outside the device level. The device level only has to correctly interpret absolute process ids and integer contexts from communicators.

6. (Maybe) all handling of user-buffer datatypes is outside the device level. The device level only deals with byte vectors.

5.1 Minimal API

The methods `isend` and `irecv` return communication request objects. A set of these request objects can then be passed to the `waitany` method, which waits until one of them completes. In principle any number of user threads are allowed (but we assume that a particular request object will not appear concurrently in `waitany` calls being executed in different threads).

5.2 Implementation notes

A communication request is *pending* if the communication has not yet started.

As a matter of taste, the implementations of the minimal API sketched here do not use polling to implement their "wait" methods. If a `waitany` method specifying a particular request has been invoked, a *wait-object* may be associated with that request. Any wait-object provides a `synch()` method, which implements barrier synchronization between precisely two threads. This can be implemented as follows:

```
class Wait {

  void synchronized int synch() {
    if(waiting) {
      waiting = false ;
      notify() ;
    }
    else {
      waiting = true ;
      wait() ;
    }
  }

  boolean waiting = false ;
}
```

Wait-objects are used for synchronization between input-handlers and user threads. In practise wait-objects will contain extra fields relating to nominated and selected sets of request objects, and these fields will provide a channel of communication between input-handlers and user threads.

Besides wait-objects, the principle means of synchronization is mutual exclusion on a single lock that controls access to the *communication sets*—data structures describing the ongoing communications. The communication sets include the *input-buffer* and the *pending-request-set*. The input-buffer contains messages that have been accepted by the input handlers, but not yet consumed by the user threads. Depending on the protocol, the input-buffer may hold request-to-send messages and/or complete messages containing user data. The pending-request-set, as its name implies, is the set of communication request objects that are currently pending.

The input-handlers are threads—one per input socket connection. These handle all input from sockets. All output to sockets occurs in the context of user threads.

5.3 Eager send protocol

Messages are sent immediately by `isend`, assuming unlimited space for data in the input buffer. In practise this protocol is most suitable for short messages.

```
isend:
  send the message

  return a non-pending (completed) request object
```

Figure 3: Pseudocode for `isend` method (eager protocol)

```
irecv:
  lock communication-sets
    if irecv matches some message in the input buffer {
      copy data into user-buffer
      acquire a non-pending (completed) request object
    }
    else
      put a request object in the pending-request-set
  unlock communication-sets

  return the request object
```

Figure 4: Pseudocode for `irecv` method (eager protocol)

```
waitany:
  lock communication-sets
    if all of the specified set of requests are pending
      associate one wait-object with all specified requests
  unlock communication-sets

  if all the requests were pending
    'synch()' on wait-object
  else
    select one of the non-pending requests
```

Figure 5: Pseudocode for `waitany` method (eager protocol)

```
loop {
  receive header

  lock communication-sets
    if message matches some request in pending-request-set {
      receive data into user-buffer

      remove the request from the pending-request-set

      if the request has an associated wait-object {
        dissociate that wait-object from all requests

        ‘synch()’ on wait-object
      }
    }
    else
      receive data into input-buffer
  unlock communication-sets
}
```

Figure 6: Pseudocode for input handler threads (eager protocol)

5.4 Rendezvous protocol

This assumes the protocol:

1. source sends ready-to-send
2. destination sends ready-to-receive
3. data is exchanged

Data is never buffered, although ready-to-send messages may be. This protocol is likely to be more efficient for long messages, because it eliminates the need to copy data from input-buffer to user space. A ready-to-receive message can include an identifier for the request object at the receiving end. The sending end can reflect this id in the header of the data packet, allowing the input handler at the receiving end to retrieve the relevant request when the data arrives.

```
isend:
  lock communication-sets
    put a send request object in the pending-request-set
  unlock communication-sets

  send a ready-to-send message

  return the request object
```

Figure 7: Pseudocode for `isend` method (rendezvous protocol)

```
irecv:
  lock communication-sets
    if irecv matches a ready-to-send message in the input buffer
      acquire a non-pending request object
    else
      put a receive request object in the pending-request-set
  unlock communication-sets

  return the request object
```

Figure 8: Pseudocode for `irecv` method (rendezvous protocol)

```
waitany:
  lock communication-sets
    if all of the specified set of requests are pending
      associate one wait-object with all specified requests
  unlock communication-sets

  if all the requests were pending
    ‘synch()’ on wait-object
  else
    select one of the non-pending requests

  if selected request was a receive request {
    associate a wait-object with the request

    send a ready-to-receive message (containing id of request)

    ‘synch()’ on wait-object (waiting for data)
  }

  if selected request was a send request
    send the data
```

Figure 9: Pseudocode for `waitany` method (rendezvous protocol)

```
input handler threads:
  loop {
    receive header

    lock communication-sets

      if message is a ready-to-send message {
        if header matches some receive request in pending-request-set {
          remove the matching request from the pending-request-set

          if the request has an associated wait-object {
            dissociate the associated wait-object from all requests

            ‘synch()’ on wait-object
          }
        }
        else
          put the ready-to-send message in the input buffer
      }

      if message is a ready-to-receive message {
        remove the matching request from the pending-request-set

        if the request has an associated wait-object {
          dissociate that wait-object from all requests

          ‘synch()’ on wait-object
        }
      }

      if message is data {
        receive the data

        ‘synch()’ on wait-object in request (identified in header)
      }

    unlock communication-sets
  }
```

Figure 10: Pseudocode for input handler threads (rendezvous protocol)

References

[1] Ken Arnold, Bryan O'Sullivan, Robert W. Scheifler, Jim Waldo, and Ann Wollrath. *The Jini Specification.* Addison Wesley, 1999.

[2] Mark Baker, Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Xinying Li. mpiJava: A Java interface to MPI. In *First UK Workshop on Java for High Performance Network Computing, Europar '98*, September 1998. http://www.cs.cf.ac.uk/hpjworkshop/.

[3] Bryan Carpenter, Vladimir Getov, Glenn Judd, Tony Skjellum, and Geoffrey Fox. MPI for Java: Position document and draft API specification. Technical Report JGF-TR-3, Java Grande Forum, November 1998. http://www.javagrande.org/.

[4] Chi-Chao Chang and Thorsten von Eiken. Interfacing Java to the Virtual Interface Architecture. In *ACM 1999 Java Grande Conference.* ACM Press, June 1999.

[5] George Crawford III, Yoginder Dandass, and Anthony Skjellum. The JMPI commercial message passing environment and specification: Requirements, design, motivations, strategies, and target users.
http://www.mpi-softtech.com/publications.

[6] Kivanc Dincer. jmpi and a performance instrumentation analysis and visualization tool for jmpi. In *First UK Workshop on Java for High Performance Network Computing, Europar '98*, September 1998. http://www.cs.cf.ac.uk/hpjworkshop/.

[7] W. Keith Edwards. *Core Jini.* Prentice Hall, 1999.

[8] Glenn Judd, Mark Clement, and Quinn Snell. DOGMA: Distributed object group management architecture. In *ACM 1998 Workshop on Java for High-Performance Network Computing. Palo Alto, February 1998*, volume 10(11-13) of *Concurrency: Practice and Experience*, 1998.

[9] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard.* University of Tenessee, Knoxville, TN, June 1995. http://www.mcs.anl.gov/mpi.

[10] MPICH—a portable implementation of MPI.
http://www.mcs.anl.gov/mpi/mpich/.

[11] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. Technical Report SMLI TR-94-29, Sun Microsystems Laboratories, 1994. Reprinted in [1].

[12] Matt Welsh. Using Java to make servers scream. Invited talk at ACM 1999 Java Grande Conference, San Francisco, CA, June, 1999.