A Status Report: Early Experiences with the implementation of a Message Passing System using Java NIO

Mark A. Baker, Hong Ong, Aamir Shafi
6th October 2004

Abstract

Since its release in 1996, Java has become a popular software development language. The reasons for its popularity can be attributed to the easy-to-use syntax, its portability, the extensive set libraries, and the support of object-oriented features like data hiding and polymorphism. One of the main drawbacks of Java was the blocking I/O package, but the situation has improved with the addition of the Java NIO package that adds scalable and non-blocking I/O to the language. The DSG is implementing a Java message passing system based on Java NIO package that runs on heterogeneous environment. In this report we discuss and evaluate our reference implementation, known as MPJ.

Table of Contents

List of Figures

List of Tables

1 Introduction

Cluster computing [1][2] has become a cost effective alternative to traditional HPC multi-processor systems for applications that have large-scale processing requirements. Shared memory and distributed memory are the two widely used programming paradigms for large-scale applications. The shared memory paradigm, as shown in the Figure 1, is used on hardware, where all the processors see a global memory space; this medium is the means of communication between the processors.

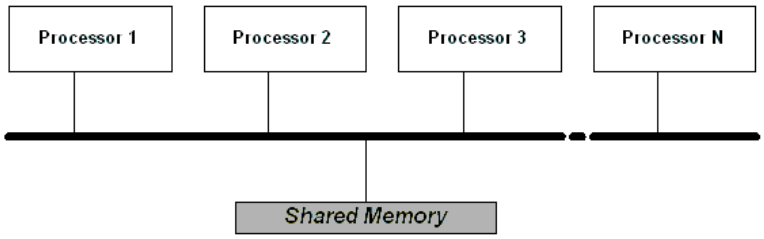

Figure 1: The Shared Memory Paradigm

In the distributed memory paradigm, as shown in the Figure 2, each node of the cluster has its own local memory that essentially means that there is no support for sharing data among nodes without explicitly passing messages between processors. On such hardware, message-passing libraries provide the communication medium between the processors. These libraries make use of the connecting networks to provide the communication medium and by sending and receiving messages; applications need to co-operate to solve a computational problem in parallel. At the programming level, the developer has to manage the data flow explicitly between the processors that are used. They have to know how to locate arrays of data, and when to set up the corresponding communicating calls between nodes participating in a computation. Further details about the two programming paradigms can be found in [3].

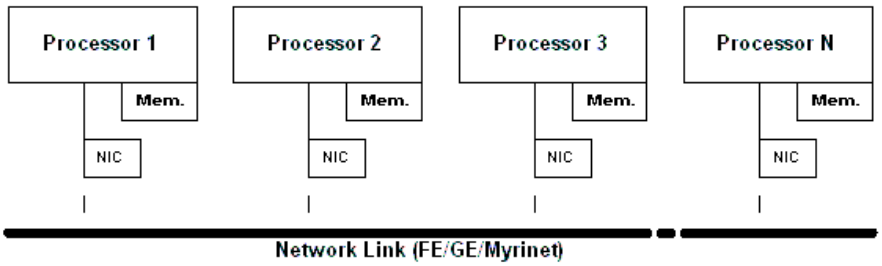

Figure 2: The Distributed Memory Paradigm

The Message Passing Interface (MPI) [4], first introduced in June 1994, is a standard for implementing message passing systems, which is used by the application developers for parallel computing over collection of machines. The current version of the MPI standard has language bindings for Fortran, C and C++. Two of the most popular libraries implementing the MPI standard are MPICH [5] and LAM [6] that support C/Fortran/C++ (LAM only) as the programming language.

In the recent past, there has been a growing interest in developing a message passing system in Java, which resulted in a Java binding for MPI [7]. The current standard and non-standard (according to bindings defined in [7]) implementations primarily follow three approaches. The first approach uses JNI [9] to invoke routines of the underlying native MPI that acts as the communication medium. The second approach uses RMI [10], which is the Java API that allows remote method invocation of distributed objects. The use of low-level "pure" Java communications based on Java sockets is the third approach. This is the preferred way to build a Java messaging system as it achieves better performance because of the use of sockets instead of RMI and ensures a truly portable system.

Despite various experimental projects that have developed Java message passing systems in the past, there is currently no single implementation that follows the API defined in [7], performs comparably well as other C MPI libraries, and makes use of the features of the Java language. Thus, the motivation behind this project is to build a pure Java message passing system that implements the bindings defined in [7], ensures portability, supports object oriented approach and attains performance comparable to the best alternative, currently mpiJava [61][63].

1.1 Project Objectives

The objective of this project is to develop a reference implementation of a Java message passing system that follows the recommended standard API, which includes a runtime library and an infrastructure that can provide all the support needed by parallel applications. The system, known as MPJ, has a messaging API based on MPI and uses Java sockets as its means of inter-process communications.

1.2 Report Outline

In Chapter 2, we discuss the motivation for this project, which is followed by a review of related work in Chapter 3. Chapter 4 and 5 discuss the design and implementation issues of MPJ. Chapter 6 evaluates the performance of MPJ and we conclude with an outline of potential future work in Chapter 7.

2 Project Motivation

2.1 Introduction

In this chapter, the motivation for developing MPJ is discussed. It starts with a discussion of some salient features of Java followed by a review of the Java New I/O package [11] (Java NIO). Java NIO adds scalable and non-blocking I/O to the Java language, which forms the basis for developing a scalable message-passing library.

2.2 Java Message Passing

There are two aspects of any language help judge its suitability for High Performance Computing (HPC). The first is the support for efficient numerical computing. Java is not best known for its support for this area due to the limitations imposed by the Java Virtual Machine (JVM). This is the layer that handles heterogeneity between different hardware and operating systems, at the cost of high performance numeric support. A detailed discussion of this area can be found in [12]. The second important requirement is the support for efficient communication using the underlying network hardware.

Bearing in mind this requirement, one may ask questions like, how long will a basic point-to-point message transfer between remote processes take? Also, how does this compare to the other popular languages, such as C or Fortran? As shown in Figure 8 and Figure 9, the point-to-point communication performance of Java is comparable to C. This indicates that a Java message

passing system would be capable of providing the application community with a tool that allows rapid application development and supports object oriented programming without compromising the communication performance. Such a tool would prove useful for teaching and simulation purposes too, where the motive is to learn and prove ideas without getting into the complex details of underlying hardware and software.

2.3 Benefits of the Java Programming Language

Since its release in 1996, Java has become an increasingly popular software development language. The reasons for its popularity can be attributed to its easy to use syntax, portability, the extensive set of libraries and the support of object oriented features like data hiding, and polymorphism. This sub-section discusses the advantages and disadvantages of the Java language in the context of HPC.

2.3.1 Portability

The most attractive feature of applications written in Java is that they are portable to any hardware or operating system, provided that there is a JVM for that system; following Sun's philosophy for Java of writing once and running anywhere. Java programs run in the JVM, which handles the complexity of dealing with the underlying hardware and operating system characteristics. The contribution of the JVM is significant, keeping in mind that it allows the new programmers and scientists to focus on issues related to their application and domain of interest and not on system heterogeneity.

2.3.2 JIT Compilers

The performance of the Java language suffered during its initial years because the JVM executed the bytecode that is generated by the Java compiler (javac). An alternative, and the more efficient approach, is to execute the native machine code as one would with the likes of C and Fortran. The pre-requisite for this is to convert the bytecode into the native machine code. This is what JIT (Just In Time) compilers are meant to do. These compilers first convert the bytecode into the native machine code and later execute the native machine code. A more detailed discussion on JIT compilers can be found in [13].

2.3.3 An Object Oriented Language

The Java language includes a large set of libraries that can be reused by the application developers in their parallel applications. It automatically brings the concept of object oriented programming, which may prove significant in terms of designing applications. This is complimented by the fact that the syntax of Java is easy to use for new programmers when compared to other languages such as C or Fortran. Code type safety is yet another feature that helps a beginner to avoid syntactic bugs in their code.

2.3.4 The Java New I/O Package

A hindrance in the development of a scalable pure java message-passing library has been the blocking I/O package of java. In the standard Java I/O package, a server requires a thread to handle the read and write operations of each I/O channel. Such a mechanism to handle I/O may become a performance bottleneck in long running applications or ones having a large number of processes. Figure 3 shows this problem where all processes are

connected to each other and each process requires a thread to handle an I/O channel for every connected process. This approach may not scale well given the large number of nodes in the modern clusters these days.

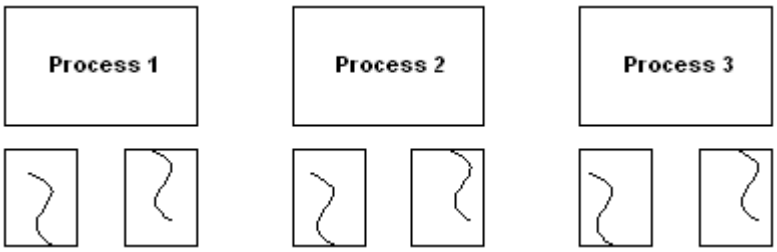

Figure 3: A Thread per I/O Channel at each Process

In this sub-section, we discuss the salient features of the Java New I/O package.

2.3.4.1 `Selectors` and `SocketChannels`

In Java NIO, there is an abstraction for a socket, called socket channel. These channels register with a selector, which is normally a separate thread in the JVM that is responsible for handling all the non-blocking I/O. Whenever there is something to read from a particular socket channel, the selector generates a matching read event, which can be handled and does the actual read. This concept is similar to `select()` in C, which helps scalable and efficient I/O. Figure 4 shows how the selectors solves the problem of needing a separate thread for each I/O channel. The number of channels are the same as in Figure 3, but now the receiving process potentially only needs the selector thread to handle all I/O channels, unlike Figure 3, where each process needs two threads to handle two I/O channels.

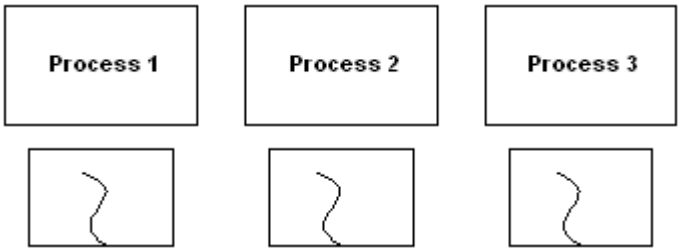

Figure 4: Single Selector Thread at Each Process

2.3.4.2 The Buffering API

Another useful feature of Java NIO is the buffering API. Here a buffer has the notion of being either direct or indirect, this concept was first conceived in the Jaguar project [14]. A direct buffer is a chunk of memory in the operating system's address space. It is not subject to garbage collection, as this does not reside like conventional objects, on the JVM heap. On the other hand, an indirect buffer is like any Java object that is created on the JVM heap. Direct and indirect buffers are shown in Figure 5.

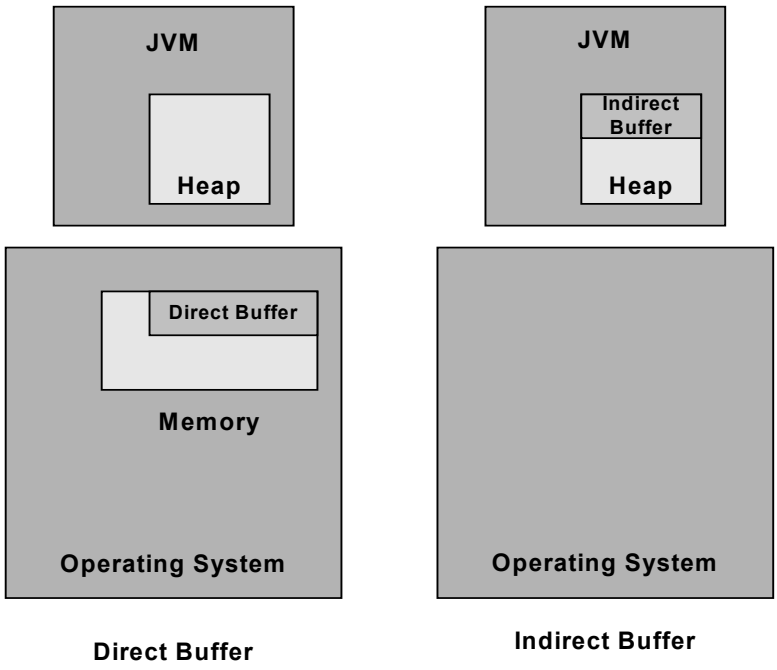

Figure 5: A Direct and an Indirect Buffer

The documentation for the Java NIO [4] buffer API suggests that creation of direct buffers is costly in terms of creation time. However, it provides faster I/O as it is managed by the operating system instead of the JVM itself. A comparison between creating a direct and indirect buffer, and time to copy `N` bytes onto a direct/indirect buffer is shown in Figure 6 and Figure 7.

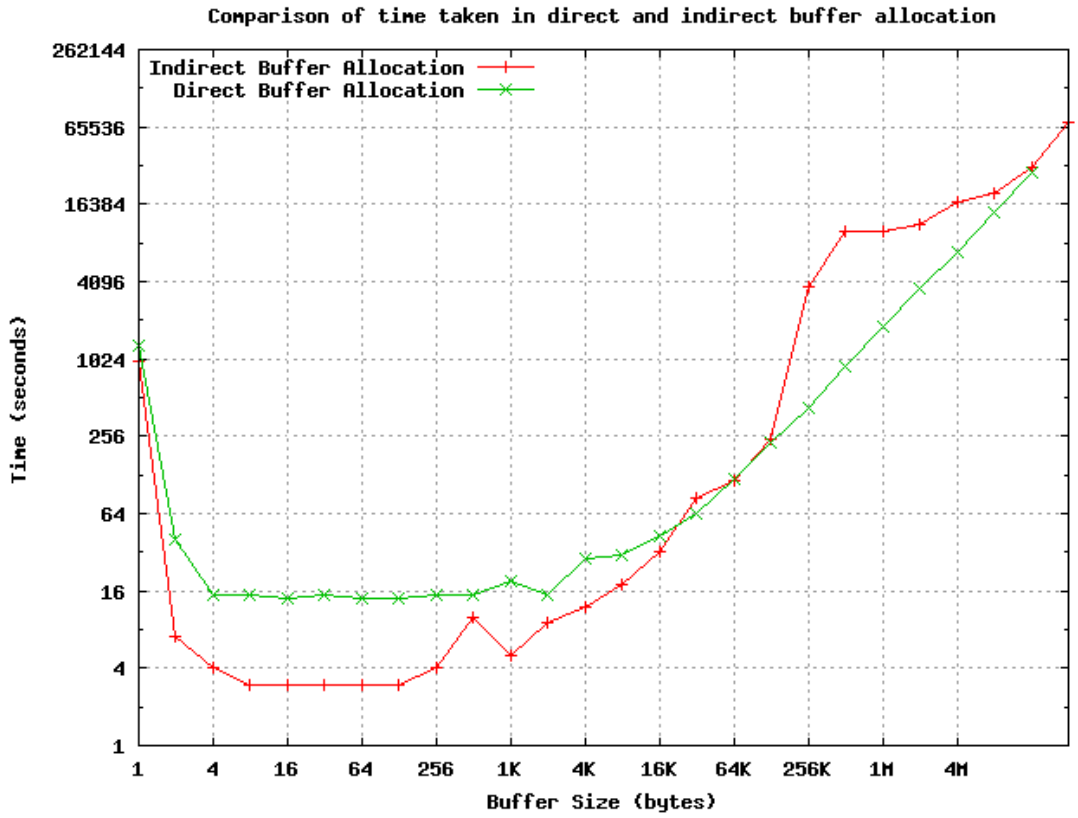

Figure 6: A Comparison between Direct and Indirect Buffer Allocation Times

The graph in Figure 6 shows a plot of the time taken in creating a fixed size buffer as a function of buffer size. As the documentation suggests the time taken in creating a direct buffer should be higher than the time taken to create an indirect buffer of the same size. As we can see in Figure 6, the time taken to create an indirect buffer actually takes less time until the buffer size reaches 128 Kbytes. Theoretically, the indirect buffer should still take less time, but in

practice it does not, as the time taken is dependant on the garbage collection, which is taking place in the background. On the other hand, the direct buffer takes predictable creation time with little variation because these buffers are not subject to garbage collection.

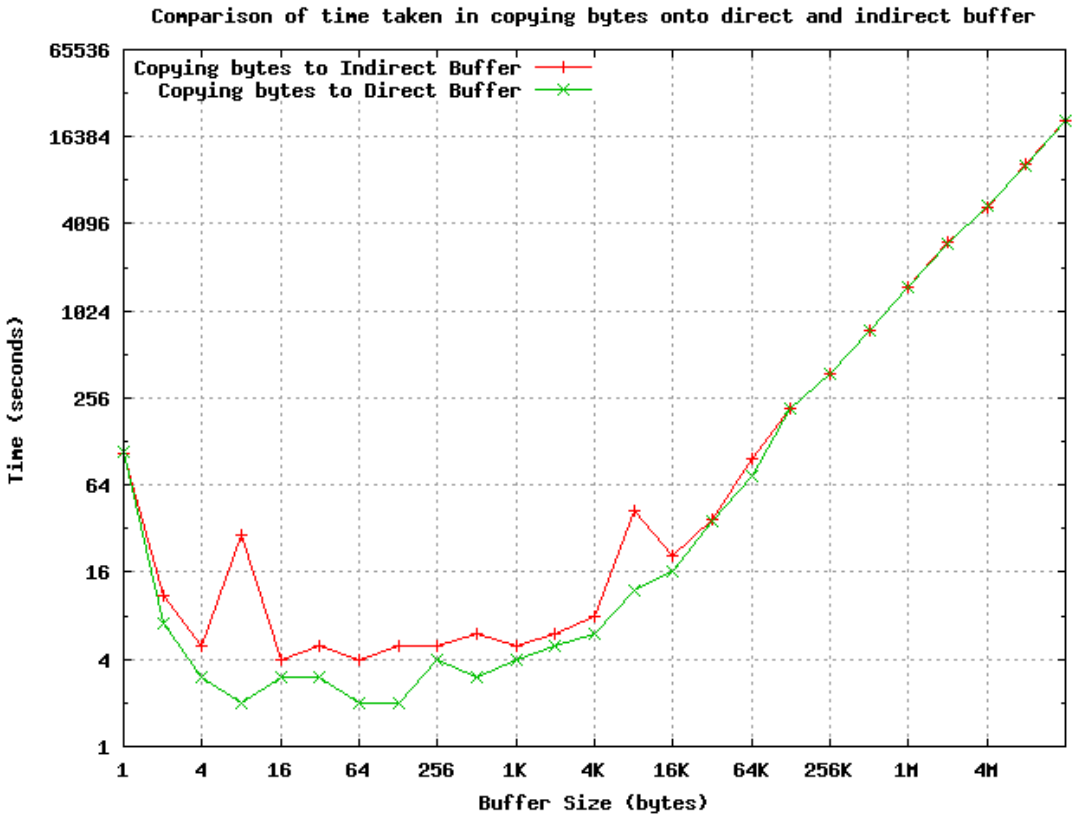

Figure 7: A Comparison between Direct and Indirect Buffer for copying `N` bytes

The graph in Figure 7 shows the time taken for copying `N` bytes onto the buffer. It is clear that the direct buffer is taking less time until the message size reaches 128 Kbytes, and after this point, the difference is negligible. Thus, the rule of thumb is to use direct buffers in the case where the buffers are reused frequently in the application. Such re-use would ensure faster I/O and prevent memory leaks that may occur because the direct buffers are not allocated on the JVM heap and are not subject to garbage collection. This means that the

JVM requests the underlying operating system to get the memory and it is not possible for the JVM to check the availability of the memory before requesting it. Thus the frequent creation of a direct buffer may cause the system to crash and is not recommended. On the other hand, if there is a need to frequently create buffers, then the use of indirect buffers is recommended.

The Buffer API has a corresponding buffer class for each of Java's basic datatypes that extends the base `Buffer` class. `ByteBuffer` is the corresponding class to byte datatype. The use of this class is recommended for building a message passing system because it has to support the transfer of all basic datatypes and Java objects. Java objects can be serialized to byte arrays before storing them onto the `ByteBuffer` class. Moreover, the read and write methods of the `SocketChannel` class only transfer data to and from the `ByteBuffer` class respectively.

The super-class `Buffer` provides three utility variables, called index, limit, and capacity. When the buffer is created, limit is undefined, capacity is the argument provided to the allocate method called for the creation of a buffer, and index is set to zero. For example, copying four bytes onto the buffer would set the index to four. To read the bytes copied onto the byte buffer, a utility method `flip()` is called, which sets the limit to the current index, and sets the index to zero. There are other utility methods like `clear()` and `rewind()` too. A detailed discussion of the Buffer API can be found in [16][15].

2.3.5 The Improved Performance Provided by Java I/O

NetPIPE [17] is a network protocol independent performance evaluation tool originally developed by Ames Laboratory [18]. NetPIPE provides information about the time required to transmit a given data block of a certain size to its destination, the maximum attainable throughput by an application, the maximum throughput for a given data block size, the communication overhead due to the legacy protocol stacks, and the latency of a communication protocol associated with a network interface. NetPIPE uses ping-pong like transfers for each data block. It increases the transfer block size from a single byte to larger blocks until the transmission time exceeds one second. Specifically, for each block size `c`, three measurements are taken for block sizes `c-p` bytes, `c` bytes and `c+p` bytes, where `p` is a perturbation parameter with a default value of three. This allows the examination of block sizes that are possibly slightly smaller or larger than an internal network buffer. The Java version of NetPIPE supports only the TCP protocol using the Java I/O package. For the purpose of our performance evaluation, we implemented the TCP protocol module using the Java New I/O package. This driver is used for the comparison shown in Figure 8 and Figure 9.

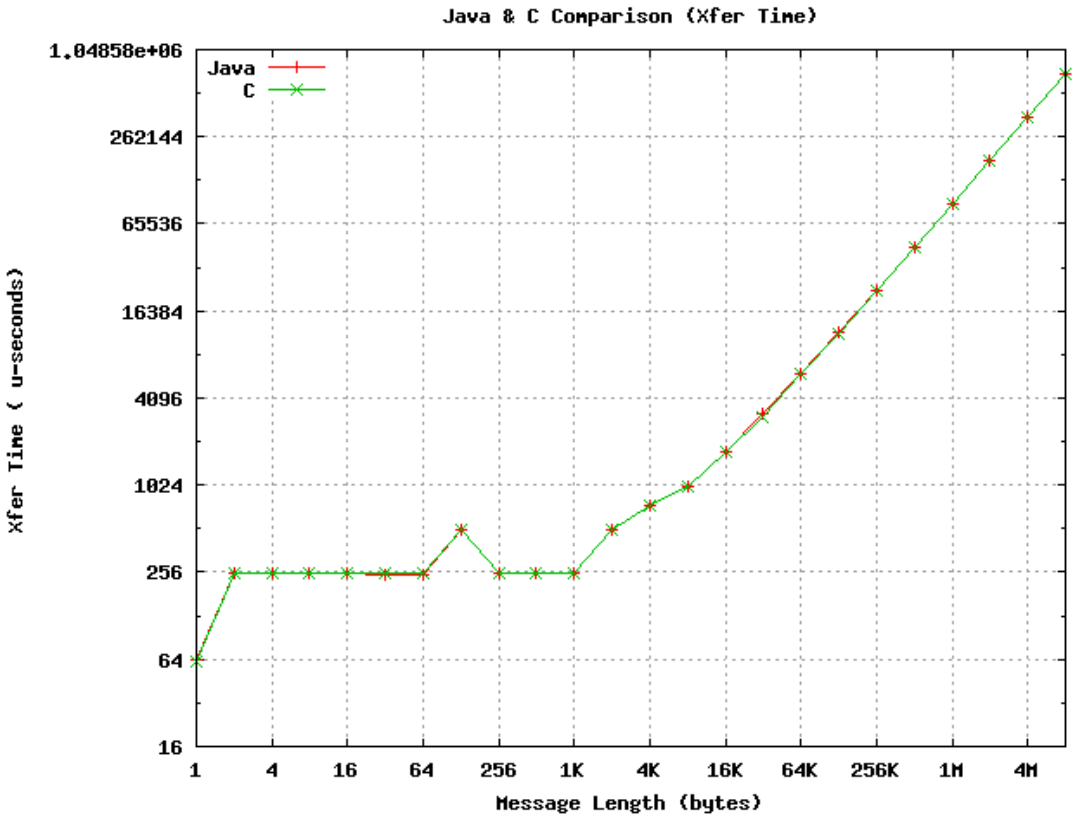

Figure 8: Transfer Time Comparison of Java Versus C (Netpipe benchmarks)

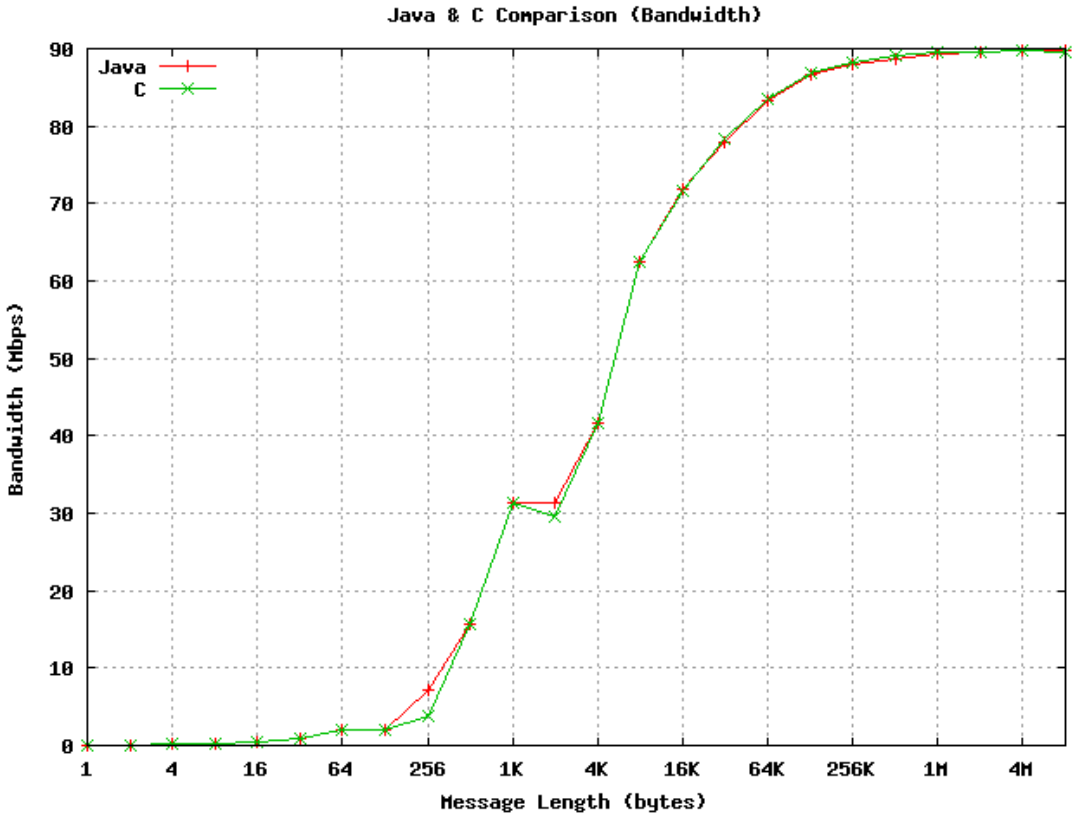

Figure 9: Bandwidth Comparison of Java Versus C (Netpipe benchmarks)

As can be seen in Figure 8 and Figure 9, the Java NIO driver performs equally well as the C driver. The latency, which we define as "the time taken in transmitting a single byte from the sender to the receiver" for both the drivers, is 62 microseconds. The peak bandwidth achieved by both the drivers is approximately 90 Mbps. A detailed discussion on the comparison can be found in [19]. The comparison shown in Figure 8 and Figure 9 provide evidence to the hypothesis made in Section 1.1 that a message passing system in Java would potentially provide the same performance as C.

2.3.6 Summary

In this chapter, the motivation for developing a message passing system has been discussed. Java has come a long way, and the release of the NIO package has added much-needed functionality for Java developers. Java NIO provides a scalable non-blocking I/O and a Buffer API that supports the transfer of the basic data-types as well as the Java objects. The comparison presented in Figure 8 and Figure 9 show that the Netpipe drivers for Java and C perform approximately the same. This point-to-point comparison presented so far is a simple one, and to assume that a message-passing library in Java will perform as well as C is still in question, but nonetheless it is encouraging to implement such a system using Java.

3 Background and Review

3.1 Introduction

In the past, there has been a significant amount of effort in developing Java message passing systems. Most of the systems were experimental and are no longer supported or in other cases, the software is not available. In this chapter we first discuss the pros and cons of each of the approaches taken to develop such message passing systems, and then we categorize each of the projects as past or present ones. The approaches used to implement a Java messaging system can be divided into three categories.

3.1.1 Using Remote Method Invocation (RMI)

Remote Method Invocation (RMI) is a Java API that allows the programmers to invoke methods on distributed remote objects. RMI uses sockets as the underlying communication medium and is primarily meant for client server interactions rather than the distributed peer processes. RMI can save a lot of programming time and effort for the developers of a message passing system, but is not the best option because of the performance issues associated with RMI. One of the reasons for these issues includes sending the basic data-types as objects, because all the arguments to the remote methods should be serializable. Secondly, at least one RMI registry should be running to locate distributed objects. The address and port of the RMI registry needs to be known to all processes that have to query the registry to locate distributed objects. J. Maassen et al in [20][21] discuss the performance issues associated with the RMI package in detail.

3.1.2 Using the JNI

The JNI (Java Native Interface) is a Java API that allows programmers to call C routines from their applications. Often developers of message passing systems use this package to interface their Java code to an underlying native MPI implementation. This technique saves a lot of additional programming and testing efforts but does not result in a portable code, which is the primary reason for implementing a message-passing system in pure Java. JNI also introduces an additional copying of the data between the Java and the native MPI code. This overhead is discussed in detail here [23]. Moreover, using JNI breaks the programming model of Java because there is no way to ensure code type safety. It also may lead to memory leaks because in C unlike Java, the programmer is responsible for allocating and freeing the memory.

3.1.3 Using Sockets

The most appropriate way to implement a Java message-passing library is to use sockets. This approach is considered to be a low level approach but ensures a portable and an efficient solution that is an important requirement of application developers. Java allows access to these sockets through two packages. The first one is the standard Java I/O package, which does not scale, as there is no support for non-blocking I/O. The second package is the Java NIO that has recently been introduced and provides programmers with non-blocking and scalable I/O.

3.2 Related Projects

In this section we discuss the Java based messaging systems. It was important to review each of these projects critically in order to learn from them and experiences they gained in order to avoid reinventing the wheel or the mistakes that they made. We have divided the projects into current (active) and past (in-active) sub-sections.

3.2.1 Past Projects

This sub-section discusses the projects that were developed in the past and are no longer active.

3.2.1.1 JavaMPI

JavaMPI [24][25] developed by University of Westminster [26] was the first attempt to provide a Java binding to MPI. JavaMPI was based on a set of wrapper functions to the native MPI implementation using the NMI (Native Method Interface) that has been replaced with JNI in the later releases of the JDK (Java Development Kit).

This system made use of a Java-to-C Interface generator (JCI) to produce the necessary C and Java method declarations files. Using JCI, the system generates similar files for the native MPI library. The native method declared Java file generated contained the interface used in the Java parallel application. The automatic generation of the Java bindings using the native MPI library resulted in an almost compatible Java binding to the MPI-1 specification. The library came with shell scripts to help start the parallel

processes over the remote hosts. Making use of the JCI tool saved programming effort but did not result in a portable solution. The source-code of this system is no longer available.

3.2.1.2 MPIJ

MPIJ [28] developed as part of Distributed Object Group Management Architecture [27] at Brigham Young University [29], was an implementation of MPI in Java. MPIJ implemented a large subset of the MPI-1 standard. This project used RMI as the communication medium, though the primitive data-types were passed using the native marshalling. This technique allowed efficient transfer of the primitive data-types as well as Java objects. One of the interesting features of this library is the applet-based execution of the parallel processes. This relieves the administrator of the manual installation of the software. A more useful and realistic approach would have been to allow remote installation of the software from one machine. The software for this library is no longer available.

3.2.1.3 JMPI

JMPI [30] was an experimental implementation of MPI developed at the Architecture and Real-Time Lab at the University of Massachusetts [31]. This library implemented a large subset of MPI's functionality. It used RMI as the communication medium and supports the transfer of Java objects using the object serialization. This library supported the transfer of multi-dimensional arrays through the use of Introspection. Recognizing the performance issues with the RMI, KaRMI [32] has also been tried as the underlying communication medium. The library had no runtime infrastructure to

support bootstrapping parallel processes on remote hosts. The software for the project is available at [33]. The architecture of this system is shown in Figure 10.

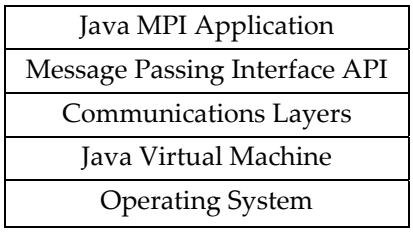

Figure 10: The Structure of JMPI

3.2.1.4 jmpi

jmpi [34] was a pure Java implementation of MPI-1 developed by Kivanc Dincer at Baskent University [35]. It was built on top of JPVM [36], which is a library that conforms to the PVM and is implemented in Java using the UDP sockets. This made jmpi unique in a sense that it was the only MPI library in Java that uses UDP as the underlying communication protocol, as opposed to using TCP. The MPI standard does not dictate the underlying communication protocol.

jmpi implements most of MPI's functionality, including support for derived datatypes, virtual topologies, attribute caching, as well as the point-to-point and the collective communication. A layered view of jmpi is shown in Figure 11. The source-code for this library is no longer available.

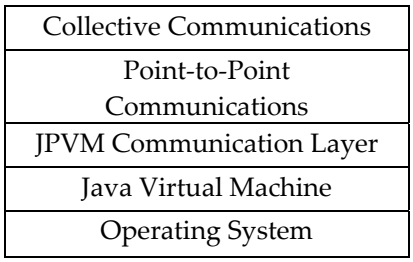

Figure 11: The Structure of jmpi

jmpi came with an instrumentation, analysis and visualization tool called JPVS (Java based Performance Visualization System) that allows application developers to monitor the progress of their applications and find out the performance issues, if any. This was the first attempt at providing a profiling tool for Java based messaging systems. Other tools that have been developed as part of C MPI libraries for debugging and profiling cannot be used with Java message passing systems, because these are not portable. An architectural view of JPVS is shown in Figure 12 taken from [37].

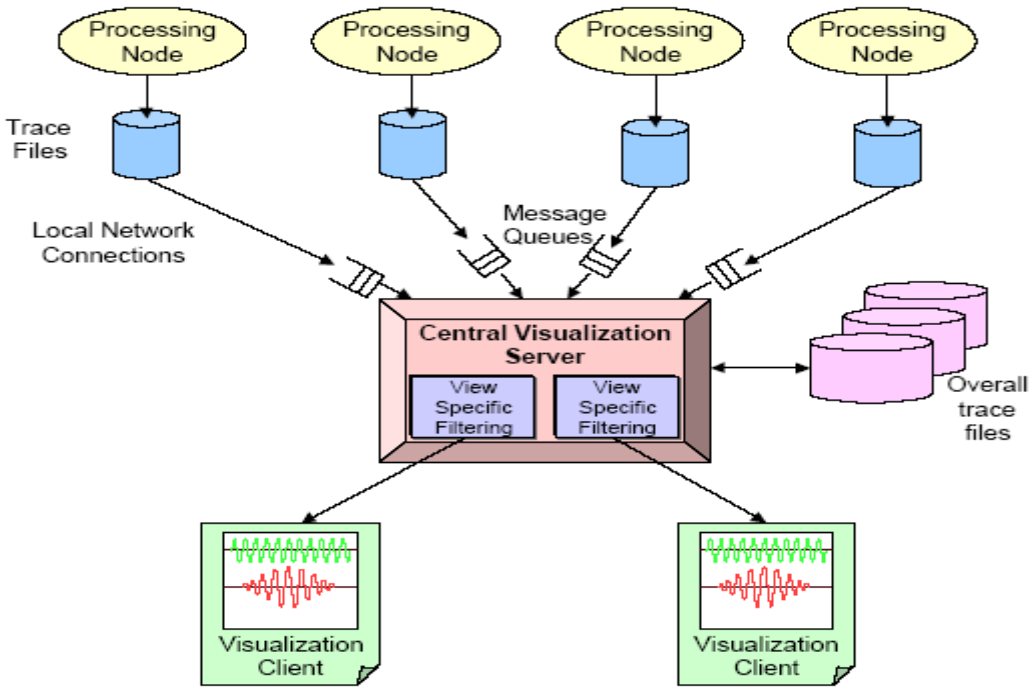

Figure 12: The Java based Performance Visualization System (JPVS) [37]

3.2.1.5 JMPI

According to [38], "JMPI is a commercial effort underway at MPI Software Technology, Inc. to develop a message-passing framework and parallel support environment for Java. It aims to build a pure Java version of the MPI-

2 standard specialized for commercial applications". Nor the software and neither any paper related to this project is available.

3.2.1.6 JUMP and PJMPI

JUMP and PJMPI [39] were two Java messaging systems developed by the Distributed and High-Performance Computing (DHPC) [40] at University of Adelaide [41]. According to [39], it is not always sensible to follow MPI standards for messaging systems in Java because they were written with procedural languages like C and Fortran in mind. For this reason, the project developed two flavours. One was PJMPI that follows the MPI standards. The other was JUMP, which was build upon Java object oriented features and did not strictly conform to the MPI standard. One of the interesting features of these libraries was that they come with a runtime infrastructure that allowed spawning parallel processes on remote hosts. This runtime infrastructure was based on RMI, and once the processes were started, the communication may take place using the sockets or RMI. The software for this project is not available.

3.2.1.7 JMPF

JMPF [42] was a Java message-passing framework developed by collaboration between Queensland University of Technology [43] and Centre of Development of Advanced Computing [44]. This framework was based on standard Java I/O package and used sockets as the communication medium. The authors argued in [42] that it was not easy for processes to keep a track of port and socket information regarding each of the process, thus they had devised a new abstraction called ports that allowed the management of

communication information like finding the address and the port number where the other process were listening. According to the MPI standard, the communicator deals with the complexity of managing the addresses and the port number, and the application developer has to only know the ranks of the processes and the total size of the communicator. JMPF did not follow the MPI standard, in fact was not even MPI-like.

The architecture of the JMPF is shown in Figure 13. The server was the entity that managed the so-called ports, and the clients got the relevant port information from the server and then communicated directly with other clients. The source code for this library is available [45] but this project is no more active.

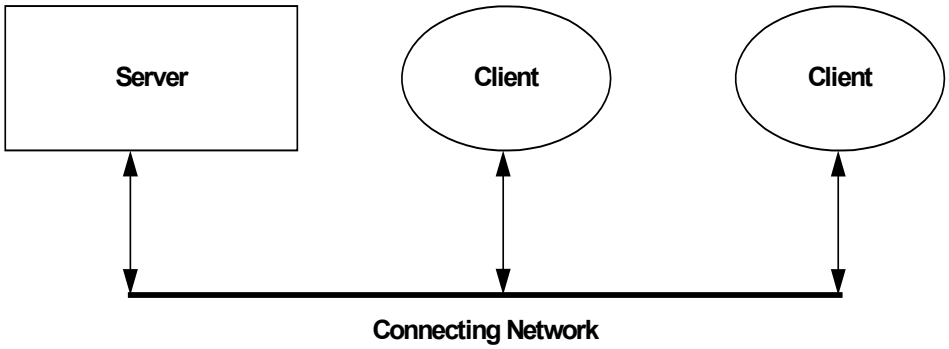

Figure 13: The Architecture of JMPF

3.2.1.8 JMPP

JMPP [46] was a message-passing package developed by National Chiao-Tung University [47]. This package was based on RMI and implemented the most of the MPI standard. The interesting feature of this library was the layered approach. In the reference implementation, the communication layer called ADI (Abstract Device Interface) used RMI but could be replaced with

other communication medium like sockets. Another interesting feature of this library was that it implemented two transfer protocols to support four modes of communication that are mentioned in the MPI standard. A so-called short protocol supported the standard and ready modes, whereas a so-called long protocol supported synchronous and buffered modes of message passing. The architecture of this library is shown in Figure 14.

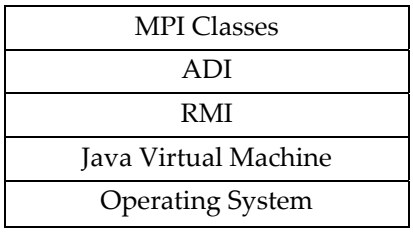

Figure 14: The Architectural View of JMPP

3.2.1.9 PJMPI

PJMPI [48] was a pure java implementation of the MPI standard developed at Shanghai University [49]. This library was based on the standard Java I/O package. A server socket was started at each of the participating node, which accepted incoming connections to form a point-to-point connection. `DataInputStream` and `DataOutputStream` were used to access the sockets for writing and reading respectively. The Java objects were communicated using the derived datatypes feature of the MPI standard. An interesting feature of this library was the runtime infrastructure. The library had a daemon called PJMPI daemon that ran over each of the machine that wished to participate in the overall execution of the parallel application. The status of the daemon could be checked visually from the client machine that was used to start up the process through a GUI called PJMPI control centre. Dynamic class loading was used to load the classes into the daemon machines JVM for execution. The software for this project is not available.

3.2.1.10 MPJava

MPJava [50] was a message-passing library that was implemented using the Java NIO package by The University of Maryland [51]. This package demonstrated that Java messaging systems based on NIO achieves performance comparable to that of C or Fortran message passing libraries. The runtime infrastructure consists of shell scripts that allowed processes to be started Linux-based nodes.

Figure 1, in [50], shows the ping-pong performance comparison between the `java.io` (`bytes`), `java.io` (`doubles`), MPJava and LAM-MPI 6.5.8. It shows that the LAM-MPI implementation attained greater throughput than MPJava for message sizes up to about 1000 `doubles`. For message sizes larger than 7000 `doubles`, MPJava attained a greater throughput than LAM-MPI. These results are suspicious because a Java messaging system cannot perform better than a C message passing system. The reason is that both C and Java use the underlying operating system TCP stack to communicate over the network, but Java adds an additional layer, which is the JVM. As a message passing system developer, we aim to keep the overhead of this additional layer minimal but it cannot be avoided. The test environment consisted of a cluster of Pentium III running Red Hat Linux 7.3. The Linux kernel used on the nodes is not mentioned in the paper, but there are some known performance issues of LAM-MPI with the Linux kernel 2.2.x. Further details about this can be found in [52]. The source code of MPJava has not been released publicly.

3.2.1.11 CCJ

CCJ [53] was a MPI-like message-passing library implemented on top of RMI developed at Vrije Universiteit Amsterdam [54]. This library did not strictly follow the MPI API in order to make use of the object-oriented features of the Java language. This library supported the transfer of the Java objects, as well as the basic data-types. Group communications had been implemented using the Java's multi-threading model, where a thread used scatter/gather like operations to send not only the arrays but also the Java objects to all the other threads within the same thread group. The library could also use Manta [22], which uses Myrinet as the underlying networking hardware. The benchmarks presented in the paper [53] show the overheads incurred using standard RMI.

3.2.2 Present Projects (Active Projects)

In this sub-section, we discuss each of the currently active projects briefly. We also discuss some of the projects that are using these Java messaging systems.

3.2.2.1 M-JavaMPI

M-JavaMPI [55] is a messaging system developed at The University of Hong Kong [56] that uses JNI to interact with underlying native MPI library. One of the important features of this library is that it supports process migration using the JVMDI (JVM Debug Interface) [57]. Figure 15 shows that the Java MPI program is compiled into the bytecode, which is modified by the pre-processing layer to insert state restoration code as the try-catch block. The Java-MPI API layer provides the necessary MPI functionality. This layer acts

as a client to the messaging layer (Restorable MPI Layer) that is responsible for delivering messages. The migration layer allows capture and saving of the execution state for later execution. The migration layer also handles the reconstruction of the communication channels in case a fault occurs.

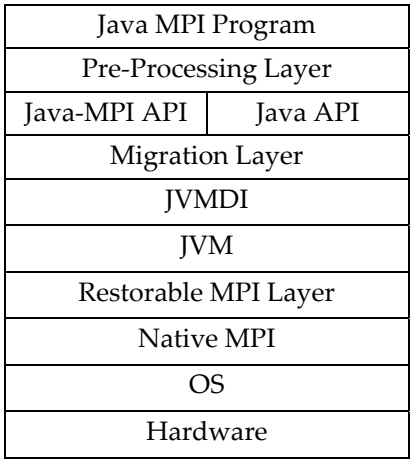

Figure 15: A Layered View of M-JavaMPI

The support for process migration is important for many applications because a minor fault may result in hours of wasted computation. Currently in J2SE 1.5, JVMDI [57] is deprecated in favour of JVMTI (JVM Tool Interface) [58], which means that JVMDI will be removed from the next major release. This essentially means that a part of this system will have to be re-written to make use of the new debugging interface. The source-code for this project is available [59]. This project has been subsumed into another project G-JavaMPI, a middleware for the Grid having support for process migration [60].

3.2.2.2 mpiJava

mpiJava [61] is a Java messaging system that uses JNI to interact with the underlying native MPI library. The project started in 1997 at NPACI (Syracuse University [64]), later moved to University of Florida [65] and is currently

being pursued at the Indiana University [66], with one of its collaborators in the Distributed Systems Group, University of Portsmouth [67]. This project is not strictly following the MPJ API specification [63], but it is slowly moving towards the standard.

During the implementation of earlier versions of the software, some conflicts between the JVM and the underlying MPI implementation were reported but the situation has improved with the evolution of the JVM. Initial versions of this software transferred only the primitive data-types, but the current version supports the transfer of Java object through the automatic serialization. [62]. mpiJava uses Perl wrapper scripts of the native MPI scripts to remotely start the execution of the processes on the remote nodes.

More recently, a native mpjdev (Native MPJ device driver) has been written in order to separate the implementation logics of mpiJava from the point-to-point and collective communications layers. This essentially means that the mpjdev driver developed in pure Java can be plugged into the already developed point-to-point and collective communications layer.

mpiJava has been widely used by the Java developers and HPC community. It is used as a teaching tool [76], a library that supports the development of performance measurement and analysis systems of parallel applications [71], and for simulating parallel applications [74][75]. Some in the community has found it hard to install [72] and some have provided execution helper scripts [70] to run it.

3.2.2.3 MPP

MPP [77] is an implementation of a subset of the MPI standard using Java NIO. This system has been developed at University of Bergen [78] as part of a larger project called Matrix Toolkits for Java (MTJ). Though MPP is built using Java NIO, it does not make use of the selectors to perform non-blocking I/O. In the non-blocking functions, a separate thread is started to handle communications. On the other hand, it does make use of the buffering API provided by Java NIO to support the transfer of primary data-types only. According to [77], MPP achieves 10 Mbytes/s over a 100 Mbps connection, and over 100 Mbytes/s over 1 Gbps connection. The runtime infrastructure consists of a shell script that uses SSH utility to start the processes on remote nodes.

3.2.3 Summary

In this chapter we have reviewed current and past Java message-passing systems. It is clear that the only real choice for application developers who want to port/develop their applications in Java is mpiJava. As mpiJava uses the native MPI as the communication medium, the application developers cannot benefit fully from the portability of Java. Moreover, JNI has drawbacks, which restricts the performance of mpiJava.

4 MPJ Design

4.1 Introduction

In this chapter the layered design of MPJ and its runtime are discussed. This chapter also outlines the design goals and associated issues.

4.2 Design Goals

The high-level design goals of MPJ are:

- Portability,
- Standard Java – we assume no language extensions,
- High-performance,
- A modular architecture, that is layered and has support for pluggable drivers for specialised devices.
- Support for the object oriented programming paradigm that allows application developers to program their applications at a higher level of abstraction.

4.3 Generic Design

MPJ is structured into a layered design to allow incremental development, and provide the capability to update and swap in/out layers as needed. Figure 16 shows the layered structure of MPJ.

High Level MPJ	Collective operations Process topologies
Base Level MPJ	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 Buffer packing/unpacking JNI Wrapper
Communication medium	Java NIO and Thread APIs Native MPI Specialised Hardware Library (For e.g. VIA communication primitives)
Process Creation and Monitoring	MPJ service daemon Java Reflection API to start processes Dynamic Class loading

Figure 16: The Layered Structure of MPJ

Figure 16 shows a generic layered view of the messaging system. The high and low-level layers rely on the MPJ Device level for actual communications and interaction with the underlying networking hardware. In other words, the MPJ Device level acts as a driver for different networking hardware, represented by the communication primitives layer that can be Java NIO, native MPI or some specialised communications library.

4.4 Instantiation of MPJ Design

Figure 17 shows an instantiation of the MPJ structure containing three different devices.

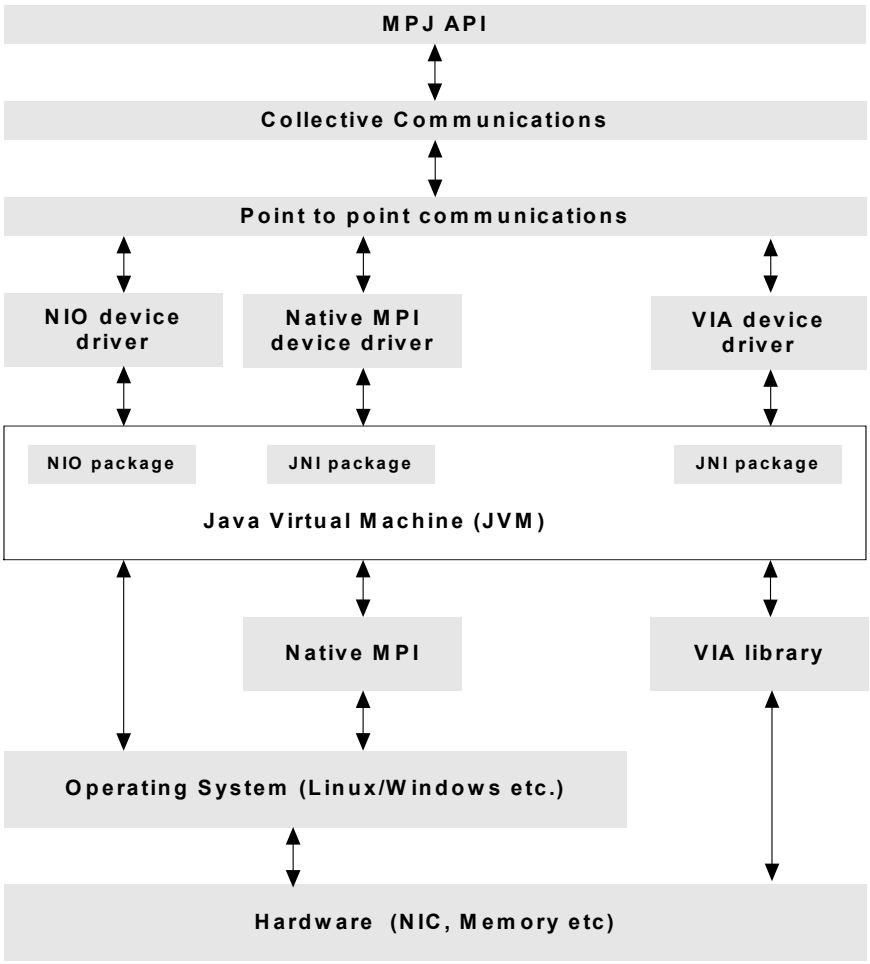

Figure 17: The Layered MPJ Design Showing Three Devices

4.4.1 Infrastructure

The bottom five layers of Figure 17 form the MPJ infrastructure, which is based on the concept of device drivers that is similar to that of MPICH [5]. MPICH uses various different communication devices, for example, ch_p4 is the communication device for BSD sockets, on top of which higher-level operations like point-to-point and collective communications are implemented. Using this approach, different drivers can be implemented for different devices, for example Java NIO, Java I/O, VIA (Virtual Interface Architecture) or native MPI. This also allows switching between various communication devices at runtime. The Java NIO device driver (hereafter

referred to as “mpjdev”) uses TCP sockets through the Java NIO package as the communication medium. The native MPI device driver (hereafter referred to as the “native mpjdev”) uses the JNI package to interact with the native MPI implementation. The VIA device driver uses the JNI to communicate with the VIA library. We envisage that in some cases, user will prefer to use the native mpjdev, which may perform better if there is some specialised hardware such as Myrinet/Infiniband/VIA.

4.4.2 Messaging API

The top three layers in Figure 17, the MPJ API, collective communications and point-to-point communications form the messaging API. The high-level messaging API provides an interface for the application developers to use. This interface hides all the underlying implementation details from users. The messaging API consists of the point-to-point communications, collective communications, and some utilities like communicators, process topologies, and derived datatypes. Table 1 shows an instantiation of the MPJ structure using mpjdev as the only communication device.

MPJ Point to Point and Collective communication: This layer manages communicators, provides virtual topologies, point-to-point and collective communication operations.
The Java New I/O device driver (mpjdev): mpjdev is implemented using the New I/O package. This layer is responsible for starting the device, providing a way for other similar devices to connect to it and connecting to other devices. This layer gets the configuration information from the runtime, For example, ports to bind the server socket to, rank of this processes, ranks of other processes, machine name/IP and ports of other processes.
Java Virtual Machine and Java libraries: This layer is the Java runtime that is required to execute Java programs. This layer provides the support for non-blocking I/O through the Java New I/O package.
Operating System (Linux/Windows)

Table 1: The Structure of Messaging API Using mpjdev

4.5 Design Constraints

A device driver implements a buffer packing/unpacking API because it is not possible to write the basic datatypes and objects into an open socket. Instead, the data is first copied onto the buffer and then a reference to this buffer is passed to the socket. The buffering API supports three types of read and write methods; `write()/read()`, `scatter()/gather()`, and multi-stride scatter and gather methods. These operations form the basis for supporting gather and scatter methods at the collective communications layer.

4.6 The Runtime Infrastructure

A runtime infrastructure solves the problem of starting the execution of the parallel application as multiple processes. There processes may execute over a cluster of nodes, or workstations present in LAN. Such an infrastructure is

necessary for a messaging system, as it is impossible for the user to login to each of the participating node, and start the process manually. We have divided the runtime infrastructure into two modules. The first module is the initiator module that allows the user to initiate the execution of the parallel application. The second module is the daemon module, which runs over the nodes that wish to participate in the overall execution. Figure 18 shows the interaction between the initiator and the daemon module.

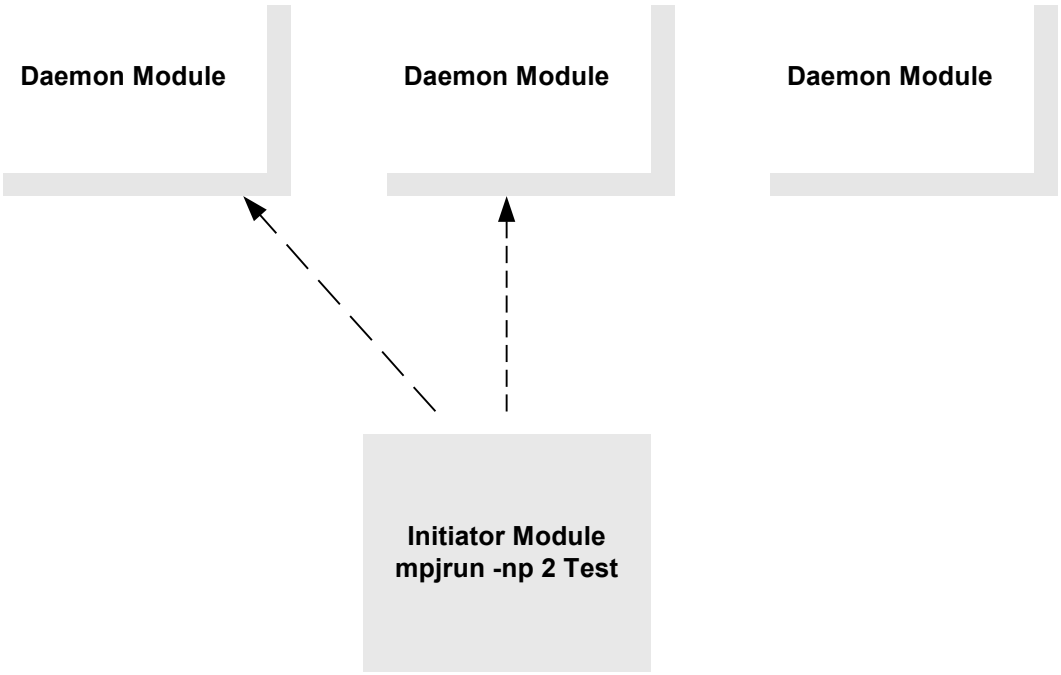

Figure 18: The Interaction of Runtime Modules

As part of MPJ, we have implemented a secure runtime infrastructure that allows a user to run the MPJ application over the remote nodes. This portable infrastructure allows different operating systems running on different hardware platforms to participate in the overall parallel job.

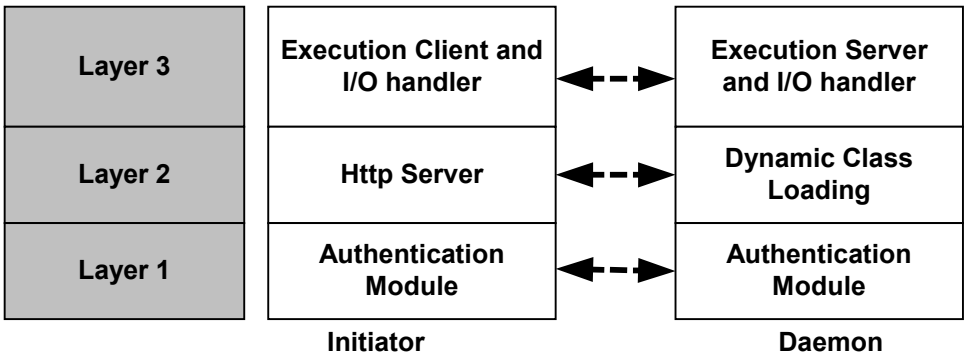

Figure 19: The Design of the Runtime Infrastructure

Figure 19 shows the layered structure of the runtime infrastructure. The initiator module runs on the node that starts the execution of the application. The daemon module runs on the nodes that are able to execute the processes. We discuss each layer of the runtime infrastructure in the sub-sections below.

4.6.1 Layer 1 (Authentication)

The authentication layer resides at the bottom of the runtime infrastructure because the first and foremost requirement is to authenticate user.

4.6.2 Layer 2 (Dynamic Class Loading)

Once the user has been authenticated, the next layer of the module downloads the binaries from the user machine so that they may be executed on the daemon node. We do not assume a shared file system, hence a dynamic class loading is required that is shown in the layer 2 of Figure 19.

4.6.3 Layer 3 (Execution of the process)

After layer 2 has loaded the binaries onto a node, the daemon executes the user MPJ job. The standard I/O and error streams are redirected to the user machine so that user may view any output or errors that the application may be producing.

4.7 Summary

In this section, the design of MPJ and its runtime infrastructure is discussed. MPJ has been structured to hide the implementation details from the application developers. Moreover, the design allows the application developers to choose amongst various communication devices. It is possible for other developers to add new communication devices to the MPJ library. The runtime infrastructure is layered in three main layers that form the basis of the interaction between the initiator and the daemon modules.

5 Implementation of MPJ

5.1 Introduction

This chapter discusses the implementation of MPJ. This chapter is divided into two sections that describe the implementation of mpjdev device driver and the runtime infrastructure.

5.2 The Implementation of mpjdev

The primary functionality provided by the device driver is the implementation of various communication protocols and buffering that allows the users to pack and unpack their data to/from the NIO buffers. Two packages have been implemented as part of the development of the mpjdev API. One package is mpjdev, which contains a Java NIO device driver, having a selector thread that assists in the non-blocking I/O. The second package is mpjbuf, which contains all the classes related to the buffering API for the mpjdev.

5.2.1 The mpjdev Communication Protocols

The mpjdev device driver encapsulates the protocols used for communication.

5.2.1.1 The Eager-Send Protocol

The Eager-send protocol is used by the message passing libraries for the communication of small messages typically less than 128 Kbytes. This protocol works on the assumption that the receiver has got an unlimited device level memory where it can store messages. There is no exchange of control messages before doing the actual data transmission, thus minimizing the overhead of control messages that may dominate the total communication time of small messages. Whenever a send method is called, the sender writes the message data into the socket channel assuming that the receiver will handle it. At the receiving side, there can be more than one scenario, depending on whether a matching receive method is posted by the user or not. If a matching receive method is posted, the message is copied onto the memory specified by the user (sketched out in Figure 20).

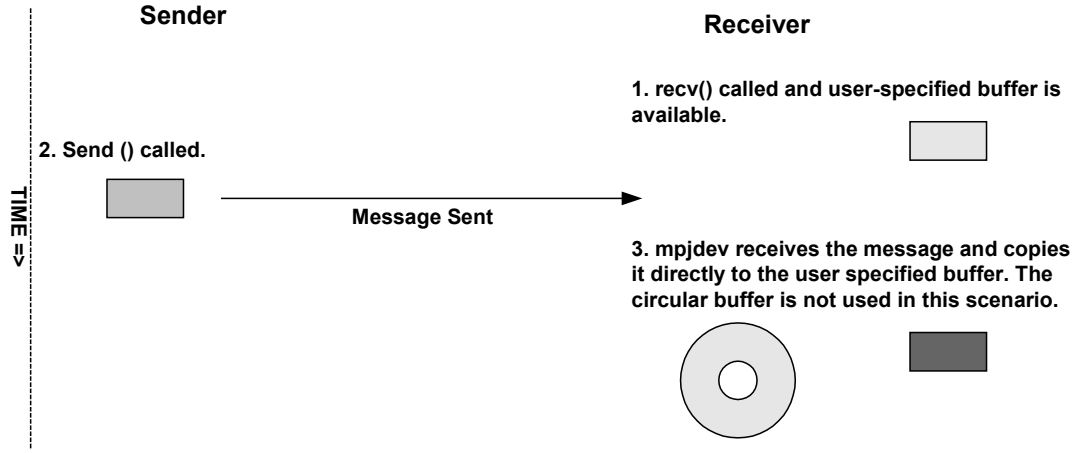

Figure 20: The Eager Send Protocol when a Matching `recv()` is Posted.

However, if a matching receive is not posted, then the mpjdev device driver stores the message temporarily in a circular buffer, and copies it to the user specified memory when, subsequently, the user calls the matching receive method (sketched out in Figure 21).

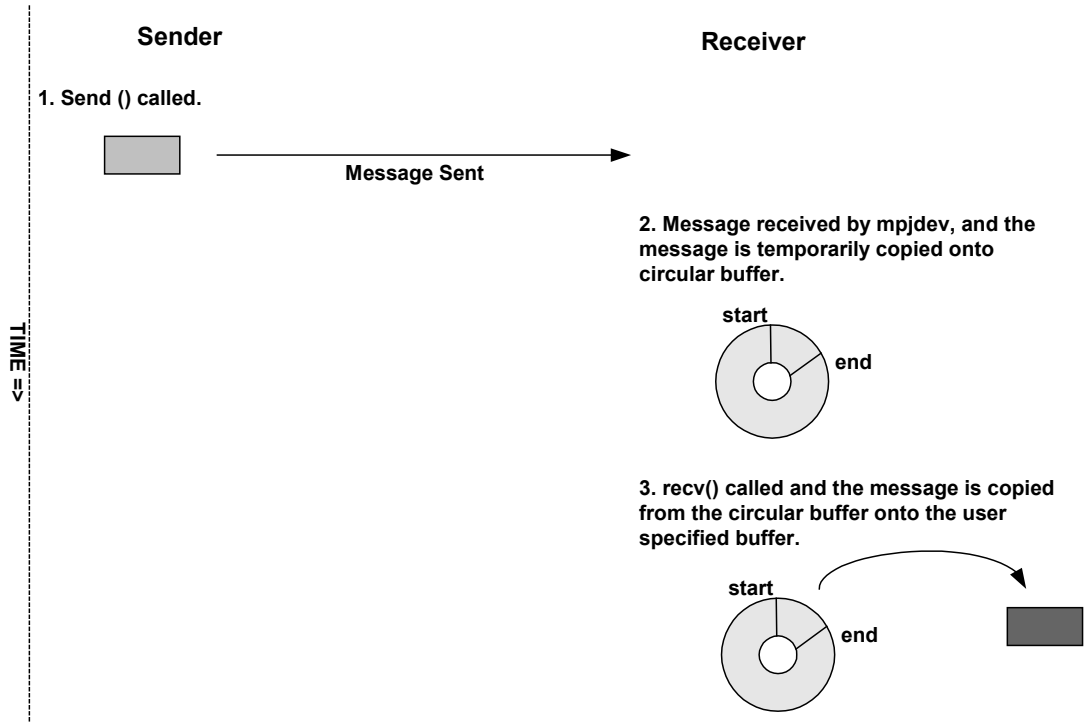

Figure 21: The Eager Send Protocol when a Matching `recv()` is not Posted.

The large circular buffer is currently a direct buffer. The mpjdev device driver associates a starting point and an ending point whenever it copies a message onto the buffer that is later used to read the message again. When the device encounters the end of the buffer, it starts to copy the message at the start point of the buffer assuming that first message copied to the buffer has been transferred to the user specified memory.

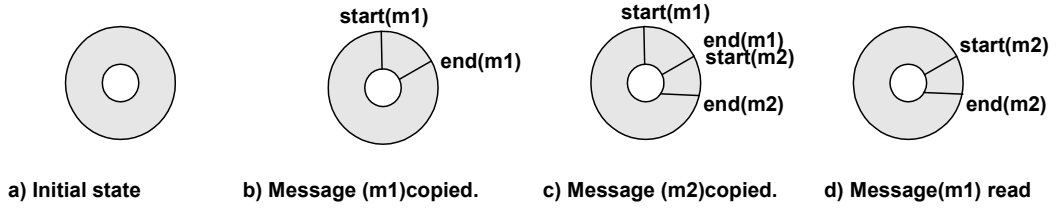

Figure 22: The Circular Memory Buffer.

5.2.1.2 The Rendezvous Protocol

The rendezvous protocol is used by the message passing libraries for communication of large messages, typically greater than 128 Kbytes. There is an exchange of messages between the sender and the receiver before the actual transmission of the data payload. The overhead of this exchange of messages is negligible in terms of the overall communication cost.

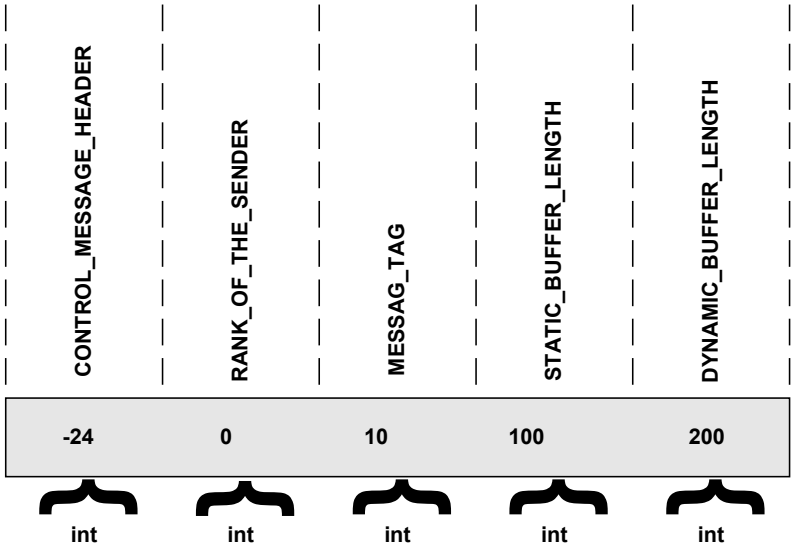

Figure 23: The Control Message sent by the Sender.

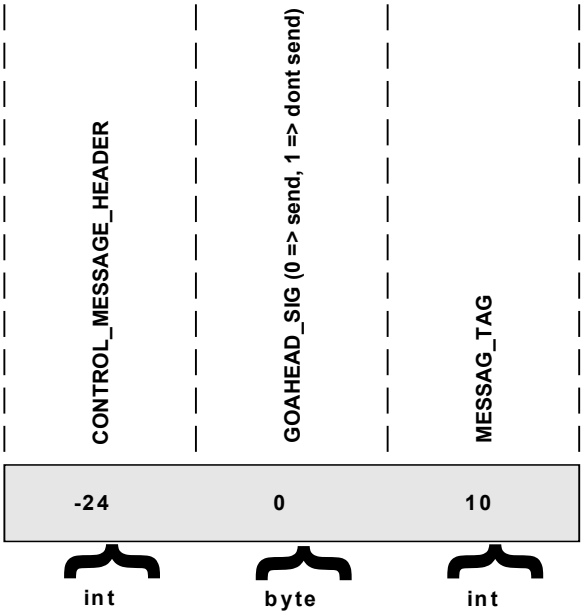

Figure 24: The Format of the Control Message Sent by the Receiver.

The rendezvous protocol when the sender intends to send a message, it first sends a control message, see Figure 23 for the format of the control message. The receiver after receipt of the control message first checks if a matching receive method is called or not. If it is called, see Figure 25 for exchange of messages, then the receiver sends a control message back, see Figure 24 for the format of the control message.

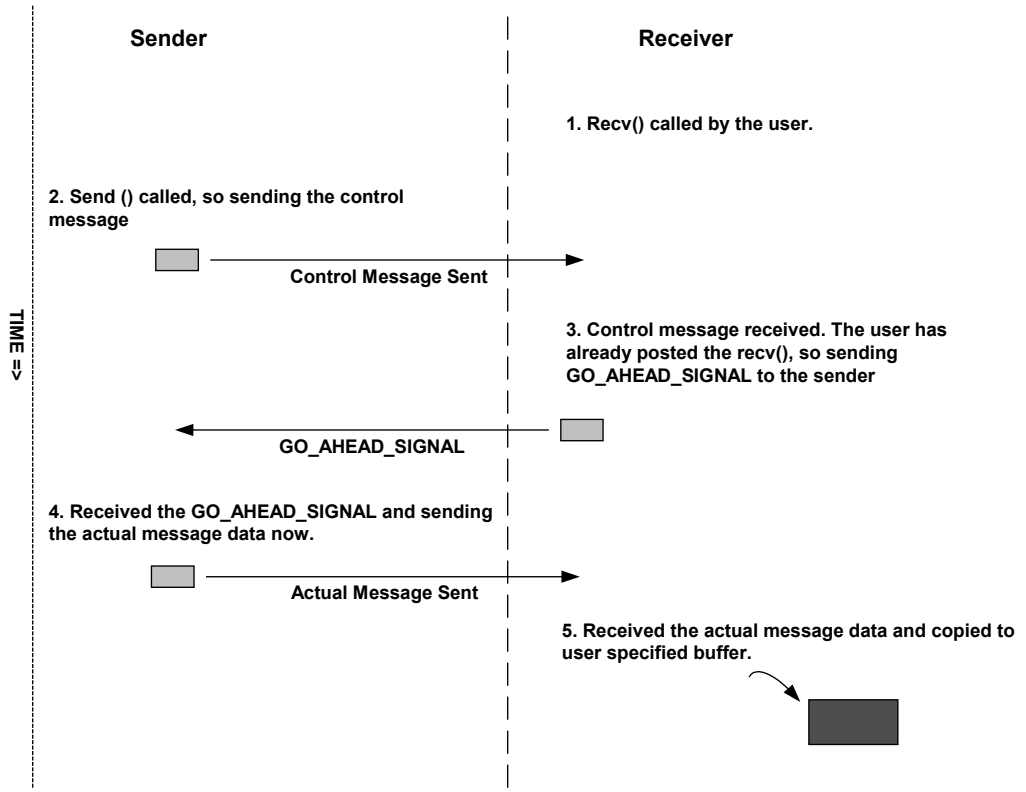

Figure 25: The Rendezvous Protocol when the `recv()` is First Posted.

If the `recv()` is not posted, see Figure 26 for exchange of messages in this scenario, then the control message is sent back to the sender when a matching receive method is posted. A matching receive has to be posted in this protocol to save the mpjdev memory. When the sender receives the acknowledgement message back from the receiver, it sends the actual data that is later received by the receiver.

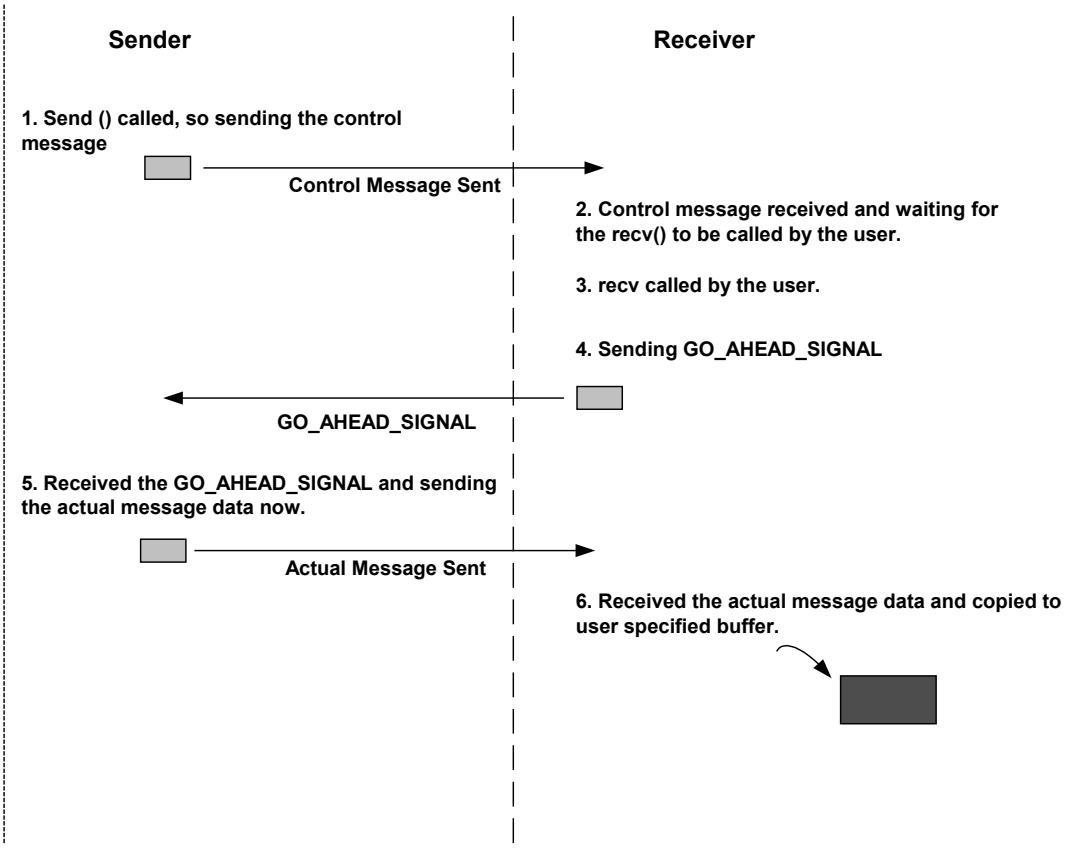

Figure 26: The Rendezvous Protocol when the `recv()` Method is not Posted.

5.2.1.3 Shared Memory (Internal process communication)

This protocol is used when a process is sending a message to itself. The ideal situation for two processes running on the same machine would be to use shared memory paradigm to pass data. This is not possible in Java where two JVMs running on the same machine can only send the message to each other by using sockets and thus the communication cost is dominated by the bandwidth of memory bus. This is because each of the JVM has its variables on its heap inaccessible to other JVM instances running on the same computer. Thus, the shared memory communication is restricted to same process communication only (one JVM).

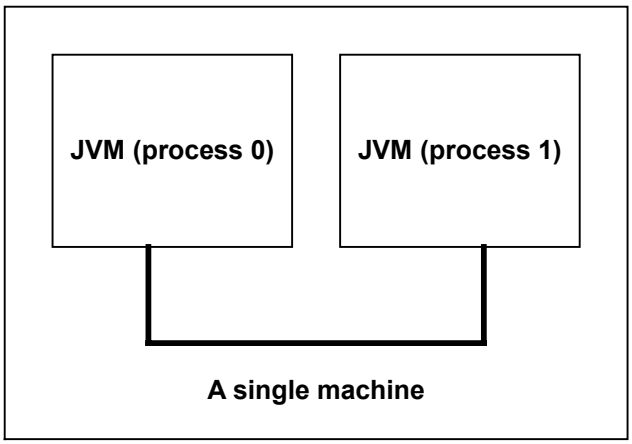

Figure 27: Shared Memory Communications.

5.2.2 The Buffering API

The buffering API allows user to pack/unpack the data to be sent to other processes. The primary reason for implementing this API is that the sockets do not directly access the memory and thus are unable to write/read the basic datatypes. Moreover, without such a buffering API, a message passing system can become quite complex for the users to manage because of the absence of pointers and the type safety feature of Java, which does not allow "`void *`" such as C. Most of the complex operations, such as communicating objects and handling gather/scatter operations; used at the higher levels of the library, are also supported by this buffering API.

A buffer object consists of two data storage structures. The first is a static buffer, in which the underlying storage primitive is the `ByteBuffer` class. The second is a dynamic buffer where a byte array is the storage primitive. The size of the static buffer is predefined, and can contain only primitive datatypes. The rationale behind this is that it is possible to calculate the number of bytes required before copying the data onto the buffer. Whereas on the other hand, the dynamic buffer is used to copy serialized Java objects, in

which case it is not possible to determine the length of the serialized objects beforehand. As a convenience to the user, Java basic datatypes may also be stored in dynamic buffers. According to [79], in general writing to or reading from a static section of the buffer is much faster than the corresponding operation on a dynamic section of the buffer. In general, use of a dynamic buffer is only recommended for short or infrequent message exchanges.

5.2.2.1 Buffer Sections

A message consists of zero or more sections. Each section can hold elements of the same type, basic datatypes or Java objects. A section consists of a header, followed by the actual data payload.

To create sections, the buffering API provides utility methods like `putSectionHeader()`, which takes as an argument one of the datatypes (possible datatypes are shown in Table 2 and Table 3) and can only be invoked when the buffer is in a writeable mode. Once the section header has been created, then the data can be copied onto the buffer using `write()` method if contiguous elements are to placed onto the buffer, or using the `gather()`/`strGather()` methods if non-contiguous elements of Java arrays are to be copied onto the buffer. To read the section, the user invokes the `getSectionHeader()` method when the buffer is in a readable mode, which is later followed by a series `of read()`, for contiguous elements of Java array, or `scatter()/strScatter()`, for non-contiguous elements of Java array.

5.2.2.2 The Layout of Buffers

The data stored in a static buffer can be represented as big-endian or little-endian. This is determined by the encoding property of the buffer, which takes on of the values `java.nio.ByteOrder.BIG_ENDIAN` or `java.nio.ByteOrder.LITTLE_ENDIAN`. The encoding property of a newly created buffer is determined by the return value of `java.nio.ByteOrder.nativeOrder()` method. A developer may change the format to match the encoding property of the underlying hardware, which results in efficient numeric representation at the JVM layer. The overall layout of the static buffer is shown in Figure 28.

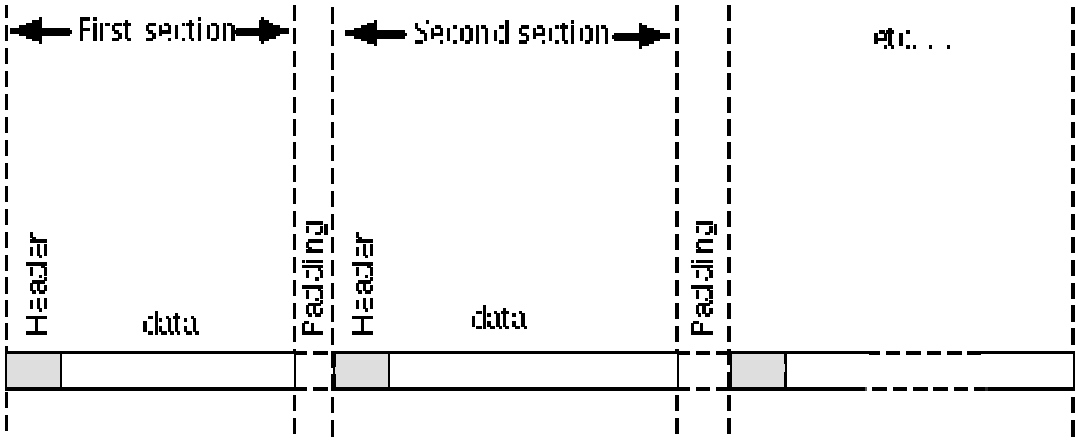

Figure 28: The Layout of a Static Buffer [79]

As shown in Figure 28, a message consists of zero or more sections. The message consists of a message header followed by the data payload. A padding that can be up to 7 bytes may follow a section if the total length of the section (header + data) is not a multiple of `ALIGNMENT_UNIT`, which has value 8. The general layout of an individual section in the static buffer is shown in Figure 29.

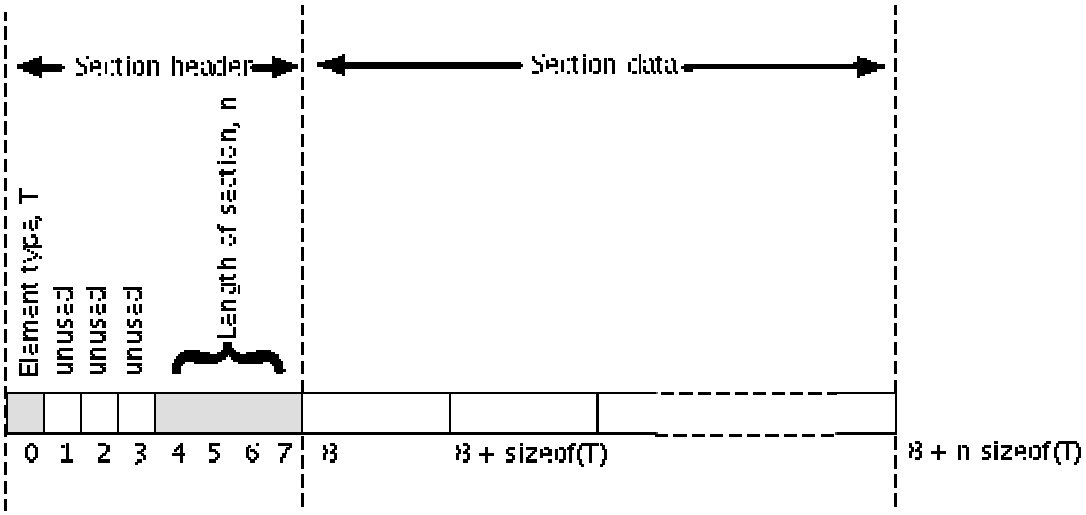

Figure 29: The Layout of a Single Section [79]

Figure 29 shows that the message header is 8 bytes long. The value of the first byte defines the elements' type contained in the section. The possible values for static and dynamic buffers are listed in Table 2 and Table 3 respectively. The next three bytes are not currently used, and reserved for possible future use. The next four bytes contain the number of elements contained in this section, i.e. the section length. This numerical value is represented according to the encoding property of the buffer. The size of the header in bytes is `SECTION_OVERHEAD`, which has value 8. If the section is static, the header is followed by the values of the elements, again represented according to the encoding property of the buffer. If the section is dynamic, the "Section data" is absent from Figure 29 because the data is in the dynamic buffer which is a byte array. The Java serialization classes (`java.io.ObjectOutputStream` and `java.io.ObjectInputStream`) dictate the format of the dynamic buffer.

Datatype	Possible Values
integer	`mpjbuf.Buffer.INT`
byte	`mpjbuf.Buffer.BYTE`
short	`mpjbuf.Buffer.SHORT`
boolean	`mpjbuf.Buffer.BOOLEAN`
long	`mpjbuf.Buffer.LONG`
float	`mpjbuf.Buffer.FLOAT`
double	`mpjbuf.Buffer.DOUBLE`

Table 2: The Datatypes Supported by a Static Buffer

Datatype	Possible Values
Java Objects	`mpjbuf.Buffer.OBJECT`
Bytes on dynamic buffer	`mpjbuf.Buffer.BYTE_DYNAMIC`
Shorts on dynamic buffer	`mpjbuf.Buffer.SHORT_DYNAMIC`
Booleans on dynamic buffer	`mpjbuf.Buffer.BOOLEAN_DYNAMIC`
Integers on dynamic buffer	`mpjbuf.Buffer.INT_DYNAMIC`
Longs on dynamic buffer	`mpjbuf.Buffer.LONG_DYNAMIC`
Floats on dynamic buffer	`mpjbuf.Buffer.FLOAT_DYNAMIC`
Doubles on dynamic buffer	`mpjbuf.Buffer. DOUBLE_DYNAMIC`

Table 3: The Datatypes Supported by a Dynamic Buffer

5.2.2.3 Packing/Unpacking Methods

There are three basic kinds of method for writing/reading data to/from a buffer section and for reading data from a buffer section. These methods are summarized in Table 4, and followed by an explanation of them.

Packing/Unpacking Methods	Section writing methods	Section reading methods
1. Write/Read Methods	`write(type[] src, int srcOff, int numEls)`	`read(type[] dest, int dstOff, int numEls)`
2. Gather/Scatter Methods	`gather(type[] src, int numEls, int idxOff, int[] indexes)`	`scatter(type[] dest, int numEls, int idxOff, int[] indexes)`
3. Gather and Scatter Methods for multi-strided regions	`strGather(type[] src, int srcOff, int rank, int exts, int srs, int[] shape)`	`strScatter(type[] dest, int dstOff, int rank, int exts, int srs, int[] shape)`

Table 4: Buffer Packing/Unpacking Methods

5.2.2.3.1 **Write/Read Methods**

The methods, shown in row 1 of Table 4, are used to write and read contiguous Java arrays of all the primitive datatypes including object arrays. The write method copies `numEls` values of the `src` array starting from `srcOff` onto the buffer. Conversely, the read method copies `numEls` values from the buffer and writes them onto `dest` array starting from `srcOff`.

5.2.2.3.2 Gather and Scatter Methods

The methods, shown in row 2 of Table 4, are used to write and read non-contiguous Java arrays of all the primitive datatypes including object arrays. The gather method copies `numEls` values of the `src` array starting from `indexes[idxOff]` to `indexes[idxOff+numEls]` onto the buffer. Conversely, the scatter method copies `numEls` values from the buffer and writes them onto dest array starting from `indexes[idxOff]` to `indexes[idxOff+numEls]`.

5.2.2.3.3 Gather and Scatter Methods for Multi-Strided Regions

The methods, shown in row 3 of Table 4, likewise transfer data from or to a subset of elements of a Java array, but in these cases the selected sub-set is a "multi-strided region" of the Java array. The specification is fairly complex, but these are useful operations for dealing with multi-dimensional data structures, which occur often in scientific programming. More details about these methods can be found in [79].

5.2.2.4 Buffer Modes

A buffer object has two modes, write and read. The write mode allows the user to copy the data onto the buffer, and the read mode allows the user to read the data from the buffer. It is not permitted to read from the buffer when it is in writeable mode. Similarly, it is not permitted to write to a buffer when it is readable mode.

The newly created buffer is always in a writeable mode. During this mode, the user may copy the data to the buffer and then call `commit()`, which puts the buffer in a readable mode. The user can now read the data from the buffer and put it back in writeable mode for any possible future use that is done by calling the `clear()`.

5.2.3 Binding It All Together

Having described the communication protocols and the buffering API, we now describe how the device drivers use the various communication protocols.

5.2.3.1 The Sending Process

The sending process is always initiated by a user thread, which means calling either the blocking or non-blocking send methods. Whenever the method is called, the send method first checks whether the destination process is equal to the current process. If it is, then the user thread uses the shared memory protocol to copy the message to itself. In case the destination process is not equal to the current process, then the message length is compared against the protocol switch limit, which is provided by the user to the device through the runtime configuration information. If the message length is smaller than the protocol switch limit, then the message is sent using the eager send protocol. If the length is equal or greater than the protocol switch limit, then the message is sent through the rendezvous protocol. As part of the rendezvous protocol, the sender first sends a control message to the receiver. The sender data flow of user thread for the rendezvous protocol in shown in Figure 30.

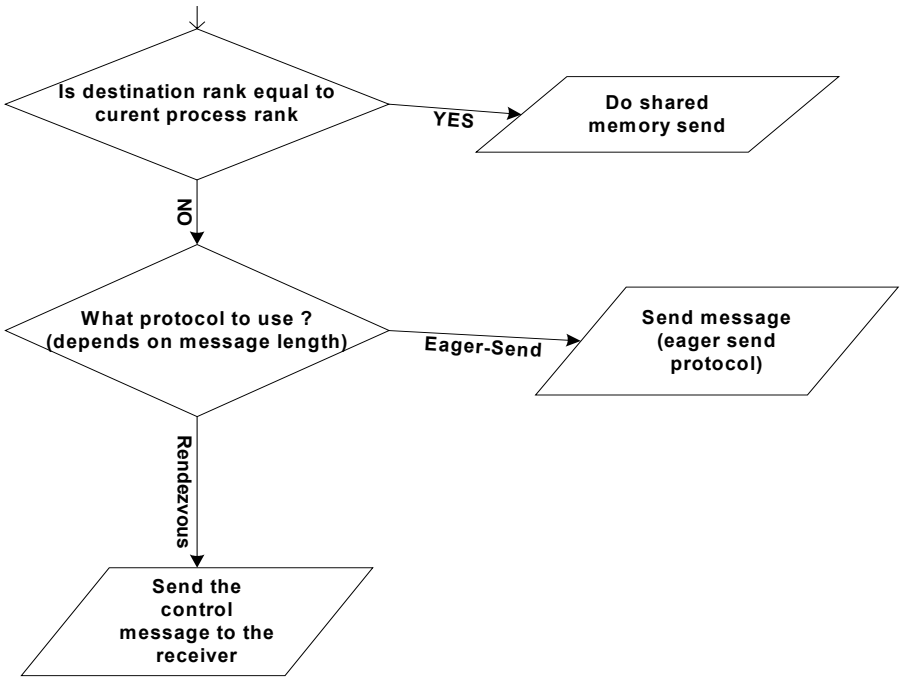

Figure 30: The Sending User Thread

The selector thread for the sending process is responsible for receiving the control message back from the receiver. If the receiver sends a `GO_AHEAD_SIGNAL`, as a response to the control message sent by the sender user thread, then the sender goes on to send the actual data, see Figure 31 for details.

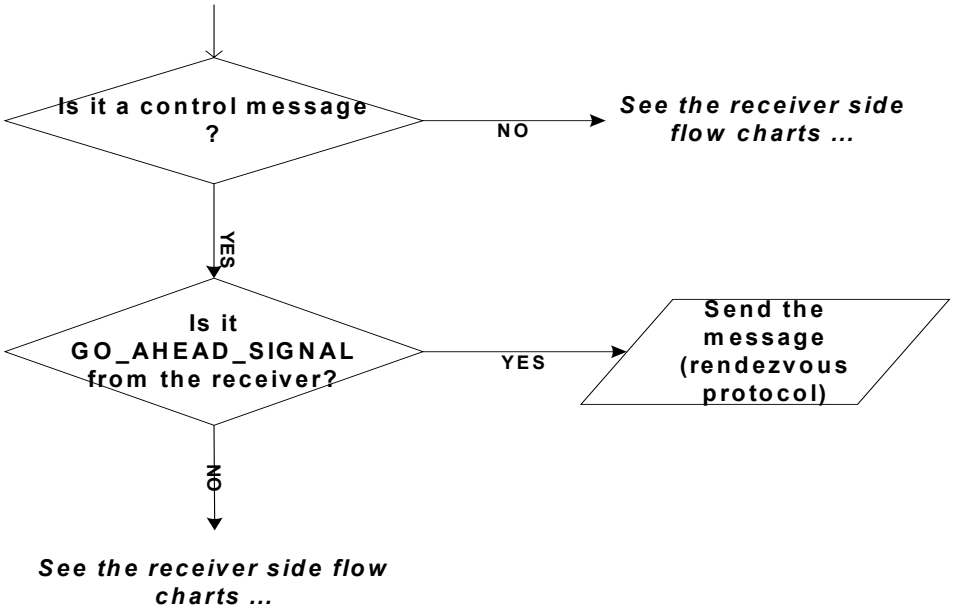

Figure 31: The Sending Selector Thread

5.2.3.2 The Receiving Process

In contrast to the sending process the receiving process may be initiated by the user thread, by posting a `recv()` call, or by the selector thread, which may receive a control message from the sender (rendezvous protocol) or receive the actual message data from the sender (eager-send protocol).

Figure 32 shows the data flowchart of the user thread at the receiving side. When the `recv()` method is called, the device first checks whether the source specified is equal to the current process rank. If it is, then the receiver uses shared memory protocol to receive the message. If it is not, then it first checks the queue (`recv()` queue) containing information about the `recv()` calls, whether the selector thread has already received a control message related to this particular `recv()`. If this queue does not contain this `recv()` call, then the user thread posts a `recv()` call and adds an entry into the `recv()` queue. If this queue contains a request related to this `recv()`, it means that the selector thread has already received the control message (for the rendezvous protocol) or has already received the actual message (for the eager send protocol). The protocol switch limit is compared to the message length to decide which protocol to use. If the message length suggests eager send, then the user thread receives the message, otherwise, a control message `GO_AHEAD_SIGNAL` is sent to the sender as a green signal to transmit the actual message data.

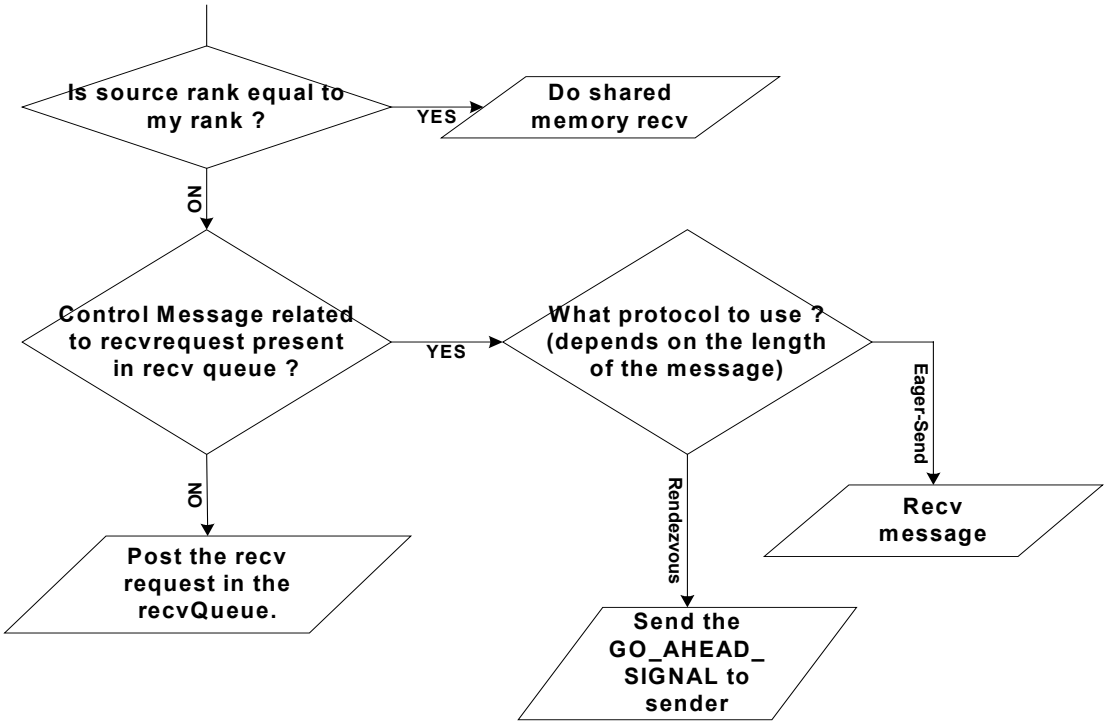

Figure 32: The Receiving User Thread

Figure 33 show the data flowechart of the selector thread at the receiving end. When selector thread receives a message, it first determines whether this is control message or not. If it is not a control message, then the sender has sent the actual data (rendezvous protocol) that is received by the receiver. This is the next step to the `GO_AHEAD_SIGNAL` sent to the sender by the user thread; see Figure 32 for details. If this is a control message, then it is checked to see if there is a matching `recv()` call present in the `recv()` queue, which is posted by a user thread or not. If there is a request then it means that the user has already posted the `recv()`, so a `GO_AHEAD_SIGNAL` is sent to the sender to give permission to send the actual data. If there is no matching `recv()` request found in the `recv()` queue, then it is posted and can be later checked by the user thread when the user posts the `recv()` method.

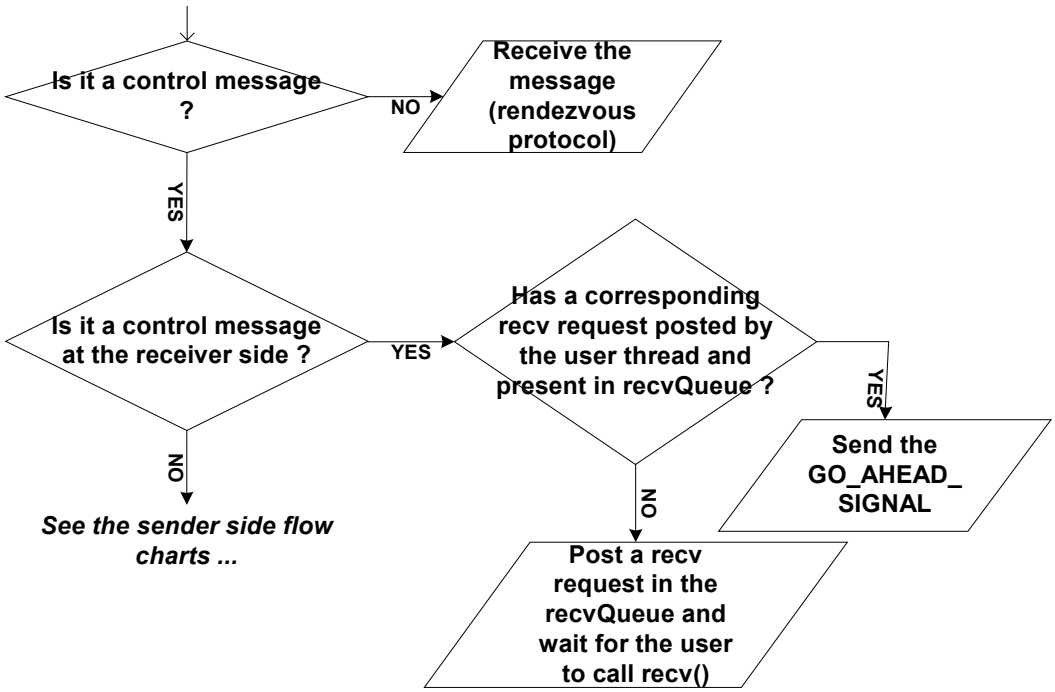

Figure 33: The Receiving Selector Thread

5.2.3.3 The Communication Primitives of mpjdev

Table 5 shows the method provided by the mpjdev API.

Modes	Send	Recv
Blocking	`send(mpjbuf.Buffer buffer, int dest, int tag)`	`Statusrecv(mpjbuf.Buffer buffer, int src, int tag)`
Non-blocking	`Req isend(mpjbuf.Buffer buffer, int dest, int tag)`	`Req send(mpjbuf.Buffer buffer, int src, int tag)`

Table 5: The Methods Provided by the mpjdev API

5.3 The MPJ Runtime

Currently, we are implementing a runtime to support the execution of the processes on remote hosts, in the first instance over a LAN. The key idea behind implementing a runtime is not to compromise the portability, which is

the basic motive to develop a message passing system. The implementation is in early stage and currently addresses how to start the execution of the parallel application. The runtime is divided into parts, an initiator module that starts the execution of the application, and a daemon that runs spawns the application on the remote nodes. Figure 34 shows an initiator module running on the head node of the cluster, and daemons running on the compute nodes and the two workstations present in the LAN. The initiator module starts the execution of the application by contacting the daemons.

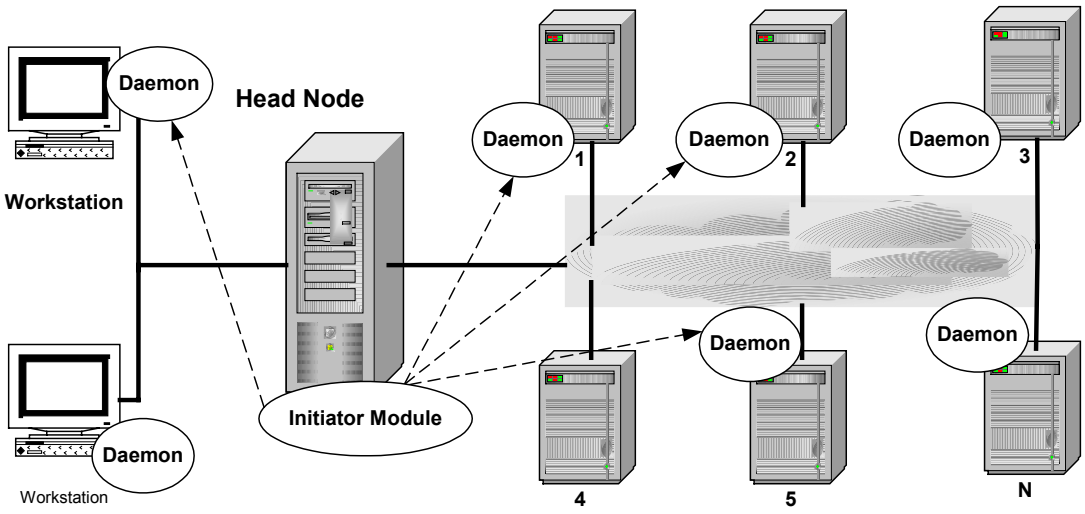

Figure 34: The MPJ Runtime Installed on a Linux Cluster and Two Workstations on LAN Running Windows

The runtime works as follows,

1. First of all, the administrator installs, configures, and starts the daemons on hosts, which may execute MPJ processes. The daemon is set up as a Windows service or in `inetd` under UNIX.

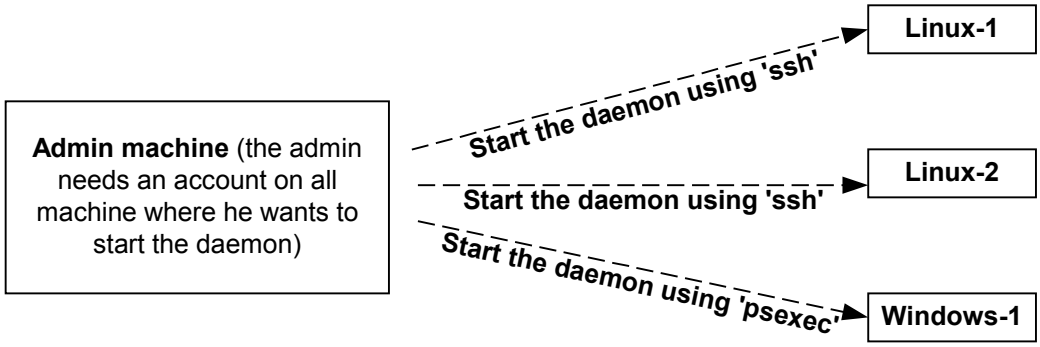

Figure 35: The Administrator Installing Daemons on Remote Nodes

2. The next step is to add users to this set-up. Only the administrator should be allowed to do this. The administrator adds the public keys of all the users under question to all the machines running the daemons.

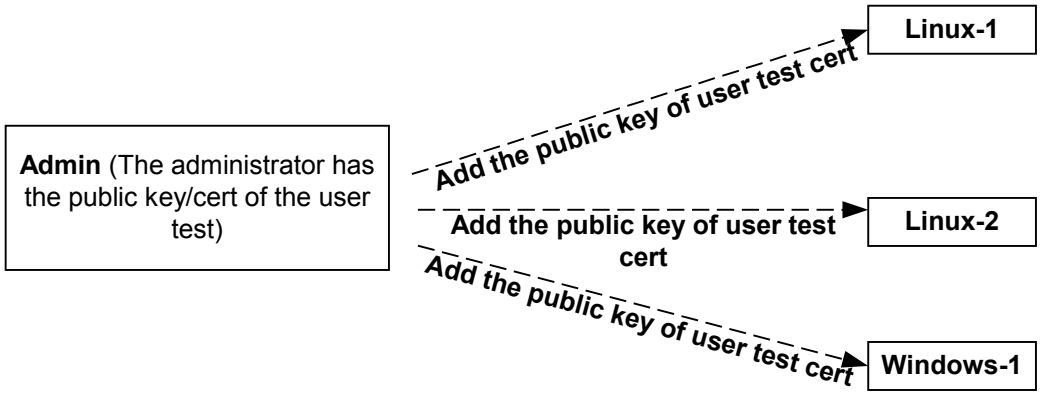

Figure 36: The Administrator Adding the Users on Daemon Machines

3. Once the daemons are up and running and the users are added, the next step is to determine how to start the processes. For example take a user 'test' trying to execute class `Test.java` on 6 nodes. This `Test.java` uses `mpjdev` and `mpj` packages imported from `mpjdev.jar` and `mpj.jar`. We do not assume a shared file system, which essentially means that there must be a way to download/copy the binaries along with the dependant classes and libraries from the user machine. The downloading/copying feature is explained in more detail in the sub-section 5.6.1.

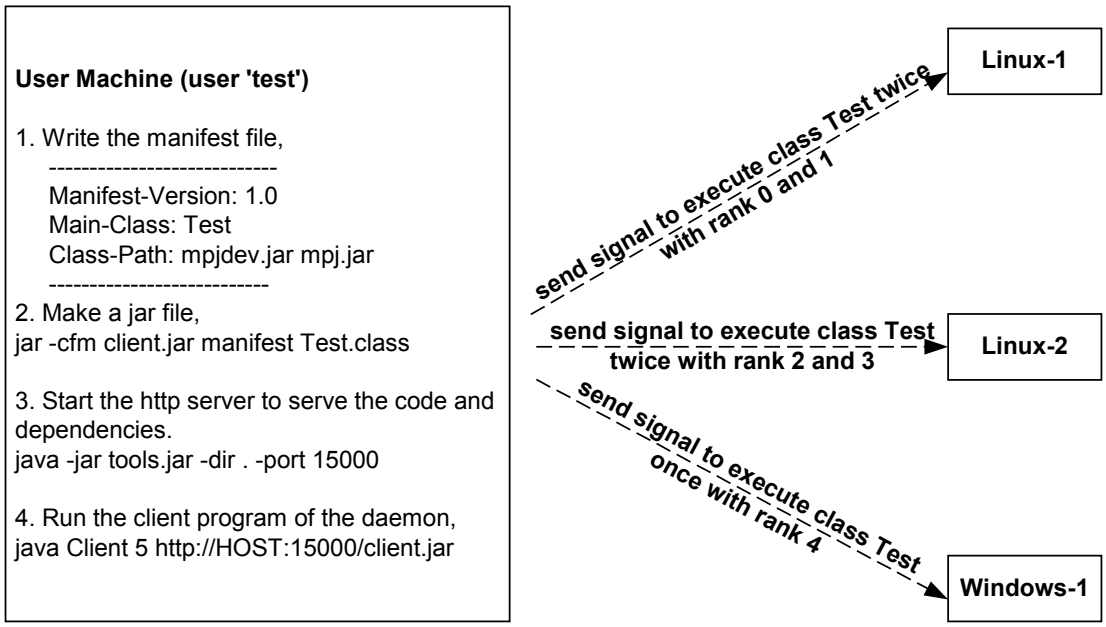

Figure 37: MPJ Job Submission to the Remote Nodes

5.3.1 Dynamic Class Loading

For step 3, described above, we need a dynamic class loading mechanism that copies the binaries along with all the dependencies from the user system that runs the initiator module to the daemon (execution nodes). We have implemented this dynamic class loading as follows,

1. The user should bundle up the program into a Java archive format called a jar. The benefit of using jar files is that the user can specify the main class, along with the dependencies. In the Figure 37, the main class is Test and the dependencies for Test class are the libraries imported from `mpjdev.jar` and `mpj.jar`. We put the names of these two dependencies in the `CLASSPATH` environmental variable.

2. The next step is to start an HTTP server that points to the directory containing the `client.jar` file. Whenever some JVM requests this jar file, `mpj.jar` and `mpjdev.jar` are automatically loaded into the requesting JVM.

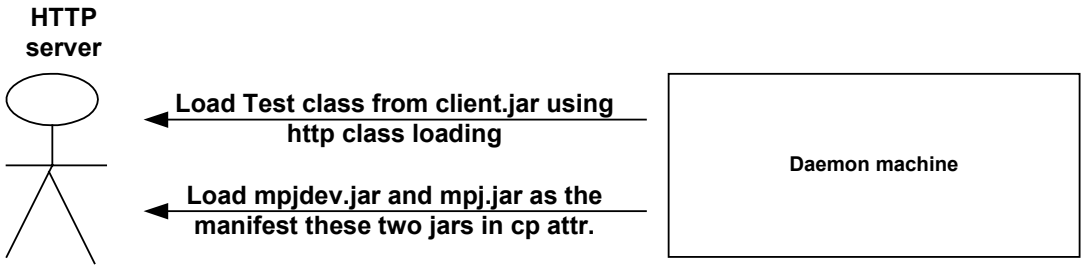

Figure 38: The Dynamic Class Loading

3. The next step is to actually start the client that sends the request to the compute nodes to execute a jar file present on a particular URL. The compute nodes use the URL to actually download the jar file, and execute the main class, which in Figure 38 is the Test class.

5.3.2 Security Issues

We have decided to implement a SSH like authentication mechanism for the MPJ runtime.

1. The administrator can install the daemons on Linux or Windows, provided they have a system account.
2. Once the daemon is running on any node, only the administrator can add the public keys of the users to the daemons. It is assumed that each of the

users of the system will have a keystore that contains their public and the private keys.

3. The initiator module sends a signal to the daemon (execution node) along with the user name. The daemon encrypts a random number with the public key of the user (added in the previous step) by the administrator, and then sends the encrypted number to the client. The initiator module decrypts it, and sends the random number back to the daemon machine. The daemon machine matches the random number with the number sent by the user. If the two match each other, then the user is authenticated. This is explained in the Figure 39.

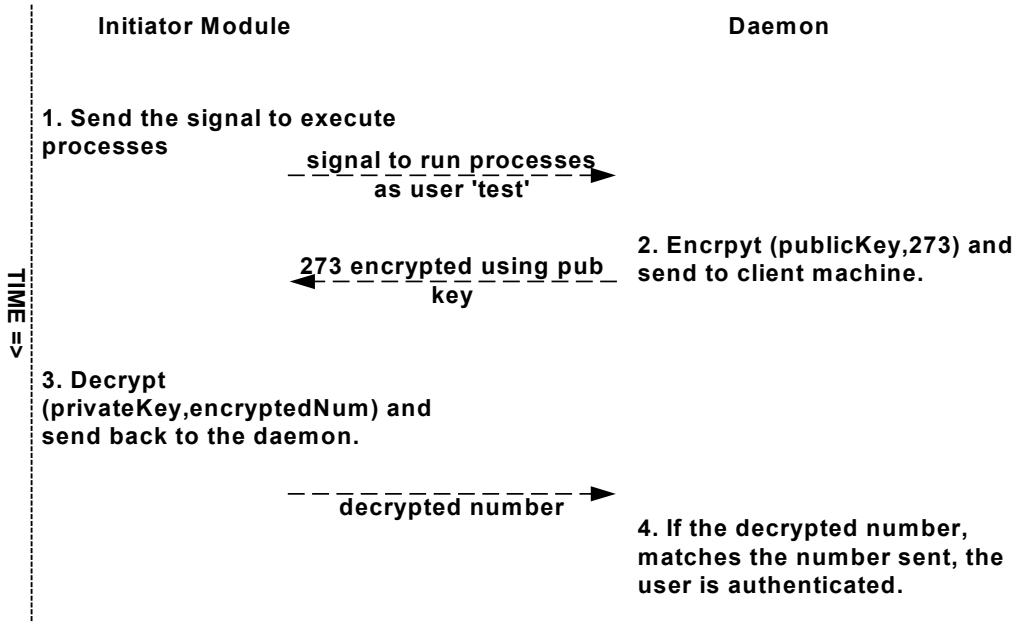

Figure 39: SSH-based Authentication

5.4 Summary

In this chapter, we have discussed the implementation of the mpjdev device driver and the associated MPJ runtime infrastructure. The buffering API implemented as part of the mpjdev allows application developers to send

objects along with the other Java primitive datatypes. The device driver implements two communication protocols. These are important to support the different modes of sending specified by the MPI standards. These modes include standard blocking and non-blocking, synchronous, and ready mode. The implementation of the MPJ higher-level communications and other features like communicators, virtual topologies, supporting derived datatypes are currently in progress.

6 Performance Evaluation

6.1 Introduction

In this chapter, we evaluate and compare the performance of the mpjdev device driver with the native mpjdev device driver.

6.2 Test Environment

The performance tests were conducted on the DSG Cluster; known as "StarBug", that consists of a head node "Holly" and eight compute nodes. The configuration of each of the nodes is described in Table 6 and Table 7.

Processor Type	Dual Xeon (Prestonia)
Processor Speed	2.8 Ghz
Processor Cache	512K L2 Cache
Front Side Bus	533 MHz
RAM	2 GB ECC
Storage	80 GB ECC
Java Version	J2SE 5.0 Beta 2
Operating System	Debian Linux (Sarge)

Table 6: The DSG Cluster Configuration

mpjdev	Version 0.1 alpha
native mpjdev	mpjdev 2.0: Internal Alpha Release
MPICH	MPICH-1.2.5

Table 7: The Software Versions

6.3 The Evaluation of mpjdev

In this sub-section, we evaluate point-to-point communications between two remote nodes using the Ping-Pong benchmark. The Ping-Pong benchmark sends a message of variable length, n, is sent from one node to another. The time for the message to travel to the remote node is half the total time recorded. The benchmark performs the test at each data point a thousand times, preceded by a warm-up loop of hundred iterations.

6.3.1 The Point-to-Point Comparison on Remote Linux Nodes

This section presents the point-to-point comparison of the two devices. Theoretically, both the devices should perform almost the same for small message with mpjdev outperforming the native mpjdev for the larger messages because of the overhead of copying the larger sized arrays between the Java and the native MPI. A detailed discussion of this overhead can be found in [80].

6.3.1.1 Transfer Time Comparison

Figure 40 shows plots of the transfer time comparison between the mpjdev and the native mpjdev. The latency that is defined as the time to transfer one byte message is 275 microseconds and 295 microseconds for mpjdev and native mpjdev respectively.

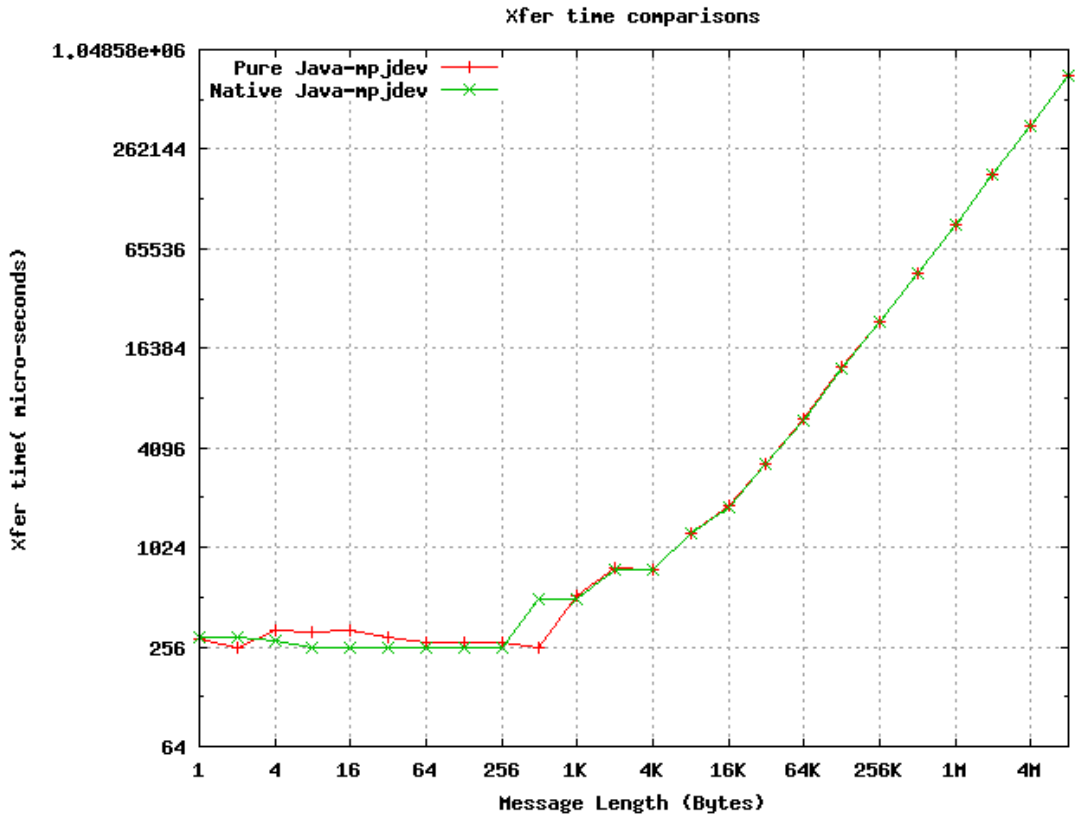

Figure 40: A Transfer time Comparison of mpjdev and the Native mpjdev

The graph shown in Figure 40 indicates that the performance of mpjdev and the native mpjdev is almost the same. mpjdev takes a fraction of a more time than the native mpjdev for the message size between 4 to 64 bytes. The reason for this overhead is the additional copy that may be incurred by the eager-send protocol. An increase in the transfer time for both the devices is noticeable as the message size reaches the length of 16 Kbytes. This is because the default socket buffer size of the Linux kernel 2.4.26 is 16 Kbytes. This plays a significant role in the overall transmission time because the messages are divided into multiple chunks for messages larger than 16 Kbytes. Overall with a slight overhead of the mpjdev for small messages, the performance of the two devices is almost the same.

6.3.1.2 Bandwidth Comparison

The maximum bandwidth achieved on Fast Ethernet by mpjdev and native mpjdev is 88.11 Mbps and 87.94 Mbps respectively.

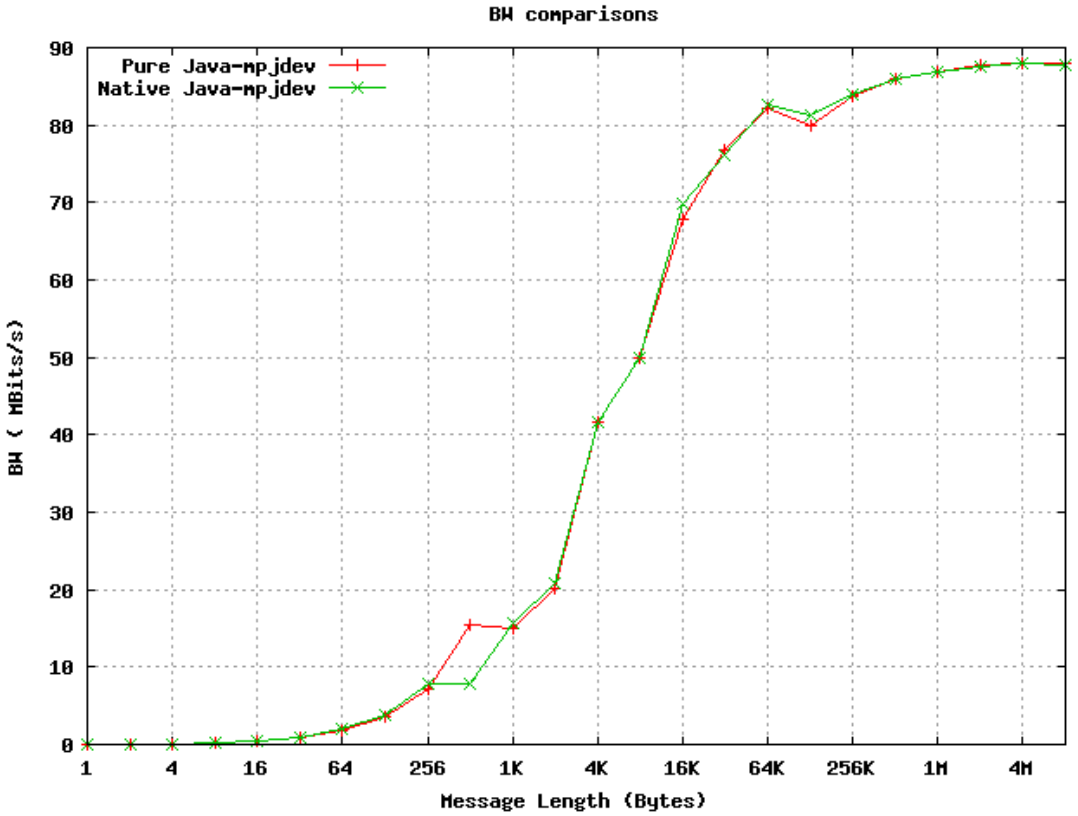

Figure 41: A Bandwidth Comparison of mpjdev and the Native mpjdev

Figure 41 indicates that there is no significant performance difference between the two devices. The performance degradation for both the devices at 128 Kbytes (message size) is due to the change of the communication protocol from the eager-send to the rendezvous. For larger message sizes, the overhead of the JNI is noticeable for the native mpjdev that allows mpjdev to achieve more throughput than the native mpjdev.

6.3.2 Point-to-Point Communications on a Single Linux Node

This section presents the local host comparison of communication of the two devices. The performance of the native mpjdev should be better than mpjdev because the native MPI can use shared memory communication. The two Java processes run as separate JVMs on the same machine and the only way to communicate is through sockets. In such communications the memory bus bandwidth becomes the bottleneck whereas the timings for native MPI is dominant by memory access time.

6.3.2.1 Transfer Time Comparison

Figure 42 shows the transfer time comparison on the localhost between the mpjdev and the native mpjdev. The latency is 169 microseconds and 25 microseconds for mpjdev and the native mpjdev respectively.

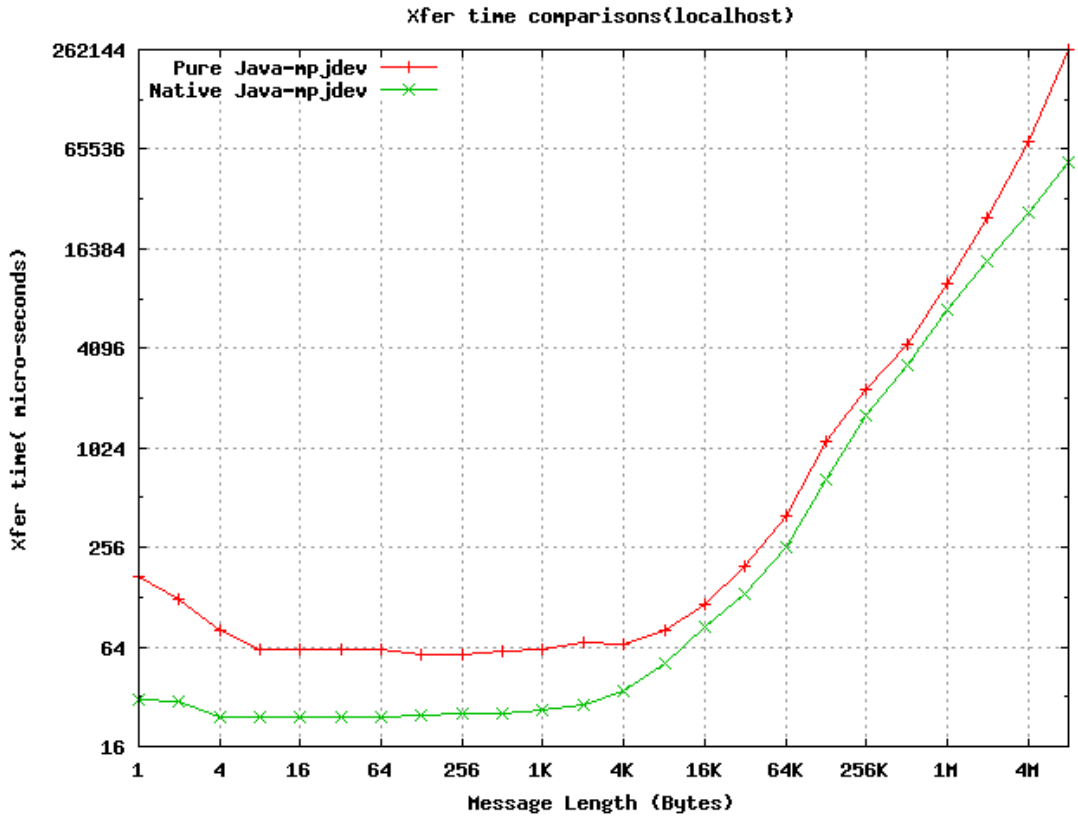

Figure 42: A Transfer Time Comparison of mpjdev and the Native mpjdev on Localhost

Figure 42 shows that the performance of native mpjdev is clearly better than the mpjdev. As mentioned, this is because the two mpjdev processes on the same machine cannot use the shared memory paradigm for communication.

6.3.2.2 Bandwidth Comparison

The maximum bandwidth achieved by mpjdev and native mpjdev is 1261.34 Mbps (for 32 Kbytes message size) and 1927.52 Mbps (for 64 Kbytes message size) respectively.

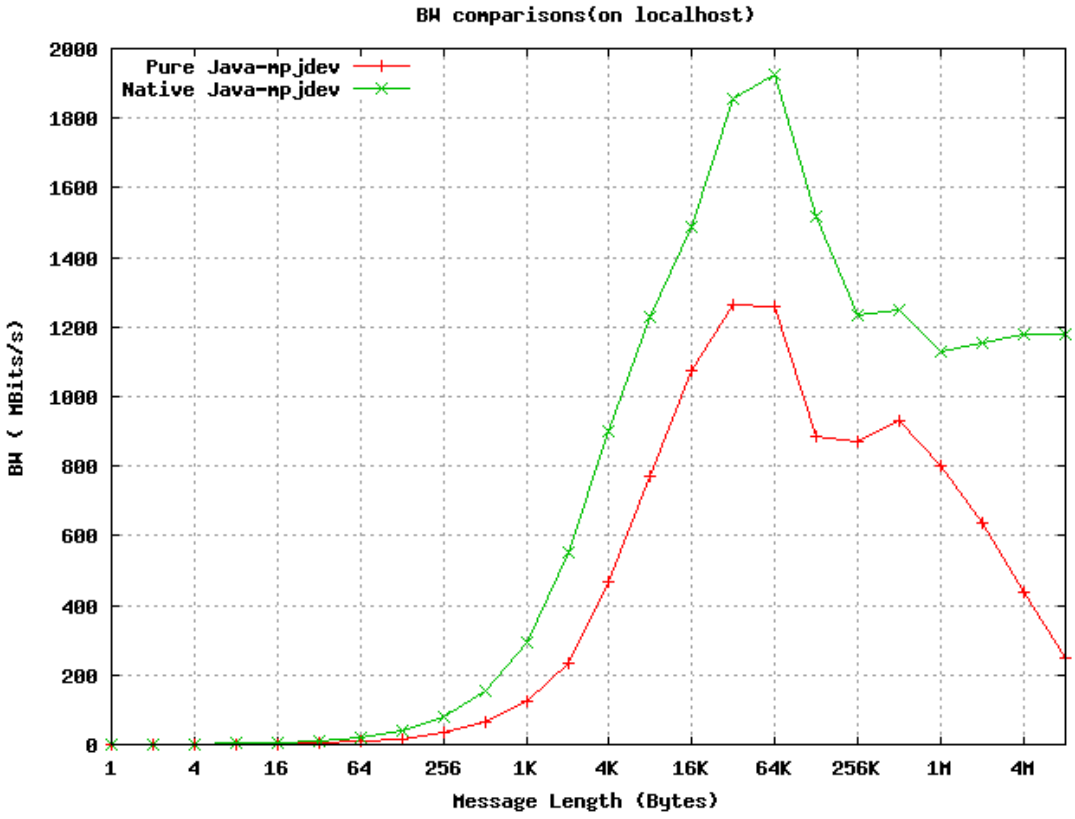

Figure 43: A Bandwidth Comparison of mpjdev and the Native mpjdev on Localhost

The native mpjdev outperforms the mpjdev by 25% at 64 Kbytes. The poor performance of mpjdev processes running on the same SMP cluster node is clearly a bottleneck especially for SMP clusters where shared memory communication is an effective way for the processors on same main board to communicate to each other.

6.3.3 Protocol Switch limit

MPICH uses 128 Kbyes (message size in bytes) as the threshold to switch from eager-send to rendezvous protocol. Though we note from Figure 45 that the performance of the eager-send protocol is better for messages sizes of 1 Mbytes, but the size of the mpjdev buffer where the messages are stored does not allow supporting eager-send messages of such length. For this reason, we have chosen 128 Kbytes as the protocol threshold, but the switch limit is not hard-coded in the application and can be changed by the users through the configuration file. If the user wishes to use the eager-send protocol, then they should increase the JVM heap size, as the default 64 Kbytes is not enough for this kind of large mpjdev buffer.

6.3.3.1 Transfer Time Graph

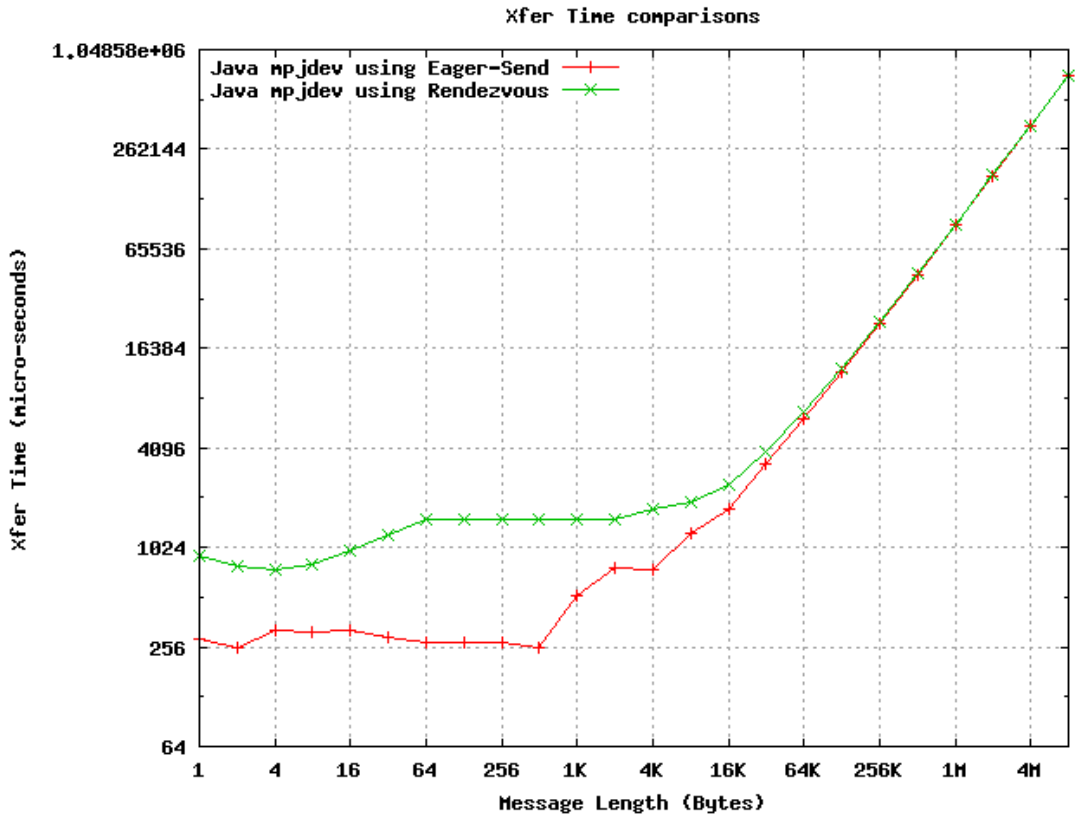

Figure 44: The Transfer Time of the Eager-send and Rendezvous Protocols

The latency is 282 microseconds for eager-send and 906 microseconds for the rendezvous protocol. There is an exchange of control messages before the actual data is transmitted using the rendezvous protocol that results in a high latency. For the same reason, the rendezvous protocol is useful for messages of larger size when the overhead of the control messages becomes negligible when compared to the overall communication cost.

6.3.3.2 Bandwidth Graph

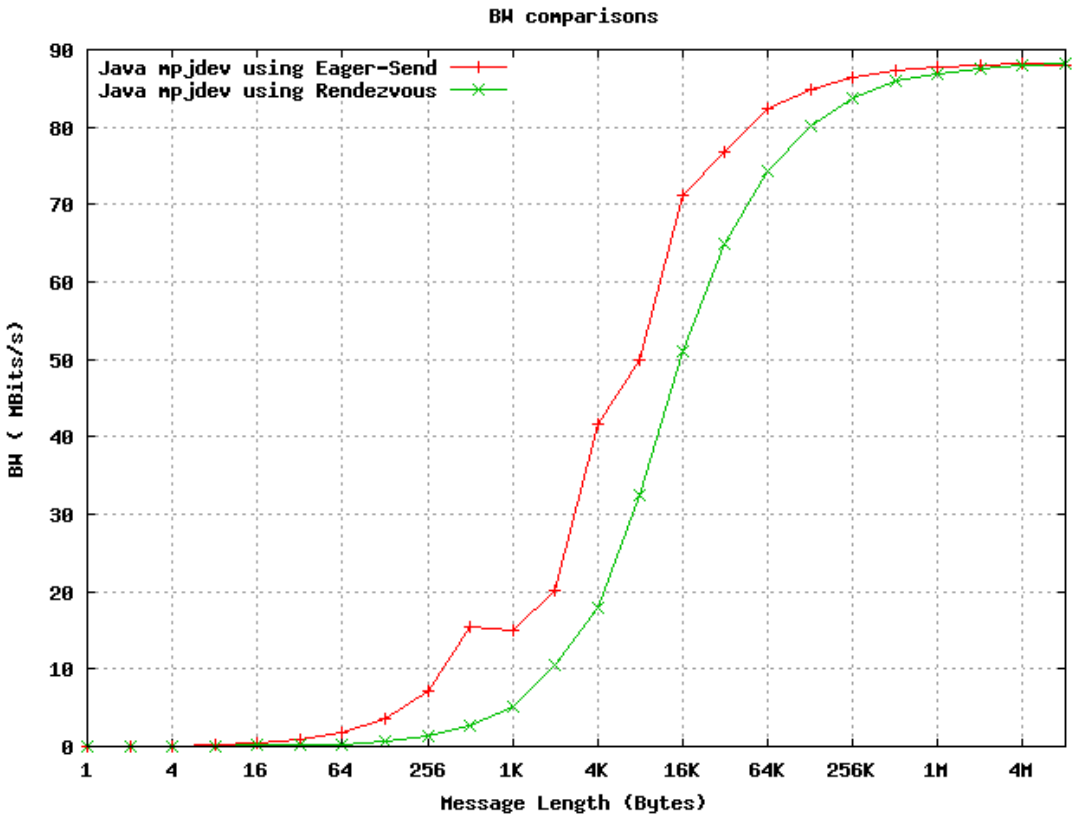

Figure 45: The Bandwidth of the Eager-send and Rendezvous Protocols

The bandwidth is 88.02 Mbps and 88.11 for eager-send and rendezvous respectively. The rendezvous protocol achieves greater bandwidth than the eager-send because it avoids the additional copying that the eager-send may incur that is more effective factor than the time spent on control messages in case of rendezvous.

6.4 The Evaluation of the MPJ Point-to-Point Layer

This sub-section presents a comparison between MPJ, mpiJava, and MPICH. Currently the point-to-point methods in MPJ are implemented on top of the mpjdev, and the implementation of the collective communications is a work in progress. Netpipe [17] is used to calculate the transfer time and the bandwidth achieved for MPICH.

6.4.1 Point-to-Point Comparison on remote nodes of Linux nodes

This section presents the comparison of point-to-point communication between MPJ, mpiJava, and MPICH.

6.4.1.1 Transfer time Comparison

Figure 46 shows a plot of the transfer times of MPJ, mpiJava, and MPICH. The latency, defined as the time to transfer one byte message, is 268 microseconds, 145 microseconds, and 247 microseconds for MPJ, MPICH, and mpiJava respectively.

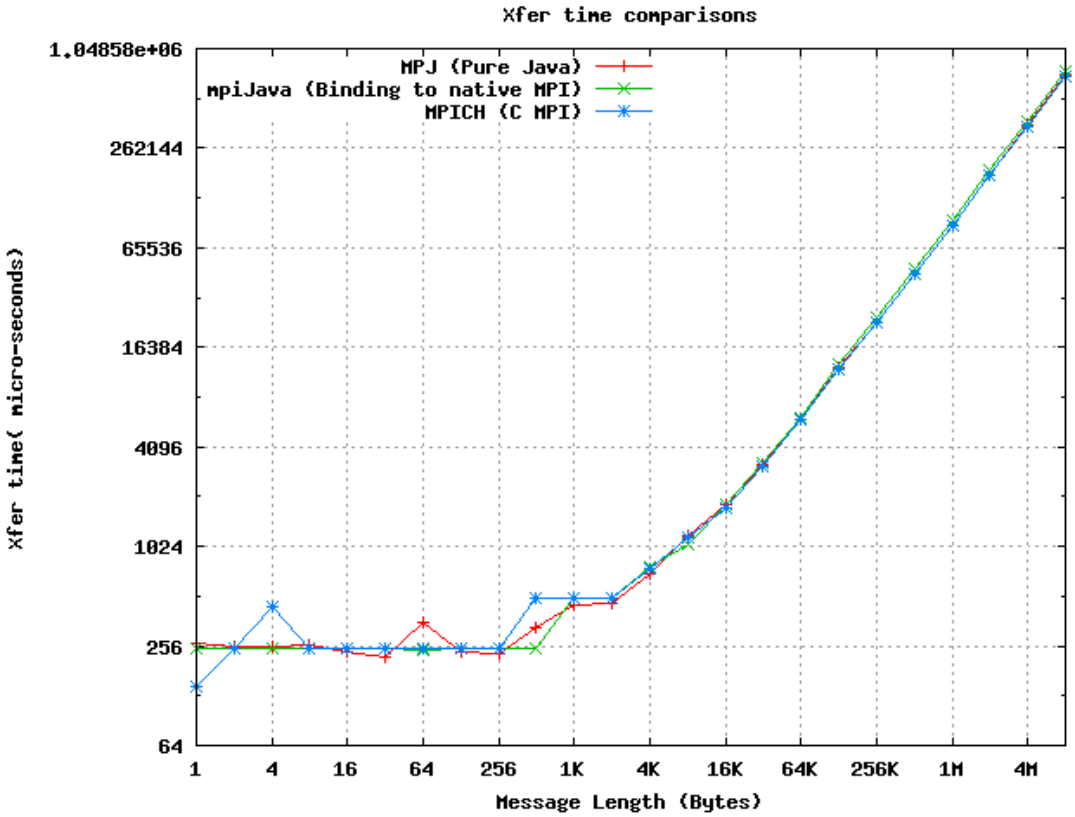

Figure 46: The Transfer Time for MPJ, mpiJava, and MPICH

Figure 46 shows the transfer times for the three libraries. The transfer time graph is significant for small messages and is useful for calculating latency. Whereas, the bandwidth graph are significant for large messages and are useful for evaluating the maximum bandwidth achieved.

6.4.1.2 Bandwidth Comparison

The maximum bandwidth achieved by MPJ, MPICH, and mpiJava is 89.26 Mbps, 89.57 Mbps, and 83.92 Mbps respectively.

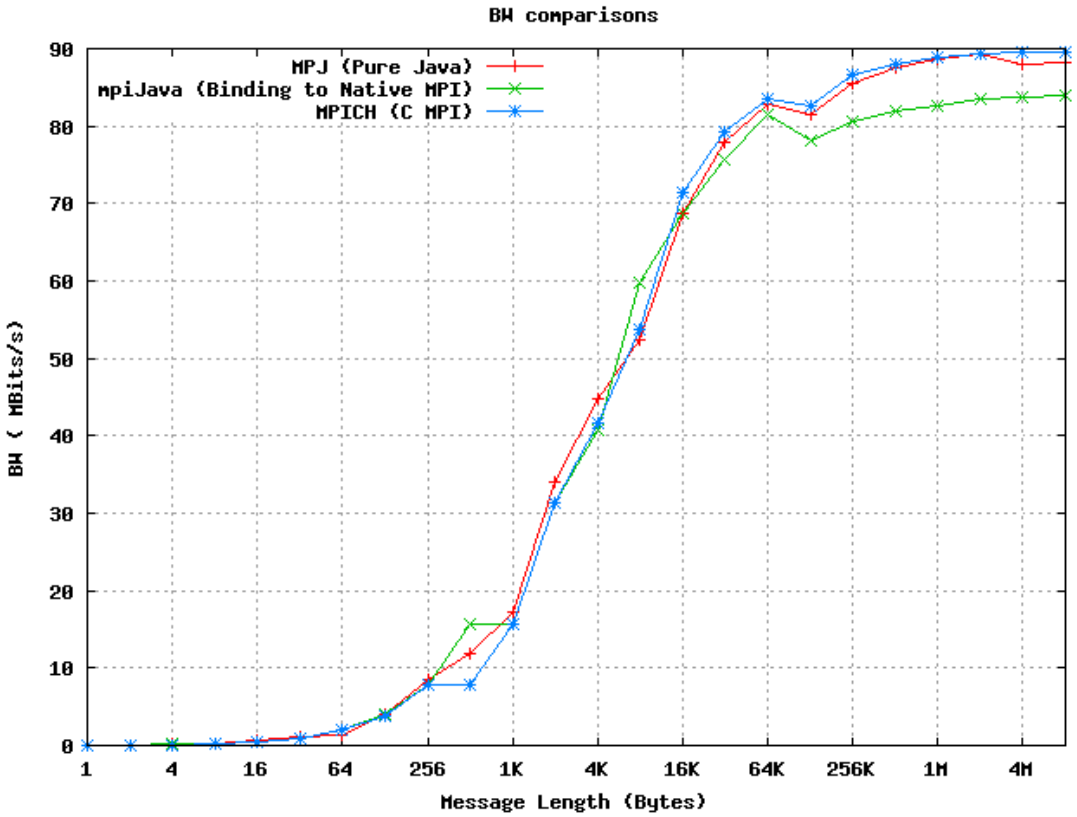

Figure 47: The Bandwidth of MPJ, mpiJava and MPICH

Figure 47 indicates that MPICH achieves the highest bandwidth, followed by MPJ. This indicates that the MPJ point-to-point communication layer incurs a small overhead. MPJ and MPICH outperform mpiJava for message sizes

larger than 128 Kbytes, this is due to which is because of the overhead of the JNI.

6.5 Summary

In this chapter, we have evaluated the performance of the mpjdev by comparing it with the native mpjdev that uses MPICH as the underlying native MPI library. MPJ's performance was found to be fairly close to the native mpjdev. This comparison was followed by a comparison between MPJ, MPICH, and mpiJava for point-to-point communications. It is clear that MPJ outperforms mpiJava and attains performance that is comparable to MPICH. Thus, our hypothesis presented in the design chapter that MPJ should be an efficient and high-performance library holds true. We intend to provide a detailed comparison of MPJ implemented on top of the mpjdev with other MPI libraries in the near future.

7 Conclusion

In this chapter, we conclude this report by discussing some of the lessons learnt during the implementation of MPJ and outline the future research.

7.1 General summary

This report presented the design and the implementation of MPJ. One of the obvious advantages of such a message passing system is a portable approach to problem solving using heterogeneous operating systems and hardware without compromising the overall communication performance. MPJ follows a layered structure that allows enhancements to the existing infrastructure. This also allows the higher communication layers to swap in various device drivers to make use of specialized hardware or protocols. The mpjdev device driver implemented as part of this project has an efficient buffering API that is used to pack/unpack the data to/from the buffer. mpjdev implements three communication protocols, inter-process, eager-send and the rendezvous protocol. mpjdev provides a simple interface that provides the basic functionality for starting up the device, setting up the communication infrastructure and sending/receiving the data to/from the other peers. We have found that the performance of the mpjdev is reasonable in comparison to the native mpjdev. The point-to-point communication layer has been implemented on top of mpjdev, and the performance comparisons to mpiJava and MPICH in Chapter 6 show that it is possible to get high-performance without compromising the portability of the Java language.

The experiences gained by the implementation of this message-passing library suggest that there are still some areas where detailed research is still needed.

Out of these areas, the lack of support for shared communication paradigm between the JVMs running on the same host and efficient multi-dimensional arrays operations are the two important areas. The JVM hides all the details of garbage collection, though the garbage collection algorithms can be changed through the use of command line switches to the JVM. It is desirable to have more information about the garbage collection thread as we suspect that it is the cause of performance overhead in some situations. It has been noticed that while developing some memory intensive application, the JVM runs out of memory that can only be handled by increasing the JVM heap size. The implications of increasing the JVM heap size are not clear, and an interfaces/tools that may expose more internal JVM information is vital for the adoption of message passing libraries in Java like MPJ.

7.2 Future Work

Currently, the implementation of mpjdev and point-to-point communications layers is complete. In the near future, the implementation of the runtime and the MPJ collective operations will be completed. Once finished, we intend to benchmark MPJ and compare it to mpiJava and other C/Fortran message passing libraries.

7.2.1 Implementing Four Modes of Point-to-Point Communications

MPJ currently only supports the standard mode of send operation. The MPI standard defines three additional modes of sending messages, which we intend to implement in the near future. The three additional modes of point-to-point communications include buffered, synchronous, and ready mode. More details of these modes can be found in [4].

The mpjdev implements three communication protocols; inter-process communication protocol, eager-send, and rendezvous protocol. The last two protocols along, with the buffering mechanism, at both the sender and the receiver side can support the four modes of sending the messages.

7.2.1.1 Implementing the Collective Communications Layers

The MPI standard defines a set of collective communications operation in order to facilitate parallel programming. Group communications makes the task easy for application developers and makes their code more understandable. Group communications include operations like barrier synchronization, broadcast, gather/scatter, and reduce operations. Currently these operations have not been implemented in MPJ, but we intend to implement these on top of point-to-point communication layer. In addition to collective communications, MPJ will support process topologies and communicators in order to be a standards-based MPI implementation.

7.2.1.2 Support for Multi-dimensional Arrays

The support for multi-dimensional arrays in Java is necessary for the adoption of any message-passing library by the technical computing community. The reason is that multi-dimensional arrays form the basic building blocks of nearly all of the scientific problems. Java's multi-dimensional arrays are fundamentally different from other traditional languages like C and Fortran. Java deals with multi-dimensional arrays as arrays of arrays that essentially means that a multi-dimensional array need not to be rectangular, which is confusing for application developers as they are accustomed to rectangular arrays like in C or Fortran. We intend to incorporate the support of multi-

dimensional arrays by using the buffering API implemented by the mpjdev device driver. Some of related work done regarding multi-dimensional arrays can be seen in [81][82].

7.2.1.3 Enhancements to the MPJ API

During the early part of 1998, several prototypes Java bindings for MPI-like libraries emerged. As a result, the Message-Passing Working Group of the Java Grande Forum [83] was formed that came up with an initial draft of an API, which was distributed at SC 98. Since then, a draft API called MPJ has been specified, but there is no complete implementation of this specification at the moment. The MPJ API specifications can be seen in [7][4].

The MPJ API is procedural and is not fully making use of the object-oriented features that Java offers. We intend to come up with a draft API that is more Java centric.

7.2.1.4 Shared Memory Communications

The Java language does not specify a way for two processes that execute in separate JVMs on the same machine to communicate. This results in costly communications between processes running on the same machine. There are two ways to tackle this problem. The first approach is to develop a separate device driver that uses JNI to invoke the native C code that handles such shared memory communications. The second and the preferred approach, is to run all the processes within the same JVM as different threads. The second approach is preferable because this does not compromise the portability of the Java language.

7.2.2 The Runtime Infrastructure

The portable nature of the Java language is one of the biggest advantages for implementing a messaging system. Some of the previously developed message passing system use shell scripts to start the processes over the remote nodes of the Linux cluster. Others use the native MPI runtime to start the execution of the parallel job. These approaches are not reasonable for a Java message passing because they compromise portability. Thus it is necessary to develop a runtime in Java that allows spawning of processes over heterogeneous operating systems. The runtime we are currently implementing will be extended to build a whole message-passing infrastructure that allows easy installation, execution, debugging, and performance profiling of the parallel applications. In addition the runtime should handle various software and hardware failures that the application may encounter during the course of their execution. We explain the runtime in more detail in this section.

7.2.2.1 Runtime Modules

In this section, we describe each of the modules of the runtime infrastructure in more detail. Figure 48 shows DSG cluster 'Starbug' that consists of the head node, connected via a private network to the compute nodes. It also shows various modules of the MPJ runtime and their interaction. The user uses a laptop, shown in Figure 48, to install and execute the parallel application.

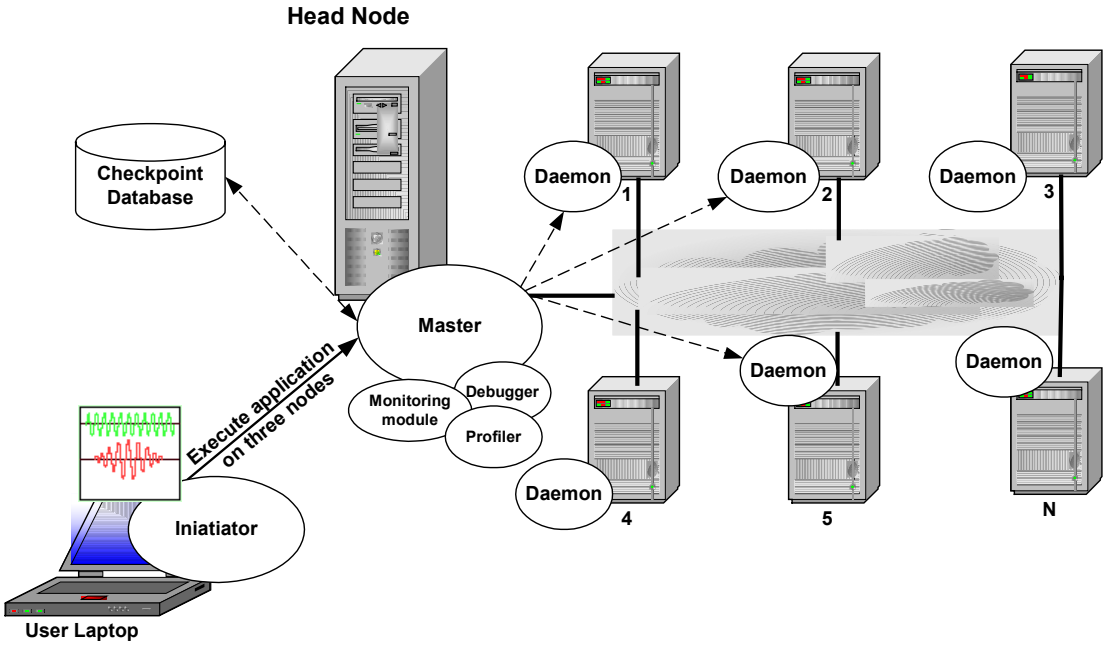

Figure 48: The MPJ Runtime Infrastructure

7.2.2.1.1 The MPJ Daemon

The daemon module runs on all the machines that participate in the execution of a MPJ parallel application. These machines may be the computational nodes of the cluster (Figure 48) or workstations present on the LAN.

7.2.2.1.2 The MPJ Master

The master module runs on the front-end of the cluster (Figure 48) or it may run on any of the machines present on LAN including the host running the initiator module. The rationale behind having this module is that it is not possible to access all the daemons running on the compute nodes from the user machine because the computational nodes of the cluster are normally only accessible from the head node of the cluster. In case, an organization does not have a cluster, then the master module can be started on any one of the machines over the LAN machine, even the user machine.

7.2.2.1.3 Initiator Module

The initiator module runs on the user machine and allows the user of this infrastructure to execute, debug, and profile the parallel application. The user has the option to start the graphical client, or run the initiator module on the command line. Once the user has written the parallel application, the initiator module allows the user to submit the application to the master module, and get the output from the master. The initiator module also allows the check the status of the executing application through the monitoring and profiling interface. Moreover, it also allows debugging of the application. Java 2D [84], Java 3D [85] and Java Analysis Studio (JAS) [86] are some of the useful Java packages to present the information regarding the execution of the process in a user-friendly and understandable way. We intend to re-use the code from some of the relevant efforts [87][37] in order to save development time.

7.2.2.2 Installation of the Runtime

One of the important issues that we intend overcome is that of simplifying the installation of the infrastructure. The task of installation may become tedious if the users have to do it by hand, for example installing and configuring MPJ on hundreds of nodes. One other issue is that Windows OS does not provide a mechanism to remotely and securely install the modules like the Linux OS. We are currently investigating ways [89] to deal with this issue.

The user must provide some configuration information like the name the names of any machine that the user would like to run the daemon on. In case of a cluster, the user only specifies the name of the head node. The master module is remotely installed on the head node that finds the compute nodes

and installs the daemon on all the compute nodes. In case of workstations, the daemons are remotely installed from the user machine.

The installation of the initiator module can be done using the Java Web Start [88] or it can be started on the user machine as a normal Java application.

7.2.2.2.1 Forming a Tree Topology

Once the daemons and the master module have been installed on the hosts, the master modules and the daemons arrange themselves in a tree topology with the master as the root of the tree. The master module is responsible to manage the topology. To avoid performance bottlenecks, multiple instances of the master modules may be started that form two trees.

7.2.3 Operations of the Runtime

In this sub-section, we outline some of the functionality provided by the runtime infrastructure to the application developers.

7.2.3.1 Execution of the Application

The user can interact with the initiator module executing on their node to execute the parallel application. This module can be started in a visual or command line mode. The GUI allows a way for the application developer to submit the application to the master, specifying the number of the processes. The master will subsequently run the application over the daemons, either the daemons over the cluster, or the daemons on the workstations present in the LAN. This is shown in Figure 48, where the user has requested the execution of an application on three nodes.

7.2.3.2 Debugging the application

The master receives the information back from the daemon about the execution of an application. It return the information back to the initiator module running on the user host to assist in the debugging of the application, that is critical for diagnosing potential bugs. All the information regarding the partial execution and the point of failure is communicated back to the user, which can be analysed to later debug the application.

7.2.3.3 Profiling and Monitoring the Application

The runtime module contains two sub-modules that keep track of the profiling and monitoring information of the application being executed. All of this information can be stored either in a database at the user machine for historical purposes, or can be viewed in real-time. Tools like JVMTI [58] dump the execution data regarding the JVM execution, which is hard to analyse by scientists and can be presented in useful manner to help them find potential problems.

7.2.3.4 Runtime Fault-Tolerance

Before we discuss the details of the fault-tolerant behaviour exhibited by the runtime infrastructure, it is important to specify what module possess the fault-tolerant behaviour, the application or the runtime infrastructure and what faults we will address. The fault tolerance discussed in this sub-section is exhibited by the runtime infrastructure. Secondly, the faults may occur because a MPJ daemon may die because of a bug in the code or a hardware

failure. Given that a fault occurs and has been detected by the master, which then takes preventive measures and marks this particular daemon as faulty. This information is sent back to the user who may manually fix the problem. The issue of correcting the fault is out of scope of this infrastructure.

7.2.4 Application Fault-Tolerance

HPC is currently moving towards clusters having thousands of the processors. One of the current advances is the IBM Blue Gene that will contain 65,536 processors when fully functional. In such a cluster, hardware and software failures will be frequent. Thus, it is essential for future parallel application infrastructure to exhibit fault tolerance. We intend to provide support for process migration to tackle this problem. Java is unique in this respect from conventional languages because it is possible to track the program counter, save the state of the process and migrate it to other machine and resume execution, which is difficult in traditional languages. Such process migration has been achieved in [55]. In addition, this allows avoidance of overheads introduced by writing the checkpoints even in the failure free execution. Al Geist et al [90] note that checkpoint and restart may not be a practical option for these large clusters as its cost is more than the time between the failures. For all the failures that can be detected at the software level, we believe that the application programmers can avoid the overhead of checkpoint and restart the applications by using process migration at the application level.

7.2.5 Conclusions

In this report, we have discussed our early experiences with an implementation of a Java messaging system called MPJ. MPJ uses the device

drivers in similar fashion to the communication devices used in MPICH. In this report we discussed the implementation of mpjdev, the device driver based on the Java NIO package. We have evaluated the performance of mpjdev and found it comparable to the native mpjdev, a device driver that uses the underlying native MPI library. We have implemented point-to-point communications based on the mpjdev and evaluated it against mpiJava and MPICH. We have found the performance to be better than mpiJava and close to MPICH. This shows that a Java messaging system built on NIO can perform as well as the MPI implementations developed in the traditional languages and better than the Java messaging systems that use JNI. We are currently implementing collective communications on top of point-to-point layer. To support the execution of parallel processes on remote hosts, we have implemented a runtime infrastructure. We intend extend the runtime infrastructure to allow application developers debug, profile and monitor their parallel applications.

References

[1] R. Buyya, editor, "High Performance Cluster Computing, volume 1: Architectures and Systems", Prentice-Hall, ISBN 0-13-13784-7, 1999.

[2] R. Buyya, editor, "High Performance Cluster Computing, volume 2: Programming and Applications", Prentice-Hall, ISBN 0-13- 13785-5, 1999.

[3] G. Couloris, J. Dollimore and T. Kinberg, Distributed Systems – Concepts and Design, Third Edition, Addison-Wesley, Pearson Education 2001, ISBN: 0201-619-180.

[4] The Message Passing Interface (MPI) standard, http://www-unix.mcs.anl.gov/mpi/

[5] MPICH, A Portable Implementation of MPI, http://www-unix.mcs.anl.gov/mpi/mpich/

[6] LAM/MPI, Parallel Computing, http://www.lam-mpi.org/

[7] B. Carpenter, V. Getov, G. Judd, T. Skjellum and G. Fox, MPI for Java - Position Document and Draft API Specification, Java Grande Forum Technical Report JGF-TR-03, November 1998.

[8] mpiJava, http://www.hpjava.org/mpiJava.html

[9] The Java Native Interface (JNI) Specifications, http://java.sun.com/j2se/1.3/docs/guide/jni/

[10] The Remote Method Invocation (RMI) Specifications, http://java.sun.com/j2se/1.4.2/docs/guide/rmi/

[11] The Java New I/O Specifications, http://java.sun.com/j2se/1.4.2/docs/guide/nio/

[12] R. Boisvert, J. Dongarra, R. Pozo, K. Remington, and G. Stewart, Developing numerical libraries in Java, ACM, Stanford University, Palo Alto, California, and March 1998.

[13] C. Austin, and M. Pawlan, Advanced Programming for the Java 2 Platform, Chapter 8 Performance Features and Tools, Addison Wesley Longman, September, 2004.

[14] The JAGUAR project, http://www.eecs.harvard.edu/~mdw/proj/old/jaguar/

[15] The java.nio.Buffer documentation, http://java.sun.com/j2se/1.4.2/docs/api/java/nio/Buffer.html

[16] R. Hitchens, Java NIO, O'Reilly & Associates, August 2003.

[17] D. Turner, A. Oline, X. Chen, and T. Benjegerdes, Integrating New Capabilities into NetPIPE, Euro PVM/MPI, Venice, Italy, September 2003.

[18] Ames Laboratory, http://www.external.ameslab.gov/

[19] M. Baker, H. Ong, A. Shafi, A study of Java networking on Linux Cluster, DSG Technical Report, November 2003.

[20] J. Maassen, R. van Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, R. Hofman, Efficient Java RMI for Parallel Programming, ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 23, Issue 6 (November 2001), pages 747 – 775, March 2000.

[21] J. Maassen, R. V. Nieuwpoort, R. Veldema, H. E. Bal, and A. Plaat, An Efficient Implementation of Java's Remote Method Invocation, Proc. Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99), pp. 173-182, Atlanta, GA, May 4-6, 1999.

[22] Manta Fast Parallel Java, http://www.cs.vu.nl/~robn/manta/

[23] M. Welsh, A System Supporting High-Performance Communication and I/O in Java, Master's Thesis, University of California, Berkeley, October 1999.

[24] V. Getov, S. Hummel and S. Mintchev, High-performance Parallel programming in Java: exploiting native libraries, Concurrency: Practice and Experience, Volume 10 (11/13), pages 863--872, 1998.

[25] S. Mintchev and V. Getov, Towards Portable Message Passing in Java: Binding MPI, Springer-Verlag, Proceedings of the 4th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, pages 135—142, 1997, ISBN 3-540-63697-8.

[26] University of Westminster, http://www.wmin.ac.uk

[27] G. Judd, M. Clement, and Q. Snell, DOGMA: Distributed object group management architecture, ACM, Concurrency: Practice and Experience, Volume 10 (11/13), pages 977-983, Workshop on Java for High-Performance Network Computing, Palo Alto, February 1998.

[28] G. Judd, M. Clement, Q. Snell, and V. Getov, Design issues for efficient implementation of mpi in java, In Proceedings of ACM 1999 Java Grande Conference, pages 58-65. ACM Press, 1999.

[29] Brigham Young University, http://home.byu.edu

[30] S. Morin, I. Koren, and C. Krishna, JMPI: Implementing the Message Passing Standard in Java, International Parallel and Distributed Processing Symposium: IPDPS 2002 Workshops, April 15 - 19, 2002, Fort Lauderdale, Florida

[31] University of Massachusetts, www.umass.edu

[32] KaRMI, http://wwwipd.ira.uka.de/JavaParty/KaRMI/

[33] JMPI download site, http://euler.ecs.umass.edu/jmpi/

[34] K. Dincer, Ubiquitous Message Passing Interface Implementation in Java: jmpi, IEEE Computer Society, Proceedings of the 13th International Symposium on Parallel Processing and the 10th

Symposium on Parallel and Distributed Processing, Pages 203, 1999, ISBN 0-7695-0143-5.
[35] Baskent University, http://www.baskent.edu.tr
[36] The Java Parallel Virtual Machine (JPVM), http://www.cs.virginia.edu/~ajf2j/jpvm.html
[37] 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, Southampton, UK, September 2-3, 1998.
[38] S. Ko, Object Based Message Passing in High Performance Computing using Java, Syracuse University, 2000, http://www.hpjava.org/theses/shko/thesis_paper/thesis_paper.html
[39] K. Hawick and H. James. A java based parallel programming support environment. In Proceedings of the 8th International Conference on High Performance Computing and Networks (HPCN), pages 363.372, Amsterdam, May 2000. Springer LNCS.
[40] Distributed and High-Performance Computing Group, http://www.dhpc.adelaide.edu.au
[41] University of Adelaide, www.adelaide.edu.au
[42] R. Buyya and V. Nagamani, JMPF: A Message Passing Framework for Cluster Computing in Java, Proceedings of the Fifth International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98), Las Vegas, Nevada, USA, CSREA Press, 1998.
[43] Queensland University of Technology, http://www.qut.edu.au
[44] Center for development of Advanced Computing, http://www.cdacindia.com/
[45] JMPF download site, http://www.buyya.com/papers/jmpf.html
[46] Y. Chen and W. Yang (2000), Java Message Passing Package-A design and implementation of MPI in Java, Proceedings of the Sixth Workshop on Compiler Techniques for High-Performance Computing, (Kaohsiung, Taiwan, March 16-17), 2000. (NSC 88-2213-E-009-006).
[47] National Chaio-Tung Univesity, www.nctu.edu.tw
[48] W. Tong, U. Ye, and W. Yao, PJMPI: Pure Java Implementation of MPI, The Fourth International Conference on High-Performance Computing in the Asia-Pacific Region-Volume 1, pp 533, May 14-17, 2000, Beijing, China
[49] Shanghai University, www.shu.edu.cn
[50] W. Pugh, and J. Spacoo, MPJava: High-Performance Message Passing in Java using Java.nio, Workshops on Languages and Compilers for Parallel Computing, Dept. of Computer Science, University of Texas at Austin, October 2003.
[51] The University of Maryland, http://www.umd.edu

[52] LAM-MPI issues with kernel 2.2.x, http://www.lam-mpi.org/linux/
[53] A. Nelise, J. Maasesen, T. Kielmann, and H. E. Bal, Object-based collective communication in Java, Java Grande Conference, Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande, Pages: 11 – 20, 2001, Palo Alto, California, United States, ISBN: 1-58113-359-6
[54] Vjire Universitete Amsterdam, http://www.vu.nl
[55] R. Ma, C. Wang, F. Lau, M-JavaMPI: A Java-MPI Binding with Process Migration Support, 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'02), p. 255, May 21 - 24, 2002, Berlin, Germany.
[56] The University of Hong Kong, http://www.hku.hk
[57] Java Virtual Machine Debug Interface, http://java.sun.com/j2se/1.4.2/docs/guide/jpda/jvmdi-spec.html
[58] Java Virtual Machine Tool Interface, http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/
[59] M-JavaMPI Project, http://www.cs.hku.hk/~clwang/projects/M-JavaMPI.html
[60] G-JavaMPI Project, http://www.cs.hku.hk/~clwang/projects/G-JavaMPI.htm
[61] M. Baker, B. Carpenter, et al, mpiJava: An Object-Oriented Java interface to MPI, the 1st Java Workshop at the 13th IPPS & 10th SPDP Conference, Puerto Rico, April 1999, LNCS, Springer Verlag, Heidelberg, Germany, ISBN 3-540-65831-9
[62] B. Carpenter, G. Fox, S. Ko, and S. Lim, Object Serialization for Marshalling Data in a Java Interface to MPI, Java Grande, Pages 66-71, 1999.
[63] mpiJava, http://www.hpjava.org/mpiJava.html
[64] Syracuse University, http://www.syr.edu
[65] University of Florida, http://www.ufl.edu/
[66] Indiana University, http://www.indiana.edu
[67] Distributed Systems Group, http://dsg.port.ac.uk
[68] K. Wieland, K. Kyle, R. Landau, Parallel Computing on the CPUG Beowulf with mpiJava, A technical report, Oregon State University, Department of Physics, July 19, 2002
[69] Many to Many Invocation for Parallel Development, Jennifer Ellis, Andrea Solecky and Dr Hans Peter Bischof, http://www.cs.rit.edu/~jle1028/Anhinga/
[70] mpiJava execution helper, http://homepages.cs.ncl.ac.uk/igor.mozolevsky/
[71] S. Shende and A. D. Malony, Integration and applications of the TAU performance system in parallel Java environments, ACM Press, Proceedings of the 2001 joint ACM-ISCOPE conference on Java

Grande, Pages 87-96, 2001, Palo Alto, California, United States, ISBN 1-58113-359-6.
[72] http://web.mit.edu/stardiff/www/6.338/progress_report.html
[73] J. Kochmar, P. Nowoczynski, J. Scott, and N. Stone, Comparing Java Agents and MPI for writing Parallel Algorithms, Technical Report, http://clan.psc.edu/docs/aglet.ps
[74] Mandelbrot, http://www.cs.rit.edu/~jle1028/Anhinga/mandelbrot.html
[75] A. Agarwal, J. Sharan, D. Gupta V. K. Garg, Fast forwarding distributed programs using computation slicing, Technical Report, http://home.iitk.ac.in/student/abhayag/reports/FastForwardingDistributed.pdf
[76] mpiJava Example Programmes, http://www.cs.cf.ac.uk/User/David.W.Walker/CM0323/code.html
[77] MPP, http://www.mi.uib.no/%7Ebjornoh/mtj/mpp/
[78] University of Bergen, http://www.uib.no
[79] B. Carpenter, mpjdev Buffer API documentation, mpjdev 2.0 Internal Release, July 2004.
[80] Matt Welsh and David Culler, Jaguar: Enabling Efficient Communication and I/O in Java, Concurrency: Practice and Experience, Special Issue on Java for High-Performance Applications Vol. 12, pp. 519-538, December 1999.
[81] Multidimensional Arrays and Multiarrays in HPJava, http://www.hpjava.org/papers/HPJava/HPJava/node12.html
[82] Numerically Intensive Java, http://www.alphaworks.ibm.com/tech/ninja/
[83] Java Grande, http://www.javagrande.org
[84] The Java2D package, http://java.sun.com/products/java-media/2D/
[85] The Java3D package, http://java.sun.com/products/java-media/3D/
[86] The Java Analysis Studio, http://jas.freehep.org/
[87] Sameer Shende and Allen D. Malony, Integration and applications of the TAU performance system in parallel Java environments, Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande, Pages: 87 – 96, 2001, ISBN: 1-58113-359-6, Palo Alto, California, United States
[88] Java Web Start Technology, http://java.sun.com/products/javawebstart/
[89] Sysinternals, Mark Russinovich and Bryce Cogswell, http://www.sysinternals.com/ntw2k/freeware/psexec.shtml
[90] A. Geist, and C. Engelmann, Development of Naturally Fault Tolerant Algorithms for Computing on 100,000 Processors, Journal of Parallel and Distributed Computing, 2002