A Peer-to-Peer Framework for Message Passing Parallel Programs

Stéphane Genaud, Choopan Rattanapoka

A Peer-to-Peer Framework for Message Passing Parallel Programs

Stéphane GENAUD [a,1], and Choopan RATTANAPOKA [b,2]
[a] *AlGorille Team - LORIA*
[b] *King Mongkut's University of Technology*

Abstract. This chapter describes the P2P-MPI project, a software framework aimed at the development of message-passing programs in large scale distributed networks of computers. Our goal is to provide a light-weight, self-contained software package that requires minimum effort to use and maintain. P2P-MPI relies on three features to reach this goal: i) its installation and use does not require administrator privileges, ii) available resources are discovered and selected for a computation without intervention from the user, iii) program executions can be fault-tolerant on user demand, in a completely transparent fashion (no checkpoint server to configure). P2P-MPI is typically suited for organizations having spare individual computers linked by a high speed network, possibly running heterogeneous operating systems, and having Java applications. From a technical point of view, the framework has three layers: an infrastructure management layer at the bottom, a middleware layer containing the services, and the communication layer implementing an MPJ (Message Passing for Java) communication library at the top. We explain the design and the implementation of these layers, and we discuss the allocation strategy based on network locality to the submitter. Allocation experiments of more than five hundreds peers are presented to validate the implementation. We also present how fault-management issues have been tackled. First, the monitoring of the infrastructure itself is implemented through the use of failure detectors. We theoretically evaluate several candidate protocols for these detectors to motivate our choice for the gossiping protocol called binary round robin. A variant of this protocol is also proposed for a greater reliability. Finally, the system scalability and the theoretical findings are validated through experiments. The second aspect of fault management concerns program executions. Fault-tolerance is provided by the communication library through replication of processes. We describe the underlying protocol and the properties that need to be met in order to insure the correctness of execution. We then discuss how to choose the number of replicas by quantifying how much more robust is an application using replication, depending on the failure model parameters.

Keywords. Grid, Middleware, Java, Peer-to-peer, MPJ, Fault-tolerance

[1] Corresponding Author: AlGorille Team - LORIA
Campus Scientifique - BP 239
F-54506 Vandoeuvre-lès-Nancy, France
Stephane.Genaud@loria.fr
[2] Department of Electronics Engineering Technology
College of Industrial Technology
King Mongkut's University of Technology North Bangkok
Bangkok, Thailand
choopanr@kmutnb.ac.th

Introduction and Motivation

The concept of *Grid* [22] has emerged to express the unprecedented perspectives offered by a large number of computers arranged in a distributed telecommunication infrastructure. At the same time, the trend in computational science shows an increasing need for computational power. This makes parallel and distributed computing an unavoidable solution. Thus, grids have been seen these last years as a promising infrastructure to deploy massively parallel applications.

However, the efficient exploitation of shared heterogeneous resources (based on different platforms, hardware/software architectures) located in different places, belonging to different administrative domains over a network is still an issue. Numerous software projects aim at supporting grid computing, such as the Globus Toolkit [21], UNICORE [18], or Condor [44], to name a few among the most well-known projects. Yet, the parallel applications deployed are often much simpler than those ran on supercomputers. Most applications are structured as embarrassingly parallel programs or workflows. Further, grids build upon resources from established institutions are currently heavyweight and complex, and require a strong administrative support for users. The burden is especially unreasonable for occasional users or users submitting small jobs. This contradicts the initial metaphor for the grid being as easy as a power grid.

Simultaneously to the development of institutional grids, *volunteer computing* has gained much popularity. This trend is exemplified by the BOINC approach [2], which enables to build *Desktop* grids using computers typically owned by members of the general public (typically home-PCs). However, the class of applications that can benefit from such an approach is also limited to embarrassingly parallel problems.

With the same spirit of easing the access to grids, many projects have added techniques borrowed from peer-to-peer (P2P) systems. The main benefit is the absence of central server, which dramatically lessens the need for system administration. However, some issues are still unclear regarding the question of deploying parallel applications. What is the best design for the middleware to keep the desired ease of use, while being able to add new plug-ins features (scheduling strategies for example) ? Do general programming models (such as message passing) have any chance to succeed, or should we limit programming models to client-server, workflows or embarrassingly parallel models? To which extent can we expect message passing parallel programming to scale in such environments? How will the communication to computation ratio impact this scalability? Do there exist appropriate fault-tolerance fault-tolerance techniques for these faulty environments? Can middleware middleware systems tackle the heterogeneity of the software and hardware? How can we improve the ease of use while maintaining these features?

The contents of the chapter will describe the P2P-MPI project, and its contributions to the above questions. Before we present our contribution, Section 1 presents some related work in the major research fields connected to our project. Then comes the description of our contribution, that we have split into two main parts. The first part deals with the framework design, which targets a user-friendly and functional software. Inside this part, Section 2 gives a quick overview of our project's targets, and the middleware description is the object of Section 3. In the second part we address the unavoidable question of fault management. Two distinct issues are linked to fault management. First, in

Section 4, we propose a transparent mechanism to provide some fault-tolerance for programs being executed by the framework. Then, in Section 5, we discuss the techniques used to detect failures in an efficient, predictable and scalable fashion.

Each contribution is of a different nature, and therefore we will present an evaluation of the work inside each section itself. In the first part, we will present the various modules, with an emphasis on the P2P layer. We will describe how peers agree to form a group to execute a parallel program. The protocol involves the discovery and the reservation of resources, and executable and data files staging. We will detail further the reservation process as it is linked to scheduling: in our context of message passing programs, all the tasks making up the program have to be *co-allocated*, that is we must choose how to simultaneously assign the tasks to a set of resources. We will discuss the pre-defined allocation strategies proposed to the user. While both strategies use the closest resources in terms of network latency to the submitting computer, one will load all computing capabilities of selected computers, while the other spreads computations over selected hosts. These strategies allow to express the user's preference to exploit the multi-core architecture of a host or not. The evaluation will show the effects of strategies on real examples through experiments involving up to 600 processors on a nation-wide grid (Grid'5000 [12]).

In the second part, we will present the protocols proposed for fault-tolerance and fault detection. We propose process replication to increase the robustness of applications. After a description of the underlying protocol needed to insure the coherence of the system, we evaluate how much more reliable is a replicated application. Next, we review the gossiping protocols used for fault detection. Our contribution is an extension of the binary round robin protocol, and we evaluate its behavior through simulation. Finally, the fault detection protocols are evaluated in a real experiment involving 256 processes scattered over three sites of Grid'5000.

1. Related Work

P2P-MPI spans a number of topics that have emerged recently in the field of grid middleware. The major topics addressed by our work relate to the middleware design on a P2P basis, to fault-tolerance, and to the message passing library implementation.

1.1. P2P based middleware and scheduling

The advantages of an organization of the grid resources in a P2P fashion over an organization using a centralized directory are now widely accepted [23]. Some of the issues addressed by P2P systems are fault tolerance, ease of maintenance, and scalability in resource discovery. One important task of the middleware is to schedule jobs on the grid. The goal of scheduling can be to maximize the grid utilization or the performance of each individual application. In the former case, the metric is for instance the job throughput of the whole system, while for the latter case the metric may be the makespan of the application (finishing time of latest job) or the flowtime (sum of finishing times of jobs). However, most grid projects only target the individual application performance [31].

The constraints on the scheduling and the metrics used may however largely differ depending on whether the jobs are independent tasks, workflows, parallel programs, etc,

and depending on the knowledge the scheduler has about the jobs. In our context of parallel programs, all jobs are dependent one from the others. The scheduler must therefore *co-allocate* (i.e. allocate simultaneously) a set of resources to start the jobs simultaneously. Moreover, the job durations are generally not known, so it is not possible to compute an optimal scheduling. The scheduling decision in that case, may be based on extra criteria known to influence the targeted metric. For parallel programs, the objective is often to maximize the efficiency of the execution to minimize the makespan. Known factors influencing the performance of parallel programs executed in distributed environment are network locality and available bandwidth between hosts, load unbalance among processors or processor heterogeneity. Ideally, the scheduler should take all these factors into account to choose the resources to allocate. Yet, combining these factors is a difficult heuristic process. Further, much of the information is not known in a decentralized organization because individual peers have generally no global knowledge of all other peers' state, distance, etc.

However, the P2P network management software can be designed to provide to the scheduler some of this information. Let us review how other P2P based projects supporting message-passing parallel programs have addressed this concern. The early Power Plant P3 project [41] used JXTA [45] to build its P2P overlay network, and offered both a message-passing and a client-server oriented API. However, there was no control on the peers selected by the middleware system as those are returned by JXTA. Another example is the long-lived project ProActive [13], to which has been added a P2P infrastructure to ease resource discovery. This P2P infrastructure offers a discovery service to the ProActive system, allowing a manager task to dynamically acquire peers at the beginning or during a program execution. However, the infrastructure has no knowledge about the network topology and therefore the selection of peers used in a computation does not take into account network locality. Very close to our work are Zorilla [17] (a part of the Ibis project, see below) and Vigne [32]. They are two middleware systems which also build a P2P overlay network aware of peer locality. For that purpose, Vigne uses algorithms from the Bamboo project [38]. In Vigne, close resources are found using a simple (yet sometimes misleading) heuristic based on DNS name affinity: hosts sharing a common domain name are considered as forming a local group. Zorilla (which also uses Bamboo) proposes *flood scheduling*: the co-allocation request originated at a peer is broadcasted to all its neighbors, which in turn broadcast to their neighbors until the depth of the request has reached a given radius. If not enough peers accepted the job, new flooding steps are successively performed with an increasing radius until the number of peers is reached. The difficulty in this strategy, lies in finding suitable values for the flooding parameters, such as the radius and minimum delays between floods.

1.2. Fault management

As will be detailed in Section 4 and 5, fault-management includes two distinct research fields. The first is fault-detection. Several works have addressed the problem of detecting in an efficient and scalable way faults in a distributed system. We will describe the principle of *gossiping*, which is a very efficient technique to tackle this issue.

The second field is fault recovery. With MPI programs, the approach used is almost always rollback and recovery. Rollback-recovery protocols have been studied extensively, and numerous projects have been proposed. The protocols are either based

on a *coordinated checkpoint* i.e, one coordinator process orders all processes to take a snapshot of their local state and then form a global checkpoint in order to recover from that point, or are based on *message logging* normally completed by asynchronous checkpoints. The coordinated checkpoint approach is very simple to implement but has a high overhead because of synchronizations and is not efficient: frequent checkpoints slow the execution, while infrequent checkpoints lead to a large loss of execution. The alternative message logging protocol stores non-deterministic events (e.g message arrivals) on a reliable media. When a process crashes, its state is restored by replaying the communications. In this family, the protocols falls in three categories [1]. In *pessimistic logging*, the reception of a message is locked until the received message has been stored on the reliable media. In case of failure, the process state recovery is straight-forward: the process is re-executed from its last-checkpoint, and further messages are replayed from the log. In *optimistic logging*, the message backups on the reliable media are asynchronous to gain performance. However, recovery is more complicated since a part of the execution between the last saved message and the failure might have affected other processes. Returning to the last consistent state for all processes may force the recovery to rollback up to the beginning of the execution (domino effect). *Causal* log protocols try to combine the advantages of the optimistic and the pessimistic approaches by piggybacking events to normal messages until these events are safely logged.

There have been implementations for many of these approaches. For example, the early CoCheck project [43], as well as the popular implementation LAM/MPI [39] have added fault-tolerance using coordinated checkpointing. In MPI-FT [33], all the traffic is buffered either by a monitoring process or by each individual process, following the pessimistic message log strategy. More recently, some research works have proposed mixed or alternatives strategies. MPICH-V adds fault tolerance to the MPICH implementation. The first version [9] is based on uncoordinated checkpointing with pessimistic message logging, but suffers a high overhead as logging to the reliable media divides the bandwidth by two. Moreover, it is preferable to dispatch messages over many reliable servers to avoid bottlenecks. This represents a non-trivial system constraint. The same authors have proposed MPICH-V2 [10], in which the message logging is split into two parts: the message data itself is stored on the computing node while the logical date and the identifier of the message are stored on the reliable server by the receiving process. The performance of MPICH-V2 is reported to get close to execution without fault-tolerance, except for small messages.

With a different perspective, FT-MPI [20] is a framework able to detect failures, and let the user code handle the possible failure cases. The application is informed, via the MPI primitive return code, of the error type and should take appropriate actions. An enriched set of primitives is provided to the programmer in order to react upon the failure. Actions focus on the communicator management (e.g shrinking the communicator when a dead process is detected). However, FT-MPI does not provide an API for fault notification and for checkpointing.

To the best of our knowledge, MPI/FT [6] is the only project that has proposed process replication to tackle fault-tolerance with MPI. MPI/FT is derived from the MPI/Pro implementation, and adds fault detection and fault tolerance features. Fault detection is implemented through extra self-checking threads, which monitor the execution by sending heartbeat messages or vote to reach a consensus about processes' states. Comparable to our approach, fault tolerance is based on process replication (termed *modular redun-*

dancy). Different strategies of replication are recommended depending on the application model (e.g master-slave, SPMD, ...) but anyhow, their protocol relies on a coordinator through which all messages (transparently) transit. This is a bottleneck that limits scalability.

Notice also that little attention has been paid to fault-tolerance in the context of wide area networks. In that respect, we only know the effort made to enable FT-MPI to work across several administrative domains using the H20 metacomputing framework [16].

1.3. Message Passing Library

The ease of use intended for P2P-MPI led us to develop its communication library in Java, for its "run everywhere" feature particularly suited to environments with heterogeneous operating systems. For a better integration, the message-passing programming model we offer should be in Java as well. MPI [42], the de-facto message passing standard library, with popular implementations such as MPICH [29] and OpenMPI [24], has no bindings defined for Java. However, an alternative recommendation called MPJ [14] MPJ has been proposed for Java. An example of MPJ program is presented in Figure 1. This is what we have chosen to implement in P2P-MPI. Among the few other MPJ implementations in "pure" Java (with no use of JNI) are MPJ Express [5] and MPJ/Ibis [8]. MPJ Express is an efficient implementation supporting communications over TCP or myrinet devices. The library however, is standalone and has no extra facility to deploy applications on grids. MPJ/Ibis from Vrije Universiteit in Amsterdam, relies on the Ibis [46] system. Ibis is a multi layer system, one of these being the Portability Layer (IPL). IPL provides an object-oriented interface to network communication primitives. Different programming models can be implemented above this layer, using the IPL interface. MPJ/Ibis is one of these programming models. Though MPJ/Ibis belongs to a much larger project than our integrated framework, we share the objective of keeping the use simple, even when targeting grids. To that aim, MPJ/Ibis can be used with Zorilla (see Section 1.1) to discover available computing resources. However, MPJ/Ibis has no support for fault-tolerance. In that respect, and to the best of our knowledge, no other MPJ implementation provide fault-tolerance features.

2. The P2P-MPI Approach

P2P-MPI's final goal is to allow the seamless execution of parallel programs in grid environments. The master word here, is *ease of use*. We have privileged a tightly integrated environment, programmed in Java only consisting in a single jar and a few scripts. The installation requires no administrator privilege. The development of parallel programs requires only a Java compiler and the jar file. In the following, we will see that the tight integration of software modules allows a user program to get a straight connection with the middleware layer to retrieve the information it needs.

Before discussing how the communication library interacts with the middleware, we give an overview of the whole architecture of P2P-MPI. The set of modules and functions that constitute P2P-MPI may conceptually be seen as a three layers stack (the three levels of grey in figure 2).

On top of the stack is the *communication library* which exposes an MPJ API. The communication library represents the execution model. The communication library relies

```
import p2pmpi.mpi.*;

public class Pi {
        public static void main(String[] args) {
        int rank, size, i;
        double PI25DT = 3.14159265358979323846264 3;
        double h, sum, x;
        MPI.Init(args);
        size = MPI.COMM_WORLD.Size();
        rank = MPI.COMM_WORLD.Rank();
        int[] n = new int[1];
        double[] mypi = new double[1];
        double[] pi   = new double[1];

        if(rank == 0)
           n[0] = 1000000; //number of intervals
        MPI.COMM_WORLD.Bcast(n, 0, 1, MPI.INT, 0);
        h = 1.0 / (double)n[0];
        sum = 0.0;
        for(i = rank + 1; i <= n[0]; i+= size) {
           x = h * ((double)i - 0.5);
           sum += (4.0/(1.0 + x*x));
        }
        mypi[0] = h * sum;
        MPI.COMM_WORLD.Reduce(mypi, 0, pi, 0, 1, MPI.DOUBLE, MPI.SUM, 0);
        if(rank == 0) {
           System.out.println("Pi is approximately " + pi[0]);
           System.out.println("Error is " + (pi[0] - PI25DT));
        }
        MPI.Finalize();
    }
}
```

Figure 1. An example MPJ program for computing an approximation of π

on a *middleware* layer which provides different services to the communication library through a set of modules. The services offered are the fault-detection service (FD), the file transfer service (FT), the reservation service (RS), and discovery service. For example, while a peer is running an application process, it may be notified by its fault-detection service of a failure on another peer. The discovery service is in charge of selecting resources to fulfill a user request, and is implemented by the multi-purpose daemon (MPD) module[3] This module relies itself on a lower layer that deals with the resource management. Resource management consists to attribute identifiers to resources, locate available resources, etc. We call this layer *infrastructure* because the way resources are managed strongly depends on how the resources can be located and reserved (e.g. through a central directory). As mentioned earlier, we assume that a P2P approach is best suited for our need. Let us discuss the role of the infrastructure layer in this context.

[3]This name is a reference to the MPD in the MPICH distribution.

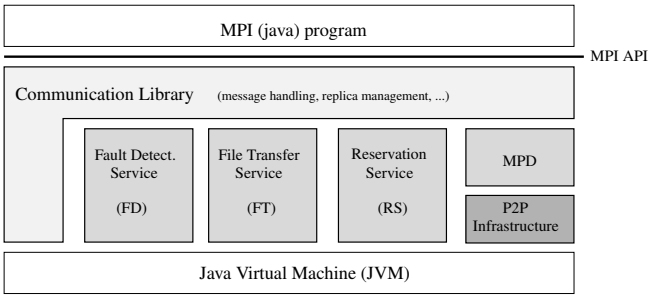


Figure 2. P2P-MPI structure.

3. A P2P based-middleware

3.1. The infrastructure layer

The role of the infrastructure management module is to maintain a local knowledge of the infrastructure. To implement this layer, we can use any software able to discover resources. For example, we initially used JXTA [45] to manage the P2P infrastructure layer. A peer in JXTA, advertizes its presence by means of an *advertisement* (a small XML message describing it). Advertisements are stored or retrieved from a distributed hash table maintained by special peers called *rendezvous*. Each rendezvous maintains a list of known other rendezvous, called the rendezvous Peer View (RPV). When a rendezvous is given an advertisement to store, it applies a hash function to determine to which rendezvous in the RVP the advertisement must be forwarded for storage. The lookup process requires the use of the same hash function to discover the rendezvous in charge of storing that advertisement. Another incentive to use JXTA is its ability to cross firewalls using *relay* peers.

However, JXTA suffers several pitfalls with respect to our requirements. First, there is no mechanism to enforce the consistency of all RPVs across the JXTA network, so a given RPV can have a temporary or permanent inconsistent view of the other rendezvous peers. As a result, the system can not guarantee to discover all existing advertisements in a given delay. The experiment [3] conducted in an environment similar to ours shows that the PRV consistency is always very limited. Second, crossing firewalls using the JXTA messaging system would involve bottlenecks at the relay peers, and hence make this feature of little use for high-performance applications. Third, JXTA does not account for network locality between peers, which is an important information to improve performance of message-passing oriented application. In replacement of JXTA, we have developed our own peer-to-peer infrastructure management system, which is simple, light, and fast. The benefits over JXTA in our context are the completeness and speed of resource discovery, and the network latencies we can capture.

From a user's point of view, there is barely no change, except that the rendezvous terminology of JXTA is replaced by the *supernode* concept. A supernode is a necessary entry point for boot-strapping a peer willing to join the overlay. The first action of a starting MPD is to connect to the supernode to register itself. The supernode keeps tracks

of all peer registrations or unregistrations, recording for each host, its services ports, and a "last seen" timestamp.

On first connection to the supernode, a MPD retrieves the list of known peers, and then maintains a local cache of this list. It then periodically contacts its supernode to update its cached list. A network latency value is associated to each host in the list. For that, each MPD periodically contacts hosts in the list and measures the round-trip time (RTT) of an empty message sent to it. Notice that this "ping" test is a standard P2P-MPI communication and does not rely on an ICMP echo measurement, such as ping system command. This approach would involve portability issues. It could also fail because the ICMP traffic is often blocked or limited by firewalls.

The current implementation has a single supernode since it is not our primary concern to demonstrate the scalability of the P2P infrastructure. The extension of the system to a distributed set of supernodes is left to a future work. However, from our experience involving up to 600 peers, the single supernode is not throttled by requests because the peers use most of the time information they have cached locally.

3.2. The middleware layer

The role of the middleware layer is to manage the program needs depending on the user's request, by provisioning, allocating a proper set of resources, and then monitoring the job execution. Let us illustrate how modules of the layer cooperate to achieve to start a parallel program execution. The program locally starts and requests the middleware module to find some other resources to run all processes in parallel. Here, the middleware task is to build a temporary set of processors, which will make up the initial *communicator* (MPI_COMM_WORLD). The communicator in MPI is an opaque object containing the necessary information for a process to contact any other in the same communicator (processes are identified by their *rank* in the communicator). A communicator may be seen as the "universe" in which a point-to-point or collective operation is to operate during program execution. The initial communicator must be known at each process. Building this communicator in our framework requires a number of steps illustrated on Figurer 3 and described below.

(1) **Booting up:** The user must first join the P2P-MPI platform by typing the command `mpiboot`, which starts the local background daemons MPD, FT, FD, and RS. MPD makes the computer join the P2P-MPI network and represents the local computer as a peer as long as it runs.

(2) **Job submission:** The job is then submitted by invoking `p2pmpirun -n` n `-r` r `-a` $alloc$ `prog`. The mandatory arguments are the n processes requested to run the `prog` program. The other arguments are optional: r is the replication degree used to request some fault tolerance (explained later), and $alloc$ tells the MPD which strategy must govern the allocation of the n processes on available resources (explained later). Then, it will start the process with rank 0 of the MPI application on local host. We call this process the *root process*.

(3) **Requesting Peers:** The application contacts its local MPD to discover enough nodes to have the capacity to execute a job of $n \times r$ processes.

(4) **Discovery and Reservation:** the local MPD selects a subset of the peers it already knows, and issues a reservation request to them via the local RS. The local RS then asks in turn to each remote RS to reserve the corresponding resource.

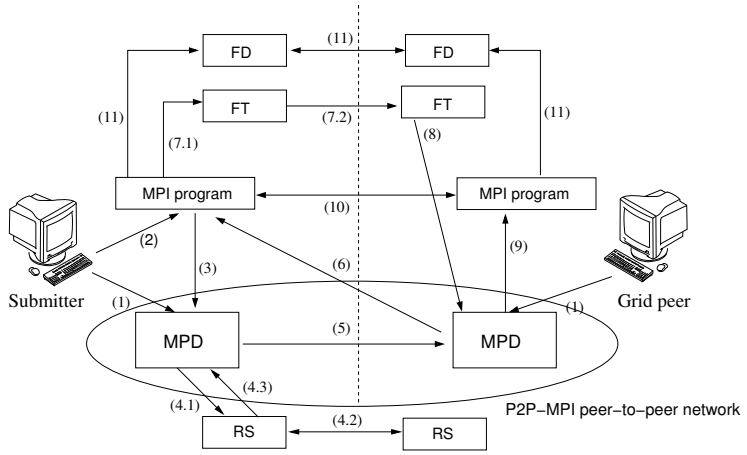


Figure 3. Steps taken to build an MPJ communicator mapped to several peers.

The local RS gathers these reservation results and returns them to the MPD. In case not enough peers are found or reserved, the MPD may initiate a new query to the supernode to know more peers, and iterate the reservation process with the new peers. If not enough new peers are returned by the supernode, the MPD aborts the request.

(5) **Registering:** After the reservation is done, the local MPD directly contacts the reserved nodes' MPDs, by sending them the application name, its MPI rank regarding the application to spawn, and the IP and port of the root process for the MPI application to be able to contact it. The application can then form its MPI communicator.

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

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

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

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

(10) **Execution preamble:** the spawn processes give their rank, IP and application port to the root process. Then, the root process creates the rank to IP address mapping communication table called `communicator`. Finally, the root process sends the communicator to all other processes.

(11) **Fault detection:** MPI processes register to 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.

In the previous section, we have enumerated the steps taken to start a parallel application. Among these, step (4) hides the complex scheduling process, that is choosing where and when the processes execute. In the following, we explain only the problemat-

ics of the resource selection and how P2P-MPI tackles the issue. The reader is referred to [28] for details about the algorithms used.

As mentioned in the related work section, there is no freedom for *when* the tasks execute because an MPJ program requires its processes to be started simultaneously. Moreover, we do not support postponed execution so it is scheduled as soon as possible. Yet, choosing a "good" set of resources is not straight-forward.

First, in a decentralized and multi-user system, it is not possible to get an instantaneous information about resource states. It is necessary to query each resource during the co-allocation process to get an up-to-date information. This inspection first reveals if the peer is still alive, and if its dynamic state is compatible with the request. This task is the purpose of the Reservation Service (RS). The local RS module contacts a remote RS module, which then behaves as a gatekeeper of the resource. It interprets the owner preferences, expressed in the configuration file, which may for instance allow or disallow such or such other peers. The preferences also concern the way the CPU is shared, and are expressed through two settings: the number J of different applications that a node can accept to run simultaneously, and the number P of processes per MPI application that a node can accept to run. For instance, J=2 and P=1 would allow two distinct users to run simultaneously one process each for their respective applications. J=1 and P=2 would allow to simultaneously run two processes of a single application (this setting is often used for dual-core CPUs). When a peer accepts to participate in a execution, the local RS locks the resource by issuing a reservation token to the remote RS, until a final decision is made about its participation in the execution. As some peers may not be selected to participate, we use overbooking. Eventually, if more than the $n \times r$ processes requested in step (3) agree to participate, we cancel extra reservations.

The second issue deals with selecting the most adequate resources. In P2P-MPI, we take into account two criteria: network locality and memory access contention. It is well known that an MPI application benefits from locality of allocated resources since it minimizes the communication costs. As multicore CPUs are becoming the most common type of processor, an option would be to favor the allocation of processes on all cores of available multicores to increase process locality. However, this strategy decreases the amount of memory available to each process mapped on a same multicore. We think the user knows its application's requirements and should advice the middleware of its specific needs. Therefore, we propose simple and understandable strategies to the user. When requesting an execution, the user can choose on the command line:

- the *spread* strategy, which maps as few processes as possible on each host (hence maximizing the available memory per process when processors share the memory), while maintaining locality as a secondary objective. The algorithm assigns one process per host in the list of selected peers, sorted by increasing latency. If the list is exhausted, processes are mapped round-robin from the beginning of the list (first host will receive a second process, etc).
- the *concentrate* strategy, which increases locality between processes by using as many cores as hosts offer. The algorithm assigns as many processes as possible to the first peer (with respect to its capacity P) in the list sorted by increasing latency. It then continues with next peers in the list, until all processes have been mapped.

3.3. The communication library layer

P2P-MPI provides to the programmer a communication library implementing the MPJ recommendation [14]. The implementation supports TCP network devices only, but comes into two flavors corresponding to two different objectives. Initially, we mainly targeted large scale environments with resources scattered over several domains, and the objective was to be competitive with other communication models such as RMI for example. Thus, we developed a communication library solely based on Java TCP sockets, in which connections are opened one at a time, so that a single open port is required. We call this implementation *single-port*. This implementation is well adapted to environments where the security policy imposes many port opening restrictions. Recently, we have achieved a new implementation which assumes no restriction on open ports. This allows us to use as many sockets as needed to speedup communications. We rely on the Java `nio` class, which provides the equivalent of the C *select* operation, allowing to monitor multiple file descriptors concurrently. This new implementation is called *multiple-ports*. Both implementations use well-known algorithms to optimize collective communications. The discussion about such optimization techniques is out of our scope, and the reader is referred to [37] for details about the algorithms used. We only summarize here (Table 1) which algorithms are used.

Operation	Algorithm	Operation	Algorithm
`Allgather`	Gather then Bcast	`Gather`	Flat tree
`Allgatherv`	Gatherv then Bcast	`Gatherv`	Flat tree
`Allreduce`	Butterfly[35] or Reduce then Bcast	`Reduce`	Binomial tree or flat tree
`Alltoall`	Asynchronous rotation	`Reduce_scatter`	Reduce then Scatterv
`Alltoallv`	Asynchronous rotation	`Scatter`	Flat tree
`Barrier`	4-ary tree	`Scatterv`	Flat tree
`Bcast`	Binomial tree		

Table 1. Algorithms currently implemented for collective communications

An other important issue is related to fault-tolerance. Numerous works have addressed this problem for message passing systems, as reported in Section 1. Most approaches are based on check-point and restart, which rely on a common network file system or dedicated checkpoint servers. As the presence of central servers does not fit into our P2P framework, we propose a different approach based on *replication* of computations. The communication layer must therefore integrate all the management operations required to handle replication transparently. This is explained in the following Section 4. The layer must also cooperate with the failure detection service in the middleware layer, and we will see how faults are detected in Section 5.

3.4. Evaluation of Allocation and Performance

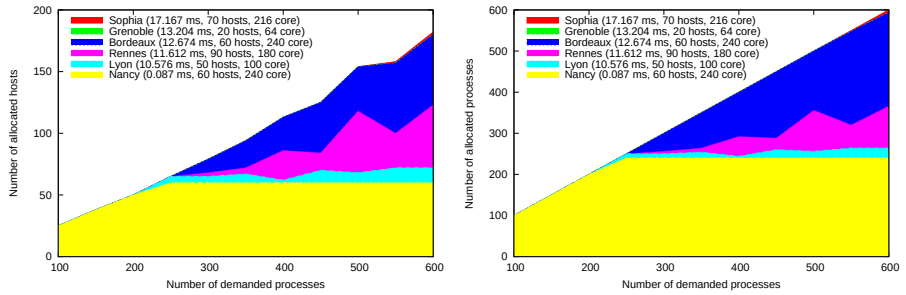
Allocation Strategies We evaluate the effectiveness of the allocation strategies at a large scale, and then its impact on performance of the communication library, in an experiment on the grid testbed Grid5000. The computers in our experiment are taken from eight clusters located at six geographical distant sites: Nancy, Lyon, Rennes, Bordeaux, Grenoble, and Sophia-Antipolis. The job submitter is located at a host in Nancy's site. The bandwidth between sites is 10Gbps everywhere except the link to Bordeaux which is

at 1Gbps. The network latencies from Nancy to the other sites are measured by an ICMP echo (ping) between frontal hosts at each site and we report the corresponding RTTs in legends of figures (top-left corners in Figure 4 and 5). We can see that latencies between Nancy and distant sites are very close for most of them. For all peers, the configuration parameter J is set to the number of cores in the host CPU.

In the experiment, we run a program whose each process simply echoes the name of the host it runs on. We run 11 times the program, requesting from 100 to 600 processes by steps of 50. Through this experiment, we observe where processes are mapped depending on the chosen strategy and the number of processes requested by counting hosts and cores allocated at each site.

For the *concentrate* strategy, we consider the closer the processes are from Nancy, the better are the results. For the *spread* strategy, a good allocation should map only one process per host as much as possible, and hosts selected should be the closest from Nancy. The effectiveness of the strategies essentially depends on the accuracy of the latency measurement, which may differ from the RTT given by an ICMP echo command (ping). The latency we measure with P2P-MPI must not necessarily be very close to the ICMP RTT, but should preserve the ranking between hosts relatively to RTT.

Figures 4 and 5 graphically represent the distribution of processes on the sites for the two strategies.

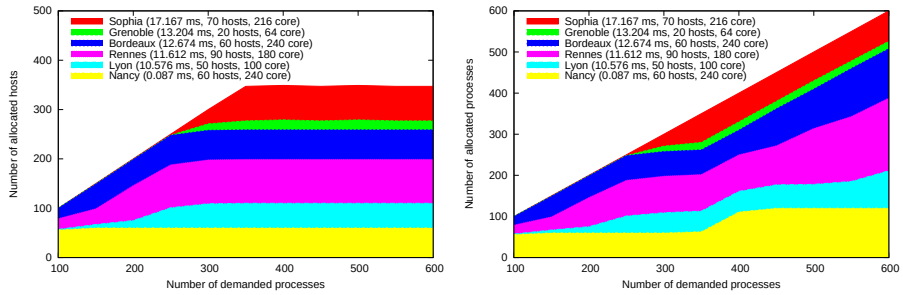


(a) Allocated hosts

(b) Allocated cores

Figure 4. Hosts and cores allocated with *Concentrate*

With *concentrate* (Figure 4) the processes are allocated on the 60 hosts available at Nancy only, up to 200 processes. Next, when the capacity of 240 cores at Nancy is exceeded by the request, further hosts are first allocated at Lyon (5 for -n 250), as expected with respect to the RTT ranking. Subsequent requests (from -n 300) reveal that hosts from Lyon, Rennes and Bordeaux fiercely compete for the latency ranking. We observe that the latency ranking for these hosts is interleaved with respect to sites. The reason of such an interleaving comes from the latencies between Nancy and any of the three sites, which are very close (they are within 0.6ms), and from the latency measurement, which is sensible to CPU and TCP load variations. Finally, the strategy selects close processes, and hence is adapted to applications involving many inter-process communications. As mentioned at the end of Section 3.2 the drawback is that the processes allocated to a same multi-core host must share the memory. Hence, memory contention is higher and the global amount of memory available to the application is limited.



(a) Allocated hosts (b) Allocated cores

Figure 5. Hosts and cores allocated with *Spread*

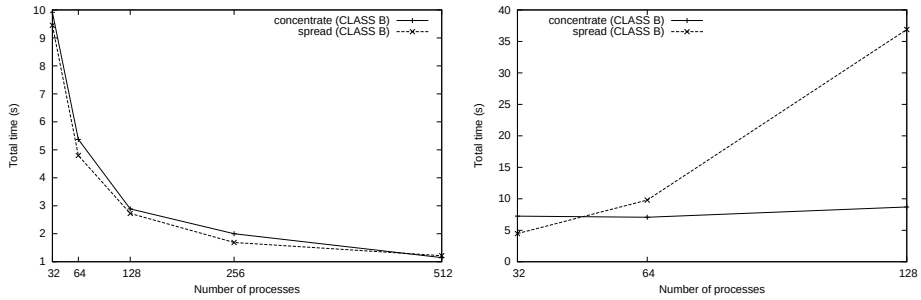
With *spread* (Figure 5) hosts are chosen from the four closest sites up to 250 processes, but contrarily to *concentrate* more hosts are allocated in each site. From 300 processes, *spread* takes hosts from all sites so that we have one process per host only. We can clearly see on Figure 5(b), the round-robin allocation of processes once the host list is exhausted: the cores allocated at Nancy makes a stair at 400 processes since there are not enough hosts (350) to map one process per host. Therefore, the closest peers are first chosen for the second process as they have extra available cores. On the whole, we observe that all peers have been discovered and the strategy tends to use them all. Hence, as compared to *concentrate*, this strategy is better suited for applications requiring much memory or making extensive memory accesses since processes have more chance not to be co-allocated with another process.

Notice also that the above experiment takes place in a stable environment made of clusters. Hence, the experiment may not evidence problems related to more volatile and heterogeneous P2P networks made up of volunteer peers.

Application Performance To observe how allocations impact applications, we have chosen two programs from the NAS benchmarks (NPB3.2) [4], a set of benchmarks originally developed for the performance evaluation of highly parallel supercomputers. The two program we have chosen to port from Fortran and C to Java have opposite characteristics regarding the computing to communication ratio. IS (Integer Sorting) involves many communications of small and large messages while EP (Embarrassingly Parallel) makes independent computations and only four final collective communication (`MPI.Allreduce` of one double).

The graph on the left of Figure 6(a) shows that EP using 32 to 256 processes is slightly faster with *spread*. We can think of two factors to explain that execution is slower with *concentrate*. First, the two computing processes running on different cores of a same host contend for access to the main memory. Second, although some inter-process communications could be optimized using the shared memory, our implementation currently uses the Ethernet stack in all communications. As each application process is monitored by one failure detector sending periodic administrative messages, the number of messages handled by a host network interface is bigger. These factors seem not to be counterbalanced by locality in the collective communication. With 512 processes, the problem

size per process becomes smaller and the overheads related to memory and communications seem to reach an equilibrium at this point.



(a) Execution time on EP benchmark.

(b) Execution time on IS benchmark.

Figure 6. Execution time for EP and IS depending on allocation strategies.

The performance curves for IS, in Figure 6(b), are due to the low computations to communications ratio. With 32 processes, *spread* leads to better performances than *concentrate*: with *spread* all processes are in the same cluster so that communications pay a low latency while there is no overhead due to concurrent memory accesses. This appears to be the case with *concentrate*. Using 64 processes with *spread* means that four processes are allocated outside the local cluster and the communication overhead leads to a slowdown. Keeping the processes inside the cluster with *concentrate* gives a roughly constant execution time. Figures for 128 processes and above show the same phenomena.

As a reference, we compare in Figure 7 the performance of EP obtained on a cluster with three different communication libraries. The results for IS are not significant and not shown here (it does not scale beyond eight processors in OpenMPI). The two other frameworks are MPJ Express [5], another MPJ implementation with which we run our Java benchmark, and OpenMPI [24], a popular MPI implementation, with which we run the original Fortran code.

MPJ Express and P2P-MPI performances are very close, but are outperformed by OpenMPI. OpenMPI is known for its efficiency, and further, it runs in this case a native binary code while MPJ implementations have the overhead of using a JVM.

4. Fault-tolerance

As stated in the introduction, the robustness of an execution is of tremendous importance for MPI application since a single faulty process makes the whole application fail. As pointed out in the related work section, we argue that usual approaches to support fault-tolerance for MPI, based on rollback recovery, do not fit easily in our peer-to-peer paradigm because they assume a reliable server where checkpoints can be stored. This is why we propose a solution based on *process replication*. The replication management is absolutely transparent for the programmer. When specifying a desired number of processes, the user can request the system to run for each process an arbitrary number of copies called *replicas*. An exception is made for the process running on the submitter

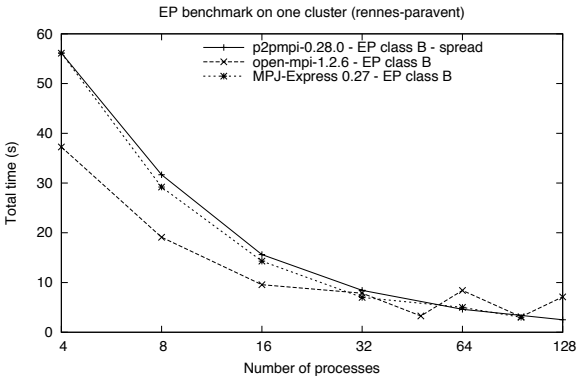


Figure 7. Execution time for EP in a cluster

host, numbered 0 by convention, 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*.

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 implemented by one or several replicas noted $P_i^0, \ldots, P_i^n$. Figure 8 shows an example of two logical processes communicating. In that example, P_1 is implemented by three replicas mapped onto three different computers. In all cases, the replicas are run in parallel on different hosts since the goal is to allow the continuation of the execution even if some hosts fail. Note that we can have replicas from different logical processes on a same host (e.g $P_{0,0}$ and $P_{1,0}$ on host A).

4.1. Assumptions

Before we can describe the replication management, we should qualify our system regarding the nature of the distributed system addressed:

- We only consider *fail-stop failures* (or *crash failures*). It means that a failed process stops performing any activity including sending, transmitting or receiving any message. This includes the three following situations: a) the process itself crashes (e.g. the program aborts on a DivideByZero error), b) the host executing the process crashes (e.g. the computer is shut off), or c) the fault-detection monitoring the process crashes and hence no more notifications of aliveness are reported to other processes. This excludes transient or byzantine failures.
- We consider a *partially synchronous* system: a) the clock drift remains the same, or the differences in the drifts are negligible, for all hosts during an application execution, b) there is no global clock and c) communications deliver messages in a finite time.
- We consider the network links to be reliable: there is no message loss.

The assumption about network communication reliability is justified by the fact that we use TCP, which is reliable, and that the middleware checks on startup that the required TCP ports are not firewalled.

4.2. Replicas coordination protocol

An execution of an application with replication must be *equivalent* to the execution of the same application without replication. We say an execution E is equivalent to an execution E' if the output of E is the same as any output E' could produce. To guarantee this property, we need a specific protocol that emulates an *atomic broadcast* when sending messages from one process to another. This is the role of the *coordination protocol* presented hereafter. Its behavior regarding atomic broadcast is examined in Section 4.3. The protocol relies on specific control structures and roles. First, in each logical process, one replica is elected as *master*. If this process fails, one replica of the group will be elected as a new master, and it will update its state to be in the same state as the master before its failure. Second, to be able to return or get to a certain state, replicas need to store some information about messages sent or received. We have added extra data structures in each process: a *backup table* and a *log table* used by a process when sending, and a *history table* used when receiving. Their roles will be explained along with the description of the protocol.

Message Identifier (MID) First, our protocol requires a unique identifier for messages. The communication library computes MIDs on the fly, from local information only. The MID is a 5-uple built from the communicator, the source, destination and tag of the message, plus a logical time (ticks are incremented at each send or receive). For example, two consecutive messages sent in the world communicator (numbered 0), from process of rank 0 to rank 2 with tag 9, will have the identifiers $(0, 0, 2, 9, 0)$ and $(0, 0, 2, 9, 1)$ respectively. The MID is incorporated into the message itself and logged at the receiving side. Thus, the MID has two properties: it is a unique identifier for messages, and it reflects the order in which messages are sent and received. In the example, the messages could be received in any order in the receive queue, but the extraction from the queue to the user program would follow the MID order. Hence, we preserve the message order according to the MPI standard.

Sending agreement protocol On the sender side, we limit the number of messages sent by introducing the following agreement protocol. In each logical process, one replica is elected as master of the group for sending. The other processes do not send the message over the network, but store it in their memory. Figure 8 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 own logical process to indicate that the message has been correctly transmitted. We say the master *commits* its send. The commit is done by sending the message's MID. The MIDs are stored into the *log tables* of each replica.
- if the replica is not the master, it first looks up its log table to see if the message has already been sent by the master. If it has already been sent, the replica just continues with subsequent instructions. If not, the message to be sent is stored into the *backup table* and the execution continues. (Execution stops only in a waiting state on a receive instruction.) When a replica receives a commit, it writes the

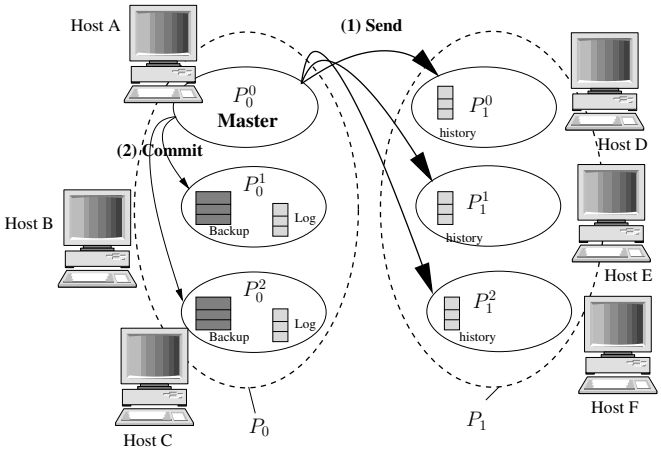


Figure 8. A message sent from logical process P_0 to P_1.

message identifier in its log and if the message has been stored, it removes it from the backup table.

Reception agreement protocol When a message arrives in the message queue, the communication library compares the message's MID with MIDs stored earlier in the history table. If MID is a duplicate, the message is simply discarded, otherwise the communication library delivers the message to the application.

Recovery When a failure is detected, the following fault recovery action is taken. If the failed process is a replica, each other process simply updates the information about the corresponding logical process, not to send further messages to the failed process. If the failed process is the master, a new master is elected among its replicas. This new master checks its backup. If it is not empty, it means the messages in the backup have not been sent at all by the previous master, sent partially, or totally sent but not committed. In any case, it starts over the multiple send operations. Thus, processes on the receiving side might have received the message from the master before it failed, and once again from the new master after the failure. This situation is handled by the reception agreement protocol that discards duplicate messages.

4.3. Theoretical Foundations

Our protocol can be considered as an *active replication* strategy [40] because the senders send their messages to all replicas of the destination group. However, our protocol differs (for a sake of performance) because we restrict the group of senders to a single process only, the group master. The conditions for such group communication to work properly have been well studied in the literature. We review below what are the requirements stated in the literature, and how our system complies to these requirements.

It is well known that active replication requires *atomic broadcast* (or *total order broadcast*) to insure the coherence of the system. The specification of the atomic broad-

cast has been formally defined using the two primitives *broadcast(m)* and *deliver(m)*[4] [30]. It is assumed that every message m can be uniquely identified, and carries the identity of its sender, denoted by $sender(m)$. This assumption holds in P2P-MPI because we use MIDs. A process that suffers no failure is termed *correct process*. The atomic broadcast is defined by the following properties, written in italics. For each property, we state how it applies to our system.

Validity *if a correct process broadcasts a message m, then it eventually delivers m.* From our assumption that our system is partially synchronous and that our communication links are reliable, this property is satisfied.

Agreement *If a correct process delivers a message m, then all correct processes eventually deliver m.* If the sender does not crash, the validity property satisfied above insures that the message will be delivered to all destination processes. If the sender crashes between any send to the destination processes, a replica of the sender will become the new master in a finite time. (Or the application crashes if it does not remain any replica in the logical process of the sender). It will then retransmit the message[5]to the destination processes. Thus, in the end all destination processes will receive the message. Hence, the property is satisfied.

Integrity *For any message m, every correct process delivers* m *at most once, and only if m was previously broadcasted by $sender(m)$*. On the receiver side, MIDs and the history table are used to detect and discard duplicated received message. Hence, we never deliver duplicated message and the property is satisfied.

Total order *If process p and q both deliver messages m and m', then p delivers m before m', if and only if q delivers m before m'.* In other words, every process gets the messages in the same order. The received message will be delivered upon the `MPI.Recv` call from the user program. The communication library fetches the received message from its temporary buffer in the order indicated by the program, as encoded in the MID.

We must note however, that MPI allows the programmer to describe communications that do not satisfy the last property. MPI has the special specifiers `MPI_ANY_TAG` and `MPI_ANY_SOURCE` that may be used in a receive call, respectively as tag or source values. Using any of these specifiers can formally lead to an inconsistent state. Let us illustrate the situation with a simple example. Suppose a process P_0 implemented by two replicas (P_0^0 and P_0^1) whose code executes two successive receive operations, 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. Therefore, the outputs produced by the master and its replica may not be the same. However, one can argue that 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

[4] *deliver* is used instead of *receive* to mean that the message is really available to the application and not just received by the network interface.

[5] Note that the messages must be the same on the master and on the replicas. We assume the same instructions produce the same values, except for `random()`, which we overload so that drawn values are the same in the replicas and in the master.

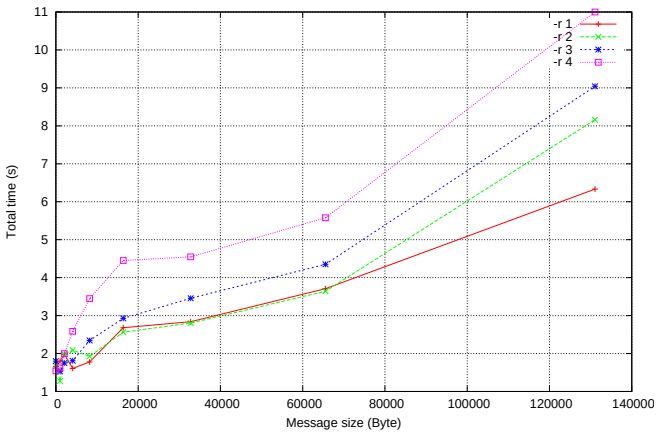


Figure 9. Time spent for 1000 ping-pong messages with different replication degrees.

of values gathered in an unspecified order which is correct because sum is associative, commutative, and has a neutral element.

4.4. Replication Overhead

The replication protocol obviously incurs an overhead. Each message normally sent once, is sent to all replicas of the destination process. An extra step is necessary as well for the master to commit the message sent to its own replicas,which requires sending a small message. To assess the overhead from an experimental point of view, we measure the performance of a simple ping-pong program between two processes. We report in Figure 9 the time taken by the round trip time of 1000 message exchanges, with different replication degrees and message sizes. The measurements are average values over ten tests, which are run on a standard cluster (1Gbps link). If we consider t_1 the execution time without replication, we observe that the overhead for replication degree r is a bit less than rt_1. For example, the communication overhead induced by a replication degree of two (r=2) appears almost negligible for messages up to 64 KB. For a 64 KB message, the overhead is 17% for r=3, and 50% for r=4. It goes up to 42% and 73% respectively for 128 KB messages. This test can be certainly considered partial but a thorough study of performance overhead would require to get through a lot of configurations. Yet, it allows us to set a realistic upper bound for the overhead. This is helpful for modeling the effect of replication on fault-tolerance, as will be seen in next section.

4.5. Replication and Failure Probability

We have examined so far how replication could be designed and implemented. In this section, we quantify the benefits and the costs of replication on program execution. We give an expression of the failure probability of an application and how much replication improves an application's robustness.

Our failure model follows previous studies on the availability of machines in wide-area environment such as the one of Nurmi et al. [34]. Such studies show that the Weibull

distribution effectively model machine availability. Based on [34], the probability that a machine fails before time t is given by:

$$Pr([0,t]) = 1 - e^{-(t\lambda)^{\delta}} \tag{1}$$

where $\lambda > 0$ is the failure rate, and $\delta > 0$ the shape of the Weibull distribution. The authors show how to compute λ and δ according to traces. They also show that $\delta < 1$, which means that we can consider that we have a failure rate decreasing with time (unreliable machines tend to leave the system rapidly). Note that the Weibull distribution is a generalization of the exponential distribution (constant failure rate) when $\delta = 1$.

Now, recall that our parallel applications consist in a set of processes, and that the failure of any of them makes the application fail. We assume failures are independent events, occurring equiprobably at each host: we note $f(t)$ the probability (that will be instantiated with our failure model of Eq. (1)) that a host fails before t. Thus, considering a p processes MPI application without replication, the probability that it crashes is :

$$\begin{array}{rcl} P_{app(p)} & = & \text{probability that 1, or 2, } \ldots \text{, or } p \text{ processes crash} \\ & = & 1 - (\text{probability that no process crashes}) \\ & = & 1 - (1 - f(t))^p \end{array}$$

With a replication degree r, a crash of the application occurs if and only if at least one MPI process has all its r copies failed. The probability that all the r copies of an MPI process fail is $(f(t))^r$. Thus, like in the expression above, the probability that a p process MPI application with replication degree r crashes is

$$\begin{array}{rl} P_{appr(p,r)} & = 1 - (1 - f(t)^r)^p \\ & = 1 - (1 - (1 - e^{-(t\ \lambda)^{\delta}})^r)^p \text{ using Eq. (1)} \end{array} \tag{2}$$

While replication makes the failure probability decrease, it also adds an overhead that lengthens the overall execution time. Hence, the failure probability is greater during this longer period. So, the question of the best tradeoff arises, which should determine the optimal replication degree. A similar question can be: which replication degree induces a given failure probability. It is out of the scope of this chapter to detail how such a decision can be computed. These details can be found in [25]. In this work, we provide a model of the duration of the program execution derived from Amdahl's law. The duration depends on the estimated sequential time, the parallel portion of the program, the number of processes involved and the replication degree, which incurs an overhead considered linear in r. We can then instantiate the failure probability of Eq. (2) with the duration computed i.e., substituting t with our duration expression. We have shown that the associated function is convex for realistic values of λ. An illustration is given in Figure 10 for an application spawning ten processes. The convex curve shows that the failure probability is quickly decreasing and reaches a minimum for $r \approx 7$. More replication is useless since it involves a higher failure probability (as the overall duration increases).

5. Fault Detection

For the replication to work properly, each process must reach in a definite period, a global knowledge of other processes states to prevent incoherence. For instance, running pro-

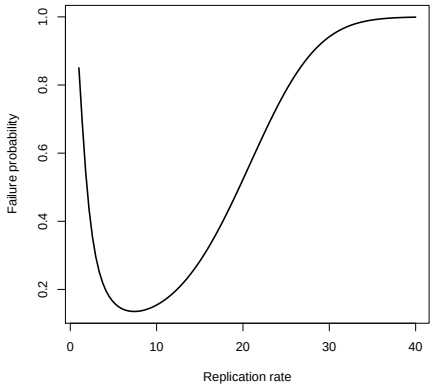


Figure 10. $P_{app(10,r)}$ with $\delta = 1$, failure rate λ=10^{-1}, sequential time $10s$

cesses should stop sending messages to a failed process. This problem becomes challenging when large scale systems are in the scope. When an application starts, it registers with a local service called the *fault-detection service*, introduced in Section 2. In each host, this service is responsible to notify the local application process of failures happening on co-allocated processes. Thus, the design of the failure detectors is of primary importance for fault-tolerance.

5.1. Gossiping

Failure detection services have received much attention in the literature and since they are considered as first-class services of distributed systems [15], many protocols for failure detection have been proposed and implemented.

Among them, we retain the so-called *gossiping* protocol after the gossip-style fault detection service presented in [47]. It is a distributed algorithm whose informative messages are evenly dispatched among the links of the system. Let us sketch the principle of the algorithm.

A gossip failure detector is a set of distributed modules, one module residing at each host to monitor, as illustrated on Figure 11. Each module maintains a local table with one entry per detector known to it. This entry includes a counter called *heartbeat*. In a running state, each module repeatedly chooses some other modules and sends them a gossip message consisting in its table with its heartbeat incremented (the table on the left of host on figure). When a module receives one or more gossip messages from other modules, it merges its local table with all received tables and adopts for each host the maximum heartbeat found (table on the right of host on figure). If a heartbeat for a host A, which is maintained by a failure detector at host B has not increased after a certain timeout, host B suspects that host A has crashed. In general, it follows a consensus phase about host A failure in order to keep the system's coherence.

Gossiping protocols are usually governed by three key parameters: the gossip time, cleanup time, and the consensus time. Gossip time, noted T_{gossip}, is the time interval be-

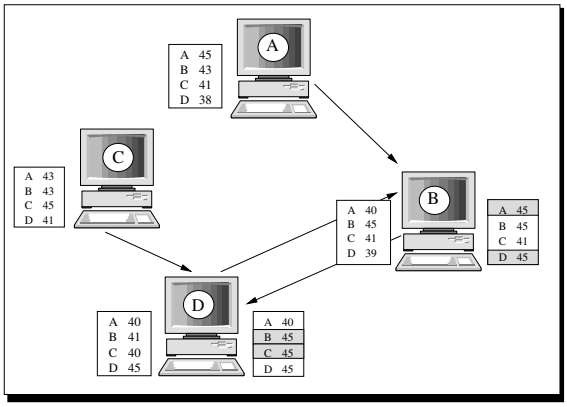


Figure 11. One step in random gossiping: each host sends its heartbeat table randomly (left tables) and updates its table keeping maximum values (right tables).

tween two consecutive gossip messages. Cleanup time, or $T_{cleanup}$, is the time interval after which a host is suspected to have failed. Finally, consensus time noted $T_{consensus}$, is the time interval after which consensus is reached about a failed node.

A major difficulty in gossiping implementations lies in the setting of $T_{cleanup}$: it is easy to compute a lower bound, referred to as $T^{min}_{cleanup}$, which is the time required for information to reach all other hosts, but this can serve as $T_{cleanup}$ only in synchronous systems (i.e. using a global clock). In asynchronous systems, the cleanup time is usually set to some multiple of the gossip time, and must neither be too long to avoid long detection times, nor too short to avoid frequent false failure detections.

Starting from this basis, several proposals have been made to improve or adapt this gossip-style failure detector to other contexts [36].

We briefly review advantages and disadvantages of the original and modified gossip based protocols and what is to be adapted to meet P2P-MPI requirements. Notably, we pay attention to the detection time ($T^{min}_{cleanup}$) and reliability of each protocol.

Random. In the gossip protocol originally proposed [47], each module randomly chooses at each step, the hosts it sends its table to. In practice, random gossip evens the communication load among the network links but has the disadvantage of being non-deterministic. It is possible that a node receives no gossip message for a period long enough to cause a false failure detection, i.e. a node is considered failed whereas it is still alive.

Round-Robin (RR). This method aims to make gossip traffic more uniform by employing a deterministic approach. Periodically, each node will receive and send a single gossip message to a pre-determined destination node d, which is computed from the source node s and the current round number r.

$$d = (s + r) \mod n, \;\; 0 \leq s < n, 1 \leq r < n \tag{3}$$

where n is the number of nodes. After $r = n - 1$ rounds, all nodes have communicated with each other, which ends a *cycle* and r (generally implemented as a circular counter) is reset to 1. This protocol guarantees that all nodes will receive a given node's updated

heartbeat within a bounded time. The information about a node's state is transmitted to one other node in the first round, then to two other nodes in the second round (one node gets the information directly from the initial node, the other from the node previously informed), etc. At a given round r, there are $1 + 2 + \cdots + r$ nodes informed, and hence the minimum cleanup time (all nodes informed) is such that $\frac{r(r+1)}{2} = n$. Hence, we can deduce the minimum cleanup time: $T^{min}_{cleanup} = \lceil r \rceil \times T_{gossip}$, where $r = (\sqrt{1+8n} - 1)/2$.

Binary Round-Robin (BRR). The binary round-robin protocol attempts to minimize bandwidth used for gossiping by eliminating all redundant gossiping messages. The inherent redundancy of the round-robin protocol is avoided by skipping the unnecessary steps. The algorithm determines sources and destination nodes from the following relation:

$$d = (s + 2^{r-1}) \mod n, \;\; 1 \leq r \leq \lceil log_2(n) \rceil \tag{4}$$

The cycle length is $\lceil log_2(n) \rceil$ rounds, and we have $T^{min}_{cleanup} = \lceil log_2(n) \rceil \times T_{gossip}$.

The elimination of redundant gossip lessens the network load and accelerates the heartbeat status dissemination at the cost of an increased risk of false detections. For example in a four nodes system, node 2 gets incoming messages from node 1 (in the 1st round) and from node 0 (2nd round) only. Therefore, if node 0 and 1 fail, node 2 will not receive any more gossip messages. After $T_{cleanup}$, node 2 will suspect node 3 to have failed even if it is not true.

5.2. Fault detection in P2P-MPI: BRR or DBRR

We have set up a list of requirements for our failure detection service. We require the protocol to be a) scalable, i.e. the network traffic that it generates does not induce bottlenecks, b) efficient, i.e. the detection time is acceptable relatively to the application execution time, c) deterministic in the fault detection time, i.e. a fault is detected in a guaranteed delay, d) reliable, .i.e. its failure probability is several orders of magnitudes less than the failure probability of the monitored application, since its failure would result in false failure detections.

From the previous proposals for failure detection, BRR meets almost all of these requirements. It is deterministic, has a low bandwidth usage and a quick detection time. However, we have shown in [27] that BRR is relatively fragile as compared to other protocols, especially with a small number of nodes. To let the user trade off between detection speed and reliability, we have derived a new protocol called double binary round-robin protocol (DBRR). It detects failures in a delay asymptotically equal to BRR ($O(log_2(n))$, which is acceptably fast in practice, while reinforcing the robustness of BRR. The idea is simply to avoid to have only one-way connections between nodes. Thus, in the first half of a cycle, we use the BRR routing in a clock-wise direction while in the second half, we establish a connection back by applying BRR in a counterclock-wise direction. The destination node for each gossip message is determined by the following relation:

$$d = \begin{cases} (s + 2^{r-1}) \mod n & \text{if } 1 \leq r \leq \lceil log_2(n) \rceil \\ (s - 2^{r - \lceil log_2(n) \rceil - 1}) \mod n & \text{if } \lceil log_2(n) \rceil < r \leq 2\lceil log_2(n) \rceil \end{cases} \tag{5}$$

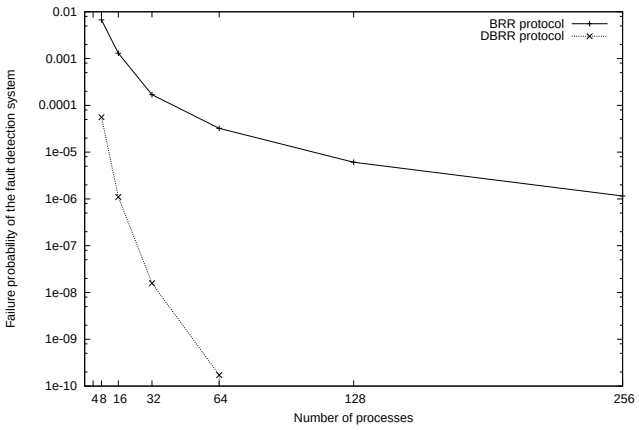


Figure 12. Failure probabilities of the FD system using BRR and DBRR ($\delta = 1, \lambda t = 10^{-1}$).

The cycle length is $2\lceil log_2(n) \rceil$ and hence we have $T^{min}_{cleanup} = 2\lceil log_2(n) \rceil \times T_{gossip}$. With the same assumptions as for BRR, we set $T_{cleanup} = 3\lceil log_2(n) \rceil \times T_{gossip}$ for DBRR. We have compared BRR and DBRR through the simulation of a large number of scenarios, in which each node may fail with a probability f. Then, we verify if the graph representing the BRR or DBRR routing is connected: simultaneous nodes failures may cut all edges from source nodes to a destination node. This situation implies a FD failure. In Figure 12, we repeat the simulation for 5.8×10^9 trials with $\delta = 1, \lambda = 10^{-3}s^{-1}$ and $t = 10^2 s$. Notice that in the DBRR protocol, we could not find any FD failure when the number of nodes n is more than 64, which means the number of our trials is not sufficient to estimate the DBRR failure probability for such n.

The chosen protocol appears in the configuration file and may change for each application (at startup, all FDs are instructed with which protocol they should monitor a given application).

5.3. Fault Detection Time Evaluation

It is important for users to have an idea about the time it will take for a failure to be signaled. Because they use a deterministic routing of information messages, BRR and DBRR allow to theoretically predict the fault detection time. We have setup an experiment in real conditions to compare the predicted detection time with the detection times observed when failures occur in a real application. We run an application (without replication) which is distributed across three geographically distant sites, namely Nancy, Rennes and Sophia-Antipolis on the Grid'5000 testbed. After 20 seconds we kill all processes on a random node to simulate a node failure. We then log at what time each node is notified of the failure and compute the time interval between failure and detection. For both protocols BRR and DBRR, T_{gossip} is set to 0.5 second. Figure 13 plots the average of these intervals on all nodes. Also plotted for comparison is $T_{cleanup}$ as specified previously, termed "theoretical" detection time on the graph.

The detection speed observed is very similar to the theoretical predictions whatever the number of processes involved, up to 256. The difference comes from the time taken

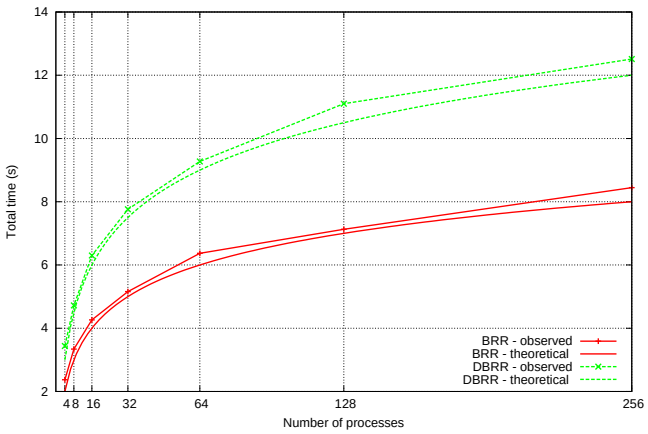


Figure 13. Time to detect a fault for BRR and DBRR

to check that a suspected host has really failed through a ping of that host (this is the consensus phase). We observed no false detection throughout our tests, hence the ping procedure has been triggered only for real failures. There are two reasons for a false detection: either all sources of information for a node fail, or $T_{cleanup}$ is too short with respect to the system characteristics (communication delays, local clocks drifts, etc). Here, given the briefness of execution, the former reason is out of the scope. Given the absence of false failures we can conclude that we have chosen a correct detection time $T_{cleanup}$, and our initial assumptions are correct, i.e. the initial heartbeat adjustment is effective and message delays are less than T_{gossip}. This experiment shows the scalability of the system on Grid5000, despite the presence of wide area network links between hosts.

6. Conclusion

We have described our proposal for an integrated middleware coupled with a communication library. This proposal has been implemented and is publicly proposed as a free software project[6].

In this chapter, we have explained our design choices to ease the deployment of the framework and to minimize the maintenance operations. We propose a P2P basis to organize the resources. The advantage lies in the greater peer autonomy, which eases the software installation and maintenance, and avoids the single point of failure risk due to central directories for resources. We put forward that the dynamic discovery of available resources upon an execution request is a highly desirable feature. We have discussed the resource allocation issue, and we have shown how the middleware could account for network locality of peers, and which simple allocation strategies may be proposed to the user.

[6]http://www.p2pmpi.org/

Another key feature of P2P-MPI is fault-tolerance. The middleware has a failure detection service, which notifies failures to the application. We have explained the difficulties to build a scalable and fast detection system, and how our service has been designed. The communication library supports fault-tolerance through replication of processes, upon a simple user request. We have described the underlying protocol, and we have shown how replication increases the robustness of applications. The overhead of replication is also studied. Thus, our proposal on fault-management contributes to show that the middleware support is beneficial to the communication library. Finally, we think P2P-MPI can encourage programmers to parallelize their applications to benefit from the computational power available even from individual computers. The applications best suited to this framework are those which can take advantage of COTS hardware. A P2P-MPI program is not as efficient as it would be in C or Fortran. Moreover, P2P-MPI only handles TCP networking devices for the moment, and hence cannot make the most of a cluster with myrinet or infiniband network cards. However, it allows to parallelize existing Java programs using message passing, which is a more general parallel programming model than the client-server model. Hence, a wide range of applications can be targeted by P2P-MPI. Let us cite the various examples found in the Java Grande Forum benchmark [11], which includes a financial simulation using Monte Carlo techniques to price products, a molecular dynamics simulation based on a N-body code, a scene renderer based on a 3D ray-tracer, etc. During this project, we have ourselves helped at the parallelization of a data clustering method [7]. This work is described in [26]. This method has a high complexity and its parallelization enhanced its usability. Clusterings with a large number of classes have been completed on COTS hardware in tens of minutes instead of hours in the sequential version. In addition, a noteworthy aspect is that using P2P-MPI is more user-friendly than using traditional high-end computing facilities. Instead of moving their application files to a cluster for example, users can keep running the application from their usual computer, and the middleware transparently discovers available computing resources.

Regarding the future work, let us list some possible directions. The middleware should rely on a more decentralized infrastructure, composed of a distributed set of supernodes, to scale beyond thousands of peers. A linked problem is to maintain an accurate estimation of the network latencies between peers, or better, being able to guess the topology of the physical network (similarly to the method used in [19]). As far as replication is concerned, a formal analysis of the protocol (e.g using model-checking) would make it a solid brick. A comparison with other approaches of fault-tolerance regarding for instance, the overhead depending on the number of faults injected, would be also interesting. Last, much work could be done on the MPJ implementation. In particular, we think P2P-MPI is a good framework to test new algorithms for collective communications involving mixed wide and local area communications. We believe the communication library could benefit from static or even dynamic information about the network that could be retrieved from the middleware layer. Such information about topology, latency, load, etc, could be used to make better decisions to choose such or such communication strategy.

References

[1] Lorenzo Alvisi and Keith Marzullo. Message logging: Pessimistic, optimistic, and causal. In *Proceeding of the 15th International Conference on Distributed Computing Systems (ICDCS'95)*, pages 229–236, 1995.

[2] David P. Anderson. Boinc: A system for public-resource computing and storage. In Rajkumar Buyya, editor, *5th International Workshop on Grid Computing (GRID 2004)*, pages 4–10. IEEE Computer Society, 2004.

[3] Gabriel Antoniu, Loïc Cudennec, Mike Duigou, and Mathieu Jan. Performance scalability of the JXTA P2P framework. In *Proc. 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007)*, Long Beach, CA, USA, March 2007.

[4] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R., Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS Paralell Benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, March 1994.

[5] Mark Baker, Bryan Carpenter, and Aamir Shafi. MPJ Express: Towards thread safe Java HPC. In *CLUSTER*, pages 1–10. IEEE, 2006.

[6] Rajanikanth Batchu, Yoginder S. Dandass, Anthony Skjellum, and Murali Beddhu. MPI/FT: A model-based approach to low-overhead fault tolerant message-passing middleware. *Cluster Computing*, 7(4):303–315, 2004.

[7] Alexandre Blansché and Pierre Gançarski. MACLAW: A modular approach for clustering with local attribute weighting. *Pattern Recognition Letters*, 27(11):1299–1306, 2006.

[8] Markus Bornemann, Rob V. van Nieuwpoort, and Thilo Kielmann. MPJ/Ibis: A flexible and efficient message passing platform for java. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3666 of *Lecture Notes in Computer Science*, pages 217–224. Springer, 2005.

[9] George Bosilca, Aurélien Bouteiller, Franck Cappello, Samir Djailali, Gilles Fedak, Cécile Germain, Thomas Hérault, Pierre Lemarinier, Oleg Lodygensky, Frédéric Magniette, Vincent Néri, and Anton Selikhov. MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes. In *SuperComputing 2002*, pages 1–18, Baltimore, USA, November 2002.

[10] Aurélien Bouteiller, Franck Cappello, Thomas Hérault, Géraud Krawezik, Pierre Lemarinier, and Frédéric Magniette. (mpich-v2): a fault tolerant MPI for volatile nodes based on the pessimistic sender based message logging. In *SuperComputing 2003*, pages 242–250, Phoenix USA, November 2003.

[11] J. Mark Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A benchmark suite for high performance java. *Concurrency - Practice and Experience*, 12(6):375–388, 2000.

[12] Franck Cappello et al. Grid'5000: a large scale and highly reconfigurable grid experimental testbed. In *6th IEEE/ACM International Conference on Grid Computing (GRID 2005)*, pages 99–106, 2005.

[13] Denis Caromel, Alexandre di Costanzo, and Clement Mathieu. Peer-to-peer for computational grids: mixing clusters and desktop machines. *Parallel Computing*, 33(4-5):275–288, May 2007.

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

[15] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[16] David Dewolfs, Jan Broeckhove, Vaidy S. Sunderam, and Graham E. Fagg. Ft-mpi, fault-tolerant metacomputing and generic name services: A case study. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 13th European PVM/MPI User's Group Meeting*, volume 4192 of *Lecture Notes in Computer Science*, pages 133–140. Springer, 2006.

[17] Niels Drost, Rob V. van Nieuwpoort, and Henri Bal. Simple locality-aware co-allocation in peer-to-peer supercomputing. In *Sixth IEEE International Symposium on Cluster Computing and the Grid Workshops (CCGRID'06)*, pages 14–21. IEEE, 2006.

[18] Dietmar W. Erwin and David F. Snelling. Unicore: A grid computing environment. In Rizos Sakellariou, John Keane, John R. Gurd, and Len Freeman, editors, *Euro-Par*, volume 2150 of *Lecture Notes in Computer Science*, pages 825–834. Springer, 2001.

[19] Lionel Eyraud-Dubois, Arnaud Legrand, Martin Quinson, and Frédéric Vivien. A first step towards automatically building network representations. In Anne-Marie Kermarrec, Luc Bougé, and Thierry Priol, editors, *Euro-Par*, volume 4641 of *Lecture Notes in Computer Science*, pages 160–169. Springer, 2007.

[20] Graham Fagg and Jack 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] 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.

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

[23] Ian T. Foster and Adriana Iamnitchi. On death, taxes, and the convergence of peer-to-peer and grid computing. In M. Frans Kaashoek and Ion Stoica, editors, *Peer-to-Peer Systems II, Second International Workshop, IPTPS*, volume 2735 of *Lecture Notes in Computer Science*, pages 118–128. Springer, 2003.

[24] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[25] Stéphane Genaud, Emmanuel Jeannot, and Choopan Rattanapoka. Fault management in P2P-MPI. *International Journal of Parallel Programming*, 2009. Submitted. Extended version of [27].

[26] Stéphane Genaud, Pierre Gançarski, Guillaume Latu, Alexandre Blansché, Choopan Rattanapoka, and Damien Vouriot. Exploitation of a parallel clustering algorithm on commodity hardware with P2P-MPI. *The Journal of SuperComputing*, 43(1):21–41, January 2008.

[27] Stéphane Genaud and Choopan Rattanapoka. Fault management in p2p-mpi. In *In proceedings of International Conference on Grid and Pervasive Computing, GPC'07*, Lecture Notes in Computer Science, pages 64–77. Springer, May 2007.

[28] Stéphane Genaud and Choopan Rattanapoka. Large-scale experiment of co-allocation strategies for peer-to-peer supercomputing in p2p-mpi. In *5th High Performance Grid Computing International Workshop, IPDPS conference proceedings*. IEEE, April 2008.

[29] 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.

[30] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, 1994.

[31] Eduardo Huedo, Rubén S. Montero, and Ignacio Martín Llorente. A framework for adaptive execution in grids. *Software, Practice and Experience*, 34(7):631–651, 2004.

[32] Emmanuel Jeanvoine, Christine Morin, and Daniel Leprince. Vigne: Executing easily and efficiently a wide range of distributed applications in grids. In *Proceedings of Euro-Par 2007*, pages 394–403, Rennes, France, 2007.

[33] 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.

[34] Daniel Nurmi, John Brevik, and Richard Wolski. Modeling machine availability in enterprise and wide-area distributed computing environments. In José C. Cunha and Pedro D. Medeiros, editors, *Euro-Par*, volume 3648 of *Lecture Notes in Computer Science*, pages 432–441. Springer, 2005.

[35] Rolf Rabenseifner and Jesper Larsson Träff. More efficient reduction algorithms for non-power-of-two number of processors in message-passing parallel systems book series lecture notes in computer science. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3241 of *LNCS*, pages 36–46. Springer, 2004.

[36] Sridharan Ranganathan, Alan D. George, Robert W. Todd, and Matthew C. Chidester. Gossip-style failure detection and distributed consensus for scalable heterogeneous clusters. *Cluster Computing*, 4(3):197–209, 2001.

[37] Choopan Rattanapoka. *P2P-MPI: A Fault-tolerant Mesasges Passing Interface Implementation for Grids*. PhD thesis, University Louis Pasteur, Strasbourg, April 2008.

[38] Sean C. Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a DHT. In *ATEC'04: Proceedings of the USENIX Annual Technical Conference 2004 on USENIX Annual Technical Conference*, pages 127–140, Berkeley, CA, USA, 2004. USENIX Association.

[39] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. *International Journal of High Performance Computing Applications*, 19(4):479–493, Winter 2005.

[40] Fred. B. Schneider. *Replication Management Using the State Machine Approach*, chapter 7, pages 169–

195. ACM Press, 1993.

[41] Kazuyuki Shudo, Yoshio Tanaka, and Satoshi Sekiguchi. 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*, pages 259–266. IEEE, May 2005.

[42] Marc Snir, Steve W. Otto, David W. Walker, Jack Dongarra, and Steven Huss-Lederman. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, USA, 1995.

[43] Georg Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, pages 526–531, 1996.

[44] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the Condor experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):323–356, 2005.

[45] Bernard Traversat, Ahkil Arora, Mohamed Abdelaziz and Mike Duigou, Carl Haywood, Jean-Christophe Hugly, Eric Pouyoul and Bill Yeager. Project jxta 2.0 super-peer virtual network, May 2003.

[46] Rob van Nieuwpoort, Jason Maassen, Rutger F. H. Hofman, Thilo Kielmann, and Henri E. Bal. Ibis: an efficient java-based grid programming environment. In José E. Moreira, Geoffrey Fox, and Vladimir Getov, editors, *Java Grande*, pages 18–27. ACM, 2002.

[47] Robbert van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. In *Middleware '98*, page 55, 1998.