Fault management in P2P-MPI

Stéphane Genaud and Choopan Rattanapoka

ICPS-LSIIT - UMR 7005
Université Louis Pasteur, Strasbourg
{genaud,rattanapoka}@icps.u-strasbg.fr

Abstract. We present in this paper the recent developments done in P2P-MPI, a grid middleware, concerning the fault management, which covers fault-tolerance for applications and fault detection. P2P-MPI provides a transparent fault tolerance facility based on replication of computations. Applications are monitored by a distributed set of external modules called failure detectors. The contribution of this paper is the analysis of the advantages and drawbacks of such detectors for a real implementation, and its integration in P2P-MPI. We pay especially attention to the reliability of the failure detection service and to the failure detection speed. We propose a variant of the binary round-robin protocol, which is more reliable than the application execution in any case. Experiments on applications 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 provide 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 accept to share. Numerous projects have recently emerged in that category [11, 5, 2] 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 requests other CPU to take part to a computation. The proposed programming model is close to MPI. We give a brief overview of the system in Section 2 and a longer presentation can be found in [7]. 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 failure detection and fault tolerance for applications. We describe in the paper several pitfalls arising when targeting such environments and what solutions have been put forward in P2P-MPI. The main issues to be addressed are (i) *scalability* since the fault detection system should work up to hundreds of processors, which implies to keep the number of messages exchanged small while having the time needed to detect a fault acceptable, and (ii) *accuracy* means the failure detection should detect all failures and failures detected should be real failures (no false positive).

This paper is organized as follows. Section 2 is a short overview of P2P-MPI which outline the principle of *robustness* of an application execution, through replication of its processes. Section 3 gives an expression of fault-tolerance as the failure probability of the application depending on the replication degree and on the failure events rate. To be effective, the failure detection must be far more reliable than the application execution. We first review in Section 4 the existing techniques to design a reliable fault detection service (FD hereafter). Then, Section 5 examines strengths and weaknesses of candidate solutions considering P2P-MPI requirements. We underline the trade off between reliability and detection speed and we propose a variant of an existing protocol to improve reliability. P2P-MPI implementation integrates the two best protocols, and we report in we report in Section 6 experimental results concerning detection speed for 256 processes.

2 P2P-MPI overview

P2P-MPI overall objective is to provide a *grid* programming environment for parallel applications. P2P-MPI has two facets: it is a middleware and 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. The other facet is the parallel programming API it provides to programmers.

API. Most of the other comparable projects cited in introduction (apart from P3 [11]) 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 this model lies in its suitability to distributed computing environments but lacks expressivity 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.

Contained in the P2P-MPI distribution is a communication library which exposes an MPI-like API. Actually, our implementation of the MPI specification is in Java and we follow the MPJ recommendation [3]. Though Java is used for the sake of portability of codes, the primitives are quite close to the original C/C++/fortran specification [8].

Middleware. A user can simply make its computer join a P2P-MPI grid (it becomes a peer of a peer group) by typing `mpiboot` which runs a local *gatekeeper* process. The gatekeeper can play two roles: (i) it advertises the local computer as available to the peer group, and decides to accept or decline job requests from other peers as they arrive, and (ii) when the user issues a job request, it has the charge of finding the requested number of peers and to organize the job launch.

Launching a MPI job requires to assign an identifier to each task (implemented by a process) and then synchronize all processes at the MPI_Init barrier. By comparison, scheduling jobs made of independent tasks gives more flexibility since no coordination is needed and a task can be assigned to a resource as soon as the resource becomes available.

When a user (the submitter) issues a job request involving several processes, its local gatekeeper initiates a *discovery* to find the requested number of resources during a limited period of time. P2P-MPI uses the JXTA library [1] to handle all usual peer-to-peer operations such as discovery. Resources can be found because they advertised their presence together with their technical characteristics when they joined the peer group. Once enough resources have been selected, the gatekeeper first checks that advertised hosts are still available (by pinging them) and builds a table listing the numbers assigned to each participant process (called the *communicator* in MPI). Then, the gatekeeper instructs a specific service to send the program to execute along with the input data or URL to fetch data from, to each selected host. Each selected host acknowledges the transfer and starts running the received program. (If some hosts fail before sending the acknowledgment, a timeout expires on the submitter side and the job is canceled). The program starts by entering the MPI_Init barrier, waiting for the communicator. As soon as a process has received the communicator it continues executing its application process.

Before dwelling into details of the application startup process and the way it is monitored by the fault-detection service (described in section 5), let us motivate the need for a failure detector by introducing the capability of P2P-MPI to handle application execution robustly.

Robustness. Contrarily to parallel computers, MPI applications in our desktop grid context must face frequent failures. A major feature of P2P-MPI is its ability to manage replicated processes to increase the application robustness. In its run request, the user can simply specify a *replication degree* r which means that each MPI process will have r copies running simultaneously on different processors. In case of failures, the application can continue as long as at least one copy of each process survives. The communication library transparently handles all

extra-communications needed so that the source code of the application does not need any modification.

3 Replication and Failure Probability

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

Assume failures are independent events, occurring equiprobably at each host: we note f the probability that a host fails during a chosen time unit. Thus, the probability that a p process MPI application without replication crashes is

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

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 MPI process has all its r copies failed. The probability that all of the r copies of an MPI process fail is f^r. Thus, like in the expression above, the probability that a p process MPI application with replication degree r crashes is

$$P_{app(p,r)} = 1 - (1 - f^r)^p$$

Figure 1 shows the failure probability curve depending on the replication degree chosen ($r = 1$ means no replication) where f has been arbitrary set to 5%. Remark that doubling the replication degree increases far more than twice

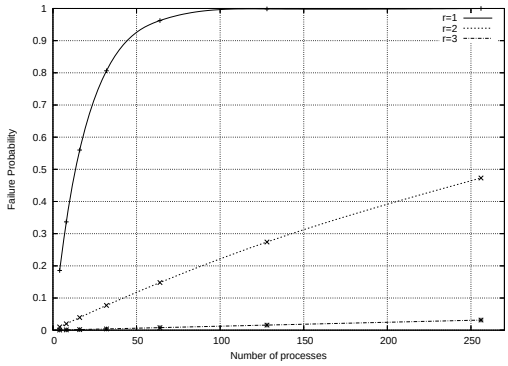

Fig. 1. Failure probability depending on replication degree r (f=0.05).

the robustness. For example, a 128 processes MPI application with a replication degree of only 2 reduces the failure probability from 99% to 27%.

But, 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 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 with a local service called the *fault-detection service*. 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. 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 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 [4], many protocols for failure detection have been proposed and implemented. Two classic approaches are the *push* and *pull* models discussed in [6], 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 [10]. It is a distributed algorithm whose informative messages are evenly dispatched amongst 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, with 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 repeatedly chooses some other modules and sends them a gossip message consisting in its table with its heartbeat incremented. 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'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 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 $T_{consensus}$, 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 $T^{min}_{cleanup}$, which is the time required for information to reach all other hosts, but this can serve as $T_{cleanup}$ only in synchronous systems. 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 [9]. 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 [10], 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.

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

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 Figure 2.

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

Fig. 2. Communication pattern in the round-robin protocol ($n = 6$).

This protocol guarantees that all nodes will receive a given node's updated heartbeat within a bounded time. The information about a state's node 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:

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

For instance in Figure 2, three rounds are required to inform the six nodes of the initial state of node 0 (boxed). We have underlined the nodes when they receive the information.

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:

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

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

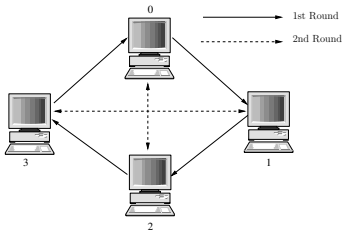

Fig. 3. Communication pattern in the binary round-robin protocol ($n = 4$).

From our experience (also observed in experiments of Section 6), in a 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 $T^{min}_{cleanup}$, i.e. $2 \times \lceil log_2(n) \rceil \times T_{gossip}$. This allows not to consider a fault, the frequent situation where 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 accelerate heartbeat status dissemination at the cost of an increased risk of false detections. Figure 3 shows a 4 nodes system. From equation 2, 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 $T_{cleanup}$ units of time, 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.

5 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 it 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 [9]: 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 lighter procedure, called *ping procedure* in which a node suspecting another node to have failed, directly ping 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 studying the behavior of P2P-MPI FD. We now describe the requirements we have set for the middleware, and which algorithms have been implemented to fulfill these requirements.

5.1 Assumptions and Requirements

In our context, we call a (non-byzantine) *fault* the lack of response during a given delay from a process enrolled for an application execution. A fault can have three origins: (i) the process itself crashes (e.g. the program aborts on a DivideByZero error), (ii) the host executing the process crashes (e.g. the computer is shut off), or (iii) the fault-detection monitoring the process crashes and hence no more notifications of aliveness are reported to other processes.

P2P-MPI is intended for grids and should be able to scale up to hundreds of nodes. Hence, 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 results in false failure detections.

We make several assumptions that we consider realistic accordingly to the above requirements and given current real systems. First, we assume an asynchronous system, with no global clock but we assume the local clock drifts remain

constant. We also assume non-lossy channels: our implementation uses TCP to transport fault detection service traffic because TCP insures message delivery. TCP also has the advantage of being less often blocked than UDP between administrative domains. We also require a few available ports (3 for services plus 1 for each application) for TCP communications, i.e. not blocked by firewalls for any participating peer. Indeed, for sake of performances, we do not have relay mechanisms. During the startup phase, if we detect that the communication could not be establish back and forth between the submitter and all other peers, the application's launch stops. Last, we assume that the time required to transmit a message between any two hosts is generally less than T_{gossip}. Yet, we tolerate unusually long transmission times (due to network hangup for instance) thanks to a parameter T_{max_hangup} set by the user (actually $T_{cleanup}$ is increased by T_{max_hangup} in the implementation).

5.2 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.

As stated above, the BRR protocol is optimal with respect to bandwidth usage and fault detection delay. The low bandwidth usage is due to 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. Let f be the failure probability of each individual node in a time unit T ($T < T_{cleanup}$), and let $P(i)$ the probability that i nodes simultaneously fail during T. In the case 2 nodes fail, if both of them are source nodes then there will be a node that can not 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 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 failure has probability $P_{brr(4)} = P(4) + P(3) + P(2) = f^4 + \binom{4}{3}f^3(1-f) + 4f^2(1-f)^2$.

In this case, using the numerical values of section 3 (i.e. f=0.05), the comparison between the failure probability of the application (p=2, r=2) and the failure probability of the BRR for n=4, leads to $P_{app(2,2)} = 0.005$ and $P_{brr(4)} = 0.0095$ which 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 robustness by not eliminating all redundancy in the gossip protocol.

5.3 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).

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. Also, the application must occasionally make a decision against the FD prediction about a failure to detect firewalls.

Double Binary Round-Robin (DBRR) We introduce the double binary round-robin protocol which detects failures in a delay asymptotically equal to BRR ($O(log_2(n))$ and acceptably fast in practice, while re-inforcing robustness of BRR. The idea is simply to avoid to have 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 back by applying BRR in a counterclock-wise direction. The destination node for each gossip message is determined by the following relation:

$$d = \begin{cases} (s + 2^{r-1}) \mod n & \text{if } 1 \leq r \leq \lceil log_2(n) \rceil \\ (s - 2^{r-\lceil log_2(n) \rceil - 1}) \mod n & \text{if } \lceil log_2(n) \rceil < r \leq 2\lceil log_2(n) \rceil \end{cases} \quad (3)$$

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.

To compare BRR and DBRR reliability, we can count following the principles of Section 5.2 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 Figure 4, we repeat the simulation for 5.8×10^9 trials with f=0.05. Notice that in the DBRR protocol, we could not not find any FD failure when the number of nodes is more than 16, which means the number of our trials is not sufficient to estimate the DBRR failure probability for such n.

Automatic Adjustment of Initial Heartbeat 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

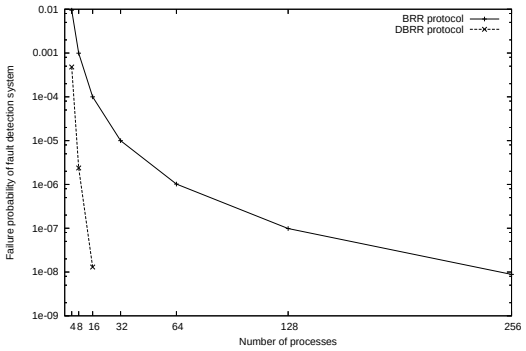

Fig. 4. Failure probabilities of the FD system using BRR and DBRR ($f = 0.05$).

submitter construct a table numbering available resources called the communicator[1], which is sent in turn to participating peers. The remote peers acknowledge this numbering 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 (i) not declared nodes that have not yet started their FD as failed, and (ii) 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, together with the communicator, the time spent sending information to previous nodes. Figure 5 illustrates the situation. We note ts_i, $1 \leq i < n$ the date when the submitter sends the communicator to peer 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. The submitter sends $\Delta t_i = ts_i - ts_1$ to any peer i ($1 \leq i < n$) which can then computes its initial heartbeat h_i as:

$$h_i = \lceil (T_i - tr_i + \Delta t_i)/T_{gossip} \rceil, \quad 1 \leq i < n \tag{4}$$

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

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,

[1] The submitter always has number 0.

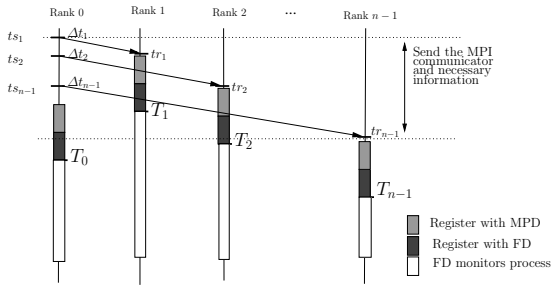

Fig. 5. Application startup

then the rest of that tree will not get any information about the communicator and the execution could not continue.

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 (eg. Send, Bcast) the sender continuously tried to send a message to the destination (ignoring socket timeouts) until it either succeeded or received a notification that the destination node is down from its FD. This allows 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 can not terminate. Our new send implementation simply installs a timeout to tackle this problem, which we set to $2 \times T_{cleanup}$. Reaching this timeout on a send stops the local application process, and soon the rest of the nodes will detect the process death.

6 Experiments

The objective of the experiments is to evaluate the failure detection speed with both BRR and DBRR monitoring a P2P-MPI application running on a real grid testbed. We use the Grid'5000 platform, a federation of dedicated computers hosted across nine campus sites in France, and organized in a virtual private network over Renater, the national education and research network. Each site has currently about 100 to 700 processors arranged in one to several clusters at each site. In our experiment, we distribute the processes of our parallel test application across three sites (Nancy, Rennes and Nice).

The experiment consists in running a parallel application without replication and after 20 seconds, we kill all processes on a random node. We then log at

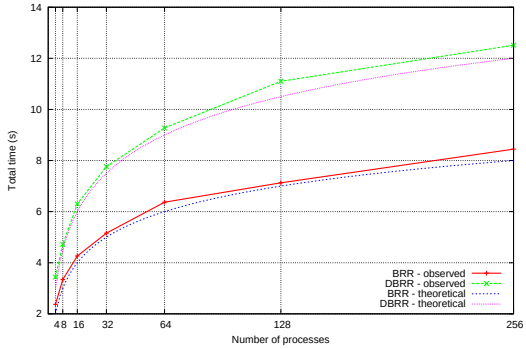

Fig. 6. Time to detect a fault for BRR and DBRR

what time each node is notified of the failure and compute the time interval between failure and detection. Figure 6 plots the average of these intervals on all nodes and for both protocols, with T_{gossip} set to 0.5 second. 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 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 Equation 4. 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, 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 hearbeat 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. Further tests should experiment smaller values of T_{gossip} for a quicker detection time. We also plan to test the system at the scale of a thousand processes.

7 Conclusion

We have described in this paper the fault-detection service underlying P2P-MPI. The first part is an overview of the principles of P2P-MPI among which is repli-

cation, used as a means to increase robustness of applications executions, and external monitoring of application execution by a specific fault-detection module. In the second part, we first describe the background of our work, based on recent advances in the research field of fault detectors. We compare the main protocols recently proposed 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, and thus improves the fault-tolerance of the failure detection service, while the detection time remains low. Last, we present the experiments conducted on Grid'5000. 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.

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)

References

[1] JXTA. `http://www.jxta.org`.
[2] N. Andrade, W. Cirne, F. Brasileiro, and P. Roisenberg. Our-grid: An approach to easily assemble grids with equitable resource sharing. In *9thWorkshop on Job Scheduling Strategies for Parallel Processing*, June 2003.
[3] B. Carpenter, V. Getov, G. Judd, T. Skjellum, and G. Fox. Mpj: Mpi-like message passing for java. *Concurrency: Practice and Experience*, 12(11), Sept. 2000.
[4] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.
[5] G. Fedak, C. Germain, V. Néri, and F. Cappello. XtremWeb: A generic global computing system. In *CCGRID*, pages 582–587. IEEE Computer Society, 2001.
[6] P. Felber, X. Defago, R. Guerraoui, and P. Oser. Failure detectors as first class objects. In *Proceeding of the 9th IEEE Intl. Symposium on Distributed Objects and Applications (DOA'99)*, pages 132–141, Sept. 1999.
[7] S. Genaud and C. Rattanapoka. A peer-to-peer framework for robust execution of message passing parallel programs. In *EuroPVM/MPI 2005*, volume 3666 of *LNCS*, pages 276–284. Springer-Verlag, September 2005.
[8] MPI Forum. MPI: A message passing interface standard. Technical report, University of Tennessee, Knoxville, TN, USA, June 1995.
[9] S. Ranganathan, A. D. George, R. W. Todd, and M. C. Chidester. Gossip-style failure detection and distributed consensus for scalable heterogeneous clusters. *Cluster Computing*, 4(3):197–209, 2001.
[10] R. V. Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *IFIP International Conference on Distributed Systems Platforms and Open Distributed Middleware*, pages 55–70, England, 1998.
[11] K. Shudo, Y. Tanaka, and S. Sekiguchi. P3: P2P-based middleware enabling transfer and aggregation of computational resource. In *5th Intl. Workshop on Global and Peer-to-Peer Computing*. IEEE, May 2005.