A Java–Based Tool for Testing Interoperable MPI Protocol Conformance

William George
National Institute of Standards and Technology
100 Bureau Drive Stop 8951
Gaithersburg MD 20899–8951
1–301–975–4943
william.george@nist.gov

John Hagedorn
National Institute of Standards and Technology
100 Bureau Drive Stop 8951
Gaithersburg MD 20899–8951
1–301–975–4339
john.hagedorn@nist.gov

Judith Devaney
National Institute of Standards and Technology
100 Bureau Drive Stop 8951
Gaithersburg MD 20899–8951
1–301–975–2882
judith.devaney@nist.gov

Abstract

Java and the World Wide Web are used as the basis of a tool that tests conformance with the Interoperable Message Passing Interface communication protocol. The user accesses the system through a Java applet that acts as the interface to a test server that is also written in Java. Test scenarios are expressed in scripts that are interpreted by a C–based interpreter. This interpreter is integrated with the test server using the Java Native Interface. A separate C program, also including the test script interpreter, is linked to the communication library to be tested. This program executes in coordination with the Java–based test server to test the implementation of the protocol. This framework has proven easy to implement, effective, and flexible; it may be useful in other software testing systems.

1. INTRODUCTION

Parallel computing environments are becoming increasingly common. These range from tightly integrated architectures like the SGI Origin 2000 to networks of (perhaps heterogeneous) workstations. These parallel environments require parallel programming models and tools to enable the development of application software that effectively uses the hardware resources. Message passing has become one of the more widely used models for application development in such environments.

The Message Passing Interface (MPI) is a standard application programming interface that implements this programming paradigm. MPI was standardized in an open international process with participants from industry, academia, and government [7] [8]. Version 1.1 of the MPI standard was completed in June of 1995; the MPI–2 standard was released in July of 1997 and specifies many extensions to the original standard.

This paper describes a flexible system for testing implementations of MPI libraries for conformance with the Interoperable Message Passing Interface (IMPI) communications protocol. This protocol enables diverse implementations of MPI to communicate and to interoperate. The users of our testing system are the developers of MPI libraries. Our system provides those developers with a framework within which their software can send and receive IMPI communications and those communications can be verified for conformance to the IMPI protocol. We have based our approach on the use of Java and the World Wide Web (WWW). Our approach was inspired by a Java/WWW–based tool developed at NIST that tests conformance to an object–based computer–integrated manufacturing framework [1].

A brief introduction to MPI is needed to describe the operation of the test software. MPI specifies a library of routines for passing messages between two or more processes. In this context, a message is a collection of values consisting of integers, floating point values, characters, or any other data type known by the MPI library. The processes are typically (but not always) separate instances of the same program with each instance performing the same computations on different portions of a problem.

As a simple example, the problem could be to find the maximum value in a large 1–dimensional vector of values. In this case, each process would be given a different portion of the vector in which to search. Without any communication among the processes, each process can know the maximum value only for its portion of the vector.

To get the final answer the processes must communicate with each other to compare values and collectively to determine the global maximum. A simple solution would be for all processes to send their local maximum to a pre–determined process, using the MPI routine `MPI_Send`, and have that process receive these messages, using the MPI routine `MPI_Recv`. The receiving process can then determine the global maximum and send it back to each of the other processes. It is also important to note that MPI defines ways of identifying processes and tagging messages that are sent between processes. This enables the application programmer great flexibility in sequencing messages.

This is an extremely simple example problem, and this description of MPI message passing overlooks many important details, but it should be sufficient for the purpose of describing our test software.

MPI–1.1 defines bindings for C and Fortran. MPI–2, in addition to specifying extended functionality, also defines bindings for C++ and Fortran 90. These standard message passing libraries enable the application programmer to develop parallel software that can be ported to a variety of parallel computing environments.

There are many implementations of MPI–1.1. There are two portable publicly available implementations (MPICH [3] and LAM [2]) and most vendors of high–performance systems have developed implementations of MPI–1.1 that are optimized for their platforms. MPI–2 implementations are under development, with some partial implementations available now.

It is important to note that while MPICH and LAM will operate in a heterogeneous computing environment (with machines from multiple vendors), the higher performance MPI implementations provided by the hardware vendors typically will not. The IMPI effort was motivated by a desire to retain the highly tuned performance of the vendor–supplied MPI implementations while enabling operation in a heterogeneous hardware environment.

The effort to define a protocol for MPI interoperation began in March of 1997 with the formation of the Interoperable MPI Steering committee. The committee has been meeting periodically since then. The current status of the protocol can be found in [4] and we expect that the first complete version of the protocol will be available by the time that this paper appears.

IMPI defines a communications protocol that enables messages to be passed between different implementations of MPI. In other words, IMPI enables a single MPI–based application to operate in such a way that a portion of the computing and message passing is handled by one vendor's MPI implementation and another portion is handled by another vendor's MPI. The participating MPI implementations pass data to one another via the protocols defined by IMPI. The IMPI protocols do not limit the number of MPI implementations that participate in a single job. The communications that IMPI defines are below the level of the MPI routines and IMPI does not change the meaning of any of the MPI calls and it does not add any new routines. Thus it does not require the application programmer to change how MPI is used. Furthermore, IMPI is intended to enable high performance to be retained for communications within a single MPI implementation.

Our organization, the National Institute of Standards and Technology (NIST), has been involved in the IMPI effort from the beginning as a facilitator. We organize the meetings, collate the reports, maintain a mailing list and ftp site, etc. As part of its role as a facilitator, NIST has developed a software tool that enables IMPI developers to test conformance to the defined protocol.

A conventional approach to providing a software test suite is to develop one or more programs, make them available for downloading, and then have the user compile and run them on an appropriate platform. The tester observes the output and other behavior of the program(s) for signs of errors. This is a perfectly reasonable approach, but several potential problems arise. For example, whenever the test software is modified, the tester must go back to the central repository and repeat the cycle of download, build, and execute. Furthermore, tests involving a communication protocol are complicated by the fact that multiple active participants are required to execute a single test.

We believe that the framework provided by Java and the World Wide Web alleviate the problems of test delivery, modification and coordination. Java's support of threads, sockets, and the Java Native Interface (JNI) were particularly important in achieving our goals. The ability to create a front–end to the system that is accessible through the web was also central to our design. We believe that this approach may be of use for other types of software test systems.

2. DESIGN

2.1 Overview

The purpose of the IMPI test system is to assist MPI library developers in implementing the IMPI protocol. We assist them by providing a testing environment that includes a user interface in the form of a Java applet and a set of tests that fully exercise the protocol. This test system is needed because a developer can only test adherence to a communication protocol by communicating with a second party that is also using that protocol. Our IMPI test system provides that second party for the IMPI protocol and provides the means to selectively test portions of the protocol. Note that the IMPI test tool only tests conformance to the communication protocol; although MPI is usually used in the context of high–performance computing, this test tool does not test (and is not concerned with) performance.

For the purpose of this discussion, the most important aspect of IMPI is that it is specified as a protocol that operates over TCP/IP in the communications protocol stack. For performance reasons we expect that, under normal operating conditions, IMPI would be used to connect systems over a local network. However, the protocol does not prevent us from connecting to any system that is accessible over a TCP/IP network, including any system connected to the Internet. Because the advantages of centralizing the test software far outweigh the need for high performance during testing, the majority of our testing software has been designed to always run on one of our local machines with IMPI communications channels opened to the developer's machine over the Internet.

The user interface, which typically runs on a machine separate from the one running the IMPI software that is being tested, is connected to the main testing application through its own separate TCP/IP channels.

Some terminology is helpful in describing the general testing scheme we have used. Because of the similarity of the task, this terminology has been borrowed from the collection by Linn and Uyar [6] on the testing of OSI (Open Systems Interconnection) communications protocols. The implementation of IMPI that is to be tested will be referred to as the *IUT* (Implementation Under Test). The IUT is not a stand−alone program, but is part of an MPI library that is compiled for a specific architecture and then linked to a special test interpreter that we provide. The IUT, the MPI library that includes the IUT, the test interpreter (to be described later), the actual machine that is to run the IUT, and the supporting run−time software are collectively referred to as the *SUT* (System Under Test). The testing scheme is further complicated by the fact that we are necessarily testing on multi−processor machines that may also include a separate front−end host machine. To simplify our discussion, we will assume that the SUT can be viewed as a single machine with a single Internet address. Finally, the developer that is using our IMPI testing system will be referred to as the *tester*.

The concept of a *protocol stack* is useful when describing a communications protocol. Referring to Figure 1, each vertical set of boxes forms a protocol stack. The only protocols shown in this figure are IMPI and the two protocols that IMPI must interface with, MPI and TCP. The other boxes in the figure represent parts of our IMPI testing software. Messages originate high in the source stack, in our case in the MPI layer, and are passed down through lower−level protocols until they reach the network layer. On the destination end, each message is passed up the stack until it reaches the same protocol level as the originating message. The horizontal dashed lines in the figure indicate that each layer of the source stack communicates *logically* with the corresponding layer in the destination stack. The IMPI protocol is specified such that a user of MPI does not need to know anything about the IMPI protocol in order to use MPI. This separation of protocol levels allows flexibility for the developer in implementing IMPI.

The testing that we perform on IMPI implementations is called *black box* testing. This means that we make no assumptions about how the protocol is implemented and make no attempt to use such knowledge in the design of the testing system. All that we rely on is the specification of the protocol and the responses we receive from the IUT when we communicate with it. In order to test a communications protocol there needs to be access to the data being transmitted so that it may be inspected and verified as being in conformance with the protocol. This inspection of the transmitted data typically occurs at one or two points along the transmission path called *PCO*s (points of control and observation). The PCOs occur at the boundary of the layers in the protocol stack.

Referring to the protocol stack in Figure 1, some possible PCOs are at the IMPI−MPI interface and at the IMPI−TCP interface. If we were testing locally, that is, if we were testing directly on the machine that is running the IUT, then those would be the obvious choices for the PCOs. Our tester, however, is designed for remote operation (relative to the SUT) so we chose to move our PCOs to the remote system. The box labeled *Lower Tester* in Figure 1 should reveal the same stream of data as if we were observing the data directly below the IUT IMPI layer. This is called the Lower Tester since it is observing the data exchanged between the IMPI layer and the protocol layer below it. Similarly, an *Upper Tester* observes the data stream between the IMPI layer and the layer above it, which in this case is MPI. The Upper Tester in our system is also on the remote system relative to the SUT and so it does not directly observe the IMPI−MPI data stream on the IUT. Our Upper Tester relies on the design of the tests to exercise the IUT in ways that allow us to indirectly observe operation of the IUT.

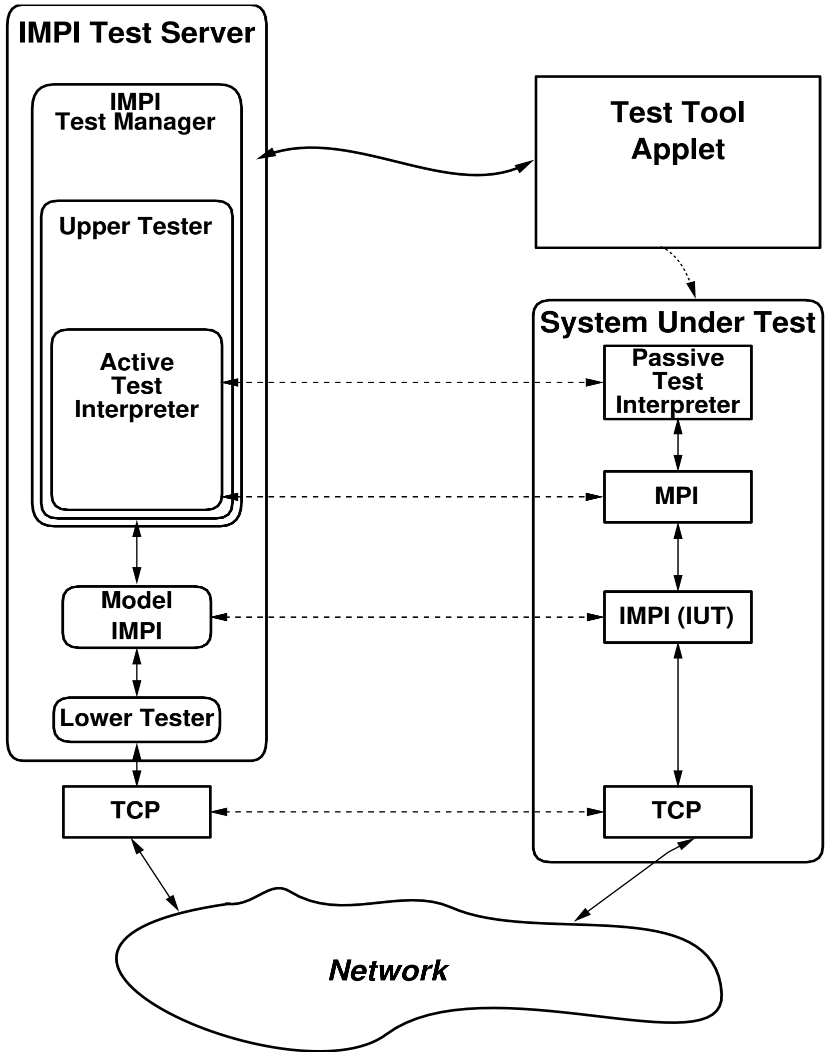

Figure 1. Communications Protocol Stack. Each vertical stack of boxes represents a protocol stack on a single machine.

The Upper Tester is based on a simple interpreter designed specifically for this system. This interpreter understands a subset of C with MPI communication routines built–in. On the SUT, the program that encompasses the interpreter is linked with the MPI library that contains the IUT. In this context, it is referred to as the *Passive Test Interpreter* since it does not have any direct interactions with main testing software. On the other end, the interpreter is linked with a Java implementation of a subset of MPI that is integrated into the Upper Tester. The tests we run, which are in the form of interpreted scripts, are designed such that we can safely conclude that the IUT is operating correctly if the data we receive back from the IUT is correct for the test that is being run. This aspect of the testing is explained in more detail in the Implementation section below. The use of a Passive Test Interpreter on the SUT minimizes the possibility that the tester will need to update the interpreter as we add and improve the tests. Most changes and additions to the testing system will take place in the scripts and in the other software that is run and maintained only at NIST. This approach is sufficiently flexible that it would be quite feasible to enable the tester to write scripts and to submit them through the applet.

The software that runs on the remote site, that is, at NIST, is represented by the stack of boxes on the left side of Figure 1 labeled *IMPI Test Server*. The IMPI Test Server includes the *Test Manager,* which coordinates the selection and running of test script and communicates with the *Test Tool Applet* that is running on the tester's workstation. The Test Server also includes the Upper and Lower Testers, the Model IMPI implementation used to accept and send messages in the IMPI protocol, and the *Active Test Interpreter* which interprets the tests within the Test Server. Within the implementation of the Test Server the Upper and Lower Testers are logical entities rather than actual programs or separate threads. The Upper Tester consists mostly of the test scripts. These scripts make MPI communications calls in order to exercise the IMPI protocol and then determine whether the IUT performed correctly based completely on the messages received back from it during the execution of the test script. The Lower Tester consists of tests embedded in the Model IMPI implementation.

During a typical test session, there will be multiple copies of the Passive Test Interpreter Program running simultaneously on the SUT. Similarly there will be multiple instances of the Active Test Interpreter running in separate threads within the Test Server. These will all be communicating with each other as they interpret the MPI calls specified in each test script. Communications between the Passive Test Interpreters on the SUT and the Active Test Interpreters must be accomplished through the IMPI protocol. It is these communications that are monitored for conformance to the IMPI protocol specification.

2.2 The Testing Process

We expect that the tester will be using a workstation, separate from the SUT, to initiate and coordinate the running of the tests. Under this assumption, we have at least three machines involved in the testing: the tester's workstation, the SUT, and the system that runs the IMPI Test Server. One piece of the testing system that is not shown in Figure 1 is the *IMPI Test Server Daemon*, a Java application, which runs continuously on a machine at NIST with the task of spawning new Test Servers as needed. The IP address and TCP port for this daemon are known by the IMPI web page so that it may be contacted when needed.

The tester's first action is to connect to the main IMPI web page at the URL `http://impi.nist.gov/IMPI` and to follow the link to the IMPI Test Tool page. On this page the tester has access to the latest Passive Test Interpreter Program, all of the test scripts, and a page of instructions on how to start and use the IMPI Test Tool Applet. If the tester has not previously obtained the Passive Test Interpreter, or if the current version is newer than the version the tester has, then they must download this new version. At some point before beginning the testing, the tester must compile this test interpreter and link it with their MPI library that contains the IMPI IUT.

Next, the tester can start the IMPI Test Tool Applet by clicking on the *Run the IMPI Test Tool Applet* link. This will actually start two programs: a Test Server on the NIST side, and the IMPI Test Tool Applet on the tester's side. The applet starts the Test Server by sending a message to the Test Server Daemon requesting a new Test Server. The Test Tool Applet will pop up in a separate window on the tester's machine. A TCP/IP socket is opened between the new Test Server and the Test Tool Applet that will remain active until the tester shuts down the Test Tool. This is the communications channel that carries all the commands from the Test Tool Applet to the Test Server as well as test results that are sent from the Test Server to the Test Tool Applet.

Before starting the Passive Test Interpreter on the SUT, several test configuration parameters must be set by the tester in the Test Tool Applet. These parameters describe some aspects of the SUT, such as the number of processes, that are important in determining which tests can be run, as well as identifying which side of the connection will control the startup process. The IMPI startup protocol, in which all of the participating systems connect to each other and negotiate various communications parameters, is an important part of the IMPI protocol. The startup of an IMPI job is coordinated by a separate process that can be run on any machine connected to a network that is visible to all participating systems.

Finally, the Passive Test Interpreter Program is started on the SUT. The starting of the Passive Test Interpreter is not automated. The tester must start this program using whatever method is appropriate for their system. The tester has the choice of running only the IMPI startup protocol and quitting, or continuing on after the startup protocol has completed and allowing the system to initialize the Active and Passive Test Interpreters so that they are ready to accept test scripts. The tester may then choose one or more test scripts to be run. This is controlled by the tester from the Test Tool Applet. The selected test scripts are read from disk by the Test Server and are transmitted as plain–text source code to the Passive and Active Test Interpreters. These interpreters then simultaneously execute the scripts. Each script is intended to exercise and test some aspect of the IMPI communication protocol. The results of the tests are determined by the Test Server and sent to the Test Tool Applet for display to the tester. The source code of the test scripts are also available to the tester, via the IMPI web page, so that the results can be better understood. In addition to the final pass/fail result for each test, a scrolling window of text messages is displayed in the Test Tool Applet giving any relevant information on the progress of each test that might be useful to the tester.

2.3 Summary

The overall design of our IMPI testing system enables us to make sure that all testers are using the latest version of the Test Server, the Test Tool Applet, and all the test scripts, without requiring any effort by the testers to keep up–to–date. This also reduces the requirement for producing documentation related to the compiling and installation of the test software on a wide variety of architectures as well as the additional testing that this would require on our part. The preliminary reaction from the IMPI vendors on this test system design has been positive.

3. IMPLEMENTATION

3.1 Test Tool Applet

The IMPI Test Tool Applet is the link between the tester and the Test Manager. It enables the tester to setup the IMPI configuration for the test session, and to select test scripts for execution. Test results are then displayed to the tester through the applet.

The IMPI test applet implementation fulfills the following goals:

- It is relatively passive, serving as a conduit of information to and from the tester.
- It embodies a minimal amount of knowledge about MPI or the rest of the test system.
- It is small, thus minimizing download time when it is launched.

- It provides a simple but effective interface to the test functionality.
- It requires no special security arrangements.

These objectives are accomplished primarily through a graphical user interface (GUI) implemented directly in the Abstract Window Toolkit (AWT). We decided not to use the Java Foundation Classes (JFC) because the package was not released at the time that we began this project. We did however use the Java 1.1 event model, although this subsequently required some of our users to upgrade their browsers. The applet interface is shown in Figure 2. Note that the GUI is organized into three sections corresponding to the applet's three major functions: configuration setup, test script selection, and test result reporting.

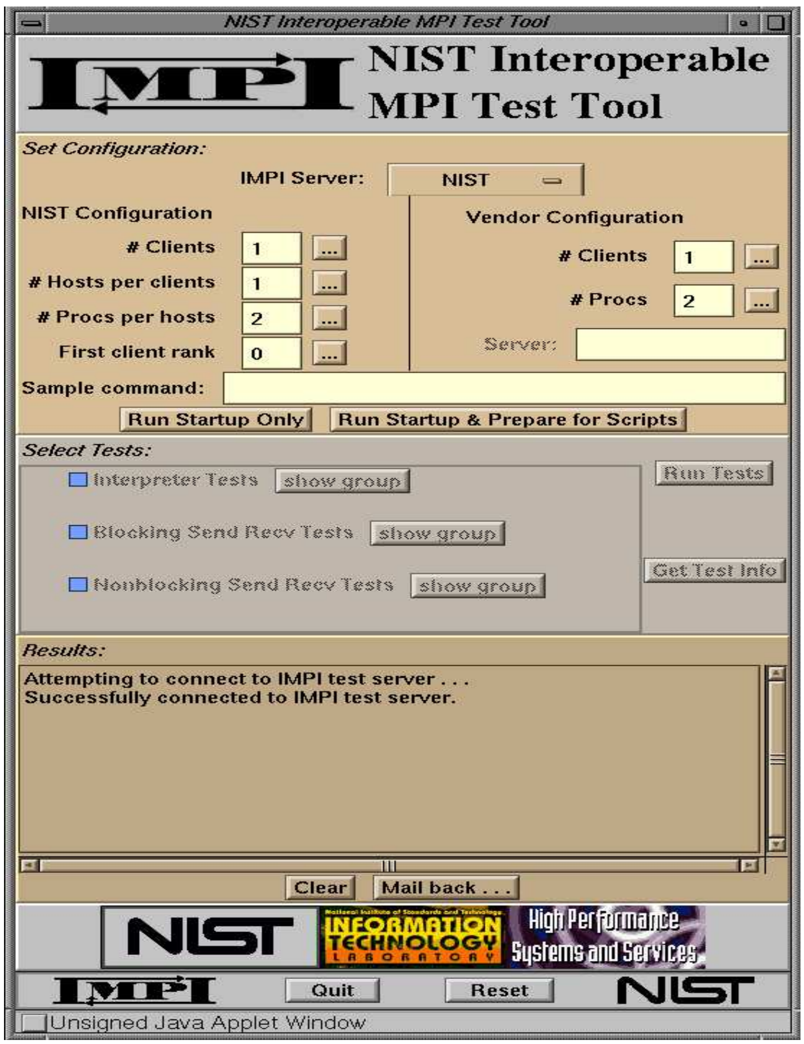

Figure 2. IMPI Test Applet Interface. The interface is divided into three sections: configuration, test script selection, and test result reporting.

When the applet is launched, it first opens a socket connection to the Test Server Daemon that is running on the IMPI web server host. The applet finds the host:port address of the server by reading it from a file that also resides on the IMPI web server. The Test Server Daemon spawns a Test Manager with which the applet coordinates the test activities.

The applet reads a file specifying the test scripts that are currently available; this file also indicates a hierarchical grouping of scripts. The information about the scripts is used to construct the test script selection interface that appears in the applet GUI. This enables us to add or delete test scripts without changing the applet code. Note that because these I/O connections are made with the web server host from which the applet was launched, the applet operates within the "sandbox model" and requires no digital signature.

As indicated above, the applet serves as a link between the tester and the Test Manager. Actions by the tester on the GUI generate a series of messages to the Test Manager through their socket connection. Similarly, the progress of the test executions generates a series of messages back to the applet describing errors, test status, and so on. The applet has no direct communication with the Passive Test Interpreter Program on the SUT. All information about the status of tests on the SUT must go first to the Test Manager, which relays that information to the applet for display to the user.

Each message that is sent between the applet and the Test Manager is in plain ASCII and is terminated with a carriage return. This enables us to use simple `PrintWriter` and `BufferedReader` objects to send and receive messages. We decided to use ASCII messages to simplify debugging and because we expected the volume of information to be relatively small. This decision has served us well.

In addition to the threads managed by the AWT, the applet executes in two major threads. One thread monitors incoming messages from the Test Manager and conveys them to appropriate message–handling methods. The other thread responds to user interactions and reconfigures the GUI as needed.

3.2 Test Manager

The Test Manager is responsible for several major tasks including: all communications to and from the Test Tool Applet, starting and supporting the Active Test Interpreters, sending the requested test scripts to the Active and Passive Test Interpreters, and monitoring the IMPI communications generated by the test scripts.

There is always a Test Server Daemon, running on our IMPI server machine, waiting to accept new connections from any Test Tool Applet. The section above on the Test Tool Applet described how this connection is made. Once this connection is made, a new Test Server is started for this session of testing. The IMPI Test Manager is the main executable in the Test Server. All other parts of the Test Server exist either as threads spawned by the Test Manager or, in the case of the Lower Tester, as a logical entity consisting of scattered pieces of code within the Test Manager

When the tester requests that testing be started, the Test Manager creates one or more *Hosts*, the number being specified in the Test Tool Applet. A Host in this context is an IMPI specified entity which manages the sending and receiving of IMPI messages for a group of one or more MPI processes. It is these Hosts that are connected via TCP/IP and not the MPI processes directly. Within an IMPI job, each MPI process is always associated with exactly one Host. MPI messages that are passed between two processes that have the same Host are handled directly by the native MPI communications routines, that is, no IMPI protocol is involved in transmitting these messages.

All other communication, which we will refer to as IMPI communication, must pass through the process' Host. As part of the IMPI startup protocol, each Host establishes a TCP/IP connection with every other Host to form a completely connected network of Hosts. For our system, if there are a total of H Hosts participating in the testing, then each Host within the Test Manager starts up $H-1$ threads, each of which connects to and listens for messages from one of the other Hosts. These threads are referred to as the *Packet Interfaces* since their task is to accept packets of data from the other Hosts.

The tester also specifies the number of *simulated* MPI processes that must be run by the Test Manager. These are called simulated MPI processes since they are not run within a full implementation of MPI, but instead are implemented as separate Java threads which are started by the Test Manager. The test interpreter which these simulated MPI processes run is implemented in C and linked to the Java based Test Manager using the JNI. To simplify the Test Manager, only MPI routines that are used in the test scripts are implemented within the Test Manager. These routines are called from the Active Test Interpreter using the JNI. Also, these supporting MPI routines are implemented with the emphasis on verifying the correct execution of IMPI and not on communications performance.

The Packet Interfaces only handle incoming messages. Outgoing messages are handled separately. For each outgoing message that a simulated MPI process sends, a separate thread is spawned which exits only after completing the message. Each simulated MPI process can interrogate its Host to determine when a message has been completed, or it can use an MPI routine which simply waits until the message has been sent before returning. The methods `wait` and notify of the Java `Thread` class are used to manage the interaction between the simulated MPI processes and the threads that handle their communications. The use of Java threading application programming interface simplified the implementation of these simulated MPI processes and their supporting Packet Interfaces and message sending threads

Some special routines called within a test script are handled differently in the Active Test Interpreter compared to the Passive Test Interpreter. These involve mostly the reporting of test results. For example, a script can call either the `report_pass`, `report_fail`, or `report_indeterminate` routine to indicate the outcome of the test. On the SUT this will usually print a message to the standard output stream. Within the Test Manager, these calls will be monitored with the final results being sent to the Test Tool Applet for display. If none of the simulated MPI processes call either `report_fail` or `report_indeterminate`, then the test will be reported as a pass.

The Lower Tester is comprised of tests scattered throughout the Test Manager, especially within the Packet Interface threads and other methods that handle the low–level IMPI packets. These tests examine the IMPI packets and report any problems discovered. These reports are then relayed to the Test Tool Applet output window and, if appropriate, the test result within the Test Manager is set to Fail.

3.3 Passive Test Interpreter Program

In order to test a vendor's implementation of the IMPI protocol, we provide a program that the vendor must compile and link to the library to be tested. We call this the Passive Test Interpreter Program. This program makes calls to MPI routines in order to exercise the protocol. We did not, however, want to encode the test scenarios directly into this program (or into a large number of special–purpose programs). In order to achieve this, we embed an interpreter into the program; the actual test scenarios are embodied in test scripts that are delivered from the Test Manager during the course of a test session. This program is passive in the sense that its primary purpose is to wait for test scripts to arrive and to execute the received scripts.

Note that the MPI routines are defined with C and Fortran bindings, but not with Java bindings. Thus we decided to provide the Passive Test Interpreter Program in C in order to minimize the difficulties in compiling, linking, and supporting this software in the vendor's environment.

The Passive Test Interpreter Program is a rather thin wrapper around the interpreter software, which is described in detail in the next section. This wrapper handles:

- interpreter startup and shutdown,
- various handshaking with the Test Manager,
- the receipt of scripts,
- the linking of the interpreter code to the MPI library calls that are the main subject of the test scripts, and
- the reporting of test status and errors detected by the Passive Test Interpreter Program.

From the tester's point of view, the Passive Test Interpreter Program is simply a C program that makes MPI calls. It is executed simultaneously in all of the processes of a single MPI job. When the tester starts up the Passive Test Interpreter Program, the IMPI startup protocol is executed. When the Test Manager verifies that the startup protocol has successfully completed, a short handshake is executed between one of the simulated MPI processes that is executing under the Test Manager and each copy of the test program on the vendor's side. This handshake is done using the most basic MPI message passing routines to verify that a stream of bytes can be correctly passed from the Test Manager to the Passive Test Interpreter Program. This communication channel is then used to pass the test scripts to the Passive Test Interpreter Program. The scripts are simultaneously interpreted by the interpreters that are embedded in both the Passive Test Interpreter Program and in the Test Manager.

3.4 Test Script Interpreter

As mentioned above, the test script interpreter is embedded in both the Test Manager (as the Active Test Interpreter) and the Passive Test Interpreter Program. In the Test Manager this embedding involves extensive use of the JNI, with Java code invoking C code and vice−versa. The same scripts are interpreted at the same time in the Test Manager and in the SUT. The interpreter's use differs only in how it is invoked and how data passes into and out of the interpreter. The actual interpretation code is identical in the two environments.

The interpreter is designed to handle scripts written in a simple C−like language. It has all of the basic data types (`int`, `float`, etc.) as well as several data types needed for the MPI functionality (such as `MPI_Comm`). Most of the MPI−1.1 routines can be called within a script and all of the symbolic constants defined in MPI−1.1 are available. Although the syntax is C−like, the language does not allow many features of C such as aggregate data types, pointers, and user−defined functions. We decided not to handle these language features because they are not necessary for our test scripts and because they would have imposed many complications in parsing and execution.

The language is defined using the standard Unix utilities lex and yacc. The language specification is used to generate parser code in C. This parser and the stack–based machine that executes the parsed code are then integrated into both the Passive Test Interpreter Program and the Test Manager. Although the code for the two interpreters is identical, there are, of course, substantial differences in how the interpreter code is integrated into the two environments.

The interpreted scripts include statements that correspond to (and look almost identical to) MPI calls such as `MPI_Send` and `MPI_Recv`. The different handling of these calls in the two environments is accomplished by having the interpreter internally invoke wrapper routines for each of these MPI routines. We then provide a different set of wrapper routines for each environment. In the Passive Test Interpreter Program, each wrapper routine simply invokes the corresponding MPI routine from the vendor's library. Within the Test Manager, the wrapper routine uses the JNI to call an appropriate Java routine to accomplish the necessary IMPI communication.

It is also interesting to look at how the test script itself is provided to the parser in the two environments. In the Test Manager, the script file name is constructed in the Java code; it is then passed via the JNI to a C routine that reads the entire file into a memory buffer. This buffer is then sent with a JNI call via our Java equivalent of the `MPI_Send` routine to each of the MPI processes on the SUT and to those that are simulated within the Test Manager. Within each of these real and simulated MPI processes, the script is received and passed to the interpreter for parsing and execution. Note that on the SUT, each MPI process is a separate copy of the Passive Test Interpreter Program.

The interpreter was a critical feature of the IMPI test software. Its integration into the Test Manager's Java environment presented some challenges, but the use of the JNI enabled us to accomplish this integration relatively easily. We were particularly pleased that there were no difficulties when the entire system was ported to a different machine. Java and the JNI lived up to their goal of full portability.

3.5 The Role of Java in the Implementation

Even though we had little experience with Java prior to this project, we chose to implement much of this testing system in Java for several reasons. Java was a natural choice for implementing the user interface due to its unique relationship with the WWW. Most WWW browsers are capable of automatically downloading and executing Java applets. This allows us to maintain the Test Tool Applet locally without the need to distribute a program, along with the required compiling and installation instructions. This has the added benefit that testers will always be using the latest version of the tool.

The majority of the Test Server, which consists of the Test Manager, Test Interpreter, model IMPI, and all the code that comprise the Upper and Lower Testers, was also implemented in Java. Two aspects of Java that greatly influenced this choice are the simple interface for the use of TCP/IP sockets, and the support for threads, both of which we expected to use heavily. Also, since we did not know the ultimate target architecture for the Test Server, the portability of Java was relied upon. The test interpreter was initially written for the SUT, and so it was implemented in C to link with the C version of the MPI library. The use of the JNI allowed us to use this same interpreter in the Test Server.

We developed the Test Server and the Test Tool Applet on SGI workstations running the IRIX operating system. The only restriction we have encountered is that the web browser used to access the IMPI Test Tool Applet must support Java 1.1.

The host that is now used to run the Test Server Daemon, and therefore the Test Servers, is a Sun workstation running the Solaris operating system. The move from the SGI to a Sun environment caused no problems, even for the JNI interface to the C based interpreter. We continue to do development work on the SGIs and to move the updated system over to the Sun server as needed.

3.6 System Size

Work on the IMPI test software has been underway since October of 1997. It is a bit difficult to estimate the amount of effort that has been expended because it has been developed in a research environment with several other on–going projects. We believe that we have spent somewhat less than one person–year to date. At times, some major and minor modifications were required due to changes made to the IMPI protocol specification during development.

The size of the software system can be broken down in this way:

Subsystem		# lines of code	# classes
Test Manager	(Java)	10526	43
Applet	(Java)	3564	30
Interpreter	(C, lex, yacc)	13496	––

These line counts do not include the C code that is generated by lex, yacc, and javah. This generated code accounts for an additional 2288 lines of C.

4. CONCLUSIONS

The use of Java and the World Wide Web aided the development and deployment of the IMPI protocol test system in a variety of ways. In particular, the Java/WWW applet–based approach provided the following advantages over other approaches:

- It greatly simplified the delivery of the test capability to the user. It is accessed directly through a standard web browser.
- There were far fewer configuration issues to be resolved on the user's side.
- Documentation needed for the user was considerably simpler.
- The software development process was facilitated by Java's interface for the use of threads, sockets, and native code through the JNI.
- The test system is easily modified and augmented without requiring any action from the user.
- There are far fewer version control problems on the user's side.
- The server–side was easily ported from the development system (SGI) to the web–server (SUN) on which it was deployed.

A few problems were encountered:

- The use of the Java 1.1 event model required some users to upgrade their browsers.
- The use of a firewall at one user's site prevented the use of the system directly from the NIST web site.

The preliminary reaction of the IMPI developers has been very positive. They seem very pleased by the general approach of accessing the test capability through the Web, and they appreciate the ease of use of the system.

In this project Java fulfilled its write−once−run−anywhere promise; this was an important factor in our success. We feel that this Java−based client/server approach with an applet front−end provides a framework that can be used in a variety of applications. For example, the testing of the IMPI protocol is similar to the testing of other communications protocols that can be viewed as application−level protocols in the 7−layer OSI−RM (Open Systems Interconnection−Reference Model) communications stack [5]. This software could easily be adapted to test such protocols.

5. ACKNOWLEGEMENTS

The authors would like to acknowledge the hard work of the IMPI Steering Committee in developing the IMPI protocol. They have also provided us with valuable guidance and suggestions as we developed this test system. We would also like to acknowledge the work of Kevin Brady and James St. Pierre of the Electricity Division of the Electronics and Electrical Engineering Laboratory at NIST. Their work on a Java/WWW based test tool inspired our design and provided us with many ideas. We thank the anonymous reviewers for improving our paper through their comments. In particular, the possibility of allowing testers to submit their own scripts was suggested by a reviewer.

6. DISCLAIMER

Certain commercial equipment and software may be identified in order to adequately specify or describe the subject matter of this work. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the equipment or software is necessarily the best available for the purpose.

7. REFERENCES

[1] Brady, K. G. & St. Pierre, J., *Conformance Testing Object−Oriented Frameworks Using JAVA*, NISTIR 6202, National Institute of Standards and Technology, 1998

[2] Burns, D. & Daoud, R. B., "LAM: An Open Cluster Environment for MPI", *Supercomputing Symposium '94*, Toronto, Canada, code available at http://www.lsc.nd.edu/lam/.

[3] Gropp, W., Lusk, E., Doss, N., & Skjellum, A., "A High−Performance, Portable Implementation of the MPI Message Passing Interface Standard", *Parallel Computing*, vol. 22, no. 6, pp. 789−828, Sept. 1996.

[4] IMPI Steering Committee, *IMPI – Interoperable Message−Passing Interface*, ftp://ftp.nist.gov/pub/hpss/interop/impi−report.current.ps, 1998.

[5] ISO: International Organization for Standardization, "Information Processing Systems – Open Systems Interconnection – Basic Reference Model, IS 7498, ISO Secretariat for JTC1/SC21. ANSI, 1984.

[6] Linn, R. J. & Uyay, M. Ü., Eds., *Conformance Testing Methodologies and Architectures for OSI Protocol*s, IEEE Computer Society Press, 1994.

[7] Message Passing Interface Forum, "MPI: A Message−Passing Interface Standard", *International Journal of Supercomputing Applications*, Vol. 8, no. 3−4, 1994.

[8] Message Passing Interface Forum, "MPI−2: A Message−Passing Interface Standard", *International Journal of Supercomputer Applications and High Performance Computing*, vol. 12, no.1−2, 1998.