A. Project Description

A.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 done *across* the Internet, although of course this is one possibility that has been widely discussed, especially 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.

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 [10]. The idea of an agreed, standard API for communication in parallel programs now seems obvious. In practise it was relatively slow in coming. As a result it benefitted from a great deal of accumulated experience from application developers using earlier, proprietary APIs for message passing. MPI supports the Single Program Multiple Data (SPMD) model of parallel computing. A group of processes cooperate by executing identical program images, manipulating distinct local data values. MPI provides reliable point-to-point communication with several blocking and non-blocking communication modes. It also provides a library of true collective operations (broadcast is the most trivial example). An extended standard, MPI 2, allows for 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 specifically adapted to the heterogeneous environment of the Internet, where downloadable and mobile code are norms, resources (including computational resources) may be discovered and lost spontaneously, unreliable networks and the associated need for fault tolerance are defining issues. In the proposed work we will be especially concerned to support use of network-oriented languages for high-performance computing. For now this means Java (although we cannot rule out the emergence of other comparably important languages in the course of the proposed work). Hence one immediate preoccupation is with refinement of MPI-like APIs for Java.

The validity of using Java for essentially "scientific" computing has been quite controversial in the past, but over the last few years the prospect 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 to close the credibility gap. By now it is widely assumed that future Java environments will meet the vital 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.

As a part of the Java Grande process, a Message-Passing Working Group has been considering MPI-like APIs for Java over the last two years. Draft specifications and prototype implementations have been produced, but work remains to be done on the final specification, and robust reference implementations are urgently needed.

In the end, just providing MPI-like APIs for Java not enough. We must address the specific features of distributed computing. MPI was designed for a *reliable* environment. According to the philosophy of Sun's Jini project, for example, a defining characteristic of distributed computing versus concurrent programming (or in our case, versus parallel programming) is the presence of *partial failures.* We intend to exploit ideas similar to those developed in the Jini project. We anticipate such ideas will facilitate construction of scalable and fault-tolerant platforms for parallel computing, harvesting spontaneously discovered computational resources. The technology adds new job initiation capabilities, beyond those found in traditional high-performance computing environments.

We will research robust pure-Java implementations of the message-passing API, *MPJ*, specified by the Java Grande Message-Passing Working Group. We argue such implementations are likely to internally adopt Jini protocols and software components to manage distributed resources and detect failures, while providing a traditional SPMD programming environment at the user level. Later research will investigate how to bring low-level techniques for dealing with partial failures into the realm of control of the parallel programmer. An overall research goal is to combine successful technologies for High Performance computing with state-of-the-art object-oriented technologies for network programming.

A.2 Motivation of the proposed work

A.2.1 Why parallel computing and Java?

To realize its full potential, parallel computing should 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 more resistance. There are two main reasons for this. One is the level of investment in existing scientific codes and libraries written in Fortran, C, or C++. This particular concern can largely be addressed by wrapping legacy codes in Java interfaces [6]. A more 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 supporting Java standards for large-scale ("Grande") applications. Through a series of ACM-supported 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's T.J. Watson Research Center [11, 12, 16], 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 [13] 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.

Also need to say something here about approaches to parallelism in JGF.

A.2.2 Where will parallel Java programs live?

We can assume that the computers that host major Web sites will either be multiprocessors or clusters of workstations. Many are now, and this trend will presumably continue *[Cite Brewer paper, somehow]*. Increasingly these servers are programmed in Java. The simple fact that the computers are parallel clearly does *not* imply that people will write parallel programs for them. Most of the time they surely will not—individual nodes will just handle independent transactions. But since these technologies—Java and parallel computers—will coexist in Internet servers, this is fertile ground in which we should see roles for Java-based parallel computation emerging. In one scenario we can imagine that in the near future compute-intensive commercial services start to make their appearance on the Web. The exact nature of these services is still uncertain. Perhaps they will be data-mining queries using parallel algorithms, or financial analysis programs based on physical optimization processes, or perhaps people will simply want to play chess against a parallel computer.

If the parallel server is a symmetric multiprocessor it may be possible to write parallel programs simply by using Java threads. But for truly scalable servers this is unlikely to be the situation. As a specific example, consider the *Ninja* vision of the future of the Internet elaborated by researchers at UC Berkeley [14]. 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 the client devices, whatever they may be. 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. The Ninja Base is assumed to be a cluster of workstations with fast, local communication, a controlled environment, and a single administrative domain [7]. The Base may be constructed from a heterogeneous set of nodes, and individual nodes may fail under unpredictable loads, and so on; but the cluster is strongly coupled and essentially trustworthy. It is not necessarily homogeneous and it is not completely reliable, so it is not exactly a conventional parallel computer. However this is one environment where we might expect parallel programs written in Java to thrive. Partly, as suggested above, this could happen because massively parallel programming will be needed to implement the individual Internet services of the future; partly it may be because the commodity parallel computers of the future will be designed primarily as Internet servers, because this is where the demand will be. Scientific programmers may exploit these resources to run their programs simply because they are readily available.

A completely different place where we might see early uptake Java-based parallel computing is in the classroom *[...]*

The last niche for parallel Java we will mention is in a sense 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. *[Cite some relevant papers from the JG workshops, if poss.]* Within the MPI community there is an ongoing effort to extend MPI specifications and implementations to support metacomputing, by allowing logical process groups (the basic cooperative units of parallel computation) to span geographically separated clusters and supercomputers. For example, an MPI interoperability standardization effort led by the National Institute of Standards and Technology [8] proposes a cross-implementation protocol for MPI that will enable heterogeneous computing. MPI implementations that support the *Interoperable MPI* (IMPI) protocol will allow parallel message passing computations to span systems, using the native vendor message passing library within each system.

Java-based metacomputing can exploit and supplement these ongoing MPI activities in various ways. Suppose, 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 the essentially MPI-like paradigm we espouse, 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). In another scenario, a parallel program may be written uniformly in Java, using our MPI-like API. An optimized implementation of the communication class library is made available at each site that hosts distributed Java jobs. Internally these implementations can use a native, vendor-supplied MPI, with IMPI protocols between sites. The platform-independent, compiled byte-code for the user's parallel program is uploaded to host sites at run-time and dynamically linked to the local message-passing stubs.

The three application areas described here—parallel servers, teaching, metacomputing—are suggestive only. This is essentially a research proposal, and we cannot predict with certainty how the results might be used.

A.2.3 Where does Jini fit in?

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. Services install serialized proxy objects in the registry; clients download the proxy for the service they need, selecting primarily on the Java class of the serialized object. 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 light-weight 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 [15]. In this vision *partial failure* is a defining characteristic, distinguishing distributed programming from the textbook discipline of concurrent programming. The principles of concurrent programming are embodied 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 something more complex when multiple JVMs are federated over a potentially unreliable 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 in the distributed events of Jini. The synchronized methods of Java are mirrored in the nested transactions of the Jini model.

A.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 platform 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) certainly have a role, we doubt 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 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.

A.3 Related Work

A.3.1 Experience with *mpiJava*

mpiJava is our object-oriented Java interface to MPI. It implements a Java API suggested in 1997. That proposal built on work on Java wrappers for MPI started at NPAC about a year earlier. 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 by JNI (Java Native Interface) wrappers to a

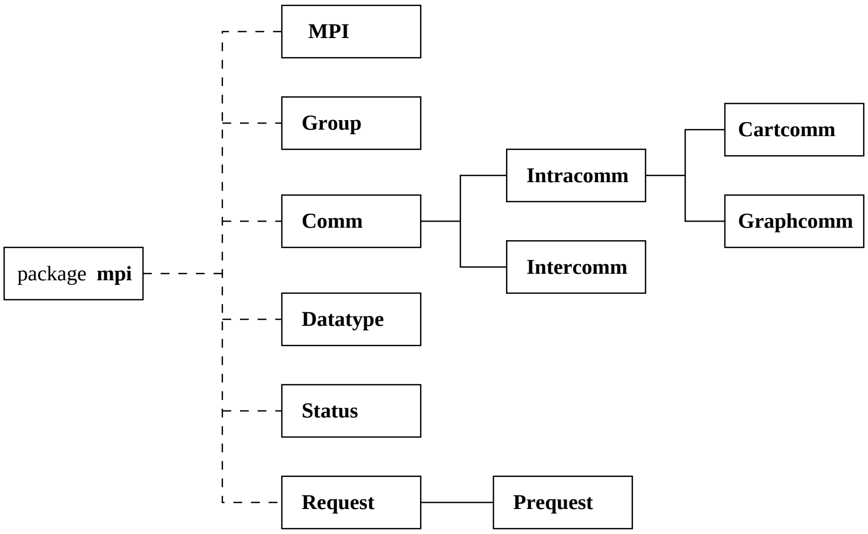

Figure A.1: Principal classes of mpiJava

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 A.1. A minimal mpiJava program is illustrated in Figure A.2.

[Select benchmarks. Explain them. Mention copying and JNI.]

mpiJava is part of the *HPJava* environment.

A.3.2 Java Grande Message-passing Working group

The Message-Passing Working Group of the Java Grande Forum was formed just over a year ago as a response to the appearance of several prototype Java bindings for MPI-like libraries. *[Describe them.]*

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, and a Birds of a Feather meeting was held at Supercomputing '99. The nascent API is now called MPJ.

[etc.]

```
import mpi.* ;

class Hello {
  static public void main(String[] args) {
    MPI.Init(args) ;

    int myrank = MPI.COMM_WORLD.Rank() ;
    if(myrank == 0) {
      char [] message = "Hello, there".toCharArray() ;
      MPI.COMM_WORLD.Send(message, 0, message.length, MPI.CHAR, 1, 99) ;
    }
    else {
      char [] message = new char [20] ;
      MPI.COMM_WORLD.Recv(message, 0, 20, MPI.CHAR, 0, 99) ;
      System.out.println("received:" + new String(message) + ":") ;
    }

    MPI.Finalize();
  }
}
```

Figure A.2: Minimal mpiJava program (run in two processes)

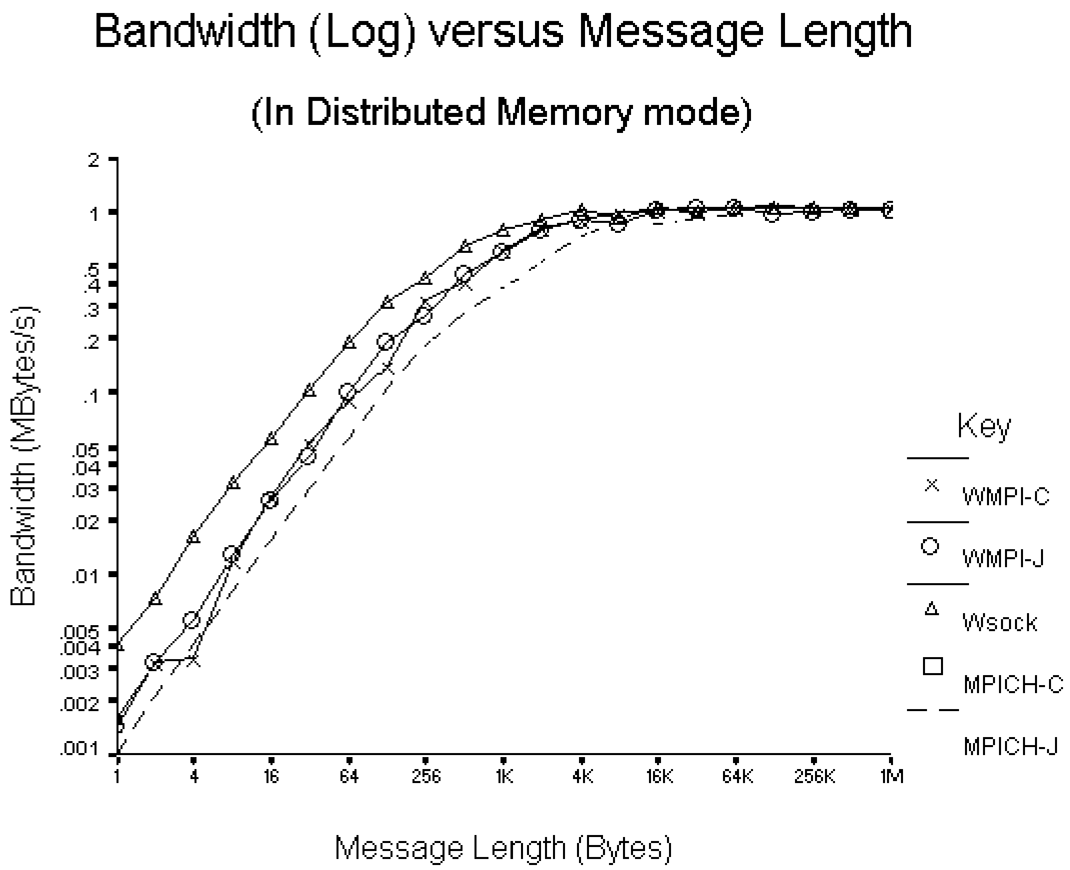

Figure A.3: PingPong Results in Distributed Memory mode

A.3.3 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 [3].

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 [2] 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 [4, 9], for example. Our goal was 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].

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 that host transparently loads all the user's class files to available hosts, and the parallel job starts. The only required parameters for the `mpjrun` program should be the class name for the application 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 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 those machines by registering a

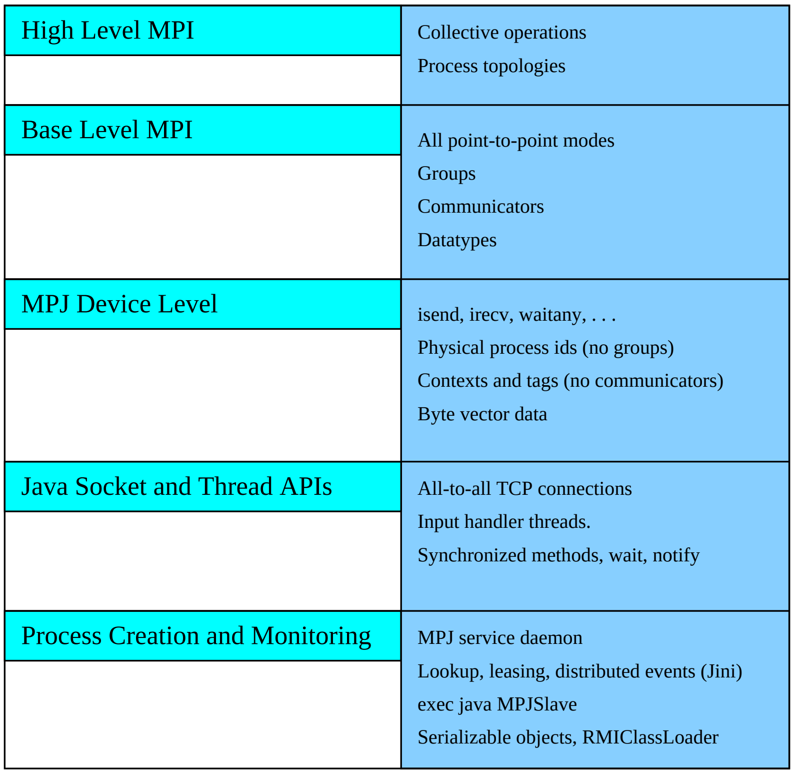

Figure A.4: Layers of an MPJ reference implementation

persistent activatable object with an existing `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 [1, 5] to facilitate location of remote MPJ daemons and to provide a framework for the required fault-tolerance.

A possible architecture is sketched in Figure A.4. 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 (Figure A.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 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 partly modelled on the abstract device interface of MPICH. A minimal API includes non-blocking standard-mode send and receive operations. All other point-to-point communication modes can be implemented with reasonable efficiency on top of this minimal set. The device level is intended to be implemented on socket `send` and `recv` operations, using standard Java threads

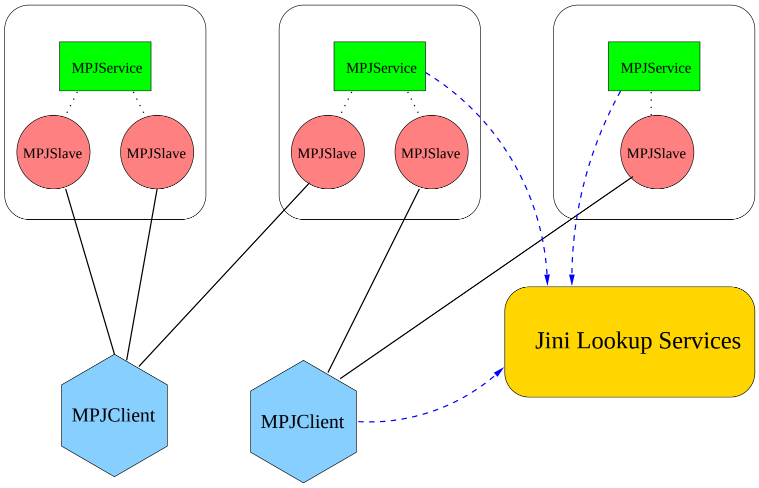

Figure A.5: Independent clients may find `MPJService` daemons through the Jini lookup service. Each daemon may spawn several slaves.

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.

A.3.4 Other approaches to parallelism in Java

[etc.]

More things to mention...

- *Object serialization*
- *MPI-2 (dynamic process creation).*
- *Java interfaces to VIA*
- *MPI-RT (Skjellum's work)*

Realistic assessment of current level of demand for MPI + Java

- *java-mpi membership: 71 subscribers.*
- *mpiJava downloads—currently around 30 per month, overall total about 330.*

Conclude: presumably not yet large enough for commercial exploitation, but enough to encourage research.

A.4 Workplan

Reiterate: this proposal will address design issues for new message-passing APIs for Java and potentially other object-oriented network programming languages. It will address the principles of reliably implementating these APIs in the dynamic environments of Internet servers and networks. In particular implementations on top of Jini and emerging successors will be developed. The proposal will consider how MPI-like parallel programming APIs may best be enhanced to make Jini-like techniques for fault-tolerant programming available directly at the parallel application level.

Complete MPJ reference implementation.

Improving the API: talk about channels, MPI-RT.

Hand waving about how we will import Jini-like fault-tolerance into MPJ. (Don't really have anything concrete to say yet. Checkpointing?)

B. Bibliography

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

[2] Mark Baker and Bryan Carpenter. MPJ: A proposed Java message-passing API and environment for high performance computing. In *International Workshop on Java for Parallel and Distributed Computing*, Cancun, Mexico, May 2000. To be presented.

[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] 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/.

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

[6] Vladimir Getov, Susan Flynn-Hummel, and Sava Mintchev. High-performance parallel programming in java: Exploiting native libraries. *Concurrency: Practice and Experience*, 10(11-13):873–877, 1998.

[7] Steven D. Gribble, Matt Welsh, Eric A. Brewer, and David Culler. The MultiSpace: an evolutionary platform for infrastructural service. In *1999 Usenix Annual Technical Conference*, Monterey, June 1999.

[8] IMPI Steering Commitee. *IMPI—Interoperable Message-Passing Interface*, January 2000. http://impi.nist.gov/IMPI/.

[9] 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.

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

[11] J. E. Moreira. Closing the performance gap between Java and Fortran in techincal computing. In *First UK Workshop on Java for High Performance Network Computing*, September 1998. http://www.cs.cf.ac.uk/hpjworkshop/.

[12] J. E. Moreira, S.P. Midkiff, and M. Gupta. From flop to MegaFlops: Java for technical computing. In *Languages and Compilers for Parallel Computing*, volume 1656 of *Lecture Notes in Computer Science.* Springer, 1998.

[13] J. E. Moreira, S.P. Midkiff, M. Gupta, and R. Lawrence. High performance computing with the array package for Java: A case study using data mining. In *Supercomputing 99*, November 1999.

[14] UC Berkeley Computer Science Division. Ninja project home page. http://ninja.cs.berkeley.edu/.

[15] 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].

[16] Peng Wu, Sam Midkiff, Jose Moreira, and Manish Gupta. Efficient support for complex numbers in Java. In *ACM 1999 Java Grande Conference*. ACM Press, June 1999.