Pluggable Architecture for Java HPC Messaging

Mark Baker[1], Bryan Carpenter[2], Aamir Shafi[1]
{aamir.shafi@port.ac.uk}

Abstract

Efforts to build Java messaging systems based on the MPI standard have typically followed either the JNI approach or the pure Java approach. On commodity platforms like Fast Ethernet, advances in JVM (Java Virtual Machine) technology now enable networking applications written in Java to rival their C counterparts. On the other hand, improvements in specialized networking hardware have continued, cutting down the communication costs to a couple of microseconds. Keeping both in mind, the key issue at present is not to debate the JNI approach versus the pure Java approach, but to provide a flexible mechanism for applications to swap between communication protocols.

Key Words:
Java, MPI, MPJ, mpiJava, Parallel Computing

1. Introduction

There were various efforts in the last decade to develop a Java messaging system based on the Message Passing Interface (MPI) [1] standard. These systems typically follow one of three approaches: use a Java Native Interface (JNI) to native MPI; implement Java messaging from scratch on top of Java RMI; or implement on top of lower level Java sockets. A drawback of the first approach [3] is that it does not comply with the 'write once run anywhere' philosophy of the Java language – also there is some performance overhead in JNI, especially for large messages, due to copying of the data from the JVM heap onto the system buffer [7]. On the other hand, JNI has the advantage that it enables Java code to access specialized communication hardware like Myrinet and Infiniband. The second approach uses Remote Method Invocation (RMI) [4]. The main aim of the RMI package is client-server interaction rather than message passing between peers. The use of low-level "pure" Java communications based on Java sockets is the third approach. Our initial benchmarks of the Java New I/O device in [8] demonstrate that it is possible to achieve performance close to C implementations of the MPI standard on Fast Ethernet. But it is not possible to access specialized hardware.

Experiences gained with these implementations suggest that there is no 'one size fits all' approach. The reason is that applications implemented on top of Java messaging systems can have different requirements. For some, the main concern could be portability, while for others high-bandwidth and low-latency could be the most important requirement. Portability and high-performance are often contradictory requirements. High performance can be achieved by making use of the specialized communication hardware, but only at the cost of comprimising portability that Java offers. The challenging issue is how to manage these contradictory requirements.

To address this issue, we are implementing *Message Passing in Java* (MPJ) [2] following a layered architecture based on an idea of *device drivers*. The idea is analogous to UNIX device drivers. We

[1] Distributed Systems Group, University of Portsmouth
[2] Open Middleware Infrastructure Institute, University of Southampton

have implemented a Java New I/O (NIO) [6] based device, a shared memory device, a GM [9] device, and a native MPI device. The ability to swap devices at runtime helps mitigate the contradictory requirements of the applications. In addition, we are implementing a runtime system that bootstraps MPJ processes over a collection of machines connected by a network. Though the runtime system is not part of the MPI specifications, it is essential to run MPJ processes across various platforms.

2. MPJ Design

MPJ has a layered design to allow incremental development, and provide the capability to update and swap layers in or out as needed. Figure 1 shows a layered view of the messaging system.

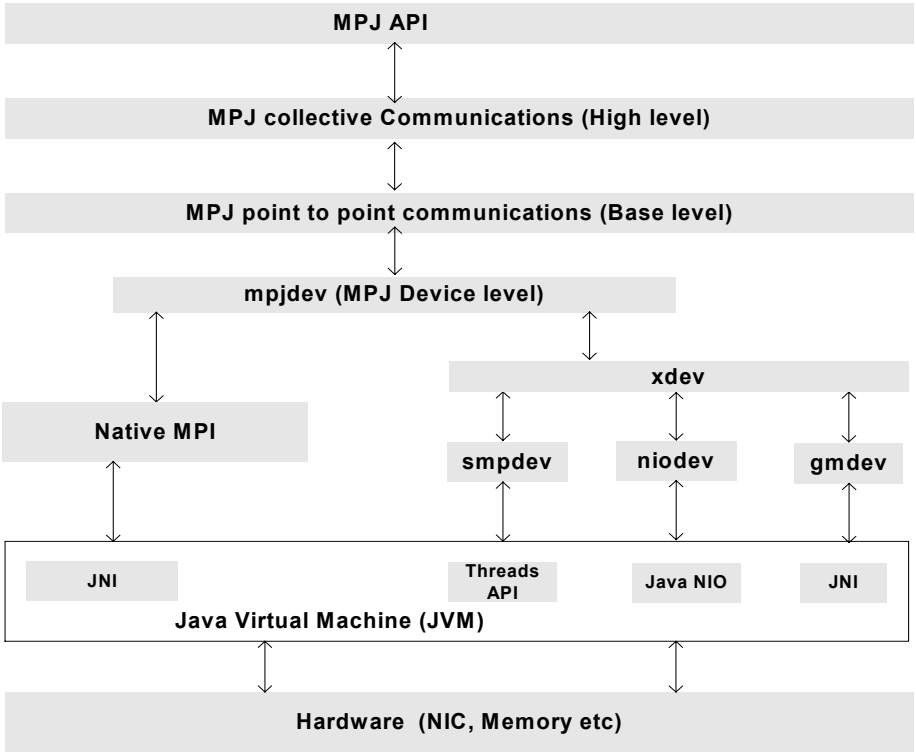

Figure 1: MPJ design

The high and base level rely on the MPJ device [11] and *xdev* level for actual communications and interaction with the underlying networking hardware. We envisage two implementations of the MPJ Device level. The first implementation provides JNI wrappers to the native MPI implementations. The second implementation uses the lower level device called *xdev* to provide access to Java sockets, shared memory, or specialized communication libraries. *xdev* is not needed by wrapper implementation because native MPI is responsible for selecting and switching different communication protocols. Figure 1 also shows three implementations of *xdev*, the first is *smpdev*, which is the shared memory device for MPJ. The second is *niodev*, which is an implementation of xdev using the Java New I/O package. The third and the last one, is *gmdev*, which is an implementation of xdev using JNI to interface with the GM communications library.

3. MPJ Implementation and Performance

As part of the on-going development of MPJ, we are implementing the advanced features of the MPI specifications. These include derived-datatypes, virtual topologies, different modes of send, and collective communications. These high level features are implemented in Java, but if the underlying device uses the native MPI, it will also be possible to cut through directly to their native implementation. Also, MPJ includes devices for Java New I/O, shared memory, GM communications library (myrinet), and a native MPI device. Currently, we are focusing on the runtime system. It consists of starter and daemon modules that execute as the native OS service by using Java Service Wrapper [10] project. More details about MPJ implementation can be found in [11]. Details about performance evaluation can be found at [8] and will be presented in the conference.

4. Conclusion

Our initial benchmarks presented in [8] suggest that Java is a viable option for implementing a messaging system. MPJ provides the capability of swapping in or out different devices, using a pluggable architecture. Such a design allows the applications to choose the communication protocol that best suits their needs.

References

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

[2] 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.

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

[4] The Remote Method Invocation (RMI) Specifications,

[5] http://java.sun.com/j2se/1.4.2/docs/guide/rmi/

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

[7] M. Welsh and D. 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.

[8] M. Baker, H. Ong, A. Shafi, A Status Report: Early Experiences with the implementation of a Message Passing System using Java NIO, A DSG Technical Report, 6th October 2004, http://dsg.port.ac.uk/~shafia/res/papers/DSG_2.pdf

[9] Myricom, The GM-2 message passing library, http://www.myri.com

[10] The Java Service Wrapper Project, http://sourceforge.net/projects/wrapper/

[11] A. Shafi, MPJ: Implementing advanced features of the MPI standard, DSG Seminar Talk, March 05, http://dsg.port.ac.uk/~shafia/res/talks/mpj-spring05.ppt

[12] S. B. Lim, B. Carpenter, G. Fox, and H. Lee, A Device Level Communication Library for the HPJava Programming Language. In proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2003), November 2003