C. Project Description

C.1 Introduction

This proposal is especially concerned with enabling parallel computation, and the associated communications, in a world dominated by Internet technologies. We will not assume that parallel computation is necessarily distributed *across* the Internet, although of course this is one possibility that has been widely discussed in the name of metacomputing. We do assume that the software and hardware technologies that will be readily available in the immediate future—the commodity technologies—will be fine-tuned for the Internet environment. On the hardware side these will include parallel—perhaps massively parallel—engines designed and deployed as Internet servers. On the software side, they will include software developed in network-aware programming languages like Java—software engineered to survive in heterogeneous and very dynamic environments.

Over the last few years the prospect of using Java for essentially "scientific" computing has has become increasingly realistic. Ongoing activities in the *Java Grande Forum*—complemented by work in academic and industrial sectors on optimizing compilers, JITs, language enhancements and libraries—have helped close an initial credibility gap. It is increasingly accepted that Java environments will meet the performance constraints needed to support large-scale computations and simulations. The work on improving the performance of Java is driven largely by its industrial application as a programming language for high-performance Internet servers. Scientific programmers will also reap the benefits.

One of the most influential developments in parallel computing over the last decade was the publication in 1994 of the Message Passing Interface (MPI) standard [22]. The idea of an agreed standard API for communication in parallel programs was relatively slow in coming. As a result it benefitted from a great deal of accumulated experience from application developers using earlier, proprietary APIs. MPI supports the Single Program Multiple Data (SPMD) model of parallel computing, providing many modes of reliable point-to-point communication, and a library of true collective operations. An extended standard, MPI 2, incorporates additional features like dynamic process creation and one-sided access to memory in remote processes.

The MPI standards specify language bindings for Fortran, C and C++. None of these languages is especially adapted to the Internet, where downloadable and mobile code are norms, resources (including computational resources) may be discovered and lost spontaneously, and fault tolerance is a crucial issue. In the proposed work we will be concerned with using network-oriented languages for high-performance computing. For now this means Java. One immediate preoccupation is with refinement of MPI-like programming models and APIs for high performance programming in Java—researching ways to get the fastest possible message passing from Java, and ways to exploit novel Java technologies like Jini to produce richer message-passing environments. A complementary concern is with use of Jini in a middle tier between client and MPI-based parallel services.

Java introduces implementation issues for message-passing APIs that do not occur in conventional scientific programming languages. One area of research is how to transfer

data between the Java program and the network while reducing overheads of the Java Native Interface. We will investigate how to apply ideas from projects like Jaguar [35] and JaVIA [9] to MPI-like APIs. Another important issue is how to minimize the overheads of serialization in communicating Java objects and multidimensional arrays. We will integrate ideas on efficient object serialization from the KaRMI project [29], for example, with MPI-specific ideas we started to explore in [7]. We will be especially interested in supporting efficient communication of the scientific Array classes supported by the Java Grande Numerics Working Group.

Just providing efficient MPI-like APIs for Java is not enough. The programming model must address features specific to distributed computing. MPI 1 was designed with relatively static platforms in mind. To better support computing in volatile Internet environments, we will need (at least) features from MPI 2 like dynamic spawning of process groups and parallel client/server interfaces. A natural framework for dynamically discovering new compute resources and establishing connections between running programs already exists in Sun's Jini project. In the proposed work, an important emphasis will be on researching synergies between parallel message-passing programming and Jini-like systems. One defining characteristic of distributed computing is the presence of *partial failures*. By combining ideas from MPI with ideas from Jini we aim to create an environment that encourages scalable, fault-tolerant parallel computing.

Finally we will explore uses of Jini in a middle tier for initiating parallel MPI jobs, where parallel program may be written in Java, or some other language that uses MPI-style message passing.

C.2 Motivation of the proposed work

C.2.1 Parallel computing and Java

To realize its full potential parallel computing will have to adapt itself to the Internet environment by embracing current Internet technologies. Many people accept that the Java language and accompanying technologies are likely to continue as major influences on the development of Internet software. But the idea that Java should also be adopted as an important language for large-scale technical computations has met some resistance. The most serious objection has been the perceived *inefficiency* of the Java language when compared with more mature languages like Fortran.

Over the last three years supporters of the *Java Grande Forum* have been working actively to address some of the difficulties. The official goal of the forum has been to develop consensus and recommendations on possible enhancements to the Java language and associated Java standards, for large-scale ("Grande") applications. Through a series of ACM-supported workshops and conferences the forum has also helped stimulate research on Java compilers and systems, and helped lay to rest some of the doubts about the potential performance of Java systems. An interesting series of papers from IBM [24, 25, 37], for example, confirmed that the current generation of Java virtual machines have rather poor performance on Fortran-like, array-intensive computations, but went on to demonstrate how to apply aggressive optimizations in Java compilers to obtain performance competitive with Fortran. In a recent paper [26] they described a case study involving a data mining

application that used the Java Array package supported by the Java Grande Numerics Working Group. Using the experimental IBM HPCJ Java compiler they reported obtaining over 90% of the performance of Fortran.

The Java Grande Forum also has a Concurrency and Applications Working Group. This group has been looking directly at uses of Java in parallel and distributed computing. Participants have studied various approaches, and we will refer to several of these in the following sections. The proposed will be particularly investigating issues relating to adoption of message-passing parallel programming in Java.

C.2.2 Niches for parallel Java programs

Computers that host major Web sites will either be multiprocessors or clusters of workstations. Many are now, and this trend will presumably continue. Increasingly these servers are programmed in Java. Since these technologies—Java and parallel computers—will coexist in Internet servers, this is fertile place to see roles for Java-based parallel computation emerging. Along with all the other commericial services that will appear on the Web in the near future we are likely to see compute-intensive ones—maybe supporting data-mining queries using parallel algorithms or financial analysis programs involving complex simulations.

Truly scalable servers are likely to be clusters rather than symmetric multiprocessors. As a specific example, consider the *Ninja* vision of the future of the Internet elaborated by researchers at UC Berkeley [32]. In their view a service should be *scalable* (able to support thousands of concurrent users), *fault-tolerant* (able to mask faults in the underlying server hardware), and highly-available. A major concern is with mobile code for service deployment—specialized active proxies that migrate out across the Internet to position themselves close to client devices. But services must maintain persistent state, and the architects of Ninja conclude that distributed, wide-area management of this state is generally intractable. "Hard", persistent state is maintained in a carefully-controlled environment—the *Base*—engineered to provide high availability and scalability. This is assumed to be a cluster of workstations with fast, local communication, a controlled environment, and a single administrative domain [16]. It is not necessarily homogeneous and it is not completely reliable, so it is not exactly a conventional parallel computer. But this is an environment where we might expect message-passing parallel programs written in Java to appear, either to implement specific Internet services or just because scientific programmers exploit them on account of their availability—parallel Internet servers become commodity hardware.

A completely different place where we might see early uptake Java-based parallel computing is in the classroom. Java has become an important teaching language in Universities. For teaching parallel computing principles to students, Java is likely to be a more attractive language than Fortran. On the basis of project descriptions given when people download our *mpiJava* software, for example, we estimate that perhaps 10% of potential users are teachers looking for classroom software. This is a not a dominant proportion, but it is an especially influential one so far as future uptake is concerned. In this context highly tuned implementations are not essential. An MPI-like package that is portable and can be installed easily on available networks of PCs is probably ideal.

The last niche for we will discuss for Java message-passing is perhaps the most obvious.

Because of its platform independence, mobility, and other associations with the Internet, Java is a natural candidate as a language for *metacomputing*. We interpret this to mean computation by parallel programs distributed across the Internet itself. Within the MPI community there is an ongoing effort to extend MPI specifications and implementations to support metacomputing, by allowing logical process groups to span geographically separated clusters and supercomputers. For example, an MPI interoperability standardization effort led by the National Institute of Standards and Technology [18] proposes a cross-implementation protocol for MPI—*Interoperable MPI* or IMPI—to support heterogeneous parallel computing. Java-based metacomputing can exploit and supplement these ongoing MPI activities in various ways. It may be, for example, that only a parallel *sub*-component of a distributed application is particularly suited to implementation in Java. If the Java part is programmed in an MPI-like paradigm the option is open for the Java component to interact with the non-Java, MPI-based part through the inherently parallel IMPI protocols (rather than, say, through a serial, performance-limiting CORBA or RMI gateway).

Many authors have discussed Java approaches to metacomputing, but they have generally emphasized different aspects of Java. Charlotte [5, 6] and Javelin [10, 28] concentrate on harvesting cycles of computers running Web browsers by downloading *applets* to them—a paradigm well-suited to task-farming but not particularly appropriate for applications that need communication between concurrent tasks. JavaParty [29, 31] and Manta [33] support an interacting SPMD style of distributed programming, but emphasize communication through remote method invocation. They provide ways to program with remote objects that are more transparent than the standard RMI interface, together with highly-tuned reimplementations of RMI. This work is clearly important, but it remains uncertain whether remote method invocation is the best model of communication for parallel computing. The message-passing model of synchronization seems a better fit to the requirements. Although not directly relevant to our goals here, we note that even in the distributed computing community there appears to be some movement towards message-passing models. According to [36], one of the lessons of Ninja 1.0 was that RMI was not the best model for their purposes—asynchronous typed message-passing would be better. Hewlett Packard's *e-speak* architecture [17] adopts message-passing as the underlying model of communication with services.

C.2.3 Jini

Jini is Sun's Java architecture for making services available over a network. It is built on top of the Java Remote Method Invocation (RMI) mechanism. The main additional features are a set of protocols and basic services for "spontaneous" discovery of new services, and a framework for detecting and handling *partial failures* in the distributed environment.

A Jini lookup service is typically discovered through multicast on a well-known port. The discovered registry is a unified first point of contact for all kinds of device, service, and client on the network. Aside from the initial act of discovery, all Jini-related operations are built on RMI. The Jini model of discovery and lookup is distinct from the more global concept of discovery in, say, the CORBA trading services or HP's e-speak. The Jini version is a lightweight protocol, especially suitable for initial binding of clients and services within multicast range. In the Ninja framework, for example, Jini technology might fit comfortably

at the periphery, near the end-user devices, or *within* the Base, addressing initial federation of nodes, crashes of individual nodes, etc. This latter setting is particularly interesting to us.

The ideas of Jini run deeper than the lookup services. Jini completes a vision of *distributed programming* started by RMI. In this vision *partial failure* is a defining characteristic, distinguishing distributed programming from the textbook discipline of concurrent programming [34]. The principles of concurrent programming are integrated in the Java language and the JVM through support for threads and monitors. But mechanisms that are appropriate within a single JVM must be replaced by more complex techniques when multiple JVMs are federated over a network. Remote objects and RMI replace ordinary Java objects and methods; garbage collection for recovery of memory is replaced by a *leasing* model for recovery of distributed resources; the events of AWT or JavaBeans are replaced by the distributed events of Jini; the synchronized methods of Java are mirrored in the nested transactions of the Jini model. The interesting question arises of whether analogous ideas can be adopted to extend conventional *parallel* programming models.

C.2.4 Bringing these things together

To support the parallel programmers of the future we will need Java implementations of lightweight messaging systems akin to MPI—the single most successful model for parallel computing. A likely physical setting is in the more or less tightly coupled (but probably heterogeneous, multi-user) clusters of trusted workstations that we expect will host the Web services of the future. While models of distributed programming other than message-passing (notably Linda-based models like JavaSpaces or JavaNOW) certainly have a role, we question whether they are the best model for SPMD computing. Most of the experience with earlier generations of parallel computer suggests that the low-latency message-passing model is a better fit.

These are likely to be volatile environments that demand the reliability provided by foundations like Java and Jini. Any software must be adaptive. Availability changes as workloads and network traffic fluctuates; nodes crash, new ones are attached and discovered on the fly, old ones are removed. Jini is a leading Java technology for dealing with these situations. Message-passing parallel programming is not exactly the same discipline as concurrent programming. An interesting research question is whether one can develop is a distributed model of parallel programming that extends the conventional MPI model in a manner similar to the way the Jini model extends concurrent programming. At the very least, as discussed in section C.4.3, Jini technology can be exploited to help implement the conventional MPI model in a distributed environment.

C.3 General plan of work

There are two principal strands in the proposed work. One strand will investigate use of Java and specifically *Jini* technologies in a middle tier for initiating parallel MPI jobs. For definiteness we refer to the architecture as *JiniMPI*. So far as the architecture itself is concerned the parallel program could be written in Java or some other language that invokes an MPI-style message layer. The second strand will be researching issues related to

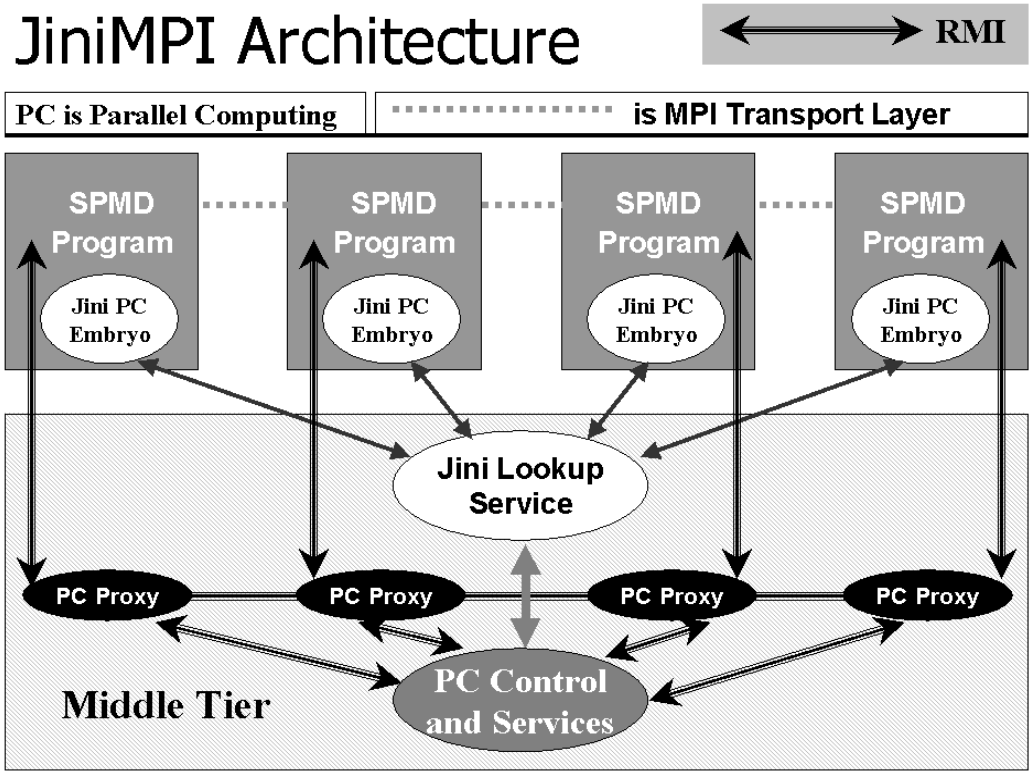

Figure C.1: JiniMPI Architecture

message-passing programs written *in* Java (and potentially other object-oriented network programming languages). It will study principles of reliably and efficiently implementating associated APIs in dynamic environments like Internet servers and networks. The concrete APIs will be derivatives of the *MPJ* specification from the Java Grande Message-Passing Working Group (section C.4.2).

In principal the *JiniMPI* and *MPJ* levels are independent, but they complement each other and are expected to be especially powerful when used together.

C.3.1 Jini-based middle tier for parallel programming

A possible JiniMPI architecture is illustrated in Figure C.1. The architecture supports MPI-based parallel computing and also includes ideas present in systems like Condor and Javelin. The diagram only shows the server layer (bottom) and the service layer (top). There would also a client layer that communicates directly with "Control and Services" module.

We assume that each workstation has a "Jini Parallel Computing Embryo"—a Jini service that registers the availability of the workstation to run either specific or generic applications. The Jini embryo can represent the machine—the ability to run general applications—or particular software. The Gateway, or Control and Services module [1], queries the Jini lookup services to find appropriate computers to run a particular MPI job. The mechanism could be used just to be run a single job, or to set up a farm of independent workers

The standard Jini mechanism is applied for each chosen embryo. This effectively establishes an RMI link from the Gateway to each SPMD node. It creates a Java proxy for the node program, which can itself be written any language (Java, Fortran, C++ etc). The Gateway-Embryo exchange should also supply to the Gateway with any needed data (such as specification of required parameters and how to input them) on behalf of the client layer. This strategy separates control from data transfer. It supports Jini (registration, lookup and invocation), services such as load balancing, and fault tolerance, in the control layer, and MPI style data messages in a fast transport layer. The Jini embryo is only used to initiate the process. It is not involved in the actual execution phase.

We will also investigate the implications of using a JavaSpace in the Control layer as the basis for a management environment (this is very different from using Linda or JavaSpaces at the execution level, so performance problems are not very relevant).

C.3.2 Research on high-performance message passing models for Java

In the early stages of the project we will complete a reference implementation of the *MPJ* specification, an MPI-inspired Java API from the Java Grande Message-Passing Working Group (section C.4.2). Our existing *mpiJava* software (section C.4.1) has been one basis for this work, but for further research a portable, *pure Java* version will be needed. Section C.4.3 describes a design. One of the conclusions of the design study was that difficult issues of reliability (and useability) in a network environment are naturally addressed in the framework of the Jini programming model. Our initial reference implementation will make extensive use of Jini for job initiation and handling failures. This implementation will be a foundation for subsequent research in the project.

The current MPJ specification supports essentially MPI-1.0 functionality, with some extensions specific to object-oriented languages (for example it has the facility to send and receive arbitrary serializable objects). With the reference implementation in place, the project will follow two directions

- Research into optimizations specific to the language and network context, to improve bandwidth and latency.
- Design and pilot implementation of extensions to the basic message-passing model, improving support for highly dynamic environments.

Associated tasks are detailed in the following two subsections.

Fast message-passing for Java

The initial reference implementation will use Java sockets as the basic transport. Later work will research different approaches to low-level transport—including calling native MPI by standard JNI or other methods, or using Java bindings to lower-level interfaces like VIA. It is also likely to involve work on improving the efficiency of object serialization, or exploiting the research of other groups on efficient object serialization.

Our earlier work on mpiJava already exploited the Java Native Interface (JNI) to call native MPI implementations. But there are concerns with the efficiency of crossing the JNI

barrier. For example, several of the current generation JITs have garbage collectors that do not support pinning of objects. A consequence is that arrays are copied every time they are passed between C and Java. More detailed criticisms of JNI can be found in [35] and [9]. The two groups involved have described ways to go beyond standard JNI, specifically to support efficient Java interfaces to VIA. Our research will attempt to leverage this work, either by adapting similar techniques to make low-overhead interfaces to native MPI, or (for suitable platforms) adopting Java VIA interfaces as a low-level transport in our Java implementation of the message-passing API (along lines comparable with [11]). Note that the new approaches typically assume limited changes to compiler or JVM, at least in the garbage collector.

Another area that needs further research is specific to object-oriented languages. To allow fast communication of *object graphs* between processors we need very efficient object serialization. Important work on improving Java serialization has been described in [29]. The authors report that their UKA-Serialization can save 76% to 96% of the time needed to serialize objects. Their work was done in the context of an optimized reimplementation of RMI, but we can use the same software in fast MPJ implementations. We hope to combine ideas from UKA-Serialization with the MPI-specific ideas we started to explore in [7], to facilitate fast communication of objects in MPJ, especially for the important case of multidimensional Java arrays (if native methods are allowed, *one*-dimensional arrays of primitive elements can be communicated with no serialization at all, at least in the common case where sender and receiver have the same number representation and endianness).

Extending the MPJ message-passing model

As noted above, the initial MPJ draft specifies functionality similar to MPI 1.1, complemented by some object-oriented features.

One set of extensions to this draft will be inspired by features of the MPI 2 standard. This standard is not yet widely implemented even for traditional languages, but it includes features that are likely to be important in the volatile Java environments we target. Dynamic process creation was not part of MPI 1 but it is undoubtedly important for our environments. An early addition to the baseline MPJ model will be operations similar to MPI_COMM_SPAWN and MPI_COMM_SPAWN_MULTIPLE, which in MPI 2 start new groups of processes and return intercommunicators connecting them to the inital group. A related form of process creation is familar from systems like PVM. A new feature in our research will be adoption of Jini or similar current technologies for discovery of the required computational resources. Another highly relevant feature of the MPI 2 specification is its introduction of a parallel client/server model, by which two running parallel programs (client and server) that do not initially share a communicator can establish a connection, and thus operate collectively as a single parallel program for some period. This kind of functionality is likely to be important in metacomputing applications. It introduces issues of how the initial rendezvous between the client and server occurs. MPI 2 specifies a simple mechanism based on entry points MPI_PUBLISH_NAME, MPI_LOOKUP_NAME, which are optional in MPI implementations. Clearly a Java environment will be able to offer much more flexible mechanisms modelled on Jini lookup.

Dynamic discovery of compute resources is one area where Jini-like ideas can help us.

A more difficult issue for scalable computing in networked environments is how to deal with dynamic *loss* of resources, due to a failure somewhere in the distributed platform. As emphasized in Section C.2.3, the general Jini framework incorporates various mechanisms designed to support programming in the presence of partial failure. One goal of our research will be to explore ways to incorporate similar concepts in the parallel message-passing context, enabling programs that are truly scalable even in the presence of partial failure. As a simple example, suppose we provided a collective checkpointing operation that dumped the current state of a parallel program to backing store, perhaps by serializing the replicated objects supporting the "main program" method (this scenario begs several questions of detail, but these can be addressed later). Invoking this collective operation at regular intervals would go some way to improving the reliability of a parallel program, by allowing restarts. But if a partial failure occurs during the checkpointing operation itself, the backed-up state will be corrupted. A scalable version of the Jini *transaction* model may be what is needed to rescue this kind of situation. This is one easy example, but it suggests that we should find roles for Jini-like ideas at the level of the SPMD program. Most likely these would be applied at the level of large, collective operations, such as global checkpointing, forking groups of slave processes for some subtask, and so on.

C.3.3 A three year workplan

Year one: In the first year we will complete ongoing work on a Jini-based pure Java implementation of MPJ—a message-passing environment with functionality similar to MPI 1.1. This implementation will be an important foundation for the subsequent research in the project. To support the later research, the transport layer must be easily replacable (similar to KaRMI, for example). This implies a layer analogous to the MPICH device level, but we will need new features to support Java and objects. Jini will be used for discovering compute hosts and (importantly) to ensure clean global termination in the event of failures. To prove the design, a native MPI-based implementation of the transport layer will also be implemented. This will complement the initial socket-based implementation, and will use standard JNI. (Our existing *mpiJava* software will probably be phased out.) In this year we will also be studying issues relating to concrete design of the JiniMPI Architecture, using Jini technologies as a gateway to parallel computing resources.

Year two: The basic message passing model will be extended with a version of dynamic process spawning using Jini to find resources, and a parallel client/server model, using Jini to establish connections between running parallel programs. Suitable Java-centric APIs will be designed. Relevant application codes from areas like parallel data-mining will be produced. We will study ideas following on from the Jaguar and JAVIA work at Berkeley and Cornell, and try to make use of those ideas to optimize our initial MPJ implementation, the goal being to develop genuine high-bandwidth, low-latency message-passing in Java. We will integrate UKA-serialization or successors, and further study MPI-specific improvements for important cases such as multidimensional arrays. Extensions needed to support efficient communication of the scientific Array classes supported by Java Grande will be a prioriy. Pilot implementations of JiniMPI architecture will be developed. We will investigate reliable checkpointing primitives for parallel programming, using Jini

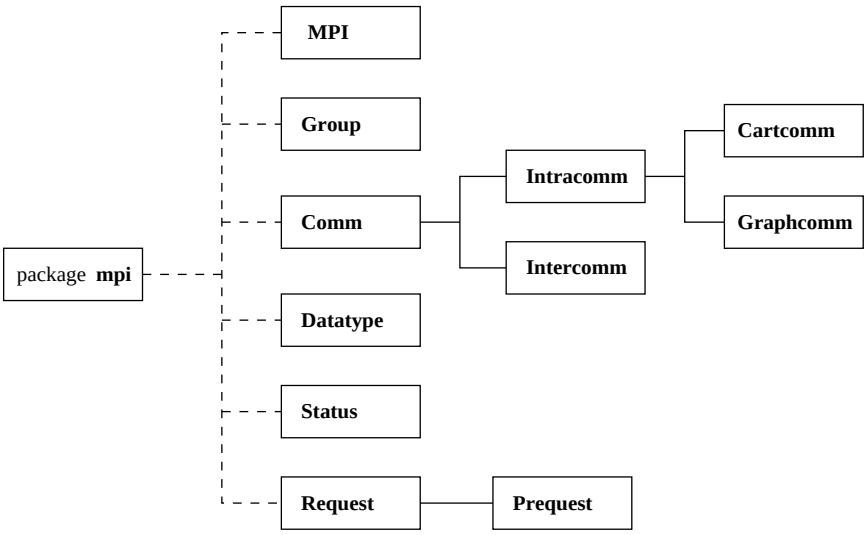

Figure C.2: Principal classes of mpiJava

transactions, or related mechanisms.

Year three: The experience of the first two years will feed into a practical model of scalable Internet computing, combining parallel-programming ideas from MPI with Jini ideas about fault tolerance. This model will be integrated with the three-tier JiniMPI model.

C.4 Related Work

C.4.1 Experience with *mpiJava*

mpiJava [4, 7, 27] is our object-oriented Java interface to MPI. The system provides a fully-featured Java binding of MPI 1.1 standard. The object-oriented API is modelled largely on the C++ binding that appeared in the MPI 2 standard. The implementation of mpiJava is through JNI (Java Native Interface) wrappers to a suitable native implementation of MPI. The software comes with a comprehensive test-suite translated from the IBM test-suite for the C version of MPI. Platforms currently supported include Solaris using MPICH or SunHPC-MPI, Linux using MPICH, and Windows NT using WMPI 1.1.

The MPI standard is explicitly object-based. The C and Fortran bindings rely on "opaque objects" that can be manipulated only by acquiring object handles from constructor functions, and passing the handles to suitable functions in the library. The C++ binding specified in the MPI 2 standard collects these objects into suitable class hierarchies and defines most of the library functions as class member functions. The mpiJava API follows this model, lifting the structure of its class hierarchy directly from the C++ binding. The major classes of mpiJava are illustrated in Figure C.2.

The benchmarks in Figure C.3 compare mpiJava ("J") timings with native C timings for communication between a pair of PCs. The timings represent two different native MPI implementations (MPICH and WMPI), and also compare with with raw Windows sockets.

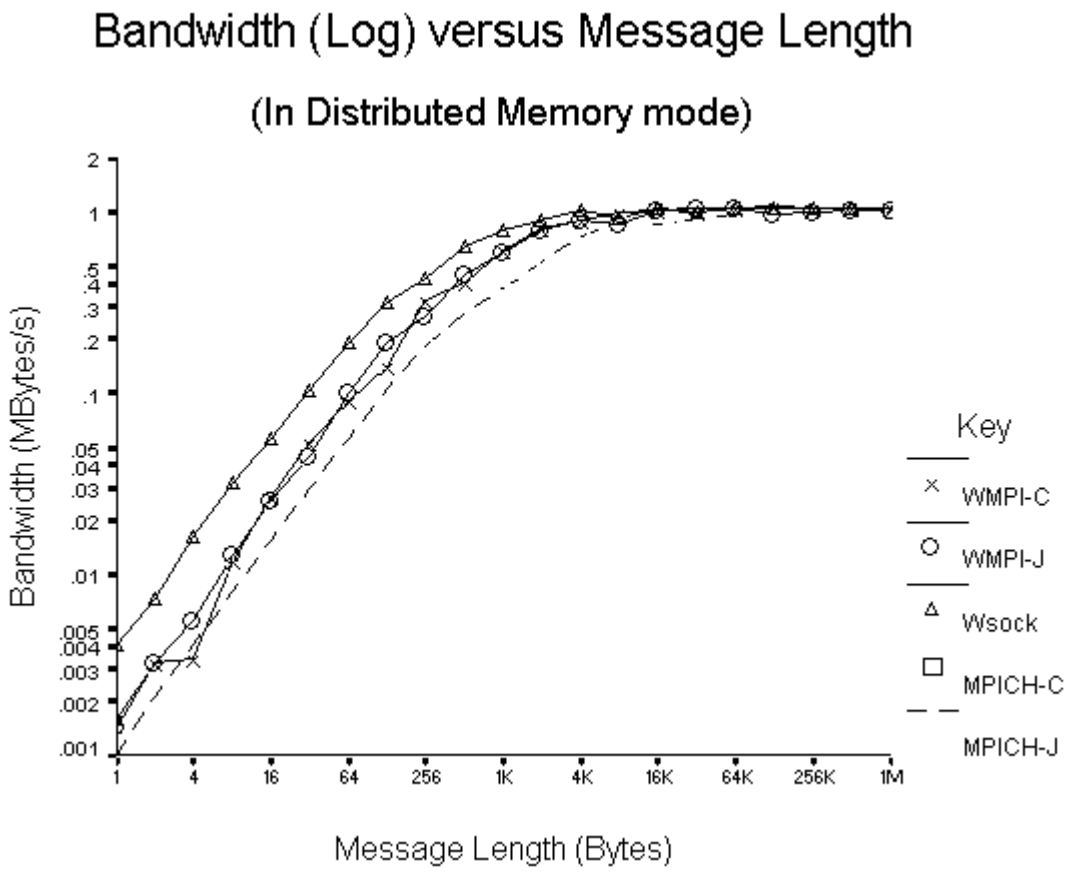

Figure C.3: PingPong Results in Distributed Memory mode

We see that the mpiJava JNI wrappers introduce a modest extra latency relative to native MPI, but for large messages bandwidth is not compromised. Although these results are encouraging, we should remark that these benchmarks were run using the Classic JVM. Current JIT compilers will degrade bandwidth because Java arrays are usually copied when they are passed to native methods. How best to avoid such overheads is an important research question [9, 35]. Also, these results are obtained for one-dimensional arrays with no serialization applied.

mpiJava is part of the *HPJava* environment [30]. We are actively developing and supporting the software. Downloads currently run at about 30 per month.

C.4.2 Java Grande Message-passing Working group

Java bindings to MPI were developed independently by several teams. One Java MPI interface was produced by Getov and Mintchev [15, 23]. In their work Java wrappers were automatically generated from the C MPI header. This eased the implementation work, but did not lead to a fully object-oriented API. A subset of MPI was implemented in the DOGMA system for Java-based parallel programming [21]. Dincer and Kadriy described an instrumented Java interface to MPI called *jmpi* [12]. Java implementations of the related PVM message-passing environment have been reported in [38] and [14].

The Message-Passing Working Group of the Java Grande Forum was formed just over

a year ago as a response to the appearance of these diverse APIs. An immediate goal was to discuss a common API for MPI-like Java libraries. An initial draft for a common API specification was distributed at Supercomputing '98 [8]. Since then the working group has met in San Francisco and Syracuse, and a Birds of a Feather meeting was held at Supercomputing '99. Minutes of meetings were published on the *java-mpi* mailing list and are available at [19, 20]. To avoid confusion with standards published by the original MPI Forum (which is not presently convening) the nascent API is now called *MPJ*.

C.4.3 Case study: reference implementation of MPJ

Presently there is no complete implementation of the draft MPJ 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 [8].

The mpiJava wrappers rely on the availability of a platform-specific native MPI implementation for the target computer. While this is a reasonable basis in many cases, the approach has some disadvantages. For one thing the two-stage installation procedure—get and build a native MPI then install and match Java wrappers—can be tedious and discouraging to potential users. Secondly, in the development of mpiJava we sometimes saw conflicts between the JVM environment and the native MPI runtime behaviour. The situation has improved, and mpiJava now runs with several combinations of JVM and MPI implementation, but some problems remain. 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. Ultimately, such packages would presumably be the best, industrial-strength implementations of systems like MPJ. Meanwhile, to address the last shortcoming listed above, we have outlined in [3] a design for a *pure-Java* reference implementation for MPJ. Design goals were that the system should be as easy to install on distributed systems as we can reasonably make it, and that it be sufficiently robust to be useable in an Internet environment. A particularly strong requirement is that in no circumstances should the software leave resource-wasting orphan processes lingering after an abrupt termination.

We are by no means the first people to consider implementing MPI-like functionality in pure Java. Working systems have already been reported in [12, 21], for example. Our goal was to build on some lessons learnt in those earlier systems, and produce software that is standalone, easy-to-use, robust, and fully implements the specification of [8].

We wish to simplify installation of message-passing software to a bare minimum. A user should download a jar-file of MPJ library classes to machines that may host parallel jobs, and run a parameterless installation script on each. Thereafter parallel java codes can be compiled on any host in the LAN (or subnet). An `mpjrun` program invoked on the development host transparently loads all the user's class files to available compute hosts, and the parallel job starts. The only *required* parameters for the `mpjrun` program should be the class name for the application's main program and the number of processors the application is to run on.

To be usable, an MPJ implementation should be fault-tolerent in at least the following

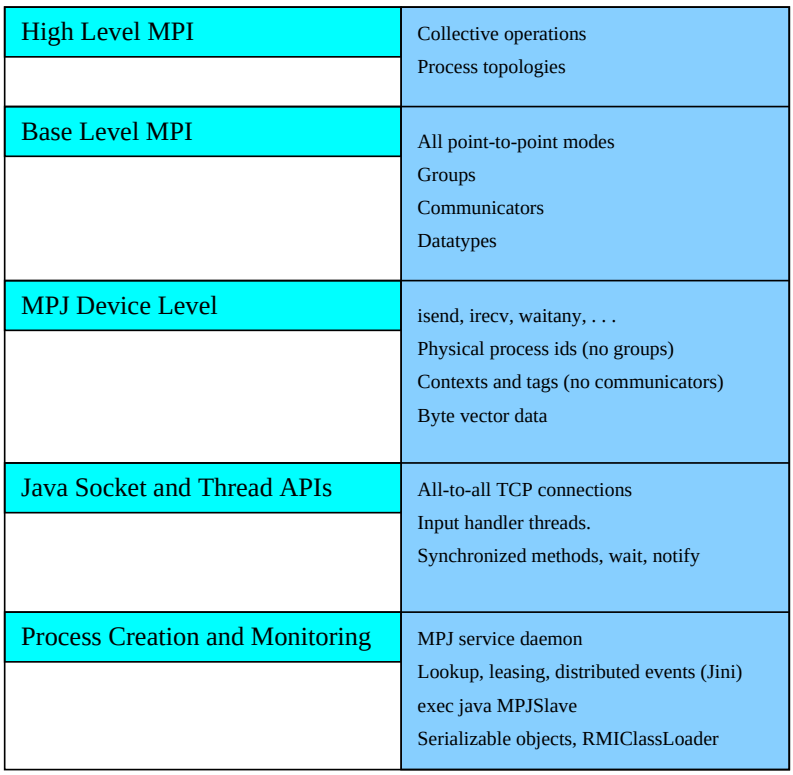

Figure C.4: Layers of proposed MPJ reference implementation

senses. If a remote host is lost during execution, either because a network connection breaks or the host system goes down, or for some other reason, *all* processes associated with affected MPJ jobs must shut down within some 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.

The paper design suggests that Jini is a natural foundation for meeting these requirements. The installation script can start a daemon on the local machine by registering a persistent activatable object with the `rmid` daemon. The MPJ daemons automatically advertise their presence through the Jini lookup services. The Jini paradigms of leasing and distributed events are used to detect failures and reclaim resources in the event of failure. These observations lead us to believe that an initial reference implementation of MPJ should probably use Jini technology [2, 13] to facilitate location of remote MPJ daemons and to provide a framework for the required fault-tolerance.

A possible architecture is sketched in Figure C.4. The base layer—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 (Figure C.5). 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

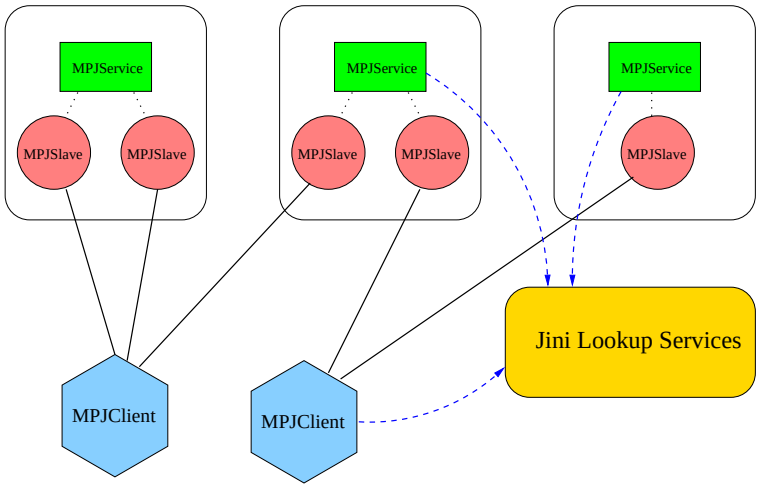

Figure C.5: Independent clients may find `MPJService` daemons through the Jini lookup service. Each daemon may spawn several slaves.

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.

Higher layers use Java sockets directly for efficient communication. The first manages low-level socket connections, establishing all-to-all TCP socket connections between the hosts. The idea of an "MPJ device" layer is inspired by the abstract device interface of MPICH. A minimal API includes non-blocking standard-mode send and receive operations. Other point-to-point communication modes are implemented with reasonable efficiency on top of this minimal set. The device level itself is meant to be implemented on socket `send` and `recv` operations, using standard Java threads and synchronization methods to achieve its richer semantics. The next layer above this 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.

C.4.4 Results from prior NSF support

Work on this proposal is related to previous NSF awards including the research grant described below from the Division of Advanced Computational Infrastructure and Research. This project is ongoing and will transfer from Syracuse to Florida State. Fox is PI.

Grant: 9872125, total award $346,827 over period 09/01/98-08/31/01, "Data Parallel SPMD Programming Models from Fortran to Java".

This involves senior personnel Fox and Carpenter, who with two students on the project will be moving from Syracuse to Florida State. The project focuses on the use of Java for *data parallel* programming but the methods are applicable to other languages. Collaborator Professor Xiaoming Li from Peking University is investigating applications to traditional scientific languages—especially Fortran. We have published several papers on this subject where the details are described. The HPJava model is less ambitious than systems like High Performance Fortran (HPF) and aims to support an SPMD model intermediate between basic message passing (MPI) and HPF. One can incorporate pure MPI code but also array based computation with automatic decomposition with a user specified mapping in the spirit of HPF. An essential capability is unified support of successful data parallel libraries like ScaLAPACK, PetSC, Kelp, Global Array Toolkit, PARTI/CHAOS and Adlib. So far we have developed an operational HPJava translator and linked to Global Arrays and Adlib. As part of our collaboration we have prepared and given in China a tutorial on HPJava and related approaches (http://www.npac.syr.edu/projects/pcpc/HPJava/beijing.html). To support MPI work from within HPJava programs we developed the MPI Java binding *mpiJava* which is available for download from (http:/www.npac.syr.edu/projects/pcrc/HPJava-/mpiJava.html), and will be form one of the foundations of the work described in the current proposal. It is a reference implementation used by the Java Grande Message Passing Working Group. Fox organized the Java Grande Forum (http://www.javagrande.org) to address all the issues connected with the use of Java in scientific computing and both the working groups and associated conferences (now sponsored by ACM) have been quite successful. HPJava ideas have greately benefitted from contacts in this arena. As well as publications given below, Sung Hoon Ko will complete his Ph.D. in this area during this semester.

Select publications

B. Carpenter, G. Zhang, G. Fox, X. Li, X. Li and Y. Wen "Towards a Java environment for SPMD programming", *4th International Europar Conference*, Springer, 1998.

G. Zhang, B. Carpenter, G. Fox, X. Li and Y. Wen, "The HPspmd Model and its Java Binding.", chapter in book, R. Buyya ed, *High Performance Cluster Computing*, Vol 2, Prentice Hall 1999.

M. Baker, B. Carpenter, G. Fox, S.H. Ko and S. Lim, "mpiJava: An Object-oriented Java Interface to MPI", *Intl. Workshop on Java for Parallel and Distributed Computing, IPPS/SPDP '99*, San Juan, Puerto Rico, April 1999.

B. Carpenter, G. Fox, S.H. Ko and S. Lim, "Object Serialization for Marshalling Data in a Java Interface to MPI", *ACM 1999 Java Grande Conference*, ACM Press 1999.