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

Stéphane Genaud (genaud@icps.u-strasbg.fr) and Choopan Rattanapoka (rattanapoka@icps.u-strasbg.fr)
ICPS-LSIIT - UMR CNRS-ULP 7005
Université Louis Pasteur, Strasbourg

© 2006 *Kluwer Academic Publishers. Printed in the Netherlands.*

1. Introduction

Grid computing offers the perspective of solving massive computational problems using a large number of computers arranged as clusters embedded in a distributed telecommunication infrastructure. It involves sharing heterogeneous resources (based on different platforms, hardware/software architectures) located in different places, belonging to different administrative domains over a network. When speaking of computational grids, we must distinguish between grids involving stable resources (e.g. a supercomputer) and grids built upon versatile resources, that is computers whose configuration or state changes frequently. (e.g. computers in a students computer room which are frequently switched off and whose OS is regularly re-installed). The latter are often referred to as *desktop grids* and may in general involve any unused connected computer whose owner agrees to share its CPU. Thus, provided some magic middleware glue, a desktop grid may be seen as a large-scale computer cluster allowing to run parallel application traditionally executed on parallel computers. However, the question of how we may program such cluster of heterogeneous computing resources remains unclear. Most of the numerous difficulties that people are trying to overcome today fall in two categories.

- **Middleware** The middleware management of tens or hundreds grid nodes is a tedious task that should be alleviated by mechanisms integrated to the middleware itself. These can be fault diagnostics, auto-repair mechanisms, remote update, resource scheduling, data management, etc.

- **Programming model** Many projects propose a client/server (or RPC) programming style for grid applications (e.g. JNGI [28], DIET [10] or XtremWeb [15]) offer such a programming model. A major advantage of this paradigm lies in the ability for the client to easily cope with servers failures. However, the *message passing* and *data parallel* programming models are the two models traditionally used by parallel programmers.

MPI [22] is the *de-facto* standard for message passing programs. Most MPI implementations are designed for the development of highly efficient programs, preferably on dedicated, homogeneous and stable hardware such as supercomputers. Some projects have developed improved algorithms for communications in grids (MPICH-G2 [17], PACX-MPI [16], MagPIe [18] for instance) but still, assume hardware stability. This assumption allows for a simple execution model where the number of

© 2006 *Kluwer Academic Publishers. Printed in the Netherlands.*

processes is static from the beginning to the end of the application run[1]. This design means no overhead in process management but makes fault handling difficult: one process failure causes the whole application to fail. This constraint makes traditional MPI applications unadapted to run on grids because failures of nodes are somehow frequent in this context. Moreover, MPI applications are OS-dependent binaries [2] which complicates execution in highly heterogeneous environments.

If we put these constraints altogether, we believe a middleware should provide the following features: a) self-configuration (system maintenance autonomy, discovery), b) data management, c) robustness of hosted processes (fault detection and replication), and d) abstract computing capability.

This paper introduces a new middleware called P2P-MPI designed to meet the above requirements. The contribution of P2P-MPI is its integrated approach: it offers simultaneously a middleware with the above characteristics and a general parallel programming model based on MPI. The integration allows the communication library to transparently handles robustness by relying on the internals of the middleware, relieving the programmer from the tedious task of explicitly specifying how faults are to be recovered. The rest of the paper shows how P2P-MPI fulfills these requirements. We first describe (section 2) the P2P-MPI middleware through its modules, so as to understand the protocols defined to gather collaborating nodes in order to form a platform suitable for a job request. In section 3 we explain the fault-detection and replication mechanisms in the context of message-passing programs. We finally discuss P2P-MPI behavior in section 4, at the light of experiments carried out on several benchmarks and on different hardware platforms.

2. The P2P-MPI Middleware

We describe in this section P2P-MPI 's structure and how its modules interact to gather requested resource to execute an MPI application.

2.1. Modules Organization

Figure 1 depicts how P2P-MPI modules are organized in a running environment. P2P-MPI proper parts are grayed on the figure. On top of diagram, a message-passing parallel program uses the MPI API (a

[1] Except dynamic spawning of process defined in MPI-2.
[2] The MPI specification defines bindings for C, C++ and Fortran only.

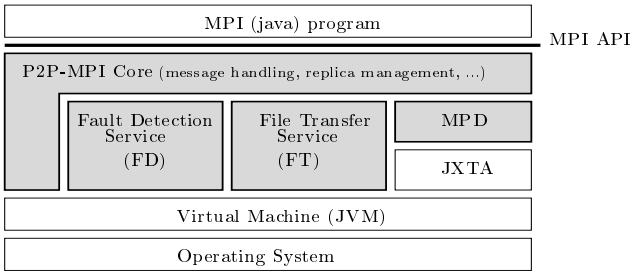

Figure 1. P2P-MPI structure.

subset of the MPJ specification [11]). The core behind the API implements appropriate message handling, and relies on three other modules. This task, as detailed in section 3 relies on processes liveliness detection. Three separate services (running as daemon processes) implement one of the desired feature listed in the introduction.

- The *Message Passing Daemon* (MPD) is responsible for self- configuration, as its role is either to search for participating nodes or to act as a gate-keeper of the local resource.
- The *File Transfer Service* (FT) module handles the data management by transferring executable code, input and output files between nodes.
- The *Fault Detection Service* (FD) module is necessary for robustness as the application needs to be notified when nodes become unreachable during execution.

In addition, we also rely on external pieces of software. The abstract computing capability is provided by a Java Virtual Machine and the MPD module uses JXTA [1] for self-configuration. JXTA defines a set of protocols that may be used in any peer-to-peer applications. The specification formalizes the role of peers, how peers group are formed to achieve a common task, and the way they can discover other peers and communicate. Currently, two implementations of the specification are actively developed with the support of Sun Micro Systems. One is in Java and the other in C, the Java implementation being always ahead the C version.

2.2. Discovery for an Execution Platform

In the case of desktop grids, the task of maintaining an up-to-date directory of participating nodes is a so tedious task that it must be

automated. We believe one of the best options for this task is *discovery*, which has proved to work well in the many *peer-to-peer* systems developed over the last years for file sharing. P2P-MPI uses the discovery service of JXTA. The discovery is depicted in JXTA as an advertisement publication mechanism. A peer looking for a particular resource posts some public advertisement and then waits for answers. (The advertisements are actually handled transparently by a set of peers called rendez-vous, whose role is to cache, broadcast, search for advertisements.) The peers which discover the advertisement directly contact the requester peer.

In P2P-MPI, we use the discovery service to find the required number of participating nodes at each application execution request. Peers in P2P-MPI are the MPD processes. When a user starts up the middleware it launches a MPD process which publishes its *pipe* advertisement. This pipe can be seen as an open communication channel that will be used to transmit boot-strap information.

When a user requests n processors for its application, the local MPD begins to search for some published pipe advertisements from other MPDs. Once enough peers have reported their availability, it connects to the remote MPDs via the pipe [3] to ask for their FT and FD services ports. The remote MPD acts as a gate-keeper in this situation and it may not return these service ports if the resource had changed its status to unavailable in the meantime. Once enough[4] hosts have sent their service ports, we have a set of hosts ready to execute a program. We call this set an *execution platform* since the platform lifetime is not longer than the application execution duration.

2.3. Job submission scenario

We now describe the steps following a user's job submission to a P2P-MPI grid. The steps listed below are illustrated on Figure 2.

(1) The user must first join the grid. By invoking `mpiboot`, it spawns the MPD process which makes the local node join the P2P-MPI group if it exists, or creates it otherwise.

(2) The job is then submitted by invoking a run command which starts the process with rank 0 of the MPI application on local host. We call this process the *root process*.

(3) Discovery: the local MPD issues a search request to find other MPDs pipe advertisements. When enough advertisements have been found,

[3] Actually a JxtaSocket built upon the pipe.
[4] We will detail in next section what is considered a minimum number of hosts.

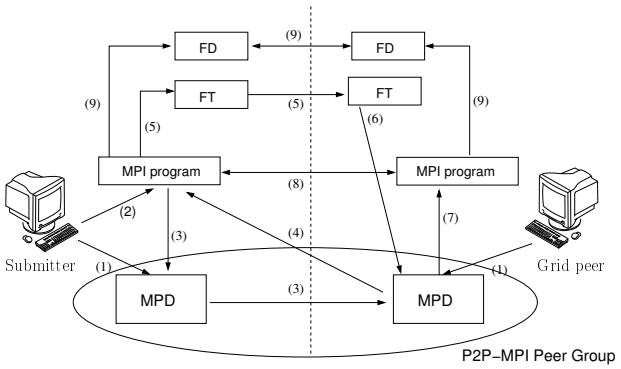

Figure 2. A submission where the submitter finds one collaborating peer.

the local MPD sends into each discovered pipe, the socket where the MPI program can be contacted.

(4) Hand-shake: the remote peer sends its FT and FD ports directly to the submitter MPI process.

(5) File transfer: program and data are downloaded from the submitter host via the FT service.

(6) Execution Notification: once the transfer is complete the FT service on remote host notifies its MPD to execute the downloaded program.

(7) Remote executable launch: MPD executes the downloaded program to join the execution platform.

(8) Execution preamble: all processes in the execution platform exchange their IP addresses to construct their local communication table, register in their local FD service and start.

(9) Fault detection: MPI processes register in their local FD service and starts. Then FD will exchange their heart-beat message and will notify MPI processes if they become aware of a node failure.

3. Replication for Robustness

As stated in the introduction, the robustness of an execution is of tremendous importance for MPI application since a single faulty process

is very likely to make the whole application fail. There is no simple solution to that problem and many of the proposals we discuss in section 5, mainly based on check-pointing are incompatible with a peer-to-peer paradigm. We have chosen for P2P-MPI a solution based on process replication.

3.1. Logical processes and replicas

Though absolutely transparent for the programmer, P2P-MPI implements a replication mechanism to increase the robustness of an execution platform. When specifying a desired number of processors, the user can request that the system run for each process an arbitrary number of copies called *replicas*. An exception is made for the root process which is not replicated because we assume a failure on the submitter host is critical. In practice, it is shorter to request the same number of replicas per process, and we call this constant the *replication degree*. Currently, because we do not take into account hosts reliability, we map processes randomly on hosts. Therefore the interest of specifying how many replicas should be chosen per process is limited. In the following we name a "usual" MPI process a *logical process*, noted P_i when it has rank i in the application. A logical process P_i is thus implemented by one or several replicas, noted $P_i^0, \ldots, P_i^n$. The replicas are run in parallel on different hosts since the goal is to allow the continuation of the execution even if one host fails.

3.2. Mapping of logical processes

We can now precise how the MPD determines if enough peers are available to satisfy the user request. If the system can not find one host per process required by the user request, then it will map several processes per node. Consider a run command requesting n logical processes with a replication degree r. In addition to the root process, we need $(n-1)r$ other nodes to run an application with one process per node. However, the minimum number of nodes we need without having replicas of the same rank residing on the same node is only r. Thus, if the MPD has found after a certain period, an "insufficient" number m of nodes such that $r \leq m < (n-1)r$, it distributes the MPI ranks $\{1, \ldots, n-1\}$ to these nodes in a round-robin way to ensure that no two replicas of the same rank run on the same node. Otherwise, the MPD will keep searching for other nodes until at least r nodes are found or a final time-out is reached.

3.3. Replicas coordination protocol

The coordination of replicas means in our case, that we must define a protocol insuring that the communication scheme is kept coherent with the semantics of the original MPI program. Such protocols have been proposed and fall into two broad classes. *Passive replication* [9] (or *primary backup*) makes senders send messages to only one process (the primary) in the group of receivers which in turns, retransmits the message to replicas of the group. In *active replication* [13], all replicas of the destination group receive the sent message. Our protocol follows the latter strategy except that specific agreement protocols are added on both sender and receiver sides.

3.3.1. *Sending message agreement protocol*

On the sender side, we limit the number of sent messages by introducing the following agreement protocol:
In each logical process, one replica is elected as master of the group for sending. Figure 3 illustrates a send instruction from P_0 to P_1 where replica P_0^0 is assigned the master's role. When a replica reaches a send instruction, two cases arise depending on the replica's status:

- if it is the master, it sends the message to all processes in the destination logical process. Once the message is sent, it notifies the other replicas in its logical process to indicate that the message has been correctly transmitted.
- if the replica is not the master, it first looks up a journal containing the identifiers of messages sent so far (*log* on Figure 3) to know if the message has already been sent by the master. If it has already been sent, the replica just goes on with subsequent instructions. If not, the message to be sent is stored into a *backup* table and the execution continues. (Execution only stops in a waiting state on a receive instruction.) When a replica receives a commit, it writes the message identifier in its log and if the message has been stored, removes it from its backup table.

3.3.2. *Reception message agreement protocol*

The communications in P2P-MPI are asynchronous and from an implementation point of view, the P2P-MPI library creates a thread for sending a message in the sender part. The receiver part also creates a thread for receiving messages. Once the receiver thread has received a message it puts it into a message buffering table. Messages are then delivered from the buffering table to the application upon the following agreement protocol.

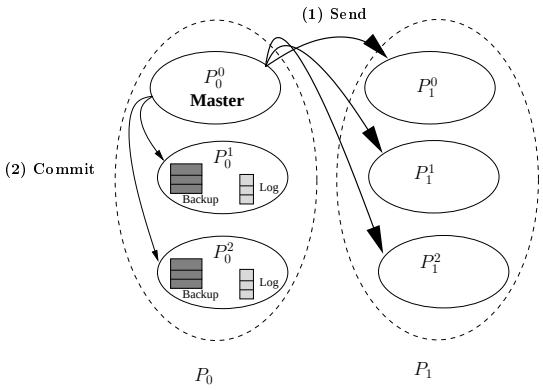

Figure 3. A message sent from logical process P_0 to P_1.

The MPI standard requires that when several messages with the same *tag* are sent, they are received in the same order they were sent. P2P-MPI implements that property using a unique identifier for each MPI message. Each message has an identifier *mid* constructed as follows:

$$mid = (cid, midkey, msg)$$
$$\text{with} \quad midkey = (src, dest, tag)$$

where *cid* is the identifier of the communicator[5], *src* and *dest* are the MPI rank of the sender and receiver processes respectively, *tag* is a tag number of the message and *msg* is the number of calling MPI_Send/MPI_Recv of a message which has the same *midkey*.

For example, in COMM_WORLD a process of rank 0 sends two messages with the same tag ($tag = 1$) to a process of rank 2. P2P-MPI constructs the identifier of a first message with *cid=0, src=0, dest=2, tag=1* and $msg = 0$. 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. Symmetrically in 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.

[5] For instance, the default communicator created by `MPI.Init()` is COMM_WORLD and has $cid = 0$.

Note that the state of replicas are not necessarily the same after the reception of a message. The only case where such a situation can happen is when using the specifiers `MPI_ANY_SOURCE` or `MPI_ANY_TAG` as source and tag values respectively in the receive call. Suppose a process P_0 implemented by two replicas (P_0^0 and P_0^1), whose code executes one receive operation followed by another, both specifying `MPI_ANY_SOURCE`. Then, assume two other processes P_1 and P_2 send to P_0 nearly at the same time, messages m_1 and m_2 respectively. It can happen that P_0^0 receives m_1 before m_2 while P_0^1 receives m_2 before m_1. However, this programming pattern denotes that the subsequent computations depending on the received values make no assumptions on the order of the receptions, and either sequence of reception is acceptable. A common example of such computation is the summation of values gathered in an unspecified order which is correct because sum is associative, commutative and has a neutral element. Yet, it remains particular situations where the consistency of results can not be guaranteed because failures can happen at any point. In the example above, consider P_0^0 sends its first received message m_1, commits its send at P_0^1, then crashes. P_0^1 becomes the master and see that the first message has been sent, so it goes on with sending its second message, which is again m_1, yielding a duplication of m_1 and a loss of m_2. A consensus on the received values could theoritically solve this problem, but given the rareness of such cases, is not implemented here for a matter of performances.

3.4. Fault Detection and Recovery

To become effective the replication mechanism needs to be notified of processes failures. The problem of failure detection has received much attention in the literature and many protocols mostly based on time-outs, have been proposed. The two classic models of fault detectors, discussed in [23], are the *push* and *pull* models. However, if these models have proved to work well in small networks, better proposals have been made for large-scale systems, such as the *gossip-style* protocol [24], which avoids the combinatorial explosion of messages. This is the protocol adopted for P2P-MPI. In this model, failure detectors are distributed and reside at each host on the network. Each detector maintains a table with one entry per detector known to it. This entry includes a counter called heartbeat counter. During execution, each detector randomly picks a distant detector and sends it its table after incrementing its heartbeat counter. The receiving failure detector will merge its local table with the received table and adopts the maximum heartbeat counter for each entry. If the heartbeat counter for some

entry has not increased after a certain time-out, the corresponding host is suspected to be down.

When the local instance of the MPI program is notified of a node failure by its FD service, it marks the node as faulty and no more messages will be sent to it on subsequent send instructions. If the faulty node hosts a master process then a new master is elected in the logical process. Once elected, it sends all messages left in its backup table.

3.5. Fault survival

We now study the probability for an application to fail after k faults have occurred. Again, we consider the root process as reliable. This leads to simpler expressions in the following discussion while having a very small impact on quantitative results when tens of processors are involved.

PROPOSITION 1. *Consider an application with p logical processes (not counting the root process), with a replication degree of r, having all its processes distributed on different hosts. If we assume faults occurrences are identically distributed among hosts, the failure probability of the application after k faults is*

$$\frac{\sum_{i=1}^{p} (-1)^{i-1} \binom{p}{i} \binom{pr-ir}{k-ir}}{\binom{pr}{k}}$$

Proof. Suppose a sample of k faulty nodes is drawn randomly. The application fails as soon as a process has its r replicas failed. Let A_i be the event "*amongst the k faulty nodes drawn, r belong to process P_i*". The probability of failure for the application is thus the union of such events for all logical processes, noted $Pr(\bigcup_{i=1}^{p} A_i)$. From the inclusion-exclusion principle (also called Poincaré formula) we have :

$$Pr(\bigcup_{i=1}^{p} A_i) = \sum_{i=1}^{p} \left((-1)^{i-1} \sum_{1 \leq j_1 < \ldots < j_i \leq p} Pr(A_{j_1} \cap \ldots \cap A_{j_i}) \right) \quad (1)$$

where the inner summation is taken over the *i-subsets* of $\{A\}_{i=1}^{p}$. (We call an i-subset a subset of a set on p elements containing exactly i elements. The number of i-subsets on p elements is $\binom{p}{i}$).

The expression $Pr(A_{j_1} \cap \ldots \cap A_{j_i})$ denotes the probability that the i logical processes $j_1, \ldots j_i$ fail, that is r faults occur in each process. Following the counting principle of an hypergeometric distribution, we have:

$$Pr(A_{j_1} \cap \ldots \cap A_{j_i}) = \frac{\binom{r}{r}^i \cdot \binom{pr-ir}{k-ir}}{\binom{pr}{k}} = \frac{\binom{pr-ir}{k-ir}}{\binom{pr}{k}} \quad (2)$$

Indeed, amongst the k faults we have r faults on each of these i processes, which is $\binom{r}{r}^i = 1$ possibility. There remains $k - ir$ faults to be chosen from the $pr - ir$ faults (on the $p - i$ remaining processes), which is $\binom{pr-ir}{k-ir}$.

We now sum the expression of equation 2:

$$\sum_{1 \leq j_1 < \ldots < j_i \leq p} Pr(A_{j_1} \cap \ldots \cap A_{j_i}) = \frac{\binom{pr-ir}{k-ir}}{\binom{pr}{k}} \cdot \sum_{1 \leq j_1 < \ldots < j_i \leq p} 1 \tag{3}$$

and by definition of the i-subsets,

$$\sum_{1 \leq j_1 < \ldots < j_i \leq p} 1 = \mid \{(j_1, \ldots, j_i) \in \{1, \ldots, p\}^p \mid j_1 < \ldots < j_i\} \mid = \binom{p}{i}$$

replacing into equation 1, we have :

$$Pr(\bigcup_{i=1}^{p} A_i) = \frac{\sum_{i=1}^{p} (-1)^{i-1} \binom{p}{i} \binom{pr-ir}{k-ir}}{\binom{pr}{k}} \tag{4}$$

□

We illustrate on figure 4 how the probability of application failure evolves with the number of faulty nodes. We make the number of faults vary in a range bounded by the minimum that can crash the application, which is r, and the maximum faults that the application can tolerate, which is $p(r-1)$. We truncate this range when probability is very close to 1. The top figure shows the behavior for an application with 50 processes and the lower figure for 100 processes.

These data help clarify what kind of platform P2P-MPI could be used on, and with which replication degree. A 50 processes application has 80% chance of surviving after 7 faults with a replication degree of 2. Even for a run that lasts more than an hour we feel the user can be confident if the execution platform consists of some pretty stable resources (e.g. computers owned by staff members in a university). On the contrary, on unstable resources (e.g. machines from student computer rooms) a replication degree of 3 would probably be more appropriate as the above probability is reached for 26 faults.

3.6. Fault survival experiments

A set of experiments has been conducted to validate some of the above analytical results. The experiments required a hundred processors during about 70 hours on the Grid5000 testbed detailed in section 4.2. We

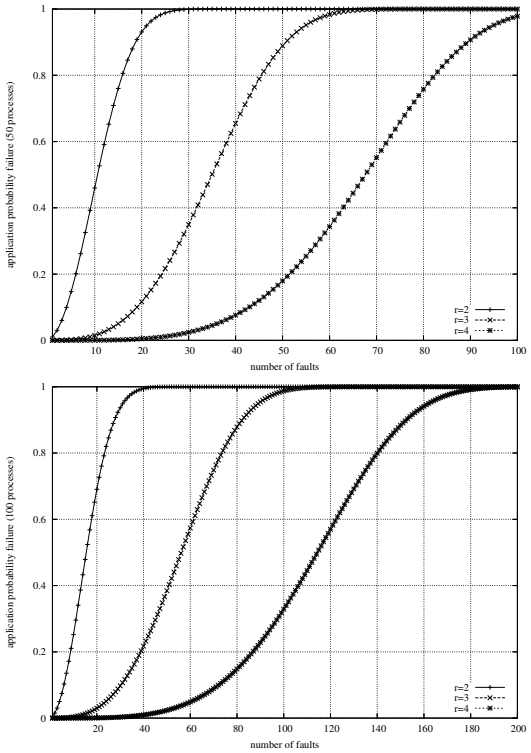

Figure 4. Probability that an application with 50 processes and 100 processes fails depending on the number of nodes and the replication degree r.

present hereafter the experimental data concerning the fault resilience for a 50 processes application with replication degree $r = 2$ and $r = 3$.

One trial in an experiment consists in killing every 50 seconds (a period long enough to make sure all processes get aware of that failure) one process chosen randomly out of the remaining application processes, until the application crashes. When the application crashes, we report the number of faults that led to the failure. The trials have been iterated a 100 times for each replication degree. Thus, the trial can be considered a random variable $\bar{X}$ whose values are the "numbers of faults leading to a crash".

In Figures 5 and 6, we plot on the left sides all the outcomes from trials. Each graduation on the horizontal axis corresponds to one trial, and the corresponding cross on the vertical axis indicates how many faults led to the application failure for that trial. We compute the sample mean $\mu_{\bar{X}}$ and standard deviation $\sigma_{\bar{X}}$ and we plot $\mu_{\bar{X}}$ on the figure as the horizontal continuous line, while $\mu_{\bar{X}} - \sigma_{\bar{X}}$ and $\mu_{\bar{X}} + \sigma_{\bar{X}}$ are represented as dashed lines.

In order to compare these data with the theoretical count, we are interested in the probability of the events "the application has failed after k faults" in the sample, noted $Pr(\bar{X} \leq k)$. If we let F_k the number of outcomes where the application failed with exactly k faults, $Pr(\bar{X} \leq k) = (F_1 + F_2 + \ldots + F_k)/N$, where N is the sample size. In our case, the above probabilities are well summarized by computing the percentile for each k in the sample. For instance in the sample of Figure 5, the event "the application has failed after 5 faults" is at the 16th percentile, that is in the lowest 16% of the dataset sorted ascending on fault numbers. On the right handsides of Figures 5 and 6 are superimposed the percentiles and the curves obtained from the theoretical count. From this graphical point of view, we see that our experimental data have a distribution that closely follows the model.

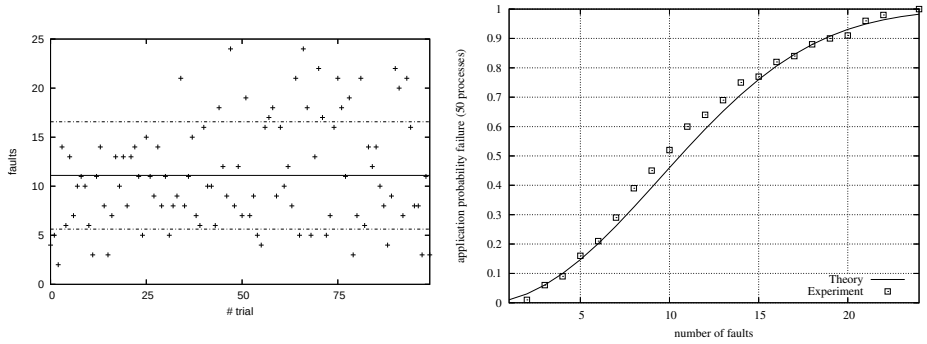

Figure 5. Number of faults leading to a crash for 100 experiments ($p = 50, r = 2$) (left), Comparison of experimental numbers of faults with theoretical count (right)

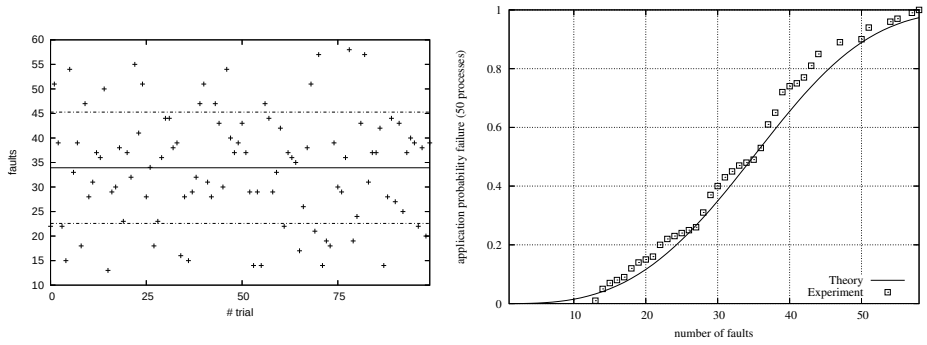

Figure 6. Number of faults leading to a crash for 100 experiments ($p = 50, r = 3$) (left), Comparison of experimental numbers of faults with theoretical count (right)

From a statistic point of view, we can state that the two samples follow our theoretical distribution thanks to a χ^2 test [12]. We test the hypothesis H_0: the number of faults observed is equivalent to the one predicted by the theoretical distribution. The variable called χ^2 measures the distance between an observed distribution and a theoretically expected one. In our case, we obtain $\chi^2 = 23.19$ and $\chi^2 = 78.34$ for the first and second experiment respectively. If we choose a typical confidence level of 95%, the previous values are well below the critic probabilities (40.11 with 27 degrees of freedom and 95.08 with 74 degrees of freedom, for experiments 1 and 2 respectively). Hence the H_0 hypothesis must not be rejected.

4. Experiments

The objective of the experiments is to study how P2P-MPI behaves in terms of performances. The experiments test communication and computation performances, how applications scale, and the impact of replication on performance.

These tests are carried out on a limited set of hardware platforms where we can reproduce the experiments. These are a) a student computer room in experiment 1, and b) a part of the Grid5000 testbed (detailed later) in experiment 2.

We do not have tests in highly heterogeneous environments such as the usual situation where nodes are dynamically chosen from PCs around. A precise assessment of P2P-MPI 's behavior on such configurations is difficult because experiments are not easily reproducible. However, the two configurations chosen for the experiments may also reflect real situations.

4.1. Experiment 1

In the first experiment, we measure the gap between P2P-MPI and some reference MPI implementations on a limited set of PCs. The hardware platform used is a student computers room (24 Intel P4 3GHz, 512MB RAM, 100 Mbps Ethernet, Linux kernel 2.6.10). We compare P2P-MPI using java J2SE-5.0, JXTA 2.3.3 to MPICH-1.2.6 (p4 device) and LAM/MPI-7.1.1 (both compiled with gcc/g77-3.4.3). We have chosen two test programs with opposite characteristics from the NAS benchmarks [3] (NPB3.2)[6]. The first one is IS (Integer Sorting) which involves a lot of communications since a sequence of one

[6] We have translated IS and EP in java for P2P-MPI from C and Fortran respectively.

`MPI_Allreduce`, `MPI_Alltoall` and `MPI_Alltoallv` occurs at each iteration. The second program is EP (Embarrassingly Parallel). It does independent computations with a final collective communication. Thus, this problem is closer to the class of applications usually deployed on computational grids.

4.1.1. *Expected Behavior*

It is expected that our prototype achieves its goals at the expenses of an overhead incurred by several factors. First the robustness requires extra-communications: regular heart-beats are exchanged, and the number of message copies increase linearly with the replication degree as can be seen on Figure 3. Secondly, compared to fine-tuned optimizations of communications of MPI implementation (e.g. in MPICH-1.2.6 [27] use four different algorithms depending on message size), P2P-MPI has simpler optimizations (e.g. binomial trees). Last, the use of a virtual machine (java) instead of processor native code leads to slower computations.

4.1.2. *Performances*

Figure 7 plots results for benchmarks IS (left) and EP (right) with replication degree 1. We have kept the same timers as in the original benchmarks. Values plotted are the average total execution time. For each benchmark, we have chosen two problem sizes (called class A and B) with a varying number of processors. Note that IS requires that the number of processors be a power of two and we could not go beyond 16 PCs.

For IS, P2P-MPI shows an almost as good performance as LAM/MPI up to 16 processors. The heart-beat messages seem to have a negligible effect on overall communication times. Surprisingly, MPICH-1.2.6 is significantly slower on this platform despite the sophisticated optimization of collective communications (e.g. uses four different algorithms depending on message size for `MPI_Alltoall`). It appears that the `MPI_Alltoallv` instruction is responsible for most of the communication time because it has not been optimized as well as the other collective operations.

The EP benchmark clearly shows that P2P-MPI is slower for computations because it uses Java. In this test, we are always twice as slow as EP programs using Fortran. EP does independent computations with a final set of three `MPI_Allreduce` communications to exchange results in short messages of constant size. When the number of processors increases, the share of computations assigned to each processor decreases, which makes the P2P-MPI performance curve tends to approach LAM and MPICH ones.

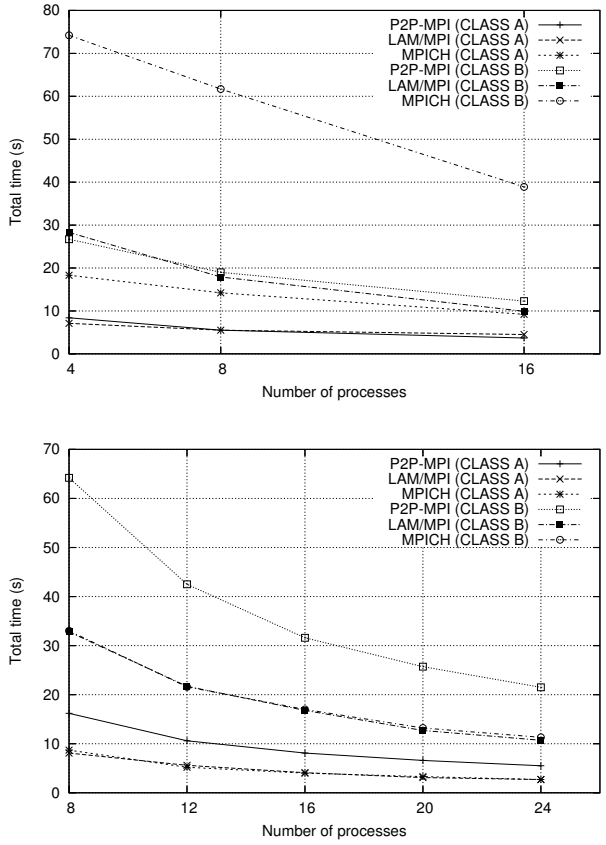

Figure 7. Comparison of MPI implementations performance for IS (left) and EP (right).

4.1.3. *Replication overhead*

Since replication multiplies communications, the EP test shows very little difference with or without replication, and we only report measures for IS. Figure 8 shows the performances of P2P-MPI for IS when each logical process has one to four replicas. For example, for curve "Class B, 8 processors", 3 replicas per logical process means 24 processors were involved. We have limited the number of logical processes so that we have at most one replica per processor to avoid load-imbalance or communications bottlenecks. As expected, the figure shows a linear increase of execution time with the replication degree, with a slope depending on the number of processors and messages sizes.

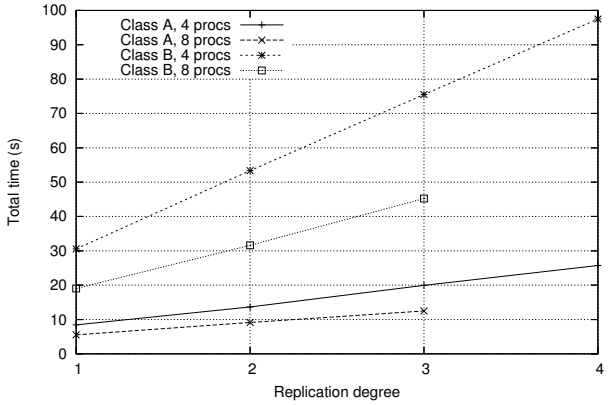

Figure 8. Performance (seconds) for IS depending on replication degree.

4.2. Experiment 2

In this second experiment we test the ability of P2P-MPI to handle an application using up to 128 processes, and the potential scalability of the application. The scalability test also analyzes the impact of having processors on distant sites.

The experiment takes place on Grid5000[7], a testbed designed as a tool for experimentation. By the end of 2005, it will allow a user to reserve a number of nodes across several geographical sites and re-install a full system on each of the nodes. Currently under construction, the testbed will eventually gather 5000 processors across nine sites in France through a virtual private network on the research and education network Renater. Two thirds of the processors are homogeneous (Opteron) while the remaining third may be different (currently Xeon, Itanium 2 and G5). Though not yet fully functional (only a part of the processors are currently available and the multi-site reservation is not implemented yet) the testbed allowed us to test P2P-MPI with a number of processors we would not be able to gather otherwise.

4.2.1. *Performances*

The application we choose is the *ray-tracer* from the Java Grande Forum MPJ Benchmark because it was reported [6] to scale well with MPJ/Ibis. This program renders a 3D scene of 64 spheres into an image of 150x150 or 500x500 pixels in the original benchmark, but we have enlarged the image resolution to 2500x2500. Each process does local

[7] http://www.grid5000.fr

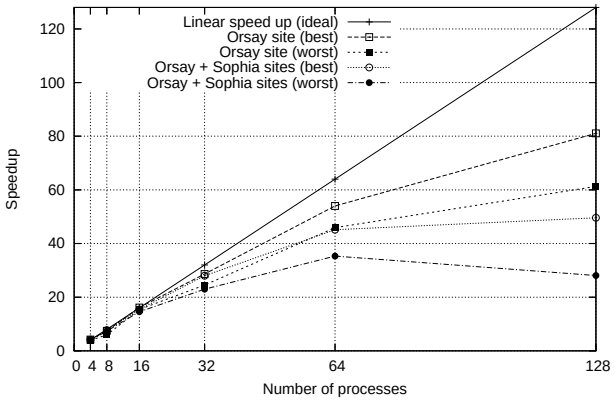

Figure 9. Ray-tracer speedups when run on a single site and on two distant sites.

computations on its part of the scene for which a checksum is combined with a `MPI_Reduce` operation by the rank 0 process. In the end, each process sends its result to process 0.

In the experiment we have run several series of executions of the application on different days of a week. A series consists in a set of executions using from 2 to 128 processes (one process per processor). We choose two sites of Grid5000 with homogeneous processors (AMD Opteron 2GHz, 2GB RAM) to isolate the impact of communications. The sites are Orsay located near Paris and Sophia near Nice, and are bound with a 2.5 Gbps network link.

We observe how the application scales on a single site (all processes at Orsay) and then when processes are distributed half on each site. We report on figure 9 the highest and lowest speedups obtained in each case. The application scales well up to 64 processors on a single site, and in some occasions the execution involving the two sites is even as quick as the lowest execution on one site. With 128 processors, the scalability largely decreases on one site, and turns to a slowdown with distant sites. We reach here a computation to communication ratio that does not allow for more parallelism.

However, the experiment confirms the good scalability of the application provided the image to compute is big enough, even when two distant sites are involved. We should extend the number of sites in future experiments to better understand what platforms are suitable for such applications.

5. Related Work

P2P-MPI 's features are linked to several issues that have been addressed previously: fault-tolerance using check-pointing or replication, parallel programming with java, and resource discovery and coordination.

5.1. Fault-tolerance

The deployment of message passing applications in unstable environments is a challenging area and numerous attempts have been made to tackle this problem. Most works are devoted to check-point and restart methods in which the application is restarted from a given recorded state.

The first such attempt has been the Co-Check [26] framework, which extends the single process checkpoint of Condor [20] to a distributed parallel application. The consistency of a saved state is insured by flushing all in flight messages before a check-point is created. However by saving the whole process states, Co-Check suffers a large overhead. Further, the flushing of messages makes the check-pointing synchronous with a potential scaling problem.

The Starfish [2] project also provides check-pointing facilities through the Ensemble execution environment Starfish is built upon. Starfish provides an event model where processes register to be notified of changes in cluster configuration and nodes failures. When a failure is detected, the system can automatically recover the application from a previous check-point. Flushing message queues before check-points is avoided thanks to the atomic communication primitives provided by the Ensemble system. However, consistency of communicators is not addressed and any node failure implies to restart the entire MPI application to recover from a single failed process.

The MPI-FT project [19] supports the development of master-slave like applications. During execution a monitoring process called observer, stores a copy of each sent message. The observer is notified when a node has been idle too long and then spawns a new process whose state is rebuild from previous messages. The drawback is the memory needed for the observer process in long running applications, as well as the bottleneck it induces. Also, the assumption that the state can be rebuilt using just the knowledge of any passed messages does not hold in the general case (e.g. non deterministic convergence of a solver).

The MPICH-V1 [7] library developed at University of Paris Sud propose an original logging protocol storing all communications on a

reliable media (called channel memory) associated with each process. After a crash, a re-executed process retrieves all lost receptions in the correct order by requesting them to its associated channel memory. The logging has however a major impact on the performance (bandwidth divided by 2) and requires a large number of channel memories. MPICH-V2 [8] improves V1 by splitting into two parts the logging: 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.

Apart from these project based on check-pointing, some other approaches represent alternatives which does not require any specific reliable resource to store system states.

FT-MPI [14], actively developed at University of Tennessee, proposes an original approach since it is up to the application programmer to call the fault-tolerance services to manage application's behavior in case of failure. When a fault occurs, all MPI processes of the communicator are informed about the fault through the returning value of MPI calls, and specific actions can be taken by the application to restore the communicator in a coherent state and then to treat the error. The main advantage of FT-MPI is its performance since it does not checkpoint nor log and let the application take the minimum number of actions needed for recovery, but its main drawback is the lack of transparency for the programmer.

MPI/FT [5] is the project closest to our proposal since it provides fault-tolerance by introducing replication of processes. The library can detect erroneous messages by introducing a vote algorithm among the replicas and can survive process failures. The drawback of this project is the increasing resource requirement by replicating MPI processes but this drawback can be overcome by using large platforms.

In comparison, we designed fault-tolerance in P2P-MPI so that it is totally transparent to the programmer and does not break the peer-to-peer paradigm in which all hosts play the same role. In consequence, we cannot rely on particular hosts considered reliable and used in check-pointing. As we do not check-point, we do not replace failed processes. Our choice is to tolerate a certain number of faults, postponing an eventual application failure.

5.2. MPI and Java

Java's ability to run on several platforms without re-compiling has raised its attractivity in today's heterogeneous environments and has motivated several attempts to adapt the MPI specification to Java.

The early JavaMPI [21] and mpiJava [4] projects were implemented as a set of JNI wrappers to native MPI packages. (It requires a native MPI library to be installed aside). The communication are hence almost as efficient as with the underlying MPI library, but the presence of an MPI library and system-dependent generated bindings reduce the portability advantage of Java programs. Later, MPJ [11] has been adopted by the Java Grande forum as a common MPI language bindings to Java, merging the above proposals. However, we know of very few MPJ implementations: mpiJava claims to move towards this specification, and recently, an implementation in pure Java from Vrije University [6] called MPJ/Ibis has been proposed. This implementation benefits from the efficient and various network drivers of the Ibis platform upon which MPJ is built. In comparison, P2P-MPI implements native Java TCP communications only since clusters or supercomputers with specific network devices (e.g. Myrinet) was not a goal. We rather focus on commodity hardware. The subset of MPJ we implement is growing regularly[8]and has shown to perform well in the experiments.

5.3. Resource discovery and coordination

To our knowledge, P3 (Personal Power Plant) [25] is the only project that really compares to P2P-MPI as far as resource discovery and coordination is concerned. The two projects share the goal to dynamically gather computational resources without any centralized directory, in order to execute a message-passing parallel application. In P3, JXTA is also used for discovery: *hosts* entities automatically register in a peer group of workers and accept work requests according to the resource owner policy. Secondly, a master-worker and message passing paradigm, both built upon a common library called *object passing*, are proposed. Unlike P2P-MPI, P3 uses JXTA for its communications (JxtaSockets). This allows to communicate without consideration of the underlying network constraints (e.g. firewalls) but incurs performance overhead when the logical route established goes through several peers. In addition, P3 has no integrated fault-tolerance mechanism for message passing programs which we believe is important for long executions.

[8] See http://grid.u-strasbg.fr/p2pmpi/documentation/javadoc/

6. Conclusion and Future Work

This paper has introduced a new middleware called P2P-MPI. It has been designed to facilitate parallel programs deployment on grids by addressing the major issues that are currently real obstacles to grids usability. To the best of our knowledge, no other project brings simultaneously: (a) an automatic discovery of resources, (b) a message-passing programming model, allowing any type of parallel program to be developed and run, and (c) a fault-detection system and replication mechanisms in order to increase robustness of applications execution.

The paper has discussed in details the design of each these features. We have analyzed how the failure probability of applications evolves with the replication degree and we have discussed further these probabilities in real situations. It was also important to understand how P2P-MPI behaves in terms of performance. Two NAS parallel benchmarks with opposite behavior have been run on a small configuration and compared with performances obtained with LAM/MPI and MPICH. We have also tested a ray-tracing application with up to 128 processors, distributed on one site and then across two sites of the Grid5000 testbed to understand the impact of wide area communications. On the whole, the performances obtained are close to what we obtain with the other message passing libraries we referred to. Hence, we believe P2P-MPI is a good tool for more experiments on various categories of applications.

Next developments should concern strategies for mapping processes onto resources. Though the peer-to-peer model abstracts the network topology, we could use some network metrics (e.g. ping time) to choose among available resources. Also, the mapping of replicas could be based on information about resources capability (e.g. CPU type, number of jobs accepted) and reliability (e.g. average uptime).

References

1. 'JXTA'. `http://www.jxta.org`.
2. Agbaria, A. and R. Friedman: 1999, 'Starfish: Fault-Tolerant Dynamic MPI programs on Clusters of Workstations'. In: *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*. Los Alamitos, California, pp. 167–176.
3. Bailey, D. H., E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga: 1991, 'The NAS Parallel Benchmarks'. *The Intl. Journal of Supercomputer Applications* **5**(3), 63–73.

4. Baker, M., B. Carpenter, G. Fox, S. H. Ko, and X. Li: 1998, 'mpiJava: A Java interface to MPI'. In: *First UK Workshop on Java for High Performance Network Computing*.
5. Batchu, R., J. Neelamegam, Z. Cui, M. Beddhua, A.Skjellum, Y. Dandass, and M. Apte: 2001, 'MPI/FT^{TM}: Architecture and Taxonomies for Fault-Tolerant, Message-Passing Middleware for Performance-Portable Parallel'. In: *Proceedings of the 1st IEEE International Symposium of Cluster Computing and the Grid*. Melbourne, Australia.
6. Bornemann, M., R. V. van Nieuwpoort, and T. Kielmann: 2005, 'MPJ/Ibis: a Flexible and Efficient Message Passing Platform for Java'. In: *Euro PVM/MPI 2005*.
7. Bosilca, G., A. Bouteiller, F. Cappello, S. Djailali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov: 2002, 'MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes'. In: *SuperComputing 2002*. Baltimore, USA.
8. Bouteiller, A., F. Cappello, T. Hérault, G. Krawezik, P. Lemarinier, and F. Magniette: 2003, 'MPIch-V2: a Fault Tolerant MPI for Volatile Nodes based on the Pessimistic Sender Based Message Logging'. In: *SuperComputing 2003*. Phoenix USA.
9. Budhiraja, N., F. Schneider, S. Toueg, and K. Marzullo: 1993, *The Primary-Backup Approach. In S. Mullender, Distributed Systems*, Chapt. 8, pp. 199–216. Addison Wesley.
10. Caron, E., F. Deprez, F. Frédéric Lombard, J.-M. Nicod, M. Quinson, and F. Suter: 2002, 'A Scalable Approach to Network Enabled Servers'. In: *8th EuroPar Conference*, Vol. 2400 of *LNCS*. pp. 907–910.
11. Carpenter, B., V. Getov, G. Judd, T. Skjellum, and G. Fox: 2000, 'MPJ: MPI-like Message Passing for Java'. *Concurrency: Practice and Experience* **12**(11).
12. Chase, W. and F. Brown: 1992, *General Statistics*. Wiley, 2nd edition.
13. F. Schneider: 1993, *Replication Management Using State-Machine Approach. In S. Mullender, Distributed Systems*, Chapt. 7, pp. 169–198. Addison Wesley.
14. Fagg, G. E., A. Bukovsky, and J. J. Dongarra: 2001, 'Harness and fault tolerant MPI'. *Parallel Computing* (27), 1479–1495.
15. Fedak, G., C. Germain, V. Néri, and F. Cappello: 2001, 'XtremWeb : A Generic Global Computing System'. In: *CCGRID*. pp. 582–587.
16. Gabriel, E., M. Resch, T. Beisel, and R. Keller: 1998, 'Distributed Computing in an Heterogeneous Computing Environment'. In: *EuroPVM/MPI*.
17. Karonis, N. T., B. T. Toonen, and I. Foster: 2003, 'MPICH-G2: A Grid-enabled implementation of the Message Passing Interface'. *Journal of Parallel and Distributed Computing, special issue on Computational Grids* **63**(5), 551–563.
18. Kielmann, T., R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang: 1999, 'MagPIe: MPI's collective communication operations for clustered wide area systems'. *ACM SIGPLAN Notices* **34**(8), 131–140.
19. Louca, S., N. Neophytou, A. Lachanas, and P. Evripidou: 2000, 'MPI-FT: Portable Fault Tolerenace Scheme for MPI'. *Parallel Processing Letters* **10**(4), 371–382.
20. M. Litzkow et al: 1998, 'Condor: A Hunter of Idle Workstations'. In: *Proceeding of the 8th International Conference on Distributed Computing Systems*. Los Alamitos, California, pp. 104–111.
21. Mintchev, S. and V. Getov: 1997, 'Towards portable message passing in Java: Binding MPI'. In: *Recent Advances in PVM and MPI*, Vol. 1332 of *LNCS*.

22. MPI Forum: 1995, 'MPI: A Message Passing Interface Standard'. Technical report, University of Tennessee, Knoxville, TN, USA.
23. P. Felber and X. Defago and R. Guerraoui and P. Oser: 1999, 'Failure detectors as first class objects'. In: *Proceeding of the 9th IEEE Intl. Symposium on Distributed Objects and Applications (DOA'99)*. pp. 132–141.
24. Renesse, R. V., Y. Minsky, and M. Hayden: 1998, 'A Gossip-Style Failure Detection Service'. Technical report, Ithaca, NY, USA.
25. Shudo, K., Y. Tanaka, and S. Sekiguchi: 2005, 'P3: P2P-based Middleware Enabling Transfer and Aggregation of Computational Resource'. In: *5th Intl. Workshop on Global and Peer-to-Peer Computing, in conjunc. with CCGrid05*.
26. Stellner, G.: 1996, 'CoCheck: Checkpointing and Process Migration for MPI'. In: *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*. Honolulu, Hawaii.
27. Thakur, R., R. Rabenseifner, and W. Gropp: 2005, 'Optimization of Collective Communication Operation in MPICH'. *International Journal of High Performance Computing Applications* **19**(1), 49–66.
28. Verbeke, J., N. Nadgir, G. Ruetsch, and I. Sharapov: 2002, 'Framework for Peer-to-Peer Distributed Computing in a Heterogeneous, Decentralized Environment'. In: *GRID 2002*, Vol. 2536 of *LNCS*. pp. 1–12.