P2P-MPI : A fault-tolerant Message Passing Interface Implementation for Grids

Choopan Rattanapoka

UNIVERSITÉ LOUIS PASTEUR STRASBOURG

Laboratoire des Sciences
de l'Image, de l'Informatique et
de la Télédétection

UMR CNRS-ULP 7005

Équipe Image et
Calcul Parallèle Scientifique

Thèse présentée pour obtenir le grade de
Docteur de l'Université Louis Pasteur
Strasbourg I

Discipline: Informatique

par Choopan Rattanapoka

Titre

P2P-MPI :A Fault-tolerant Message Passing Interface Implementation for Grids

Soutenue publiquement le 22 avril 2008

Membres du jury

Rapporteurs externes :	M. Thilo Kielmann, Associate Professor Vrije Universiteit, Amsterdam M. Franck Cappello, Directeur de Recherches INRIA, Orsay
Rapporteur interne :	M. Jean-Jacques Pansiot, Professeur Université Louis Pasteur de Strasbourg
Examinateur :	Mme. Françoise Baude, Maître de Conférences Université de Nice-Sophia Antipolis
Directeurs :	Mme. Catherine Mongenet, Professeur Université Louis Pasteur de Strasbourg M. Stéphane Genaud, Chargé de Recherches INRIA, Nancy

Ce document a été composé avec LaTeX.

Contents

List of Tables

List of Figures

Résumé en français

1 Introduction

Les grilles de calcul offrent de nouvelles perspectives pour résoudre des problèmes nécessitant des calculs massifs en utilisant de nombreux ordinateurs à large échelle géographique. Ceci implique de partager des ressources hétérogènes (du point de vue du matériel ou du logiciel) qui sont administrées par des personnes ou des organisations différentes. L'un des freins majeurs à l'utilisation des grilles aujourd'hui, est la complexité d'y déployer certains types de programmes. Si certains programmes séquentiels ou distribués correspondent à des problèmes qui se prêtent bien à ces environnements, de nombreux autres programmes parallèles posent de sérieuses difficultés. L'éventail des difficultés est large. Elles vont des difficultés pratiques liées aux lacunes des intergiciels actuels (par exemple l'absence de tolérance aux pannes), aux difficultés théoriques que posent ces plateformes très hétérogènes impliquant des acteurs divers (qui requièrent par exemple un ordonnancement multi-critères).

Lorsqu'on parle de grilles de calcul, on distingue souvent les grilles composées de ressources fiables (super-ordinateurs) de celles composées par des ressources volatiles (par exemple des ordinateurs personnels dont la configuration et l'état du système changent fréquemment). Cette dernière catégorie est souvent appelée *desktop grid*.

De manière très majoritaire, les applications pour le calcul parallèle sont développées en utilisant le standard MPI (Message Passing Interface)[1]. Ce standard définit des primitives permettant la programmation parallèle par passage des messages. Cependant, MPI a clairement été conçu pour des environnements d'exécution stables. Le modèle d'exécution sous-jacent est en effet très fragile vis-à-vis des pannes: il suffit qu'un seul des processus de l'application tombe en panne pendant l'exécution pour que l'application entière ne puisse plus continuer.

Cette thèse est consacrée au développement d'un intergiciel qui intègre plusieurs contributions. Le but ultime est d'offrir un environnement d'exécution accessible de manière simple aux utilisateurs, et aux programmeurs un moyen de développer des applications parallèles du type passage de message (MPI) dans un environnement de grille. L'intergiciel s'appelle P2P-MPI.

2 P2P-MPI

P2P-MPI est basé sur un modèle pair-à-pair. Chaque ordinateur démarrant le logiciel devient un pair, au même titre que les autres ordinateurs. Démarrer le logiciel permet d'exprimer des requêtes de calcul utilisant les ordinateurs distants, mais implique également de partager son propre ordinateur. P2P-MPI est développé uniquement en Java, et il peut donc être exécuté sur presque tous les système d'exploitation sans re-compiler les codes sources. Enfin, P2P-MPI fournit un sous-ensemble de l'API de MPI, permettant le développment de programmes parallèles.

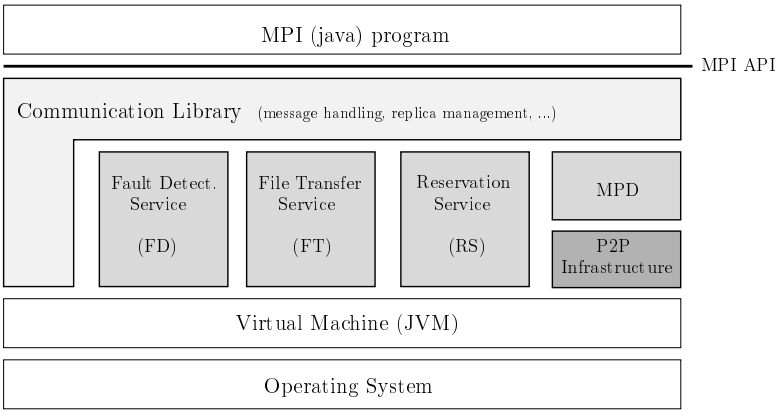


Figure 1: P2P-MPI structure.

La Figure 1 présente l'infrastructure de P2P-MPI (en gris). P2P-MPI est composé de trois éléments importants:

L'infrastructure pair-à-pair De nombreux travaux ont porté sur l'étude et la conception d'infrastructures pair-à-pair. JXTA [2], projet open-source soutenu par Sun MicroSystems, est un des projets les plus connus car il est le seul à avoir développé des spécifications synthétisant ce qu'on doit trouver dans un système pair-à-pair. Les principales implémentations de cette spécification sont en C et en Java, et sont disponibles publiquement.

Ces implémentations fournissent, en plus de l'API, des services comme la découverte automatique des pairs, les canaux de communication abstraits (JXTA pipe), etc. P2P-MPI, jusqu'à récemment, a utilisé JXTA pour son infrastructure pair-à-pair. Cependant, à l'usage, JXTA s'est montré inadapté vis-à-vis de nos besoins, principalement en raison du fait que son service de découverte n'essaye pas de découvrir tous les pairs. Nous avons donc, dans les versions récentes de P2P-MPI, développé notre propre infrastructure pair-à-pair, plus simple, mais intégrant de nouvelles fonctionnalités comme la notion de distance réseau. Cette évolution

nous a permis de tester nos stratégies standards d'allocation des pairs à travers des expériences de déploiement réel de l'ordre de 1000 processus.

L'intergiciel L'intergiciel représente une grande partie du développement de P2P-MPI. Il implémente un certain nombre de services qui sont d'une grande importance pour faciliter l'accès et l'exploitation d'un réseau de machines disponibles. Ces services sont:

- **Le service de détection des pannes (FD)** est utilisé pour la détection des pannes pendant l'exécution des applications.
- **Le service de transfert des fichiers (FT)** est utilisé pour transférer le ou les fichiers exécutables et de données aux machines distantes.
- **Le service de réservation (RS)** est utilisé pour réserver un ensemble de machines apte à satisfaire une requête d'exécution exprimée par un utilsiateur.
- **Le processus MPD** représente la ressource locale comme un pair dans le réseau pair-à-pair. Il filtre les requêtes extérieures demandant l'utilisation de la machine locale, et symétriquement, coordonne les actions nécessaires pour transmettre une requête d'exécution initiée localement.

La bibliothèque de communication P2P-MPI suit la spécification MPJ [3], c'est-à-dire une adaptation du standard MPI à Java. L'interface de programmation ressemble donc beaucoup à MPI (défini pour C, C++ et Fortran). D'autres projets ont également proposé des implémentations de MPJ. Le mécanisme de tolérance aux pannes proposé par P2P-MPI, qui n'impose aucune modification au code source, n'empêche donc pas la conformité avec MPJ.

3 L'intergiciel

Dans cette section, on présente la fonction principale de l'intergicielle qui est de prendre en charge l'exécution d'un application parallèle. Nous décrivons ci-dessous comment les processus (FD, FT, RS, et MPD) interagissent lorsque un utilisateur soumet un job. Le mécanisme vise à réserver un nombre adéquat de ressources disponibles, à transférer les fichiers nécessaires à l'exécution, et à démarrer simultanément l'ensemble des processus formant l'application. A l'issue de la procédure, un numéro unique est attribué à chaque processus démarré, formant ainsi un *communicateur* MPI. Les étapes sont illustrées dans le détail par la Figure 2.

(1) **Booting up:** L'utilisateur joint la plate-forme P2P-MPI en tapant la commande `mpiboot` qui démarre MPD, FT, FD, et RS.

(2) **Job submission:** le job est lancé en tapant une commande du type `p2pmpirun -n` n `-r` r `-a` *alloc* `prog`. Les paramètres obligatoire sont n le nombre de processus pour éxecuter l'application `prog`. Les autres paramètres sont optionels: r est le taux de replication et *alloc* est la strategie d'allocation des ressources.

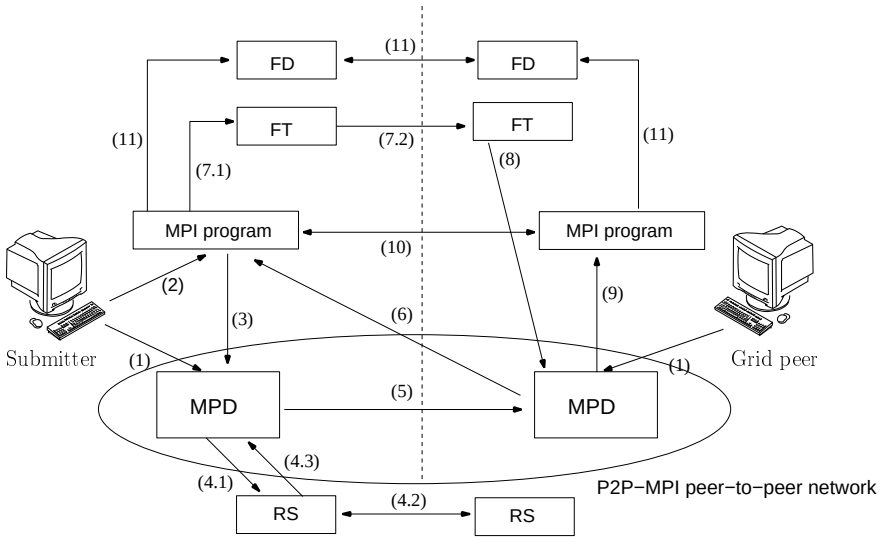


Figure 2: Les étapes de la soumission d'un job.

(3) **Requesting Peers:** l'application contacte son MPD en lui demandant de découvrir un nombre adéquat de ressources disponibles.

(4) **Discovery and Reservation:** le MPD demande au RS local de procéder aux réservations des ressources distantes. Le RS local négocie avec les RS distants pour réserver les ressources puis retourne la liste des ressources réservées au MPD local.

(5) **Registration:** la réservation terminée, le MPD local contacte les MPD des machines réservées en transmettant le nom d'application, le rang dans le communicateur du processus MPI, et l'IP et le port du processus à l'origine de la soumission.

(6) **Hand-shake:** les machines distantes communiquent les ports de leurs processus FT et FD au processus à l'origine de la soumission pour établir une communication directe.

(7) **File transfer:** le code exécutable et les fichiers de données en entrée sont envoyés via le service FT.

(8) **Execution Notification:** lorsque le transfert est fait, les services FT des machines distantes notifient leurs MPD d'exécuter le code exécutable qui vient d'être transféré.

(9) **Remote executable launch:** le MPD exécute l'application.

(10) **Execution preamble:** les processus qui viennent de se lancer contactent le processus à l'origine de la soumission pour constuire le communinateur MPI.

(11) **Fault detection:** les processus MPI s'enregistrent auprès du service FD pour pouvoir détecter les pannes pendant l'exécution.

Toutes les étapes de cette procédure permettant d'établir un environnement d'exécution sont bien sûr totalement transparentes à l'utilisateur.

Strategie d'allocation des ressources

Notre objectif est de proposer des stratégies d'allocation intuitives pour l'utilisateur. Actuellement, deux stratégies sont proposées pour illustrer ce qui nous semble manipulable facilement par un utilisateur. L'intergiciel propose à l'utilisateur, à travers ces deux stratégies, d'arbitrer entre la répartition des processus sur le plus grand nombre de machines possibles ou au contraire de concentrer les processus quand les machines en offrent la possibilité. Aujourd'hui, il y a beaucoup de CPUs multi-cœurs et l'allocation des processus en utilisant le maximum de cœurs d'une machine peut être un choix judicieux car on accroît la localité des processus. Cependant, si l'application a besoin beaucoup de mémoire, le choix précédent sera pénalisant car plusieurs cœurs dans une machine se partagent la mémoire, diminuant ainsi la quantité disponible par processus et augmentant la contention des accès mémoires. Nous pensons que l'utilisateur connaît son application et qu'il est le plus à même de choisir quelle stratégie est la mieux adaptée. Ces deux stratégies s'appellent *spread* et *concentrate* :

- **Spread** essaie de placer les processus MPI en maximisant le nombre de machines allouées pour maximiser la mémoire totale utilisable par l'application.

- **Concentrate** essaie de placer des processus MPI en maximisant le nombre de cœurs alloués par machine pour respecter la localité et minimiser les coûts des communications entre machines.

Experiences

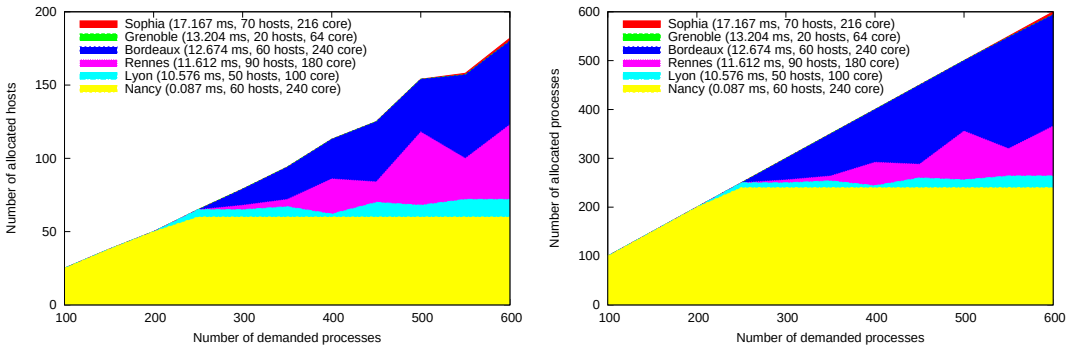
La mise en œuvre effective des stratégies à été testée à grande échelle dans la thèse. Les expériences sont faites sur Grid5000 en utlisant six sites : Nancy, Lyon, Rennes, Bordeaux, Grenoble, and Sophia-Antipolis. La soumission est faite à partir du site de Nancy. Le tableau ci-dessous résume les caractéristiques de la plate-forme de test utilisée.

Type d'environment	Grid5000 – clusters détaillés ci-dessous.				
Site	Cluster name	CPU	#Nodes	#CPUs	#Cores
Nancy	grelon	Intel Xeon 5110	60	120	240
Lyon	capricorn	AMD Opteron 246	50	100	100
Rennes	paravent	AMD Opteron 246	90	180	180
Bordeaux	bordereau	AMD Opteron 2218	60	120	240
Grenoble	idpot	Intel Xeon IA32	8	16	16
Grenoble	idcalc	Intel Itanium 2	12	24	48
Sophia-Antipolis	azur	AMD Opteron 246	32	64	64
Sophia-Antipolis	sol	AMD Opteron 2218	38	76	152
Système d'exploitation	Linux 2.6.18				
Software	jdk1.6.0_04, JXTA-J2SE 2.3, p2pmpi-0.28.0				

Expériences de co-allocation

Dans cette expérience, on lance une application dont chaque processus affiche simplement le nom de machine sur laquelle il s'exécute. On observe où les processus sont placés, en fonction de la stratégie et du nombre de processus demandé.

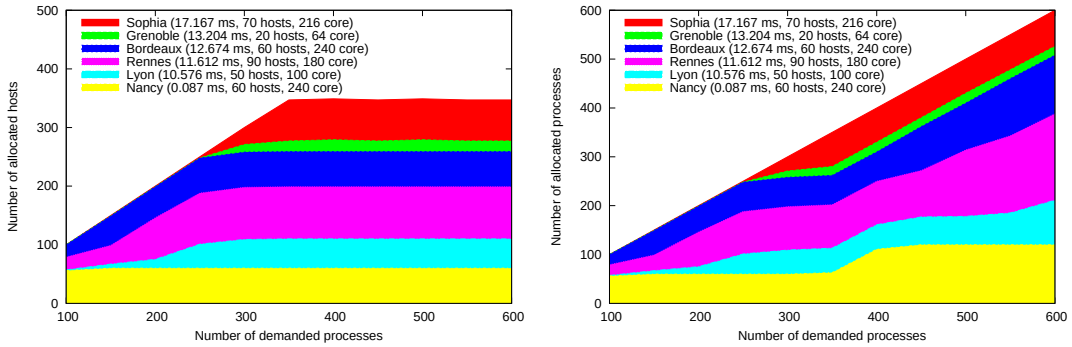
Les figures 3 et 4 montrent le placement des processus en utilisant les stratégies concentrate et spread. La légende en haut à gauche donne le RTT à partir du site de Nancy et le nombre de machines et cores disponible à chaque site. L'application est lancée en demandant de 100 à 600 processus, par pas de 50.



(a) Allocated hosts (b) Allocated cores

Figure 3: Machines et cores alloués avec *concentrate*.

Le comportement de *concentrate* est illustré par la figure 3. Les processus sont placés sur les 60 machines disponibles à Nancy jusqu'à 200 processus. Puis, lorsque la capacité de 240 cores à Nancy est utilisée, des machines à Lyon sont choisies (5 pour -n 250), ce qui est conforme aux attentes étant donné le classement par RTT. Les demandes suivantes à partir de -n 300 prennent des machines à Lyon, Rennes et Bordeaux. Cela s'explique par le fait que les latences entre Nancy et les trois sites sont très proches et que P2P-MPI mesure la latence en utilisant le port ouvert par l'application et non par



(a) Allocated hosts (b) Allocated cores

Figure 4: Machines et cores alloués avec *spread*.

ICMP dont le port peut être bloqué. De fait, le ping applicatif est sensible à la charge de la CPU et du réseau au moment de sa mesure qui peut différer de celle d'ICMP.

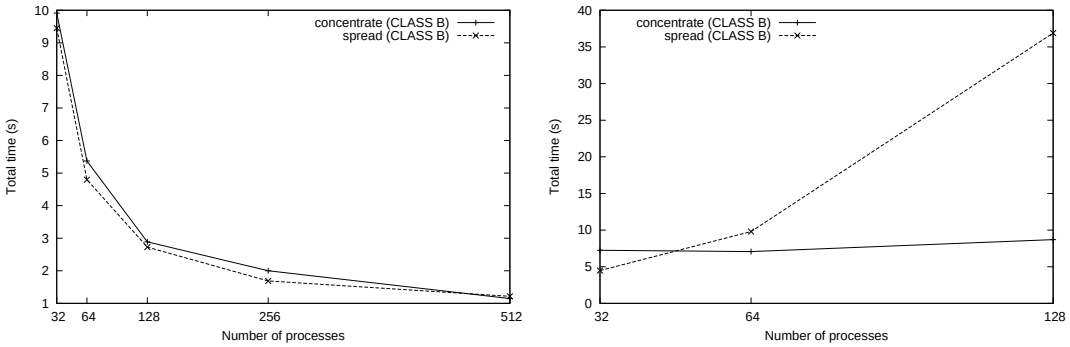
Avec *spread*, illustré par la figure 4, les machines sont choisies sur les quatre sites les plus proche jusqu'à 250 processus. A partir de 300 procesus, la stratégie prends des machines sur tous les sites et place seulement un processus MPI par machine. On peut voir clairement dans la figure 4(b), que le nombre de cores utilisés à Nancy augmente lorsqu'on demande 400 processus. La raison est qu'on dispose de 350 machines au total, et lorque le nombre de processus excède le nombre de machines disponibles, la stratégie place les processus non encore placés dans les machines les plus proches, ici à Nancy.

Performance d'une application

Pour observer l'efficacité de chaque stratégie sur l'application, on a choisi de tester deux applications qui ont des caracteristiques opposées, tirées du NAS benchmarks (NPB3.2): IS (Integer Sorting) et EP (Embarrassingly Parallel). IS est une application qui communique beacoup tandisqu'EP est une application qui fait de nombreux calculs indépendants.

La figure 5(a) présente le temps d'exécution de EP de 32 à 512 processus. EP n'invoque que quatre opérations de communication collective (`MPI.Allreduce` de un double) donc la ratio calcul sur communication est important. Quelque soit la stratégie utilisée pour l'allocation, on obtient des performances très similaires.

La performance de IS est présentée sur la figure 5(b). Avec 32 procesus, *spread* obtient une meilleure performance que *concentrate*. On peut l'expliquer par le fait qu'avec *spread*, les 32 processus restent dans le même clusteur et que le coût des communications est assez faible. D'autre part, il n'y a pas de concurrence d'accès à la mémoire par des processus, comme c'est le cas pour *concentrate*. A partir de 64 processus demandés, l'utilisation de *spread* implique que des processus sont placés hors du clusteur local et les communications entre clusters pénalisent les performances.



(a) Execution time on EP benchmark.

(b) Execution time on IS benchmark.

Figure 5: Temps d'exécution de EP et IS en fonction de la strategie d'allocation.

4 La gestion des pannes

Pour cette partie, nous distinguons deux questions: (1) le comportement de l'application en cas de pannes et (2) la détection des pannes.

Comportement en cas de panne

P2P-MPI propose la réplication des processus comme mécanisme de tolérance aux pannes. Un utilisateur peut demander combien de processus seront repliqués au lancement du programme. La gestion des réplicas par P2P-MPI est totalement transparente pour l'utilisateur. Le code de l'application n'a pas besoin de changer. En cas de panne, l'application MPI peut continuer de s'exécuter tant qu'il existe au moins une copie non-défaillante de chaque processus.

Côté envoyeur, on limite le nombre de messages envoyés en introduisant un *agreement* protocole. Pour chaque processus logique, un processus repliqué est élu comme le maître du groupe pour envoyer les messages. Les autres processus repliqués n'envoient pas de messages mais les gardent en mémoire. La figure 6 illustre le déroulement d'une insruction *send* de P_0 à P_1 où le processus répliqué P_0^0 est le maître.

Détection des pannes

P2P-MPI intègre un service de détection des pannes. Comme P2P-MPI est basé sur un modèle pair-à-pair, nous excluons la possibilité de recourir à un serveur centralisé pour détecter des pannes pendant l'exécution des applications. Les chercheurs de la communauté des systèmes distribués ont proposé des détecteurs de défaillances, basés sur le *gossip protocol* [4]. C'est un protocole permettant aux différentes machines de détecter des pannes de machines distantes sans serveur centralisé. L'idée de base du protocole est que chaque machine augmente régulièrement ses pulsations (heartbeat),

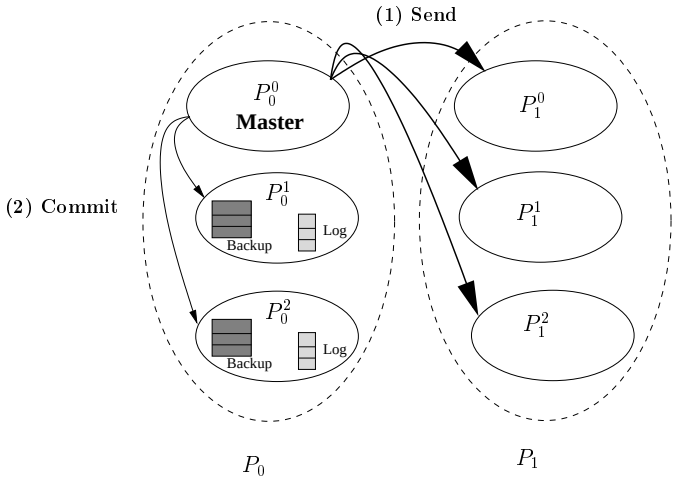


Figure 6: Un message envoyé de processus logique P_0 à P_1.

puis échange régulièrement avec quelques autres l'état global du système, consitué des pulsations de chaque machine. Si les pulsations d'une machine n'ont pas augmenté pendant un certain temps alors cette machine est suspectée d'être en panne.

Le temps de détection d'une panne est difficile à définir, car les échanges d'état globaux sont asynchrones. Notre contribution est l'étude des protocoles de détection de pannes permettant de prédire un temps de détection de panne en fonction du nombre de machines. Les routages que nous retenons en fin de compte sont des routages fixes, i.e le calcul des destinataires des informations de pulsations est déterminé à l'avance. Nous retenons les protocoles Binary round-robin (BRR) et Double binary round-robin (DBRR), qui sont proposés dans P2P-MPI. BRR fournit un temps de détection des pannes plus rapide que DBRR mais le protocoles est moins fiable que DBRR lorsque le nombre de machines est petit.

Concernant la fiabilité des deux protocoles, nous l'établissons par simulation du système. La figure 7 montre une simulation de 5.8×10^9 tirages aléatoires avec à chaque fois une probabilité de $0,05$ qu'un processus FD quelconque tombe en panne. Les courbes donnent les probabilités que les protocoles BRR et DBRR soit défaillants (suite à la défaillance de plusieurs processus FD) à l'issue des tirages aléatoires. On voit que les protocoles deviennent très résistants quand le nombre de machines augmente.

Expérience: temps de détection des pannes

Nous avons validé expérimentalement le temps de détection des pannes, tel que prédit pour les deux protocoles proposés. Dans cette expérience menée sur Grid5000, on lance une application sur trois sites: Nancy, Rennes, et Sophia-Antipolis puis on tue aléatoirement un processus MPI dans une machine et on mesure le temps entre la mort du processuet le temps de détection par le service de détection des pannes. La configuration

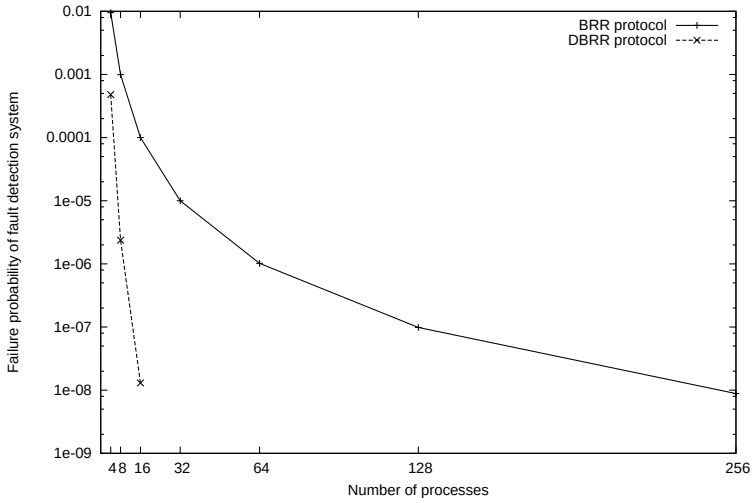


Figure 7: Probabilités de défaillance du FD service en utilisant BRR and DBRR, pour 5.8×10^9 pannes individuelles.

utilisée pour l'experience est la suivante.

Type d'environment	Grid5000 – grillon.nancy, paravent.rennes azur.sophia clusters
Matériel	dual-cores AMD Opteron 2GHz, 2GB RAM
Operating System	Linux 2.6.14
Interconnexion	2 ports GE cards intra-cluster, 10 Gbps/s entre sites.
Logiciel	jdk1.5, p2pmpi-0.20.0

La figure 8 montre le temps moyen mis par les processus pour détecter la panne. Sont aussi repésentés sur la figure les courbes des temps "théoriques" de détection des pannes.

Le temps observé de détection des pannes est très proche de ce qu'on peut prédire en théorie.

5 L'implémentation de MPJ

P2P-MPI suit la spécification MPJ [3]. L'interface de programmation ressemble donc beaucoup à MPI (défini pour C, C++ et Fortran). De plus, le mécanisme de tolérance aux pannes proposé par P2P-MPI, qui n'impose aucune modification du code source, n'altère pas la conformité avec MPJ.

Dans ce travail de thèse, nous avons testé deux approches différentes pour implémenter la couche bas niveau de la bibliothèque de communication. Nous désignons par *device* cette couche bas niveau, qui est la partie de l'implémentation gérant les envois, réceptions, file d'attente des messages, etc. Nos deux approches sont :

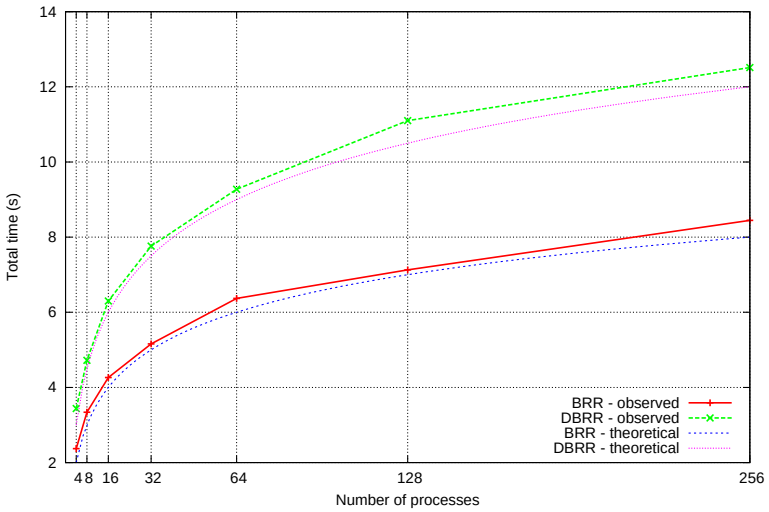


Figure 8: Temps de détection d'une panne en utilisant BRR et DBRR

Single-port device (SP) Le but est de réduire le nombre de ports utilisés par chaque processus MPI. Cette hypothèse est pertinente dans un environnement de grille, lorsque les utilisateurs ne peuvent ouvrir un nombre illimité de ports car la politique adoptée pour le filtrage (utilisation de pare-feux) restreint les ports utilisables. Dans cette approche, chaque processus MPI utilise un seul port de communication, ce qui rend le déploiement nettement plus facile en cas de restriction sur l'ouverture des ports. Pour envoyer un message, il faut ouvrir la connexion, envoyer le message, et puis fermer la connexion.

Multiple-port device (MP) Le device SP est pénalisé lorqu'il y a beaucoup de communications dans l'application, car le temps système nécessaire à l'ouverture et à la fermeture de l'unique connexion devient problématique. L'approche proposée par le device MP consiste à ouvrir plusieurs ports simultanément pour améliorer la performance. Ceci implique qu'il n'y a pas de restriction importante sur le nombre de ports ouverts. Dans ce device, chaque processus MPI a deux liens permanents vers les autres processus. Un lien est utilisé pour écrire les messages et un autre est utilisé pour lire les messages.

Optimisation des opérations collectives

P2P-MPI introduit des optimisations pour l'implémentation des opérations collectives. Actuellement, on utilise des algorithmes connus qui ont de bonnes performances en LAN, mais dont les performaces dans un WAN ou dans un réseau avec des communications à forte et faible latence, ne sont pas forcément optimales. Les opérations codant les communications collectives se trouvent dans la classe `IntraComm` (appendix B.4, page

132). Le tableau 1 montre le détail des méthodes et des algorithmes utilisés.

Méthode	Algorithme
Allgather	Gather puis Bcast
Allgatherv	Gatherv puis Bcast
Allreduce	Butterfly ou Reduce puis Bcast
Alltoall	Asynchronous rotation
Alltoallv	Asynchronous rotation
Barrier	4-ary tree
Bcast	Binomial tree
Gather	Flat tree
Gatherv	Flat tree
Reduce	Binomial tree ou flat tree
Reduce_scatter	Reduce puis Scatterv
Scatter	Flat tree
Scatterv	Flat tree

Table 1: La liste des méthodes dans la classe `IntraComm`.

Expériences

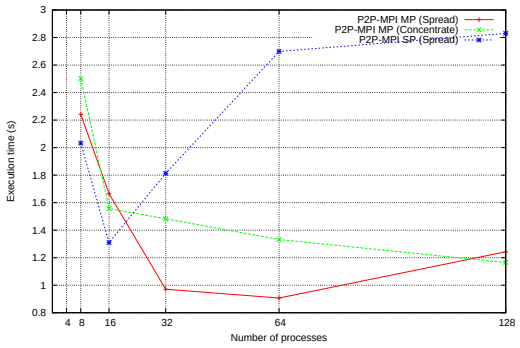
La configuration du système est :

Type d'environment	Grid5000 (grelon.nancy)
Matériel	64 nodes/128 cores Intel Xeon 5110, 2GB RAM
Operating System	Linux 2.6.24-1-amd64
Interconnexion	Gigabit Ethernet.
Java runtime	Java 1.5.0_08.
Benchmark suites	JGF section 2 (CLASS B) and JGF section 3 (CLASS A)
P2P-MPI implementation	P2P-MPI-0.27.1 (SP device) and P2P-MPI-0.28.0 (MP device)

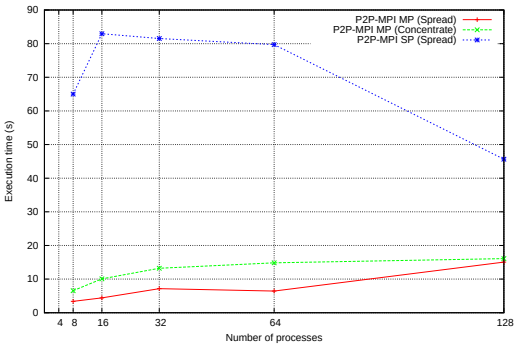
On présente ici les résultats des sections 2 et sections 3 de JGF benchmark. Le résultat de la section 1 de JGF benchmark se trouve dans l'annexe D, page 147. Les figures 9 et 10 montrent les résultat de Section 2 and Section 3, respectivement.

On utilise 64 machines, chaque machine a deux cores. Donc, on a 128 cores de calcul au total. L'expérience compare les deux types de devices (SP et MP).

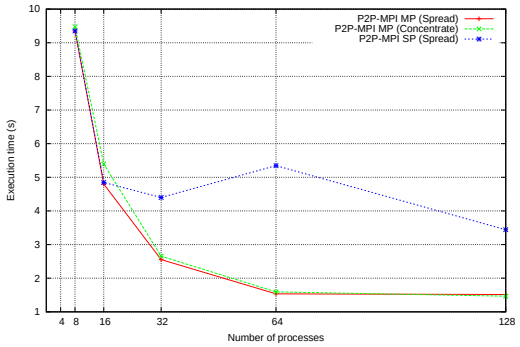
La strategie *spread* obtient de meilleures performances que la strategie *concentrate* dans ces benchmarks. Nous avons observé sur les premiers tests que l'utilisation de plusieurs cores sur une machine pouvait provoquer une baisse de performance, probablement liée à de la contention des accès mémoires. Dans plusieurs tests, nous avons noté que la plus grande localité des processus qu'implique cette stratégie, et donc des temps de communications moindes, ne contre-balancent pas les pénalités dues aux contentions mémoire. On peut conclure que ces benchmarks impliquent un ratio de calcul sur communication important. Concernant le device SP, les performances atteintes sont inférieures comme prévu.



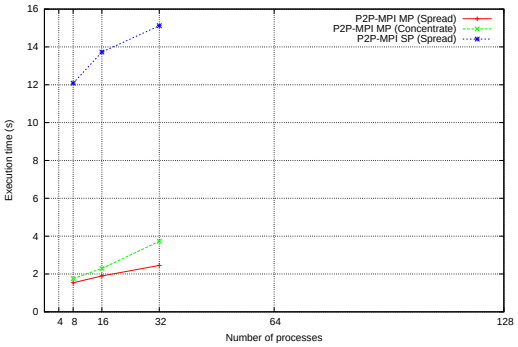
(a) crypt



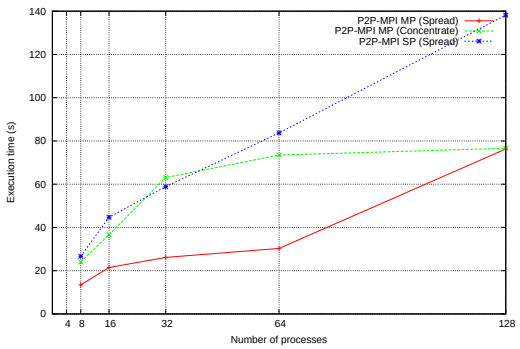
(b) lufact



(c) serie



(d) sor

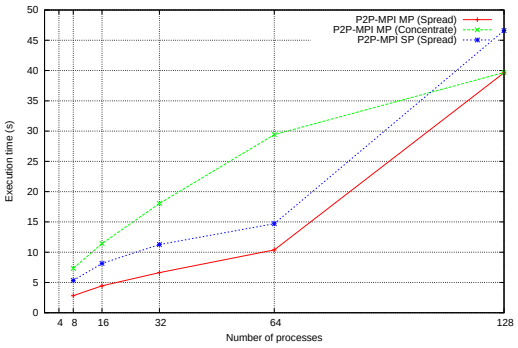


(e) sparseMatMult

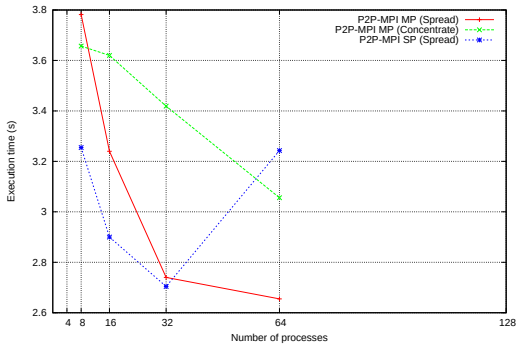
Figure 9: JGF section 2: résultat du benchmark Kernels

6 Conclusion et Perspectives

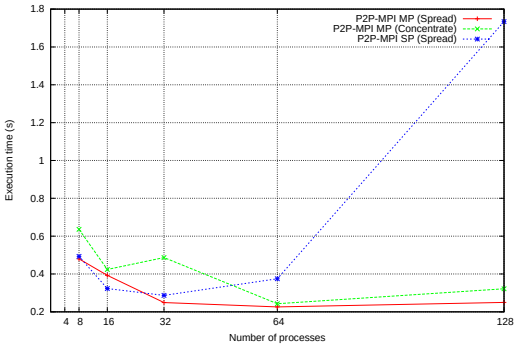
Ce travail de thèse a requis des développements très importants. Ils se concrétisent dans la réalisation de l'intergiciel P2P-MPI, dont nous assurons la maintenance et la distribu-



(a) moldyn



(b) montecarlo



(c) raytracer

Figure 10: JGF section 3: résultat du benchmark Large-scale applications

tion sous une licence publique. L'objectif souhaité est que d'autres équipes de recherche puissent l'utiliser et reproduire nos expériences. En plus de l'effort consacré à rendre ce logiciel utilisable par tous, un pan complémentaire de ce travail a consisté à l'évaluer dans des conditions expérimentales inacessibles jusqu'à peu. En effet, depuis 2005, nous avons accès à la plate-forme expérimentale nationale Grid'5000. Les expérimentations présentées dans ce travail ne sont pas triviales et ont demandé un temps important de mise en place. Peu de travaux ont exploré expérimentalement l'exécution de programmes à passage de messages comptant 600 processus, et répartis géographiquement à l'échelle de la France.

Cette thèse démontre la faisabilité d'un intergiciel destiné aux grilles de calcul, prenant en compte la dynamicité de ce type de plateforme, et les impératifs des programmes parallèles à passage de message. Pour cela, nous mettons en avant l'intérêt d'utiliser une architecture la plus distribuée possible : nous reprenons l'idée d'une infrastructure pair-à-pair pour l'organisation des ressources, qui facilite notamment la découverte des ressources, et nous retenons les détecteurs de défaillance distribués pour gérer la tolérance aux pannes.

La dynamicité de ce type d'environnement est également un problème pour le modèle d'exécution sous-jacent à MPI, car la panne d'un seul processus entraine l'arrêt de l'application. La contribution de P2P-MPI dans ce domaine est la tolérance aux pannes par réplication. Nous pensons qu'elle est la mieux adaptée à une architecture pair-à-pair, les techniques classiques basées sur le *check-point and restart* nécessitant un ou des serveurs de sauvegardes. De plus, la réplication est totalement transparente à l'utilisateur et rejoint ainsi l'objectif de simplicité d'utilisation que nous nous sommes fixés. Nous pensons que garder un environnement très simple d'utilisation, entièrement maîtrisable par un utilisateur, est un des facteurs permettant d'augmenter le nombre de resources disponibles sur la grille. Enfin, la contribution majeure de P2P-MPI est la librairie de communication proposée, qui est une implémentation de MPJ (MPI adapté à Java), et qui intègre la réplication des processus. Ce point particulier de notre travail plaide pour une collaboration étroite entre l'intergiciel, qui connaît l'état de la grille (détection des pannes par exemple) et la couche de communication qui peut adapter son comportement en connaissance de cause.

Chapter 1

Introduction

The concept of *Grid* has recently emerged to express the possibilities that networking technologies let encompass in terms of computer usage. An overview of these possibilities and problems to overcome is given by Foster and Kesselman [5]. *Grid computing* offers the perspective of solving massive computational problems using a large number of computers arranged as clusters embedded in a distributed telecommunication infrastructure. It involves sharing heterogeneous resources (based on different platforms, hardware/software architectures) located in different places, belonging to different administrative domains over a network. When speaking of computational grids, we must distinguish between grids involving stable resources (e.g. a supercomputer) and grids built upon versatile resources, that is computers whose configuration or state changes frequently. (e.g. computers in a students computer room which are frequently switched off and whose OS is regularly re-installed). The latter are often referred to as *desktop grids* and may in general involve any unused connected computer whose owner agrees to share its CPU. Thus, provided some magic middleware glue, a desktop grid may be seen as a large-scale computer cluster allowing to run parallel application traditionally executed on parallel computers. However, the question of how we may program such cluster of heterogeneous computing resources remains unclear.

This thesis work mainly focuses on this challenging issue. Even though some popular projects, such as SETI@home[6], Folding@home[7], etc, have been able to demonstrate the use of up to thousands of personal computers or even gaming consoles as a distributed computing infrastructure, their applicability is limited to embarrassingly parallel computations (fully independent computation tasks). Moreover, each of these projects is often dedicated to a unique application. Our work on the contrary, addresses the capability to program and run on personal computers any parallel application type. For that, we propose a programming model based on message-passing.

A number of research work have proposed more general programming environments, which we detail in Chapter 2. Most of the numerous difficulties that appear when designing such general programming fall in two categories.

- **Middleware** The middleware management of tens or hundreds grid nodes is a

tedious task that should be alleviated by mechanisms integrated to the middleware itself. These can be fault diagnostics, auto-repair mechanisms, remote update, resource scheduling, data management, etc.

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

MPI [1] is the *de-facto* standard for message passing programs. Most MPI implementations are designed for the development of highly efficient programs, preferably on dedicated, homogeneous and stable hardware such as supercomputers. Some projects have developed improved algorithms for communications in grids but still, assume hardware stability. This assumption allows for a simple execution model where the number of processes is static from the beginning to the end of the application run[1]. This design means no overhead in process management but makes fault handling difficult: one process failure causes the whole application to fail. This constraint makes traditional MPI applications unadapted to run on grids because failures of nodes are somehow frequent in this context.

Another drawback of most MPI implementations lies in the cumbersome management of files. First, running on different operating systems implies to manage several executable file types, as MPI applications are made of OS-dependent binaries[2]. Secondly, an MPI application often relies on an existing file sharing system such as *NFS* (Network File System), to stage executables and input data files to all processors. When the set of computers is composed of more than one operating system (imagine the campus Desktop Grid which has Windows and Linux PCs), we have to compile two versions of the executable file (one for Windows and one for Linux) and then we have to copy each version of the executable file to the proper machine manually.

If we put these constraints altogether, we believe a middleware should provide the following features:

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

Data management. It is a tedious task to copy the executable files (MPI applications) and input data to all computing hosts. Thus, the middleware should handle the data management which means transfer the executable files and input data to all computing hosts transparently and automatically when the users execute a task.

[1] Except dynamic spawning of process defined in MPI-2.

[2] The MPI specification defines bindings for C, C++ and Fortran only.

Fault management. When the size of the Grid becomes significant, the mean time between failure (MTBF) of CPU nodes becomes a seriously limiting factor. The middleware should provide automatic and transparent mechanisms to detect and handle nodes failures.

Abstract computing capacity. A Grid is by nature composed of heterogeneous resources, and in particular, we may have to deal with a variety of operating systems. The middleware should provide some facilities for programmers to deploy their applications regarding this aspect.

The idea that we propose is related to the *peer-to-peer* model we will discuss in detail later in chapter 2, page 47. These last years, many projects in the field of distributed systems have been based on the peer-to-peer model, especially for file sharing. They proved to be reliable and efficient enough from the user point of view if we consider their popularity. We think this model has interesting properties that could serve as a basis for fault-tolerance, self-configuration and autonomy. Our work aims to propose a middleware infrastructure able to support the execution of parallel programs using a message passing programming model and whose features meet the list above. We call this infrastructure a *platform*. This platform is designed to support a subset of the standard MPI specification (the minimum set required to program with the message passing paradigm).

Publications

International Conferences

(1) A Peer-to-Peer Framework for Robust Execution of Message Passing Parallel Programs, Stéphane Genaud and Choopan Rattanapoka, *EuroPVM/MPI 2005*, LNCS, vol. 3666, Springer-Verlag, pages 276–284, Ed. B. Di Martino et al., September 2005.

(2) Fault management in P2P-MPI, Stéphane Genaud and Choopan Rattanapoka, *In proceedings of International Conference on Grid and Pervasive Computing, GPC'07*, LNCS, vol. 4459, Springer, Ed. C. Cérin and K.-C. Li, Paris, May 2007.

(3) Large-Scale Experiment of Co-allocation Strategies for Peer-to-Peer Supercomputing in P2P-MPI, Stéphane Genaud and Choopan Rattanapoka, *5th High Performance Grid Computing International Workshop, IPDPS conference proceedings*, IEEE , Miami, April 2008.

Journals

(1) P2P-MPI: A Peer-to-Peer Framework for Robust Execution of Message Passing Parallel Programs on Grids, Stéphane Genaud and Choopan Rattanapoka, *in Journal of Grid Computing*, volume 5(1), pages 27-42, Springer, ISSN:1570-7873 2007.

(2) Exploitation of a parallel clustering algorithm on commodity hardware with P2P-MPI, Stéphane Genaud, Pierre Gançarski, Guillaume Latu, Alexandre Blansché, Choopan Rattanapoka and Damien Vouriot, *in The Journal of SuperComputing*, volume 5, Ed. Springer, Springer, ISSN:0920-8542 (Print) 2007.

Manuscript Organization

This manuscript is divided into five main chapters. Chapter 1, this chapter, is an introduction to my thesis. Then, in Chapter 2 we discuss the ideas and how the other research projects in the area compares to our work. The main part of the work is detailed throughout chapters 3, 4, and 5. The contribution of P2P-MPI is its integrated approach: it offers simultaneously a middleware with many of the desired features cited in introduction, and a general parallel programming model based on a MPI-like programming model. The integration allows the communication library to transparently handles robustness by relying on the internals of the middleware, relieving the programmer from the tedious task of explicitly specifying how faults are to be recovered. We describe these three aspects in separate chapters. Chapter 3 explains the middleware core of P2P-MPI. Chapter 4 is a study of the fault management in P2P-MPI. This point is linked both to the middleware resource allocation to meet replication constraints, and fault detection and notification) and to the communication library (handling of communications in presence of replicated processes). Finally, Chapter 5 explains the communication library capabilities.

Each chapter contains some experimental results. The experiments we conducted aim to assess the validity of our proposals. When it was possible, we tried to conduct these experiments at a large scale. We had the opportunity during this thesis to use the Grid'5000 experimental platform. So, except some simulation results, all experimental assessments have been done on a real environment.

Finally, we conclude in Chapter 6 by discussing the advantages, disadvantages, limitation and future work to be done on P2P-MPI.

Chapter 2

State of the Art

The term *computational grid* encompasses many different usages. We have mentioned in the introduction projects that uses thousands of individual CPUs to solve some so-called *embarrassingly parallel* problems i.e, made of independent tasks. We could cite as another example, the European EGEE grid infrastructure [8], which provides scientific communities with a distributed computing equipment, claims to count 41,000 CPU and 5 PB disk. Originally designed for the needs of two scientific fields, namely high energy physics and life sciences, EGEE now integrates applications from many other scientific fields, ranging from geology to computational chemistry.

Corresponding to such varied infrastructures or such different application requirements are different programming models or programming tools. A user may need to execute a workflow of sequential tasks, explore a solution space using a parameter-sweep application, use a problem-solving environment such as Ninf [9] or Netsolve [10], or run a parallel scientific code, to name just a few of these usages.

We are interested here in the latter of the above examples. A user has developed a parallel program using the message-passing paradigm and he seeks computational resources to run it. The *de-facto* standard to write message-passing parallel programs is the MPI (Message Passing Interface) specification [1]. As the MPI standard has been initially designed for high performance, it is used in most cases for applications aimed at clusters and dedicated MPP systems. In our work, we consider the requirements to (seamlessly) run an MPI application on computational grids which are not solely composed of clusters or supercomputers. This consideration involves to solve a number of problems.

Fault-tolerance A key feature of MPI is that applications using it are designed along a static process model[1]. More precisely, the static process model implies that during one MPI application run, MPI creates and manages a communication table called *communicator*, for each MPI process to know how to contact each other. So when one of the

[1]MPI-2 has extra facilities to bypass the static model (e.g. MPI_Spawn). However, it has not been completely designed to fit the dynamicity of grid environments.

computing node fails, there will be a hole in the communicator that causes the whole application to fail. Currently, the trend is to exploit platforms with more and more nodes so we cannot ignore the problem of node failures and the heterogeneity of systems.

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

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

System State Dynamicity MPI implementations usually consider a static set of computational resources: a list of computers is listed in a *hostfile* and processes are mapped in round-robin fashion onto these computers. This is totally unadapted to the dynamicity of Grids: the set of available computers changes frequently, the CPU occupation of each node varies continuously, the bandwidth between network links is also constantly changing, the software on nodes changes regularly and in the worst case (desktop grids) nodes can join and leave at anytime. It is hard for programmers to handle this situation themselves. We should provide a middleware which keeps the dynamicity of nodes in Grid transparent to programmers by providing some mechanism that would dynamically request available nodes for a computation. Currently, the peer-to-peer model has proved to be good in harness environments, as demonstrated by the success of peer-to-peer file sharing applications.

In this chapter, we first discuss grid usages and then focus on the software infrastructure parts and we will be particularly interested in the programming model for grids. Then, we review existing research works on many projects that have tried to adapt MPI to more versatile environments than the traditional parallel computers and we review the efforts made in this domain. Thus, we do need to consider fault tolerance which has been studied for a long time in distributed systems as a key feature. Last, we will give a quick overview of peer-to-peer topologies.

2.1 Grid Usages

As stated in the introduction, the usages of Grids today, are extremely varied depending on the users' needs, the nature of the resources in the network, and the grid software deployed. Many trends of large-scale distributed systems in different areas, started as early as in the mid 1980's, have found a common denominator in the Grid concept. Yet, these trends in using distributed systems may have very different focuses. Hence, any classification of Grid usages is subject to controversy. So, the classification we give below mainly aims at citing some of the most well-known projects in the field.

Meta-computing

The origin of the terms *meta-computer* and *meta-computing* are believed to have come out of the CASA project [11], one of several U.S. Gigabit testbeds around in the late 1980s. Catlett and Smarr have related the term meta-computing to "the use of powerful computing resources transparently available to the user via a networked environment" [12]. We consider here the term meta-computing as applying for grids composed of stable resources, and often expensive computing equipments, with well-provisioned networks.

A couple of middleware systems have been used extensively in this context.

Globus [13]. Started in 1995 by the U.S. Argonne National Laboratory, the University of Southern California's Information Sciences Institute and the University of Chicago, the Globus project has given rise to the middleware that had the highest impact on the grid community and on grid technologies evolution.

It is a enabling technology for the Grid, letting people share computing power, files, and other tools securely online across corporate, institutional, and geographic boundaries. One key feature of Globus is its security infrastructure (a public key infrastructure) which makes possible to gather resources from multiple administrative domains.

The success of Globus lies in its "toolkit" design, that is some minimal software bricks able to collaborate to provide some services. More precisely, the basic bricks for which the Globus Toolkit has implementations are:

- Resource management: Grid Resource Allocation and Management Protocol (GRAM),
- Information Services: Monitoring and Discovery Service (MDS),
- Security Services: Grid Security Infrastructure (GSI),
- Data Movement and Management: Global Access to Secondary Storage (GASS) and GridFTP.

The development roadmap of Globus follows the specifications developed inside the Open Grid Forum [2] (OGF), formerly known as Global Grid Forum before 2006.

[2] http://www.ogf.org/

Globus has been used as a basis in a number of other projects, including the GLite middleware (and its predecessors EDG then LCG) used in the EGEE grid, and the Advance Resource Connector (ARC) middleware for NorduGrid.

Condor-G [14]. The Condor system [15], developed at the University of Wisconsin-Madison, was originally termed by their authors as a *High Throughput Computation* system [16]. Behind this term is the idea of a system able to deal with coarse-grained computationally intensive tasks. Tasks can be either sequential or parallel jobs. Parallel jobs support the MPI and PVM standards in addition to its own Master Worker MW library for extremely parallel tasks. Condor is typically used to schedule computational jobs on a dedicated cluster of computers, or to farm out work to idle desktop computers in a cycle stealing way.

Condor-G adds to the original Condor software extensions to support some of the Globus protocols. The compatibility with the Globus security and authentication infrastructure widens the geographic scale at which Condor can be deployed. With the Globus extension, Condor-G combines the inter-domain resource management protocols of the Globus Toolkit and the intra-domain resource and job management methods of Condor to allow the user to harness multi-domain resources as if they all belong to one personal domain.

Global Computing

Global Computing achieves throughput computing by harvesting numerous unused computing resources connected to the Internet. The aim is to aggregate a substantial computational power in order to tackle problems that cannot be solved on a single system. Global Computing differs from Meta-computing in the nature of the resources involved. This has a deep influence on the middleware design as well as the candidate applications. Contrarily to Meta-computing, Global Computing does not assume the presence of stable resources in general. Hence, the middleware must address the problem of the resource volatility. Some interesting projects in this category are Condor, XtremWeb or Legion. Condor has been described above through Condor-G.

XtremWeb [17] XtremWeb is a typical example of middleware designed to tackle this goal. XtremWeb's software architecture is composed of: (1) clients which submit tasks, (2) workers which represent the pool of computational resources, (3) servers which connect clients with workers and (4) result collectors. Clients submit their tasks to a server which maintains a pool of these tasks. Upon starting, workers register and authenticate to a server (last contacted or root server) and receive back from the server a list of servers (including itself) which may provide tasks. Workers then send a work request together with a description of their environment. According to this information, the server selects a task, and sends back a description of the task, the tasks inputs, the binary of the application, and the address of a collector. During the computation, a worker periodically sends *alive* messages to the server. The server which monitors the worker will re-schedule the

task to another worker if no alive message has been received before a time-out. In this protocol, all network connections are initiated by clients or workers with the objective to avoid firewall problems. It is noteworthy that this design feature is dictated by the nature of resources.

Legion [18] is one of the pioneer middleware system that has addressed grid computing. It is based on an integrated object-oriented architecture, to which all services and program must conform to. To develop a new component, a programmer plugs its new object into the common programming interface so it can communicate with the already established object model. A communication library called the *Legion run-time library* is the building block for high level languages (e.g Mentat).

Internet computing

Internet computing can be seen as a particular case of distributed computing. The constraints of the previous projects are that the applications must be *embarrassingly parallel*, and have low requirements in the data volume communication. The principle of harvesting CPU cycles when a user leaves its machine idle is comparable to many projects in global computing. Many internet computing projects use the processor in low priority, for instance when the screen-saver starts. Contrarily to global computing, most projects of internet computing are centered onto one application only. One of the first Internet computing project is SETI@Home[6].

Data storage

In data-intensive applications, the focus is on synthesizing new information from data that is maintained in geographically distributed repositories, digital libraries, and databases. This synthesis process is often computationally and communication intensive as well. The first example of this application was the European DataGrid project [19]. One of the primary project's aims was to store the huge amount of data the upcoming Large Hadron Collider (LHC) instrument will produce. More generally, this initiative pursued by the EGEE project has the objective to federate scientific communities in virtual organizations. The argument is that grids will enable next generation scientific exploration which requires intensive computation and analysis of shared large-scale database, across widely distributed scientific communities.

2.2 Programming Environments for Grids

Most people are convinced that Grids offer unprecedented possibilities for a wide range of distributed or parallel applications. However, the question of how such a set of resources may be programmed seems to be at its beginning. The programming models that are currently used are not new. So far, researchers have tried to adapt existing programming models, and have tried to identify which application type fits better into which programming environment. That is exactly what this thesis work is also doing:

we give some insights to evaluate how the message-passing paradigm could be adapted to a Grid environment.

In this section, we summarize the main approaches regarding programming models, and we illustrate these through typical research projects. We distinguish three major categories of programming models: the client/server programming model, the peer-to-peer programming model and the parallel programming model.

2.2.1 Client/Server Programming Model

This model is probably the most popular programming model for Grids because of its simple concept. The model assumes a client which handles the main program sequence of instructions, and initiates requests to servers when needed. For example, a client can request a server hosted on a powerful computer to perform a complex computation.

This model is characterized by its communication scheme, which is typically limited to communications between the clients and the servers. Many well-known projects (such as [6]) follow this model to implement embarrassingly parallel applications. From a technical point of view, there is no problem with making clients communicate with other clients, or make a server become a client. However, such sophistications in an application's structure leads to complex problems regarding the management of failures. While the simple client-to-servers communication scheme allows to easily recover from server failures (for instance, if a server has not returned a result before a time-out, ask another available), the management of synchronizations and interdependence between clients is difficult.

We now list some representative projects using this programming model:

DIET [20] stands for Distributed Interactive Engineering Toolbox. The project targets the development of scalable middleware with initial efforts focused on distributing the scheduling problem across multiple agents. DIET consists of a set of elements that can be put together to build applications using GridRPC [21], which is a RPC paradigm. The middleware is able to find an appropriate server according to the information given in the client's request, the performance of the target platform and the local availability of data stored during previous computations. The scheduler is distributed using several collaborating hierarchies connected either statically or dynamically (in a peer-to-peer fashion). Data management is provided to allow persistent data to stay within the system for future re-use.

2.2.2 Peer-to-Peer Model

In a peer-to-peer architecture, computers that have traditionally been used solely as clients communicate directly among themselves and can act as both clients and servers, assuming whatever role is most efficient for the network. This reduces the load on servers and allows them to perform specialized services more effectively. As computers become ubiquitous, ideas for implementation and use of peer-to-peer computing are developing rapidly and gaining importance. Both peer-to-peer and grid technologies focus on the

flexible sharing and innovative use of heterogeneous computing and network resources. In section 2.6, we describe more precisely peer-to-peer topologies and some existing projects that might be considered a basis to build a peer-to-peer model application.

2.2.3 Parallel Model

We call *parallel model* a programming model in which the program consists in a set of processes, all running concurrently. The synchronizations between processes are explicitly expressed by the programmer in its source code through communications, which are based on send and receive primitives. The distinctive characteristic of this execution model as compared to the client/server or peer-to-peer models, is that any process has an exact and persistent knowledge of all the other processes addresses and may communicate with them at any moment.

Popular representatives of this programming model are PVM [22] and MPI [1]. In the field of high performance applications for clusters and supercomputers, MPI is definitively the most used API today. Some of the most popular implementations of MPI are MPICH[23], MPICH2[24] and LAM/MPI[25] before OpenMPI [26] was created.

It is foreseen that grids may also become an interesting exploitation platform for high performance applications. Yet, several inherent characteristics of grids make their efficient exploitation a challenge. The first challenge is related to the heterogeneity of all the elements composing grids. Processors, as well as network links are heterogeneous. In the following section, we report the efforts made in several research works to improve standard MPI implementations so as to improve communication performances in the presence of mixed wide and local area networks. The problem of processors heterogeneity is not addressed by these improved libraries. A second challenge lies in the management of the grid dynamicity. This challenge involves the improvement of middleware services. It is especially important for MPI applications and its support for fault tolerance. This issue is reviewed in Section 2.4.

2.3 MPI and Grids

One of the first efforts made by researchers has been to adapt MPI existing implementations to the wide-area network context found in meta-computing. The major advantage is to allow a straight-forward port of existing MPI applications to grids. Such an example is MPICH-G2 (see below) which is a specific device of MPICH developed to work with Globus. An early demonstration of a large application deployment can be found in [27]. This is an MPI application for numerical simulation in the astrophysics field. About 1500 processors were used on two sites (SDSC at San-Diego and NCSA at Champaign-Urbana) with four parallel supercomputers (IBM Power-SP and three Origin 2000). The wide-area network link between the sites had a 622 Mb/s bandwidth, while the link between the three Origin 2000 is a gigabyte ethernet link.

In addition to the technical difficulty to make several sites communicate, the main

contribution of these projects is the algorithms designed to take advantage of the network heterogeneity. Here under, we review a couple of the most well-known projects in this area:

MagPIe [28], developed at the Vrije Universiteit, Amsterdam, has been one of the first proposals and concrete implementations (on top of mpich-1.1) of collective communication optimizations for grids. MagPIe tries to take advantage of the hierarchical structures within the communication network to improve collective communications in the context of meta-computing. The network structure assumed in this work is a two-level structure: several clusters are linked with wide area links which have a relatively low bandwidth and high delay as compared to cluster intra-communications. The optimization relies on the idea that it is possible to use the wide-area link only once during any collective operation. The results have been experimentally validated on the DAS system, a federation of clusters throughout the Netherlands. The network structure of this testbed at the time of writing the paper [28] matched the above assumptions: the wide-area links had a bandwidth of 6Mb/s and a latency of 10ms, versus 66MB/s and 20μs latency within the clusters.

PACX-MPI [29] is an implementation of the Message Passing standard MPI, optimized for Metacomputing. The major goal of the library is to make MPI applications run on a cluster of MPP's and PVP's without any changes in the sources and by fully exploiting the communication subsystem of each machine. To reach this goal, PACX-MPI makes use of the vendor MPI library on the systems, since this is currently the fastest portable API to the communication subsystem of each machine.

MPICH-G2 [30] is a grid-enabled implementation of the MPI v1.1 standard. That is, using services from the Globus Toolkit (e.g., job startup, security), MPICH-G2 allows you to couple multiple machines, potentially of different architectures, to run MPI applications. MPICH-G2 automatically converts data in messages sent between machines of different architectures and supports multi-protocol communication by automatically selecting TCP for intermachine messaging and (where available) vendor-supplied MPI for intramachine messaging.

GridMPI [31] is a recent project from the National Institute of Advanced Industrial Science and Technology (AIST) of Japan. This is another grid-enabled MPI implementation whose aim is to optimize collective communication performances. The main difference with projects cited above, is that the authors make the assumption that today, the wide-area links between sites have a much higher bandwidth than intra-cluster communication links. Indeed, their claim is verified by modern network backbones performance. Hence the algorithms of collective communications must be redesigned accordingly. They propose algorithms adapted from work from van de Geijn et al [32] and Rabenseifner [**?**]. The algorithms utilize multiple node-to-node connections while regulating the number of nodes simultaneously communicating, and improve the performance of collective operations on large messages.

Experiments using an emulated WAN environment with 10 Gbps bandwidth and a 10ms latency confirm the gain over standard MPI implementations.

The MPI implementations listed above have tackled the performance issue in mixed wide and local area networks but they do not address the *dynamicity* issue. Indeed, grids are dynamic environments where resources may see their hardware or software configuration evolve, or where resources may appear, disappear or change their availability status at anytime. This is of course a major factor that makes grid application highly failure-prone.

In the next section, we discuss fault tolerance, which we think is a key feature in a grid context. Fault-tolerance involves two different issues. The first one is fault detection. It is a problem in itself to design a scalable, reliable fault detection system, and we review existing work in Section 4.6. The second issue is fault recovery. Several approaches have been proposed to design systems able to prevent failures, and we list the main ideas in Section 2.4.2. Finally, we list in Section 2.4.3 several projects that have integrated fault tolerance in their MPI implementations.

2.4 MPI and Fault Tolerance

2.4.1 Fault Detection

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

The implementation of failure detection protocols are based on timeouts. There are two basic models of fault detector which are discussed in [33]. One is the *push model* and the other is the *pull model*.

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

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

Pull Model In this model, monitored components are passive while the monitor or failure detector is active. The monitor periodically sends liveness requests ("*Are you alive?*" messages) to monitored components. Upon reception of a liveness request, the

monitored component sends a reply to the monitor. When the monitor does not receive a reply from a monitored component within a certain time interval (timeout), it suspects the monitored component has failed.

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

Thus, there are several caveats we should keep in mind when choosing or designing our failure detection system.

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

Scalability MPI applications running on a Grid system may require a large number of resources distributed over a wide area network. A failure detection service must be able to efficiently monitor such large number of resources. It must be able to quickly detect failure while minimizing the number of wrong suspicions.

The two previous models (push and pull) behave well in small-scale systems but at a larger scale, a more interesting approach has been proposed in the last decade. Following the idea that failure detectors should be considered as first class services of distributed systems [34], many protocols for failure detection have been proposed and implemented. After a review of existing proposals, we retain the *gossip-style* fault detection service proposed by van Renesse [4].

Gossip-style Protocol

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

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

2.4.2 Fault Recovery Techniques

As previously explained, a major drawback of MPI on Grids is that the execution model assumes a constant number of processors during program execution. This assumption is not suitable regarding the dynamicity of grids. Hence, MPI implementations for Grids should be *fault tolerant*. Fault tolerance means that the application should not automatically abort at the first process fault, but should take an appropriate action instead. It might be recovery but other alternatives (e.g shrinking the communicator) are possible. FT-MPI[35] (detailed below) proposes the most general framework for that purpose. However, most of the time users wish their executions to complete in spite of failures without modification to their application code. This requires to provide the runtime support with one of two following following fault tolerance mechanisms. The first one is *rollback-recovery*, also known as *check-point and restart*. This approach has been widely studied and many implementations have been proposed. The other approach is *replication*. Applied to MPI, fault-tolerance consists in replicating some or all processes of the application. This approach has not been well studied and to the best of our knowledge we do not know any other project dealing with process replication for MPI.

Rollback-Recovery Techniques

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

The global checkpoint consists in taking a snapshot of the entire system state regularly without the assumption of a global clock, but by using the concept of logical clock introduced by Lamport [36]. So, when a failure occurs on any process, the whole system can roll back to the latest checkpointing image and continue the computation.

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

Global Checkpoint Based There are three classes of global checkpoint protocols [37]: *uncoordinated*, *coordinated checkpoint* and *communication induced*.

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

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

- Communication Induced Checkpoint (CIC) tries to take advantage of uncoordinated and coordinated checkpoint techniques. Based on the uncoordinated approach, it piggybacks causality dependencies in all messages and detects risks of inconsistent states. When such a risk is detected, some processes are forced to checkpoint. While this approach is very appealing theoretically, relaxing the necessity of global coordination, it turns out to be inefficient in practice [38]. The two main drawbacks in the context of cluster computing is (1) CIC protocols do not scale well (the number of forced checkpoints increases linearly with the number of processes) and (2) the storage requirement and usage frequency are unpredictable and may lead to checkpoint as frequently as the coordinated checkpoint technique.

Message Log based According to Alvisi and Marzullo [39], message log protocols fall into three categories: *pessimistic*, *optimistic* and *causal*.

- Pessimistic log protocols ensure that all messages received by a process are first logged by this process before it causally influences the rest of the system. MPICH-V (see Section 2.4.3) is based on this type of protocol. It uses reliable processes called Channel Memories. Every MPI computing is connected to a channel memory. When a node sends a message, it sends it to the channel memory of the receiver, and when it wants to receive a message it asks its own memory channel for it.

- The optimistic log protocols [40] eventually log receptions but do not wait for them before sending new messages. Therefore, they are faster in non-faulty executions but do not exclude to rollback some non-crashed processes if a fault occurs before the reception logging.

- Causal log protocols try to combine the advantages of the optimistic and the pessimistic approaches. Its has a much lower overhead than pessimistic logging while there is no rollback for non faulty processes. This is achieved by piggybacking events (its past receptions) to messages until these events are safely logged.

Replication Techniques

In replication techniques, a process is replicated and we called the replicated process *replica*. These replicas are placed on different computers. Even if some of the replicas fail, the others continue to process the application. There are two kinds of replication techniques, one is the *active replication* [41] and the other one is *passive replication* [42] (also known as *primary backup*).

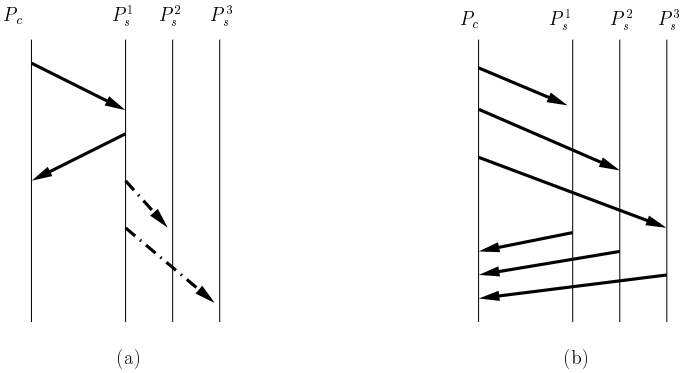


Figure 2.1: Passive and active replication.

To illustrate these two techniques and for the sake of clarity, we choose to replicate only the server process. Let P_c be a client process which sends a message to a server process P_s. To deal with possible fault-tolerance of server process P_s we use replication and provide $P_s^1, P_s^2, ..., P_s^n$ processes where n is the number of replicas of P_s. We say a replica is *operational* if it is not idle and executes its code. Otherwise, a replica is idle waiting to be woken up.

Passive Replication In the passive replication, only one replica is operational. In figure 2.1 (a), client P_c sends a message to only one replica, for example P_s^1. Then only P_s^1 performs the requested operation and returns the result. The other replicas P_s^i $(2 \leq i \leq n)$ are not operational. The state of the passive replicas are updated by receiving the newest state information from the operational replica P_s^1 from time to time. This is called a *checkpoint*. If P_s^1 fails, one of the passive replicas, say P_s^2 takes over it. In order to catch up with the failed P_s^1, P_s^2 starts to execute the application from the most recent checkpoint. However, the recovery procedure takes time since P_s^2 becomes operational at the checkpoint and re-executes the operations that P_s^1 had already done since the last checkpoint.

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

2.4.3 Fault Tolerant MPI implementations

Let us review now some research projects that proposed MPI implementations supporting fault-tolerance. Some have proposed modifications or extensions to an existing MPI implementations while others have developed their library from scratch.

CoCheck [43] is one of the earliest efforts to make MPI more reliable. CoCheck extends the single process checkpoint mechanism used in Condor to a distributed message passing application. Common problems with checkpointing and recovery such as global inconsistent states and domino effects are eliminated through the use of a protocol to flush all in-transit messages before a checkpoint is created. Consequently, CoCheck faces the problem of a large overhead because it checkpoints the entire process state.

Starfish [44] provides a parallel execution environment that adapts to changes in the cluster caused by node failure and recovery. The Starfish environment for execution of dynamic MPI programs is based on the Ensemble group communication system. Starfish uses an event model in which application processes register to listen for events reflecting changes in cluster configuration and process failures. Starfish also provides application- and system-driven checkpointing facilities. When a process failure is detected, Starfish can automatically recover the application from a previous checkpoint. However, consistency of communicators is not addressed in Starfish: in order to recover a single failed process, the entire MPI application must be restarted. Essentially, many of the powerful dynamic process management features of Starfish cannot be used directly by MPI applications.

MPI-FT [45] uses a similar approach to the one proposed for real-time data-driven systems. It is based on a monitoring process, called the *observer*, which will notify the rest of the processes in the event of a failure, and the action to be executed for recovery. Two different modes are proposed. In the first one, each process is responsible for buffering all message traffic it sends out while in the second case, all message traffic is buffered by the observer. Checkpoints are inserted explicitly in the code. They are actually tests for the arrival of the failure message which is received asynchronously by a non-blocking receive. The failure message is sent by the observer to the alive peers to invoke the recovery routine. MPI-FT solves the MPI problem of the dead communicator which refers to the fact that there is a death of a process by proposing two different solutions, either the preparation of spawning communicators in advance (one extra communicator to exclude each potential hole) or the pre-spawning of the replacement process when the program starts executing. However, the drawback of this system is the amount of memory needed for the observer process in long running applications.

FT-MPI [35] handles failures at the MPI communicator level and lets the application manage the recovery. When a fault occurs, all MPI processes of the communicator are informed about the fault. This information is transmitted to the application

through the returning value of MPI calls. The main advantage of FT-MPI is its performance since it does not checkpoint nor log, but its main drawback is the lack of transparency for the programmer.

MPICH-V [46] is a mix of uncoordinated checkpointing and a pessimistic message logging protocol storing all communications of the system on a reliable media. To ensure this property, every computing process is associated with a reliable process called Channel Memory. Every communication sent to a process is stored and ordered on its associated Channel Memory. To receive a message, a process sends a request to its associated Channel Memory. After a crash, a re-executing process retrieves all lost receptions in the correct order by requesting them to its Channel Memory. The use of Channel Memory however, has a major impact on the performance (dividing the bandwidth by a factor of two) and on the cost of the fault tolerance system (high performance requires a large number of Channel Memories).

MPICH-V2 [47] is an improved version of MPICH-V designed to overcome the major impact on the performance of using Channel Memory. In MPICH-V2, the message logging is split into two parts: on one hand, the message data is stored on the computing node, following a sender-based approach. On the other hand, the corresponding event (the date and the identifier of the message reception) is stored on an event logger which is located on a reliable machine. However, MPICH-V2 still needs reliable nodes for the fault tolerant system.

MPI/FT [48] is the closest project to our proposal. It provides fault-tolerance to MPI by introducing process replication. Using these techniques, the library can detect erroneous messages by introducing a vote algorithm among the replicas and can survive process-failures. The drawback of this project is the increasing resource requirement by using replicating MPI processes but this drawback can be overcome by using large platforms such as Grid or desktop Grid.

Open MPI [26] Open MPI initially represented the merger between three well-known MPI implementations:

- FT-MPI from the University of Tennessee
- LA-MPI from Los Alamos National Laboratory
- LAM/MPI from Indiana University

with contributions from the PACX-MPI team at the University of Stuttgart. Each of these MPI implementations excelled in one or more areas. The driving motivation behind Open MPI is to bring the best ideas and technologies from the individual projects and create one world-class open source MPI implementation that excels in all areas.

Open MPI was started with the best of the ideas from these four MPI implementations and ported them to an entirely new code base. As such, Open MPI also

contains many new designs and methodologies based on (literally) years of MPI implementation experience.

2.5 MPI and Java

Due to its popularity in nearly all fields of software development, Java has been also considered as a candidate for parallel programming. Java integrates a number of handy constructs for network programming and propose RMI for distributed computing. However, it has the reputation of being "slow" i.e, there does not exist a Java Virtual Machine able to execute Java code as fast as its pendant in C for example. This reputation has first discourage people to use it for high-performance computing. With grids and heterogeneous resources, the "run everywhere" property of Java becomes a strong argument in the tradeoff between execution efficiency and deployment efficiency. It is also noteworthy that JVM performances have improved their performances a lot.

As a matter of fact, a community of researchers have put efforts to extend Java with constructs dedicated to high performance computing. Many discussions have taken place at the Java Grande Forum between 1998 and 2003. One recommendation from the *Message Passing* Group of this forum is of particular interest for us: the MPJ (Message Passing for Java) [3] offers an equivalent to the MPI specification for C/C++/Fortran.

Several research works have proposed implementations of MPJ. For the most of them, the goal is to attain as good performances as the most popular MPI implementations (mpich, OpenMPI, ...). In addition, several network devices typical of clusters, such as TCP, Myrinet, Infiniband, are often considered in performance comparisons. Our proposal P2P-MPI is another MPJ implementation but its first objective was not to perform better regarding communication times. As we target grids, we initially put forward two features:

- the ability to restrict the port range used in each computer for communication. Even if the firewall rules limit the range of open ports, P2P-MPI is able to open and close TCP connections to match the rules.
- the communication library supports fault-tolerance.

P2P-MPI goal is to demonstrate that we can provide these features in an MPJ implementation. Nevertheless, we have recently worked at a more efficient implementation, dropping off the first feature (port range restriction). As other concurrent project, we have re-implemented the communication library using the Java NIO (detailed in section 3.1.3, page 57).

Below is a list of projects that have proposed a MPI-like implementation for Java.

Java-MPI [49] is a java interface to standard MPI. It is also an implementation of this interface which makes use of JNI wrappers to a native MPI package. In Java-MPI,

Java wrappers are automatically generated from the C MPI headers. This eases the implementation work, but does not lead to a fully object-oriented API.

MpiJava [3] is an object-oriented Java interface to standard MPI. MpiJava provides the full functionality of MPI 1.1. It is implemented as a set of JNI wrappers to native MPI packages.

MPJ Express [50] The motivation of MPJ Express project is that the earlier efforts for building a Java messaging system have typically followed either the JNI approach, or the pure Java approach. On commodity platform like fast ethernet, advances in JVM technology now enable networking applications written in Java to rival their C counterparts. On the other hand, improvements in specialized networking hardware have continued, cutting down the communication costs to a couple of microseconds. Keeping both in mind, the key issue at present is not to debate the JNI approach versus the pure Java approach, but to provide a flexible mechanism for applications to swap communication protocols. MPJ offers such a mechanism.

MPJ-Ibis [51] The implementation of MPJ at Vrije Universiteit, called MPJ/Ibis, relies on the Ibis [52] system. Ibis is a multi layer system, one of these being the Portability Layer (IPL). IPL provides an object-oriented interface to network communication primitives. Different programming models can be implemented above this layer, using the IPL interface. MPJ/Ibis is one of these programming models. It is a pure-Java implementation which has shown to deliver high-performance communications, while being deployable on various platforms, from Myrinet-based clusters to grids.

2.6 Peer-to-Peer Topologies

Generally, a peer-to-peer (or P2P) computer network refers to any network that does not have fixed clients and servers, but a number of peer nodes that function as both clients and servers to the other nodes on the network. This model of network arrangement is contrasted with the client-server model. Any node is able to initiate or complete any supported transaction. Peer nodes may differ in local configuration, processing speed, network bandwidth, and storage quantity. Popular examples of peer-to-peer are file sharing-networks. The peer-to-peer model consists of several topologies. Figure 2.2 shows three main topologies of peer-to-peer model (a) centralized topology, (b) decentralized topology, and (c) hybrid topology.

2.6.1 Centralized Topology

The centralized systems are the most familiar form of topology, typically seen as the client/server pattern used by databases, web servers, and other simple distributed systems. All functions and information are centralized into one server with many clients connecting directly to the server to send and receive information. Many applications

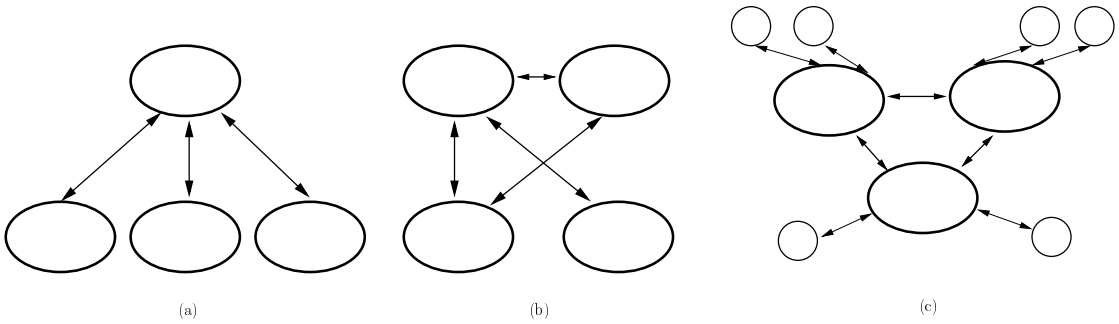


Figure 2.2: Three main peer-to-peer topologies.

called "peer-to-peer" also have a centralized component. SETI@Home is a fully centralized architecture with the job dispatcher as the server. Similarly, the original Napster's search architecture [53] was centralized, although the file sharing was not. The advantage of this topology is that searching other peers or services that other peers provide is efficient since the centralized server maintains all the information. However, the disadvantage is the bottleneck of the centralized server.

2.6.2 Decentralized Topology

Decentralized systems is where all peers communicate symmetrically and have equal roles. Gnutella [54] is probably the purest decentralized system used in practice today, with only a small centralized function to bootstrap a new host. Many other file-sharing systems are also designed to be decentralized, such as Freenet[55] or OceanStore[56]. Decentralized systems are not new; the Internet routing architecture itself is largely decentralized, with the Border Gateway Protocol used to coordinate the peering links between various autonomous systems. There is no bottleneck in this topology because there is no special centralized server. However, the performance of this topology suffers when searching other peers or their provided services.

2.6.3 Hybrid Topology

The distributed systems often have a more complex organization than one from a simple topology. Real-world systems often combine several topologies into one system, making a hybrid topology. Nodes typically play multiple roles in such a system.The hybrid topology overcomes the bottleneck of centralized topology and performance of decentralized topology.

2.6.4 Peer-to-Peer Infrastructure Projects

The peer-to-peer infrastructure is a main key for developing peer-to-peer applications. The peer-to-peer infrastructures should provide some basic facilities such as:

- joining the peer-to-peer network,
- discovering the other peers,
- self configuring and insuring robustness,
- providing application scalability.

The research projects in this domain aim to develop the peer-to-peer infrastructures that are easy for users to work with and also provide some more advantage features over other implementations. The list below gives some descriptions of the existing projects working on the improvement of peer-to-peer infrastructure.

CAN [57] the "Content Addressable Networks" work is being done at AT&T Center for the Internet Research at ICSI (ACIRI). In the CAN model, nodes are mapped onto a N-dimensional coordinate space on top of TCP/IP. The space is divided up into N dimensional blocks based on servers density and load information, where each block keeps information on its immediate neighbors. Because addresses are points inside the coordinate space, each node simply routes to the neighbor which makes the most progress towards the destination coordinate. Object location works by the object server pushing copies of location information back in the direction of the most incoming queries.

Chord [58] aims to build scalable, robust distributed systems using peer-to-peer ideas. The basis for much of its work is the Chord distributed hash lookup primitive. Chord is completely decentralized and symmetric, and can find data using only log(N) messages, where N is the number of nodes in the system. Chord's lookup mechanism is provably robust in the face of frequent node failures and re-joins.

Pastry [59] is a generic, scalable and efficient substrate for peer-to-peer applications. Pastry nodes form a decentralized, self-organizing and fault-tolerant overlay network within the Internet. Pastry provides efficient request routing, deterministic object location, and load balancing in an application-independent manner. Furthermore, Pastry provides mechanisms that support and facilitate application-specific object replication, caching, and fault recovery.

Tapestry [60] is an overlay location and routing infrastructure that provides location-independent routing of messages directly to the closest copy of an object or service using only point-to-point links and without centralized resources. The routing and directory information within this infrastructure is purely soft state (loosely coupled, anonymous fashion) and easily repaired. Tapestry is self-administering, fault-tolerant, and resilient under load.

JXTA [2] is a set of open, generalized peer-to-peer protocols, defined as XML messages. Using the JXTA protocols, peers can cooperate to form self-organized and self-configured peer groups independently of their positions in the network, and without the need of a centralized management infrastructure. Peers may use the JXTA protocols to advertise their resources and to discover network resources (service, pipes, etc.) available from other peers. Peers form and join peergroups to create special relationships. Peers cooperate to route messages allowing for full peer connectivity. The JXTA protocols allow peers to communicate without needing to understand or manage the potentially complex and dynamic network topologies which are becoming common.

Chapter 3

The P2P-MPI Middleware

P2P-MPI's final goal is to allow the seamless execution of parallel programs in grid environments. In this thesis, we try to demonstrate that having an execution model tightly coupled with the middleware brings many benefits with respect to that objective. Before discussing how the execution model interacts with the middleware, we give an overview of the whole architecture of P2P-MPI. The set of modules and functions that constitute P2P-MPI may conceptually be seen as a three layers stack.

On top of the stack is the *communication library* which exposes an MPJ API. The communication library represents the execution model. The MPJ specification (see Section 2.5, page 46) allows to write message-passing parallel programs in Java. The communication library relies on a *middleware* layer which provides different services to the communication library. These are the fault-detection service, the file transfer service, the reservation service, and discovery service. Most of these services rely themselves on a lower layer that deals with the resource management. Resource management consists to attribute identifiers to resources, locate available resources, etc. We call this layer *infrastructure* because the way resources are managed strongly depends on how the resources are organized. Very often, resources are registered in a centralized directory. Another approach is the organization of resources in a peer-to-peer architecture. This is the approach we chose because it has proved to ease self-configuration and autonomy of resources in grid environments.

This chapter first presents the general organization of all modules constituting P2P-MPI, their role and the layer they belong to. Then, we explain in Section 3.2 how these modules cooperate to fulfill the successive tasks needed to achieve a program execution. This requires from the middleware to build dynamically a suitable environment for the execution. A major point in this task is the discovery and the reservation of the resources. This is the subject of Section 3.3. Section 3.4 discusses the strategies which govern resource reservation. Experiments have been conducted to check that the strategies objectives are reached in real conditions, and results are reported in Section 3.5. Finally, we describe a complementary feature of the middleware in Section 3.6 : the monitoring

of peers. This is done with a graphical tool shipped with P2P-MPI that allows to have a global snapshot of the grid state.

3.1 General Architecture

Figure 3.1 presents the position of the P2P-MPI software in a usual Java running environment. P2P-MPI's parts are grayed out on the figure.

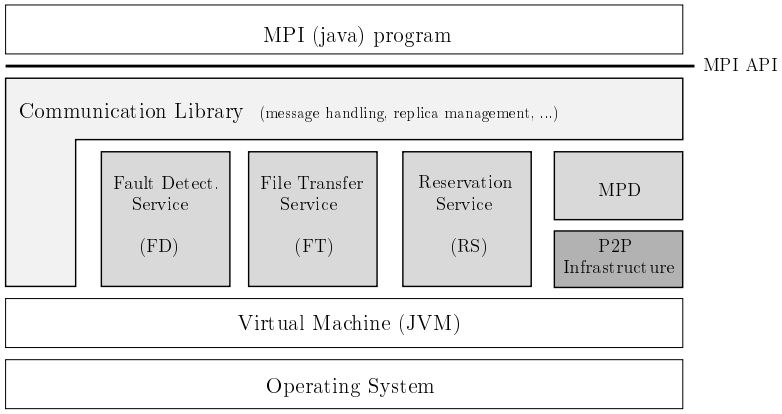


Figure 3.1: P2P-MPI structure.

P2P-MPI consists of three main parts. These are:

- the peer-to-peer infrastructure,
- the middleware,
- the communication library.

On top of the diagram, a message-passing parallel program written in Java uses the MPJ API to trigger functions from the communication library. The communication library implements appropriate message handling and relies on four other modules which are the daemon processes called `MPD`, `FT`, `FD`, and `RS`. The MPD process relies on the peer-to-peer infrastructure management layer to make the local resource a peer in peer-to-peer network. As P2P-MPI is written solely in Java and runs Java programs, it relies on the Java virtual machine installed locally, and of course on the local operating system.

3.1.1 The Peer-to-Peer Infrastructure

Since the beginning of the project, we have chosen a peer-to-peer (P2P) architecture because we believe this kind of resource networking allows a simpler infrastructure management. Our intention was to rely on some general purpose software able to handle

an underlying overlay network of peers. Thus, we considered that this layer should be implemented by an external, well-tested piece of software. We have studied some of the projects in this area, such as CAN, Chord, Pastry, Tapestry which were described in the previous chapter (section 2.6.4, page 49). Recall that these projects share the objective to store and retrieve objects in some *key* spaces. Objects and peers are given keys, which may be seen as their identifiers. Then, the infrastructure management service is able to search or insert (for example) an object on the peer whose key is the closest to the object's key in the key space.

For P2P-MPI, we first require the P2P infrastructure management service to provide a discovery service able to report some of the resource characteristics (for example CPU, RAM, hard disk capacity, etc.). In other words, we need to discover nodes (computers) more than objects on anonymous nodes. Our second requirement is technical: we need a freely available implementation that we can potentially adapt to our needs (open-source code), so that we can ship it as a library with the other modules of P2P-MPI. Moreover, this has to be a Java implementation not to break our "run everywhere" paradigm.

JXTA JXTA [61] is the P2P framework we originally chose to manage the infrastructure of P2P-MPI peers. To complete the description of JXTA made in Section 2.6.4, the strong advantages of JXTA are:

- it is an open-source initiative,
- it produces standard open protocols about what should be objects and operations in a P2P application. These are platform- and language-independent, XML-based protocols.
- it benefits from professional-quality implementations of the specifications in Java, C, C++ and C#. The project is supported by Sun Microsystems.

JXTA meets our requirements in that it provides an elegant publish/subscribe mechanism well adapted to our infrastructure management needs. A computer starting a JXTA application first joins a universal peer group called the *NetPeerGroup* (every operation takes place in a peer group in JXTA). Once it has joined the group, the peer inherits the services of the group, for instance peer discovery, pipe services, etc.

To advertise about its characteristics, the new peer may build a small XML file called *advertisement*, containing custom information besides the administrative data (unique identifier of the object, peer group, ...). The advertisement is then sent to a publisher called a *Rendezvous*. In fact, there are several Rendezvous that cooperate to store the advertisements in a distributed fashion. P2P applications may then call the discovery service to find other peers. Once a discovery request is issued, the service will asynchronously triggers events when advertisements matching the request's criteria are found.

However, due to its design, JXTA only discovers a "sufficient" number of resources, and there is no means to enforce the discovery service to deliver all known advertisements.

This can be considered a desired feature when targeting overlays at the scale of tens or hundreds thousands peers, to keep an acceptable amount of network traffic. A detailed study has been recently carried out about the behavior of JXTA RendezVous [62], which clearly shows that the peer view of the whole overlay broadens slowly in time and is always very partial. Moreover, because the JXTA design strongly relies on asynchronous events, we have no guarantee on the time usage to discover the resources. Also, the time needed for joining the JXTA NetPeerGroup on startup may be considered long, and may depend on the bootstrapping rendezvous (maintained by Sun) availability.

The delays observed in previous versions of P2P-MPI based on JXTA, were due to two startup operations. The MPD (which instantiates a JXTA peer) first joined the NetPeerGroup. Then, each MPD tried to join (or create if not yet created) a private peer group to isolate operations related to the P2P-MPI application. Typically, the time to complete both operations was 30 to 60 seconds using JXTA-J2SE version 2.3 in a 100 Mbps LAN environment.

Custom Infrastructure Management Due to further requirements concerning resource allocation (see Section 3.4) we have replaced [1] the JXTA layer with a new P2P infrastructure, designed more specifically for our needs. Our main requirement concerns network locality in the peer overlay: we want peers to be able to know how far are other peers in terms of network latency. We have reviewed related work sharing this concern. They share an architecture based on a P2P network. They face the same issue regarding resource allocation depending of network locality. For instance, the long-lived project ProActive [63] has added a P2P infrastructure to ease resource discovery. However, selection of resources for a computation only depends on their CPU load, as the infrastructure has no knowledge about network locality. Very close to our work are Zorilla [64] and Vigne [65]. They are two middleware systems which also build a P2P overlay network aware of peer locality. For that purpose, Vigne uses algorithms from the Bamboo project [66]. In Vigne, close resources are found using a simple (yet sometimes misleading) heuristic based on DNS name affinity: hosts sharing a common domain name are considered as forming a local group. Zorilla (which also uses Bamboo) proposes *flood scheduling*: the co-allocation request originated at a peer is broadcasted to all its neighbors, which in turn broadcast to their neighbors until the depth of the request has reached a given radius. If not enough peers accepted the job, new flooding steps are successively performed with an increasing radius until the number of peers is reached. The difficulty in this strategy, lies in finding suitable values for the flooding parameters, such as the radius and minimum delays between floods.

However, no software that we could use in replacement of JXTA was available as a well-separated and independent library. We thus implemented our own peer-to-peer infrastructure. This infrastructure management layer is simple, light, and fast. The benefits over JXTA in our context are the completeness and speed of resource discovery, and

[1] since p2pmpi-0.27.0.

the network latencies we can capture. Moreover, we could achieve experiments involving 600 processes with our new infrastructure whereas we were struggling to discover about half of the peers running when using JXTA.

From a user's point of view, there is barely no change, except that the Rendezvous terminology of JXTA is replaced by the *supernode* concept. A supernode is a necessary entry point for boot-strapping a peer willing to join the overlay. When connecting to a supernode, the MPD registers to the supernode and retrieves a list of peers that it will maintain in its internal cache. Thus, in this first implementation of our peer-to-peer infrastructure, it is a centralized topology where the peers first register to the supernode by giving its IP and some necessary communication ports. It is left to a future work to extend the single supernode to a distributed set of supernodes in order to improve scalability.

3.1.2 The Middleware

The P2P-MPI middleware part consists of the four processes noted MPD, FT, FD, and RS on introductory Figure 3.1. This is the core of P2P-MPI, which required most of the developments done in this thesis work. In the following section, we give a brief overview of the roles of these processes. Details and further discussions will follow in next sections.

The Message Passing Daemon (MPD) is the peer-to-peer module which acts as a peer-to-peer node in P2P-MPI peer-to-peer network. The MPD's roles are mainly:

- to maintain the peer membership to the overlay by joining on startup and by subsequently sending periodic alive signals to the supernode,
- to manage the local peer's neighborhood knowledge: each neighbor in the cache is periodically pinged to assess network latency to it,
- when an application requests a number of resources, it has to coordinate the discovery of peers, the reservation of resources and to organize the job launch.

The File Transfer Service (FT) is a simple service in charge of what is often called *file staging*. This task consists in transferring the executable code and input files from the submitter (the node requesting the parallel program execution) to the computing nodes when they need it. The FT service can perform the staging in two possible modes.

In the first mode, the full files and data are transfered to the computing nodes. The FT proceeds sequentially, that is it waits to complete the transfer of all execution files and input files to a computing node before it starts to transfer files to another node. This mode hence imposes a startup time which increases linearly with the number of nodes. We did not implement other methods such as broadcasting data along a tree because in an environment with frequent failures, the failure of any intermediate node causes a whole branch of the tree not to receive the files. In P2P-MPI, the fault detection service monitors nodes failures for running applications only. Since we cannot

rely on this service during file staging, it is more difficult to identify missing nodes in a tree-like broadcasting procedure than in a linear one. Implementing an improved startup mechanism for an advanced file transfer system is left as a future work.

In the second mode, only the addresses of data are transfered, which allows communication pipelining and overlapping, as illustrated on Figure 3.2.

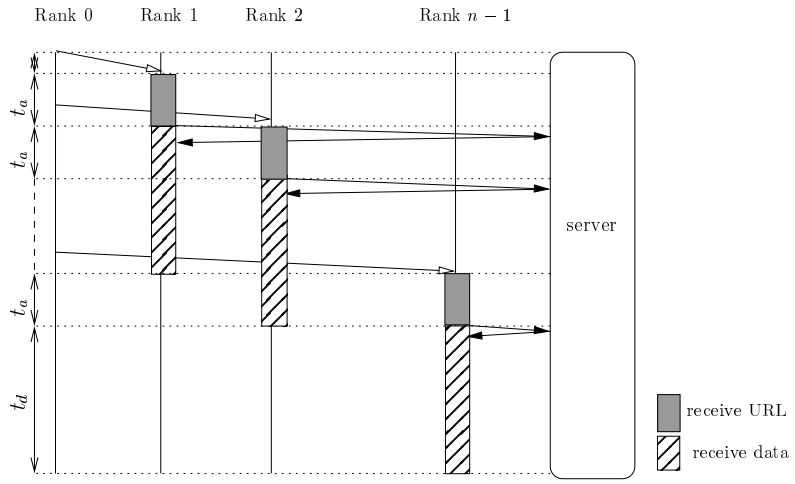


Figure 3.2: File staging using a web server.

The code and data and stored on a web server and the FT service sequentially sends the URL to the computing nodes. When the computing nodes receive the URL, they start to download the files from the indicated URL. As the communication time needed to transfer the URL is generally shorter that the time needed to download the real data, the communications fetching data follow a pipeline pattern, provided the web server(s) does not become a bottleneck. In the best case, almost all the time spent in downloading real data is overlapped by the time needed to transfer the URLs. This situation corresponds to the figure. If we consider t_a the time to send the address of the file to any computing node, and t_d the time for any node to download the real data from the web server, the file staging time t_s for n nodes is $nt_a + t_d \leq t_s \leq t_a + nt_d$. The upper bound occurs if the web server can serve only one connection at a time, hence making the data delivery sequential. In this worst case, the file staging time is close to the first mode behavior, whose cost is nt_d. The lower bound applies if the web server is able to serve all requests simultaneously. This is possible if URLs point to different web servers or if a specialized hardware with several network cards is used. In practise, as the aggregated throughput obtained with multiple TCP connections is higher than with one connection in most situations, we have $t_s \ll t_a + nt_d$.

The Fault Detection Service (FD) is the service invoked to monitor the resources in charge of an application execution. The communication library is notified by the FD service when nodes become unreachable during execution. The communication library can then take appropriate actions to react to failures. To fit in the peer-to-peer model of P2P-MPI, we implement the fault detection service as a fully distributed service, using a gossip-style fault detection. This service is an important actor in the fault tolerant capabilities of our middleware, and Chapter 4 is devoted to this aspect.

The Reservation Service (RS) is a resource broker. When a user requests a number of processes for a program execution, the local MPD computes a list of candidate peers and mandates the local RS to negotiate and reserve computation capabilities among this set of peers. The local RS contacts each of the remote RS on candidate peers, which may give all, part, or none of the requested capabilities. The RS role is thus to request resources, or conversely, it acts as a gate-keeper of the local resource. Each RS decides on the computing capability it can offer to other RS, based on the user configuration file. An example configuration file is shown in Section C.1 in appendix. This file describes the user policy, and may specify for instance a list of denied hosts or the maximum number of jobs running simultaneously. More details on the role of the RS service are given hereafter in the discovery and reservation protocol description (c.f Section 3.3.2).

3.1.3 The Communication Library

The communication library exposes an MPI-like API, following the MPJ specification. The status of the functions can be found in appendix B. Below is a summary of the role and issues in the development of the communication library. A complete discussion is in Chapter 5.

The implementation of the API provided by the communication library is original as compared to other projects in that it integrates a transparent fault-tolerance mechanism based on process replication. The communication library, in addition to the standard communication primitives, handles the coherence of process states with respect to the programming model semantics. The way fault-tolerance is integrated in the communication library is detailed in Chapter 4.

Concerning the implementation of the standard communication primitives, the communication library implements two *devices* called *single-port* device and *multi-port* device. A device is the set of structures and internal functions that handles the network communications over TCP. The two proposed devices correspond to two different strategies we have explored.

The single-port device has been first proposed with P2P-MPI. It encapsulates all messages in Java objects and uses only one TCP port for communications. All communications open a connection, send the message, and then close the connection. The idea

is that communications between different administrative domains may be restrained by firewall policies and that a limited port range may ease the software usage. This is in contradiction with the strategy followed by most MPI implementations that open a new socket as soon as a new communication is needed, or even open one connection between each pair of processes at startup time (e.g `lamboot` in LAM/MPI).

Of course, the drawback of this device is the performance. The performance suffers of the opening and closing connection cost for every message. However, this overhead is small for applications communicating mostly large size messages.

The multi-ports device improves performance of communications at the price of a larger range of used ports. The technical difference with the single-port device is that the Java `nio` class is used to make messages transit through `ByteBuffers`, and to simultaneously monitor multiple network connections. This implementation is mostly interesting in environments with high-performance network connections with low restrictions regarding the firewall policy.

3.2 Application Start-up Protocol

In this section, we give an overview on how modules in P2P-MPI interact among themselves when a user submit a job to execute on a P2P-MPI grid. The steps listed below are illustrated on Figure 3.3.

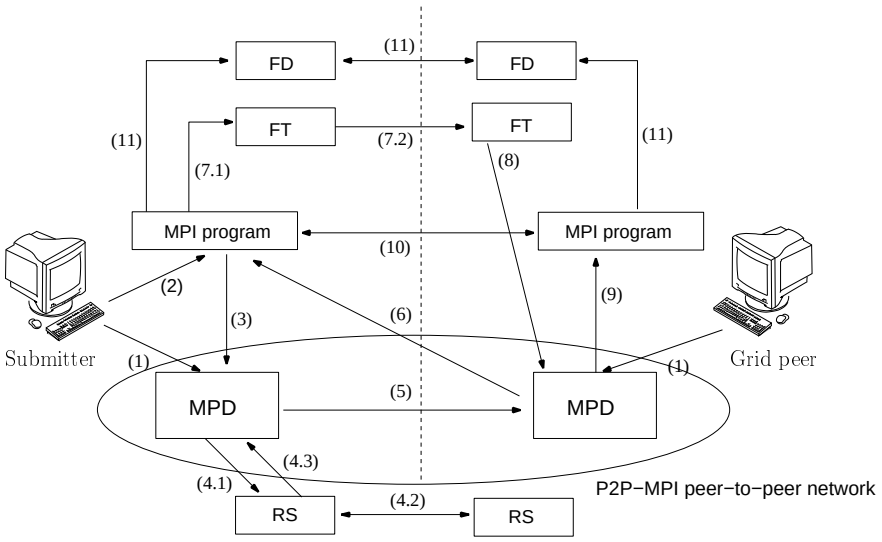


Figure 3.3: Steps taken to build an MPJ communicator mapped to several peers.

(1) **Booting up:** The user must first join the P2P-MPI platform by typing command `mpiboot` which starts the local background daemons MPD, FT, FD, and

RS. MPD acts as a peer in the peer-to-peer network, and makes the computer join the P2P-MPI peer-to-peer network.

(2) **Job submission:** The job is then submitted by invoking run command `p2pmpirun -n` n `-r` r `-a` *alloc* `prog`. The mandatory arguments are the n processes requested to run `prog` program. The other arguments are optional: r is the replication degree used to request some fault tolerance (explained in Chapter 4), and *alloc* tells the MPD which strategy must govern the allocation of the n processes on available resources (details about allocation strategy are in Section 3.4).

Then, it will start the process with rank 0 of the MPI application on local host. We call this process the *root process*.

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

(4) **Discovery and Reservation:** the local MPD looks into the list of its known nodes and then issues a reservation request via the local RS to reserve available nodes by giving the list of subsets of its known nodes. The local RS negotiates and reserves the remote RS and then returns the result to MPD (detailed in Section 3.3).

(5) **Registering:** After the reservation is done, the local MPD directly contacts the reserved nodes MPDs. It declares to the remote MPDs that the job will be executed by giving the application name, its MPI rank regarding the application to spawn, and the IP and port of the root process for the MPI application to contact it. The application will then be able to form its MPI communicator.

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

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

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

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

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

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

Note that all the steps listed above are transparent to the user. The peer just needs to be started once with `mpiboot`. Once it belongs to the peer group, it may request other peers participation or it can be solicited an unlimited number of times until it halts (`mpihalt`).

3.3 Discovery and Reservation

In a grid context, it is not realistic to maintain a static list of resources (such as the `machinefile` of most MPI implementations) and hence we rely on the discovery capabilities of the middleware. Subsequently to the run request `p2pmpirun`, P2P-MPI dynamically tries (during a limited time) to reserve a suitable set of resources able to host all processes involved.

In the previous section, we have enumerated the steps taken to start a parallel application. Among these, step (4) hides a complex problem. Choosing among the discovered resources, which are the most adequate for a specific execution is a difficult problem as several objectives may be followed. Let us list some considerations:

- First, we need co-allocation and hence resource should be available simultaneously. We have introduced the Reservation Service (RS) for that purpose.
- Second, the grid is a multi-user platform and the allocation must accommodate to the local policies of resources, not known in advance, like e.g, the number of processes that the owner of the resource accept to run simultaneously,
- Third, an MPI application generally benefits from locality of allocated resources since it minimizes the communication costs.

3.3.1 Entities involved and Notations

Each service maintains a complete or partial knowledge of the P2P network. The supernode maintains the registration of peers through a list called *host list*. Each list element is basically the host IP, its services ports, and a "last seen" timestamp.

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

Each RS, as a gatekeeper of the local resource, also manages the resource owner preferences. The owner preferences, expressed in the configuration file, may for instance allow or disallow such or such other peers. The preferences also concern the way the CPU is shared, through two settings:

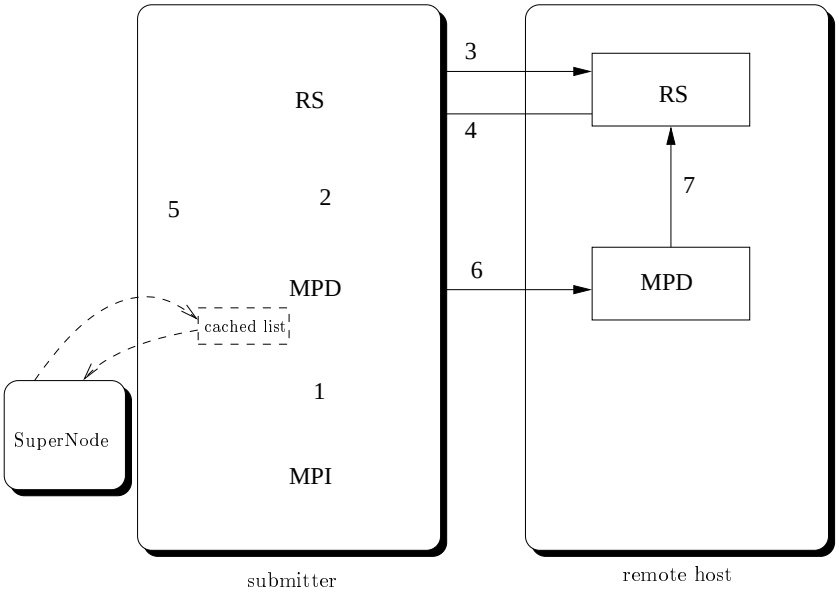


Figure 3.4: The job reservation procedure.

- the number J of different applications that a node can accept to run simultaneously.
- the number P of processes per MPI application that a node can accept to run.

For instance, J=2 and P=1 would allow two distinct users to run simultaneously one process each for their respective applications. J=1 and P=2 would allow to simultaneously run two processes of a single application (this setting is often used for dual-core CPUs).

3.3.2 Reservation Schema

We now describe step by step the reservation procedure, as depicted in Figure 3.4. It actually details steps (3) to (5) in Figure 3.3.

(1) **Requesting Peers:** Recall a user submits a job with `p2pmpirun -n` n `-r` r `-a` *alloc* `prog`. This starts the MPI application, which in the initialization phase (`MPI.Init`) assigns the local MPD the task of discovering and reserving the set of hosts able to executes $n \times r$ processes.

(2) **Booking:** First, the local MPD verifies if it knows enough (i.e, at least $n \times r$) nodes in its cached list. If not, it triggers a cached list update request to supernode to try to acquire recently registered peers. The list is then sorted by ascending latency values. The MPD asks the local RS to book a number of hosts, starting from the beginning of its cached list (hence starting with hosts having the

lowest network latencies). Actually, when possible, the request is an *overbooking* to anticipate unavailable hosts. In the current version $(n \times r) + (3 \log_2 (n \times r))$ hosts are requested.

(3) **RS-RS Brokering:** Local RS generates a *unique hash key*, we can see it as reservation ticket. Then, RS sends a reservation request message to others RS with this unique hash key.

(4) The RS receiving the reservation request message verifies whether it can accept this request by checking if the current number of applications being run does not exceed J. It also checks at this stage if the requester belongs to the denied IP list. If the request is acceptable, it replies back to the requester by sending an OK message with the value P. If not, it replies back to the requester with a NOK message.

(5) **RS-MPD Response:** The local RS gathers answers from remote RS to form the list *rlist* of reserved hosts. This list is then passed back to MPD. Nodes that have not responded before a given timeout are also marked as dead at this step. The MPD receives the *rlist*, and updates its cached list regarding peers marked dead.

(6) **Allocation**: Then, the MPD allocates the processes to all or a subset of the hosts in *rlist*. Because of overbooking, the number of reserved hosts is often larger than necessary: we call *slist* the selected subset chosen to map the application processes. It is the same as *rlist* except that it is limited to $n \times r$ hosts (what we need at most). Formally $slist = rlist[1, \ldots, \min(|rlist|, n \times r)]$. The implication is that all reservations for hosts in *rlist* but not in *slist* are cancelled since they will not be used. Once *slist* has been extracted, and before the MPI ranks distribution can take place, the MPD must decide whether the allocation is feasible. It is feasible if the two following conditions are met:

   (a) $|slist| \geq r$
   (b) $\sum_{i=0}^{|slist|} c_i \geq n \times r$, where $c_i = \min(P_i, n)$.

   The first constraint says we have at least r selected hosts to insure that no two replicas would have to reside on a same host. The second constraint is about the number of processes that can be hosted on the whole: we call c_i the capacity of host i, which is P except for marginal cases (we must not allocate more than n processes to a single host even if $P > n$ since two copies would be on that host). We therefore check that the sum of individual host capacities is large enough to execute all processes. Finally, MPD sends a request to start the MPI application. The request includes the rank and a unique hash key. The other MPDs are chosen accordingly to one of the selected allocation strategy (see Section 3.4).

(7) **Verify Reservation:** The remote MPD verifies that the unique hash key matches the one its RS holds for the reservation. If the key matches, then the rest of the job submission process follows at step (6) on Figure 3.3.

3.4 Host Allocation Strategies

There are today many multicore CPUs and we should favor the allocation of processes on all cores of a CPU if we strictly follow the locality principle. However it might be more important for the application to access more memory on the whole, which is in contradiction to the allocation strategy that chooses all cores on each resource as they share the same memory. We think the user, most of the time, knows these requirements and should advice the middleware of the application's specific needs.

In our context, an allocation strategy must meet two criteria.

(a) First, it must assign the $n \times r$ processes to the $|slist|$ reserved hosts in a sensible and understandable way regarding the user's concerns. An example of "bad" distribution would for example be one that allocates as many processes as possible on the last host of *slist*, that is the host with higher network latency.

(b) Second, in case some processes are replicated, the rank assigned to mapped processes must guarantee that no two copies of a process are on the same processor.

For the first criterion, we propose two simple strategies called *spread* and *concentrate*. Below are the algorithms for each strategy, in which we use the following notations: d is the number of distributed processes so far, and u_i the number of processes mapped onto host i.

Spread tends to map processes on hosts so as to maximize the total amount of available memory while maintaining locality as a secondary objective. The strategy is to assign the MPI processes to all selected hosts (the $|slist|$ closest hosts regarding latency) in a round-robin fashion.

Algorithm 1: Spread algorithm

```
d := 0
∀_i, u_i := 0
cont:= TRUE
while cont do
    i := 0
    while (i < |slist|) AND cont do
        if (u_i < c_i) then
            u_i := u_i + 1
            d := d + 1
        if (d = n × r) then
            cont := FALSE
            //all processes are allocated
        i := i + 1
```

Concentrate tends to maximize locality between processes by using as many cores as hosts offer. The strategy is to assign the maximum MPI processes to the capacity of

each host (c_i).

Algorithm 2: Concentrate algorithm

```
d := 0
∀_i, u_i := 0
cont := TRUE
while cont do
    i := 0
    while (i < |slist|) AND cont do
        u_i := min(c_i, (n × r) − d)
        d := d + u_i
        if (d = n × r) then
            cont := FALSE
            //all processes are allocated
        i := i + 1
```

Once either strategy has reserved enough processes place-holders, we must meet criterion (b) when numbering the processes, i.e, assigning MPI ranks to processes. The assignment algorithm host is straight-forward: we assign the MPI rank from rank 0 to $n-1$ according to u_i and continue along with host i in *slist*. If some $u_i = 0$, it means no process has been mapped to host i and we simply cancel the reservation. The algorithm is as follows:

Algorithm 3: Rank distribution algorithm

```
rank := 0
for host i in slist do
    if u_i = 0 then
        cancel reservation on host i
    l := 0
    while l < u_i do
        assign rank rank to host i
        rank := rank + 1
        l := l + 1
        if rank ≥ n then
            rank := 0
```

3.5 Experiments with Co-allocation

Objectives

The main objective is to assess if the allocation strategies behave really as predicted at the scale of applications composed of hundreds of processes. A secondary objective is to observe the impact of both strategies on parallel program executions. This last point would obviously deserve a larger study, but these preliminary tests sketch important tendencies.

Experiment Setup

Environment type	Grid5000 – clusters as detailed below.				
Site	Cluster name	CPU	#Nodes	#CPUs	#Cores
Nancy	grelon	Intel Xeon 5110	60	120	240
Lyon	capricorn	AMD Opteron 246	50	100	100
Rennes	paravent	AMD Opteron 246	90	180	180
Bordeaux	bordereau	AMD Opteron 2218	60	120	240
Grenoble	idpot	Intel Xeon IA32	8	16	16
Grenoble	idcalc	Intel Itanium 2	12	24	48
Sophia-Antipolis	azur	AMD Opteron 246	32	64	64
Sophia-Antipolis	sol	AMD Opteron 2218	38	76	152
Operating System	Linux 2.6.18 or close				
Software	jdk1.6.0_04, JXTA-J2SE 2.3, p2pmpi-0.28.0				

Table 3.1: Characteristics of available computing resources at the different sites

The experiment in this section uses the experimental grid testbed Grid5000. The resources in our experiment are taken from six sites: Nancy, Lyon, Rennes, Bordeaux, Grenoble, and Sophia-Antipolis. The job submitter is located at a node in Nancy's site. Available resources are summarized in table 3.1. The distant sites are sorted by round-trip time (RTT) to local site Nancy. RTT are measured by an ICMP echo (ping) between frontal hosts at each site and are reported in table 3.2. We can see that latencies between Nancy and distant sites are very close for most of them. The bandwidth between sites is 10Gbps everywhere except the link to Bordeaux which is at 1Gbps.

For all nodes, the parameter that allows the number of executing process on each application in the configuration is set to the number of cores in the host's CPU.

3.5.1 Co-allocation Experiments

In this experiment, we run a program whose each process simply echoes the name of the host it runs on. Through this experiment, we observe where processes are mapped depending on the chosen strategy and the number of processes requested by counting

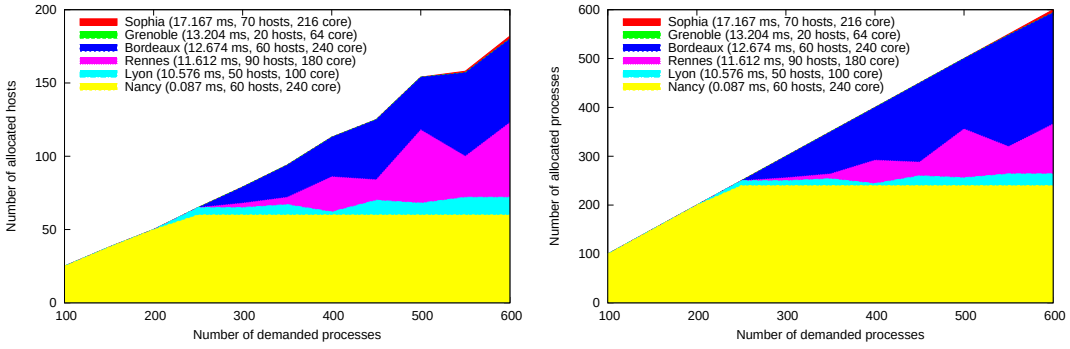
Site	RTT(ms)
Lyon	10.5
Rennes	11.6
Bordeaux	12.6
Grenoble	13.2
Sophia-Antipolis	17.1

Table 3.2: The round-trip time by ping between Nancy and other sites

hosts and cores allocated at each site.

For the *concentrate* strategy, we consider the closer the processes are from Nancy, the better are the results. For the *spread* strategy, a good allocation should map only one process per host as much as possible, and hosts selected should be the closest from Nancy. The effectiveness of the strategies essentially depends on the accuracy of the latency measurement, which may differ from the RTT given by an ICMP echo command (ping) as explained in Section 3.3.1. The latency we measure with P2P-MPI must not necessarily be very close to the ICMP RTT, but should preserve the ranking between hosts relatively to RTT.

Figures 3.5 and 3.6 plot the repartition of processes throughout the sites for the two strategies. The legends in top-left corners give the RTT to Nancy site and the overall number of hosts and cores available at each site. The experiment consists in running the *hostname* program, requesting from 100 to 600 processes by steps of 50.

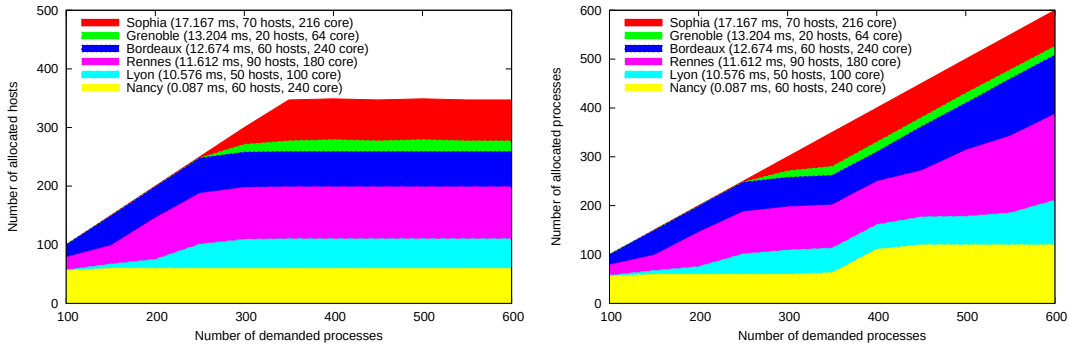


(a) Allocated hosts

(b) Allocated cores

Figure 3.5: Hosts and cores allocated in concentrate allocation method

For *concentrate*, in Figure 3.5, the processes are allocated on the 60 hosts available at Nancy only, up to 200 processes. Next, when the capacity of 240 cores at Nancy



(a) Allocated hosts

(b) Allocated cores

Figure 3.6: Hosts and cores allocated in spread allocation method

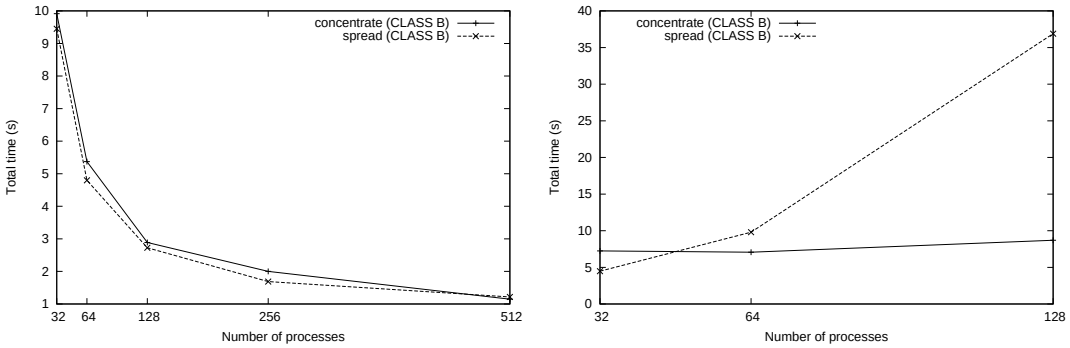
is exceeded by the request, further hosts are first allocated at Lyon (5 for -n 250), as expected with respect to the RTT ranking. Subsequent requests (from -n 300) reveal that hosts from Lyon, Rennes and Bordeaux fiercely compete for the latency ranking. We observe that the latency ranking for these hosts is interleaved with respect to sites. This is easily explained by the fact that the latencies to Nancy for the three sites are within 0.6ms (RTT 1.1ms), while the latency measurements made by peers are sensible to CPU and TCP load variations. This mapping thus seems adapted to applications involving many communications because of the nearness of processes.

With *spread*, in Figure 3.6, hosts are chosen from the four closest sites up to 250 processes, but contrarily to *concentrate* more hosts are allocated in each site. From 300 processes, the strategy leads to take hosts from all sites to keep the load on each peer to only one process. We can clearly see on Figure 3.6(b), the round-robin allocation of processes once the host list is exhausted: the number of cores allocated at Nancy makes a stair at 400 processes since there are not enough hosts (350) to map one process per host and the closest peers are first chosen to host a second process as they have extra available cores. On the whole, we observe that all peers have been discovered and the strategy tends to use them all. So, this is a good strategy to use for application demanding much memory, as only one application process will be mapped per host provided there are enough hosts.

3.5.2 Application Performance

To observe the effectiveness of each strategy on applications, we have chosen to test two programs with opposite characteristics from the NAS benchmarks (NPB3.2), IS (Integer Sorting) and EP (Embarrassingly Parallel). IS involves many communications and EP (Embarrassingly Parallel) does independent computations with a final collective communication.

As a concrete example of allocation strategy impact, we run the benchmark EP from 32 to 512 processes. As mentioned, EP only makes four final collective communication (`MPI.Allreduce` of one double) so that the computing to communication ratio is very high. The graph on the left of Figure 3.7(a) shows that EP using 32 to 256 processes is slightly faster when allocation strategy *spread* than with *concentrate*. This is probably due to the intensive memory accesses that may represent a bottleneck with *concentrate*, not compensated by locality in the collective communication. With 512 processes, the problem size per process becomes smaller and the overheads related to memory and communications seem to reach an equilibrium at this point.



(a) Execution time on EP benchmark.

(b) Execution time on IS benchmark.

Figure 3.7: Execution time for EP and IS depending on allocation strategies.

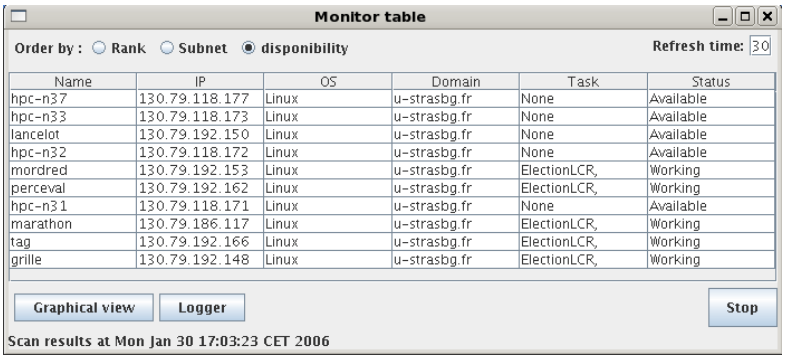
The performance curves for IS, in Figure 3.7(b), are due to the low computations to communications ratio. With 32 processes, *spread* leads to better performances than *concentrate*: with *spread* all processes are in the same cluster so that communications pay a low latency while there is no overhead due to concurrent memory accesses. This appears to be the case with *concentrate*. Using 64 processes with *spread* means that four processes are allocated outside the local cluster and the communication overhead leads to a slowdown. Keeping the processes inside the cluster with *concentrate* gives a roughly constant execution time. Figures for 128 processes and above show the same phenomena.

3.6 P2P-MPI Graphical Monitoring Tool

P2P-MPI contains a visualization tool which provides a global snapshot of the peer-to-peer network. It provides a graphical GUI displaying the network, either under the form of a table listing computers name and IP addresses or as a graphical view of peers, with a layout organized around domain names.

This tool comes in addition to the query command `mpihost`, which lists the peers known by the local MPD only. Thus, the information returned by `mpihost` is incomplete

because more peers may be running and no information is given regarding to what peers are doing. Figure 3.8 illustrates the monitoring table. In this snapshot, we have a partial



Name	IP	OS	Domain	Task	Status
hpc-n37	130.79.118.177	Linux	u-strasbg.fr	None	Available
hpc-n33	130.79.118.173	Linux	u-strasbg.fr	None	Available
lancelot	130.79.192.150	Linux	u-strasbg.fr	None	Available
hpc-n32	130.79.118.172	Linux	u-strasbg.fr	None	Available
mordred	130.79.192.153	Linux	u-strasbg.fr	ElectionLCR,	Working
perceval	130.79.192.162	Linux	u-strasbg.fr	ElectionLCR,	Working
hpc-n31	130.79.118.171	Linux	u-strasbg.fr	None	Available
marathon	130.79.186.117	Linux	u-strasbg.fr	ElectionLCR,	Working
tag	130.79.192.166	Linux	u-strasbg.fr	ElectionLCR,	Working
grille	130.79.192.148	Linux	u-strasbg.fr	ElectionLCR,	Working

Figure 3.8: The monitor table

view of the peers running (IP addresses, operating system type) as well as what they are doing. Here, five of them are executing an *ElectionLCR* application. This is often handy for the user to see how its processes were mapped to the network. A complementary

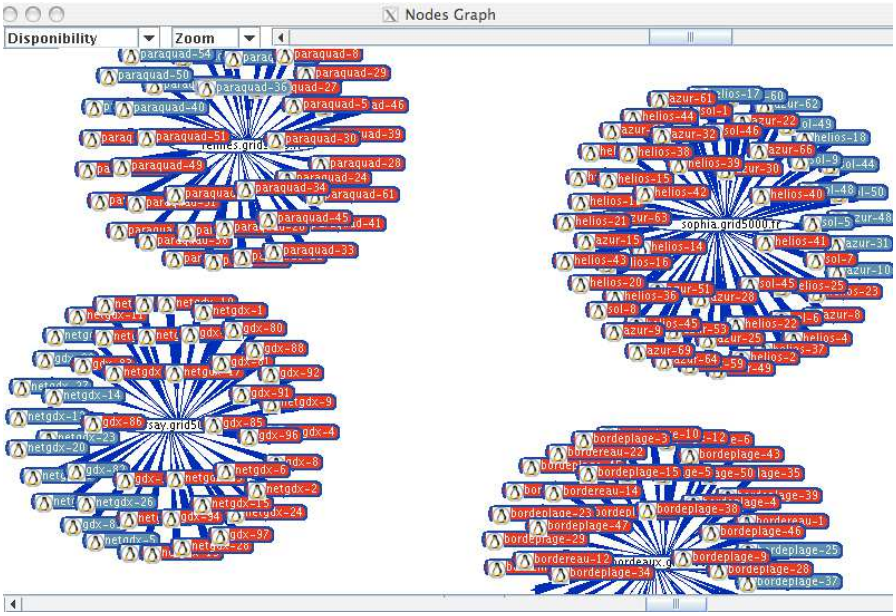


Figure 3.9: Graphical view: screenshot for a couple hundreds of peers on Grid5000.

view is the graphical layout illustrated by Figure 3.9. This example screenshot shows a couple hundreds of P2P-MPI peers running on four sites of Grid5000.

The graphical view gives a zoomable view of peers, represented as clusters centered

on their domain name. Clicking on a specific peer makes a popup window appear with the main characteristics of the resource when available: CPU type, CPU speed, memory available, etc.

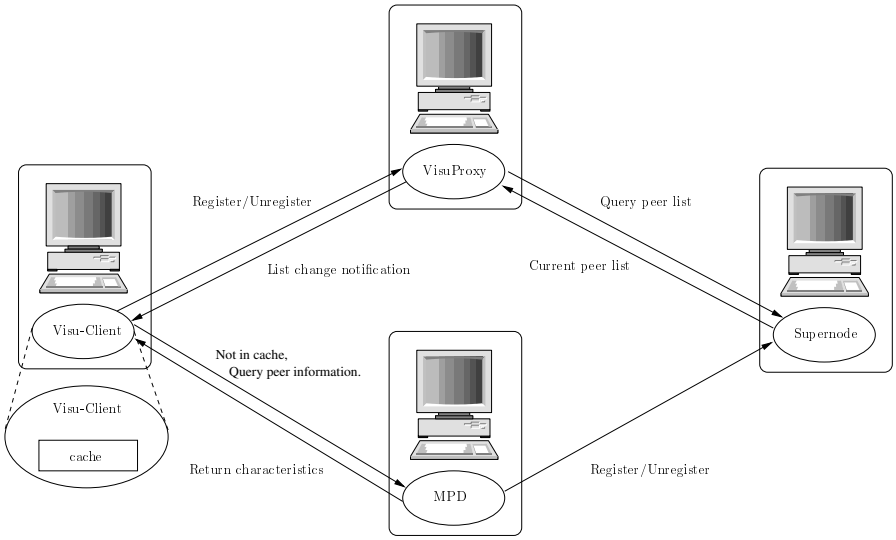


Figure 3.10: Overview of the visualization service organization

The visualization tool design has two requirements:

- the visualization process should be independent of the other processes: a client should be able to visualize the network without starting a peer.
- the visualization process should be as little intrusive as possible, and no client should be able to overflow the network with visualization queries.

To this end, we provide an extra service called *VisuProxy*. The VisuProxy can be seen as an intermediate layer between the visualization clients and the supernode. The VisuProxy service periodically queries the Supernode about the peers currently present. Once a VisuProxy is known, visualization clients may register to the VisuProxy. Registered clients are then notified when the peer list changes. The VisuProxy does not hold all information about peers. We do not want a centralized service that would maintain information about peers dynamic state. Instead, the visualization client has the charge to contact directly the MPDs of peers announced by the VisuProxy, to get information about their hardware and software characteristics as well as their state. Each visualization client maintains a cache on its disk, containing information returned by previously queried MPDs. If the announced IP addresses are already known, information from the cache is used. Otherwise, a query is issued to the corresponding MPDs. If the user explicitly asks to refresh the information, queries to the remote MPDs are forced.

3.7 Conclusion

We have described in this chapter the P2P-MPI middleware. We have explained how our initial design and implementation choices have evolved to face the problems targeted by P2P-MPI. Recall that our goal is to address the deployment of large-scale parallel message-passing programs. In the present case, we have to deal with applications involving hundreds of processes scattered on computers over a wide geographic area. Since the beginning of the project, we have proposed a P2P basis to organize resources in a Grid. We put forward the autonomy of peers, which enables an easy software installation of individual resources and the absence of a single point of failure since there is no central directory for resources.

We have also put forward the benefit for applications to cooperate with the middleware. An example is the failure detection service that will be examined in the next chapter.

Another benefit can be an efficient allocation of resources by the middleware with respect to the application's needs. During, this work, we have modified the middleware to improve the allocation resources. The middleware now accounts for network locality of peers. This has allowed us to devise two allocation strategies. We propose the simple and understandable paradigms *spread*, which maps only one process on the closest peers, and *concentrate*, which uses computing resources of closest peers as much as possible. Users can easily decide, depending on the execution environment and on their application which strategy is best suited. On one hand, *spread* involves more network communications but let each computer memory accessed by only one process. On the other hand, *concentrate* increases locality of processes but may lead to memory contention or exhaustion. The experiment presented contribute to show that such strategies can be implemented effectively to tackle the goal of allocating up to 600 processes. Further, the allocation strategy effects on program executions have also been verified on two NAS benchmarks. As a future work, we should focus on improving the accuracy of our latency measurement so that it becomes closer to ICMP values and less sensitive to external load. Also, we should work at the design of mixed strategies, or more complex ones which still do not require the user to be knowledgeable about the platform characteristics. Last, a broad study may be carried out to better understand the impacts of such allocation strategies on a wider range of applications.

Chapter 4

Fault Management

As stated in the introduction, the robustness of an execution is of tremendous importance for MPI application since a single faulty process is very likely to make the whole application fail. Much research work has been done in the area of fault-tolerance for MPI, and we have reviewed a number of the proposals in Section 2.4.2. These proposals are all based on check-pointing. We argue that this approach does not fit easily in our peer-to-peer paradigm because it assumes a reliable server where checkpoints can be stored. This is the reason why we propose for P2P-MPI, a solution based on *process replication*.

This chapter presents how fault management is handled in P2P-MPI. This topic covers two aspects.

The first one is related to the replication itself, and is covered by Sections 4.1 to 4.5. Section 4.1 introduces the replication scheme in P2P-MPI. Section 4.2 recalls the issues related to this replication strategy and what has been stated in the literature, in particular for the atomic broadcast problem. Our contribution is presented in Section 4.3, where we describe our protocol for replication. We then show in 4.4 that our protocol meets the requirements that have been stated in the literature regarding atomic broadcast. Finally, we present in Section 4.5 a quantitative study about the failure probability when using replication or not.

The second aspect deals with fault detection. For an application to be able to recover using a copy of a failed process, it must first be efficiently informed about the failure. We present in Sections 4.6 to 4.7 how fault detection is implemented in our framework. Finally, we present in Section 4.8 experimental results regarding two aspects of our approach. The first experiment measures how the real system behaves as compared to prediction in terms of fault detection time. The second experiment shows the overhead induced by replication on some test cases.

4.1 Logical processes and replicas

P2P-MPI implements a replication mechanism to increase the robustness of an execution. This replication management is absolutely transparent for the programmer. When specifying a desired number of processes, the user can request the system to run for each process an arbitrary number of copies called *replicas*. An exception is made for the process running on the submitter host, numbered 0 by convention, which is not replicated because we assume a failure on the submitter host is critical. In practice, it is shorter to request the same number of replicas per process, and we call this constant the *replication degree*. Currently, we do not take into account host reliability when mapping processes during allocation. (Criteria for resource allocation are discussed in Section 3.4). Therefore, the interest of specifying how many replicas should be chosen for one or several specific processes, is pointless.

In the following, we name a "usual" MPI process a *logical process*, noted P_i when it has rank i in the application. A logical process P_i is implemented by one or several replicas, noted $P_i^0, \ldots, P_i^n$.

Figure 4.1 shows the group of replicas in a logical process P_1 with a replication degree of three.

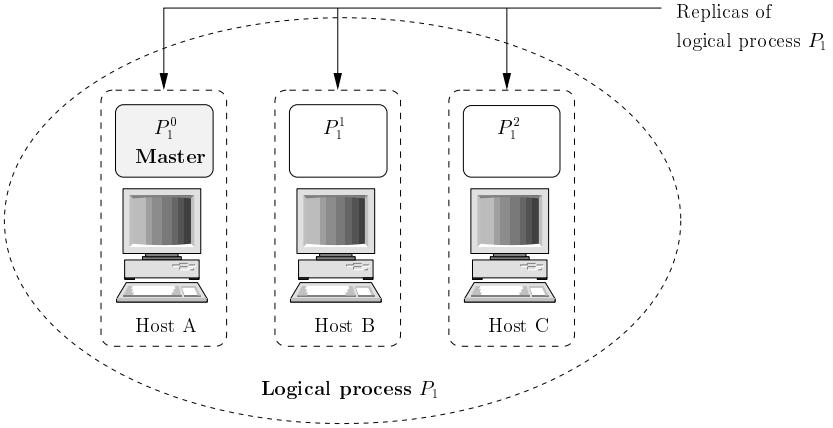


Figure 4.1: The logical process P_1 with a replication degree of three.

The replicas are run in parallel on different hosts since the goal is to allow the continuation of the execution even if some hosts fail.

The replication scheme we introduce should not break the application global coherence. In order to insure the coherence, we must keep the communications coherent with the semantics of the original MPI program. The following section presents a protocol, called *coordination protocol*, whose aim is to insure such a coherence.

The coordination protocol relies on two notions we introduced specifically to maintain coherence. First, in each logical process, one replica is assigned a special role. This replica is called *master*. If this process fails, one replica of the group will be elected as a new master to replace it, and it will update its state to be in the same state as the master before its failure.

Second, to be able to return or get to a certain state, replicas need to store some information about messages sent or received. We have added extra data structures in each process: these are the tables presented in Figure 4.2. Their roles will be explained with the protocol presented hereafter.

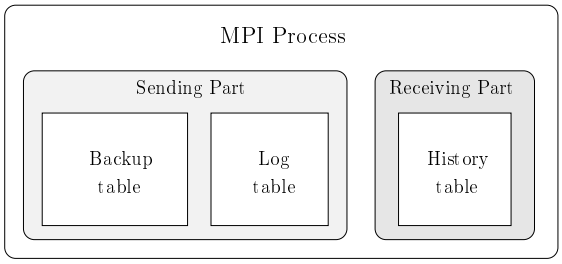


Figure 4.2: Extra data structures used in a process for replication.

4.2 Related Issues in the Literature

As discussed in the paragraph devoted to replication techniques in the literature (in Section 2.4.2), such protocols for replication have been proposed. They fall into two broad classes. With *Passive replication*, senders send messages to only one process (the primary) in the group of receivers which in turns, retransmits the message to replicas of the group. The other approach is *active replication*, in which senders send their messages to all replicas of the destination group. Our protocol follows the latter strategy except that specific agreement protocols are added on both sender and receiver sides. The conditions for such group communication to work properly have been well studied in the literature. We review below what are the requirements stated in the literature. In the next section (Section 4.3), we explain our protocol, and finally we show how our system complies to these requirements (Section 4.4).

4.2.1 Properties of Atomic Broadcast

It is well known that active replication requires *atomic broadcast* (or *total order broadcast*) to insure the coherence of the system. The survey article [67] gives a detailed description of a number of research work addressing the atomic broadcast issue. The

specification of the atomic broadcast has been defined formally [68] using the two primitives *broadcast(m)* and *deliver(m)*[1]. We assume that every message m can be uniquely identified, and carries the identity of its sender, denoted by $sender(m)$. A process that suffers no failure is usually termed *correct process*. The atomic broadcast is defined by the following properties:

Validity If a correct process broadcasts a message m, then it eventually delivers m.

Agreement If a correct process delivers a message m, then all correct processes eventually deliver m.

Integrity For any message m, every correct process delivers m at most once, and only if m was previously broadcast by *sender(m)*.

Total order If process p and q both deliver messages m and m', then p delivers m before m', if and only if q delivers m before m'.

4.2.2 Assumptions

It is also important to qualify our system regarding the nature of the distributed system addressed, in terms of type of failure considered, synchrony of the system and network links characteristics. Let us list our assumptions for our framework:

- We only consider **fail-stop failures** (also termed *crash* failures). It means that a failed process stops performing any activity including sending, transmitting or receiving any message. This includes the three following situations: a) the process itself crashes (e.g. the program aborts on a DivideByZero error), b) the host executing the process crashes (e.g. the computer is shut off), or c) the fault-detection monitoring the process crashes and hence no more notifications of aliveness are reported to other processes.

  This excludes transient failures as well as byzantine failures.

- We consider a **partially synchronous** system:
  - the clock drift remains the same, or the differences in the drifts are negligible, for all hosts during an application execution,
  - there are no global clock.
  - communication deliver messages in a finite time.

- We consider the network links to be reliable: there are 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.

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

4.3 Replicas coordination protocol

A first requirement, as stated in the previous section, is to be able to uniquely identify messages. To that end, we use unique identifiers for messages and we detail hereafter how they are implemented.

4.3.1 Message Identifier (MID)

The communication library computes for each message a unique identifier *mid*. It is assumed that any send instruction has a matching receive instruction. The *mid* is built only from information local to the process. It has the following form:

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

where *cid* is the identifier of the communicator[2], *src* and *dest* are the MPI rank of the sender and receiver processes respectively, *tag* is a tag number of the message and *count* is the number of the calling `MPI.Send` or `MPI.Recv` for a message which has the same *midkey*.

For example, in `COMM_WORLD` a process of rank 0 sends two messages with the same tag ($tag = 1$) to a process of rank 2. The communication library constructs the identifier of a first message with *cid=0, src=0, dest=2, tag=1* and $count = 0$. Assume that this is the first time that `MPI.Send/MPI.Recv` is called with $midkey = (0, 2, 1)$. Thus, in `MPI.Send`, the identifier of the first message is $(0, (0, 2, 1), 0)$ and $(0, (0, 2, 1), 1)$ for the second message. Symmetrically in the receiver, the first `MPI.Recv` call will wait for the message with the identifier $(0, (0, 2, 1), 0)$ and $(0, (0, 2, 1), 1)$ for the second `MPI.Recv` call.

Thus, the MID has two properties: it is a unique identifier for messages, and it reflects the order in which messages are sent and received. As we will see in Chapter 5, it may be useful in some asynchronous communication implementation. In the example, the messages could be received in any order in the receive queue, but the extraction from the queue to the user program would follow the MID order. Hence, we preserve the message order according to the MPI standard.

Note also that the MID computed on the sender side, is embedded in the header of the message sent. Indeed, MPI specifies some receive constructs allowing a non-deterministic order of reception. As discussed in Section 4.3.4, we must be able to check for already received MID via the history table to prevent incoherences.

4.3.2 Sending message agreement protocol

On the sender side, we limit the number of sent messages by introducing the following agreement protocol. In each logical process, one replica is elected as master of the group

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

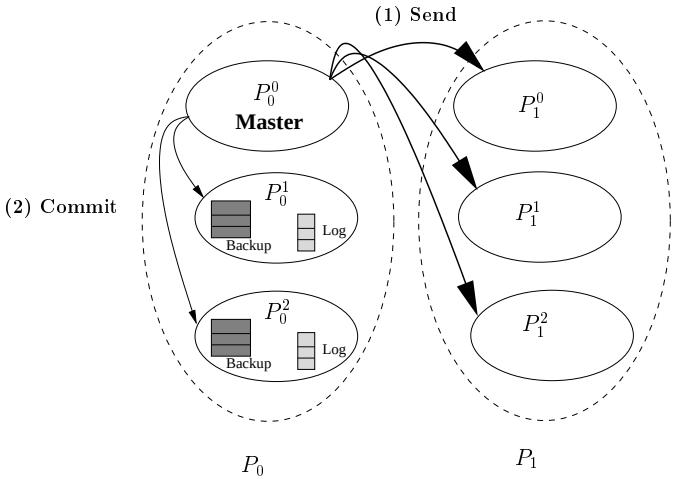


Figure 4.3: A message sent from logical process P_0 to P_1.

for sending. The other processes do not send the message over the network, but store it in their memory. Figure 4.3 illustrates a send instruction from P_0 to P_1 where replica P_0^0 is assigned the master's role. When a replica reaches a send instruction, two cases arise depending on the replica's status:

- if it is the master, it sends the message to all processes in the destination logical process. Once the message is sent, it notifies the other replicas in its own logical process to indicate that the message has been correctly transmitted. We say the master *commits* its send. The commit is done by sending the message's MID. The MIDs are stored into the *log tables* of each replica.

- if the replica is not the master, it first looks up its log table to see if the message has already been sent by the master. If it has already been sent, the replica just goes on with subsequent instructions. If not, the message to be sent is stored into the *backup table* and the execution continues. (Execution stops only in a waiting state on a receive instruction.) When a replica receives a commit, it writes the message identifier in its log and if the message has been stored, it removes it from the backup table.

The overview of the sending message agreement protocol is given by Algorithm 4. The algorithm is divided into two parts. The first one is the algorithm for `MPI.Send`. The second part only applies for a non-master replica, and is the action sequence to take when the replica receives a commit message.

Algorithm 4: Sending message agreement protocol on process P.

```
//(1). When MPI.Send is executed.
if P is master of logical process then
    //Master process
    Send message M to all replicas of destination process
    Send commit message to all replicas of its logical process
else
    //Non-master process
    if mid is in log table then
        //Message M already transmitted successful by its master
        do nothing
    else
        //Status of sending message M is unknown
        put M in its backup table

//(2). When a replica receives a commit message.
Receive a commit for message identified by mid
put mid in its log table
if M with mid is in backup table then
    //P already invoked MPI.Send and stored message in its backup table
    remove M from its backup table
else
    //P did not reach MPI.Send yet
    do nothing
```

4.3.3 Reception message agreement protocol

We have stated in the sending message agreement, that the master sends a message to all replicas of the receiving logical process. As these multiple send operations cannot be made atomic, a failure occurring at the master when sending a message may lead to an incoherent state regarding the replicas on the receiving side. After the failure, some processes may have got the message while some others may not have.

When the fault detection service detects a node failure, the fault recovery method is called (see Section 4.3.5). If the master of the sending side has failed, a new master is elected among its replicas. As this new master did not receive a commit message to signal the multiple send completion, it starts over the multiple send operation. Thus, some processes on the receiving side might have received the message from the master before it failed, and once again from the new master after the failure. To avoid this, a receiving process uses its *history table*, which stores MIDs of received messages. So, before actually receiving the message, the communication library on the receiver side verifies that the MID is not yet in the table. Otherwise, it simply discards the message. Algorithm 5 shows the pseudo-code corresponding to this protocol.

Algorithm 5: Reception message agreement protocol on process P.

```
//(1). When the communication library receives a message
Receive a message with MID = mid
if mid is in history table then
    //This message is already handled
    ignore this message
else
    //First time to receive this message
    wait MPI.Recv to handle message

//(2).When MPI.Recv is executed
read message M from queue and copies it to user buffer
put mid in history table
```

4.3.4 Non-deterministic Situations

Let us now examine what are the implications of replication on the *coherence* of an application. We say an application using replication is coherent if all its master processes produce the same outputs as the same application without replication.

The protocols specified above can lead to situations where real processes have different states, and thus may produce different outputs. The origin of a different state is a non-deterministic operation. We distinguish two types of non-deterministic operations during the execution. The first type is related to instructions executed internally in the process. The second type is related to the communication of values.

Internal Cause

The general case for such situation is a logical process :

(1) assigns a variable a different value on the master and on a replica,

(2) then the master starts to send its value to another logical process,

(3) then the master fails before it can commit its send on the replica.

In this scenario, the replica of the master will restart the send with its own value. The receivers that did receive the first value will discard the second message, while those that did not receive the message will accept the second (different) value.

To clarify this, the pseudo code listed in Algorithm 6 exemplifies the situation with the most evident source of non-determinism: here, the variable is assigned a random value.

Algorithm 6: A sample code with random operations.

```
if (rank == 0) then
    R = Random()
    Send R to rank 1
    Display R
else
    Recv R from rank 0
    Display R
```

Assume this program is executed with two processes and replication degree two.

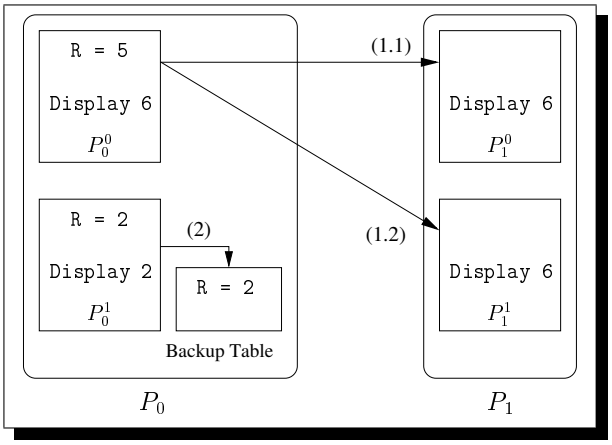


Figure 4.4: Scenario for Algorithm 6 with two processes and replication degree two.

No Fault Scenario Let us first examine the scenario in which no fault occurs during the execution. This is illustrated by Figure 4.4. Logical process P_0 (made of two real processes P_0^0, the master, and P_0^1 its replica) first issues a call to random(). Variable R is assigned the random value 5 on P_0^0 and 2 on P_0^1. In step (1.1) and (1.2), P_0^0 sends R to all processes of rank 1, P_1^0 and P_1^1. Meanwhile, P_0^1 has reached the send instruction, and in the absence of commit from the master, saves the message into its backup table in step (2). All processes are then instructed to output the value of R. The display outputs of non-master replicas are always discarded[3]. Hence, all processes display the same value, which is coherent with the MPI semantics.

Fault Scenario If a fault occurs, it may happen during the send operation. Figure 4.5 shows a fault occurring at the weakest point in the protocol. P_0^0 had nearly finished to send R to P_1. However, it could send R to only P_1^0 in step (1.1) and failed before

[3]Actually, outputs of all remote processes are routed through a `StreamGobbler` to the display device of the submitter. During this redirection, display from non-master replicas are discarded.

sending R to P_1^1. As in the previous scenario, P_0^1 had already stored its message in its backup table in step (2). Figure 4.5(b) shows that P_0^1 has been notified of P_0^0's failure. Hence, it becomes the master for P_0. It verifies its backup table and retransmits messages P_0^0 did not commit[4]. On the receiving side, P_1^0 which already had a message from P_0^0 discards the retransmitted message (with the same MID). On the contrary, P_1^0 accepts the message. Finally, the new master P_0^1 and P_1^0 output different values, leading to an incoherent state. The weakness of the protocol lies in the impossibility to make the send operation atomic.

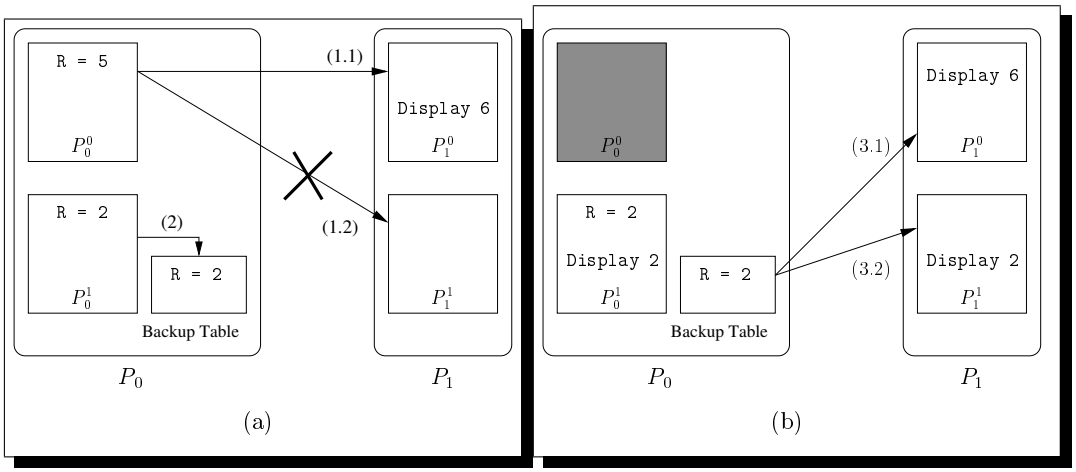


Figure 4.5: MPI process schema in algorithm 6, when there is fault during the execution.

We could imagine that the above situation could be solved by a consensus on the receiving side: if not all of the processes receive a message then all processes ignore the message. However, this does not work in the situation the messages have really been sent but the failure occurs before the commit (cf. footnote 4).

To solve this problem on random numbers, we introduce a new method `MPI.Random` which guarantees that all the processes in the same logical process generate the same random values. Inside the `MPI.Random`, we use the *jobID* (each executing MPI execution has its own jobID) and MPI rank number to generate a seed number as an input to the `Random` class in Java. Nonetheless, each execution generates different random numbers because of the different jobIDs and because each rank has a different seed number.

Non-deterministic Communication Case

As explained in Section 4.3.1, we can compute a unique message identifier as a function of the source, destination, communicator, tag and sequence number of the message.

[4]Note also that the failure could have intervened once the send is completed but before the commit is done.

There is one exception to it. MPI specifies the particular constants `MPI_ANY_SOURCE` and `MPI_ANY_TAG`, which can be used in the receive call as source and tag values respectively. In that case, the receiver cannot compute a unique identifier for all messages. Without extra-information, the receiver could face an undecidable problem after a failure, whether to accept or not a retransmitted message.

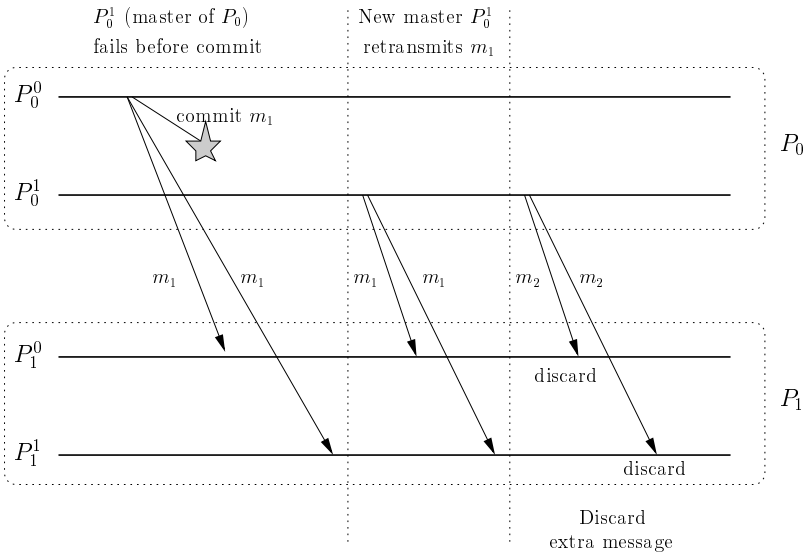


Figure 4.6: Replication problem on `MPI_ANY_SOURCE` and `MPI_ANY_TAG`.

Figure 4.6 illustrates the problematic situation. Suppose logical process P_0 has to send two messages m_1 and m_2. These messages are received on P_1 specifying `MPI_ANY_SOURCE`. A failure occurs after P_0 has really sent m_1 but before it committed its send. When the replica becomes the master, it retransmits m_1 and then sends m_2.

The figure presents the situation where P_1 receives m_1 again. Indeed, the receiver is unable to determine, based on the MID computed from the receive instruction arguments, if it is the same message. If these m_1 messages were accepted (and then used by the user program), the following extra message m_2 would be discarded. This execution would be incoherent with the duplication of messages m_1.

To solve this problem, P2P-MPI uses its history table. In the situation above, though m_1 are taken in the receive queue, the m_1 messages' headers are then examined. The receivers see that such MIDs already exist in the history table and discard the messages, as shown in Figure 4.7. Further messages m_2 are accepted.

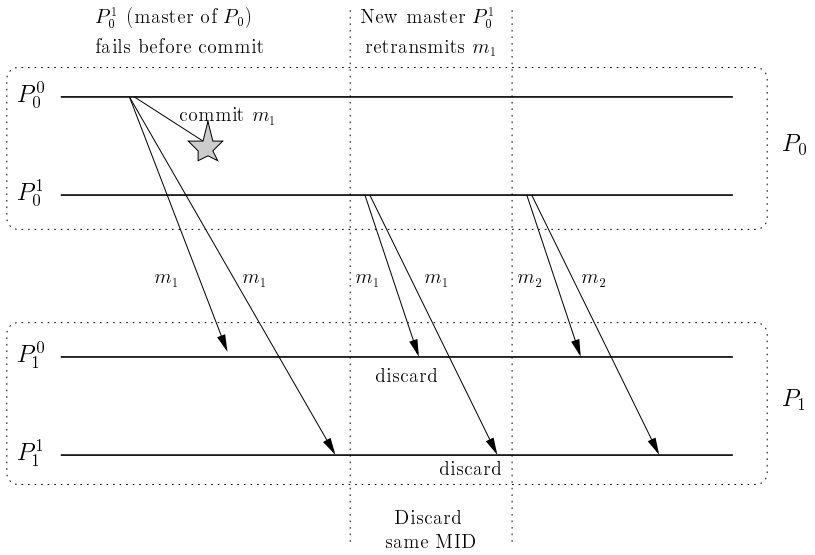


Figure 4.7: Replication problem solved on `MPI_ANY_SOURCE` and `MPI_ANY_TAG`.

4.3.5 Fault Recovery protocol

It remains now to specify how a failure is to be recovered. We must distinguish if the failure crashes a master or a non-master process. Algorithm 7 summarizes the actions to be taken at any real process P upon failure notification.

Algorithm 7: Fault recovery protocol.

```
//When a failure for process D is notified by FD service.
Mark D as dead
if D is the master of my logical process then
    elect new master
    if P is a new master then
        retransmit + commit all messages in backup table
        commit messages to its replicas
```

If the master of a logical process fails, alive replicas will elect a new master. The election requires no communication between replicas. Since the communicator contains all real processes ranks, they simply choose the process with the lowest rank in the group as the new master. Then, the new master checks for messages in its backup table. If the backup table contains messages, it means the previous master failed before it could complete the sending of these messages (completion involves to commit the corresponding MIDs at the replicas). The new master takes the charge to retransmit all the messages present in its backup table.

In case a non-master process fails the execution continues without any interruption. Replicas in the same logical process mark this process as dead and the master of the logical process will not send commit message to this process anymore. Meanwhile, replicas in all other logical processes also mark this process as dead and will stop sending MPI messages to this process.

4.4 Correctness of the protocol

4.4.1 Atomic broadcast compliance

As stated in Section 4.2, the active replication technique requires the atomic broadcast to satisfy four properties. We now explain that our replica coordination protocol matches the atomic broadcast requirements:

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

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

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

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

4.4.2 Handling of Failure Situations inside Atomic Broadcast

Let us illustrate with the following example the possible points of failures inside an atomic broadcast. We consider a process P_1 implemented by three replicas, and P_2 being two replicas. Figure 4.8 shows the steps taken by P_1 when it invokes `MPI.Send` to send a message to P_2. This is the only possible interleaving of messages since the messages are synchronous and the sending order to replicas is fixed.

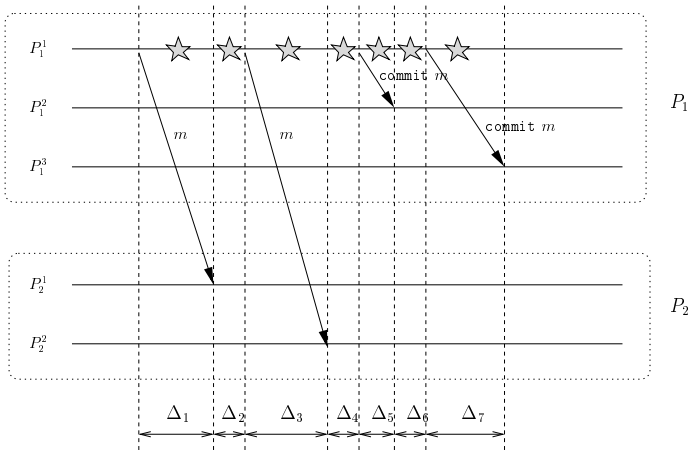


Figure 4.8: Possible failures on the master while sending to the destination processes

In this case, the broadcaster P_1^1 may crash at seven different moments $(\Delta_1, \ldots, \Delta_7)$ during the atomic broadcast. We now examine how failures are handled depending of the moment.

Period	State of the system and actions taken
Δ_1	All of destination processes (P_2^1 and P_2^2) do not receive the message. When the fault detection service notifies that P_1^1 failed, then the new master on logical process P_1 is chosen and redo this send operation.
Δ_2	This situation is identical to Δ_1.
Δ_3	P_2^1 received the message but P_2^2 did not. However this crash of P_1^1 occurs before the commit stage. Thus, the new master of P_1 will resend this message. Thus, P_2^2 can receive this message whereas P_2^1 discards this message because the message is already received (looking up its history table).
Δ_4	This situation is identical to Δ_3.
Δ_5	All replicas in P_2 received the message but the failure happens before the commit message reaches P_1^2 and P_1^3. Hence, whatever process (P_1^2 or P_1^3) is chosen to be the new master of P_1, it will resend the message. Thanks to the history table, all replicas of logical process P_2 will discard the message.
Δ_6	This situation is identical to Δ_5.
Δ_7	At this stage, P_1^2 knows that the message is transmitted while P_1^3 does not. When the new master is chosen, it can be either P_1^2 or P_1^3. If P_1^2 becomes the new master then it does nothing because it knows that the message has been transmitted. If P_1^3 becomes the new master then it will retransmit the message, but the destination processes will discard the message because the message is already received (looking up its history table).

Notice that it is sufficient to observe the behavior of the sender to check the protocol

coherence against the atomic broadcast properties. If a failure occurs at any of the other processes involved, the failed process definitively leaves the group (since we consider fail-stop failures) and its state should not be considered anymore.

4.5 Replication and Failure Probability

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

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, considering a p processes MPI application without replication, the probability that it crashes is :

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

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, considering a p processes MPI application with replication degree r, the probability that it crashes is

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

Figure 4.9 shows the failure probability curve depending on the replication degree chosen ($r = 1$ means no replication) where f has been arbitrary set to 5%.

Notice that doubling the replication degree increases far more than twice 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*, introduced in Section 3.1.2. In each host, this service is responsible to notify the local application process of failures happening on co-allocated processes. Thus, the design of the failure detectors is of primary importance for fault-tolerance. We discuss this issue in the following section.

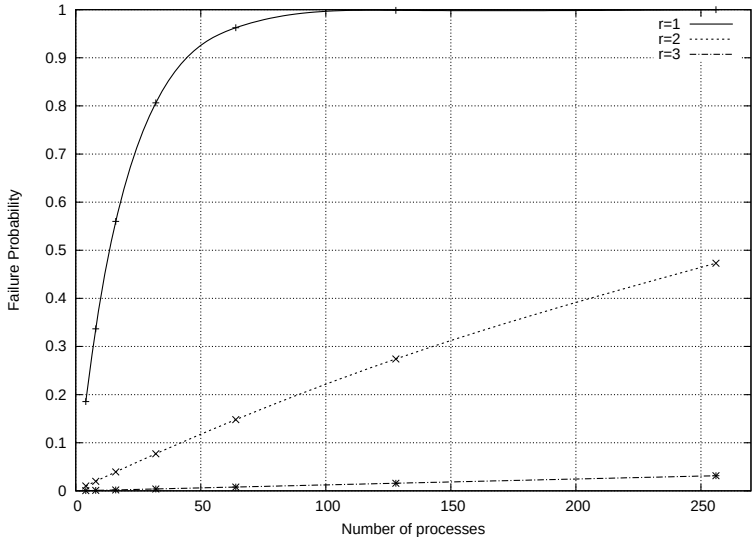


Figure 4.9: Failure probability depending on replication degree r (f=0.05).

4.6 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 [34], many protocols for failure detection have been proposed and implemented. Two classical approaches are the *push* and *pull* models discussed in [33], 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 more scalable protocol is called *gossiping* after the gossip-style fault detection service presented in [4]. 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, 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 order to keep the system's coherence, a consensus phase generally follows to acknowledge that host A has failed.

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 value 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 [69]. We briefly review advantages and disadvantages of the original and modified gossip based protocols and what has 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 [4], each module randomly chooses at each step, the hosts it sends its table to. In practice, random gossip evens the communication load among the network links but has the disadvantage of being non-deterministic. It is possible that a node receives no gossip message for a period long enough to cause a false failure detection, i.e. a node is considered failed whereas it is still alive. 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. Destination node d of a message is determined from source node s and current round number r, as follows :

$$d = (s + r) \mod n, \quad 0 \leq s < n, 1 \leq r < n \tag{4.1}$$

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 six nodes system, the set of communications taking place is represented in the table in Figure 4.10.

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 another 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

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	$0 \rightarrow 5$, $1 \rightarrow 0$, $2 \rightarrow 1$, $3 \rightarrow 2$, $4 \rightarrow 3$, $5 \rightarrow 4$

Figure 4.10: Communication pattern in the round-robin protocol ($n = 6$).

we can deduce the minimum cleanup time, depending on an integer number of rounds r such that:

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

For instance in Figure 4.10, 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:

$$d = (s + 2^{r-1}) \mod n, \quad 1 \leq r \leq \lceil log_2(n) \rceil \tag{4.2}$$

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

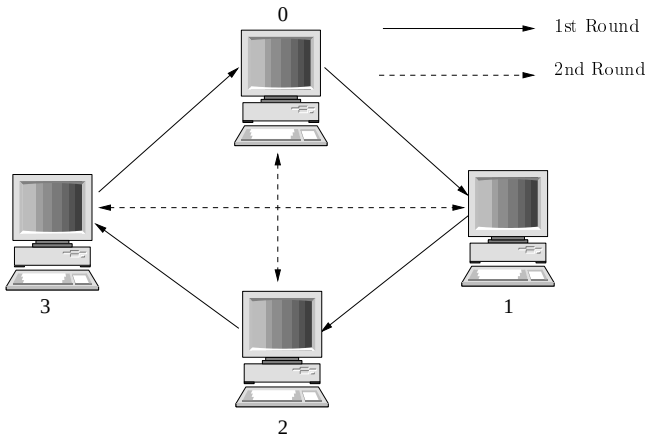


Figure 4.11: Communication pattern in the binary round-robin protocol ($n = 4$).

From our experience (also observed in experiments of Section 4.8.1), in a partially synchronous system, provided that we are able to make the distributed FD start nearly

at 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 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 4.11 shows a 4 nodes system. From equation 4.2, 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.

4.7 Fault Detection in P2P-MPI

From the above 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 [69]. 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 that the node has 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 Kb message (and 256 Kb 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 the FD service. We now describe the requirements we have set for the middleware, and which algorithms have been implemented to fulfill these requirements.

4.7.1 Assumptions and Requirements

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.

The assumptions we make regarding our system are those formulated in Section ??: we assume partial asynchronous system (there is no global clock, the local clock drift differences from one host to another is negligible during an application execution). 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).

4.7.2 Design issues

In early versions of P2P-MPI, the fault detection 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 results from 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 four 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 $P(i)$ be the probability that i nodes simultaneously fail during T. When 2 nodes fail, if both of them are source nodes then there will be a node that can not get any gossip messages. There are 4 such cases, which are the failures of {2,3},{0,3},{0,1} or {1,2}. When 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 4.5 (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.

4.7.3 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.

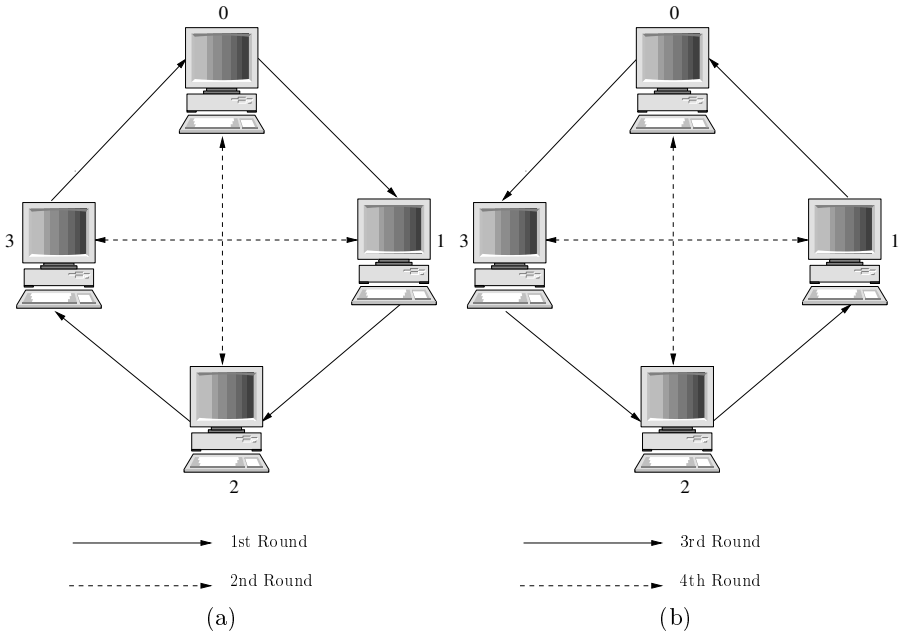


Figure 4.12: Communication pattern in the double binary round-robin protocol ($n = 4$).

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 is acceptably fast in practice, while reinforcing robustness of BRR. The idea is simply to avoid to have only one-way connections between nodes. Thus, in the first half of a cycle, we use the BRR routing in a clock-wise direction while in the second half, we establish a connection back by applying BRR in a counterclock-wise direction. The destination node for each gossip message is determined by the following relation:

$$d = \begin{cases} (s + 2^{r-1}) \mod n & \text{if } 1 \leq r \leq \lceil log_2(n) \rceil \\ (s - 2^{r-\lceil log_2(n) \rceil - 1}) \mod n & \text{if } \lceil log_2(n) \rceil < r \leq 2\lceil log_2(n) \rceil \end{cases} \tag{4.3}$$

The cycle length is $2\lceil log_2(n) \rceil$ and hence we have $T^{min}_{cleanup} = 2\lceil log_2(n) \rceil \times T_{gossip}$. Figure 4.7.3 shows the communication pattern in the double round-robin protocol for four processes. 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 4.7.2 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. This case implies a FD failure. In Figure 4.13, 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.

4.7.4 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 submitter constructs a table numbering available resources called the communicator[5], 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. Thus, the idea is 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 having the submitter sends

[5]The submitter is always assigned the number 0.

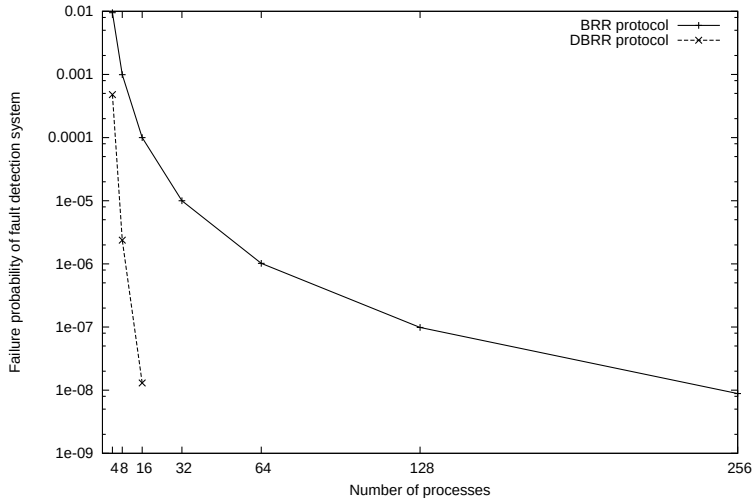


Figure 4.13: Failure probabilities of the FD system using BRR and DBRR ($f = 0.05$).

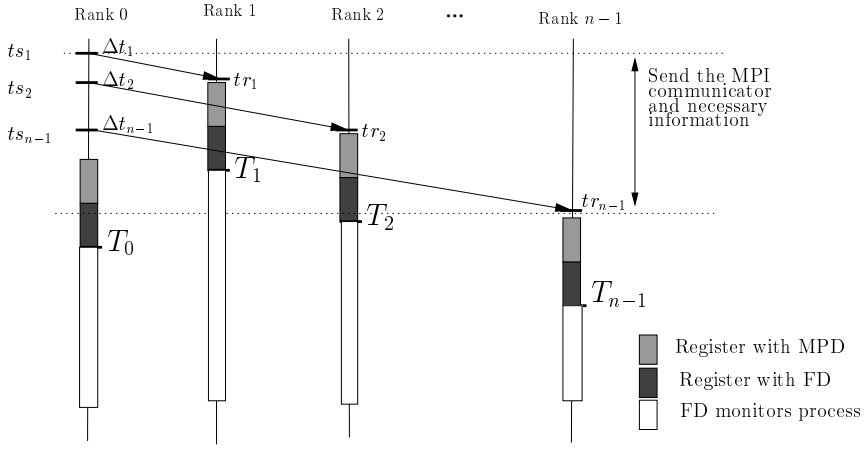


Figure 4.14: Application startup.

to each node, together with the communicator, the time spent sending information to previous nodes. Figure 4.14 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 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 compute its initial heartbeat h_i as:

$$h_i = \lceil (T_i - tr_i + \Delta t_i)/T_{gossip} \rceil, \quad 1 \leq i < n \tag{4.4}$$

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

4.8 Experiments

We present in this section some experimental results regarding two aspects of our approach of fault-tolerance. The first experiment tests the behavior of the fault detection algorithms in the FD service in real conditions. The second experiment shows the overhead induced by replication on some test cases.

4.8.1 Fault Detection Time

Objectives

We have seen that P2P-MPI provides two gossip-style protocols: the Binary Round Robin (BRR) and Double Binary Round Robin (DBRR) algorithms. Because they use a deterministic routing of information messages, these two modified gossip-style protocols allow to predict the fault detection time in theory. So, the experiment's objective is to compare the predicted detection time with the detection times observed when failures occur in a real application.

Experiment Setup

We use Grid5000 to get enough processors for the experiment. We run an application distributed across three the distant sites Nancy, Rennes and Sophia-Antipolis.

Environment type	Grid5000 – grillon.nancy, paravent.rennes azur.sophia clusters
Hardware	dual-cores AMD Opteron 2GHz, 2GB RAM
Operating System	Linux 2.6.14
Interconnection	2 ports GE cards intra-cluster, 10 Gbps/s between sites.
Software	jdk1.5, p2pmpi-0.20.0

The experiment consists in running a parallel application without replication. After 20 seconds we kill all processes on a random node to simulate a node failure. We then log at what time each node is notified of the failure and compute the time interval between failure and detection. For both protocols BRR and DBRR, the T_{gossip} value is set to 0.5 second.

Experiment Results

Figure 4.15 plots the average of these intervals on all nodes. Also plotted for comparison is $T_{cleanup}$ as specified previously, termed "theoretical" detection time on the graph.

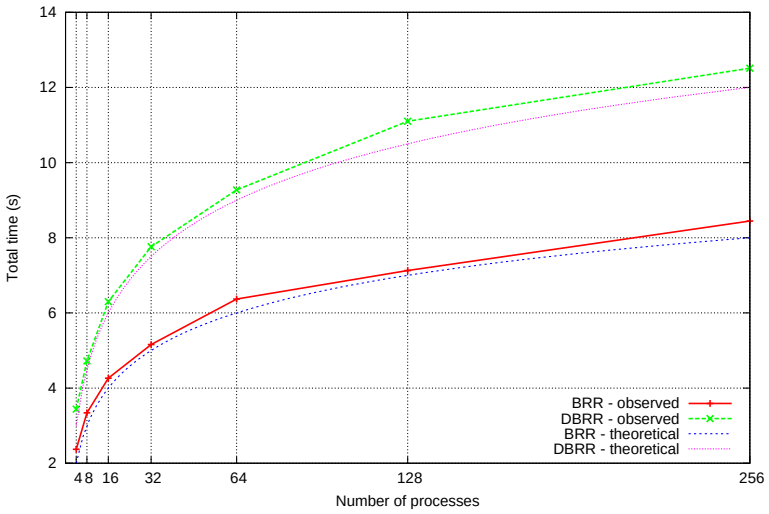


Figure 4.15: Time to detect a fault for BRR and DBRR

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 4.4. 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 heartbeat adjustment is effective and message delays are less than T_{gossip}. This experiment shows the scalability of the system on Grid5000, despite the presence of wide area network links between hosts.

4.8.2 Replication Overhead

Objectives

It is difficult to give a fair estimation of the cost of replication. The main cost is the extra resources it requires. The secondary cost is the time penalty since replication involves extra network communications. We should consider a huge panel of situations to reflect the costs of replication. Indeed, it depends on the application itself, on the network environment, and on the available resources. When there are not enough resources, replication will map several processes per processor (hence sharing the CPU power) and the cost will be much higher than when enough computers are available to run one replica each. Obviously, assessing the cost of replication would deserve a thorough study. As a first evaluation, we present results obtained on a test application with an early version of P2P-MPI on commodity hardware, and recent tests with our latest implementation.

The general idea is that the communication cost of the application should grow linearly with the replication degree, since each message is sent to all replicas. Our first tests with replication were conducted in a student computer room, and we observed the linear cost of replication on some simple ping-pong test. Unfortunately, we did not gather enough data to present comparative results here. In *Experiment 1* however, we show the impact of replication on an application, the IS program from the NAS benchmark.

We have conducted further experiments using our new multiple port implementation (see Chapter 5). The environment we used is a state of the art cluster. We chose such a platform to ease our experiments as they required up to 128 processors. Our general conclusion is that the replication cost is far more complicated to predict in this high-performance environment, maybe due to network congestion. Nonetheless, we present in *Experiment 2* results as a first evaluation of replication cost.

Experiment 1 Setup

The experiment uses a student computer room of 24 PCs, when the computers were available.

Environment type	Student computer room
Hardware	24 Pentium-IV 3 GHz, 512 MB RAM.
Interconnection	100 Mbps Ethernet, LAN.
Operating System	Linux 2.6.10.
Software	jdk1.5.0, JXTA-J2SE 2.3.3, p2pmpi-0.2.0

Application test

We initially tested the EP (Embarrassingly Parallel) and IS (Integer Sorting) programs from the NAS benchmarks. As explained in Section 3.5.2 page 67, we chose those pro-

grams for their opposite characteristics.

We do not present results for EP as the test shows very little difference whether we use replication or not. Results for IS in this environment are reported in Figure 4.16. IS requires a number of processes being a power of two, so only eight out of the twenty-four PCs could be used. NAS benchmark proposes several classes for each test, which denotes different problem sizes and computation complexities.

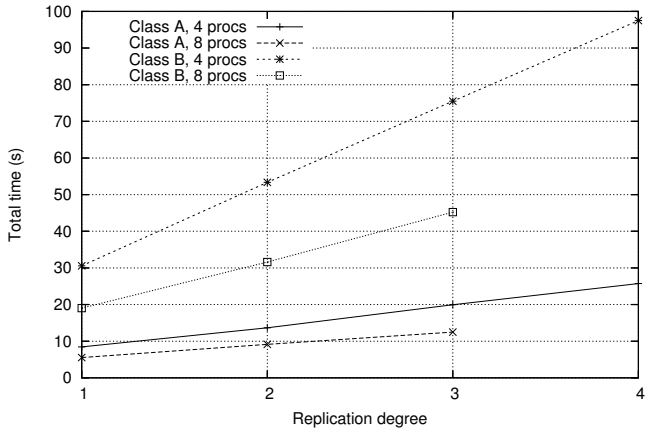


Figure 4.16: Performance for IS depending on replication degree.

The plot on figure 4.16 shows the time needed to compute a given problem class with four or eight processes, and its evolution depending on the replication degree. For example, for curve "Class B, 8 processors", 3 replicas per logical process means 24 processors were involved. We have limited the number of logical processes so that we have at most one replica per processor to avoid load-imbalance or communications bottlenecks.

The figure shows a linear increase of execution time in the replication degree, with a slope depending on the number of processors and messages sizes.

Experiment 2 Setup

The cost of replication is measured in this recent experiment with the new multiple port implementation. We use one of the Grid5000 clusters to have enough nodes to go up to a replication of four without hosting more than one process per CPU (core).

Environment type	A cluster in Grid5000 – paravent.rennes
Hardware	64 dual-processors AMD Opteron 246 2.0GHz, 2 GB RAM.
Interconnection	Gigabit Ethernet.
Operating System	Linux 2.6.19.
Software	jdk1.5.0_09, p2pmpi-0.28.0

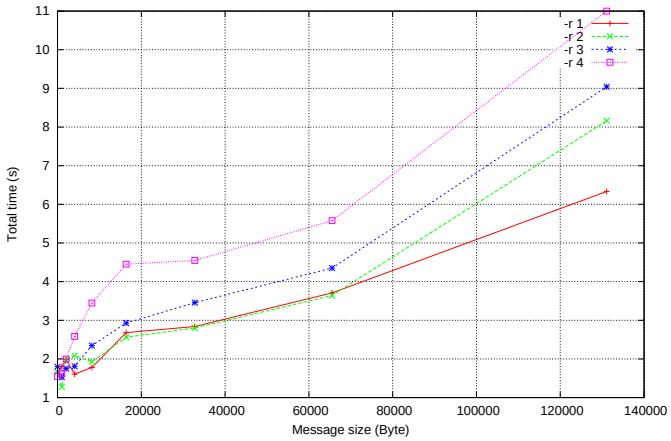


Figure 4.17: Time spent for 1000 ping-pong messages with different replication degrees.

Ping-pong

We first try to isolate the communication overhead with a sample *ping-pong* program, which simply sends a message to another process and receives it back. The sender is process P_0 with rank 0, and the receiver is process P_1 with rank 1. Notice that P_0 is not replicated (by convention) and hence we measure only once the replication overhead when P_1 sends its message back to P_0. The ping-pong message is sent 1000 times to reduce possible start-up side effects (e.g TCP slow start).

The test is done for different message sizes, from 1 KB to 128 KB. We vary the replication degree for this program, from one to four. We report in Figure 4.17 the round trip time for the 1000 message exchanges.

We observe that the minimal overhead for replication is less than expected. We expected the execution of ping-pong with a replication degree r to be r times longer than ping-pong without replication (t_1). If t_r is the time for ping-pong with replication degree r, we always have $t_r < r \cdot t_1$ in the range of message sizes tested. 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. Thus, it seems that the communication library efficiently manages multiple connections in this hardware environment.

Application test

Figure 4.18 shows the performances of IS in the setup described above. Like in Experiment 1, we have at most one process per node (CPU).

The application shows a good speed-up until sixteen processes, while using thirty-

two processes nearly leads to a slowdown. (These results outperform the performance shown in Figure 4.16 because of the vast superiority of hardware used in Experiment 2.) Hence, if we look at the execution in the relevant range from four to sixteen processes, we observe that replication adds an overhead smaller than in Experiment 1 with commodity hardware.

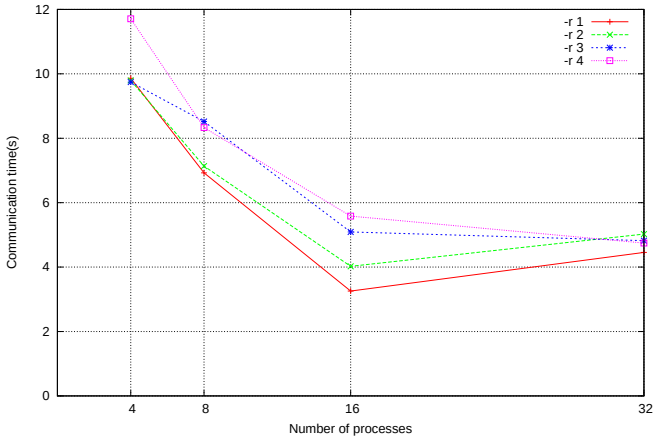


Figure 4.18: Performance for IS class B depending on replication degree and number of processes.

Given the type of application IS represents in terms of communications involved, the results observed in both a high performance cluster and in a commodity hardware environment are encouraging. A larger panel of applications, as well as a precise analysis of communication costs for replication should be studied in a future work.

4.9 Conclusion

We have described in this chapter the fault management underlying P2P-MPI. The first part introduces replication for fault tolerance, and we explain how this fits in our framework. We show how replication increase the robustness of applications execution. The fault detection issue is also a part of fault management. Fault detection in our work consists in an external monitoring of execution done by a specific fault-detection service. 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 Grid5000. One experiment addresses

the fault detection speed and accuracy. 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. The second experiment is a first experimental evaluation of replication overhead. The figures show the cost of replication on a reference application, in two different computing environments, namely, a set of networked commodity PCs and a high performance cluster. In both situations, the communication cost increased with the replication overhead appears to be at most linear in the replication degree.

Chapter 5

MPJ Implementation

We have seen in Chapter 2 that the MPI standard documents provide a language-independent specification as well as language-specific (C/C++/Fortran) bindings. However, no Java binding has been offered or is planned by the MPI Forum. In the late 1990's, with the evident success of Java as a programming language, and its inevitable use in connection with parallel as well as distributed computing, the absence of a well-designed language-specific binding has been considered problematic. Indeed several different MPI-like bindings for Java were developed independently. We have listed some of these in section 2.5, page 46. To tackle this problem, a community of researchers have set up a forum, the Java Grande Forum[1]. Participants to the forum have been working at the development of a consensus and recommendations on possible enhancements to the Java language and associated Java standards, for large-scale applications. The Message-Passing Working Group of the Java Grande Forum formed in 1998, came up with a recommendation for a common API for MPI-like Java libraries. The chosen name for the recommendation is MPJ (Message Passing interface for Java) to avoid confusion with standards published by the original MPI Forum. The rationale for the API design can be found in [3].

5.1 Introduction

In the early design phase of P2P-MPI, we have rapidly come to the conclusion that our API had to conform to MPJ. Independently from performance considerations, we wanted our API to allow us to run on as large panel of codes as possible.

Concerning the implementation, we have not concentrated on performance issues at first. We wanted to offer a message-passing paradigm, able to execute programs in a large variety of environments. Among the major limitations to the execution of programs distributed over different administrative domains are the firewall policies. With

[1] http://www.javagrande.org/

P2P-MPI, our strategy has been to limit the range of ports that need to be open to a minimum. We even developed a first prototype using JXTA pipes, in the spirit of the P3 project [70]. The idea of P3 was also to propose a message-passing programming model, even if their API is much simpler than MPJ. We rapidly abandoned the JXTA option to rely on a more stable communication layer, with a well-controlled behavior.

Thus, until very recently, the MPJ implementation we proposed was targeted to large scale environments and should only be competitive in terms of performance with other communication models such as RMI for example. This implementation is solely based on the Java implementation of TCP sockets, and connections are opened one at a time so that a single open port is required[2]. We call this implementation the *single-port* implementation.

Recently, we have started a new implementation which assumes no restriction on open ports. This allows us to use as many sockets as needed to speedup communications. We rely on the java NIO class (available since JDK 1.4). This class provides the equivalent of the *select* operation of *libc*, which allows a program to monitor multiple file descriptors, waiting until one or more of the file descriptors become "ready" for some I/O operation. This new implementation is called *multiple-ports*.

This chapter explains the design and the implementation of these two strategies. To differentiate the two kinds of implementation we use the word *device*, because the only difference lies in the communication device they use. Though the merge of the common code base is not achieved yet, we plan to release a single implementation containing both devices in a near future. The user could choose its preferred device at boot-time.

5.2 The Single-Port Device

In this strategy, each MPI process uses only one local port of communication. Every communication involves three steps : open the connection, send messages, and then close the connection.

Figure 5.1 shows the structure of a single-port device. After the MPI communicator is created, a device thread is initiated. The device thread is used to receive MPI messages and hence the main thread can continue with computations without interruption. The use of a device thread makes this device behave in asynchronous mode by nature. Whenever some messages arrive, the message header information is extracted to compute the message unique identifier (see Section 4.3.1), and the message is inserted accordingly into the message queue, implemented as a hash table.

Recall that the MPI standard requires that when several messages with the same *tag* are sent, they are received in the same order they were sent. The use of the MID to

[2]Actually, four extra ports need to be open for the MPD, FD, FT and RS services.

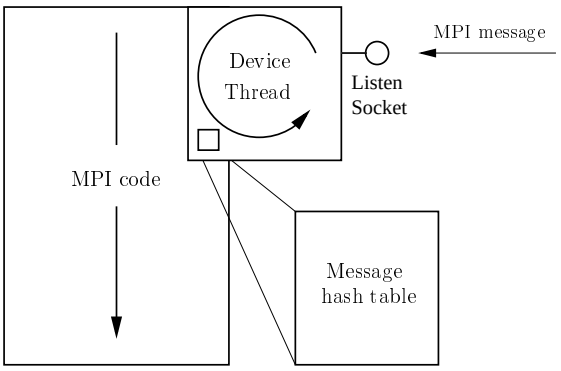


Figure 5.1: The structure of single-port device.

later extract messages from the receive queue insures that the communication operations sequence is respected. There is no head-to-head problem in this device because receivers are always ready to consume messages. The behavior of the basic communication primitives can be summarized as follows.

Send messages. When a `MPI.Send` is issued, the first step consists in the encapsulation of the message data into an MPI message object. Then, the main thread (MPI process) opens the connection to the destination MPI processes. The MPI message object is serialized and is sent. Finally, the connection between the two processes is disconnected.

Receive messages. There are two receiving modes in MPI, namely blocking and non-blocking modes. Also, we must consider two situations regarding the receipt of a message: the message arrives before or after the receive primitive has been issued by the user application. Hence, combining the modes and these situations, there are four possible scenarios which requires specific handling:

In blocking mode : (i) if the message arrives before the MPI receive is called, the message is extracted, and its header information is used to create an MID. Then, the message is put in the hash table using its MID as a hash key. When the MPI receive is triggered, the MPI main thread looks in the hash table by MID. The message corresponding to the first matching MID is taken from the hash table to the user-space buffer. (ii) if MPI receive is called before the message arrives, the main thread loops looking for the message in hash table. When the message arrives in the hash table, the main thread takes the message to the user-space buffer.

In non-blocking mode : (i) if the message arrives before the receive is called, the message is handled as in blocking mode, except that the message is taken out of the queue during `MPI.Wait`. (ii) if the message arrives after the receive is called, the main thread instantiates an MPI Request object, then it continues executing

the subsequent application code. Later, the main thread stops at `MPI.Wait` which loops looking for the message in the queue. When the message arrives in the hash table, `MPI.Wait` takes the message to the user-space buffer.

Summary. This device offers a solution in some situations where the execution environment has strong limitations with respect to the number of TCP ports that can be opened. However, it costs some overhead: the major overheads are the numerous connection openings and closings and the serialization/deserialization of Java object messages. Even though, this overhead can be acceptable when applications have a high computation to communication ratio.

5.3 The Multiple-Ports Device

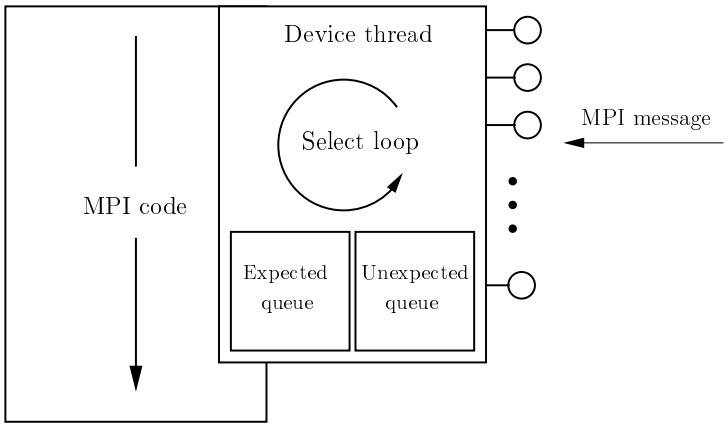


Figure 5.2: The structure of multi-port device.

With this device, our goal is to improve communication performance when no restriction applies regarding the TCP ports open. Figure 5.2 shows the structure of the device. After an MPI communicator is created, a device thread is initiated. This device is based on java NIO and uses `SocketChannel` as communication channel. The `SocketChannel` handles its data through Java `ByteBuffer`s only. So, it is mandatory to cast user messages so that they transit through `ByteBuffer`.

Each process connects to every other processes with two channels. One is the *write channel* and the other is the *read channel*. Write channels are in blocking mode while read channels are in non-blocking mode. The `Selector` provided in Java NIO lets us handle multiple channels in the select loop. Semaphores are used on each channel to insure that no concurrent processes write or read data from this channel at the same time.

Two queues are handled by the device thread. They are called *Expected queue* and *Unexpected queue*, and are implemented with two hash tables. The message identifier

(MID) is a hash key and message data is a hash value. We use one queue or the other depending if the message arrives at the receiver side before or after the corresponding receive operation is called. To better understand these two queues, let us examine them from two view points: one from the MPI receive call and another from a message arrival.

The MPI receive call's view : when a MPI receive is called, a `Request` object is instantiated. A `Request` object is implemented by a Java thread which inherits from the `Wait()` method to block and make a thread sleep, and from the `Notify()` method to wake up a thread. The first action of `Request` is to look in the unexpected queue to find a message with a MID matching the receive instruction. If no message with such MID exist, `Request` puts the MID and an associated null message in the expected queue. Then, it calls `Wait()` to stay blocked waiting for this message. Otherwise, if the message with the corresponding MID exists in the unexpected queue, it is moved from the queue to user-space buffer.

The arrived message's view : when a message arrives, the read message handler in the selector looks in the expected queue if there already exists an MID for the arrived message. If the MID exists, it means the MPI receive has already been executed. The message handler invokes `Notify()` to wake up a `Request` object to read the message. If the MID does not exist, the message handler puts the message data into the unexpected queue.

Like many MPI implementations, we define two modes for communication. One for small messages, that the receiver can accept directly in its queue. It is the *eager mode*. The implementation switches to the other mode called *rendez-vous mode* when the message's size exceed a fixed limit.

Eager mode

The eager mode is used for sending small messages. In P2P-MPI we set eager size limit to 128 KB. For any message whose size is lower than 128 KB, the eager mode will be used. This mode assumes that the receiving buffer on the receiver side is big enough to store the whole message. There is no exchange of control messages before the actual data transmission. This minimizes the overhead of control message that may dominate the total communication time for small messages.

Send message: whenever a send method is called, the sender main thread transforms the message into a java `ByteBuffer` then writes the message data into the writing channel. Except in non-blocking send, the main thread spawns a thread to send messages.

Receive message: when a message arrives to a receiver, the selector notifies the main thread to perform an action. If the receive is called before the message arrives in eager mode, the receiver reads a message and puts it in the queue.

Rendez-vous mode

The rendez-vous mode is used for communicating large messages, typically greater than 128 KB. There is an exchange of message between the sender and the receiver before the actual transmission of the data. For large enough messages, the overhead of this exchange of messages is negligible in terms of the overall communication cost.

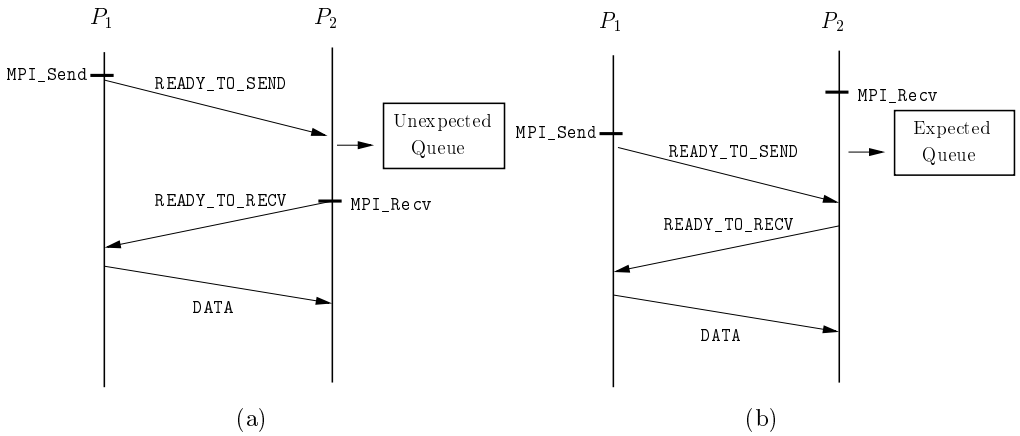


Figure 5.3: The rendez-vous protocol for sending a message.

Figure 5.3 shows the message exchange in rendez-vous mode before the actual data is transmitted. There are three steps to send the message. First, sender P_1 which invokes one kind of `MPI.Send` sends a `READY_TO_SEND` message to receiver P_2. A `READY_TO_SEND` message contains all the header information corresponding to a real message although it contains no actual data. If we consider the events of the `READY_TO_SEND` arrival and the `MPI.Recv` call, there are two possible interleaving shown in figure 5.3(a) and 5.3(b) respectively.

When a `READY_TO_SEND` message arrives before a `MPI.Recv` is called, the select loop which receives the message put it in the unexpected queue with empty data and set the flag that this message is in rendez-vous mode. Then, `MPI.Recv` is called which verifies whether the rendez-vous flag is set or not. Since the message is in rendez-vous mode, the receiver replies to the sender with a `READY_TO_RECV` message to tell the sender that it is now ready to receive the actual data. Finally, when the sender receives the `READY_TO_RECV` message, it transmits the actual data to the receiver.

In another case, when a `READY_TO_SEND` message arrives after a `MPI.Recv` is called, `MPI.Recv` puts the user-buffer in expected queue and waits for the message. When `READY_TO_SEND` arrives to the receiver, it replies back with a `READY_TO_RECV` message. Then, the transmission of the actual data begins.

5.4 Collective Communication Operations

We detail in this section the optimizations introduced in the collective communication operations of P2P-MPI. Currently, we use well-known algorithms which have better performances in local clusters than in wide area networks. We have reviewed in Section 2.3 several contributions in that field. We have not yet integrated these ideas because the network topology has been taken into account very recently in our middleware. P2P-MPI in its current state might be seen as a first step towards a more sophisticated framework using the best suited among the various available algorithms, depending on the execution platform allocated.

The collective communication operations are found in the `IntraComm` class (appendix B.4, page 132). Table 5.1 shows methods and its algorithm.

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

Table 5.1: List of `IntraComm` methods.

The base methods are `Alltoall`, `Alltoallv`, `Barrier`, `Bcast`, `Gather`, `Gatherv`, `Reduce`, `Scatter` and `Scatterv`. The other primitives `Allgather`, `Allgatherv`, `Allreduce` and `Reduce_scatter` are constructed from these base methods. The `Allreduce` implementation switches between two algorithms. The butterfly algorithm is used when the number of processes is a power of two. Otherwise, it simply calls `Reduce` and then `Bcast`. In `Reduce`, there are also two algorithms. A binomial tree algorithm is used when the operation of `Reduce` is commutative. Otherwise, we apply a flat tree algorithm.

Asynchronous rotation

This algorithm is used for `Alltoall` and `Alltoallv`. Figure 5.4 shows a step-by-step trace on four processes. It completes in $N - 1$ steps for N processes. At each step

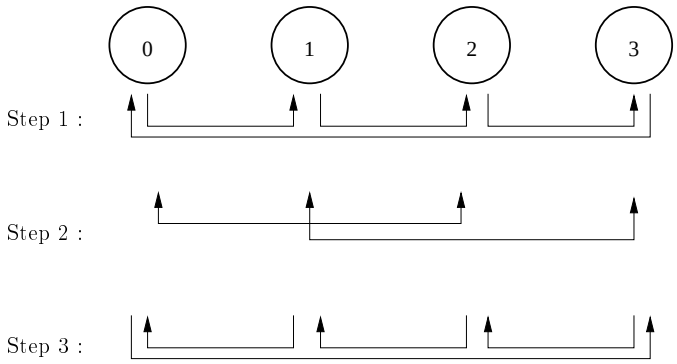


Figure 5.4: The steps of asynchronous rotation on four processes.

$i \in [1, N-1]$, all processes $s \in [0, N-1]$ computes the rank of a single destination process d as $d = (s + i) \mod N$.

In the example P_0 sends data to P_1 then sends to P_2 and P_3, in step 1, 2, and 3 respectively. In the meantime, P_1 sends data to P_2 at first step and then sends it to P_3 and P_0 respectively. We chose this rotation technique to reduce the network simultaneous load as `Alltoall` operations are highly congestive. Note also that the network load is equally balanced between processes in this algorithm.

4-ary Tree

This is the algorithm used in `MPI.Barrier`. We have made the same choice as MPJ Express [50] because of the good performance exhibited by this implementation on that point.

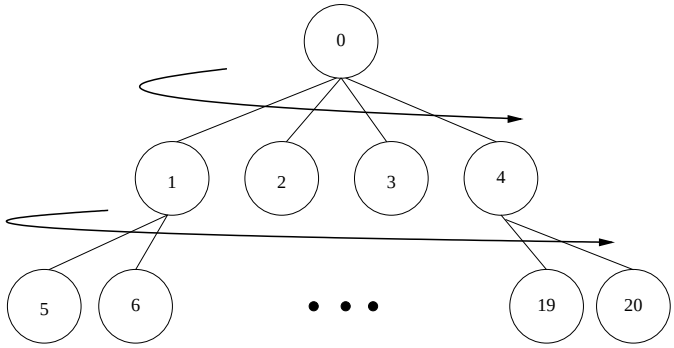


Figure 5.5: 4-ary tree structure.

Figure 5.5 shows the structure of a 4-ary tree structure. Each node can have a maximum of four children. The tree is build considering node 0 as the root of the tree.

The tree has N nodes, and its depth is $\lceil log_4(N) \rceil$. When building the tree, each node is assigned an index $i \in [0; N-1]$. The algorithm used defines for any node i:

- its children are node with index $4i+1, 4i+2, 4i+3$, and $4i+4$,
- its parent is the node with index $\lceil \frac{i-1}{4} \rceil$

The `MPI.Barrier` implementation performs a two phase tree traversal. First, node 0 sends a dummy message (a 1 byte data MPI message) to its children, which in turn transmit the message to their own children. When the message reach nodes with no child, these nodes send the message back to their parent. Thus, all messages traverse the tree back until they are collected at the root.

Binomial Tree

The binomial tree is introduced to reduce network contention. We apply this algorithm to `Bcast` and `Reduce`. A binomial tree is built up recursively, the whole tree at step $j-1$ is appended to the root node in step j. The principle is shown in figure 5.6.

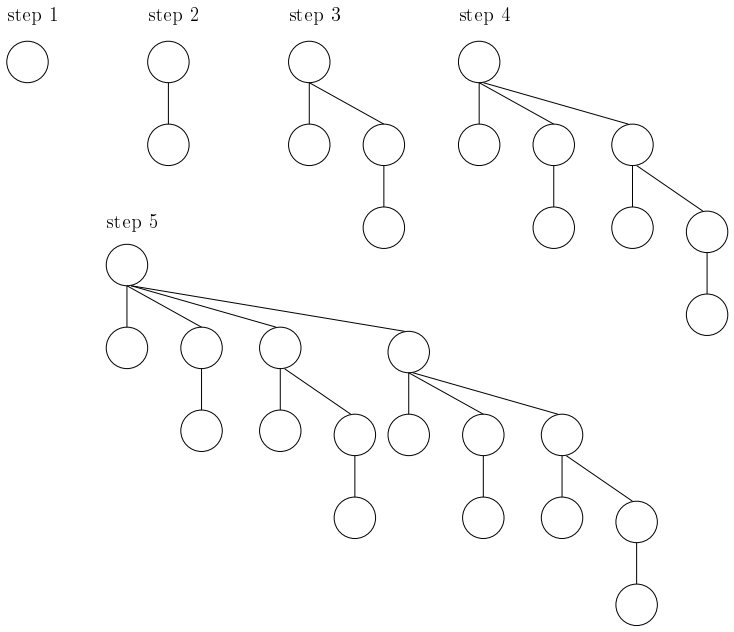


Figure 5.6: Example for building a binomial tree.

Contention on a single node is avoided thanks to the distribution of nodes along the binomial tree: each network link is utilized at most once per round. To manage the process-to-tree-node assignments of n processes, the following numbering scheme is used:

- each node is numbered in binary digits (from 0 to $n-1$)
- each node calculates its parent by resetting the leftmost "1" in its own id to "0"
- each node calculates its children by adding 2^i to its own id where $i = \{i \in N \wedge log_2(id) < i < \lceil log_2(id) \rceil \wedge id + 2^i < n\}$

The binomial tree also minimizes the concurrency at the root node. One child of the root node finishes each round. The root node has typically $\lceil log_2(n) \rceil$ children. Thus, in the `Bcast` case, the root node knows after $\lceil log_2(n) \rceil$ that all nodes received the broadcast message.

Butterfly

The butterfly algorithm [71] is implemented in the `Allreduce` method. This algorithm is called only when the number of processes involved is a power of two, and when the operation used in `Allreduce` is commutative. If n is the number of processes involved, the algorithm performs $log_2(n)$ steps of pairwise synchronizations.

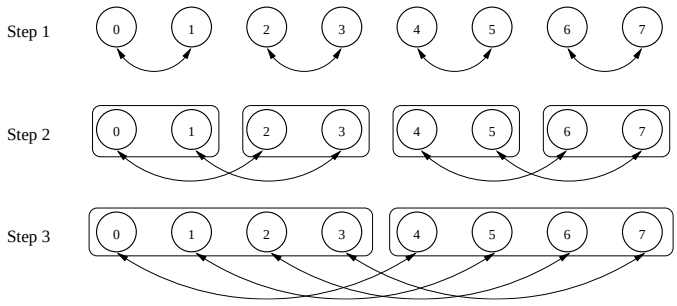


Figure 5.7: The butterfly algorithm for 8 processes.

Figure 5.7 shows the butterfly algorithm for eight processes. Let us call d and s the destination and source node numbers respectively. The $\oplus$ symbol refers to exclusive or (xor) operation and $\ll$ is left-bit shift. At each step i, a node calculates its pair with: $d = s \oplus (1 \ll i)$.

5.5 Experiments

Similarly to the experiments conducted in Section 4.8, we present performance results for different computing environments. We have collected results for earlier versions of P2P-MPI, which should not behave differently than the current single-port implementation. The set of benchmarks based on a older version of P2P-MPI is interesting because of the experimental conditions we had: the first experiment uses commodity hardware (a student computer room), while the second uses two sites of Grid5000 in august 2005.

Hence, the tests are complementary to more recent experiment.

We have conducted further experiments using our new multiple-port implementation. The environment we used is a state of the art cluster. We chose such a platform for two reasons. First, it is easier to plan our experiments as they required up to 128 processors. Second, this platform is a well-controlled environment in which we can reproduce experiments. This is a highly desirable feature to make a fair comparison between different implementations.

Experiments using highly heterogeneous environments such as nodes being dynamically chosen from PCs around is out of the scope. A precise assessment of P2P-MPI's behavior on such configurations is difficult because experiments are not easily reproducible.

5.5.1 Single-Port implementation

Objectives

The aim was to have a first feedback on the communication library performance. We found a student computer room was an adequate environment for testing, regarding the type of computing environment targeted by P2P-MPI. Then, we got an account of the Grid5000 testbed, which allowed us to test the software scalability (with more than a hundred processors for the first time), and to see how latency between site affected the performance.

Note that all tests are done with a replication degree of one, as replication overhead has been studied in the previous chapter.

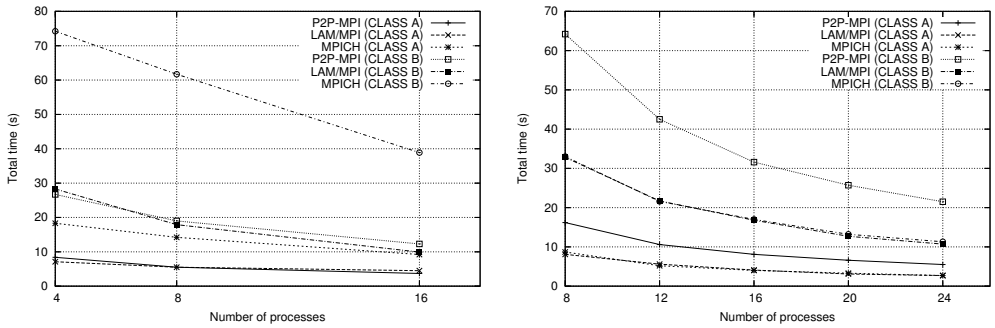
Experiment 1 Setup

The computers are simple PCs, fully available during the experiment. The benchmarks for these experiments are IS and EP from NAS benchmarks (NPB3.2), like in Section 3.5.2.

Environment type	Student computer room
Hardware	Intel Pentium4 3GHz, 512MB RAM
Operating System	Linux 2.6.10
Interconnection	100 Mbps Ethernet.
Java runtime	J2SE-5.0.
MPI Implementations	MPICH-1.2.6 (p4 device), LAM/MPI-7.1.1, and p2pmpi-0.2.0

Experiment 1 Results

It is expected that P2P-MPI achieves its goals at the expenses of an overhead incurred by several factors. First, the FD service sends regular heart-beats and therefore uses the network card from time to time. Second, the protocols for replication impose bigger



(a) Performance on IS benchmark.

(b) Performance on EP benchmark.

Figure 5.8: Comparison of MPI implementations performance for IS and EP.

message headers than those for simple communications. Moreover, compared to fine-tuned optimizations of communications of MPI implementation (e.g in MPICH-1.2.6, Thakur [72] uses four different algorithms depending on message size), P2P-MPI has simpler optimizations (e.g. binomial trees). Finally, the use of a virtual machine (Java) instead of processor native code leads to slower computations.

Figure 5.8(a) and 5.8(b) plots result from benchmarks IS and EP respectively. We have kept the same timers as in the original benchmarks. Values plotted are the average total execution time. For each benchmark, we have chosen two problem sizes (called class A and B) with a varying number of processors. Note that IS requires the number of processors be a power of two and that we could not go beyond 16 PCs.

For IS, P2P-MPI shows an almost as good performance as LAM/MPI up to 16 processors. The heart-beat messages seem to have a negligible effect on overall communication times. Surprisingly, MPICH-1.2.6 is significantly slower on this platform despite the sophisticated optimization of collective communications (e.g it uses four different algorithms depending on message size for `MPI_Alltoall`). It appears that the `MPI_Alltoallv` instruction is responsible for most of the communication time because it has not been optimized as well as the other collective operations. The EP benchmark clearly shows that P2P-MPI is slower for computations because it uses Java. In this test, we are always twice as slow as EP programs using Fortran. EP does independent computations with a final set of three `MPI_Allreduce` communications to exchange results in short messages of constant size. When the number of processors increases, the share of computations assigned to each processor decreases, which makes the P2P-MPI performance curve tends to approach LAM and MPICH ones.

Experiment 2 Setup

We choose two sites from Grid5000 testbed with homogeneous processors to isolate the impact of communications. The sites are Orsay and Sophia-Antipolis. At the time of the experiment (august 2005) the backbone link between these two sites has a 2.5 Gbps bandwidth.

Environment type	Grid5000 (gdx.orsay and azur.sophia)
Hardware	128 nodes AMD Opteron 246, 2GB RAM (64 nodes at Orsay and 64 nodes at Sophia)
Operating System	Linux 2.6.12
Interconnection	Gigabit Ethernet.
Java runtime	Java 1.5.0_08.
Benchmark suites	Modified RayTracer from JGF section 3
P2P-MPI implementation	P2P-MPI-0.10.0

The application used in this experiment is the *ray-tracer* from the Java Grande Forum MPJ Benchmark. We choose this application because it was reported in [51] to scale well with MPJ/Ibis. This program renders a 3D scene of 64 spheres into an image of 150x150 or 500x500 pixels in the original benchmark, but we have enlarged the image resolution to 2500x2500. Each process does local computations on its part of the scene for which a checksum is combined with a `MPI.Reduce` operation by the rank 0 process. In the end, each process sends its result to process 0.

Experiment 2 Results

In this experiment, we have run several series of executions of the application on different days of a week. A series consists in a set of executions using from 2 to 128 processes (one process per processor). We observe how the application scales on a single site (all processes at Orsay) and then when processes are distributed half on each site. We report on figure 5.9 the highest and lowest speedups obtained in each case. The application scales well up to 64 processors on a single site, and in some occasions the execution involving the two sites is even as quick as the lowest execution on one site. With 128 processors, the scalability largely decreases on one site, and turns to a slowdown with distant sites. We reach here a computation to communication ratio that does not allow for more parallelism. However, the experiment confirms the good scalability of the application provided the image to compute is big enough, even when two distant sites are involved.

5.5.2 Multiple-Port Implementation

Objectives

This section presents the benchmarks for our recent multiple-port implementation. As we have not yet been able to set up a complete test including other MPJ or MPI imple-

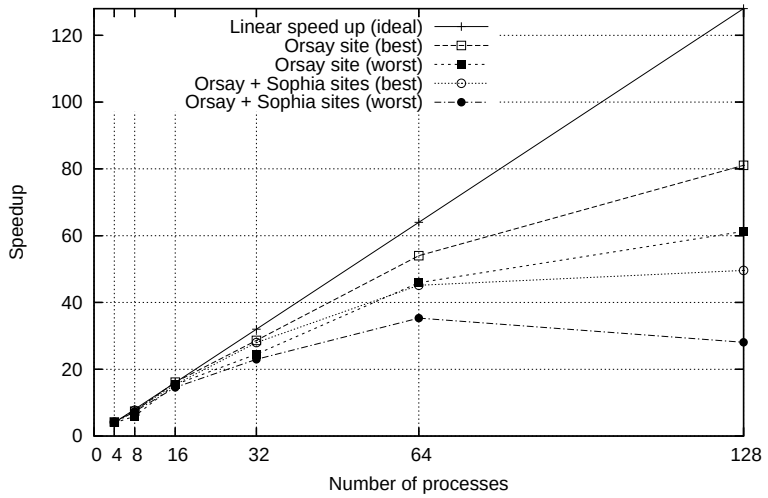


Figure 5.9: Ray-tracer speedups when run on a single site and on two distant sites.

mentations, we have mostly compared MP and SP performance.

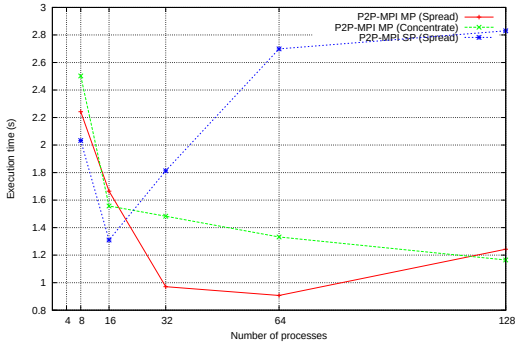
We have chosen the Java Grande Forum MPJ benchmark suites (JGF in the following) for the test. A detailed description of the benchmark can be found in Appendix A, page 123). Section 1 measures the performance of point-to-point operations in the communication library. We have also included MPJ-Express performance results for that section because it passes successfully this test. For Section 2 (kernels) and Section 3 (large-scale applications) we could only compare P2P-MPI SP and MP because MPJ-Express has a problematic implementation of `Allreduce`. It modifies the input buffer during the communication, which invalidates the computed results (the test ends with the 'validation failed' message).

Experiment Setup

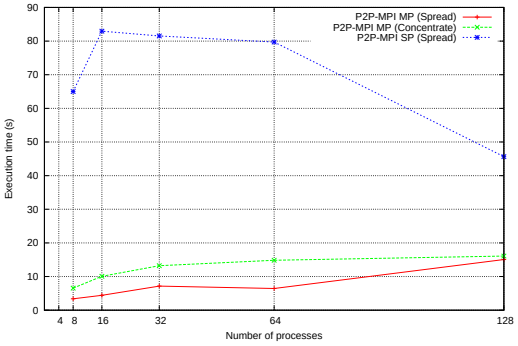
Environment type	Grid5000 (grelon.nancy)
Hardware	64 nodes/128 cores Intel Xeon 5110, 2GB RAM
Operating System	Linux 2.6.24-1-amd64
Interconnection	Gigabit Ethernet.
Java runtime	Java 1.5.0_08.
Benchmark suites	JGF section 2 (CLASS B) and JGF section 3 (CLASS A)
P2P-MPI implementation	P2P-MPI-0.27.1 (SP device) and P2P-MPI-0.28.0 (MP device)

Experiment Results

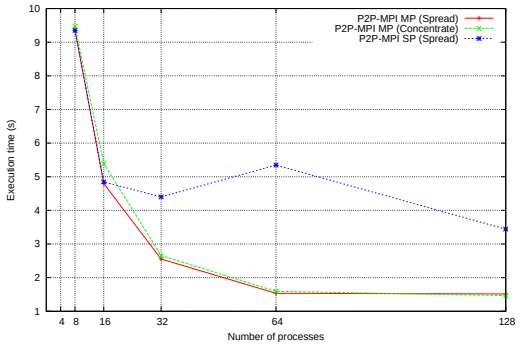
We present here the results for JGF Section 2 and Section 3. For sake of clarity, Section 1 results are reported in Appendix D, page 147. Figures 5.10 and 5.11 plots result from Section 2 and Section 3, respectively.



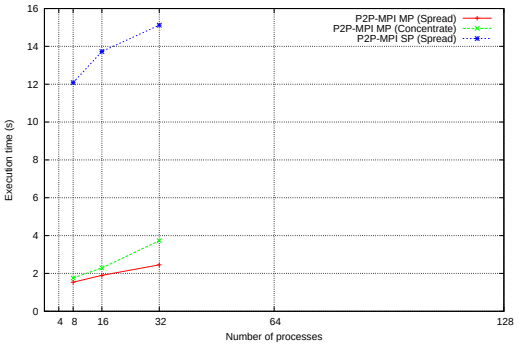
(a) crypt



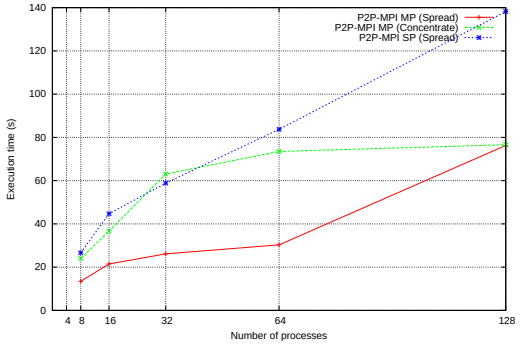
(b) lufact



(c) serie

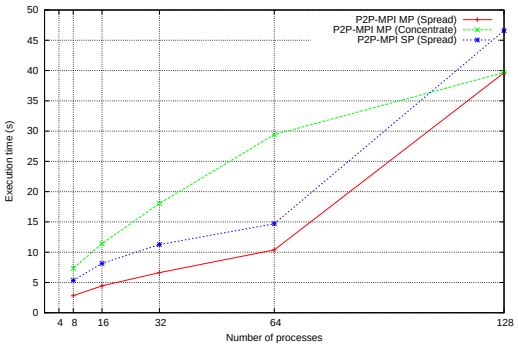


(d) sor

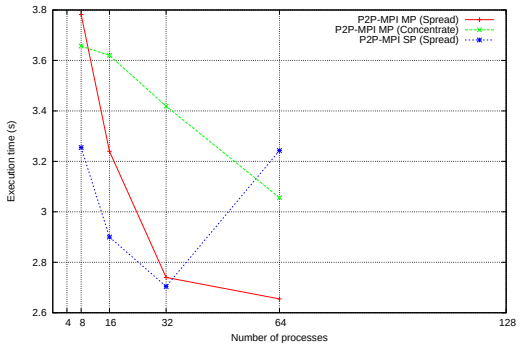


(e) sparseMatMult

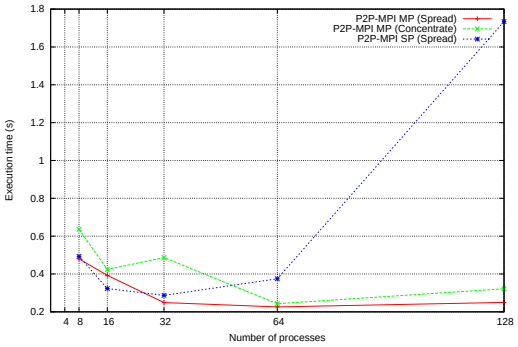
Figure 5.10: JGF section 2: Kernels benchmark results



(a) moldyn



(b) montecarlo



(c) raytracer

Figure 5.11: JGF section 3: Large-scale applications benchmark results

Note that we have used 64 nodes with two cores each, for a total of 128 computing cores. The experiment compares the two types of P2P-MPI devices SP and MP. As we have introduced allocation strategies in our latest versions, the impact of the chosen strastegy is also tested for MP. For SP, we only plot the curves for the *spread* allocation strategy because *concentrate* showed very poor performance as compared to MP. Note that when the execution requires 128 processes, both strategies should result in the same allocations because all the cores are used.

The first observation is that the *spread* strategy gives better results than *concentrate* in these benchmarks and on this cluster. The reason is probably that these benchmarks suites are much CPU and RAM demanding, and that putting two processes on a same node is not as efficient as using two cores on two separate nodes. The second observation is that MP outperforms SP in nearly all tests. In the *crypt* test, the SP implementation could not scale beyond sixteen processors. Except in *montecarlo*, where SP is better up to thirty-two processors, MP-spread generally has the quickest execution times. In SOR benchmark, we cannot go beyond 32 processes because JGF benchmark stops with an error ('negative array index exception'). This is the same for *montecarlo* for more than 64 processes.

5.6 Conclusion

We have explained the two alternatives explored regarding the communication library implementation. The first alternative is an implementation using as few TCP ports as possible. We can see it as a specific device of the communication library, and we call it the *single-port* (SP) device. The assumption here is that it is possible to open firewalls between different administrative entities for a restricted range of ports. The aim is thus to ease P2P-MPI deployments over several networks.

The second alternative is the classical approach of most MPI implementations, which establishes permanent links between processes. In order to execute an application, a wide range of ports might be opened. The implementation of this communication device is called *multiple-port* (MP).

It is up to the user to understand which implementation best fits its needs and constraints. The SP device optimizes the number of ports used but suffers from a lower performance. The MP device performs better but requires no restrictions on ports.

We have also detailed which algorithms were used in the communication library, especially for collective communications. Then, the two implementations have been benchmarked in several environments. The benchmarks must now take into account the allocation strategies that P2P-MPI provides to its user (see Chapter 3) as it greatly influences performance. Currently, we lack a sufficient number of programs to make a fair comparison of well-known MPI and MPJ implementations with P2P-MPI. We mainly used the JGF benchmark suite to conduct this first evaluation, but we could not achieve enough comparisons with other implementations (mainly MPJ-Express) to give definitive comparative results. Our feeling however, is that MPJ-Express performs better on small messages, and we leave as a future work to establish a thorough performance test, and to optimize the MP communication device.

We can conclude that no strategy is better in all cases. Allocating all the cores of a multicore computer with one process per core has sometimes shown a greater overhead than spreading the processes over several computers. It depends on how much intensive are the memory accesses in the application. In our context, several benchmarks seemed to reveal much memory contention with the *concentrate* strategy.

Finally, we have tested P2P-MPI against MPJExpress on a cluster. MPJExpress has better performances than P2P-MPI in general. A quick explanation is that P2P-MPI had not targeted performance at first. So far, we have a short experience at optimizing our primitives, while MPJExpress has invested much effort in it. However, we are currently investigating performance results to understand why some operations in P2P-MPI (such as `Gather`) outperform the MPJExpress version, while most others are slower. Thus, improving the performance of the MP implementation is still under work.

Chapter 6

Conclusion

We have described in this manuscript, a proposal for an integrated middleware coupled with a communication library. This proposal has been implemented and is publicly proposed as a free software project named P2P-MPI. A major design feature of P2P-MPI is its integrated approach. Our thesis is that an effective deployment of message passing programs on Grids is possible, provided the execution runtime can rely on appropriate middleware services.

The minimal set of services or features the middleware should provide has been described in Chapter 3 and Chapter 4.

Chapter 3 explains the design choices made to address the deployment of large-scale parallel message-passing programs. Since the beginning of the project, we have proposed a P2P basis to organize resources in a Grid. We have put forward the autonomy of peers, which enables an easy software installation of individual resources and the absence of a single point of failure since there is no central directory for resources. We put forward that the dynamic discovery of available resources upon an execution request is a highly desirable feature. Another benefit for the application can be an efficient allocation of resources by the middleware with respect to the application's needs. During, this work, we have modified the middleware to improve the allocation resources. The middleware now accounts for network locality of peers. Based on this information, P2P-MPI proposes simple and understandable resource allocation strategies to the user. We have shown through real experiments, that we could deploy applications using up to to 600 processes.

Chapter 4 discusses fault-tolerance. The middleware has a failure detection service, which notifies failures to the application. We have explained the difficulties to build a scalable and fast detection system, and how our service has been designed. The communication library supports fault-tolerance through replication of processes, upon a simple user request. We have described the underlying protocol, and we have shown how replication increases the robustness of applications. The overhead of replication is also studied. Thus, our proposal on fault-management contributes to show that the middleware support is beneficial to the communication library.

The last Chapter has detailed the communication library implementation. We have discussed the alternatives of using either a single communication port, better adapted to Grids, or multiple port to improve communication performance.

Finally, we think P2P-MPI can encourage programmers to parallelize their applications to benefit from the computational power available even from individual computers. During this thesis, we have helped at the parallelization of a data clustering method [73]. This work is described in [74]. This method has a high complexity and its parallelization enhanced its usability. Clusterings with a large number of classes have been completed in tens of minutes instead of hours in the sequential version. In addition, a noteworthy aspect is that P2P-MPI makes the parallel execution nearly transparent for the user. Users keep running the application from their usual computer, as the middleware transparently discovers available computing resources.

Throughout the chapters, we have discussed what could be improved in our proposals. Let us finally summarize the points that deserve future work. The middleware should rely on a more decentralized infrastructure, composed of a distributed set of supernodes, to scale beyond thousands of peers. A linked problem is to maintain an accurate estimation of the network latencies between peers, or better, being able to guess the topology of the physical network (similarly to the method used in [75]). As far as replication is concerned, a formal analysis of the protocol (e.g using model-checking) would make it a solid brick. A comparison with other approaches of fault-tolerance regarding for instance, the overhead depending on the number of faults injected, would be also interesting. Last, much work could be done on the MPJ implementation. In particular, we think P2P-MPI is a good framework to test new algorithm for mixed wide and local area communications. Many research works have proposed improved collective communications, which are very important in the Grid environment we target. Thanks to the cooperation with the middleware, the communication library could get static or even dynamic information about the network (topology, load, ...) to make maybe better decisions in its algorithms.

Appendix A

Experiment Testbeds and Benchmark Suites

This chapter gives an overview on experimental testbeds and benchmarks, we conducted during this thesis to test P2P-MPI. In an early experiment, we used a student computer room as a testbed to simulate a campus grid. We could get up to the maximum of 24 machines for the experiment. Later on, we conducted experiment on a Grid developed in France, called Grid5000. On this platform, we could reserve and experiment with up to 350 machines (i.e. a total of 700 CPUs/1040 cores). During these tests we used applications from two different benchmarks: NAS and JGF.

Grid5000. The project Grid5000[1] [76] is a testbed designed as a experimental grid. It is different from other computation grids by its high capacity to let users to do the reconfiguration and to totally control it. For example, it permits each user to deploy its proper operating system via Kadeploy2[77]. The term Grid5000 comes from the idea to build a experimental grid which has the total 5000 processors distributed across nine sites in France : Bordeaux, Grenoble, Lille, Lyon, Nancy, Orsay, Rennes, Sophia-Antipolis, and Toulouse. The Interconnection between the different sites are assure by RENATER-4[78] that provides the bandwidth of 10 Gb/s. The latency between the machines of different sites varies from 4ms to 29ms. Figure A shows the interconnection between the different sites in Grid5000. The processors in Grid5000 are quite heterogeneous, as there are AMD Opteron, Xeon, Itanium2 and PowerPC.

NAS Parallel Benchmarks. The NAS Parallel Benchmarks (NPB)[79], developed by NASA advanced supercomputing (NAS) division, are the small set of programs designed to help evaluate the performance of parallel supercomputers. The benchmarks consist of five kernels : Multigrid(MG), Conjugate gradient(CG), Fast Fourier transform(FT), Integer sorting(IS), and Embarrassingly parallel(EP) and three pseudo-

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

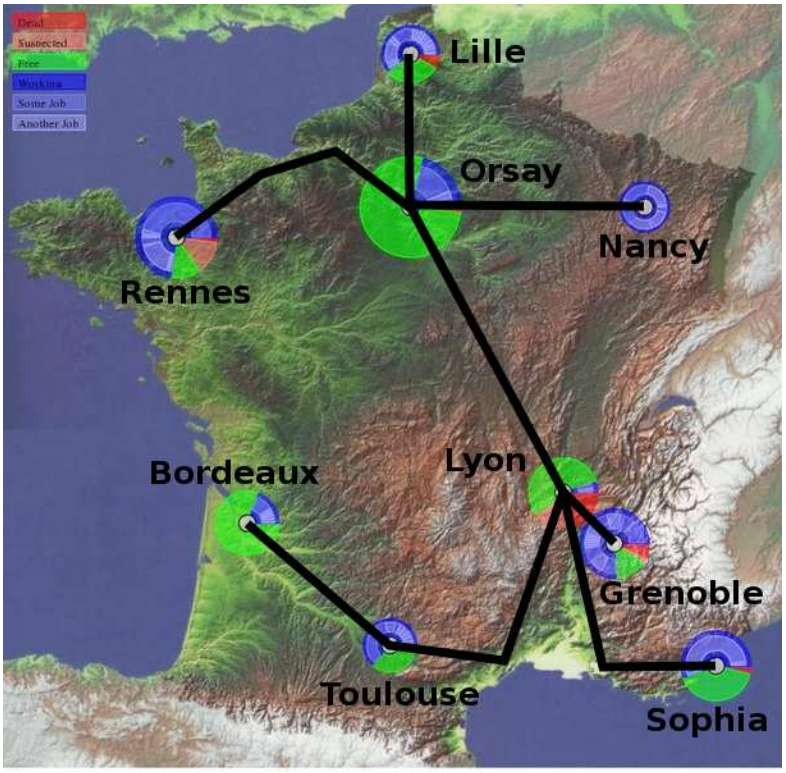


Figure A.1: The interconnection between nine sites in Grid5000.

applications : Block-tridiagonal(BT), Scalar-pentadiagonal(SP), and Low-upper symmetric gauss-seidel(LU).

We have chosen two kernels, IS and EP from NPB version 3.2, which have opposite characteristics and translated them in java from C and Fortran respectively, in order to run them with P2P-MPI.

IS (Integer Sorting). The IS benchmark is based on a bucket sort. The number of keys ranked, the number of processors used, and the number of buckets employed are all presumed to be powers of two. The number of buckets is a tuning parameter. Communication costs are dominated by an `MPI.Alltoallv`, in which each processor sends to all others those keys which fall in the key range of the recipient. This benchmark is used to measure the integer computation speed and communication performance.

EP (Embarrassingly Parallel). In EP benchmark, each processor independently generates pseudo-random number (PNs) and uses these to compute pairs of normally-distributed numbers. No communication is needed until the very end. This benchmark provides an estimate of the upper achievable limits for floating point performance. EP is closer to the class of applications usually deployed on computational grids where the computation takes the major part of the execution and the communication part is less

important.

Java Grande MPJ benchmarks. The benchmarks from the Java Grande Forum, a community initiative to promote the use of Java for so-called *Grande* applications. A Grande application is an application which has large requirements for any or all of : memory, bandwidth, processing power. Java Grande MPJ benchmark suite is one of Java Grande Forum(JGF) benchmark suites which is designed to measure the performance for parallel execution on distributed memory multiprocessors.

This benchmark suite is divided into three sections :

- **Section 1: Low level operations** - measuring the performance of low level operations.
  - **PingPong**: Point-to-point communication
  - **Barrier**: Barrier synchronization
  - **Alltoall**: All-to-all communication
  - **Bcast**: Broadcast (one-to-all communication)
  - **Scatter**: Scatter (one-to-all communication)
  - **Gather**: Gather (all-to-all communication)
  - **Reduce**: Reduction
- **Section 2: Kernels** - short codes which carry out specific operations frequently used in Grande applications.
  - **Series**: Fourier coefficient analysis
  - **LUFact**: LU Factorization
  - **SOR**: Successive over-relaxation
  - **Crypt**: IDEA encryption
  - **Sparse**: Sparse Matrix multiplication
- **Section 3: Large scale applications** - large codes, representing complete Grande applications.
  - **MolDyn**: Molecular Dynamics simulation
  - **MonteCarlo**: Monte Carlo simulation
  - **RayTracer**: 3D Ray Tracer

We use all three sections in our experiments.

Appendix B

P2P-MPI API

P2P-MPI provides a subset of MPI application programming interface (API) followed by MPJ specification. P2P-MPI is consist of nine main classes (table B.1).

Classname	Description
Comm	Point-to-point communication
Datatype	primitive datatypes
Group	MPI group
IntraComm	Collective communication
MPI	Main MPI class
MPI_User_function	Abstract class to implement a user-defined operation
Op	MPI collective operation
Request	The handle of asynchronous communication
Status	The status of a message communication

Table B.1: List of P2P-MPI API classes

B.1 Comm

An point-to-point communication class. The class contains the following methods.

- ```
  Group Group()
  ```

  Returns group of this communicator.

- ```
  Request Irecv(Object recvBuffer, int offset, int count,
                Datatype datatype, int src, int tag)
  ```

recvBuffer	receive buffer
offset	initial offset in recvBuffer
count	number of elements to receive
datatype	received data type
src	rank of source
tag	message tag

Non-blocking receive operation which returns an `Request` object.

- ```
  Request Isend(Object sendBuffer, int offset, int count,
                Datatype datatype, int dst, int tag)
  ```

sendBuffer	send buffer
offset	initial offset in sendBuffer
count	number of elements to send
datatype	send data type
dst	rank of destination
tag	message tag

Non-blocking send operation which returns an `Request` object.

- ```
  int Rank()
  ```

Returns rank of process in this communicator.

- ```
  Status Recv(Object recvBuffer, int offset, int count,
              Datatype datatype, int src, int tag)
  ```

recvBuffer	receive buffer
offset	initial offset in recvBuffer
count	number of elements to receive
datatype	received data type
src	rank of source
tag	message tag

Blocking receive operation which returns an `Status` object.

- ```
  int Send(Object sendBuffer, int offset, int count,
           Datatype datatype, int dst, int tag)
  ```

sendBuffer	send buffer
offset	initial offset in sendBuffer
count	number of elements to send
datatype	send data type
dst	rank of destination
tag	message tag

Basic send operation. it returns number of sent elements in sendBuffer.

- ```
  Status Sendrecv(Object sendBuffer, int sendOffset, int sendCount,
                  Datatype sendType, int dst, int sendTag,
                  Object recvBuffer, int recvOffset, int recvCount,
                  Datatype recvType, int src, int recvTag)
  ```

sendBuffer	send buffer
sendOffset	initial offset in sendBuffer
sendCount	number of elements to send
sendType	send data type
dst	rank of destination
sendTag	send message tag
recvBuffer	receive buffer
recvOffset	initial offset in recvBuffer
recvCount	number of elements to receive
recvType	received data type
src	rank of source
recvTag	receive message tag

Send and then receive operation. It returns an `Status` object.

- ```
  int Size()
  ```

Return the number of processes in this communicator.

- ```
  int Ssend(Object sendBuffer, int offset, int count,
            Datatype datatype, int dst, int tag)
  ```

sendBuffer	send buffer
offset	initial offset in sendBuffer
count	number of elements to send
datatype	send data type
dst	rank of destination
tag	message tag

Synchronized send operation which returns the number of sent elements.
  ```

B.2 Datatype

The primitive datatypes provides in P2P-MPI.

Static types	Java types
BOOLEAN	boolean
BYTE	byte
CHAR	char
DOUBLE	double
FLOAT	float
INT	int
LONG	long
NULL	no object
OBJECT	java object
SHORT	short
STRING	string

It contains the following methods for the operation on datatypes.

- `Datatype Contiguous(int count)`

  count number of elements

  Creates a contiguous datatype and return contiguous datatype.

- `int Extent()`

  Returns the extent of a datatype.

- `int Lb()`

  Returns the lower bound of datatype.

- `int Size()`

  Returns the size of a datatype.

- `int Ub()`

  Returns the upper bound of a datatype.

B.3 Group

Contains operation on MPI groups.

- `static int Compare(Group group1, Group group2)`

  group1 first group
  group2 second group

  Compare two groups.

- `static Group Difference(Group group1, Group group2)`

  group1 first group
  group2 second group

  Creates new group from the difference between group1 and group2.

- `Group Excl(int[] rank)`

  rank list of rank to be excluded from group

  Creates a new group which excludes some ranks from original group.

- `Group Incl(int[] rank)`

  rank list of rank to be included to new group

  Creates a new group which includes some rank from original group.

- `static Group Intersection(Group group1, Group group2)`

  group1 first group
  group2 second group

  Creates a new group from the intersection between group1 and group2.

- `int Rank()`

  Returns rank in this group.

- `int Size()`

Returns size of this group.

- ```
  static Group Union(Group group1, Group group2)
  ```

  group1 first group
  group2 second group

  Creates a new group from the union between group1 and group2.

B.4 IntraComm

This class provides collective communication operations.

- ```
  void Allgather(Object sendBuffer, int sendOffset,
                 int sendCount, Datatype sendType,
                 Object recvBuffer, int recvOffset,
                 int recvCount, Datatype recvType)
  ```

  sendBuffer send buffer
  sendOffset initial offset in sendBuffer
  sendCount number of elements to send
  sendType send data type
  recvBuffer receive buffer
  recvOffset initial offset in recvBuffer
  recvCount number of elements to receive
  recvType received data type

  Gathers data from all tasks and distribute it to all.

- ```
  void Allgatherv(Object sendBuffer, int sendOffset,
                  int sendCount, Datatype sendType,
                  Object recvBuffer, int recvOffset, int[] recvCount,
                  int[] displs, Datatype recvType)
  ```

  sendBuffer send buffer
  sendOffset initial offset in sendBuffer
  sendCount number of elements to send
  sendType send data type
  recvBuffer receive buffer
  recvOffset initial offset in recvBuffer
  recvCount Array of the number of elements to receive
  displs Array of displacement in recvBuffer
  recvType received data type

  Gathers data from all tasks and distributes it to all (variable size).
  ```

- ```
  void Allreduce(Object sendBuffer, int sendOffset,
                 Object recvBuffer, int recvOffset,
                 int count, Datatype datatype, Op op)
  ```

sendBuffer	send buffer
sendOffset	initial offset in sendBuffer
recvBuffer	receive buffer
recvOffset	initial offset in recvBuffer
count	number of elements in buffer
datatype	data type
op	operation to do on sendBuffer and recvBuffer buffers

Reduces the result by *op* operation then broadcast to all processes.

- ```
  void Alltoall(Object sendBuffer, int sendOffset,
                int sendCount, Datatype sendType,
                Object recvBuffer, int recvOffset,
                int recvCount, Datatype recvType)
  ```

sendBuffer	send buffer
sendOffset	initial offset in sendBuffer
sendCount	number of elements to send
sendType	send data type
recvBuffer	receive buffer
recvOffset	initial offset in recvBuffer
recvCount	number of elements to receive
recvType	received data type

Exchanges data to all processes.

- ```
  void Alltoallv(Object sendBuffer, int sendOffset, int[] sendCount,
                 int[] sendDispl, Datatype sendType,
                 Object recvBuffer, int recvOffset, int[] recvCount,
                 int[] recvDispl, Datatype recvType)
  ```
  ```

sendBuffer	send buffer
sendOffset	initial offset in sendBuffer
sendCount	Array of the number of elements to send
sendDispls	Array of displacement in sendBuffer
sendType	send data type
recvBuffer	receive buffer
recvOffset	initial offset in recvBuffer
recvCount	Array of the number of elements to receive
recvDispls	Array of displacement in recvBuffer
recvType	received data type

Exchanges data to all processes in varied size.

- ```
  void Barrier()
  ```

Synchronized MPI processes.

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

buffer	send/receive buffer
offset	initial offset in buffer
count	number of elements to send/receive
datatype	send/receive data type
root	rank process which sends buffer

Broadcasts a message to all MPI processes.

- ```
  IntraComm Create(Group group)
  ```

group	MPI group

Creates a new intra-communicator.

- ```
  void Gather(Object sendBuffer, int sendOffset,
              int sendCount, Datatype sendType,
              Object recvBuffer, int recvOffset,
              int recvCount, Datatype recvType, int root)
  ```

sendBuffer	send buffer
sendOffset	initial offset in sendBuffer
sendCount	number of elements to send
sendType	send data type
recvBuffer	receive buffer
recvOffset	initial offset in recvBuffer
recvCount	number of elements to receive
recvType	received data type
root	rank process to gather data

Gathers together values from a group of tasks.

- ```
  void Gatherv(Object sendBuffer, int sendOffset,
               int sendCount, Datatype sendType,
               Object recvBuffer, int recvOffset,
               int[] recvCount, int[] displs,
               Datatype recvType, int root)
  ```

sendBuffer	send buffer
sendOffset	initial offset in sendBuffer
sendCount	number of elements to send
sendType	send data type
recvBuffer	receive buffer
recvOffset	initial offset in recvBuffer
recvCount	Array of the number of elements to receive
displs	Array of displacement in recvBuffer
recvType	received data type
root	rank process to gather data

Gathers together values from group of tasks (varied size).

- ```
  void Reduce_scatter(Object sendBuffer, int sendOffset,
                      Object recvBuffer, int recvOffset,
                      int[] recvCount, Datatype datatype, Op op)
  ```

sendBuffer	send buffer
sendOffset	initial offset in sendBuffer
recvBuffer	receive buffer
recvOffset	initial offset in recvBuffer
recvCount	Array of the number of elements to receive
datatype	data type
op	operation

Combines value and scatters the results.

- ```
  void Reduce(Object sendBuffer, int sendOffset,
              Object recvBuffer, int recvOffset,
              int count, Datatype datatype, Op op, int root)
  ```

sendBuffer	send buffer
sendOffset	initial offset in sendBuffer
recvBuffer	receive buffer
recvOffset	initial offset in recvBuffer
count	number of elements
datatype	data type
op	operation type
root	rank process to perform the operation on data

  Performs an operation on root process.

- ```
  void Scan(Object sendBuffer, int sendOffset, Object recvBuffer,
            int recvOffset, int count, Datatype datatype, Op op)
  ```

sendBuffer	send buffer
sendOffset	initial offset in sendBuffer
recvBuffer	receive buffer
recvOffset	initial offset in recvBuffer
count	number of elements
datatype	data type
op	operation type

  Computes the scan (partial reductions) of data on a collection of processes.

- ```
  void Scatter(Object sendBuffer, int sendOffset,
               int sendCount, Datatype sendType,
               Object recvBuffer, int recvOffset,
               int recvCount, Datatype recvType, int root)
  ```

sendBuffer	send buffer
sendOffset	initial offset in sendBuffer
sendCount	number of elements to send
sendType	send data type
recvBuffer	receive buffer
recvOffset	initial offset in recvBuffer
recvCount	number of elements to receive
recvType	received data type
root	rank process to scatter data

  Sends data from root process to all other processes in a group.
  ```

- ```
  void Scatterv(Object sendBuffer, int sendOffset,
                int[] sendCount, int[] displs, Datatype sendType,
                Object recvBuffer, int recvOffset,
                int recvCount, Datatype recvType, int root)
  ```

sendBuffer	send buffer
sendOffset	initial offset in sendBuffer
sendCount	Array of number of elements to send
displs	Array of displacement in sendBuffer
sendType	send data type
recvBuffer	receive buffer
recvOffset	initial offset in recvBuffer
recvCount	number of elements to receive
recvType	received data type
root	rank process to scatter data

  Sends a buffer from root process in parts to all processes in a group.

- ```
  IntraComm Split(int color, int key)
  ```

color	control of subset assignment
key	control of rank assignment

  Split communicator from color and key.

B.5 MPI

The main MPI class.

- Special parameters

`ANY_SOURCE`	source parameter in receive methods to indicate any sources
`ANY_TAG`	tag parameter in receive methods to indicate any tags

- Default operations

`BAND`	bit-wised AND
`MAX`	max value
`MAXLOC`	max value and its location
`MIN`	min value
`MINLOC`	min value and its location
`PROD`	production
`SUM`	summation

- MPI group comparison results

IDENT	two groups are identical
SIMILAR	two groups are similar (same members, different ranks)
UNEQUAL	two groups are not identical

- Predefined Datatypes

BOOLEAN	boolean
BYTE	byte
BYTE2	two bytes
CHAR	character
CHAR2	two characters
DOUBLE	double
DOUBLE2	two doubles
FLOAT	float
FLOAT2	two floats
INT	integer
INT2	two integers
LONG	long
LONG2	two longs
OBJECT	java object or array
SHORT	short
SHORT2	two shorts
STRING	string

- `IntraComm COMM_WORLD`

  Default communicator.

- `static void Finalize()`

  Finalization the MPI program.

- `static String Get_processor_name()`

  Returns a local hostname.

- `static String[] Init(String[] args)`

  Initialization the MPI program.

- `static int Rand()`

Returns a pseudo-random, uniformly distributed int value from this random number generator's sequence.

- `static Rand(int n)`

Returns a pseudo-random, uniformly distributed int value between 0 (inclusive) and the specified value (exclusive), drawn from this random number generator's sequence.

- `static double Wtick()`

Returns the resolution of Wtime.

- `static double Wtime()`

Returns an elapsed time on the calling processor (seconds).

B.6 MPI_User_function and Op

MPI operations (`Op` class)

- `Op(MPI_User_function function, boolean commute)`

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

The constructor of Op class.

- `boolean isCommute()`

Checks if operation is commutative.

MPI programmers can build their own MPI operation by constructing a new class.

- The abstract base class `MPI_User_function` is defined by:

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

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

B.7 Request

The handle of asynchronous communication

- `Status Test()`

  Tests if message reception has completed.

- `Status Wait()`

  Block until a waiting asynchronous message is received.

- `static Status[] Waitall(Request[] requests)`

  `requests` array of request objects
  Block until all of the operations associated with the active requests in the array have completed.

B.8 Status

The status of a message communication. It contains

- two public variables

`int source`	source rank
`int tag`	tag number

- and one method

  `int Get_count(Datatype type)`

  Returns the number of elements depends on its datatype.

Appendix C

P2P-MPI User's Guide

C.1 P2P-MPI Configuration File

Table C.1 show default configuration file, P2P-MPI.conf, which resides in environment variable `P2PMPI_HOME`. It is divided into subsets of setting as follows :

Bootstrap Setting : `SUPERNODE` is used to define a machine which is running a *supernode* process. `VISU_PROXY` is used to define a machine which is running a *visu_proxy* process. It is an optional setting.

Local Machine Setting : `MPD_PORT`, `FT_PORT`, `FD_PORT`, and `RS_PORT` are the ports that P2P-MPI processes (MPD, FT, FD, RS) will be used respectively. `MIN_PORT` and `MAX_PORT` define port range that MPI applications will use. `EXTERNAL_IP` is optional and will be used when the machine is behind the firewall or when the machine has a private IP address. If the user knows that all the machines he needs to use are in a private network, then he does not have to use `EXTERNAL_IP` option. `HOST_DENY` specifies the list of IP addresses that this machine does not allow to execute MPI applications. The IP address can be in format `XX.XX.XX.XX` for a single machine or `XX.XX.XX.` to deny all machines which IP address starts with `XX.XX.XX` (i.e. 192.168.0. refers to all the machines which have IP address from 192.168.0.0 to 192.168.0.255).

Resource Contribution Setting : `MAX_PROCESSES_PER_JOB` defines the number of MPI processes an MPI application can use on the user's machine. `MAX_JOBS` defines the number of MPI applications that can be executed simultaneously on the user's machine machine (it is set to 0, if an unlimited number of MPI applications can be executed simultaneously).

Fault Detector Setting : `T_GOSSIP` is the period between each fault detection service's gossip message in microseconds. `T_MAX_HANG` is used to prevent false fault detection from a temporary network link failure. It is the additional time over the normal

detection time. The unit is in microsecond. `GOSSIP_PROTOCOL` defines the protocol, either DBRR (double binary round-robin) or BRR (binary round-robin) protocol. Thus, the actual fault detection time in theory is $((3\log_2(n))\times$ `T_GOSSIP`$) +$ `T_MAX_HANG` for DBRR protocol and $((2\_log_2(n))\times$ `T_GOSSIP`$) +$ `T_MAX_HANG` for BRR protocol.

C.2 Command lines

P2P-MPI is distributed with a set of command lines. We divide into three categories : supernode commands, MPI commands, and visu commands.

Supernode Commands

The list of supernode commands are :

- `runSupernode` to start a supernode.

```
choopan@mordred:~$ runSupernode
```

- `stopSupernode` to stop supernode process.

```
choopan@mordred:~$ stopSupernode
```

- `supernode_stat` to check how many MPDs are known in this supernode.

```
choopan@mordred:~$ supernode_stat
Host            MPD Port        Last update
130.79.192.153  19897           0D 0:2:1:441
130.79.192.150  19897           0D 0:0:7:996

Total : 2 MPD known.
```

MPI Commands

For a machine that need to participate in P2P-MPI network to share and to use shared resources.

- `mpiboot` to start all P2P-MPI processes (MPD, FD, FT, and RS).

```
choopan@mordred:~$ mpiboot
[Booting mpd 0.28.0]
.
MPD started. Log is in /home/stagiaires/choopan/p2pmpi/tmp/mpd-mordred.log
```

- `mpihalt` to stop all P2P-MPI processes.

```
choopan@mordred:~$ mpihalt
FT Shutdown ... Done.
FD Shutdown ... Done.
RS Shutdown ... Done.
MPD Shutdown ... Done.
P2P-MPI Shutdown .. Completed
```

- `mpihost` to see the list of machines running P2P-MPI in the local hostcache.

```
choopan@mordred:~$ mpihost
Hostcache entry of MPD : 127.0.01

Host                   MPD Port RTT(ms) Alive   Last update
lancelot.u-strasbg.fr  19897            27      true    0D 0:0:42:543
pellinore.u-strasbg.fr 19897            40      true    0D 0:0:42:596

Total : 2 MPD known.
```

- `mpistat` to see information of executing MPI applications on this machine.

```
choopan@mordred:~$ mpistat
Trying to connect to 127.0.0.1:19897...
============================================================
   Gatekeeper (mpd 0.28.0) up and running.
------------------------------------------------------------
   Application Name : Dummy
   MPI Rank   : 0
   Local Port : 19816
   Rank 0 IP  : 130.79.192.153
============================================================
```

- `p2pmpirun` to run an MPI application.

```
choopan@mordred:~$ p2pmpirun
Usage : p2pmpirun -n <numproc> [-r <numreplica> -l <input filelist>
        -w <time> -a <strategy>] <command> [args]

 -a <strategy>   : name of allocation strategy (gather or scatter)
                   (default is scatter)
 -n <numproc>    : number of processes MPI
 -r <numreplica> : number of replica per rank
                   (not needed for 1 replica per rank)
 -l <filelist>   : list of input file
                   (not needed if only the executable file is
                   to be transfered)
 -w <time>       : maximum time in seconds to wait for searching nodes
 <command>       : executable file without .class
```

```
 args            : arguments of executable file

choopan@mordred:~$ p2pmpirun -n 2 Dummy
```

Visu Command

The set of commands for P2P-MPI graphical interface monitoring tools.

- `runVisu` to start P2P-MPI graphical interface monitoring tools.

```
choopan@mordred:~$ runVisu
```

- `runVisuProxy` to start a proxy server for visu program to reduce load on MPD.

```
choopan@mordred:~$ runVisuProxy
```

C.3 Sample Codes

Table C.2 shows the example of parallel Pi program using P2P-MPI. To program with P2P-MPI, programmers first need to `import` P2P-MPI package (`import p2pmpi.mpi.*;` in line 1). `MPI.Init(args)` (in line 9) needs to be called before using other MPI methods. Because `MPI.Init(args)` is used to create a default MPI communicator `COMM_WORLD`. Finally, all MPI applications must be finished with `MPI.Finalize()` (in line 41). This method negociates with MPD to tell MPD that the application is terminated. Thus, MPD can clean this application from its process table.

```
1  ##############################
2  # SuperNode
3  ##############################
4  SUPERNODE=tcp://pellinore.u-strasbg.fr:9700
5  VISU_PROXY=tcp://tag.u-strasbg.fr:9701
6
7  ##############################
8  # MPD, FT, FD fixed ports
9  ##############################
10 MPD_PORT=9897
11 FT_PORT=9898
12 FD_PORT=9899
13 RS_PORT=9900
14
15 ################################
16 # MPI application port range
17 ################################
18 MIN_PORT=9801
19 MAX_PORT=9900
20
21 ##############################################
22 # PC behind firewall (after doing port forward)
23 # uncomment here and put your external IP
24 ##############################################
25 #EXTERNAL_IP=
26
27 ####################################################################
28 # Maximum Number of simultaneous process per job >= 1
29 # [ Need to restart P2P-MPI ]
30 ####################################################################
31 MAX_PROCESSES_PER_JOB=1
32
33 ####################################################################
34 # Maximum number of jobs (applications) accepted simultaneously
35 # (0 : unlimited)
36 ####################################################################
37 MAX_JOBS=0
38
39 ####################################################################
40 # Hosts IP whose requests will be ignored
41 # coma separated list of IP or networks
42 ####################################################################
43 #HOST_DENY=130.79.192.150,213.23.45.
44
45 ####################################################################
46 # Fault detector service
47 ####################################################################
48 # Period to send gossip message (ms)
49 T_GOSSIP=500
50 # Tolerate a network failure at maximum T_max_hang (ms)
51 T_MAX_HANG=5000
52
53 # Gossip protocol (DBRR, BRR) [default: DBRR]
54 # DBRR (3 log2(n) detection time)
55 # BRR  (2 log2(n) detection time)
56 GOSSIP_PROTOCOL=DBRR
57
58 ####################################################################
59 # Cache file
60 ####################################################################
61 PEER_CACHE=/tmp/cache.xml
```

Table C.1: The default P2P-MPI configuration file.

```
1  import p2pmpi.mpi.*;
2
3  public class Pi {
4      public static void main(String[] args) {
5          int rank, size, i;
6          double PI25DT = 3.14159265358979323846264 3;
7          double h, sum, x;
8
9          MPI.Init(args);
10         double startTime = MPI.Wtime();
11
12         size = MPI.COMM_WORLD.Size();
13         rank = MPI.COMM_WORLD.Rank();
14
15         int[] n = new int[1];
16         double[] mypi = new double[1];
17         double[] pi   = new double[1];
18
19         if(rank == 0) {
20             n[0] = 1000000; // number of interval
21         }
22
23         MPI.COMM_WORLD.Bcast(n, 0, 1, MPI.INT, 0);
24
25         h = 1.0 / (double)n[0];
26         sum = 0.0;
27         for(i = rank + 1; i <= n[0]; i+= size) {
28             x = h * ((double)i - 0.5);
29             sum += (4.0/(1.0 + x*x));
30         }
31         mypi[0] = h * sum;
32
33         MPI.COMM_WORLD.Reduce(mypi, 0, pi, 0, 1, MPI.DOUBLE, MPI.SUM, 0);
34
35         if(rank == 0) {
36             System.out.println("Pi is approximately " + pi[0]);
37             System.out.println("Error is " + (pi[0] - PI25DT));
38             double stopTime = MPI.Wtime();
39             System.out.println("Time usage = " + (stopTime - startTime) + " s");
40         }
41         MPI.Finalize();
42     }
43 }
```

Table C.2: The example of Pi program.

Appendix D

Benmarks (JGF section 1)

This chapter gives the results on the JGF section 1 benchmark of three MPJ implementation : P2P-MPI version 0.27.1 (SP device), P2P-MPI version 0.28.0 (MP device), and MPJExpress.

D.1 Experiment Setup

We have used Rennes site in Grid5000, using 128 nodes.

Environment type	Grid5000, Rennes site (paravent cluster and paraquad cluster)
Number of nodes/cores	128 nodes/128 cores
Hardware	Intel Xeon 5148 LV, 4GB RAM AMD Opteron 246, 2GB RAM
Operating System	Linux 2.6.24-1-amd64
Interconnection	Gigabit Ethernet.
Java runtime	Java 1.5.0_08.
Benchmark suites	JGF section 1 : Point-to-point communication
MPJ implementation	P2P-MPI-0.27.1 (SP device), P2P-MPI-0.28.0 (MP device), and MPJExpress

D.2 Benchmark Results

In the following figures, the caption names refer to the different communication calls we tested : barrier, reduce, bcast and reduce, followed either by *d* (for double) or *o* (for object), then by the number of elements in the array. For example : Figure D.2 consists of two sub-figures reduce-d-4 and reduce-d-2048. reduce-d-4 shows the result of `MPI.Reduce` operation on an array of doubles whose size is 4. and reduce-d-2048 shows the result of `MPI.Reduce` operation on an array of doubles whose size is 2048.

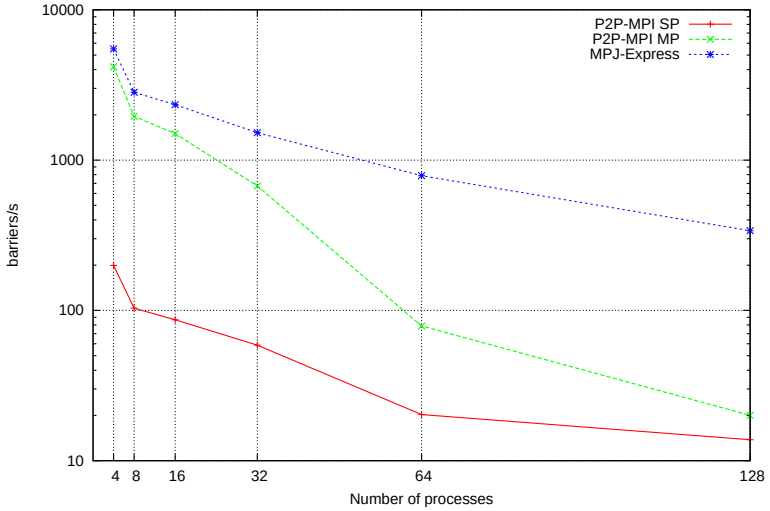
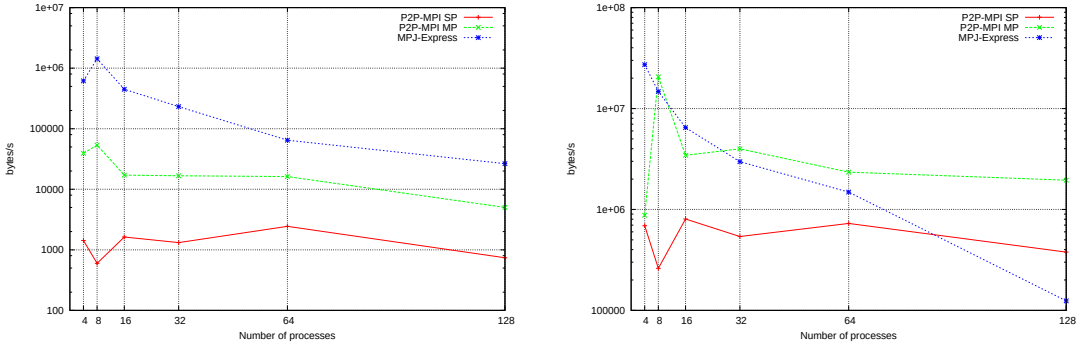
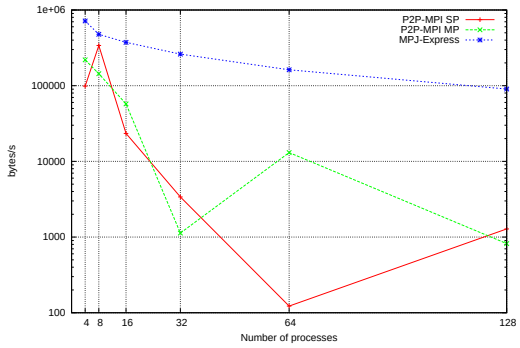


Figure D.1: Barrier test

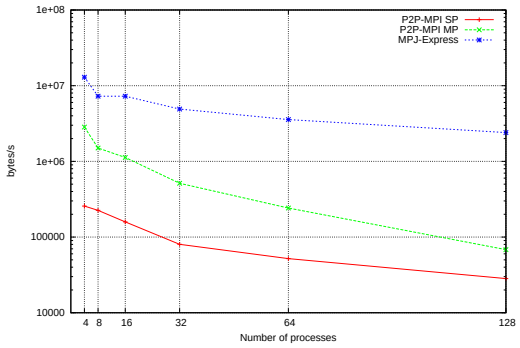


(a) reduce-d-4 (b) reduce-d-2048

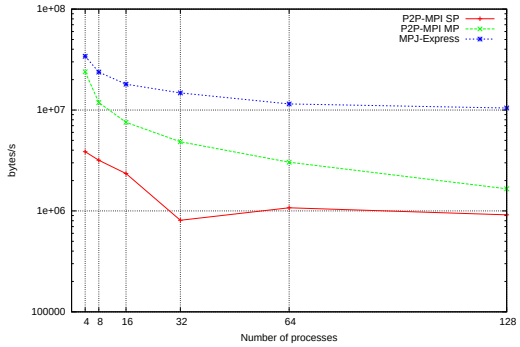
Figure D.2: Reduce test



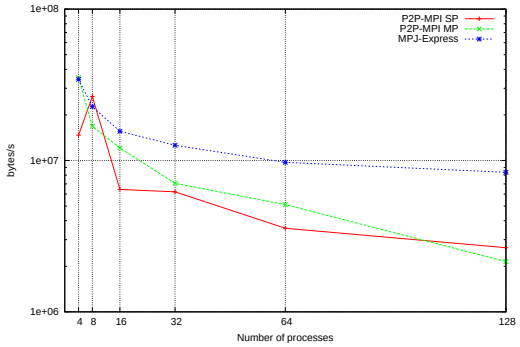
(a) bcast-d-4



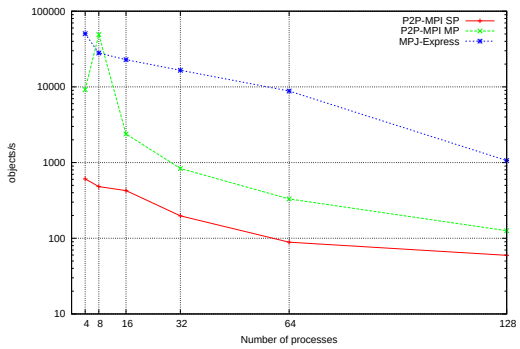
(b) bcast-d-90



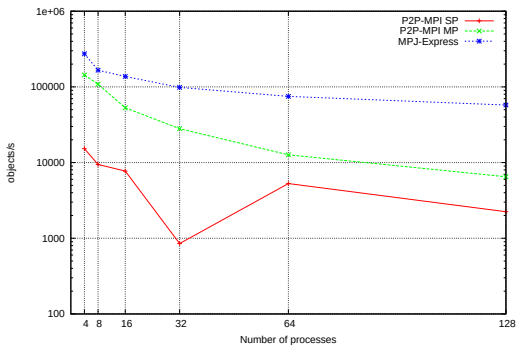
(c) bcast-d-2048



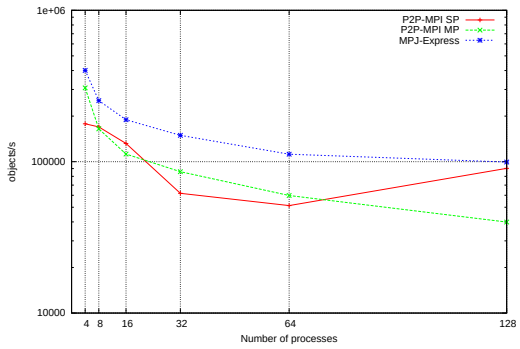
(d) bcast-d-46340



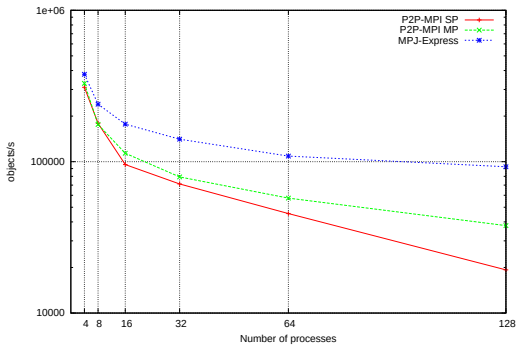
(e) bcast-o-4



(f) bcast-o-90

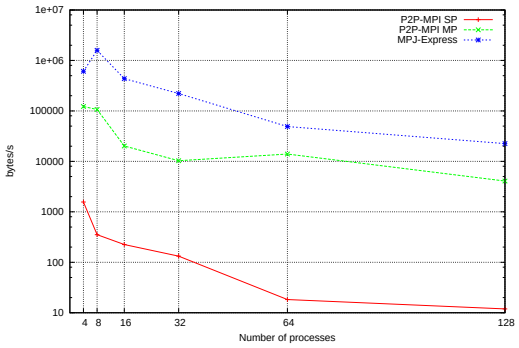


(g) bcast-o-2048

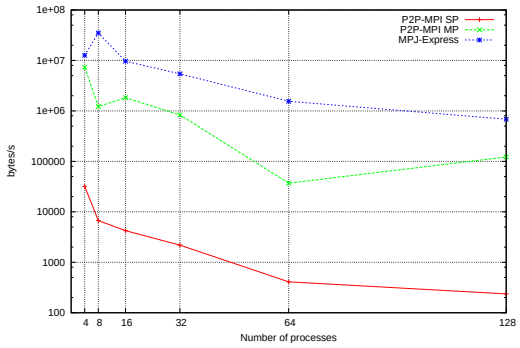


(h) bcast-o-46340

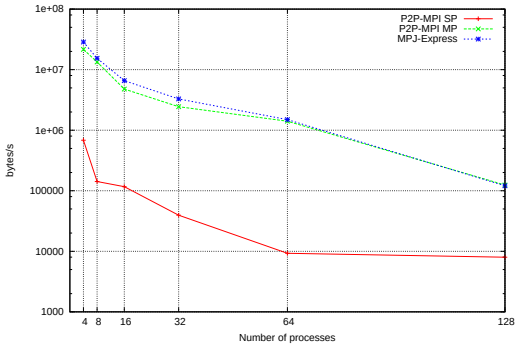
Figure D.3: Bcast test



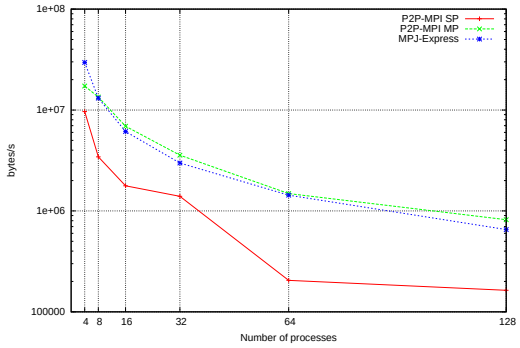
(a) gather-d-4



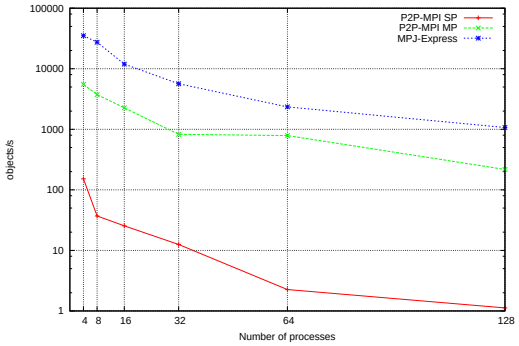
(b) gather-d-90



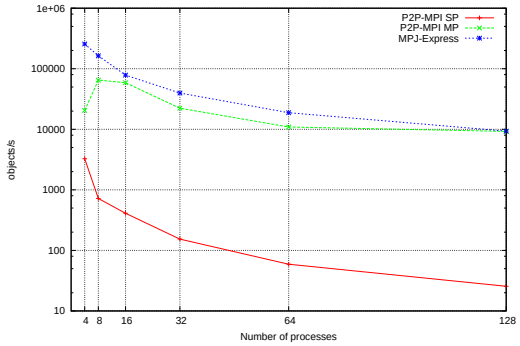
(c) gather-d-2048



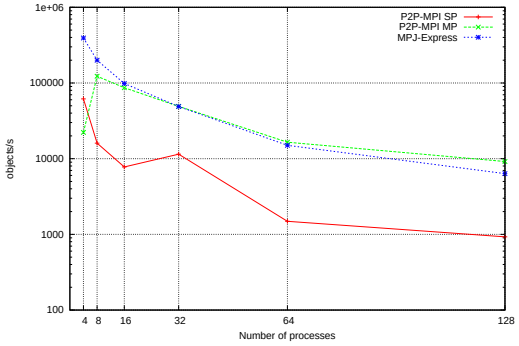
(d) gather-d-46340



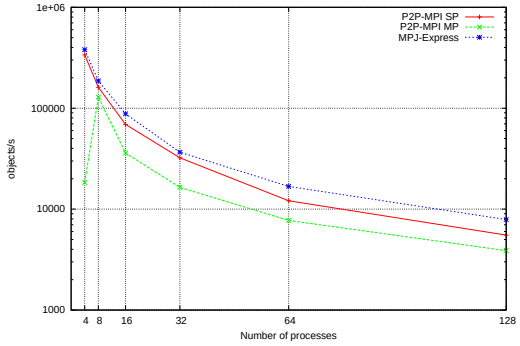
(e) gather-o-4



(f) gather-o-90



(g) gather-o-2048



(h) gather-o-46340

Figure D.4: Gather test

Bibliography

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

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

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

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

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

[6] SETI@home. `http://setiathome.berkeley.edu`.

[7] Folding@home. `http://folding.stanford.edu`.

[8] Egee (Enabling Grids for E-sciencE). `http://www.eu-egee.org/`.

[9] Satoshi Sekiguchi, Mitsuhisa Sato, Hidemoto Nakada, and Umpei Nagashima. –ninf–: Network base information library for globally high performance. In *Parallel Object-Oriented Methods and Applications (POOMA)*, February 1996.

[10] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Seymour, K. Sagi, Z. Shi, and S. Vadhiyar. Users' Guide to NetSolve V1.4.1. Innovative Computing Dept. Technical Report ICL-UT-02-05, University of Tennessee, Knoxville, TN, June 2002.

[11] Lyster P., Bergman L., Li P., Stanfill D., Crippe B., Blom R., Pardo C., and Okaya D. Casa gigabit supercomputing network: Calcrust three-dimensional real-time multi-dataset rendering. *Supercomputing'92*, 1992.

[12] Larry Smarr and Charles E. Catlett. Metacomputing. *Commun. ACM*, 35(6):44–52, 1992.

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

[14] James Frey, Todd Tannenbaum, Ian Foster, Miron Livny, and Steve Tuecke. Condor-G: A computation management agent for multi-institutional grids. *Cluster Computing*, 5:237–246, 2002.

[15] Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor - a hunter of idle workstations. In *ICDCS*, pages 104–111, 1988.

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

[17] Gilles Fedak, Cécile Germain, Vincent Néri, and Franck Cappello. Xtremweb: A generic global computing system. In *CCGRID*, pages 582–587. IEEE Computer Society, 2001.

[18] Andrew S. Grimshaw, William A. Wulf, James C. French, Alfred C. Weaver, and Paul F. Reynolds Jr. Legion: The next logical step toward a nationwide virtual computer. Technical Report CS-94-21, University of Virginia, August 1994.

[19] EU Data Grid project. `http://www.eu-datagrid.org`.

[20] Eddy Caron and Frédéric Desprez. Diet: A scalable toolbox to build network enabled servers on the grid. *International Journal of High Performance Computing Applications*, 20(3):335–352, 2006.

[21] Keith Seymour, Hidemoto Nakada, Satoshi Matsuoka, Jack Dongarra, Craig A. Lee, and Henri Casanova. Overview of gridrpc: A remote procedure call api for grid computing. In Manish Parashar, editor, *GRID*, volume 2536 of *Lecture Notes in Computer Science*, pages 274–278. Springer, 2002.

[22] A. Geist, A. Beguelin, Jack Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, Mass., 1994.

[23] MPICH. `http://www-unix.mcs.anl.gov/mpi`.

[24] MPICH2. `http://www.mcs.anl.gov/research/projects/mpich2`.

[25] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[26] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew

Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[27] Gabrielle Allen, Thomas Dramlitsch, Ian Foster, Nicholas T. Karonis, Matei Ripeanu, Edward Seidel, and Brian Toonen. Supporting efficient execution in heterogeneous distributed computing environment with cactus and globus. In *Proceedings of SuperComputing 2001*, page 52. ACM/IEEE, November 2001.

[28] Thilo Kielmann, Rutger F. H. Hofman, Henri E. Bal, Aske Plaat, and Raoul A. F. Bhoedjang. MagPIe: MPI's collective communication operations for clustered wide area systems. *ACM SIGPLAN Notices*, 34(8):131–140, August 1999.

[29] Amnon Barak, Shai Guday, and Richard Wheeler. *The MOSIX Distributed Operating System, Load Balancing for UNIX*, volume 672 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993. `http://www.mosix.cs.huji.ac.il/`.

[30] Nicholas Karonis, Brian Toonen, and Ian Foster. MPICH-G2: A grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing*, 63(5):551–563, 2003.

[31] Motohiko Matsuda, Tomohiro Kudoh, Yuetsu Kodama, Ryousei Takano, and Yutaka Ishikawa. Efficient mpi collective operations for clusters in long-and-fast networks. In *CLUSTER*, 2006.

[32] Michael Barnett, Lance Shuler, Satya Gupta, David G. Payne, Robert A. van de Geijn, and Jerrell Watts. Building a high-performance collective communication library. In *SC*, pages 107–116, 1994.

[33] Pascal Felber, Xavier Défago, Rachid Guerraoui, and Philipp Oser. Failure detectors as first class objects. In *DOA*, pages 132–141, 1999.

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

[35] Graham Fagg and Jack Dongarrar. FT-MPI: Fault tolerant mpi, supporting dynamic applications in a dynamic world. In *EuroPVM/MPI User's GroupMeeting 2000*, pages 346–353. Springer-Verlag, Berlin, Germany, 2000.

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

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

[38] Lorenzo Alvisi, E. N. Elnozahy, Sriram Rao, Syed Amir Husain, and Asanka De Mel. An analysis of communication induced checkpointing. In *29th Symposium on Fault-Tolerant Computing (FTCS'99)*, pages 242–249. IEEE CS Press, June 1999.

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

[40] Robert E. Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, 1985.

[41] Fred. B. Schneider. *Replication Management Using the State Machine Approach*, chapter 7, pages 169–195. ACM Press, 1993.

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

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

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

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

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

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

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

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

[50] Mark Baker, Bryan Carpenter, and Aamir Shafi. Mpj express: Towards thread safe java hpc. In *Proceedings of the 2006 IEEE International Conference on Cluster Computing, September 25-28, 2006, Barcelona, Spain*, 2006.

[51] Markus Bornemann, Rob V. van Nieuwpoort, and Thilo Kielmann. *MPJ/Ibis: A Flexible and Efficient Message Passing Platform for Java*, volume 3666 of *Lecture Notes in Computer Science*. Springer, 2005.

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

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

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

[55] The Freenet Project. `http://freenetproject.org`.

[56] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *the Ninth international conferance on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, November 2000.

[57] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. Technical Report TR-00-010, Berkeley, CA, 2000.

[58] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[59] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–350, 2001.

[60] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.

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

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

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

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

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

[66] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a DHT. In *ATEC'04: Proceedings of the USENIX Annual Technical Conference 2004 on USENIX Annual Technical Conference*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[67] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

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

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

[70] Kazuyuki Shudo, Yoshio Tanaka, and Satoshi Sekiguchi. P3: P2P-based middleware enabling transfer and aggregation of computational resource. In *5th Intl. Workshop on Global and Peer-to-Peer Computing, in conjunc. with CCGrid05*. IEEE, May 2005.

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

[72] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operation in mpich. *International Journal of High Performance Computing Applications*, 19(1):49–66, February 2005.

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

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

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

[76] Franck Cappello, Eddy Caron, Michel J. Daydé, Frédéric Desprez, Yvon Jégou, Pascale Vicat-Blanc Primet, Emmanuel Jeannot, Stéphane Lanteri, Julien Leduc, Nouredine Melab, Guillaume Mornet, Raymond Namyst, Benjamin Quétier, and Olivier Richard. Grid'5000: a large scale and highly reconfigurable grid experimental testbed. In *GRID*, pages 99–106. IEEE, 2005.

[77] Kadeploy2. `http://www-id.imag.fr/Logiciels/kadeploy/`.

[78] Renater: Le réseau national de télécommunications pour la technologies, l'enseignement et la recherche. `http://www.renater.fr/`.

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