JMPF: A Message Passing Framework for Cluster Computing in Java

RAJKUMAR
School of Computing Science
Queensland University of Technology
825a, Level 8, ITE Building,
Gardens Point Campus, Brisbane, Australia
rajkumar@fit.qut.edu.au

VIJAYA NAGAMANI
Operating Systems Group
Centre for Development of Advanced Computing
2/1, Ramanashree Plaza, Brunton Road
Bangalore, India

Abstract*:* *As we enter 21st century, our dependency on the parallel or distributed computing to solve large scientific problems is going to increase each day. This observation has inspired many researchers to invest their efforts into the development of efficient platform independent Message Passing Interface. JMPF (A **M**essage **P**assing **F**ramework for Cluster Computing in **J**ava) is one alternative interface developed in Java. Using JMPF, programmers can easily design and develop application programs for cluster computing. In this paper, we describe the need of JMPF and JMPF components, architecture, implementation, Application Programmer's Interface, interaction between its components, and sample applications.*

1. Introduction

High-speed network and improved microprocessor performance are making networks of workstations an appealing vehicle for cost effective parallel and distributed computing [1]. By relying solely on commodity hardware and software, network of workstations (NOW) or cluster of workstations (COW) can offer parallel processing at low cost. A large network interconnects many computers from different vendors having different machine architectures and running different operating systems. In such a network of heterogeneous systems, the programming model offered must be portable.

To move into the real era of high performance computing, the operating environment of these machines must provide a unified system view or unified access to system resources, which is popularly called as a Single System Image (SSI). The user need not aware of the underlying system architecture to use these machines effectively. The programmer must be provided with the view of globalized file system, processes, and network. This allows the user to access system resources such as memory, processors, network, etc., transparently irrespective of whether they are available locally or remotely.

As stated above, high performance computing on commodity hardware is gaining wide acceptance. For this to be practicable, these HPC systems need to support a single system image at any one or more of the following levels:

- Hardware Level
- Operating System Level
- Message Passing Interfaces Level
- Language / Compiler Level
- Tools / Applications Level

The above approaches for achieving unified access to system resources have been discussed in the paper [2]. In this paper, we discuss a framework in Java, which provides a unified access to network at language level, and at the same time offers program portability and allows heterogeneous computing.

Java has many features, as highlighted by its definition [3], "Java: A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multi-threaded, and dynamic language". Among these features, Java makes network programming easier by encapsulating connection functionality in socket classes [4]. Java is mostly used in writing distributed computing programs because of its portability feature and it goes a lot further than most languages to obtain not mainly just portability but identical program behavior on different platforms.

In distributed computing, an application is made up of multiple jobs/tasks. Jobs execute on different nodes in a NOW/COW and they communicate with other jobs for sharing/distributing data and results. Jobs of a distributed application developed using generic Java communicate with each other through sockets, which are basically a communication channel through which messages can be sent or received. But communicating through sockets is complex and there is no guarantee that the sockets that are free in one platform are also free in another platform and hence, socket specific applications suffer from portability. Even if programmer tries to overcome this by finding available free sockets at runtime, but managing them becomes very tedious task when an application is made up of multiple jobs. It is also difficult for programmers to remember socket numbers while developing an application.

The above stated difficulties necessitated the requirement of a transparent message passing system, which has led to the design of JMPF, a **M**essage **P**assing **F**ramework for Cluster Computing in **J**ava. It proposes the following: Instead of communicating through predefined sockets we can have a mechanism to detect the available free socket in the host machine at runtime and name each socket by unique name. These named sockets, *which we call as ports in JMPF*, are easy to remember, manage, and operate.

2. JMPF Architecture

In Java, message passing occurs when a Java program executing in one address space (machine) communicates with another Java program executing in an entirely different address space.

JMPF supports both Point-to-Point (one to one) and Group (one to many) communication. Point to point communication allows one or multiple senders to send messages to a single destination port. Group communication allows a single sender to communicate to multiple destinations simultaneously. In group communication, the ports are assembled into *Port Groups*. When a message is sent to a port group, all the members of the group receive it. Port groups can be created and destroyed dynamically. Ports can be inserted to or removed from a port group. Also, a port can be a member of more than one port group at the same time.

The architectural view of the **JMPF** is shown in Figure 1. This architecture is the fundamental building block for all the classes that will involve in the Message Passing Framework. The rest of this section outlines the features and design of JMPF as portrayed in the Figure 1.

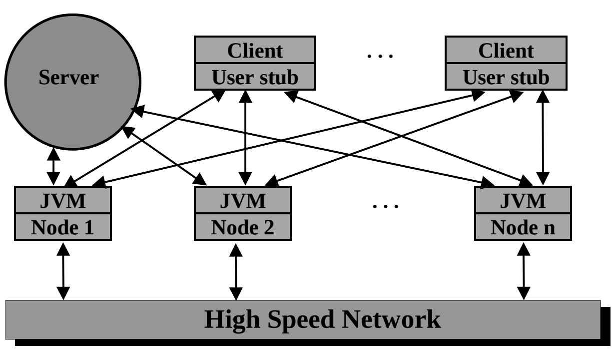

Figure 1: Architectural view of the system

The main function of the JMPF server should be to serve the clients in the network, who wish to communicate with each other, by providing information about the jobs, tasks, groups, and ports needed for communication.

The JMPF user stub should provide the transparent means for message passing among tasks of an application running on network of computers. This should provide the naming of ports and groups for communication, manipulate the ports/groups and provide means for communication through ports using point to point and group communication mechanism.

The services port management and port communication offered by JMPF are discussed below:

Port Management Functions allows to

- create a port by the given name
- locate a port by the given name
- delete a port by the given name
- create a group by the given name
- locate a group by the given name
- delete a group by the given name
- add a member port to the specified group
- remove a member port from the specified group
- show the status of the specified group

Port Communication Functions allows to

- create the DataInputStream and PrintStream, to and from the specified port.
- send the messages through the specified port or through the Output Stream.
- receive the message through the specified port or through the Input Stream.
- send the messages to the specified group synchronously

The message-passing mode can be **synchronous** or **asynchronous.** In synchronous mode of communication, the *sender* blocks until the receiver receives the message and the *receiver* blocks until the sender does a send, whereas in asynchronous mode of communication, the sender does not block. If there is no receiver to receive the message, the message is queued or buffered. The receiver blocks and waits for message arrival when there is no queued or buffered message.

3. Design of JMPF

The main components of JMPF are **JMPF Server** and **JMPF** user stub (client interface). The stub resides on the client machine and the Server on any one of the host machines in the network. Table 1 shows the various services that are supported by the JMPF in coordination with and JMPF Server services.

JMPF Server stores all the information about different jobs of an application running in the network and its ports and groups for facilitating communication among them. Depending on the client request the server responds to the client, which helps in message passing among distributed jobs.

JMPF user stub is a Java class containing methods, which allows creation and management of named sockets. It also allows application tasks to communicate each other transparently.

JMPF User Stub	JMPF Server
public int CreatePortGroup(String) public int CreatePort(string) public int LocatePortGroup(String) public int LocatePortGroup(String,boolean) public int LocatePort(String) public int LocatePort(String, boolean) public int DeletePortGroup(String) public int DeletePort(String) public int AddMember(String, int) public int RemoveMember(String, int) public Vector ShowStatus(String) public PrintStream CreatePS(int) public DataInputStream CreateDIS(int) public int SyncSendGroup(String, String) public int SyncSendGroup(String, int) public int SyncSendGroup(String, char [ ]) public int Send(PrintStream, String) public int Send(PrintStream, int) public int Send(PrintStream, char[ ]) public int Send(int, String) public int Send(int, int) public int Send(int, char[ ]) public String BlockReceiveString(int) public int BlockReceiveInt(int) public char[ ] BlockReceiveCharArray(int) public String ReceiveString(int) public int ReceiveInt(int) public char[ ] ReceiveCharArray(int) public String ReceiveString(DatainputStream) public int ReceiveInt(DataInputStream) public char[ ] ReceiveCharAr- ray(DataInputStream) public void ClosePorts( )	Jobs Tasks Groups Ports

Table 1: JMPF class and JMPFServer services

When the client wishes to communicate with other clients, it accesses appropriate service from the JMPFServer. This is achieved by invoking a suitable method of the stub object (which is an instance of the client stub class) and then the stub object performs the requested operation in cooperation with the server.

JMPF hides all the low-level networking issues and offers a simple and transparent means for message passing.

Communicating with JMPFServer

To access services of the server, the user stub objects should know the address of the host machine where the server is running in the network. The IP address of the host machine where the server object is running is stored in the file `java.hpc`. Prior to creating a stub object for message passing, every client reads the IP address of the server from this file, for establishing communication path with the Server.

4. JMPF Implementation

The three components of JMPF are: JMPF server, JMPF client interface/stub, and JMPF clients (which are distributed tasks of an application).

JMPF Server

JMPF Server is designed as a multithreaded server so that it can serve multiple client requests simultaneously. It extends the standard Java `Thread` class to achieve this, so that for every client request, a thread object can be created to serve client requests.

JMPF allows execution of multiple applications of the same or different type simultaneously. This is achieved by creating a separate object space for each job, which maintains task objects, group objects and port objects for that job. The Hash Table data structure is chosen to store these objects as it allows storing of objects of different classes with a unique key into it.

JMPF Client Interface

JMPF client interface methods allow to create a port, locate a port, and to communicate over a port. In `CreatePort()` and `CreatePortGroup()`, the port name and group name respectively, are bound to sockets which are automatically selected. These details are maintained in the local hash table of a task creating it and also in JMPF server for global access. When a client other than owner of a communicator object (which is an instance of client stub) wishes to communicate to some other client, it establishes communication path with another client by accessing information maintained in the JMPF server. Once communication path is established, both clients communicate with each other directly without interacting with the server.

JMPF Class Specification

The specification of the classes `JMPF`, `GroupSocket` and `PortSocket` and their components such as data members, constructors, methods, etc. are shown below.

Class JMPF
Data members : **Hashtable** Groupids; **Hashtable** Portids; **PrintStream** ToServer; **DataInputStream** FromServer; **int** JobId; **String** Hostname; **int** HostId; **String** Servername;
Constructor : JMPF(int job_id, int task_id);

Methods :
public void InitComm( );
public String ReadHostname( );
public int CreatePortGroup(String GroupName);
public int CreatePort(String PortName);
public int LocatePortGroup(String GroupName);
public int LocatePortGroup(String GroupName, boolean Flag);
public int LocatePort(String PortName);
public int LocatePort(String PortName, boolean Flag);
public int DeletePortGroup(String GroupName);
public int DeletePort(String PortName);
public int AddMember(String GroupName, int PortNo);
public int RemoveMember(String GroupName, int PortNo);
public Vector ShowStatus(String GroupName);
public PrintStream CreatePS(int PortNo);
public DataInputStream CreateDIS(int PortNo);

public int SyncsendGroup(String GroupName, String Message);
public int SyncsendGroup(String GroupName, int Message);
public int SyncsendGroup(String GroupName, char [ ] Message);
public int Send(PrintStream PS, String Message);
public int Send(PrintStream PS, int Message);
public int Send(PrintStream PS, char [ ] Message);
public int Send(int PortNo, String Message);
public int Send(int PortNo, int Message);
public int Send(int PortNo, char [ ] Message);
public String BlockReceiveString(int PortNo);
public int BlockReceiveInt(int PortNo);
public char [ ] BlockReceiveCharArray(int PortNo);
public String ReceiveString(DataInputStream DIS);
public int ReceiveInt (DataInputStream DIS);
public char [ ]ReceiveCharArray(DataInputStream DIS);
public String ReceiveString(int PortNo);
public int ReceiveInt(int PortNo);
public char [ ]ReceiveCharArray(int PortNo);
public void ClosePorts( );

class GroupSocket
Data members : **String** GroupName; **int** GroupId; **ServerSocket** Group_socket;
Constructor : public GroupSocket(String GroupName, int GroupId,ServerSocket sk);
Class PortSocket
Data members : **String** PortName; **int** PortId; **ServerSocket** Port_socket;
Constructor : public PortSocket(String PortName, int PortNo, ServerSocket sk);

JMPFServer Class Specification

The specification of the classes `JMPFServer`, `Connection`, `Jobs`, `Tasks`, `Groups`, `Ports` and their components such as data members, constructors, methods, etc. are shown below.

class JMPFServer
Data members : **ServerSocket** Server_socket; **InetAddress** Address; **Hashtable** JobIds; **String** Hostname;
Constructor : public JMPFServer( );
Method : public void run( );
class Connection
Data members : **Socket N**etClient; **DataInputStream** FromClient; **PrintStream** ToClient; **Hashtable** JobIds; **String** Hostname; **InetAddress** Address;
Constructor : public Connection(Socket client, Hashtable job)

Methods :
public void run( );
public Jobs GetJobObject(int job_id);

class Jobs
Data members : **Hashtable** TaskIds **Hashtable** GroupIds **Hashtable** PortIds **int** JobId; **String** JobHostname; **int** JobHostId;
Constructor : public Jobs(int job_id)
Methods : public int addtask(String Hostname, int HostId) public int removetask(String Hostname, int HostId)

public int createport_group(String Hostname, int HostId, String name, int GroupId) public int createport(String Hostname, int HostId, String name, int PortId) public int add_member(String GroupName, int PortNo) public int locateport_group(String name) public int locateport(String name) public Vector show_status(String GroupName) public int deleteport_group(String GroupName) public int deleteport(String PortName) public int remove_member(String GroupName, int PortNo) public String where_to_send(int PortNo)
class Tasks
Data members : **String** TaskName; **int** TaskId;
Constructor : public Tasks(String name, int id)
Class Groups
Data members : **Hashtable** PortIds; **String** Hostname; **int** HostId; **String** GroupName; **int** GroupId;
Constructor : public Groups(String Hostname, int HostId, String name, int gpno)
Methods : public int addportid(String PortName, int PortNo) public int removeportid(String PortName)
class Ports
Data members : **String** PortName; **int** PortId; **String** Hostname; **int** HostId;
Constructor : public Ports(String Hostname, int HostId, String name, int PortNo)

5. JMPF Components Interaction

The JMPF components interaction along with stepwise operations when methods `CreatePortGroup()`, `Send()`, and `Receive()` are invoked is shown in Figure 2, 3, and 4 respectively.

`CreatePortGroup()` Method

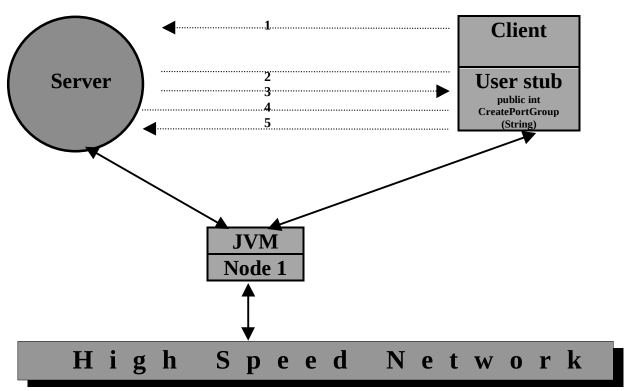

Figure 2: JMPF components interaction when CreatePortGroup() is initiated

- **public int CreatePortGroup (String);**

1. Send the request "LocatePortGroup" to the server to find out if any group already exists with the specified name.

2. Server will process the request.
   if (Group is not existing by the specified name)
      then return -1
   else
      return the GroupId of the existing group by the specified name
3. Send the response to the requester.
4. Proceed according to the received response.
   If(received response > 0) then
      GroupId = -1;
   else
      get free port in the host machine;
      This can be achieved by invoking method ServerSocket(0), which will search for a free port and reserve it as the server socket. The ServerSocket details will be maintained in GroupSocket and GroupIds HashTable objects.
5. Store this GroupName, GroupId information in the server.
6. Return the GroupId to the called program.

`Send()` Method

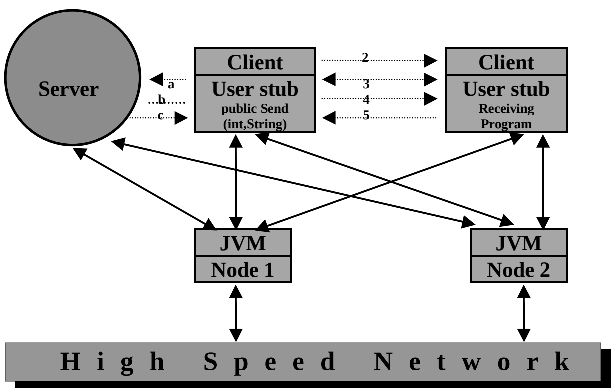

Figure 3: JMPF components interaction when Send() is executed

- **public Send (int, String);**

1. if(the Sender is the owner of the socket) then
      Open ServerSocket connection
   else
   a. Send the request to the server.
   b. Process the response (hostname of the receiver) from the server.
   c. Send the hostname of the receiver to the sender.
2. Open Socket or ServerSocket connection to the receiver.
3. Open the i/p and o/p streams to the receiver.
4. Write the message into the o/p stream of the receiver.
5. Read the status of receiver through the i/p stream from the receiver.
6. Return the status of receiver to the called program.

`Receive()` Method

- **public String BlockReceiveString (int);**
- **public String ReceiveString (int);**

1. if(the Receiver is the owner of the socket) then
      Open ServerSocket connection
   else
   a. Send the request to the server.
   b. Process the request in the server.
   c. Send the hostname of the owner.
2. Open Socket or ServerSocket connection to the sender.
3. Open the i/p and o/p streams to it.
4. Read the message through the i/p stream.
5. Write the status of receiver into the o/p stream of the sender.
6. Return the received message to the called program

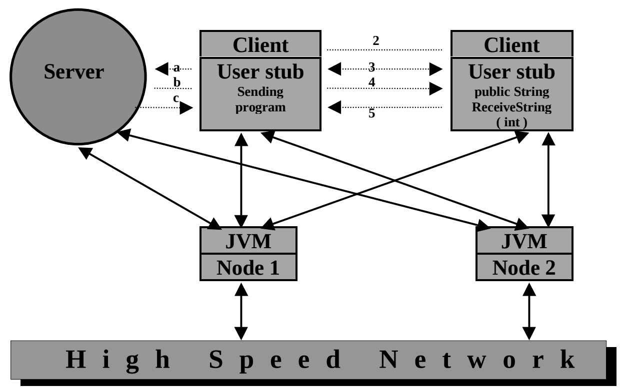

Figure 4: JMPF components interaction when Receive() is executed

Note*: Any one of the Entities involved in point-to-point communication must be the owner of a port/group.*

6. JMPF API

The various methods (application programming interface-API) that the user stub supports along with arguments, usage, functioning, and values it is returning are discussed below.

The methods of JMPF have to be accessed through an instance of the JMPF class. The JMPF class can be instantiated by supplying job and task identification codes as follows:

JMPF Obj = new JMPF(job_id, task_id);

public int	CreatePortGroup (String)
Usage	int GroupId = Obj.CreatePortGroup (String GroupName)
Function	Creates the group specified by the GroupName
Returning values	GroupId < 0 if group is not created GroupId > 0 if group is successfully created.

public int	LocatePortGroup (String)
Usage	int GroupId = Obj.LocatePortGroup (String GroupName)
Function	Locates the group specified by the GroupName
Returning values	GroupId < 0 if group is not located GroupId > 0 if group is successfully located.

public int	LocatePortGroup (String, boolean)
Usage	int GroupId = Obj.LocatePortGroup (String GroupName, boolean Flag)
Function	Locates group by the given name and value of flag. If Flag == true then Wait until the specified group is located successfully. Else Send the status of location
Returning values	GroupId < 0 if group is not located GroupId > 0 if group is successfully located.

public int	DeletePortGroup (String)
Usage	int GroupId = Obj.DeletePortGroup (String GroupName)
Function	Deletes the group specified by the GroupName
Returning values	GroupId < 0 if group is not deleted GroupId > 0 if group is successfully deleted.

public int	CreatePort (String)
Usage	int PortId = Obj.CreatePort (String PortName)
Function	Creates the port specified by the PortName
Returning values	PortId < 0 if port is not created PortId > 0 if port is successfully created.

public int	LocatePort (String)
Usage	int PortId = Obj.LocatePort (String PortName)
Function	Locates the port specified by the PortName
Returning values	PortId < 0 if port is not located PortId > 0 if port is successfully located.

public int	LocatePort (String, boolean)
Usage	int PortId=Obj.LocatePort(String portname, boolean Flag)
Function	Locates port by the given name and the value of flag. If Flag == true then Wait until the specified port is located successfully. Else Send the status of location.
Returning values	PortId < 0 if port is not located PortId > 0 if port is successfully located.

public int	DeletePort (String)
Usage	int PortId = Obj.DeletePort(String PortName)
Function	Deletes the port specified by the PortName
Returning values	PortId < 0 if port is not deleted PortId > 0 if port is successfully deleted.

public int	AddMember (String, int)
Usage	int Status = Obj.AddMember(String GroupName, int PortId)
Function	Adds member port to the specified group
Returning values	Status < 0 if port is not successfully added Status > 0 if port is successfully added to the group

public int	RemoveMember (String, int)
Usage	int Status = Obj.RemoveMember(String GroupName, int PortId)
Function	Removes member port from the specified group
Returning values	Status < 0 if port is not successfully removed Status > 0 if port is successfully removed from group

public	Vector ShowStatus (String)
Usage	Vector vector = Obj.ShowStatus (String GroupName)
Function	This function will accept GroupName as input argument and returns the list of members of that group.
Returning value	Vector the list of members of the specified group.

public	PrintStream CreatePS (int)
Usage	PrintStream PS = Obj.CreatePS (int PortId)
Function	Creates PrintStream to the specified port.
Returning value	PrintStream Makes connection to a receiver and returns PrintStream to the program.

public	DataInputStream CreateDIS (int)
Usage	DataInputStream DIS = Obj.CreateDIS (int PortId)
Function	Creates DataInputStream to the given port.
Returning value	DataInputStream Makes connection to the sender and return DataInputStream.

public int	SyncsendGroup (String, String)
Usage	int Status = Obj.SyncsendGroup (String GroupName, String Message)

public int	SyncsendGroup (String, int)
Usage	int Status = Obj.SyncsendGroup (String GroupName, int Message)
public int	**SyncsendGroup (String, char [ ])**
Usage	int Status = Obj.SyncsendGroup (String GroupName, char [ ] Message)
Function	This function will accept GroupName and message as input argument and returns the status of sending.
Returning values	Status = -1 if receiver hasn't received the message properly. Status = 1 if receiver has received the message properly.

public int	Send (int, String)
Usage	int Status = Obj.Send (int PortNo, String Message)

public int	Send (int, int)
Usage	int Status = Obj.Send (int PortNo, int Message)
public int	**Send (int, char [])**
Usage	int Status = Obj.Send (int PortNo, char[ ] Message)
Function	This function will accept PortNo and message as input argument and returns the status of sending.
Returning values	Status = -1 if receiver hasn't received the message properly. Status = 1 if receiver has received the message properly.

public int	Send (PrintStream, String)
Usage	int Status = Obj.Send (PrintStream ps, String Message)
public int	**Send (PrintStream, int)**
Usage	int Status = Obj.Send (PrintStream ps, int Message)
public int	**Send (PrintStream, char [])**
Usage	int Status = Obj.Send (PrintStream ps, char [ ] Message)
Function	This function will accept PrintStream, which is already made The connection and opened the PrintStream for writing into it and Message as input arguments and returns the status of sending.
Returning values	Status = -1 if receiver hasn't received the message properly. Status = 1 if receiver has received the message properly.

public	String BlockReceiveString (int)
Usage	String Message = Obj.BlockReceiveString (int PortNo)
public	**int BlockReceiveInt (int)**
Usage	int Message = Obj.BlockReceiveInt (int PortNo)
public	**Char[] BlockReceiveCharArray (int)**
Usage	char message[ ] = Obj.BlockReceiveCharArray (int PortNo)
Function	This function will accept PortNo as input argument and returns the received message of type String.
Returning values	Null if receiver has not received message properly message if receiver has received the message properly

public	String ReceiveString (int)
Usage	String Message = Obj.Receive (int PortNo)
public	**int ReceiveInt (int)**
Usage	int Message = Obj.Receive (int PortNo)
public	**Char[] ReceiveCharArray (int)**
Usage	char Message[ ] = Obj.Receive (int PortNo)
Function	This function will accept PortNo as input argument and returns the received message.
Returning values	Null if receiver has not received the message properly. Message if receiver has received the message properly.

public	int ReceiveInt (DataInputStream)
Usage	int message = Obj.ReceiveInt (DataInputStream dis)
public	**String ReceiveString (DataInputStream)**
Usage	String message = Obj.ReceiveString(DataInputStream dis)
public	**Char [] ReceiveCharArray(DataInputStream)**
Usage	char message[ ] = Obj.ReceiveCharArray (DataInput-

	Stream dis)
Function	This function will accept DataInputStream, which has already made the connection and opened the DataInputStream for reading from it as input arguments and returns the received message.
Returning values	Null if receiver has not received the message properly. message if receiver has received the message properly.

public	Void ClosePorts( )
Usage	Obj.ClosePorts( )
Function	This function will close all the groups and ports that the current task has created and remove this task from JMPFServer.

7. Sample Applications of JMPF

In this section, we discuss four sample applications that demonstrate the usability and suitability of JMPF for cluster computing. The first application involves implementation of Fast Fourier Transformation (FFT) algorithm. This implementation is useful in Polynomial Arithmetic. The second application involves implementation of sorting. The third application involves distributed matrix multiplication. The fourth application involves solving N queens problem and showing its output in GUI using Applets.

All the four implementations use the Master-Worker model [5, 6]. The master will communicate with the workers and distribute the data and the intermediate results to the workers. The workers in turn receive the data from the master, perform the required computation and send the results back to the master.

FFT Algorithm

FFT is one of the oldest and most useful algorithms used extensively in a wide variety of applications. Hence, we selected this as one of the potential candidate for parallelisation.

FFT of an N vector X is a linear transformation of X defined by Y = F*X, where F is an N X N matrix with i, j entry is (w)^i.j for 0 <= i, j < N and w is defined as the Nth primitive root of unity. An N point FFT can be computed in log(N) steps.

The following are the steps involved in the implementation:

[a] Each x^i is input into a node on $(\log N)^{th}$ level by employing decimation technique. The decimation technique involves entering the input x^i into the row numbered by the reversal of i, reverse (i).

[b] At each node (a, j), multiply the input from higher numbered row (row with a 1 in j+1 position) by w^(i*2^j) and adds the result to the input from lower numbered row, where i is the integer consisting of logN-j least significant bits of a. Here a is the row number in the butterfly network. These values are computed during step logN - j and are then output the level j - 1 node. level 0 nodes are the final output nodes.

The FFT algorithm is useful in convolution computation and polynomial arithmetic. It is also useful in integer multiplication of numbers of large size.

The implementation of this algorithm uses the functions available in the communication class of the framework. The master and the workers communicate through the message-passing interface. This interface called the server, creates and manages the communication ports for the master and the worker. The speed of computation can be made very fast provided the network speed is high.

Sorting Algorithm

Sorting of data is one of the most vital applications in databases. Herein, a large pool of data usually stored in files in flat-file structure or relational database. The current implementation accepts two data files as input, sorts each file simultaneously on a different computer system in the cluster and finally merges the two sorted data files. It can easily be extended to a single large file whose data can be broken up into a set of smaller and easily sortable modules, which can then be merged. Thus effectively reducing the time required to sort a large file into the time required to sort a small data module and the merging process. The overall time saved raises exponentially with size of data to be sorted.

Matrix multiplication

When we closely observe matrix multiplication operation, C = A * B, it can be noticed that every element of resultant matrix C, can be computed independently. It can also be observed that each row of matrix C is independent of other rows and hence, we can simultaneously compute each row elements by distributing matrix A and B elements to different nodes in the network and then gathering the computed results to form resultant matrix C.

To implement this application we used master worker model of computation. In this we accept the square matrices A and B and distribute each row in matrix A to each nodes using the method Send(int PortNo, String message) and whole matrix B to all the nodes working for this job, using method Syncsend-Group(String GroupName, String message) in the user stub. Then each worker programs working in each node will compute each row in matrix C simultaneously and send the results back to the master program. Master program will collect all the results from all the worker nodes and formulates the resultant matrix C.

N Queens Problem

The requirement of N queens problem is, to place the N queens in a N * N chessboard in such a way that no two queens are in the same row or column and they should not be next to each other diagonally.

We can place N number of queens in N! number of ways. The first queen can be placed in N number of columns in (N-1)! ways. Then we will be having (N-1) ways to place the second queen in second row in (N-2)! ways. Depending on the position of the first queen other queens can be placed. The combinations of the first queen placed in one column are independent of the combinations of that queen placed in another column. So each combination of placing the first queen in each column can be calculated independently, which means that, the work can be distributed across nodes of a cluster computer and executed in parallel.

This application is built using Master-Worker style of programming. The Master program will accept the number of queens and calculate all the combinations of placing the queens on the chessboard. Each combination of placing the first queen in each column has been sent to each worker, which will be running on the clustered computers. These workers will find the combinations, which will satisfy the required criteria and send back the results to the Master program.

8. Conclusions and Future Directions

We have presented an overview of JMPF along with the detailed design of JMPF stub and JMPF Server. JMPF that has been designed for cluster computing is tested on multiple machines and for different applications. It allows simultaneous execution of multiple instances of the same application. As JMPF adheres to the pure Java standard, the applications developed using JMPF services are portable as normal Java programs.

We planned to extend JMPF to support features such as multicast, scatter, gather, etc. and develop more scientific applications to demonstrate its suitability for cluster computing.

9. Availability

The JMPF is available in the public domain by remote FTP access. Access the site, *http://www.fit.qut.edu.au/~rajkumar/jmpf.html*, which displays the abstract of this paper. Click on Download JMPF for downloading this message passing system. The JMPF source files are stored in the file, *jmpf.tar*, by using the *tar* utility. Suggestions for further improvements of this work are welcome and they can be mailed to the authors.

Acknowledgments

The authors are grateful to all the members of C-DAC, Bangalore, and in particular to all the members of Operating System Group. We thank Ramesh Naidu V, Mohammed Safir, TSR Mohan, and Satish B Hampali, M. Tech. Computer Science and Engineering students, Karnataka Regional Engineering College (KREC), Surathkal, Karnataka, India for using JMPF to build sample applications discussed in this paper. We thank S Y Chan, Rosziati, Nishanth, and Kapaleeswaran of Queensland University of Technology for proof reading drafts of this paper.

References

[1] Achutha Raman, Rajkumar, et. al., *PARDISC: A Cost Effective Model for Parallel and Distributed Computing*, Proceedings of the 3rd International Conference on High Performance Computing, IEEE Press, 1996.

[2] Rajkumar, *Single System Image: Need, Approaches, and Supporting HPC Systems*. Proceedings of the Fourth International Conference on Parallel and Distributed Processing, Techniques and Applications (PDPTA'97), CSREA Publishers, Las Vegas, USA, 1997.

[3] Sun Microsystems, *Java Language - A White Paper*, Sun Microsystems Computer Company, 1996.

[4] Patrick Naughton and Herbert Schildt, *JAVA: The Complete Reference*, Mc-Graw Hill Inc., 1997.

[5] F Thomson Leighton., *Introduction to Parallel Algorithms and Architectures. Trees. Arrays*. Hypercubes. Morgan Kaufmann Publishers, 1992.

[6] Angus, G Fox, J Kim, and D Walker, *Solving Problems on Concurrent Processors. Volume II - Software for Concurrent Processors*. Prentice Hall Inc, 1990.

[7] Xavier Defago (EPFL) and Akihiko Konagaya. *Issues in Building a Parallel Java Virtual Machine on Cenju-3/DE.* NEC Cenju Workshop and Cenju-4, HPC Asia, Seoul, Korea, 1997.