Mémoire de DEA
Université Louis-Pasteur de Strasbourg

P2P-MPI : A Peer-to-Peer Framework for Robust Execution of Message Passing Parallel Programs on Grids

Choopan Rattanapoka

DEA Informatique 2003/2004
Image et Calcul Parallèl Scientifique

Stage encradé par Stéphane Genaud

Contents

1 Introduction **5**

2 State of the Art **9**

2.1 Fault Tolerant Mechanisms 10

2.1.1 Rollback-Recovery Techniques 10

2.1.2 Replication Techniques 12

2.2 Fault Detectors 13

2.2.1 Push Model 14

2.2.2 Pull Model 14

2.2.3 Gossip-style Protocol 14

2.3 Enhancing MPI Projects 15

2.3.1 Fault Tolerant MPI 15

2.3.2 Heterogeneous MPI 17

2.3.3 Self-configuration MPI 17

2.4 Peer-to-Peer Topologies 17

2.4.1 Centralized Topology 18

2.4.2 Decentralized Topology 18

2.4.3 Hybrid Topology 18

2.5 JXTA 19

3 P2P-MPI Platform Proposal **21**

3.1 Objectives 21

3.2 Definitions 21

3.3 P2P-MPI Architecture 22

3.3.1 Overview 22

3.3.2 P2P-MPI Platform Startup 23

3.3.3 Nodes Finding to Run an MPI Application 24

3.3.4 Process Replication 25

3.3.5 Fault Detector in P2P-MPI 26

3.3.6 Message Buffering Table 27

3.3.7 Sending Message Agreement Protocol 27

3.4 P2P-MPI in Programmers' View 31

3.5 Experiments 33

3.5.1 Time for Joining Groups 34
3.5.2 Time for Creating a MPI Platform 35
3.5.3 The Impact of Replicas on Sending a Message 36

4 Conclusion 39

A Algorithms 41
A.1 Message Sending Management 42
A.2 Fault Detection Notification 43
A.3 Internal Message Management 43

B P2P-MPI APIs 45
B.1 MPI Package 45
B.1.1 MPI Datatypes 45
B.1.2 MPI Operations 46
B.1.3 Startup and Timers 46
B.2 Groups Package 47
B.3 Comm Package 47
B.4 IntraComm Package 48

Chapter 1

Introduction

The concept of *Grid* has recently emerged to express the possibilities that networking technologies let encompass in terms of computer usage. An overview of these possibilities and problems to overcome is given in [1]. *Grid computing* offers a model for solving massive computational problems using a large number of computers arranged as clusters embedded in a distributed telecommunication infrastructure. Grid computing involves sharing heterogeneous resources (based on different platforms, hardware/software architectures) located in different places belonging to different administrative domains over a network using open standards. Desktop grid computing is a particular type of Grid computing. It exploits unused resources in the Intranet environments and across the Internet. We often see the words of campus desktop Grid and company desktop Grid.

In the parallel computing domain, we can see Grids and desktop Grids as a large-scale computer cluster, provided some adequate middleware is installed and configured to form an extremely powerful computer. However, the question of how we may program such a cluster of heterogeneous computing resources remains. Most of the numerous difficulties that people are trying to overcome today in order to develop Grid applications, fall in two categories.

- The middleware deployment is not as straight-forward as it should be. It is indeed difficult to install and maintains hundreds or thousands of nodes across several organizations. The most advanced Grid software today (in terms of developpment effort) is Globus [2] but lacks fault diagnostics and auto-repair mechanisms whereas software complexity is high.

- The other major issue is concerned with the programming model. Many projects have focused on a client/server programming style (also called *Remote Procedure Call* (RPC)). Projects like JNGI [3], DIET [4], XtremWeb [5] offer such a programming model. However, when dealing with high performance computing, programmers should be able to follow a programming model more adapted to the many computing nodes available. The *message*

passing model and the *data parallel* programming model are the two models traditionnally used by parallel programmers.

One implementation of the message passing model is MPI. The Message Passing Interface (MPI) [6] is currently the *de-facto* standard system used to build high performance application for clusters and dedicated MPP systems. However, the current trend in high performance computing is the use of large scale computing infrastructure not only for clusters but also for Grids. MPI was initially designed to allow a very high efficiency for applications using it, by using a *static process model*, that is the number of processes used for an application remains the same from the beginning until the end of the application. This constraint makes MPI applications unadapted to run on Grids because failure of nodes are somehow frequent in this context.

Another drawback of the standard MPI is the difference in the operating systems and the dependences on the sharing file system. In practise, the use of MPI applications is often based on a computer cluster using *NFS* (Network File System) to ease the deployment of files. When the set of computers is composed of more than one operating system (imagine the campus desktop Grid which has Windows and Linux PCs), we have to compile two versions of executable file (one for Windows and one for Linux) and then we have to copy each version of the executable file to the proper machine manually.

So our new approach in designing MPI for current trend in high performance applications should meet the following three properties.

Heterogeneity. To form a Grid or desktop Grid, we cannot guarantee the homogeneity of resources such as operating systems. Our software should provide some facilities for programmers to deploy their applications in heterogeneous systems.

Fault tolerance. When the size of the Grid becomes significant, the mean time between failure (MTBF) of CPU nodes becomes a serious limiting factor. Our software should provide automatic and transparent mechanisms to handle nodes failures.

Self-configuration and autonomy. As the number of nodes in a Grid gets bigger, the difficulty for setting up a coherent platform is also higher. We need something that gives the platform self-configuration and autonomy properties. It means that as soon as a node is online, it should automatically register into the platform and declare itself ready to run a task.

The idea that we propose is related to the *peer-to-peer* model we will discuss in detail later in chapter 2, page 17. These last years, many projects in the field of distributed systems have been based on the peer-to-peer model, especially for file sharing. They proved to be reliable and efficient enough from the user

point of view if we consider their popularity. We think this model has interesting properties that could serve as a basis for fault-tolerance, self-configuration and autonomy. Our work aims to propose a middleware infrastructure able to support the execution of parallel programs using a message passing programming model and whose features meet our three properties listed above. We call this infrastructure a *platform*. This platform is designed to support a subset of the MPI standard (the minium set required to program with the message passing paradigm).

Chapter 2

State of the Art

The MPI is a *de-facto* standard system used to build high performance applications for clusters and dedicated MPP systems. The MPI standard has been initially designed for high performance and one key design feature is that any MPI application relies on a static process model[1]. More precisely, the static process model implies that during one MPI application run, MPI creates and manage a communication table called *communicator*, for each MPI process to know how to contact each other. So when one of the computing node fails, there will be a hole in the communicator that causes the whole application to fail. Currently, the trend is to exploit platforms with more and more nodes so we cannot ignore the problem of node failures and the heterogeneity of systems.

A lot of research work has been devoted to fault tolerance in various contexts. We review in this chapter the main streams developped for fault tolerance, and we focus on efforts made to integrate fault tolerance into MPI.

The heterogeneity of operating system is also a challenge for executing parallel applications. To compile source codes to an executable file for all operating systems seems to be a tedious tasks for programmers. One solution may be to use *byte code* represented applications that would abstract the application from the low system layer. The most popular product based on byte code comes from SUN who designed the *Java* language. Java allows indeed to create platform independant applications. First, the java compiler compiles java source codes into byte code programs. Then, we use a java interpreter (also well-known under the term java virtual machine (JVM)) to execute an application from the byte code. The Java's byte code has a stardard format thus it can be executed for all platforms that have a java interpreter.

An important point in platforms such as Grids are their dynamicity: the CPU occupation of each node varies continously, the bandwidth between network links is also constantly changing, the software on nodes changes regularly and in the

[1]The forthcoming MPI-2 implementations will offer the `MPI_Spawn` primitive to bypass the static model.

worst case (desktop grids) nodes can join and leave at anytime. It is hard for programmers to handle this situation themselves. We should provide a middleware which keeps the dynamicity of nodes in Grid transparent to programmers by providing some mechanism that would dynamically request available nodes for a computation. Currently, the peer-to-peer model has proved to be good in harness environments, as demonstrated by the success of peer-to-peer file sharing applications. Thus, we propose to deploy a peer-to-peer model in a middleware to run in Grid.

We should therefore review existing research work done in several fields. Fault tolerance has been studied for a long time in distributed systems and we do need to introduce fault tolerance in our project. Many projects have tried to adapt MPI to more versatile environments than the traditional parallel computers and we review the efforts made in this domain. Last, we give a quick overview at peer-to-peer systems and we introduce some concepts developped in the JXTA protocols that we will use for our project.

2.1 Fault Tolerant Mechanisms

In our case, the MPI applications which are executed on Grid should be fault tolerant. There are two kinds of techniques that provide fault tolerance. One is *replication* and the other is *rollback-recovery*.

2.1.1 Rollback-Recovery Techniques

Two principal techniques have been proposed for rollback-recovery protocols: *global checkpoint* or *message log*.

The global checkpoint consists in taking a snapshot of the entire system state regularly without the assumption of a global clock, but by using the concept of logical clock introduced by [7]. So, when a failure occurs on any process, the whole system can roll back to the latest checkpointing image and continue the computation.

In message log protocol, all processes can checkpoint without begin coordinated. A process execution is supposed to be piecewise deterministic, which means it is governed by its message receptions. Thus all communications are logged in a stable media so that only the crashed processes rollback to a precedent local snapshot and execute the same computation as in the initial execution, receiving the same messages from the stable storage.

In the next section, we describe the global checkpoint and message log approaches.

Global Checkpoint Based

There are three classes of global checkpoint protocols [8]: *uncoordinated*, *coordinated checkpoint* and *communication induced.*

- In uncoordinated checkpoint without message log, the checkpoints of each process are executed independently of the other processes and no further information is stored on a reliable media leading to the well known domino effect (processes may be forced to rollback up to the beginning of the execution). Since the cost of a fault is not known and there is a chance for losing the whole execution, these protocol are not used in real application.

- In coordinated checkpoint, all processes coordinate their checkpoints so that the global system state composed of the set of all process checkpoints, is coherent. The drawback of this mechanism is the performance, the processes need to wait for a synchronization of the checkpoint and when a failure is detected the whole application needs to be restarted from the previous image.

- Communication Induced Checkpointing (CIC) tries to take advantage of uncoordinated and coordinated checkpoint techniques. Based on the uncoordinated approach, it piggy backs causality dependencies in all messages and detects risks of inconsistent state. When such a risk is detected, some processes are forced to checkpoint. While this approach is very appealing theoretically, relaxing the necessity of global coordination, it turns out to be inefficient in practice [9]. The two main drawbacks in the context of cluster computing is (1) CIC protocols do not scale well (the number of forced checkpoints increases linearly with the number of processes) and (2) the storage requirement and usage frequency are unpredictable and may lead to checkpoint as frequently as the coordinated checkpoint technique.

Message Log based

There are three classes of message log protocols: *pessimistic*, *optimistic* and *causal* which are discussed in [10].

- Pessimistic log protocols ensure that all messages received by a process are first logged by this process before it causally influences the rest of the system. MPICH-V (see section 2.3.1) is based on this type of protocol. It uses reliable processes called Channel Memories. Every MPI computing is connected to a channel memory. When a node sends a message, it sends it to the channel memory of the receiver and when it wants to receive a message, it asks its own memory channel for it.

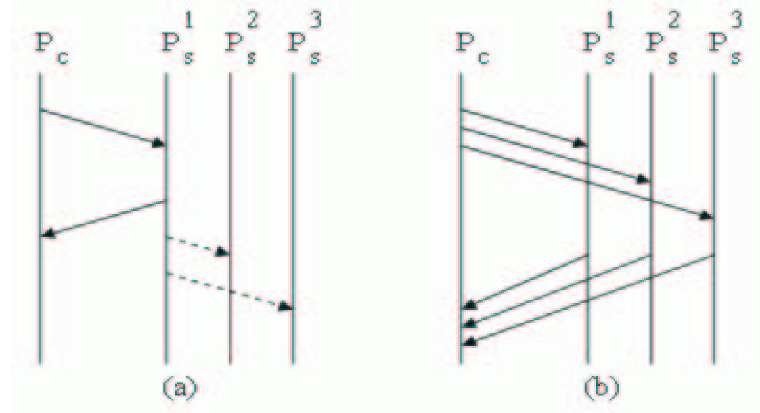


Figure 2.1: Passive and active replication.

- The optimistic log protocols [11] eventually log receptions but do not wait for them before sending new messages. Therefore they are faster in non-faulty executions but do not exclude to rollback some non-crashed processes if a fault occurs before the reception logging.

- Causal log protocols try to conceal the optimistic and the pessimistic approaches. When a process sends a message, it logs it locally and appends information about its past receptions. Thus when a process crashes, it can either retrieve information about its initial execution's receptions or no process depends on its precedent computation.

2.1.2 Replication Techniques

In the replication technique, a process is replicated and we called the replicated process *replica*. These replicas are placed on different computers. Even if some of the replicas fail, the others continue to process the application. There are two kinds of replication techniques, one is the *active replication* [12] and the other one is *passive replication* [13].

For the sake of clarity, we will assume the replication of server processes and a client process. Let us assume P_c is a client process which sends a message to a server process P_s. For fault-tolerance in a server process, we use replication and provide $P_s^1, P_s^2, ..., P_s^n$ processes where n is the number of replicas of P_s. We say a replica is *operational* if it is not idle and executes its code. Otherwise, a replica is idle waiting to be woken up.

Passive Replication

In the passive replication, only one replica is operational. In figure 2.1 (a), a client P_c only sends a message to the replica master, for example P_s^1. Then only

P_s^1 performs a requested operation and returns the result. The other replicas P_s^i $(2 \leq i \leq n)$ are not operational. The state of the passive replicas are updated by receiving the newest state information from the operational replica P_s^1 from time to time. This is called a *checkpoint.* If P_s^1 fails, one of the passive replicas, say P_s^2 takes over it. In order to catch up with the failed P_s^1, P_s^2 starts to execute the application from the most recent checkpoint. However, the recovery procedure takes time since P_s^2 becomes operational at the checkpoint and re-execute the operations that P_s^1 had already done.

Active Replication

In this scheme, all the replicas are operational (see figure 2.1(b)). A client P_c sends messages to all the replicas P_s^i $(1 \leq i \leq n)$. Each replica performs the requested operation and returns the result. Since all the replicas are operational, even if a certain replica $P_s^{i'}$ fails, the other replicas P_s^i $(i \neq i')$ can continue to execute the application without the suspension time. Hence, the recovery procedure in the active replication requires less overhead than that in the passive one. However, this technique needs more resources than the previous one.

2.2 Fault Detectors

Failure detection services have received much attention in the literature and many protocols for failure detection have been proposed and implemented. Many implementations of failure detection services have been proposed and are efficient for local area networks. However they do not perform well in the context of a large scale distributed system.

The implementation of failure detection protocols are based on timeouts. There are two basic models of fault detector which are discussed in [14]. One is the *push model* and the other is the *pull model.* However, these two models are designed for small-scale systems. For the large-scale system, there is a research on *the gossip-style fault detection service* [15].

The main principles that should be considered when implementing the failure detectors for MPI in a Grid system are the following.

Message explosion Despite the large number of components that need to be monitored and their distribution in the system, the failure detector must prevent flooding or overloading the network with failure detection related messages.

Scalability MPI Applications running on a Grid system may require a large number of resources distributed over a wide area network. A failure detection service must be able to efficiently monitor such large number of

resources. It must be able to quickly detect failure while minimizing the number of wrong suspicions.

2.2.1 Push Model

In this model, monitored components are active and the monitor (failure detector) is passive. Each monitored component periodically sends messages (heartbeat messages) to the failure detector which is monitoring the component. A failure detector suspects a component failure, which means crashed component, when it fails to receive a heartbeat message from the component within a certain time interval T (timeout).

In the push model, the monitor suspects the failure of a component in the system after a certain time interval T. However, there is a large number of messages sent on the network. If there is a large number of monitored components, the heartbeat messages can flood the network (problem of the message explosion).

2.2.2 Pull Model

In this model, monitored components are passive while the monitor or failure detector is active. The monitors sends liveness requests ("*Are you alive?*" messages) periodically to monitored components. Upon reception of a liveness request, the monitored component sends a reply to the monitor. When the monitor does not receive a reply from a monitored component within a certain time interval (timeout), it suspects the failure of the monitored component.

In the pull model the load on the network is reduced and depends on the number of liveness requests sent by the monitor. However, the monitor can not suspect or detect the failure of a component until after it sends it a liveness request.

2.2.3 Gossip-style Protocol

In this model, a failure detector is not centralized but distributed as a module and resides at each host on the network. It maintains a table with an entry for each failure detector module known to it. This entry includes a counter called heartbeat counter that will be used for failure detection. Each failure detector module picks another failure detection module randomly (without concern to the network topology) and sends it its table after incrementing its heartbeat counter. The receiving failure detector module will merge its local table with the received table and adopts the maximum heartbeat counter for each member. If a heartbeat counter for a host member A which is maintained at a failure detector at another host B has not increased after a certain timeout, host B suspects that host A has crashed.

Gossip-style protocol is quite simple and can address the problem of message explosion. The number of messages is reduced even if this protocol is used in distributed systems with a large scale network. However, the drawback of this protocol is it does not work well when a large percentage of components crash or become partitioned away. Then, a failure detector may spend a long time to detect crashed components by gossip message.

2.3 Enhancing MPI Projects

2.3.1 Fault Tolerant MPI

CoCheck [16] is one of the earliest efforts to make MPI more reliable. CoCheck extends the single process checkpoint mechanism used in Condor[17] to a distributed message passing application. Common problems with checkpointing and recovery such as global inconsistent states and domino effects are eliminated through the use of a protocol to flush all in-transit messages before a checkpoint is created. In consequence, CoCheck faces the problem of a large overhead because it checkpoints the entire process state.

Starfish [18] provides a parallel execution environment that adapts to changes in the cluster caused by node failure and recovery. The Starfish environment for execution of dynamic MPI programs is based on the Ensemble group communication system. Starfish uses an event model wherein application process register to listen for events reflecting changes in cluster configuration and process failures. Starfish also provides application- and system-driven checkpointing facilities. When a process failure is detected, Starfish can automatically recover the application from a previous checkpoint. However, consistency of communicators is not addressed in Starfish; in order to recover a single failed process, the entire MPI application must be restarted. Essentially, many of the powerful dynamic process management features of Starfish cannot be used directly by MPI applications.

MPI-FT [19] uses the approach which is similar to the one proposed for Real-time data-driven systems. The existence of a monitoring process,the Observer who will notify the rest of the processes in the event of a failure, and the action to be executed for recovery. Two different modes are proposed. In the first one, each process is responsible for buffering all message traffic it sends out while in the second case, all message traffic is buffered by Observer. Checkpoints are inserted explicitly in the code which are actually tests for the arrival of the failure message which is received asynchronously by a non-blocking receive. The failure message is sent by the Observer to the alive peers to invoke the recovery routine. MPI-FT solves the MPI problem of the dead communicator which refers to the fact that there us

a death of a process by proposing two different solutions, either the preparation of spawning communicators in advance (one extra communicator to exclude each potential hole) or the pre-spawning of the replacement process when the program starts executing. However, the drawback of this system is the memory needed for the observer process in long running applications

FT-MPI [20] handles failures at the MPI communicator level and lets the application manage the recovery. When a fault occurs, all MPI processes of the communicator are informed about the fault. This information is transmitted to the application through the returning value of MPI calls. The main advantage of FT-MPI is its performance since it does not checkpoint nor log, but its main drawback is the lack of transparency for the programmer.

MPICH-V [21] is a mix of uncoordinated checkpointing and a pessimistic message logging protocol storing all communications of the system on a reliable media. To ensure this property, every computing processes is associated with a reliable process called Channel Memory. Every communication sent to a process is stored and ordered on its associated Channel Memory. To receive a message, a process sends a request to its associated Channel Memory. After a crash, a re-executing process retrieves all lost receptions in the correct order by requesting them to its Channel Memory. The use of Channel Memory however has a major impact on the performance (dividing the bandwidth by a factor of 2) and the cost of the fault tolerance system (high performance requires a large number of Channel Memories).

MPICH-V2 [22] is an improved version of MPICH-V designed to overcome the major impact on the performance of using Channel Memory. In MPICH-V2, the message logging is split into two parts: on one hand, the message data is stored on the computing node, following a sender-based approach. On the other hand, the corresponding event (the date and the identifier of the message reception) is stored on an event logger which is located on a reliable machine. However, MPICH-V2 still needs reliable nodes for the fault tolerant system.

MPI/FT [23] is the closest project to our proposal which provides fault-tolerance to MPI by introducing the MPI processes replication. Using these techniques, the library can detect erroneous messages by introducting a vote algorithm among the replicas and can survive process-failures. The drawback of this project is the increasing resource requirement by using replicating MPI processes but this drawback can be overcome by using large platforms such as Grid or desktop Grid.

2.3.2 Heterogeneous MPI

Java is receiving increasing attention as one of the most popular platform for distributed and collaborative computing. One of its key feature is that Java's bytecode can run on several platforms without re-compiling. Our first property of the new approach on MPI, which involves heterogeneous systems, can be acquired by implementing Java for MPI. There are several research works that have lead to MPI implementations for Java.

Java-MPI [24] is an java interface to standard MPI by using JNI wrappers to native MPI package. In Java-MPI, Java wrappers are automatically generated from the C MPI headers. This eases the implementation work, but does not lead to a fully object-oriented API.

MpiJava [25] is an object-oriented Java interface to standard MPI. MpiJava provides the full functionality of MPI 1.1. It is implemented as a set of JNI wrappers to native MPI packages.

However, these implementations bind the Java interface to the standard MPI which does not handle the fault tolerance. JNI Wrapper to native MPI packages can cause an overhead and makes the performance drop. The full implementation of MPI in Java language instead of using JNI wrappers seems to perform better.

2.3.3 Self-configuration MPI

In our knowledge, there is no direct research on self-configuration that integrates MPI. However, there are some research on middle-ware which provides information to MPI. In consequence, we need to install and configure the middle-ware to handle and provide information to MPI. We propose to integrated self-configuration into MPI platform without the help of other middle-ware to eliminate the installation difficulty. The adoption of a peer-to-peer model helps us to provide self-configuration mechanisms in MPI platform.

2.4 Peer-to-Peer Topologies

Generally, a peer-to-peer (or P2P) computer network refers to any network that does not have fixed clients and servers, but a number of peer nodes that function as both clients and servers to the other nodes on the network. This model of network arrangement is contrasted with the client-server model. Any node is able to initiate or complete any supported transaction. Peer nodes may differ in local configuration, processing speed, network bandwidth, and storage quantity. Popular examples of peer-to-peer are file sharing-networks. The peer-to-peer model consists of several topologies, we will discuss three main topologies of peer-to-peer model below.

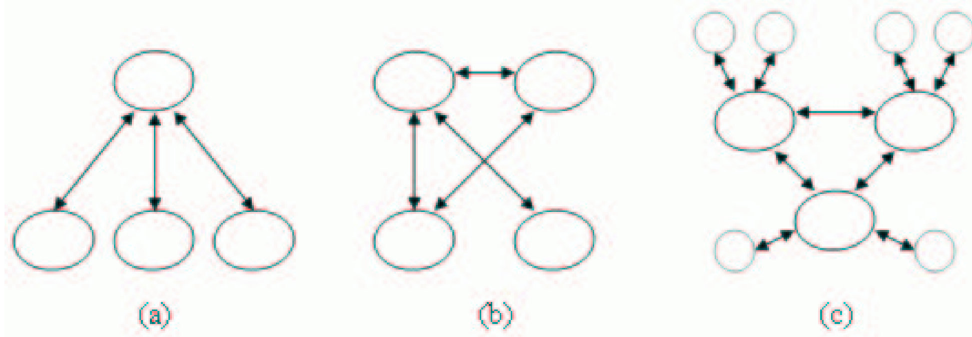


Figure 2.2: Three main peer-to-peer topologies.

2.4.1 Centralized Topology

The centralized systems are the most familiar form of topology, typically seen as the client/server pattern used by databases, web servers, and other simple distributed systems. All function and information is centralized into one server with many clients connecting directly to the server to send and receive information. Many applications called "peer-to-peer" also have a centralized component. SETI@Home is a fully centralized architecture with the job dispatcher as the server. And the original Napster's search architecture[26] was centralized, although the file sharing was not. The advantage of this topology is performance in searching other peers or service that other peers provide because the centralized server maintains all the information. However, the disadvantage is the bottle's neck of the centralized server.

2.4.2 Decentralized Topology

Decentralized systems is where all peers communicate symmetrically and have equal roles. Gnutella [27] is probably the most "pure" decentralized system used in practice today, with only a small centralized function to bootstrap a new host. Many other file-sharing systems are also designed to be decentralized, such as Freenet or OceanStore. Decentralized systems are not new; the Internet routing architecture itself is largely decentralized, with the Border Gateway Protocol used to coordinate the peering links between various autonomous systems. There is no bottle's neck in this topology because there is no special centralized server. However, the performance of this topology suffers when searching other peers or their provided services.

2.4.3 Hybrid Topology

The distributed systems often have a more complex organization than one from a simple topology. Real-world systems often combine several topologies into one

system, making a hybrid topology. Nodes typically play multiple roles in such a system.The hybrid topology overcomes the bottle's neck of centralized topology and performance of decentralized topology. JXTA is an example of the peer-to-peer infrastructure using hybrid topology.

Sun has founded the development of two implementation of JXTA protocols (one in C and one in Java). The java implementation being more up-to-date relatively to the protocol definitions.

2.5 JXTA

JXTA [28] is an open-source framework, which specifies a set of language- and platform-independent XML-based protocols. JXTA provides a rich set of building blocks for the management of peer-to-peer systems: resource discovery, peer group management, peer-to-peer communication, etc.

Peers. The basic entity in JXTA is the *peer*. Peers are organized in networks. They are uniquely identified by IDs. An ID is a logical address independent of the location of the peer in the physical network. JXTA introduces several types of peers. The most relevant as far as we are concerned are the *edge peer* and *rendezvous peers*. Edge peers are able to communicate with other peers in the JXTA virtual network. They can also store advertisements of resources they discover in the network. Rendezvous peers have the extra ability of forwarding the requests they receive to other rendezvous peers. They can also offer a storage area for advertisements that have been published by edge peers.

Peer groups. Peers can be members of one or several *peer groups*. A peer group is made up of several peers that share a common set of interests. The main motivation for creating peer groups is to build services collectively delivered by peer groups, instead of individual peers. Indeed, such services can then tolerate the loss of peers within the group, as its internal management is not visible to the clients.

Pipes. Communication between peers or peer groups within the JXTA virtual network is made by using *pipes*. Pipes are unidirectional, unreliable and asynchronous logical channels. JXTA offers two types of pipes: point-to-point pipes, and propagate pipes. Propagate pipes can be used to build a multicast layer at the virtual level.

Advertisements. Every resource in the JXTA network (peer, peer group, service, etc.) is described and published using *advertisements*. Advertisements are structured XML documents which are published within the network of rendezvous peers. To request a service, a client has first to *discover* a matching advertisement using specific localization protocols.

JXTA protocols. JXTA proposes several protocols such as *Peer Discovery Protocol*, which allows for advertisement publishing and discovery.

The mechanism of JXTA framework is simple. Once a JXTA instance is created, it joins the default peer group called *Net Peer Group* by connecting via the default rendezvous peers. The implementations from SUN proposes a default list of computers (owned by SUN) hosting some rendezvous peers to bootstrap a Net Peer Group. Every request for searching the JXTA advertisements operates by sending a request message to its connected rendezvous peer. If this rendezvous peer does not find the advertisment locally, it may forward the request to other rendezvous peers which may return the result to the requester.

Chapter 3

P2P-MPI Platform Proposal

3.1 Objectives

The main objective of P2P-MPI platform is to provide a robust infrastructure for MPI applications in the harness environment such as Grid and desktop Grid. P2P-MPI is based on a peer-to-peer model to provide the self-configuration and autonomy properties. P2P-MPI is programmed in pure Java and, in our first prototype, provides some necessary MPI application programming interfaces (API) for programmer in order to make their MPI applications able to run on several computing platforms. Thus, we meet the heterogeneous property requirement. For the robustness, we choose to bring fault tolerance to MPI with a process replication. With the process replication technique, we can avoid the overhead issue of using checkpoint technique. Though the process replication technique requires indeed many available nodes, we consider this limitation not as a strong limitation because we target large-scale Grids.

3.2 Definitions

Our work considers any parallel program composed of communicating processeses running concurrently. However, following the practises of MPI programmers we focus in the discussion below on the SPMD (Single Program Multiple Data) programming model, i.e. all processes execute a similar algorithm parameterized by their process number. In order to implement fault tolerance, we need to speak of processes at two different level. We introduce the notion of *abstract processes*, which are processes from a logical point of view, and the notion of *replicas* to speak about real processes.

Definition 1 Parallel program
A parallel program is a set of n abstract processes *noted* $\{P_1, P_2, .., P_n\}$.

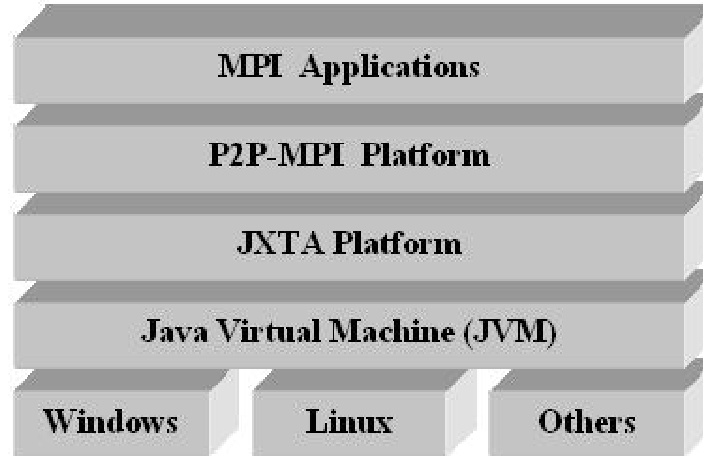


Figure 3.1: Overview of P2P-MPI platform layer.

In the rest of this document, we will simply use the term *processes* instead of *abstract processes* when we speak from the user point of view. The term *abstract processes* is introduced to distinguish between real processes that implement the replication of processes and the logical processes visible to the user.
The processes of a standard MPI application should be seen as abstract processes.

Definition 2 Replicas
Replicas *are real processes, cooperating in their communication scheme so as to emulate the behavior of an abstract process. An abstract process* P_i *is implemented by a set of* replicas, *noted* $\{P_i^1, P_i^2, .., P_i^r\}$.

The code of each abstract process is redunded by several processes forming a set of replicas. These processes have all the same functional code and are coordinated in their communications to other abstract processes. Replicas being the actual processes implementing the abstract process, there is at least one replica for each abstract process. The existence of replicas is transprent for the user of P2P-MPI, except at the application start where the number of replicas should be specified on the command line.

3.3 P2P-MPI Architecture

3.3.1 Overview

P2P-MPI is a *middleware* based on the JXTA implementation. Figure 3.1 shows how the different layers interact : our P2P-MPI relies on the JXTA layer, which

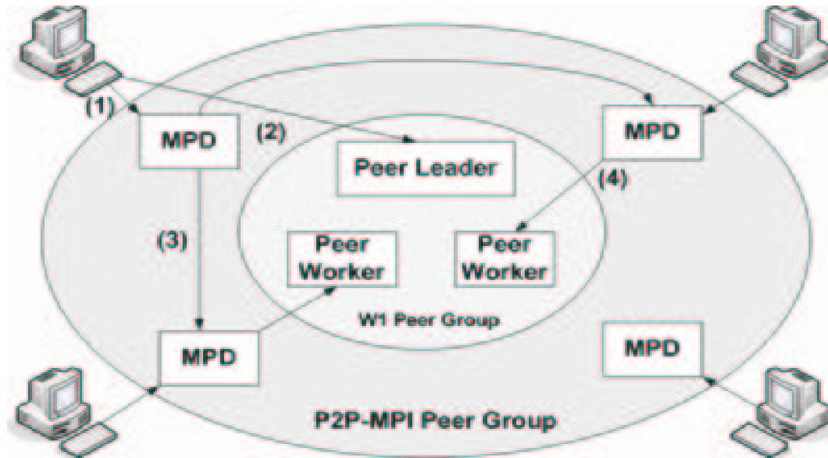


Figure 3.2: P2P-MPI Platform in JXTA.

in turn may need a Java Virutal Machine for the java implementation. The P2P-MPI layer consists of three main processes that we called *MPD*, *PeerLeader* and *PeerWorker*.

MPD is the main process which resides on all computers that wish to share the computing resources through P2P-MPI. MPD is an instance of a JXTA peer. Its main functions are to search for other MPD processes when it is requested by PeerLeader and to request other MPD processes to spawn PeerWorker process to perform an MPI application.

PeerWorker is an instance of a JXTA peer which encapsulates a user's MPI process. It is spawned by MPD to perform a task represented by that MPI process.

PeerLeader is also an instance of a JXTA peer which encapsulates user's MPI process rank 0. For one MPI application there is only one PeerLeader but multiple PeerWorkers. PeerLeader is launched on the user's machine when a user asks to execute an MPI application. It communicates with the local MPD for requesting PeerWorkers to perform an MPI application. (Hence his name, *leader* since it has the active role of requesting other nodes to collaborate, while *workers* are passive until they accept a work request).

In next sections, we describe step by step how these three kinds of peer interact one with each other.

3.3.2 P2P-MPI Platform Startup

The computers which accept to share their computing resources or to use the resource from P2P-MPI platform have to join the P2P-MPI platform first. Joining the platform only requires to execute the MPD process. As soon as MPD is

launched, it registers to the JXTA net peer group, and then tries to search the P2P-MPI peer group advertisement. If it finds the P2P-MPI peer group advertisement, it joins that peer group (see figure 3.2, step (1)). Otherwise, it creates a new P2P-MPI peer group, publishes the corresponding advertisement and joins the P2P-MPI peer group. When MPD is in P2P-MPI peer group, it creates a special advertisement and publishes it to declare that it is ready to provide resources for the P2P-MPI platform. Thus, one MPD advertisement refers to one available machine for executing MPI applications.

3.3.3 Nodes Finding to Run an MPI Application

To run an MPI application, we need to find nodes to execute its processes. This node finding task is automatic in P2P-MPI platform and is performed by executing PeerLeader. PeerLeader takes the number of abstract processes, the number of replicas per abstract process, the transferred file list and the name of the MPI application as arguments. The command usage is shown on figure 3.3.

Moreover, we provide more facility to users by allowing them to specific the number of replicas per rank themselves. As we see in figure 3.3, the second command line allows users to give a node-configuration file as an argument. A node-configuration file contains a specific number of replicas correspond to MPI ranks.

```
$java PeerLeader
Usage: PeerLeader <NumPeer> <NumReplica> <FileList> <run command> OR
     : PeerLeader <Node-ConfigFile> <FileList> <run command>
Example : PeerLeader 4 3 TransferedFileList Test_P2P_MPI
```

Figure 3.3: The command line for launching an MPI application.

It has the same role as the *mpirun* command of most MPI implementation (e.g. MPICH [29]) but adds two extra parameters : the number of replicas and the transferred file list parameters which are necessary to acquire the fault tolerance and the heterogeneous properties.

Let us show the scenario of node finding mechanism step-by-step :

1. When a user executes PeerLeader with the appropriated arguments, PeerLeader joins the P2P-MPI peer group and then creates a new peer group with a unique name inside the P2P-MPI peer group. The figure 3.2 step (2), shows that PeerLeader creates and joins the *W1* peer group.

2. PeerLeader sends its created peer group name and the number of nodes it needs for performing an MPI application to the MPD in local machine.

3. MPD in local machine searches for advertisements of other MPDs. When it finds enough advertisements for executing the MPI application then it sends a request message to other MPDs to ask for PeerWorkers (see figure 3.2, step (3)).

4. When one distant MPD receives a request message, it spawns a PeerWorker process (see figure 3.2, step (4)).

5. The spawned PeerWorker process joins the P2P-MPI peer group first and then joins the new peer group created by PeerLeader.

So we can see now that all of the necessary nodes for executing the MPI application are in the same peer group, which is a necessary condition for them to communicate.

3.3.4 Process Replication

We now explain how the process replication is related to MPI rank assignment. P2P-MPI uses an active replication technique which implies that all processes are active to avoid the use of checkpoint mechanisms.

As soon as PeerLeader is executed, it communicates with the MPD in local machine. MPD takes the number of MPI nodes and the number of replicas as arguments and then tries to search the advertisements to meet the constraint of the following formula

$$A = \sum_{i=0}^{n} R_i \tag{3.1}$$

where A is the number of advertisements, R_i is the number of process replicas of rank i, and n is the number of abstract processes needed for executing the MPI application.

This formula refers to the use of one MPD peer as one node in MPI application. However, MPD tries to find advertisements for a certain time. If it cannot find enough advertisements to meet the constraint of the formula 3.1 , then it checks with the formula 3.2 to ensure that the replicas of the same MPI rank are not executed in the same machine.

$$A = \underset{0 \leq i \leq n}{MAX}(R_i) \tag{3.2}$$

However, when MPD gets enough advertisements for executing an MPI application, it assigns the MPI rank for each process using a round-robin distribution (see table 3.1).

In the table 3.1, we show the MPI rank assignment computed by the local MPD when the user executes an MPI application requiring three abstract processes (the equivalent of `mpirun -np 3 ...`). In addition, it is asked three replicas

node 1	0
node 2	1
node 3	1
node 4	1
node 5	2
node 6	2

(a)

node 1	0 2
node 2	1 2
node 3	1
node 4	1

(b)

node 1	0 1
node 2	1 2
node 3	1 2

(c)

Table 3.1: MPI rank assignment.

of rank 1, and two replicas of rank 2 ($n = 2, R_0 = 1, R_1 = 3, R_2 = 2$). Let us examine different scenarios, summarized in table 3.1:

- In (a), 6 advertisments (i.e. 6 available nodes) were found. This is the maximum number of advertisement that we need to execute the MPI application. MPD assigns rank 0 to node 1, rank 1 to node2 and so on in a round-robin fashion.

- In (b), we assume that MPD found 4 advertisments: there is not enough advertisements to meet the constraint of the formula 3.1 but it is more than the minimum required. MPD assigns the MPI ranks so that node 1 executes both MPI process of rank 0 and MPI process of rank 2.

- In (c), MPD found the minimum number of advertisements to execute the MPI application without having same MPI processes of the same rank executed in the same node. MPD tries to search the advertisements until it gets at least the minimum number of advertisements.

Rank 0 is always executed as one process without any extra-replica, because rank 0 is executed at requesting user's machine and normally the rank 0 of MPI application is the process which collects the result. Thus, if rank 0 crashes, we assume that the MPI application can not collect the results back. Later on, when others MPI processes get the notify from the fault detector that rank 0 dies, they will silently quit the program.

3.3.5 Fault Detector in P2P-MPI

P2P-MPI is based on a peer-to-peer model to form the large-scale grid computing platform. Thus, we chose to implement the gossip-style fault detector which can reduce the bottle's neck over the *push* and *pull* models.

The gossip-style fault detector is simple to implement. It is embedded in PeerWorker and PeerLeader module. As soon as PeerWorker or PeerLeader spawns a

real MPI application process, the gossip-style fault detector is launched and periodically performs the gossip (sends gossip messages, merges heartbeat counter from other gossip messages) until the MPI application terminates.

3.3.6 Message Buffering Table

The messages in P2P-MPI are asynchronous which means P2P-MPI library creates a thread for sending a message in the sender part. The receiver part also creates a thread for receiving that message. Once the receiver thread receives a message it puts it into the message buffering table.

The MPI standard requires that when several messages with the same *tag* are sent, they are received in the same order they were sent. The P2P-MPI follows that property by implementing the unique identifier for each MPI message. The identifier of each MPI message is constructed as following :

midkey = <src>_<dest>_<tag>
MID = <commID>_<midkey>_<nummessage>

where *commID* is the identifier of the communicator (The COMM_WORLD[1] identifier is 0), *src* is the MPI rank of sender process, *dest* is the MPI rank of receiver process, *tag* is a tag number of MPI message and *nummessage* is the number of calling MPI_Send/MPI_Recv of a message which has the same *midkey*.

For example, in COMM_WORLD (commID = 0), a MPI process of rank 0 sends two messages with the same tag (tag=1) to a MPI process of rank 2. P2P-MPI library constructs the identifier of a first message with *commID = 0, src = 0, dest = 2, tag = 1* and *nummessage = 0* (Assume that this is the first time that MPI_Send/MPI_Recv is called with *midkey = 0_2_1*). Thus, the identifier of the first message is **0_0_2_1_0** and **0_0_2_1_1** for the second message. The same phenomenon for the receiver, the first MPI_Recv call will wait for the message with the identifier **0_0_2_1_0** and **0_0_2_1_1** for the second MPI_Recv call. If the second message arrives first, it is put into the message buffering table. Thus, P2P-MPI preserves the message order according to the MPI standard.

3.3.7 Sending Message Agreement Protocol

In the previous chapter, we discussed the active replication of processes. We know that active replication can reduce the recovery time. For MPI applications, we can not use the active replication technique as simply as in distributed systems.

Let us consider again figure 2.1(b), page 12. In MPI applications, we also have replicas of p_c process which execute the same operation. For example, suppose

[1]The default communicator which is created by `MPI.Init()`.

we have P_0^1, P_0^2, P_0^3 as replicas of the process rank 0 which behaves like p_c by executing MPI_Send. That means in total, one call of MPI_Send function in MPI application causes nine emissions of real messages. The network bandwidth becomes the problem.

To eliminate this problem, we propose algorithms to make an agreement between replicas to decide which replica takes a chart to send the real message. Normally, MPI programmers try to avoid the exchange of messages between MPI nodes as much as possible. Because the communication between nodes via network interconnection is not fast. Thus, MPI programmers tries to pack small messages into bigger ones before sending it. This practise is called *coarse-grain approach*. Our agreement messages are small and can be negligible when compareid with the real message from the coarse-grain approach. We discuss in the following the algorithms that are implemented in P2P-MPI to reduce the number of messages.

Information Table Notation

Each process maintains an information table (IT) which has n entries $\{it_1, it_2, .., it_n\}$ (n is the number of buffered messages). Each entry is composed of the following fields:

- Message identifier *(it.mid)*
- Message status *(it.status)* {Req , Sending , Done , None }
- Set of message sender *(it.sender)*
- Set of message destination *(it.sendto)*
- Buffered message *(it.msg)*
- Set of asked permission process *(it.ask)*
- Set of confirm process *(it.confirm)*

Message Notation

We have two types of messages *internal messages* and *external messages.*

- **Internal Message (IM)** is used for communicating between replicas of the same MPI rank to make an agreement for sending the real MPI message (External message). Its role is also to update the message status. the IM protocol is constructed as ⟨Command, MessageID, Sender, SendTo⟩ where we will denote *im.cmd*, *im.mid*, *im.sender*, and *im.sendto* the values of Command, MessageID, Sender, SendTo respectively. We implement four commands for IM (Req , Ack , Done , and Query).

- **External Message (EM)** is a real MPI message. Each EM has an unique identifier. Let us denote *EM.mid* the identifier of EM.

Message Sending Management

When a user invokes the P2P-MPI library for sending a message (equivalent to a MPI_Send instruction), it follows the behavior specified by algorithm 1 (presented in appendix A, page 42). The P2P-MPI library checks if a message with the same identifier already exists in its information table.

If a message with the same identifier exists, it means that one of its replicas already requested to send this message. Thus, it checks the message status. If the message status is Sending (one of its replicas is sending that message) or Done (one of its replicas already sent that message). It simply puts that message into its information table. Otherwise, if the message status is None , it means one of its replicas requested to send that message but crashed before completly sending that message. So, it puts the message in the information table and sends a Req message to its replicas for an agreement on sending a message while it putis its replica name into the *ask list*.

But if the message with the same identifier does not exist in the information table, the request from that replica is the first requests for sending an external message to that destination. Thus, it makes Req message to its replicas for an agreement including adding its replicas name into the *ask list*.

Presented on figure 3.4 is the case where one replica implementing an MPI process P_0 is the first of the replica group to execute the `MPI_Send` to P_1. It asks other replicas of the group if the communication is already done or in progress somewhere. As it receives only an ACK, it proceeds to the real communication, and send broadcast a DONE status to replicas of its group.

Internal Messages Management

When the P2P-MPI receives an internal message, it behaves accordingly to algorithm 3 (presented in A, page 44). The P2P-MPI library checks the command type of an internal message (*IM.cmd*). If the command is Req , it means one of its replicas wants a confirmation for sending an external message. Then, it looks at its information table.

If the message does not exist then it creates a new entry in the table and sends an Ack message back to the requester. Otherwise it checks the message status:

- If the message status is Req the requester is also asking for an agreement. Thus, it compares the requester's priority with its own priority. If the requester's priority is less than its priority, it ignores the Req message. Otherwise, it sends an Ack message back to the requester.

- If the message status is Sending then one of its replica is sending the message, so it simply replies an Ack message and add the requester in the *sender*

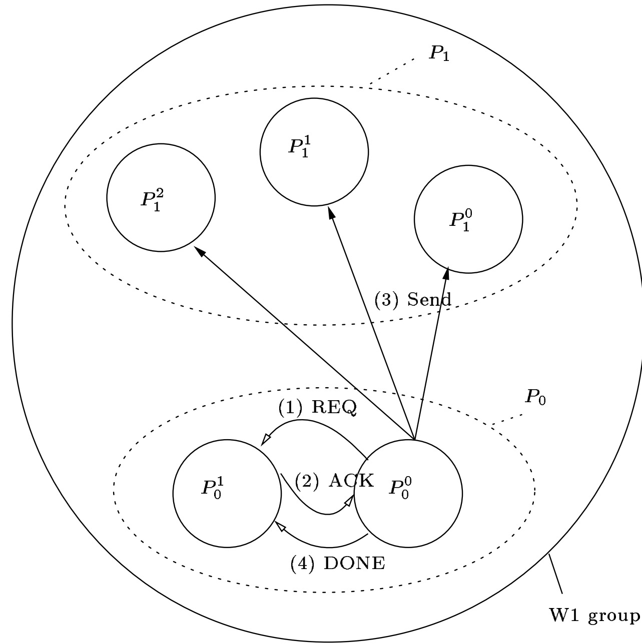


Figure 3.4: Message sending from abstract process P_0 to P_1. Only the replica P_0^0 performs the communication after an agreement with the other replica P_0^1.

list. If the message status is None , it means one of its replicas crashed before sending a Done message. Thus, it changes the message status to Sending and sends an Ack message back to the requester.

- If the message status is Done it means the message was sent completely. In that case, the process notifies all of its replicas to update their information to theDone state.

If the command is Ack which is sent by its replicas when its replicas agree on let this process send the external message. This process adds replicas who send it an Ack message (confirmed replica) into the *confirm list* and checks if all of its replicas agree on sending an external message. If all replicas agree, it sends the external message and then sends an internal message with the Done command to all of its replicas.

If the command is Done which means that one of its replicas completes sending an external message so it changes the message status in its information table to Done state.

Otherwise if the command is Query which is sent by one of its replicas to verity if the message is sent completely. If the message is sent completely it sends a reply internal message to a sender with the Done command. Otherwise, it ignores the Query message.

If there is no sender left in the sender list then it sets the message status to the None state. Then, it checks if the message status is None and if so, looks if there is a message contents. In that case, it checks in its *ask list* to see if it already asked for an agreement.

If an agreement has been asked and the *ask list* and *confirm list* are equal (the intersection of *ask list* and *confirm list* are null) then it sends the external message. If it has never asked for an agreement before then it send a Req message to its replicas.

Fault Detection Notification Management

Algorithm 2 (in appendix A, page 43) describes the way P2P-MPI behaves when it receives a notification message from a fault detector (FD). Suppose a PeerWorker or PeerLeader receives a fault notification for a crashed process P_i^j. There is two tasks to be done to maintain the system in a coherent state:

- **Unlock the pending messages**: as we may wait for authorization from P_i^j to send a message (i.e. for some message m, the status of m is not Done and P_i^j is in the *ask list*). Therefore, we must scan the information table to find such messages and when one is found, P_i^j is removed from the *ask list* and added to the *confirm list* to simulate P_i^j sent an Ack message to agree on the external message sending.

- **Update the communicator**: in subsequent communications, no more message should of course be sent to P_i^j. It is hence removed from the communication table.

3.4 P2P-MPI in Programmers' View

Let us now give an overview of what a user (i.e. an MPI programmer) of P2P-MPI interact with the system.

The various mechanisms detailed previously (the fault tolerant mechanisms, the node finding mechanism, and the transfer of executable files and dependent data files) are transparent to the programmer. The programmer is provided with an API offering some basic MPI-like functions. The first apparent difference with the C/C++ or Fortran MPI specification is the object-oriented style of the P2P-MPI API. A complete list of the API function is given in appendix B.

Thus, programmers can write message-passing programs without difficulty. Adapting an existing C/C++ or Fortran MPI program to our Java P2P-MPI framework is also possible if the semantics of the MPI primitives used may be replaced by one primitive of our P2P-MPI subset. For instance, communications in JXTA are inherently asynchronous, so that adapting a synchronous MPI

`MPI_Ssend` call requires to explicitely manage an acknowledgment. In the table 3.2, we compare the send and receive routines of the MPI specification in C and C++ with our Java bindings for P2P-MPI.

In C :

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)

int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm, MPI_Status *status)
```

In C++ :

```
void MPI::Comm::Send(void *buf, int count, const Datatype& datatype,
                     int dest, int tag) const

void MPI::Comm::Recv(void *buf, int count, const Datatype& datatype,
                     int source, int tag, Status& status) const
```

In Java P2P-MPI :

```
void Comm.Send(Object buf, int offset, int count,
               Datatype datatype, int dest, int tag)

void Comm.Recv(Object buf, int offset, int count,
               Datatype type, int source, int tag)
```

Table 3.2: The comparison of the MPI specification in C,C++ and P2P-MPI for send and receive routines.

Another difference of our P2P-MPI which provides Java bindings and the MPI specification in C and C++ is that Java has no *pointer*. Thus, programmers can not pass a pointer variable with an operation for setting an offset of a starting point in a buffer (i.e. $(buf + 5)$ if buf is a pointer to an array of integer) as an argument for send and receive routines. We fix this problem by providing a new parameter named offset. So now, programmers can set the offset at a starting point in a buffer via this argument. Because of the lack of time, we could not go further in the adaptation of the MPI routines. The `MPI_Status` data structure is one example of heavy task that we leave for future work.

To get an idea of what P2P-MPI programs look like, let us show an example. The book [30] has an example page 25, of a sample Pi calculation in MPI. In table

3.3, we have written the same example using P2P-MPI function calls instead.

```
1  public class Pi {
2     public static void main(String[] args) {
3        int rank, size, i;
4        double PI25DT = 3.141592653589793238462643;
5        double h, sum, x;
6
7        MPI.Init();
8        double startTime = MPI.Wtime();
9
10       size = MPI.COMM_WORLD.Size();
11       rank = MPI.COMM_WORLD.Rank();
12
13       int[] n = new int[1];
14       double[] mypi = new double[1];
15       double[] pi   = new double[1];
16
17       if(rank == 0) {
18          n[0] = 1000000; // number of interval
19       }
20
21       MPI.COMM_WORLD.Bcast(n, 0, 1, MPI.INT, 0);
22
23       h = 1.0 / (double)n[0];
24       sum = 0.0;
25       for(i = rank + 1; i <= n[0]; i+= size) {
26          x = h * ((double)i - 0.5);
27          sum += (4.0/(1.0 + x*x));
28       }
29       mypi[0] = h * sum;
30
31       MPI.COMM_WORLD.Reduce(mypi, 0, pi, 0, 1, MPI.DOUBLE, MPI.SUM, 0);
32
33       if(rank == 0) {
34          System.out.println("Pi is approximately " + pi[0]);
35          System.out.println("Error is " + (pi[0] - PI25DT));
36          double stopTime = MPI.Wtime();
37          System.out.println("Time usage = " + (stopTime - startTime) + " ms");
38       }
39
40       MPI.Finalize();
41    }
42 }
```

Table 3.3: The example of Pi program in P2P-MPI.

3.5 Experiments

The platform proposal made in the previous section has led to the development of a first prototype of P2P-MPI which has served to do some preliminary experiments. These experiments would deserve much more time and effort in order to get a more precise picture of the behavior and performance of the system. We consider that the first part of the work was to test the feasibility of the project, and we leave to the conclusion some future work ideas. There is no doubt that

extensive testing and numerous experiments will give many ideas and suggestions of improvements.

In the following sections, we only give some hints about what can be tested, and what we observed on the first runs we made using one students computers room. Eight computers were available at the time of the tests. Each computer has the following specification :

CPU: Pentium-II 350 MHz.

RAM: 256 MB.

Operation System: Linux Mandrake, Kernel 2.4.21-0.13mdk.

Interconnection: 100 Mbps Ethernet, LAN.

Java Runtime: Java version 1.4.2_04.

JXTA Library: JXTA version 2.3.

3.5.1 Time for Joining Groups

Table 3.4 shows the time elapsed for joining the JXTA net peer group and joining the P2P-MPI peer group. The time for joining a JXTA group depends on the CPU and network speed. We can see that the time elapsed between joining the JXTA net peer group and the P2P-MPI peer group is nearly the same. Thus, we can conclude that this is the average time JXTA takes to form a peer group.

In practise, the time required for constituting the P2P-MPI peer group is longer if it is the first attemp to build the P2P-MPI platform. The reason is that the local MPD spends much time looking for the P2P-MPI peer group advertisement amongst possible rendezvous peers the first time. On subsequent runs, it uses the P2P-MPI peer group advertisement from its cache. Results presented in table 3.4 are times measured when peers may have advertisements left in their cache.

Join JXTA Net Peer Group (s)	Join P2P-MPI Peer Group (s)
30-35	31-35

Table 3.4: The time usage for joining a JXTA peer group.

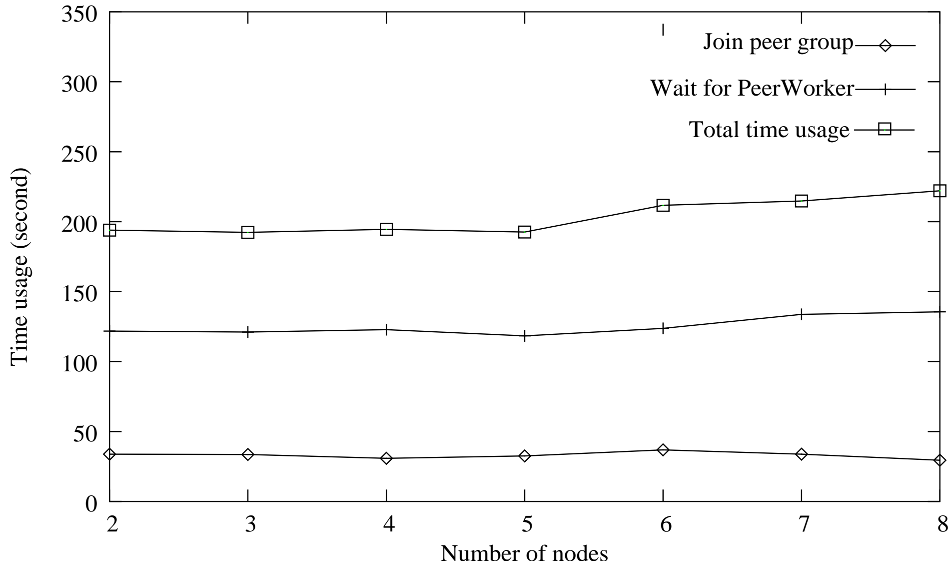


Figure 3.5: The P2P-MPI startup time usage.

3.5.2 Time for Creating a MPI Platform

The first experiment measures how long does P2P-MPI take for creating a platform for an MPI application. We decompose the platform creation in three steps as in figure 3.5:

- The time used for joining a peer group. We see in table 3.4 that the time needed for a peer to join both the JXTA net peer group and the P2P-MPI peer group is nearly the same. This step takes around 30-35 seconds for joining one peer group.

- The time spent waiting for enough PeerWorkers to be available. We can see that this step takes more than a half of the time of the MPI platform creation. The reason is because each PeerWorker is spawned on demand. Thus, PeerWorker has to join the JXTA net peer group first, join the P2P-MPI peer group and then join the MPI application peer group. This step takes about 120 seconds

- The total time used to create a MPI platform. As we can simply calculate, the total time is composed of 2 times using in the first step (joining JXTA net peer group and joining P2P-MPI peer group) and a time using in the second step. Thus, it takes roughly 180-190 seconds. However, there are minor operations which consume a little time. So, the average time for creating a MPI platform is about 200-220 seconds.

From the experiment, we found that the total time needed to create the MPI platform is roughly the same for 2 to 8 nodes. Though the local MPD has to

transfer one executable file plus a set of input data files plus one communication table (communicator) per node involved, the time needed to send these data is quite small relatively to the group joining durations. Furthermore, the executable and data files used in our experiment were small (about 910 KB).

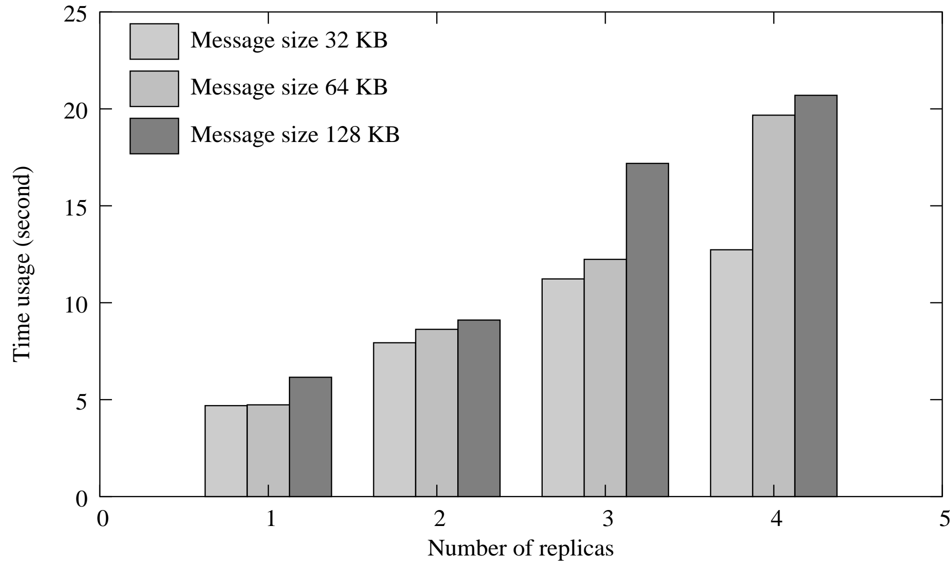


Figure 3.6: The impact of the number of replicas on a ping-pong application.

3.5.3 The Impact of Replicas on Sending a Message

The second experiment measures the impact of the number of replicas on message sendings. We measure with a sample *ping-pong* program the time taken to send a message and to receive it back. At time t_0, the abstract process of rank 0 sends a message to all replicas of the abstract process of rank 1. Then, at t_1, the abstract process of rank 0 receives that message back. We evaluate this test by increasing the number of replicas from one to four. Figure 3.6 shows the time usage for the ping-pong application depending of the number of replicas, from one to four. Moreover, we test it with three message sizes (32 KB, 64 KB, and 128 KB).

The results show that the time spent in sending/receiving messages increases linearly with the number of replicas. The reason lies in the use of the active replication, which involves that the message sender sends the message to all replicas of a received abstract process. However, we can see that the time usage incrases only around in order of the number of replicas which shows that our sending agreement algorithm can reduce bandwidth usage and time usage. Because the replicas of abstract process rank 1 make an agreement before sending a message back thus there is only 1 message back instead of n messages.

For the message size, we can see from the experiment that it is less significant. It means that P2P-MPI spends more time on connecting other peers to send a

message than time on sending message.

The results from the experiments are sometimes difficult to interpret. The reason lies in the communications in P2P-MPI transmit via JXTA pipes. It seems that sometimes it takes too long to establish a communication. However, it gives us the idea of average time usage and performance of P2P-MPI.

Chapter 4

Conclusion

Our goal was to provide a robust infrastructure for message passing applications in harness environment such as Grid and desktop Grid. We aim to provide a middleware that can serve as a robust infrastructure for MPI applications and one possible solution in our opinion, is to design our middleware on a peer-to-peer model basis.

We developed a first prototype that implements a subset of the MPI specification in a object oriented programming style. Eventhough, the performance is probably far under classical implementations of MPI, we gain the *heterogeneity*, *fault tolerance* and *auto-configuration* properties which the standard MPI does not provide.

We rely on an implementation of the JXTA set of protocols for all tasks that we can consider of lower level in the middleware between the application and the operating system: peer group management, discovery service, communication channel management, ... AS the JXTA project is quite active and interests many universities in term of research, we reckon that the P2P-MPI performance may benefit of newer versions of the JXTA platform. Having used the under-development version of JXTA throughout the project development, we have seen a lot of changes in the successive JXTA implementations.

However, there is a few technical details that need to be fixed in our first prototype. We must find a way to tackle the fact that the JXTA java implementation[1] consumes much memory, making very difficult to have several peers on

[1] The upcoming Jxta-C implementing the version 2 of protocols will probably be less memory consuming. Changing from JXTA-Java to Jxta-C would not cause any problem since the P2P-MPI library uses TCP sockets to communicate with JXTA instances. Later on, we can even imagine to provide P2P-MPI libraries for many programming languages. However the Java programming language is still our priority because of its bytecode programs allowing to overcome the operating systems and processors heterogeneity issue.

computers with a small amount of memory. The firewalls are still an obstacle for node finding, and a related problem is the impossibility to set the configuration so that we can use P2P-MPI in a private network (we need the default net peer group from Sun). These problems should be solved in the short term, since they need only some extra-time to make it fully functionnal. In the mid-term, more time will be necessary to experiment the middle-ware as a large-scale system so as to identify its main weaknesses.

In our P2P-MPI prototype, we provide a sample fault tolerant mechanism by replication to enable more robust MPI executions. However, an MPI application run can fail if all the replicas crash. To overcome this, we think we can add a recovery mechanism at a relatively low (development) cost: we could introduce the use of an information table which stores every sent messsage contents. After the crash of a replica process, the local MPD could request a new node to re-execute the MPI application in order to replace the crashed process. That new process wold then have to request the sent messages from the owner of messages.

In the long term, the perspective is to deliver a thoroughly tested version of P2P-MPI to a large number of people having access to internet. People running P2P-MPI on their home PCs would enable one of the first message passing applications runs on desktop grids. We know however, that much work must be done to be able to deploy application at such a large scale.

Appendix A

Algorithms

In this appendix are presented the algorithms implemented in the core of P2P-MPI, essentially to deal with communications management. We detail the behavior of P2P-MPI when a message is sent to an abstract process (Algorithm 1), how fault detection notifications are treated (Algorithm 2) and finally, how internal messages are handled (Algorithm 3).

A.1 Message Sending Management

When a user invokes the P2P-MPI library for sending a message (equivalent to a MPI_Send instruction), it follows the behavior specified by the following algorithm.

```
When P_i^r decides to send a message EM to P_j;
Lookup in IT where it.mid = EM.mid;
if Found it.mid matches EM.mid in IT then
    switch it.status do
        case SENDING, DONE
            it.msg = EM;

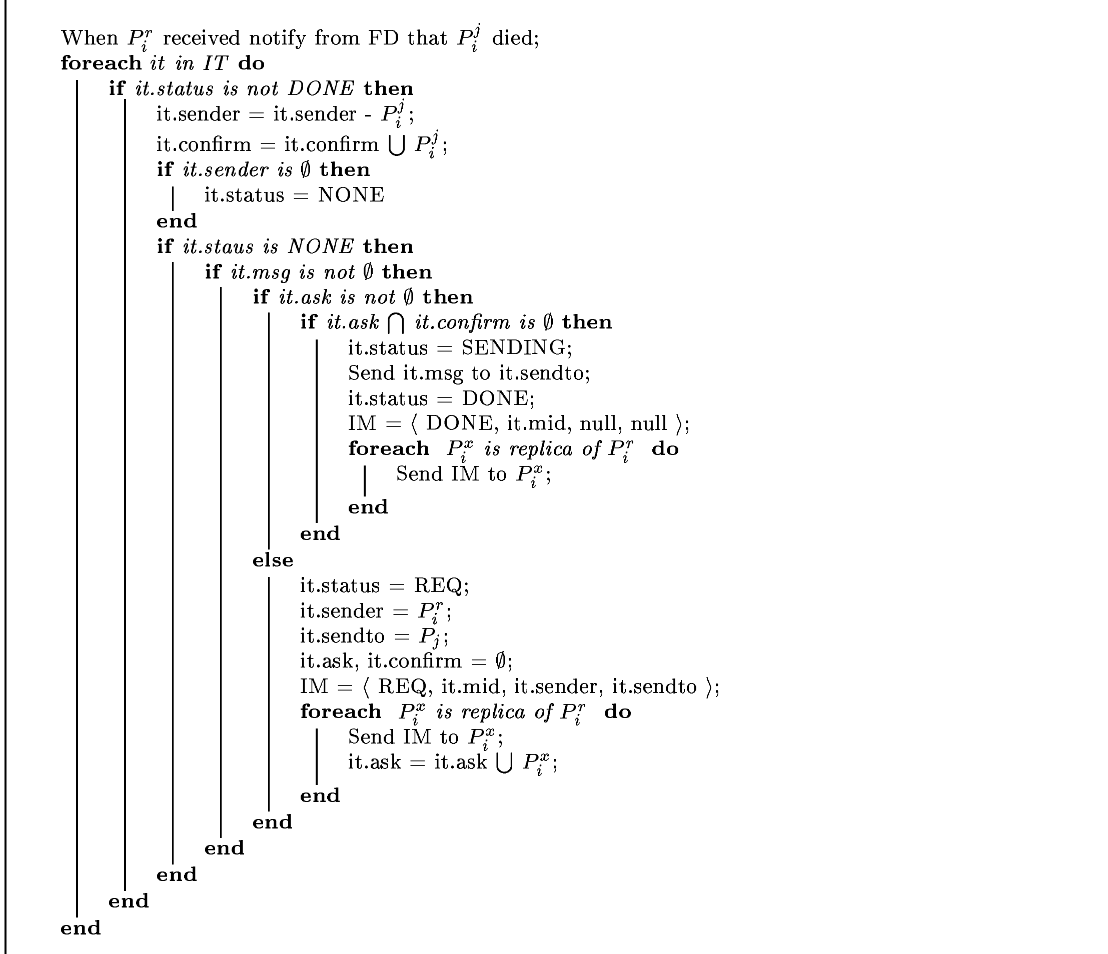
        case NONE
            it.status = REQ;
            it.sender = P_i^r;
            it.sendto = P_j;
            it.msg = EM;
            it.ask, it.confirm = ∅;
            IM = ⟨ REQ, it.mid, it.sender, it.sendto ⟩;
            foreach P_i^x is replica of P_i^r do
                Send IM to P_i^x;
                j.ask = j.ask ⋃ P_i^x;
            end

    end
else
    Create new j entry in IT;
    j.mid = EM.mid;
    j.status = REQ;
    j.sender = P_i^r;
    j.sendto = P_j;
    j.msg = EM;
    j.ask, j.confirm = ∅;
    IM = ⟨ REQ, j.mid, j.sender, j.sendto ⟩;
    foreach P_i^x is replica of P_i^r do
        Send IM to P_i^x;
        j.ask = j.ask ⋃ P_i^x;
    end
end
```

Algorithm 1: MPI_Send management.

A.2 Fault Detection Notification

This algorithm describes the way P2P-MPI behaves when it receives a notification message from a fault detector (FD).



When P_i^r received notify from FD that P_i^j died;
foreach *it in IT* **do**
  if *it.status is not DONE* **then**
    it.sender = it.sender - P_i^j;
    it.confirm = it.confirm $\bigcup P_i^j$;
    if *it.sender is* $\emptyset$ **then**
      it.status = NONE
    end
    if *it.staus is NONE* **then**
      if *it.msg is not* $\emptyset$ **then**
        if *it.ask is not* $\emptyset$ **then**
          if *it.ask* $\bigcap$ *it.confirm is* $\emptyset$ **then**
            it.status = SENDING;
            Send it.msg to it.sendto;
            it.status = DONE;
            IM = ⟨ DONE, it.mid, null, null ⟩;
            foreach P_i^x *is replica of* P_i^r **do**
              Send IM to P_i^x;
            end
          end
        else
          it.status = REQ;
          it.sender = P_i^r;
          it.sendto = P_j;
          it.ask, it.confirm = $\emptyset$;
          IM = ⟨ REQ, it.mid, it.sender, it.sendto ⟩;
          foreach P_i^x *is replica of* P_i^r **do**
            Send IM to P_i^x;
            it.ask = it.ask $\bigcup P_i^x$;
          end
        end
      end
    end
  end
end

Algorithm 2: Fault notification management.

A.3 Internal Message Management

When the P2P-MPI receives an internal message which is a special message in P2P-MPI. The internal message can be : a message for requesting an agreement on sending an external message (Req), an acknowledge message to agree on let other processes send an external message (Ack), an status update message to tell other processes that the external message is sent completely (Done), or an query message to ask for current message status (Query). The internal message management algorithm behaves accordingly to the following algorithm.

```
When P_i^r received IM from P_i^j;
switch IM.cmd do
    case REQ
        Lookup in IT where it.mid = IM.mid;
        if Found it.mid matches IM.mid in IT then
            switch it.status do
                case REQ
                    if Priority(P_i^r) <Priority(P_i^j) then
                        it.status = SENDING; it.sender = IM.sender;
                        IM = ⟨ ACK, it.mid, P_i^r, null ⟩;
                        Send IM to P_i^j;
                    end
                    if Priority(P_i^r) = Priority(P_i^j) then
                        IM = ⟨ ACK, it.mid, P_i^r, null ⟩;
                        Send IM to P_i^j;
                    end
                case SENDING
                    it.sender = it.sender ⋃ P_i^j;
                    IM = ⟨ ACK, it.mid, P_i^r, null ⟩;
                    Send IM to P_i^j;
                case NONE
                    it.status = SENDING; it.sender = IM.sender; it.sendto = IM.sendto;
                    it.ask, it.confirm = ∅;
                    IM = ⟨ ACK, it.mid, P_i^r, null ⟩;
                    Send IM to P_i^j;
                case DONE
                    IM = ⟨ DONE, it.mid, null, null ⟩;
                    foreach P_i^x is replica of P_i^r do
                        Send IM to P_i^x;
                    end
            end
        else
            Create new j entry in IT;
            j.mid = IM.mid; j.status = SENDING; j.sender = IM.sender; j.sendto = IM.sendto;
            j.ask, j.confirm = ∅;
            IM = ⟨ ACK, j.mid, P_i^r, null ⟩;
            Send IM to P_i^j;
        end
    case ACK
        Lookup in IT where it.mid = IM.mid;
        it.confirm = it.confirm ⋃ IM.sender;
        if it.ask - it.confirm = ∅ then
            it.status = SENDING;
            Send it.msg to it.sendto;
            it.status = DONE;
            IM = ⟨ DONE, it.mid, null, null ⟩;
            foreach P_i^x is replica of P_i^r do
                Send IM to P_i^x;
            end
        end
    case DONE
        Lookup in IT where it.mid = IM.mid;
        it.status = DONE;
    case QUERY
        Lookup in IT where it.mid = IM.mid;
        if it.status is DONE then
            IM = ⟨ DONE, it.mid, null, null ⟩;
            Send IM to P_i^j;
        end
end
```

Algorithm 3: Internal message management.

Appendix B

P2P-MPI APIs

P2P-MPI provides application programming interface in object-oriented model which is devided into four main packages : MPI package, Group Package, Comm Package, Intracomm Package.

B.1 MPI Package

In MPI package composes of a default *IntraComm* object called **COMM_WORLD**, static MPI basic data types, static MPI basic operations, and some MPI function calls as following.

B.1.1 MPI Datatypes

In our first prototype of P2P-MPI, we provide eight basics MPI data types :

MPI datatype	Java datatype
MPI.BYTE	byte
MPI.CHAR	char
MPI.SHORT	short
MPI.BOOLEAN	boolean
MPI.INT	int
MPI.LONG	long
MPI.FLOAT	float
MPI.DOUBLE	double

Table B.1: P2P-MPI datatypes.

B.1.2 MPI Operations

P2P-MPI has four built-in MPI basic operations **MPI.MAX**, **MPI.MIN**, **MPI.SUM** and **MPI.PROD**. However, MPI programmers can build their own MPI operation by constructing a new object of *Op* class

- `Op.Op(MPI_User_function function, boolean commute)`

function	user defined function
commute	**true** if commutative, otherwise **false**

The abstract base class **MPI_User_function** is defined by

```
class MPI_User_function {
    public abstract void Call(Object invec, int inoffset,
                              Object inoutvec, int inoutoffset,
                              int cont, Datatype datatype);
}
```

To define a new operation, the programmer should define a concrete subclass of **MPI_User_function**, implementing the **Call** method, then pass an object from this class to the **Op** constructor. The **MPI_User_function.Call** method plays exactly the same role as the **function** argument in the standard bindings of MPI. The actual arguments **invec** and **inoutvec** passed to **call** will be arrays containing **count** elements of the type specified in the **datatype** argument. Offsets in the arrays can be specified as for message buffers. The user-defined **Call** method should combine the arrays element by element, with results appearing in **inoutvec**

B.1.3 Startup and Timers

- `static void MPI.Init()`

  Initialize MPI. Java binding of the MPI operation *MPI_INIT*.

- `static void MPI.Finalize()`

  Finalize MPI. Java binding of the MPI operation *MPI_FINALIZE*.

- `static double MPI.Wtime()`

  Returns wallclock time. Java binding of the MPI operation *MPI_ WTIME*.

- `static double MPI.Wtick()`

  Returns resolution of timer. Java binding of the MPI operation *MPI_ WTICK*.

B.2 Groups Package

The *Group* class provides set of MPI processes which is used to create a communicator. P2P-MPI introduces some method for group management as following

- `void Group.Size()`

  Size of group. Java binding of the MPI operation *MPI_GROUP_SIZE*.

- `void Group.Rank()`

  Rank of process in group. Java binding of the MPI operation *MPI_GROUP_RANK*.

- `void Group.Incl(int [] ranks)`

  ranks ranks from this group to appear in new group

  Create a subset group including specified processes. Java binding of the MPI operation *MPI_GROUP_INCL*.

- `void Group.Excl(int [] ranks)`

  ranks ranks from this group *not* to appear in new group

  Create a subset group excluding specified processes. Java binding of the MPI operation *MPI_GROUP_EXCL*.

B.3 Comm Package

The point-to-point communication operations of MPI function is in *Comm* class which provides the basic communication operations *send* and *receive*.

- ```
  void Comm.Send(Object buf, int offset, int count,
                 Datatype datatype, int dest, int tag)
  ```

buf	send buffer array
offset	initial offset in send buffer
count	number of items to send
datatype	datatype of each item in send buffer
dest	rank of destination
tag	message tag

  Non-blocking send operation. Java binding of the MPI operation *MPI_SEND*. The data part of the message consists of a sequence of **count** values, each
  ```

of the type indicated by **datatype**. The actual argument associated with **buf** must be an array. The value **offset** is a subscript in this array, defining the position of the first item of the message.

- ```
  void Comm.Recv(Object buf, int offset, int count,
                 Datatype datatype, int source, int tag)
  ```

buf	receive buffer array
offset	initial offset in receive buffer
count	number of items in receive buffer
datatype	datatype of each item in receive buffer
dest	rank of source
tag	message tag

  Blocking receive operation. Java binding of the MPI operation *MPI_RECV*. The actual argument associated with **buf** must be an array. The value **offset** is a subscript in this array, defining the position of the first item of the incoming message will be copied.

- `int Comm.Size()`

  Size of group of the communicator. Java binding of the MPI operation *MPI_COMM_SIZE*.

- `int Comm.Rank()`

  Rank of this process in group of this communicator. Java binding of the MPI operation *MPI_COMM_RANK*.

B.4 IntraComm Package

In general P2P-MPI bindings of collective communication operation realize the MPI functions as members of the **IntraComm** class.

- ```
  void Intracomm.Bcast(Object buffer, int offset, int count,
                       Datatype datatype, int root)
  ```

buffer	buffer array
offset	initial offset in buffer
count	number of items in buffer
datatype	datatype of each item in buffer
root	rank of broadcast root

  Broadcast a message from the process with rank **root** to all processes of the group. Java binding of the MPI operation *MPI_BCAST*.

- `void Intracomm.Reduce(Object sendbuf, int sendoffset, Object recvbuf, int recvoffset, int count, Datatype datatype, Op op, int root)`

sendbuf	send buffer array
sendoffset	initial offset in send buffer
recvbuf	receive buffer array
recvoffset	initial offset in receive buffer
count	number of items in send buffer
recvtype	datatype of each item in send buffer
op	reduce operation
root	rank of root process

Combine elements in input buffer of each process using the reduce operation, and return the combined value in the output buffer of the root process. Java binding of the MPI operation *MPI_REDUCE*.

Bibliography

[1] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Publishers, August 1998.

[2] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.

[3] Jerome Verbeke, Neelakanth Nadgir, Gref Ruetsch, and IIya Sharapov. Framework for peer-to-peer distributed computing in a heterogeneous, decentralized environment. In *Grid Computing - GRID 2002*, volume 2536 of *Lecture Notes in Computer Science*, pages 1–12, Baltimore, MD, USA, November 2002. Springer-Verlag.

[4] Eddy Caron, Frédéric Desprez, Frédéric Lombard, Jean-Marc Nicod, Martin Quinson, and Frédéric Suter. A Scalable Approach to Network Enabled Servers. In B. Monien and R. Feldmann, editors, *Proceedings of the 8th International EuroPar Conference*, volume 2400, pages 907–910, Paderborn, Germany, August 2002. Springer-Verlag.

[5] Franck Cappello, Abderrahmane Djilali, Gilles Fedak, Cécile Germain, Oleg Lodygensky, and Vincent Néri. *Calcul réparti à grande échelle*, chapter XtremWeb, une plate-forme de recherche sur le Calcul Global et Pair à Pair, pages 153–188. Hermès Science Paris, 2002. ISBN 2-7462-0472-X.

[6] MPI Forum. MPI: A message passing interface standard. Technical report, University of Tennessee, Knoxville, TN, USA, June 1995.

[7] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7), July 1978.

[8] M. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message passing systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, October 1996.

[9] L. Alvisi, E. Elnozahy, S. Rao, S. A. Husain, and A. D. Mel. An analysis of communication induced checkpointing. In *29th Symposium on Fault-Tolerant Computing (FTCS'99)*, Los Alamitos, California, June 1999. IEEE CS Press.

[10] L. Alvisi and K. Marzullo. Message logging: Pessimistic, optimistic, and causal. In *Proceeding of the 15th International Conference on Distributed Computing Systems (ICDCS 1995)*, pages 229–236. IEEE CS Press, 1995.

[11] E. Strom and S. Yemini. Optimistic recovery in distributed systems. In *Transactions on Computer Systems*, volume 3(3), pages 204–226. ACM, 1985.

[12] F. Schneider. *Replication Management Using State-Machine Approach. In S. Mullender, Distributed Systems*, chapter 7, pages 169–198. Addison Wesley, 1993.

[13] N.Budhiraja and F.Schneider and S.Toueg and K. Marzullo. *The Primary-Backup Approach. In S. Mullender, Distributed Systems*, chapter 8, pages 199–216. Addison Wesley, 1993.

[14] P. Felber and X Defago and R. Guerraoui and P. Oser. Failure detectors as first class objects. In *Proceeding of the 9th IEEE International Symposium on Distributed Objects and Applications (DOA'99)*, pages 132–141, September 1999.

[15] R. van Renesse and Y. Minsky and M. Hayden. A gossip-style failure detection service. In *Middleware '98*, 1998.

[16] Georg Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, Honolulu, Hawaii, 1996.

[17] M. Litzkow et al. Condor: A Hunter of Idle Workstations. In *Proceeding of the 8th International Conference on Distributed Computing Systems*, pages 104–111, Los Alamitos, California, 1998. IEEE CS Press.

[18] A. Agbaria and R. Friedman. Starfish: Fault-Tolerant Dynamic MPI programs on Clusters of Workstations. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, pages 167–176, Los Alamitos, California, 1999.

[19] Soulla Louca, Neophytos Neophytou, Arianos Lachanas, and Paraskevas Evripidou. MPI-FT: Portable fault tolerenace scheme for MPI. In *Parallel Processing Letters*, volume 10, pages 371–382. World Scientific Publishing Company, 2000.

[20] G. Fagg and J.J. Dongarra. Ft-mpi: Fault tolerant mpi, supporting dynamic applications in a dynamic world. In *EuroPVM/MPI User's GroupMeeting 2000*, pages 346–353. Springer-Verlag, Berlin, Germany, 2000.

[21] George Bosilca, Aurelien Bouteiller, Franck Cappello, Samir Djailali, Gilles Fedak, Cecile Germain, Thomas Herault, Pierre Lemarinier, Oleg Lodygensky, Frederic Magniette, Vincent Neri, and Anton Selikhov. MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes. In *SuperComputing 2002*, Baltimore, USA, November 2002.

[22] Aurelien Bouteiller, Franck Cappello, Thomas Herault, Geraud Krawezik, Pierre Lemarinier, and Frederic Magniette. MPICH-V2: a Fault Tolerant MPI for Volatile Nodes based on the Pessimistic Sender Based Message Logging, November 2003.

[23] R. Batchu, J. Neelamegam, Z. Cui, M. Beddhua, A.Skjellum, Y. Dandass, and M. Apte. MPI/FT^{TM}: Architecture and taxonomies for fault-tolerant,message-passing middleware for performance-portable parallel computing. In *Proceedings of the 1st IEEE International Symposium of Cluster Computing and the Grid*, Melbourne, Australia, 2001.

[24] S. Mintchev. *Writing Programs in JavaMPI.* School of Computer Science, University of Westminster, 1997. MAN-CSPE-02.

[25] Bryan Carpenter, Vladimir Getov, Glenn Judd, Anthony Skjellum, and Geoffrey Fox. MPJ: MPI-like message passing for Java. *Concurrency: Practice and Experience*, 12(11):1019–1038, 2000.

[26] Napster protocol specification. `http://opennap.sourceforge.net/napster.txt`, April 2000.

[27] Andy Oram. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*, chapter Gnutella, pages 94–122. O'Reilly, May 2001.

[28] JXTA. `http://www.jxta.org`.

[29] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[30] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI, Portable Parallel Programming with the Message-Passing Interface.* Scientific and Engineering Computation Series. MIT Press, 2nd edition edition, 1999.