Int J Parallel Prog (2009) 37:433–461
DOI 10.1007/s10766-009-0115-8

Fault-Management in P2P-MPI

Stéphane Genaud · Emmanuel Jeannot · Choopan Rattanapoka

Received: 23 May 2008 / Accepted: 21 July 2009 / Published online: 5 August 2009

Abstract We present in this paper a study on fault management in a grid middleware. The middleware is our home-grown software called P2P-MPI. This framework is MPJ compliant, allows users to execute message passing parallel programs, and its objective is to support environments using commodity hardware. Hence, running programs is failure prone and a particular attention must be paid to fault management. The fault management covers two issues: fault-tolerance and fault detection. Fault-tolerance deals with the program execution: P2P-MPI provides a transparent fault tolerance facility based on replication of computations. Fault detection concerns the monitoring of the program execution by the system. The monitoring is done through a distributed set of modules called failure detectors. The contribution of this paper is twofold. The first contribution is the evaluation of the failure probability of an application depending on the replication degree. The failure probability depends on the execution length, and we propose a model to evaluate the duration of a replicated parallel program. Then, we give an expression of the replication degree required to keep the failure probability of an execution under a given threshold. The second contribution is a study of the advantages and drawbacks of several fault detection systems found in the literature. The criteria of our evaluation are the reliability of the failure detection service and

S. Genaud (✉) · E. Jeannot
AlGorille Team, LORIA, Campus Scientifique, BP 239,
54506 Vandoeuvre-lès-Nancy, France
e-mail: Stephane.Genaud@loria.fr; genaud@icps.u-strasbg.fr

E. Jeannot
e-mail: Emmanuel.Jeannot@loria.fr

C. Rattanapoka
Department of Electronics Engineering Technology,
College of Industrial Technology,
King Mongkut's University of Technology North Bangkok, Bangkok, Thailand
e-mail: choopanr@kmutnb.ac.th

the failure detection speed. We retain the *binary round-robin protocol* for its failure detection speed, and we propose a variant of this protocol which is more reliable than the application execution in any case. Experiments involving of up to 256 processes, carried out on Grid'5000, show that the real detection times closely match the predictions.

Keywords Grid computing · Middleware · Parallelism · Fault-tolerance

1 Introduction

Many research works have been carried out these last years on the concept of *grid*. Though the definition of grid is not unique, there are some common key concepts shared by the various projects aiming at building grids. A grid is a distributed system potentially spreading over multiple administrative domains which provides its users with a transparent access to resources. The big picture may represent a user requesting some complex computation involving remotely stored data from its basic terminal. The grid middleware would then transparently query available and appropriate computers (that the user is granted access to), fetch data and eventually transfer results to the user.

Existing grids, however, fall into different categories depending on needs and resources managed. At one end of the spectrum are what is often called "institutional grids", which gather well identified users and share resources that are generally costly but not necessarily numerous. At the other end of the spectrum are grids with numerous, low-cost resources with few or no central system administration. Users are often the administrators of their own resource that they choose to share. Numerous projects have recently emerged in that category [7,11,20], which have in common to target desktop computers or small clusters. P2P-MPI is a grid middleware that falls into the last category. It has been designed as a peer-to-peer system: each participant in the grid has an equal status and may alternatively share its CPU or request other CPUs to take part to a computation. The proposed programming model is close to MPI. We give a brief overview of the system in Sect. 2 and a longer presentation can be found in [14]. P2P-MPI is particularly suited to federate networks of workstations or unused PCs on local networks.

In this context, a crucial point is fault management, which covers both *fault tolerance* for applications and *failure detection*.

Concerning fault tolerance, almost all projects have adopted approaches based on checkpoint and restart mechanisms [1]. The problem of this approach is that it requires a reliable server to store checkpoints. In order to solve this issue another approach—the one adopted here, deals with replication of computational processes.

When a failure is detected the application must take recovery actions. For instance, if a node fails the other processes have to stop sending messages to this failed resource. Therefore, the application must be notified of failures by an efficient detection system. A good fault detector has to avoid several pitfalls arising when targeting large-scale distributed environments. We consider that the main issues to be addressed are (1) *scalability* since the fault detection system should work up to hundreds of processors, which implies keeping the number of messages exchanged small while having the time

needed to detect a fault acceptable, (2) *accuracy* means the failure detection should detect all failures and failures detected should be real failures (no false positive), (3) *reliability* means that the failure detection is robust enough to keep making accurate detections despite the failures of some of its detectors.

The contribution of this paper is twofold. First, we model a program execution duration depending on the replication degree. We are able to compute the probability of a correct execution of a program according to different parameters of the environment (sequential execution time, number of processes, failure rate, etc.). We also give an expression of the optimal replication degree that leads to the minimal failure probability and an interval of the replication degree required to keep the failure probability of an execution under a given threshold. Second, we analyze the advantages and drawbacks of the failure detectors of the literature for a real implementation. We pay special attention to the reliability of the failure detection service. We also consider the failure detection speed. We show that existing solutions not always fulfill our requirements regarding the accuracy and the reliability of the fault detector. Therefore, we propose a more reliable variant of the binary round-robin protocol (BRR), at the cost of extra control messages.

This paper is organized as follows. Section 2 is a short overview of P2P-MPI which outlines how replication of processes increases an application execution *robustness* (the more robust an execution is, the less chance it has to fail). In Sect. 3, we discuss in details how replication enables fault tolerance. The discussion concludes with an expression of fault tolerance as a failure probability depending on the replication degree and on the failure events rate. To be effective, a system that provides fault tolerance must rely on an effective failure detection service. Section 4 discusses the issues with the design of a failure detection service. The section includes a review of the existing techniques to design a reliable fault detection service, and a discussion of strengths and weaknesses of candidate solutions considering P2P-MPI's requirements. In particular, we require the service to be far more reliable than the application execution. We underline the trade off between reliability and detection speed and we propose a variant of an existing protocol to improve reliability. Finally, this section about failure detection describes how these principles have been implemented. Last, in Sect. 5, we present experimental results regarding the detection speed. A first experiment conducted in a real distributed environments (up to 256 processes distributed over three sites at a nation wide scale) shows the detection time for a failure using the two protocols retained in P2P-MPI. A second experiment is carried out with an application using replication, and we evidence that the type of application executed has no influence on the failure detection time.

2 P2P-MPI Overview

P2P-MPI's overall objective is to provide a *grid* programming environment for parallel applications. As the scope of this paper is fault management of P2P-MPI, we do not describe this framework in details. We refer the reader to [14] for a precise description. In short, P2P-MPI has two facets: it is a communication library with a

parallel programming API provided to programmers. The other facet is a middleware. As such, it has the duty of offering appropriate system-level services to the user, such as finding requested resources, transferring files, launching remote jobs, etc.

2.1 API

Most of the other comparable projects cited in introduction (apart from P3 [20]) enable the computation of jobs made of independent tasks only, and the proposed programming model is a client-server (or RPC) model. The advantage of the client-server model lies in its suitability to distributed computing environments but lacks expressiveness for parallel constructs. P2P-MPI offers a more general programming model based on message passing, of which the client-server can be seen as a particular case.

P2P-MPI is an MPJ implementation. MPJ (Message Passing for Java) [8] is a recommendation issued from the Java Grande Forum which is an adaptation for Java of the MPI specification [21] targeting C, C++ and Fortran. Although we have chosen Java for portability purpose, the primitives are quite close to the original MPI specification. This means a P2P-MPI user benefits from a communication library exposing an MPI-like API. Regarding that aspect, P2P-MPI competes with projects such as MPJ-Express [3] and MPJ/Ibis [5,23].

2.2 Middleware

P2P-MPI's middleware is based on a peer-to-peer (P2P) infrastructure. The P2P infrastructure is maintained at some hosts by peers playing the role of *supernodes*. (The terminology and the protocols we use are close to those of Gnutella.) A supernode is a necessary entry point for boot-strapping a peer willing to join the P2P infrastructure.

A user simply makes its computer join a P2P-MPI grid by typing `mpiboot`, which starts a local background process called MPD. Assuming the MPD knows at least one supernode, it registers to it and that way represents the local resource as a peer in the P2P infrastructure. At the same time, the MPD retrieves from the supernode a list of peers that it will maintain in its internal cache. The MPD's roles are mainly:

- to maintain the peer membership to the infrastructure by joining on startup, and by subsequently sending periodic alive signals to supernodes,
- to manage the local peer's neighborhood knowledge: each neighbor in the cache is periodically ping'ed to assess network latency to it,
- when an application requests a number of resources, it has the charge of coordinating the discovery of peers, the reservation of resources and to organize the job launch. In particular, if some resources leave or join the environment between two executions, P2P-MPI transparently keeps up-to-date the list of usable machines.
- upon a run request from another peer, it acts as a gate-keeper of the local resource by controlling how many processes and applications can be run simultaneously.

Job execution An application execution is typically invoked from the command line, e.g: `p2pmpirun -n n -r r prog`. In this example, the mandatory arguments are the *n* processes requested and the `prog` program to run. The optional argument *r* is the replication degree (see below) by which the user requests the middleware to handle the program execution with some fault tolerance.

2.3 Fault Tolerance

Fault-tolerance of MPI applications is difficult to handle because during an MPI application execution, a single failure of any of the processes makes the whole application fail. This is particularly important in a grid context, where failures are far more frequent than on supercomputers, the traditional environments for high-performance applications. Solutions commonly proposed to this issue are *checkpoint and restart* solutions, which use some *rollback-recovery* protocol. A rollback recovery protocol can be 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* usually completed by asynchronous checkpoints. Message logging consists in storing non-deterministic events (e.g. message arrivals) on a reliable media, so that in case of failure, a failed process is re-executed from its last checkpoint and further messages are replayed from the log. Examples of coordinated checkpointing can be found in the early CoCheck project [22], as well as the in popular LAM/MPI implementation [18]. The extension of MPICH proposed in MPICH-V2 [6] is an example of rollback-recovery based on message logging. However, almost all checkpoint and restart strategies require the presence of some reliable resources to store the system states. This approach does not fit into our P2P framework, for which we do not want to rely on a common network file system or dedicated checkpoint servers. Note, however, that some recent work on check-point and restart get rid of this constraint by distributing the checkpoints to other nodes in a peer-to-peer fashion [24].

2.3.1 Replication, Background and Assumptions

In contrast, P2P-MPI proposes fault-tolerance by the means of *replication* of computations. Replication means that any process may have one or several copies running simultaneously on different hosts. The MPI application can survive failures as long as at least one copy of each process has not failed. This does not prevent the application to crash after a number of failures, but increases the application robustness. As we generally do not know if a specific host is more failure-prone than another, we use the same number of copies for all processes which we call the *replication degree*. So, in the above run command, `-r r` means that each MPI process will have *r* copies running simultaneously on distinct hosts.

To the best of our knowledge, MPI/FT [4] is the only project that has proposed process replication (termed *modular redundancy*) to tackle failures in 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. 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. Although is has never been evidence by experiments at a large-scale, this unique coordinator obviously constitutes a bottleneck that limits scalability. In contrast, as we will see in Sect. 2.3.2, the replication protocol in P2P-MPI uses one coordinator per group of replicated process. The failure detection system in P2P-MPI is also different since it is external to the application processes and entirely distributed (see Sect. 4.2).

The advantage of the replication approach is that the source code of the application does not need any modification. Indeed, the communication library transparently handles all extra-communications needed to keep the system in a coherent state. Details regarding the coherence protocol can be found in [14]. In brief, our approach falls in the category called *active replication* [19] in the literature. In this scheme, senders send their messages to all replicas of the destination group (i.e., a logical process in our context, explained hereafter). Our protocol is slightly different in that only one process (the master) sends the messages to all replicas of the destination group. It is well known that active replication requires *atomic broadcast* [12] (or *total order broadcast*) to insure the coherence of the system. In our context, it is possible to implement such an operation because of our assumptions on the environment.

- We only consider **fail-stop failures**: a failed process stops performing any activity including sending, transmitting or receiving any message.
- 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.

2.3.2 Replication in P2P-MPI

Applied to the context of P2P-MPI, the destination group in a send operation is the group of replicas (process copies) running a same code. Such a group is called a *logical process*. Let us briefly sketch how P2P-MPI handles communication when replication is used. This helps to understand the differences in the execution model between a program with and without replication, and in particular how it impacts performance.

To illustrate how the system manages communications with replication, consider a user program sending a message from P_0 to P_1, as depicted in Fig. 1. In each logical process, only one replica called *master*, is in charge of send instructions. On Fig. 1 the send instruction is from P_0 to P_1, and we note P_0^0 the replica which is assigned the master's role.

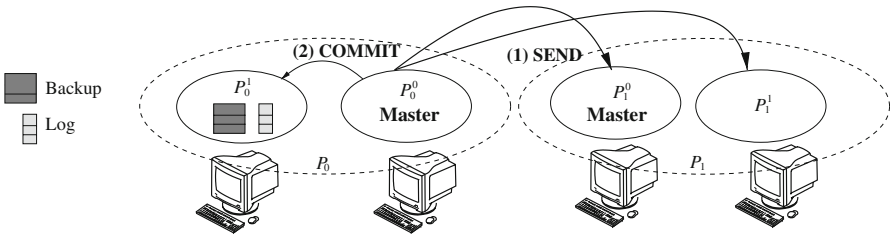

Fig. 1 A message sent from logical process P_0 to P_1

The other processes do not send the message over the network, but store it in their memory. When a replica reaches a send instruction, two cases arise depending on the replica 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 that the message has been correctly transmitted by sending a commit message. The commit message is received and logged by the replicas.
– if the replica is not the master, it first looks up its log table to see wether 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 a 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 message identifier in its log and if the message has been stored, it removes it from the backup table.

2.3.3 Replication Overhead

From the above description of the replication protocol, we see that replication incurs an overhead. Each message normally sent once, is sent to all replicas of the destination process when executing a program with replication. An extra step is necessary as well, for the master to commit the message sent to its own replicas by sending them 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 Fig. 2 the time taken for the round trip time of 1,000 message exchanges, with different replication degrees and message sizes. If we consider t_1 the execution time without replication (t_1), we observe that the overhead for replication degree r is a bit less than $r\ t_1$. 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.

In the next section, we study fault-tolerance with the help of a failure model taking execution time as one of its arguments. Thus, the above figures will help us instantiate our model parameters with realistic values.

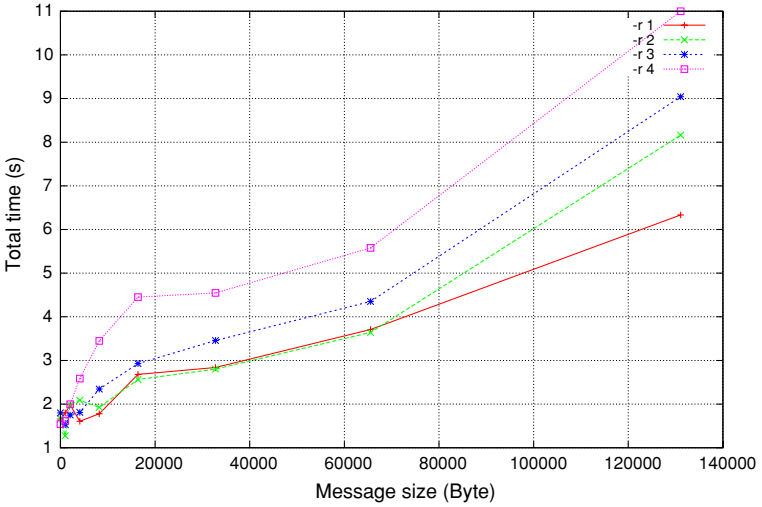

Fig. 2 Time spent for 1,000 ping-pong messages with different replication degrees

3 Fault Tolerance with Replication

In this section, we quantify the failure probability of an application and how much replication improves the application robustness.

3.1 Failure Model

Our failure model follows previous studies on the availability of machines in wide-area environment such as the one of Nurmi et al. [15]. Such studies show that the Weibull distribution effectively model machine availability. Based on Nurmi et al. [15] the probability that a machine fails before time t is given by:

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

where $\lambda > 0$, the inverse of the scale 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 $\delta = 1$. Therefore this section is more general than only dealing with exponentially distributed failures. It is out of the scope of this paper to compute the actual value of the Weibull parameters, as it is highly dependent on the environment. We will therefore tackle the general case where λ and δ are undetermined. However, as shown in [15] the determination of these parameters can easily be done by studying the traces of the environment.

Assuming that failures are independent events, occurring with the same probability on each host, the probability that a host fails before time t is denoted $f(t)$. Thus, the probability that a p process MPI application during t seconds without replication crashes is

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

Now, when an application has its processes replicated with a replication degree r, a crash of the application occurs if and only if at least one logical process has all its r copies failed. The probability that all of 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

$$\begin{aligned} P_{\text{app}(p,r)} &= 1 - (1 - f(t)^r)^p \\ &= 1 - (1 - (1 - e^{-(t\ \lambda)^\delta})^r)^p \text{ using (1)} \end{aligned} \tag{3}$$

where t is the duration of the MPI application on p processors using r replicas (using $p \times r$ processors in total). Since the replication incurs an overhead, we must provide an expression of the execution duration depending on the replication. We propose a model of the duration execution time derived from Amdahl's law [2]. Consider:

- T_1 the execution time of the sequential version of the application,
- $T(p, 1)$ the execution on p processes, without replication ($r = 1$).
- $T(p, r)$ execution on p processes, with replication degree r.

Following Amdahl's law, where β is the portion that can be parallelized, we have the relation:

$$T(p, 1) = \left((1 - \beta) + \frac{\beta}{p}\right) \cdot T_1 \tag{4}$$

To take into account the overhead of replication, we make the assumption that the replication overhead is linear in r. This is a bit simpler than the reality observed in Fig. 2, but this can be considered as an upper bound. Thus, we model the time of a replicated application by adding the overhead $k(r - 1)$, where k is a constant that has to be tuned to reflect the execution platform characteristics.

$$T(p, r) = \left((1 - \beta) + \frac{\beta}{p}\right) \cdot T_1 + k(r - 1) \tag{5}$$

Finally, combining (3) and (5) we have:

$$P_{\text{app}(p,r)} = 1 - (1 - (1 - e^{-(\lambda(((1-\beta)+\frac{\beta}{p})\cdot T_1 + k(r-1)))^\delta})^r)^p$$

3.2 Optimal Replication Degree

Hence, we see that replication increases the robustness of the execution, while the overhead linked to the replication lengthens the execution and therefore increases the risk of a failure occurrence. The question is thus to determine the optimal replication degree.

The best trade-off strongly depends on the parameters of the model. Figure 3 plots the failure probability P_{app} for an application using 10 processes, with a reference sequential execution time of 10 s ($T_1 = 10$), $\lambda = 10^{-1}\text{s}^{-1}$ and $\delta = 1$. These values are mostly chosen to exemplify the convexity of the curve.

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).

An interesting question is when p is fixed, what is the optimal value for replication. In order to find out this optimal value, we have to study:

$$g(r) = 1 - (1 - (1 - e^{-(\lambda(((1-\beta)+\frac{\beta}{p})\cdot T_1 + k(r-1)))^{\delta}})^r)^p$$

g has the form:

$$g(r) = 1 - (1 - (1 - e^{-(a+br)^{\delta}})^r)^p$$

where

$$a = \lambda\left(\left((1-\beta)+\frac{\beta}{p}\right)\cdot T_1 - k\right) \tag{6}$$

and

$$b = \lambda k \tag{7}$$

Let us consider the following function: $h(x) = 1 - e^{\frac{\ln(1-x)}{p}}$, h is clearly an increasing function. Therefore g is minimum when $g_2(r) = h(g(r))$ is minimum.

Let us find the minimum of g_2. We have:

$$g_2(r) = (1 - e^{-(a+br)^{\delta}})^r$$

We compute the first and second derivatives $g_2'(r)$ and $g_2''(r)$. We have reported these computations in appendix for sake of readability.

We have the constraints that $a > 0$, $b > 0$, $r \geq 1$ and $\delta \in]0, 1]$. A study of this function with *Maple* [10] shows that when $b < 0.3$ (i.e. $\lambda < \frac{3}{10k}$, which is a very realistic hypothesis), g_2'' is always positive. This means that g_2 is convex and is minimum for

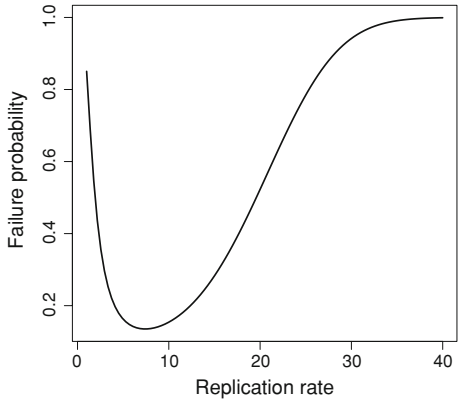

Fig. 3 $P_{\text{app}(10,\,r)}$ with $\delta = 1$, $\lambda = 10^{-1}$, $k = 1$, $T_1 = 10$

$$g_2'(r) = 0 \Leftrightarrow \frac{\left(1-e^{-(a+br)^{\delta}}\right)^{r}\left(\ln\left(1-e^{-(a+br)^{\delta}}\right)\left(e^{(a+br)^{\delta}}-1\right)+r\,(a+br)^{\delta-1}\,\delta b\right)}{e^{(a+br)^{\delta}}-1}$$
$$= 0 \Leftrightarrow \ln\left(1-e^{-(a+br)^{\delta}}\right)\left(e^{(a+br)^{\delta}}-1\right)+r\,(a+br)^{\delta-1}\,\delta b = 0$$

This equation does not have root that can we expressed using a closed form. We suggest using a binary search to find the root.

However, it is simple to apply this general expression to specific cases. For instance, the example of Fig. 3 can be determined numerically. We search for $g_2' = 0$, which, with the values of the example, is equivalent to:

$$\ln(1-e^{-0.09-1/10\,r})(e^{0.09+1/10\,r}-1)+1/10\,r = 0$$

This equation has root $r = 7.411621064$, for which g_2 and g reach their minimum. The minimum failure probability of the application execution will thus be obtained for a replication degree of 7 or 8. We can immediately determine that the minimum is obtained for 7 ($\min(g(7), g(8)) = g(7) = 0.1358499562$).

The most useful question for the user is probably, "which replication degree leads to a failure probability less than a given threshold ϵ?".

$$\begin{aligned} P_{\text{app}(p,r)} &\leq \epsilon \\ &\Leftrightarrow 1-(1-(1-e^{-(\lambda(((1-\beta)+\frac{\beta}{p})\cdot T_1+k(r-1)))^{\delta}})^{r})^{p} \leq \epsilon \\ &\Leftrightarrow 1-(1-(1-e^{-(a+br)^{\delta}})^{r})^{p} \leq \epsilon \end{aligned} \tag{8}$$

where a and b are defined in Eq. 6 and 7. Finding the replication ratio that satisfies Eq. 8 can be done using Maple. For instance finding the minimum and maximum replication ratio such that the probability of failure is under 0.3 for the case described in Fig. 3, is solved using the following Maple code:

```
with(Optimization); Minimize(r,{1 - (1 - (1 -
exp(-(a+b*r)^d))^r)^p<=e,
```

```
          a=0.09,b=0.1,r>=1,a+b*r>=0,
          e=0.3,d=1,p=10});
Maximize(r,{1 - (1 - (1 - exp(-(a+b*r)^d))^r)^p<=e,
          a=0.09,b=0.1,r>=1,a+b*r>=0,
          e=0.3,d=1,p=10});
```

which gives the following results: $r \in [4.065, 12.365]$. This means a replication factor between 5 and 12 leads to a failure probability of the program being always under 0.3.

4 Fault Detection

Let us now look at the failures from the system point of view. For the replication to work properly, each process must get, in a definite period, a global knowledge of other processes states to prevent incoherence. For instance, running processes 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 to a local service called the *fault detection service* (noted FD in the following). In each host, this service is responsible to notify the local application process(es) of failures happening on co-allocated processes.

Figure 4 gives an overview of the system while an application using replication is running. In this example, we depict three logical processes (numbered 1, 2 and 3). In this snapshot, the process 1 issues a send to 2 and 3, depicted by thicker arrows on the figure. As explained previously, the replication involves extra communications to each replica (drawn as dashed arrows).

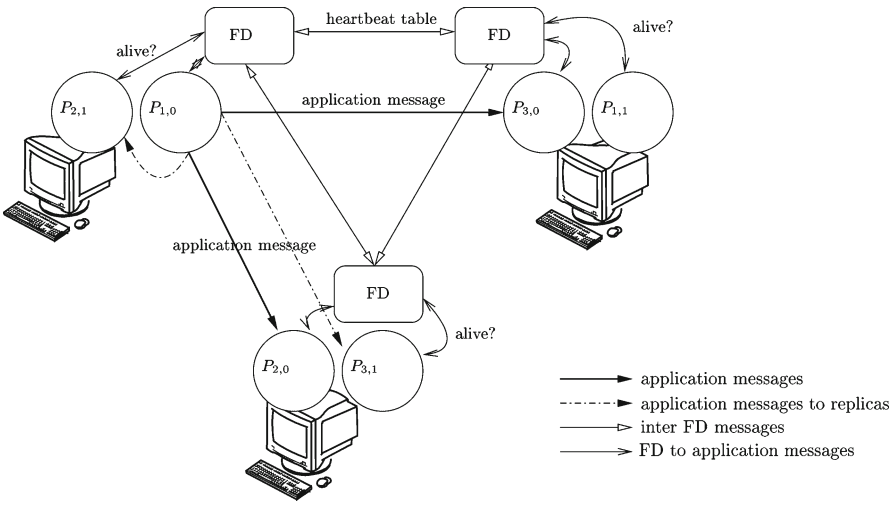

Fig. 4 System overview: the application processes (*circles*) exchange messages, while the failure detectors periodically send *alive* queries to the local application processes, and send their heartbeat table to some other failure detectors according to a gossiping protocol

At the same time, the failure detectors exchange information about process health. Each failure detector periodically (every 30 s in the default configuration) queries the local application processes to check they are alive. If the process answers positively, the failure detector increases a counter called *heartbeat* for that application and report it to a table that will be exchanged with other failure detectors. The way these tables are exchanged is the object of the rest of this section. We will explain in details the design of the failure detectors with respect to the important desired feature which is scalability. For this discussion we first need to review state of the art proposals concerning fault detection since some of these concepts are the basis for our work.

4.1 Fault Detection: Background

Failure detection services have received much attention in the literature and since they are considered as first class services of distributed systems [9], many protocols for failure detection have been proposed and implemented. Two classic approaches are the *push* and *pull* models discussed in [13], which rely on a centralized node which regularly triggers push or pull actions. Though they have proved to be efficient on local area networks, they do not scale well and hence are not adapted to large distributed systems such as those targeted for P2P-MPI.

A much more scalable protocol is called *gossiping* after the gossip-style fault detection service presented in [17]. It is a distributed algorithm whose informative messages are evenly dispatched among the links of the system. In the following, we present this algorithm approach and its main variants.

A gossip failure detector is a set of distributed modules, one module residing at each host to monitor. 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 repeats the following steps: first it increases its own heartbeat and then sends its table to some other modules. 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. 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 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 between 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 $T_{\text{consensus}}$, is the time interval after which consensus is reached about a failed node.

Notice that a major difficulty in gossiping implementations lies in the setting of T_{cleanup}: it is easy to compute a lower bound, referred to as $T^{\min}_{\text{cleanup}}$, 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 [16], namely random, round-robin and binary round robin. We briefly review advantages and disadvantages of the original and modified gossip based protocols and what is to be adapted to meet P2P-MPI's requirements. Notably, we pay attention to the detection time ($T^{\min}_{\text{cleanup}}$) and reliability of each protocol.

Random. In the gossip protocol originally proposed in [17], each module randomly chooses at each step, the hosts it sends its table to. In practice, random gossip evens the communication load amongst 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. To minimize this risk, the system implementor can increase T_{cleanup} at the cost of a longer detection time.

Round-Robin (RR). This method aims to make gossip messages traffic more uniform by employing a deterministic approach. In this protocol, gossiping takes place in definite round every T_{gossip} seconds. In any one round, each node will receive and send a single gossip message. The destination node d of a message is determined from the source node s and the current round number r.

$$d = (s + r) \bmod n, 0 \leq s < n, 1 \leq r < n \tag{9}$$

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. For a 6 nodes system, the set of communications taking place is represented in the table in Fig. 5.

This protocol guarantees that all nodes will receive the updated heartbeat status of a given node within a bounded time. The information about a node 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. Hence, knowing n we can deduce the minimum cleanup time, depending on an integer number of rounds r such that:

$$T^{\min}_{\text{cleanup}} = r \times T_{\text{gossip}} \text{ where } r = \lceil \rho \rceil \ , \ \frac{\rho(\rho+1)}{2} = n$$

r	$s \to d$
1	$\boxed{0} \to \underline{1}$, $1 \to 2$, $2 \to 3$, $3 \to 4$, $4 \to 5$, $5 \to 0$
2	$0 \to \underline{2}$, $1 \to \underline{3}$, $2 \to 4$, $3 \to 5$, $4 \to 0$, $5 \to 1$
3	$0 \to \underline{3}$, $1 \to \underline{4}$, $2 \to \underline{5}$, $3 \to \underline{0}$, $4 \to 1$, $5 \to 2$
4	$0 \to 4$, $1 \to 5$, $2 \to 0$, $3 \to 1$, $4 \to 2$, $5 \to 3$
5	$0 \to 5$, $1 \to 0$, $2 \to 1$, $3 \to 2$, $4 \to 3$, $5 \to 4$

Fig. 5 Communication pattern in the round-robin protocol ($n = 6$)

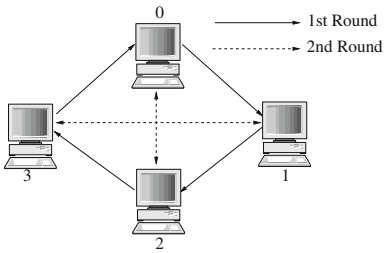

Fig. 6 Communication pattern in the binary round-robin protocol ($n = 4$)

For instance in Fig. 5, three rounds are required to inform the six nodes of the initial state of node 0 (boxed). In the table of Fig. 5, we have underlined the nodes to show at which round they receive the information. For example, node 1 is underlined in the first row because it first gets the information from node 0 during round 1.

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:

$$d = (s + 2^{r-1}) \mod n, \quad 1 \leq r \leq \lceil \log_2(n) \rceil \tag{10}$$

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

From our experience (also observed in experiments of Sect. 5), in an asynchronous system, provided that we are able to make the distributed FD start nearly a the same time, i.e. within a time slot shorter (logical time) than a cycle, and that the time needed to send a heartbeat is less than T_{gossip}, a good choice for T_{cleanup} is the smallest multiple of $T^{\min}_{\text{cleanup}}$, i.e. $2 \times \lceil \log_2(n) \rceil \times T_{\text{gossip}}$. This allows the system to handle the frequent and correct situation where, due to asynchronism and transmission time, the last messages sent within a cycle c on source nodes arrive at cycle $c+1$ on their corresponding receiver nodes.

Note, however, that the elimination of redundant gossip alleviates network load and accelerates heartbeat status dissemination at the cost of an increased risk of false detections. Figure 6 shows a 4 nodes system. From (10), we have that 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. This point is thus to be considered in the protocol choice.

4.2 Fault Detection in P2P-MPI

From the previous description of state of the art proposals for failure detection, we retain BRR for its low bandwidth usage and quick detection time despite its relative fragility. With this protocol often comes a consensus phase, which follows a failure detection, to keep the coherence of the system (all nodes make the same decision about

other nodes states). Consensus if often based on a voting procedure [16]: in that case all, nodes transmit, in addition to their heartbeat table, an extra $(n \times n)$ matrix M. The value $M_{i,j}$ indicates what is the state of node i according to node j. Thus, a FD suspecting a node to have failed can decide the node is really failed if a majority of other nodes agree. However, the cost of transmitting such matrices would induce an unacceptable overhead in our case. For a 256 nodes system, each matrix represents at least a 64 Kib message (and 256 Kib for 512 nodes), transmitted every T_{gossip}. We replace the consensus by a pragmatic procedure, called *ping procedure* in which a node suspecting another node to have failed directly pings this node to confirm the failure. If the node is alive, it answers to the ping by returning its current heartbeat.

This is an illustration of problems we came across when designing the P2P-MPI's fault detection service. We now describe the requirements we have set for the middleware, and which algorithms have been implemented to fulfill these requirements.

4.2.1 Assumptions and Requirements

Our assumptions about the environment are naturally the same as those stated in Sect. 2.3.1. Let us precise what is a fail-stop failure in our context. This type of failure is characterized by a lack of response during a given delay from a process enrolled for an application execution. A fault can have three origins: (1) the process itself crashes (e.g. the program aborts on a DivideByZero error), (2) the host executing the process crashes (e.g. the computer is shut off or rebooted), or (3) the fault-detection monitoring the process crashes and hence no more notifications of aliveness are reported to other processes.

From the assumption that messages are delivered in a finite time and that the communication channel is reliable, we enforce our assumption by expecting that the time required to transmit a message between any two hosts is less than T_{gossip}. Yet, we tolerate unusually long transmission times (due to network hang-up for instance) thanks to a parameter $T_{\text{max\_hangup}}$ set by the user (actually T_{cleanup} is increased by $T_{\text{max\_hangup}}$ in the implementation).

In addition, we have the following legitimate requirements for a middleware whose objective is to be able to scale up to hundreds of nodes. We demand its fault detection service 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. Hence, the accuracy property mentioned in the introduction is linked to the reliability.

4.2.2 Design Issues

Until the present work, P2P-MPI's fault detection service was based on the random gossip algorithm. In practice, however, we were not fully satisfied with it because of its non-deterministic detection time.

Table 1 Sample values where the application is more reliable than FD

λt	$P_{\text{app}(2,2)}$	$P_{\text{brr}(4)}$
1	6.3×10^{-1}	7.4×10^{-1}
10^{-1}	1.8×10^{-2}	3.2×10^{-2}
10^{-2}	1.9×10^{-4}	3.9×10^{-4}

As stated above, the BRR protocol is optimal with respect to bandwidth usage and fault detection delay. The low bandwidth usage is caused by the small number of nodes (we call them *sources*) in charge of informing a given node by sending to it gossiping messages: in a system of n nodes, each node has at most $\log_2(n)$ sources. Hence, BRR is the most fragile system with respect to the simultaneous failures of all sources for a node, and the probability that this situation happens is not always negligible: In the example of the 4 nodes system with BRR, the probability of failure can be counted as follows. Consider the failure probability $f(t)$ for each individual node over a period of length t ($t < T_{\text{cleanup}}$), as defined in Eq. 1. Let $P(i)$ be the probability that i nodes simultaneously fail during t. In the case 2 nodes fail, if both are source nodes then there will be a node that cannot get any gossip messages. Here, there are 4 such cases, which are the failures of {2,3},{0,3},{0,1} or {1,2}. In the case 3 nodes fail, there is no chance the FD can resist. There are $\binom{4}{3}$ ways of choosing 3 failed nodes among 4, namely {1,2,3},{0,2,3},{0,1,3},{0,1,2}. And there is only 1 case 4 nodes fail. Finally, the FD's failure probability is $P_{\text{brr}(4)} = P(4) + P(3) + P(2) = f(t)^4 + \binom{4}{3} f(t)^3(1 - f(t)) + 4f(t)^2(1 - f(t))^2$. In this case, the comparison between the failure probability of the application ($p = 2, r = 2$) and the failure probability of the BRR for $n = 4$ on some typical failure rates (for a shape parameter $\delta = 1$), exhibit values such those in Table 1. This means the application is more resistant than the fault detection system itself. Even if the FD failure probability decreases quickly with the number of nodes, the user may wish to increase FD reliability by not eliminating all redundancy in the gossip protocol.

4.3 P2P-MPI Implementation

Users have various needs, depending on the number of nodes they intend to use and on the network characteristics. In a reliable environment, BRR is a good choice for its optimal detection speed. For more reliability, we may wish some redundancy and we allow users to choose a variant of BRR described below. 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).

4.3.1 Double Binary Round-Robin (DBRR)

We introduce the double binary round-robin (DBRR) protocol which detects failures in a delay asymptotically equal to BRR ($O(\log_2(n))$ and acceptably fast in practice, while re-reinforcing robustness of BRR. The idea is simply to avoid having one-way connections only 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

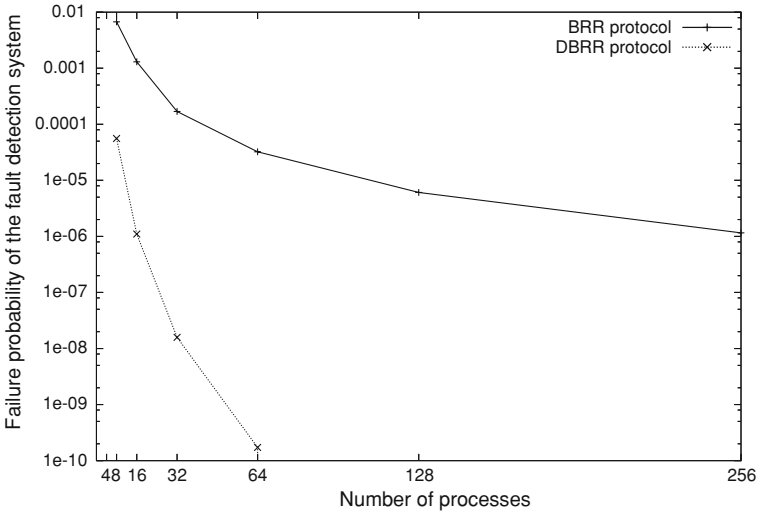

Fig. 7 Failure probabilities of the FD system using BRR and DBRR ($\delta = 1, \lambda t = 10^{-1}$)

back by applying BRR in a counterclock-wise direction. The destination node for each gossip message is determined by the following relation:

$$d = \begin{cases} (s + 2^{r-1}) \mod n \\ \qquad \text{if } 1 \leq r \leq \lceil \log_2(n) \rceil \\ (s - 2^{r-\lceil \log_2(n) \rceil - 1}) \mod n \\ \qquad \text{if } \lceil \log_2(n) \rceil < r \leq 2\lceil \log_2(n) \rceil \end{cases} \tag{11}$$

The cycle length is $2\lceil \log_2(n) \rceil$ and hence we have $T^{\min}_{\text{cleanup}} = 2\lceil \log_2(n) \rceil \times T_{\text{gossip}}$. With the same assumptions as for BRR, we set $T_{\text{cleanup}} = 3\lceil \log_2(n) \rceil \times T_{\text{gossip}}$ for DBRR.

To compare BRR and DBRR reliability, we can count following the principles of Sect. 4.2.2 but this quickly becomes difficult for a large number of nodes. Instead, we simulate 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 sources nodes to a destination node, which implies a FD failure. In Fig. 7, we repeat the simulation for 5.8×10^9 trials with $\delta = 1$, $\lambda = 10^{-3}\text{s}^{-1}$ 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 128 and 256 (this is why these points are not plotted).

4.3.2 Automatic Adjustment of Initial Heartbeat

The choice of an appropriate protocol is important but not sufficient to get an effective implementation. We also have to correctly initialize the heartbeating system so that the delayed starts of processes are not considered failures.

In the startup phase of an application execution (contained in `MPI_Init`), the submitter process first queries advertised resources for their availability and their will to accept the job. The submitter constructs a table, called the communicator, in which available resources are assigned a rank (an integer). In P2P-MPI, the submitter always has rank 0. Then, the submitter sends the communicator in turn to all participating peers. The remote peers acknowledge the communicator by returning TCP sockets where the submitter can contact their file transfer service. It follows the transfer of executable code and input data. Once a remote node has completed the download, it starts the application which registers with its local FD instance.

This causes the FDs to start asynchronously and because the time of transferring files may well exceed T_{cleanup}, the FD should (1) not declared nodes that have not yet started their FD as failed, and (2) should start with a heartbeat value similar to all others at the end of the `MPI_Init` barrier. The idea is thus to estimate on each node, how many heartbeats have been missed since the beginning of the startup phase, to set the local initial heartbeat accordingly. This is achieved by making the submitter send to each node i, together with the communicator, the time Δt_i spent sending information to previous nodes. Figure 8 illustrates the situation. Note that we cannot compute the duration of the communication between the submitter and any process of rank i, as we have no global clock ($tr_i - ts_1$ would make no sense).

We note ts_i, $1 \leq i < n$ the date when the submitter sends the communicator to process i, and tr_i the date when peer i receives the communicator. Each peer also stores the date T_i at which it registers with its local FD, which is also the time when the application begins. Let us define $\Delta t_i = ts_i - ts_1$ the time the submitter has spent sending data since the beginning (ts_1) and before it sends its data to a process i ($1 \leq i < n$). We also define comm_i the time it takes for submitter to send its data to process i. Therefore, a process i can start its execution after $T_i - tr_i + \Delta t_i + \text{comm}_i$ units of time. For example on Fig. 8, the last process with rank $n-1$, receives Δt_{n-1} from the

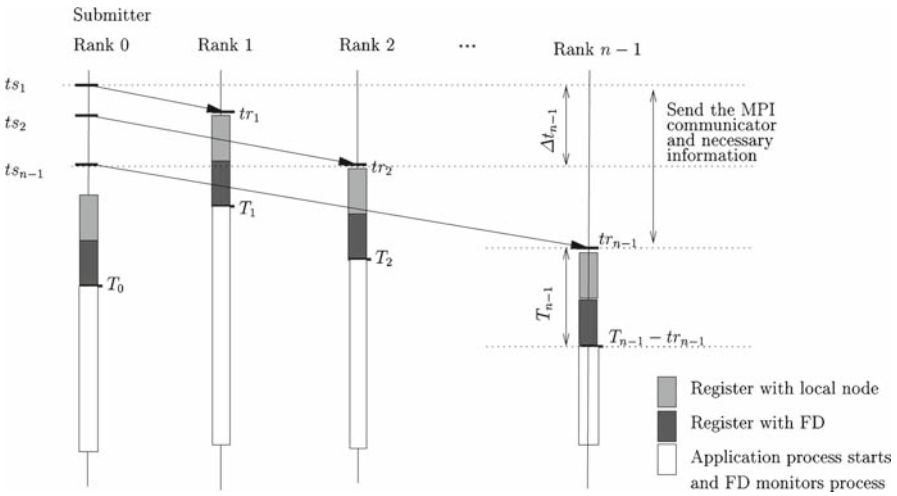

Fig. 8 Application startup

submitter, i.e. the time the submitter has spent sending data since the beginning (ts_1). It then simply adds the time needed for administrative registration to the local service ($T_{n-1} - tr_{n-1}$) before the application can start locally.

A process i can compute its initial heartbeat accordingly, by setting:

$$h_i = \lceil (T_i - tr_i + \Delta t_i + \text{comm}_i)/T_{\text{gossip}} \rceil, \quad 1 \leq i < n \tag{12}$$

while the submitter adjusts its initial heartbeat to $h_0 = \lceil (T_0 - ts_1)/T_{\text{gossip}} \rceil$.

Although it is possible to estimate the value comm_i, we can simplify the previous expression with the assumption that $\text{comm}_i = \text{comm}_j$ for any process i and j.

Indeed, a failure is detected if the initial heartbeats differ by more than a given threshold, that is:

$$h_i - h_j = \frac{(T_i - tr_i + \Delta t_i + \text{comm}_i) - (T_j - tr_j + \Delta t_j + \text{comm}_j)}{T_{\text{gossip}}} > T_{\text{cleanup}}$$

$T_i - tr_i$ is the time to register, which can be considered constant whatever the host, so that $T_i - tr_i = T_j - tr_j$. Therefore, the difference between the complete and the simplified expression is $(\text{comm}_i - \text{comm}_j)/T_{\text{gossip}}$, which means that we would detect failures with the original expression not detected by the simplified one only if $\text{comm}_i - \text{comm}_j > T_{\text{cleanup}} \times T_{\text{gossip}}$. T_{gossip} is in the order of the second (.5 s in the default configuration), and for DBRR with 32 processes $T_{\text{cleanup}} = 15$ rounds, one communication would have to be longer by 7.5 s than another one to make a difference. As this situation is unusual, we simplify the computation of the initial heartbeat in:

$$h_i = \lceil (T_i - tr_i + \Delta t_i)/T_{\text{gossip}} \rceil, \quad 1 \leq i < n \tag{13}$$

And even if it leads to a false failure detection, the consensus procedure will eventually prevent the false detection.

Note that we implement a flat tree broadcast to send the communicator instead of any hierarchical broadcast scheme (e.g. binary tree, binomial tree) because we could not guarantee in that case, that intermediate nodes always stay alive and pass the communicator information to others. If any would fail after receiving the communicator and before it passes that information to others, then the rest of that tree will not get any information about the communicator and the execution could not continue.

4.3.3 Application-Failure Detector Interaction

At first sight, the application could completely rely on its FD to decide whether a communication with a given node is possible or not. For instance, in our first implementation of *send* or related function calls (e.g. Send, Bcast) the sender continuously tried to send a message to the destination (ignoring socket timeouts) until it either succeeded or received from its FD a notification that the destination node is down. This allows us to control the detection of network communication interruptions through the FD configuration.

However, there exist firewall configurations that authorize connections from some addresses only, which makes possible that a host receive gossip messages (via other nodes) about the aliveness of a particular destination while the destination is blocked for direct communication. In that case, the send function will loop forever and the application cannot terminate. Our new send implementation simply installs a time-out to tackle this problem, which we set to $2 \times T_{\text{cleanup}}$. Reaching this timeout on a send stops the local application process, and soon the rest of the nodes will detect the process death.

5 Experiments

The objective of the experiments is to evaluate the failure detection speed in a real environment. We carried out two experiments.

In the first experiment (Sect. 5.1), we observe how long it takes for the two protocols BRR and DBRR to detect the failure of an application executed without replication. In this experiment, the failure is simulated by killing all the user's processes on a host. This is the most realistic cause of failure, which happens when a computer is shut off, rebooted or disconnected from the network.

In the second experiment (Sect. 5.2), we test if the failure detection is impacted by the type of application executed. As the failure detection is based on message exchanges between the application and the FD, and between remote FDs, we can suspect a communication-intensive application to affect the failure detection behavior. Thus, this experiment is based on a program execution for which we can set the computation to communication ratio. We run the program with a replication degree of two, we kill one of the application processes, and we observe how much longer is the application execution with fault occurrences as compared to a fault-free execution.

Since the nature of the application itself is of little interest, we have performed both experiments using synthetic benchmarks. The advantage of using such benchmarks is that they are easy to calibrate and the experimental condition are much more controllable which improve the overall reproducibility of the experiments.

5.1 Experiment 1: Fault Detection Speed With BRR and DBRR

5.1.1 Experimental Conditions

The application is simulated by a parallel program whose each process does dummy operations and communications in an infinite loop.[1]

Our testbed is the Grid'5000 platform [2], a federation of dedicated computers hosted across nine campus sites in France, and organized in a virtual private network over

[1] There is no relationship between what the application does and the failure detection in case of a host failure: because the failure detector itself has died, the host is eventually declared failed only because no more heartbeats are received from it. In other words, the detection time would be the same whatever the application.

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

Table 2 Latencies between sites in ms (average/jitter over a day)

	Nancy	Rennes	Nice
Nancy	–	7.64/0.03	11.92/0.02
Rennes	7.80/0.02	–	9.14/0.01
Nice	12.23/0.02	9.11/0.01	–

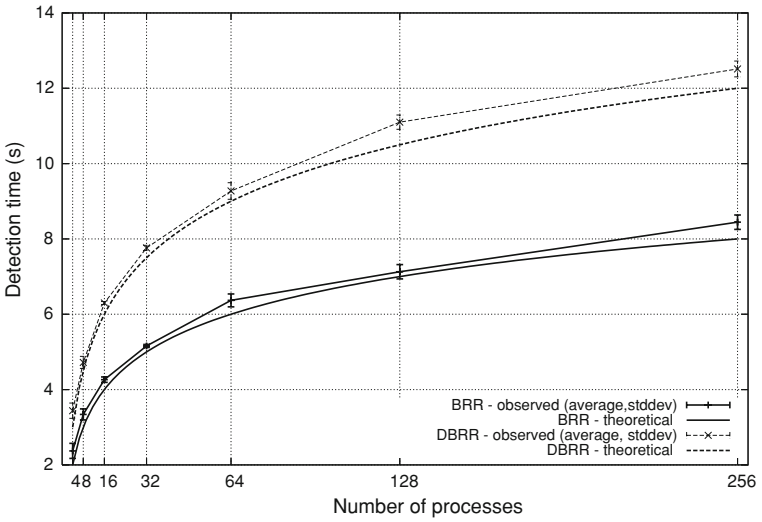

Fig. 9 Time to detect a fault for BRR and DBRR

Renater, the national education and research network. In our experiment, we evenly distribute the processes of our benchmark parallel program across three sites (Nancy, Rennes and Nice). The distance between the sites is about 1,000 km. The sites are interconnected with a 10 Gbps backbone, and typical latencies are reported in Table 2.

The experiment is as follows:

- We start our parallel benchmark without replication.
- After 20 s, we choose a random node, and we kill all processes (application and fault detector) on that node.
- Sometimes after, each process is notified of the failure by its FD. We log the date (called the *detection date*) of the notification at each process.

5.1.2 Results

We are interested in the time interval between the time at which the failure occurs and the detection date. As the FD are fully distributed, this interval is different on every peer. Figure 9 plots the average as well as the standard deviations of the intervals for both protocols, with T_{gossip} set to 0.5 s. Also plotted for comparison is the theoretical prediction, i.e. T_{cleanup} as specified previously ($2\lceil\log_2(n)\rceil$ for BRR and $3\lceil\log_2(n)\rceil$ for DBRR) 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 with the predictions (about 0.5 s) comes from the ping procedure which adds an overhead, and from the rounding to an integer number of heartbeats in (12). This difference is about the same as the T_{gossip} value used, and hence we see that the ping procedure does not induce a bottleneck.

It is also important to notice that no false detection has been observed throughout our tests. Indeed, 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, the execution is very brief, therefore 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 Grid'5000, despite the presence of wide area network links between hosts.

5.2 Experiment 2: Fault Detection Speed Depending on Application Type

5.2.1 Experimental Conditions

In this experiment, we use 32 hosts of a cluster. Each host has two dual core CPUs. We run the application with a replication degree of two. The FD is configured with the default settings: it uses DBRR with $T_{\text{gossip}} = 0.5\,\text{s}$.

Our aim is to observe if the nature of the application modifies the failure detection behavior. The application in our context is mainly characterized by the number of interprocess communications the program performs, or in other words, the communication to computation ratio.

For the experiment, we have written a benchmark that performs iteratively a communication and then a computation step, typically for twenty steps. In the communication step, each process i sends one message of a certain size to its neighbor $i + 1$ modulo the number of processes. The message size is calibrated so that the communication step last one unit of time. The computation step is simulated by a sleep instruction during n unit of times, such that we obtain the desired computation to communication ratio (CCR in the following). In the experiment, we test CCR equal to 1, 2, 5 and 10. Note that CCR = 1 means the time spent in computations is equal to the time spent in communications when there is no replication, but because we use a replication degree of two the time spent in communication is actually longer (see Fig. 2).

The experiment measures on a set of sample executions, the execution durations when a number of faults occurs, and for different CCRs. Varying the number of faults injected aims to see if the overhead is linear in the number of faults. We carry out this experiment for the two common kinds of failures: first, we experiment with the application process failures (one failure corresponds to an MPI process that stops), and then with failures of all processes of a host (both applications processes and the failure detector are stopped). The former case may happen in case of a programming

error, or a specific hardware requirement not matched on a given host. It is therefore less usual than the latter situation where all processes die (like in our first experiment) that can happen after a reboot or a shutdown.

The first experiment is as follows:

- We start our benchmark with the CCR parameter set to 1, 2, 5 or 10.
- After some iterations, we choose a random host, log on that host and kill the first application process found (we cannot distinguish if it is a master or a replica process from an operating system point of view).
- The previous step is repeated in a subsequent iteration if we have not reached the desired number of fault injections.
- We log the execution time for this run with these parameters (CCR and number of faults injected).

We have repeated this experiment from four to eight times for each value in the parameter space. We have also run an equivalent number of fault-free executions with the different values of CCR which serve as reference. We have a total of 91 executions.

5.2.2 Results

Figure 10 shows the average overhead per fault compared to the average time of a fault-free execution. For example, for two faults injected during the execution, the average overhead per fault for CCR = 1 is about 20 s. It means that the average execution duration with CCR = 1 and two faults is 40 s longer than the average of fault-free executions with CCR = 1. This overhead includes:

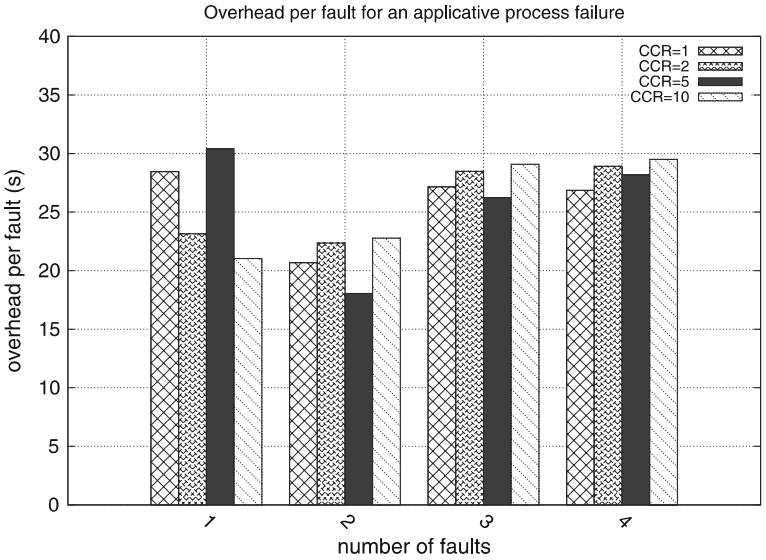

Fig. 10 Overhead due to application process failures, depending on the computation to communication ratio and the number of faults

(t_1) the detection time of the application process failure by the local FD (no response to 'alive?' message on Fig. 4). Such queries are sent every 30 s by the FD. Hence, a detection takes 15 s on average,
(t_2) the failure detection time at every FD by means of the DBRR protocol. Our settings lead to a detection time of $3 \times \log_2(64) \times 0.5 = 9$ s, plus 0.5 s for the ping procedure to confirm the failure,
(t_3) the time for the application processes to be notified of the failure by its local FD, plus the time needed to retransmit possible lost messages and update the communicators.

If time t_3 depends on the application, t_1 and t_2 depends on P2P-MPI's settings. Therefore, depending on the usage and the failure rate such numbers can be lowered at the cost of some management overhead. It is also important to note that these overheads do not depend on the application duration. Hence, for very long applications, the global overhead becomes negligible.

First, we observe that the average overheads are in the expected range, between 9.5 s to 39.5 s if we do not take into account the t_3 part, with an average of 24.5 s. In the graph of Fig. 10 most of the results are between 20 s and 30 s. This shows the accuracy of our modeling. Second, the overhead is almost linear in the number of faults. Third, there is no indication that a given CCR leads to a longer detection time than another. Hence, the application type has no significant influence on the FD behavior.

A last but noteworthy point concerns the process killed. We could expect this process to be a master or a replica with the same probability. However, we found by inspecting the log that, due to the way processes are listed and selected, only one third of the processes killed were master processes in this experiment. It is important since if it is a master, some messages may need to be retransmitted by the replica, and hence the execution is longer and explains why, in this first experiment, the t_3 part is negligible.

To complete this first experiment, we test in the same conditions the detection behavior when all processes of a host fail. The detection is expected to be quicker here since we get rid of t_1. However, the number of processes killed is greater than in the previous experiment since a host can run two to three application processes on its two dual-core CPUs. This number depends on the dynamic allocation of resources made by P2P-MPI, based on the latency from the submitter to peers. Typically, stopping three hosts leads to kill seven or eight processes. Results are shown in Fig. 11.

We first observe as in the previous experiment, that the CCR seems to have no influence on the detection time. Second, the average overhead is greater than 9.5 s (the average of all bars in Fig. 11 is 12.7 s), and hence the t_3 part is bigger than in the previous experiment. There are two reasons for that. First, a host failure corresponds to several simultaneous faults of the application processes, which are master processes in one case out of two on average. Actually, by inspecting our logs, we found that 46% of the processes stopped were masters. As a result, more message retransmissions occurred than in the previous experiment. Second, every FD notifies its local application processes of all the failures. As this notification is sent in a single message, it is done sequentially, and it takes twice as much time as in the single process failure case.

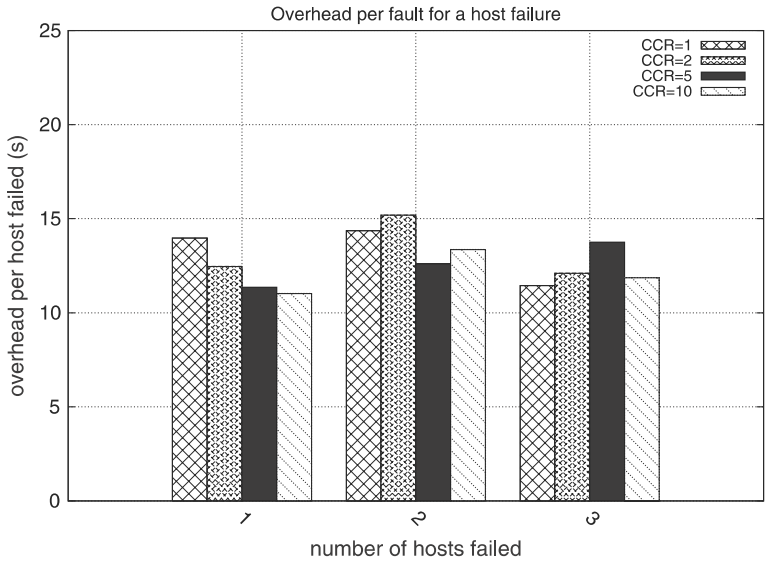

Fig. 11 Overhead due to host failures, depending on the computation to communication ratio and the number of failures

6 Conclusion

We describe in this paper the fault management in P2P-MPI, which addresses the fault tolerance and the fault detection issues.

The first part is an overview of the principles of P2P-MPI. We have focused on two of them: the first principle is replication used as a means to increase robustness of application runs, and the second one is the external monitoring of applications by a specific fault-detection module. For modeling failures, we use the general Weibull distribution, which is known to correctly cope with the reality. Then, we model the failure probability of a P2P-MPI application using replication. We provide methods to compute the optimal replication degree (the one that provides the smallest failure probability) or a replication degree that meets a given level of robustness.

In the second part, we address the failure detection issue. We describe the failure detection service implemented in P2P-MPI and we analyze its performance (scalability, reliability and detection speed). We compare the main protocols recently proposed in the literature regarding their robustness, their speed and their deterministic behavior, and we analyze which is best suited for our middleware. We introduce an original protocol that increases the number of sources in the gossip procedure. It improves the fault-tolerance of the failure detection service, while the detection time remains low. Last, we present the experiments conducted in a real distributed environment. The results show that the fault detection speeds observed in experiments for applications of up to 256 processes, are really close to the theoretical figures, and demonstrate the system scalability. We have also shown that the computation to communication ratio of the parallel program does not influence the failure detection behavior.

Acknowledgments Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see https://www.grid5000.fr).

Appendix

Determining the replication degree requires to study the function

$$g_2(r) = \left(1 - e^{-(a+br)^\delta}\right)^r$$

We compute the first and second derivative of g_2 and we show that g_2'' is always positive using the following Maple code:

```
g:=(1 - exp(-(a+b*r)^d))^r; gp:=simplify(diff(g,r));
gpp:=simplify(diff(gp,r)); with(Optimization);
Minimize(gpp,{a+b*r>=0,a>=0,b>=0,b<=0.3,r>=1,d>=0,d<=1});
```

The results are:

$$g_2'(r) = \frac{\left(1 - e^{-(a+br)^\delta}\right)^r \left(\ln\left(1 - e^{-(a+br)^\delta}\right)\left(e^{(a+br)^\delta} - 1\right) + r\,(a+br)^{\delta-1}\,\delta b\right)}{e^{(a+br)^\delta} - 1}$$

and

$$\begin{aligned}
g_2''(r) = {} & \frac{\left(1 - e^{-(a+br)^\delta}\right)^r}{(a+br)\left(e^{(a+br)^\delta} - 1\right)^2}\left(2\,(a+br)^{\delta-1}\,\delta b e^{(a+br)^\delta} a\right. \\
& - \ln\left(1 - e^{-(a+br)^\delta}\right)(a+br)^{\delta-1}\,\delta b^2 e^{(a+br)^\delta} r \\
& - \ln\left(1 - e^{-(a+br)^\delta}\right)(a+br)^{\delta-1}\,\delta b e^{(a+br)^\delta} a \\
& + \ln\left(1 - e^{-(a+br)^\delta}\right)(a+br)^{\delta}\,\delta b e^{(a+br)^\delta} \\
& - \delta b e^{2\,(a+br)^\delta}\,(a+br)^{\delta}\ln\left(1 - e^{-(a+br)^\delta}\right) \\
& + \ln\left(1 - e^{-(a+br)^\delta}\right)(a+br)^{\delta-1}\,\delta b^2 e^{2\,(a+br)^\delta} r + \left(\ln\left(1 - e^{-(a+br)^\delta}\right)\right)^2 br \\
& - 2\left(\ln\left(1 - e^{-(a+br)^\delta}\right)\right)^2 e^{(a+br)^\delta} a - 2\left(\ln\left(1 - e^{-(a+br)^\delta}\right)\right)^2 e^{(a+br)^\delta} br \\
& + r^3\,(a+br)^{2\delta-2}\,\delta^2 b^3 - 2\,(a+br)^{\delta-1}\,\delta b^2 r - 2\,(a+br)^{\delta-1}\,\delta b a \\
& - r^2\,(a+br)^{\delta-2}\,b^3\delta^2 + r^2\,(a+br)^{\delta-2}\,b^3\delta + r\,(a+br)^{\delta-2}\,b^2\delta^2 a e^{(a+br)^\delta} \\
& + r^2\,(a+br)^{\delta-2}\,b^3\delta^2 e^{(a+br)^\delta} - r\,(a+br)^{\delta-2}\,b^2\delta^2 a \\
& - r\,(a+br)^{\delta-2}\,b^2\delta a e^{(a+br)^\delta} - r^2\,(a+br)^{\delta-2}\,b^3\delta e^{(a+br)^\delta} \\
& \left. + r\,(a+br)^{\delta-2}\,b^2\delta a - \delta^2 b^2 e^{(a+br)^\delta}\,(a+br)^{2\delta-1}\,r\right)
\end{aligned}$$

$$
\begin{aligned}
&+2\,\ln\left(1-e^{-(a+br)^{\delta}}\right) r\,(a+br)^{\delta-1}\,\delta bae^{(a+br)^{\delta}}\\
&+2\,\ln\left(1-e^{-(a+br)^{\delta}}\right) r^2\,(a+br)^{\delta-1}\,\delta b^2e^{(a+br)^{\delta}}\\
&-2\,\ln\left(1-e^{-(a+br)^{\delta}}\right) r^2\,(a+br)^{\delta-1}\,\delta b^2\\
&-2\,\ln\left(1-e^{-(a+br)^{\delta}}\right) r\,(a+br)^{\delta-1}\,\delta ba\\
&+r^2\,(a+br)^{2\,\delta-2}\,\delta^2b^2a+2\,(a+br)^{\delta-1}\,\delta b^2e^{(a+br)^{\delta}}r\\
&+\left(\ln\left(1-e^{-(a+br)^{\delta}}\right)\right)^2 a+\left(\ln\left(1-e^{-(a+br)^{\delta}}\right)\right)^2 e^{2\,(a+br)^{\delta}}br\\
&+\left(\ln\left(1-e^{-(a+br)^{\delta}}\right)\right)^2 e^{2\,(a+br)^{\delta}}a\\
&+\ln\left(1-e^{-(a+br)^{\delta}}\right)(a+br)^{\delta-1}\,\delta be^{2\,(a+br)^{\delta}}a\Big)
\end{aligned}
$$

References

1. Alvisi, L., Marzullo, K.: Message logging: pessimistic, optimistic, and causal. In: Proceeding of the 15th International Conference on Distributed Computing Systems (ICDCS'95), pp. 229–236 (1995)
2. Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of AFIPS 1967 Spring Joint Computer Conference, vol. 30, pp. 483–485, (1967)
3. Baker, M., Carpenter, B., Shafi, A.: MPJ express: towards thread safe java HPC. In: CLUSTER. IEEE (2006)
4. Batchu, R., Dandass, Y.S., Skjellum, A., Beddhu, M.: MPI/FT: a model-based approach to low-overhead fault tolerant message-passing middleware. Clust. Comput. **7**(4), 303–315 (2004)
5. Bornemann, M., van Nieuwpoort, R.V., Kielmann, T.: MPJ/Ibis: a flexible and efficient message passing platform for java. In: Euro PVM/MPI 2005 (2005)
6. Bouteiller, A., Cappello, F., Hérault, T., Krawezik, G., Lemarinier, P., Magniette, F.: Mpich-v2: a fault tolerant mpi for volatile nodes based on pessimistic sender based message logging. In: Proceedings of the ACM/IEEE SC2003 Conference on High Performance Networking and Computing, p. 25. ACM (2003)
7. Cappello, F., Djilali, S., Fedak, G., Hérault, T., Magniette, F., Néri, V., Lodygensky, O.: Computing on large-scale distributed systems: Xtremweb architecture, programming models, security, tests and convergence with grid. Future Generation Comp. Syst. **21**(3), 417–437 (2005)
8. Carpenter, B., Getov, V., Judd, G., Skjellum, A., Fox, G.: Mpj: Mpi-like message passing for java. Concurr. Pract. Exp. **12**(11), 1019–1038 (2000)
9. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM **43**(2), 225–267 (1996)
10. Char, B.W., Geddes, K.O., Gonnet, G.H., Monagan, M.B., Watt, S.M.: MAPLE reference manual. University of Waterloo, Waterloo Maple Software, Waterloo (1989)
11. Cirne, W., Brasileiro, F.V., Andrade, N., Costa, L., Andrade, A., Novaes, R., Mowbray, M.: Labs of the world, unite!!!. J. Grid Comput. **4**(3), 225–246 (2006)
12. Défago, X., Schiper, A., Urbán, P.: Total order broadcast and multicast algorithms: taxonomy and survey. ACM Comput. Surv. **36**(4), 372–421 (2004)
13. Felber, P., Defago, X., Guerraoui, R., Oser, P.: Failure detectors as first class objects. In: Proceeding of the 9th IEEE Intl. Symposium on Distributed Objects and Applications (DOA'99), pp. 132–141, (1999)
14. Genaud, S., Rattanapoka, C.: P2P-MPI: a peer-to-peer framework for robust execution of message passing parallel programs on grids. J. Grid Comput. **5**(1), 27–42 (2007)

15. Nurmi, D., Brevik, J., Wolski, R.: Modeling machine availability in enterprise and wide-area distributed computing environments. In: Cunha, J.C., Medeiros, P.D. (eds.) Euro-Par, volume 3648 of Lecture Notes in Computer Science, pp. 432–441. Springer, Berlin (2005)
16. Ranganathan, S., George, A.D., Todd, R.W., Chidester, M.C.: Gossip-style failure detection and distributed consensus for scalable heterogeneous clusters. Clust. Comput. **4**(3), 197–209 (2001)
17. Renesse, R.V., Minsky, Y., Hayden, M.: A gossip-style failure detection service. In: IFIP International Conference on Distributed Systems Platforms and Open Distributed Middleware, pp. 55–70, England, (1998)
18. Sankaran, S., Squyres, J.M., Barrett, B., Lumsdaine, A., Duell, J., Hargrove, P., Roman, E.: The LAM/MPI checkpoint/restart framework: system-initiated checkpointing. Int. J. High Perform. Comput. Appl. **19**(4), 479–493 (2005)
19. Schneider, F.B.: Replication management using the state machine approach, Chapter 7. pp. 169–195. ACM Press, New York (1993)
20. Shudo, K., Tanaka, Y., Sekiguchi, S.: P3: P2P-based middleware enabling transfer and aggregation of computational resource. In: 5th International Workshop on Global and Peer-to-Peer Computing. IEEE, (2005)
21. Snir, M., Otto, S.W., Walker, D.W., Dongarra, J., Huss-Lederman, S.: MPI: the complete reference. MIT Press, Cambridge (1995)
22. Stellner, G.: CoCheck: checkpointing and process migration for MPI. In: Proceedings of the 10th International Parallel Processing Symposium (IPPS'96), pp. 526–531 (1996)
23. van Nieuwpoort, R., Maassen, J., Wrzesinska, G., Hofman, R.F.H., Jacobs, C.J.H., Kielmann, T., Bal, H.E.: Ibis: a flexible and efficient java-based grid programming environment. Concurr. Pract. Exp. **17**(7-8), 1079–1107 (2005)
24. Walters, J.P., Chaudhary, V.: A scalable asynchronous replication-based strategy for fault tolerant MPI applications. In: Aluru, S., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) HiPC, volume 4873 of Lecture Notes in Computer Science, pp. 257–268. Springer, Berlin (2007)