1987

MPJ: A Proposed Java Message Passing API and Environment for High Performance Computing

Mark Baker
Syracuse University

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

MPJ: A Proposed Java Message Passing API and Environment for High Performance Computing

Mark Baker
University of Portsmouth
Hants, UK, PO1 2EG
Mark.Baker@port.ac.uk

Bryan Carpenter
NPAC at Syracuse University
Syracuse, NY 13244, USA
dbc@npac.syr.edu

January 24th 2000

Abstract

In this paper we sketch out a proposed reference implementation for message passing in Java (MPJ), an MPI-like API from the Message-Passing Working Group of the Java Grande Forum [1,2]. 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.

1. Introduction

The Message-Passing Working Group of the Java Grande Forum was formed late 1998 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 present API is now called MPJ.

Currently 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 [1]. We will release a version 1.3 of mpiJava, implementing the new API.

The mpiJava wrappers [2] 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 the Java wrappers – is tedious and off-putting to new users.
- 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.
- 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 headache of separately installing the native software and Java wrapper. Also they are presumably in the best position to iron-out low-level conflicts and bugs. Ultimately, such packages should represent the fastest, industrial-strength implementations of MPJ.

Meanwhile, to address the last shortcoming listed above, this paper considers production of a pure-Java reference implementation for MPJ. The design goals are that the system should be as easy to install on distributed systems as we can make it, and that it be sufficiently robust to be useable in an Internet environment. 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.

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 [3, 4], 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 [1].

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 section 5.

2. Some design decisions

A 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 methods to optimize operations (like marshalling arrays of primitive elements) that are difficult to do efficiently in Java. Our proposed system focuses on the latter possibility – essentially pure Java, although experience with DOGMA [3] 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 and requires the minimum of support for anomalies found in individual systems.

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 host machines. This script installs a daemon (perhaps 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. In the end we decided this is not the best approach for us. Important targets, notably networks of NT workstations, 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 optimistic assumption that `rmid` will become as widely run across Java-aware platforms as `rshd` is on current UNIX systems.

In the initial reference implementation it is likely that we will use Jini technology [5, 6] 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. In terms of our aim to simplify *using* the system, Jini multicast discovery relieves the user of the need to create a "hosts" file that defines where each process of a parallel job should be run. If the user actually *wants* to restrict the hosts, a unicast discovery method 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.

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 in a *reliable* way. 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.A partial failure is the situation where some members of a group of cooperating processes are unable to continue because other members of the group have crashed, or the network connection between members of the group has failed. To quote [7]: *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 a principal role of the base layer is to detect partial failures and cleanly abort the whole parallel program when they occur.

Once a reliable cocoon of user processes has been created through negotiation with the daemons, we have to establish connectivity. In the reference implementation this will be based on Java sockets. Recently there has been interest in producing Java bindings to VIA [8, 9]. 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 do not really anticipate a need for multiple device-specific implementations. 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 – 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 (ADI) 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 on top of this minimal set. Unlike the MPICH device level, we do not incorporate direct support for groups, communicators or (necessarily) datatypes at this level (but we do assume support for message contexts). Message buffers are likely to 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.

3.1 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 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.

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

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

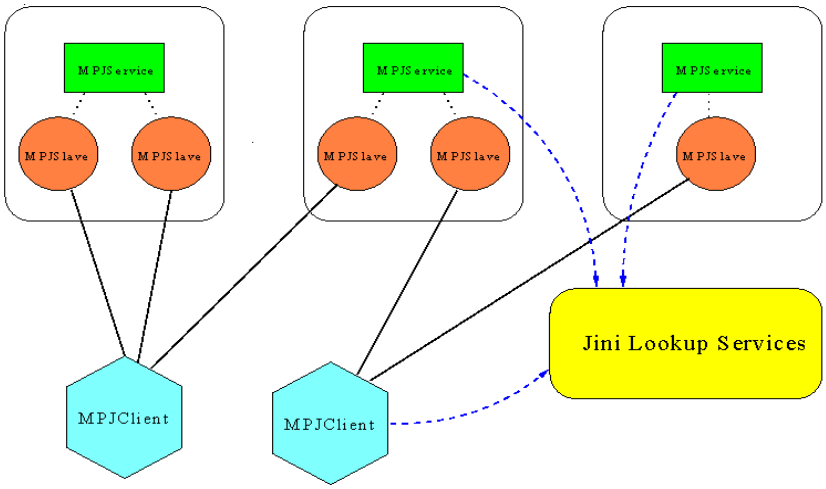

Figure 2: Independent clients may find `MPJService` daemons through the Jini lookup service. Each daemon may spawn several slaves.

3.3 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 intact), all remaining slave processes associated with the job are immediately destroyed.

3.4 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,
- the daemon process is killed while it has some active slaves,
- the case of network failures that do not 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. This does not deal with the 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.

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

Whereas the previous section was concerned with true *distributed programming* where partial failure is the overriding concern, this section is mainly concerned with *concurrent programming* within a single JVM – providing a reliable environment. 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 the introduction of at least one receive thread for each input socket connection.
2. It should be efficiently implementable (and 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. It is probable that 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. It may be necessary that all handling of user-buffer datatypes is outside the device level – here the device level only deals with byte vectors.

4. Conclusions and Future Work

In this paper we have discussed the findings from preliminary research into the design and implementation issues for producing a reference message passing interface for Java (MPJ). Our proposed design is based on our experiences from creating and then supporting the widely used `mpiJava` wrappers, experimental prototypes, and from on-going discussions with the Message-Passing Working Group of the Java Grande Forum. Our preliminary research has highlighted a number of design issues that the emerging Java technology, Jini, can help us address. In particular the areas of application distribution, resource discovery and fault tolerance. Issues that other message passing systems typically fail to address. Apart from a few minor questions about the exact syntax of the MPJ API, the design of our reference MPJ environment is complete and preliminary implementation work is underway. We believe that with our current man power we will be able to report on a limited release by the time that the workshop takes place in early May 2000.

5. References

1. B. Carpenter, et al. MPI for Java: Position Document and Draft API Specification, Java Grande Forum, JGF-TR-3, Nov. 1998, `http://www.javagrande.org/`
2. M. Baker, D. Carpenter, G. Fox, S. Ko and X. Li, `mpiJava`: A Java MPI Interface, to appear in the *International Journal Scientific Programming* – ISSN 1058-9244, `http://www.cs.cf.ac.uk/hpjworkshop/`
3. G. Judd, et. al., DOGMA: Distributed Object Group Management Architecture, ACM 1998 Workshop on Java for High-Performance Network Computing. Palo Alto, February 1998, *Concurrency: Practice and Experience*, 10(11-13), 1998,
4. K. Dincer, jmpi and a Performance Instrumentation Analysis and Visualization Tool for jmpi, *First UK Workshop on Java for High Performance Network Computing, Europar '98*, September 1998, `http://www.cs.cf.ac.uk/hpjworkshop/`
5. K. Arnold et. al., The Jini Specification, *Addison Wesley*, 1999
6. W. Edwards, Core Jini, *Prentice Hall*, 1999
7. J. Waldo, et. al., A Note on Distributed Computing, *Sun Microsystems Laboratories*, SMLI TR-94-29, 1994
8. M. Welsh, Using Java to Make Servers Scream, Invited talk at ACM 1999 Java Grande Conference, San Francisco, CA, June, 1999
9. C. Chang and T. von Eiken, Interfacing Java to the Virtual Interface Architecture, ACM 1999 Java Grande Conference, June, 1999, ACM Press
10. MPICH – A Portable Implementation of MPI, `http://www.mcs.anl.gov/mpi/mpich/`
11. MPI Forum, MPI: A Message-Passing Interface Standard, University of Tennessee, Knoxville, TN, USA, June 1995, `http://www.mcs.anl.gov/mpi`
12. G. Crawford, Y. Dandass and A. Skjellum, The JMPI Commercial Message Passing Environment and Specification: Requirements, Design, Motivations, Strategies, and Target Users, `http://www.mpi-softtech.com/publications`